Diplomarbeit

Effizienzanalyse methodenbasierter Hardware/Software Kommunikation aus Synthesesicht

Henning Kleen

14. Mai 2007

Carl von Ossietzky Universität Oldenburg
Fakultät II - Department für Informatik
Abteilung Eingebettete Hardware-/Software-Systeme

Erstgutachter:	Prof. Dr.-Ing. Wolfgang Nebel
Zweitgutachter:	Dr. rer. nat. Frank Oppenheimer
Betreuende Mitarbeiterin:	Dipl.-Ing. Cornelia Grabbe

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Oldenburg, den 14. Mai 2007

Henning Kleen

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Arbeit beigetragen haben.

Im Einzelnen:

- Der Abteilung „Eingebettete Hardware-/Software-Systeme“ für die interessante Aufgabenstellung sowie die ausgezeichnete Betreuung.
- Meinen Eltern, die mir mein Studium ermöglicht haben.
- Insa, Marie, Daniela und Ulrich für das Korrekturlesen meiner Arbeit.
- Sowie allen anderen, die mich während der Entstehung dieser Arbeit begleitet und unterstützt haben.

Zusammenfassung

Die steigende Komplexität eingebetteter Systeme erfordert zunehmend die Verlagerung der Designaktivitäten auf höhere Abstraktionsebenen sowie die Entwicklung neuer Modellierungs- und Synthesetechnologien. Einen Ansatz zum Design auf Systemebene unter Ausnutzung objektorientierter Methoden bietet das Oldenburg System Synthesis Subset (OSSS).

In dieser Diplomarbeit wird die Effizienz des von OSSS bereitgestellten methodenbasierten Interfaces zur Hardware / Software-Kommunikation eingehend untersucht. Diese Untersuchung erfolgt dabei nicht auf Basis theoretischer Betrachtungen oder Simulationen, sondern anhand der tatsächlichen Implementierung eines Designbeispiels.

Die durchgeführten Untersuchungen ergaben, dass die Designeffizienz durch die Verwendung der OSSS-Methodik gesteigert wird. Jedoch ist der durch die Verwendung der methodenbasierten Kommunikation verursachte Overhead so groß, dass eine praktische Verwendung des Verfahrens in der derzeitigen Form nicht sinnvoll erscheint. Untersuchungen mit einem optimierten Verfahren zeigten jedoch, dass dieses Problem nicht durch die generelle Verwendung des Kommunikationsprotokolls verursacht wird, sondern durch das zur Serialisierung der zu übertragenden Daten verwendete Verfahren. Durch Optimierungen der OSSS-Bibliothek ist es möglich, den verursachten Overhead derart zu reduzieren, dass eine Verwendung der OSSS-Methodik aufgrund der erzielten Verbesserung der Designeffizienz sinnvoll erscheint.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listing-Verzeichnis

1 Einleitung

1.1 Motivation

Der stetige Fortschritt bei der Verkleinerung der Strukturgröße und die damit verbundene Zunahme auf einem Chip integrierbarer Funktionen führen dazu, dass die Komplexität von integrierten Schaltungen ständig wächst. Die Produktivität des Designvorganges konnte in der Vergangenheit jedoch nicht im selben Maße gesteigert werden. Dies führte zu einer wachsenden Produktivitätslücke und zeigt somit den Bedarf für neue, die Produktivität steigernde Designkonzepte. Besonders beim Entwurf eingebetteter Systeme ergeben sich daraus neue Herausforderungen, da sie meist heterogen sind und somit der Entwurf komplexer Hard- und Software-Komponenten notwendig ist. Ein Entwicklungstrend von zunehmender Bedeutung ist das so genannte *System Level Design*, also die Verlagerung des Designvorgangs auf eine höhere Abstraktionsebene, um so auch komplexe Systeme für den Designer handhabbar zu machen. Die Aufgabe des Designers soll dabei insofern erleichtert werden, als die Verfeinerung des Entwurfs aus der Systembeschreibung weitgehend automatisiert erfolgen sollte. Leider unterstützen viele heute am Markt vertretene Lösungen diese Automatisierung nicht, oder nur sehr eingeschränkt.

Einen Ansatz, den Herausforderungen des Designs eingebetteter Systeme gerecht zu werden, bietet das auf SystemC basierende OSSS[1]. Diese im Rahmen des ICODES[2]-Projekts[ICO, ODE] entwickelte Erweiterung von SystemC soll dessen Möglichkeiten zum Entwurf auf Systemebene ergänzen und einen ganzheitlichen Entwurf des Systems ermöglichen. Ein wesentlicher Ansatz von OSSS ist, dem Designer den fehlerträchtigen Entwurf von Kommunikationsschnittstellen zwischen Hard- und Software abzunehmen und durch ein automatisch generiertes Kommunikationsinterface zu ersetzen.

1.2 Aufgabenstellung

Im Rahmen dieser Arbeit soll die Effizienz dieser automatisch generierten Kommunikation im Vergleich zu einem herkömmlich entwickelten Kommunikationsinterface

[1] Oldenburg System Synthesis Subset
[2] Interface and Communication based Design of Embedded Systems

bestimmt werden und somit eine Aussage zur Eignung dieses Ansatzes getroffen werden. Die dieser Arbeit zugrunde liegende Fragestellung lautet also:

„Wie groß ist der durch die Verwendung der methodenbasierten Kommunikation verursachte Overhead und steigert die OSSS-Methodik die Effizienz des Designvorganges in ausreichendem Maße, um diesen Overhead zu rechtfertigen?“

Die Effizienzanalyse soll hierbei nicht auf theoretischer Basis, sondern anhand eines praktischen Designbeispiels erfolgen. Hierzu soll das HW/SW Co-Design eines MPEG-Audio-Dekoders dienen.

1.3 Struktur der Arbeit

In diesem Abschnitt wird kurz auf die Struktur der folgenden Kapitel eingegangen.

In Kapitel 2 wird zunächst der Hintergrund der Arbeit dargelegt. Hierbei wird besonders im Hinblick auf SystemC und OSSS kurz auf die Grundlagen des HW/SW Co-Designs eingegangen. Im Anschluss daran wird das MPEG-Verfahren zur Kompression von Audiodaten näher beschrieben.

In Kapitel 3 wird der bei der Entwicklung des Testsystems durchgeführte Entwurfsprozess und die dabei getroffenen Entwurfsentscheidungen detailliert beschrieben.

Kapitel 4 befasst sich mit der Implementierung der verschiedenen Entwürfe und erläutert die Details der Implementierung näher.

In Kapitel 5 erfolgt eine Auswertung der durch die verschiedenen Tests gewonnen Daten sowie eine Diskussion der Ergebnisse.

Kapitel 6 schließlich gibt eine kurze Zusammenfassung der Auswertung sowie der im Rahmen dieser Arbeit gewonnenen Erkenntnisse.

2 Hintergrund

In diesem Kapitel wird ein Überblick über die im Rahmen dieser Arbeit verwendeten Techniken gegeben. Zunächst soll auf SystemC als moderne Beschreibungssprache für den Entwurf eingebetteter Systeme eingegangen werden. Hierauf folgend soll OSSS als ein darauf aufbauender Ansatz zur High-Level Modellierung und Synthese vorgestellt werden. Abschließend wird das MPEG[1]-Audio-Verfahren zur Kompression von Audiodaten vorgestellt.

2.1 SystemC

Bei SystemC[Ope, GLMS02] handelt es sich nicht um eine eigenständige Sprache, sondern um eine C++-Klassenbibliothek, die den Sprachumfang von C++ [Str00] um Konstrukte zur Modellierung von Hardware erweitert. SystemC bietet einen Ansatz zur Lösung der beim Entwurf eingebetteter Systeme auftretenden Probleme. Diese und die von SystemC gebotenen Lösungsansätze sollen im Folgenden kurz beschrieben werden.

2.1.1 HW/SW Co-Design

Eingebettete Systeme bestehen häufig sowohl aus programmierbaren Komponenten (Prozessoren, Mikrocontrollern), als auch aus Komponenten mit fester Funktionalität (zum Beispiel ASICs[2]). Daher ist es nötig, im Verlauf des Entwurfs sowohl Hardware- als auch Softwarekomponenten zu entwickeln. HW/SW Co-Design bezeichnet eine Methodik zum Entwurf solcher heterogenen Hardware-/Softwaresysteme[Gaj96].

Abbildung 2.1 zeigt eine grafische Darstellung des Designflows beim HW/SW Co-Design. Am Beginn des Entwurfs steht eine Spezifikation des zu erstellenden Systems. Diese Spezifikation wird häufig als ausführbares Softwaremodell des Gesamtsystems in einer Programmiersprache, wie zum Beispiel C++, verfasst.

[1] Motion Picture Experts Group
[2] Application Specific Integrated Circuit

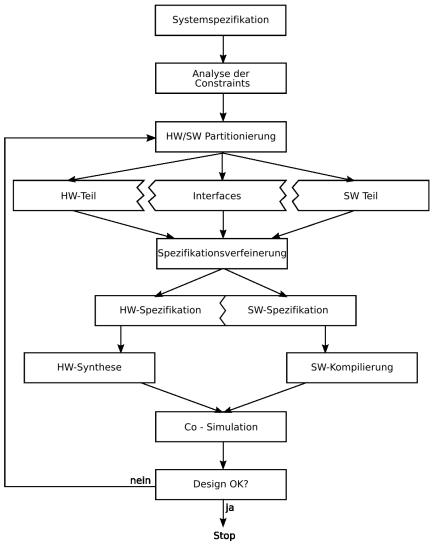

Abbildung 2.1: Generischer HW/SW Co-Designflow[Nie98]

Basierend auf einer Kostenabschätzung, in die verschiedene Faktoren wie beispielsweise die benötigte Leistungsfähigkeit des Systems einfließen, wird eine Partitionierung, das heißt eine Aufteilung in SW- und HW-Komponenten, durchgeführt.

Die weitere Entwicklung der durch die Partitionierung gewonnenen HW- beziehungsweise SW-Komponenten erfolgt parallel. Die Hardwarekomponenten werden mit Hilfe einer Hardwarebeschreibungssprache, zum Beispiel VHDL[3] oder Verilog, modelliert, Softwarekomponenten in einer Programmiersprache, wie beispielsweise C oder C++, entwickelt. Daraus ergibt sich die Notwendigkeit, die Hardwarekomponenten des Systems beim Übergang von der ausführbaren Spezifikation zum synthetisierbaren Hardwaremodell neu zu implementieren. Da dieser Vorgang nicht trivial ist, ergibt sich somit eine zusätzliche Fehlerquelle während des Entwurfs.

Am Ende der Entwicklung erfolgt eine Co-Simulation von Hard- und Software. Diese Simulation dient einerseits der Überprüfung der Funktionalität und andererseits der Überprüfung der nicht-funktionalen Anforderungen, zum Beispiel der Leistungsfähigkeit. Ergibt die Co-Simulation, dass die an das System gestellten Anforderungen nicht erfüllt werden, so erfolgt eine Neupartitionierung und der Designflow wird erneut durchlaufen. Dieser Vorgang wird gegebenenfalls wiederholt, bis das System die gestellten Anforderungen erfüllt [MWE01, Gaj96].

[3] Very High Speed Integrated Circuit Hardware Description Language

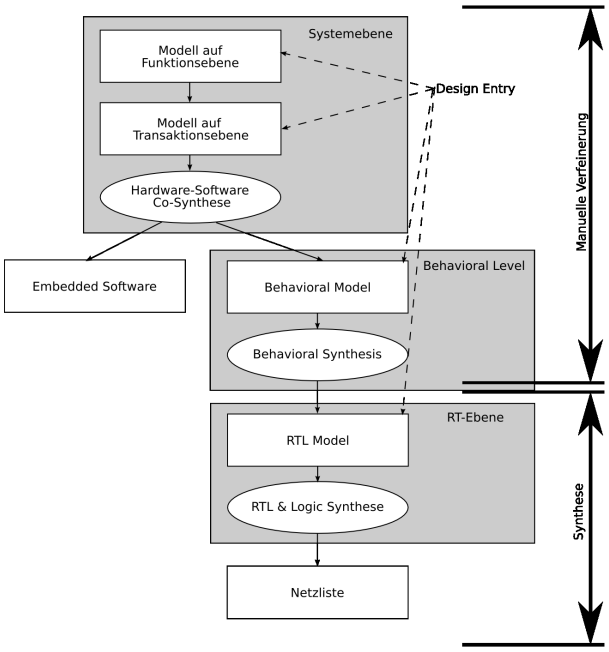

Abbildung 2.2: Der SystemC Designflow[Pan01]

2.1.2 Konzepte von SystemC

Die SystemC Klassenbibliothek enthält neben den für die Modellierung von Hardware benötigten Konstrukten einen Simulationskernel, der eine Ausführung des SystemC-Modells ohne einen zusätzlichen Simulator ermöglicht.

Die Konstrukte zur Modellierung von Hardware beinhalten Konzepte zur Beschreibung von hierarchischen Strukturen, parallelen Prozessen sowie Signalen zur Kommunikation zwischen Prozessen.

Verglichen mit klassischen Hardwarebeschreibungssprachen, wie zum Beispiel VHDL oder Verilog, ermöglicht die SystemC-Methodik ein Design auf höherer Abstraktionsebene. Die SystemC-Methodik sieht vor, die initiale Spezifikation in SystemC zu erstellen. Dieses Vorgehen spart den fehlerträchtigen Übersetzungsschritt von der Spezifikation zur ersten HW-Beschreibung, wie er beim klassischen HW/SW Co-Design nötig ist. Weiterhin ist es dadurch möglich, die Spezifikation als Testbench während des weiteren Designvorgangs zu verwenden.

Ausgehend von der Spezifikation sieht der in Abbildung 2.2 dargestellte SystemC-Designflow vor, das Design in mehreren Schritten manuell bis zu einem Modell auf Register Transfer-Ebene zu verfeinern. Dieses kann dann durch Synthesewerkzeuge in eine Beschreibung auf Hardwareebene umgewandelt werden.

Dem Entwickler steht für die Spezifikation eines Systems in SystemC der gesamte Sprachumfang von C++ und SystemC zur Verfügung. Für die Synthese kann jedoch, so wie auch in anderen Hardwarebeschreibungssprachen, nur eine Untermenge der gesamten Sprache verwendet werden, das so genannte *Synthesisable Subset*. Dies folgt daraus, dass einige SW-Konstrukte, wie zum Beispiel Zeiger, nicht in Hardware übersetzt werden können. Ähnliches gilt für Bibliotheksfunktionen wie `printf` oder die dynamische Erzeugung von Objekten.

2.1.3 Transaction Level Modeling

Eine Herausforderung, mit der Entwickler eingebetteter Systeme konfrontiert werden, ist die stetig wachsende Komplexität der Systeme, die durch die wachsende Anzahl der auf einem Chip zu integrierenden Funktionen bedingt wird. Problematisch ist dies besonders dadurch, dass die Steigerung der Produktivität der Entwickler nicht mit dem technischen Fortschritt bei der notwendigen Integration zusätzlicher Funktionen Schritt halten kann. Somit ergibt sich eine wachsende Lücke zwischen der Komplexität technisch realisierbarer Systeme und der Komplexität der Systeme, die von einem Team von angemessener Größe innerhalb angemessener Zeit entwickelt werden können[GO03].

Ein Ansatz, diese Lücke zu schließen, oder zumindest zu verkleinern, besteht darin, die Entwurfsaktivitäten auf immer höhere Abstraktionsebenen zu verlagern[GZD97, JRV+97]. Während es für Entwürfe geringer Komplexität möglich war, den Entwurf auf niedriger Abstraktionsebene (*Register Transfer Layer* - RTL oder noch darunter) durchzuführen, so erfordert die Integration des gesamten Systems auf einen Chip (SoC[4]) einen Entwurf auf Systemebene[Pan01]. Ausgehend von einem Entwurf auf Systemebene ist das Ziel, weitere Verfeinerungen des Modells zu automatisieren. Dadurch soll das Modell mit minimalem Aufwand seitens des Entwicklers auf niedrigere Abstraktionsebenen übertragen werden. Letztendlich soll eine Steigerung der Produktivität erzielt werden und dadurch eine Verkürzung der *Time-to-Market*.

Um diesen Anforderungen gerecht zu werden, bietet SystemC das so genannte *Transaction Level Modeling*[CG03] für den Entwurf auf Systemebene. SystemC beinhaltet, neben Mechanismen zur Kommunikation auf niedriger Abstraktionsebene (Signale), auch Channels zur Modellierung der Kommunikation auf höherer Ebene. Diese stellen ein methodenbasiertes Interface für die Kommunikation der Prozesse untereinander zur Verfügung. In SystemC existieren zwei verschiedene Varianten von Channels, primitive und hierarchische Channels. Der Hauptunterschied zwischen diesen beiden Varianten ist, dass primitive Channels keinerlei interne Struktur besitzen, sondern lediglich ein abstraktes Kommunikationsinterface implementieren. Sie stellen somit Signale dar. Hierarchische Channels hingegen können eine aus Prozessen bestehende interne Struktur besitzen, implementieren jedoch ebenfalls ein Kommunikationsinterface. Der Vor-

[4] System on Chip

teil hierarchischer Channels ist, auch komplexere Funktionalitäten, die über den reinen Datentransport hinausgehen, mit einem methodenbasierten Kommunikationsinterface modellieren zu können.

2.1.4 Schwächen des SystemC Ansatzes

Obwohl SystemC als C++-Klassenbibliothek auf einer objektorientierten Programmiersprache basiert, werden die Möglichkeiten des objektorientierten Ansatzes (wie beispielsweise Polymorphie) allenfalls für die Modellierung und Simulation, nicht jedoch für die Synthese, ausgeschöpft.

Ein weiteres Defizit von SystemC ist das Fehlen einer Synthese-Semantik für hierarchische Channels. Somit ist der Entwickler darauf angewiesen, die channelbasierte Kommunikation manuell bis auf Signalebene zu verfeinern. Generell ermöglicht SystemC zwar die Verwendung einer einzigen Sprache von der Spezifikation bis zur Implementierung, jedoch wird die Verfeinerung des Entwurfs nicht automatisiert, sondern muss vom Entwickler manuell durchgeführt werden.

Obwohl eine Unterstützung für die Modellierung von Software für zukünftige Versionen von SystemC geplant ist, bietet SystemC in der zum Zeitpunkt dieser Arbeit aktuellen Version keinerlei Unterstützung für die Modellierung von Software-Tasks innerhalb eines Entwurfs. Es ist zwar möglich, Software-Tasks analog zu Hardwarekomponenten zu beschreiben, jedoch bleiben die speziellen Eigenschaften der Software, wie zum Beispiel das Zeitverhalten, unberücksichtigt. Des Weiteren bietet SystemC auch keine Unterstützung für eine automatische Softwaresynthese.

2.2 OSSS

Wie im vorigen Abschnitt festgestellt, besitzt SystemC einige Schwächen, OSSS versucht diese zu überwinden. Es erweitert das synthetisierbare Subset von SystemC, unter anderem um Ansätze zur objektorientierten Beschreibung von Hardware, zur Beschreibung von Softwarekomponenten sowie zur methodenbasierten Kommunikation. In dem folgenden Abschnitt soll auf die Möglichkeiten, die OSSS für den Entwurf eingebetteter Systeme bietet, eingegangen werden.

2.2.1 Die Designebenen von OSSS

OSSS erlaubt dem Entwickler bei dem Entwurf eines Systems die Anwendung und die Architektur unabhängig voneinander zu beschreiben.

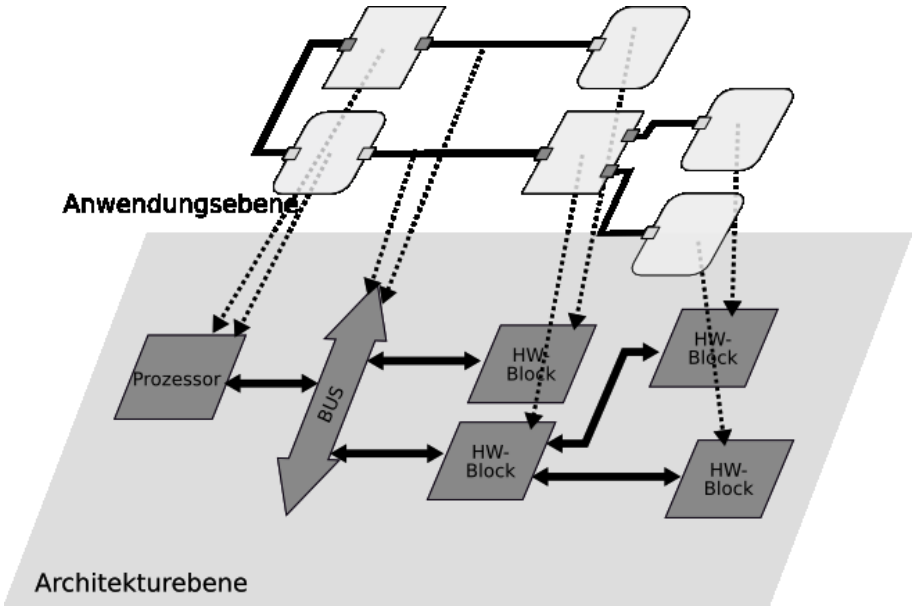

Abbildung 2.3: Anwendungs- und Architekturebene in OSSS[GGS06]

2.2.1.1 Anwendungsebene

Die Erstellung eines Entwurfs auf Anwendungsebene erfolgt direkt nach der Partitionierung des Systems. Das heißt für die Komponenten eines Systems wurde in einem vorherigen Design-Schritt entschieden, ob sie als Hardware oder als Software implementiert werden sollen. Für den Entwurf auf dieser Ebene wird jedoch von der Beschaffenheit der tatsächlichen Zielhardware sowie der Kommunikation zwischen den Prozessen abstrahiert. Das Modell besteht aus einzelnen Prozessen, die über Methodenaufrufe miteinander kommunizieren. Software-Tasks benötigen keinen Prozessor, auf dem sie ausgeführt werden.

2.2.1.2 Architekturebene

Bei der Verfeinerung des Entwurfs von der Anwendungs- zur Architekturebene erfolgt eine Abbildung des Anwendungsebenen-Modells auf eine virtuelle Zielplattform. Dabei wird die abstrakte methodenbasierte Kommunikation auf die tatsächliche Kommunikationsinfrastruktur abgebildet sowie eine Zuordnung der Software-Tasks zu Prozessoren vorgenommen. Der Designer wählt eine konkrete Implementierung der Kommunikationskanäle. Hierzu stellt die OSSS-Bibliothek einige Implementierungen (sowohl Bus-Verbindungen, als auch Punkt-zu-Punkt-Verbindungen) bereit, es ist jedoch auch möglich, eigene OSSS-Channels zu implementieren. Die Kommunikation erfolgt aus Sicht der Software-Tasks und Hardware-Module weiterhin über Methodenaufrufe. Diese wer-

den jedoch transparent über die in der Bibliothek implementierten Channels ausgeführt. Das dazu benötigte RMI[5]-Protokoll wird dabei automatisch durch das Synthesewerkzeug generiert. Dies gestattet eine einfache Exploration verschiedener Kommunikationsstrukturen. So können verschiedene Zuordnungen der methodenbasierten Kommunikation zu tatsächlichen Kommunikationsstrukturen evaluiert werden. Es wäre sogar denkbar, basierend auf gegebenen Testdaten, eine automatisierte Bewertung verschiedener solcher Zuordnungen durchzuführen. Ziel ist es dabei, einerseits dem Entwickler die fehlerträchtige Verfeinerung der Kommunikationsmechanismen abzunehmen und andererseits eine automatisierte Synthese zu ermöglichen.

2.2.2 Objektorientierte Hardwarebeschreibung

Es existieren verschiedene Ansätze[KOSK+01, SKWS+04, GZD+00], objektorientierte Methoden für die Beschreibung von Hardware zu verwenden, die sich jedoch zumeist auf die Spezifikation oder die Simulation beschränken [KRK98]. Andere Ansätze wie SystemC basieren auf objektorientierten Sprachen (zum Beispiel C++), ohne jedoch die Möglichkeiten der Objektorientierung für die Beschreibung der Hardware auszunutzen. OSSS erweitert SystemC derart, dass es möglich ist, objektorientierte Konzepte, wie zum Beispiel Polymorphie, auch für die Synthese nutzbar zu machen. Im Gegensatz zu C++ wird die Polymorphie nicht über Zeiger erreicht, sondern mit Hilfe so genannter *Tagged Objects*[GNOS03]. Dies ist nötig, um eine Synthese-Semantik für polymorphe Objekte zu erhalten.

2.2.3 Modellierung von Software

Neben der objektorientierten Beschreibung von Hardware bietet OSSS die Möglichkeit, auch Software, die Teil eines Systems ist, zu modellieren. Hierzu bietet OSSS die Klasse `osss_software_task` als Basisklasse für Software-Tasks. Ein System kann mehrere solcher Tasks enthalten, jedoch kann in der zum Zeitpunkt dieser Arbeit aktuellen Version (2.0) nur jeweils ein Task auf einen Prozessor abgebildet werden, da weder ein Betriebssystem noch ein sonstiger Mechanismus zum Scheduling verschiedener Software-Tasks eingesetzt wird. Um das Zeitverhalten der auf den Prozessoren ausgeführten Software modellieren zu können, erlaubt OSSS die geschätzten Ausführungszeiten (EET [6]) und die maximal erlaubten Ausführungszeiten (RET [7]) zu annotieren, um den Entwurf auf die Einhaltung von Zeitschranken überprüfen zu können.

[5] Remote Method Invocation
[6] Estimated Execution Time
[7] Required Execution Time

2.2.4 Methodenbasierte Kommunikation

Ein weiteres Ziel von OSSS ist es, beim Entwurf des eingebetteten Systems auf Systemebene von den verwendeten Kommunikationsmechanismen zu abstrahieren. Daher erfolgt die Kommunikation auf Systemebene nicht mittels Signalen, sondern über Methodenaufrufe. SystemC bietet mit dem Channel-Konzept ebenfalls einen Ansatz für methodenbasierte Kommunikation, jedoch fehlt den SystemC-Channels eine Synthese-Semantik, so dass der Designer die Channels während des Designvorgangs manuell verfeinern muss. OSSS erweitert das SystemC-Channel-Konzept um OSSS-Channels und so genannter Shared Objects.

Shared Objects zeichnen sich durch eine Reihe von Eigenschaften aus.
Sie:

- sind passiv.
- modellieren geteilte Ressourcen oder Interfaces zur Hardware.
- dienen der Kommunikation und Synchronisation von Prozessen.
- beinhalten Funktionalitäten zum wechselseitigen Ausschluss.

Die Kommunikation der Prozesse mit den Shared Objects folgt dem Client-Server-Paradigma. Das Shared Object übernimmt dabei die Rolle des Servers und kann Anfragen mehrerer Prozesse, den so genannten Klienten, bearbeiten. Jede Kommunikation mit dem Shared Object wird von den Klienten initiiert. Die Anfragen der Klienten werden vom Shared Object angenommen und ausgeführt. Da mehrere Klienten konkurrierend auf ein Shared Object zugreifen können, beinhalten Shared Objects Funktionen zum wechselseitigen Ausschluss. Sollte es zu konkurrierenden Zugriffen kommen, so entscheidet ein Scheduler, welchem Klienten als nächstes der Zugriff auf das Shared Object gewährt wird. Weiterhin sind die Methoden, die das Shared Object anbietet, durch so genannte *Guard Conditions* geschützt. Die Ausführung einer Methode wird nur gestattet, falls ihre *Guard Condition* „wahr„ ist. Sollte diese nicht erfüllt sein, so blockiert der Aufruf, bis die Bedingung wahr wird. Ein Beispiel für eine solche *Guard Condition*, dass ein Lesezugriff auf eine FIFO[8] nur erfolgen darf, wenn diese nicht leer ist. Dadurch wird sichergestellt, dass sich das Shared Object zu jedem Zeitpunkt in einem konsistenten Zustand befindet.

Abbildung 2.4 zeigt ein einfaches Beispiel für die Verwendung eines Shared Objects als geteilte Ressource zwischen zwei Prozessen. Das Shared Object modelliert einen Speicher, in diesem Fall als FIFO organisiert, über den Producer und Consumer miteinander kommunizieren. Das Shared Object bietet zwei Methoden an, `put` und `get`, die jeweils blockierend sind. Das heißt sollte der Producer die `put`-Methode aufrufen, aber kein

[8] First In - First Out

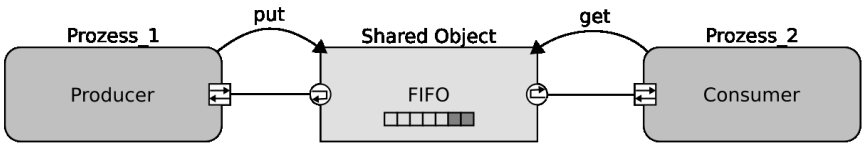

Abbildung 2.4: Producer - Consumer Beispiel

freier Platz innerhalb der FIFO vorhanden sein, so blockiert der Aufruf, bis ein Element aus der FIFO entfernt wurde und somit wieder Platz für ein weiteres Element vorhanden ist. Ruft der Consumer die `get`-Methode auf und es sind keine Elemente in der FIFO vorhanden, so wird der Consumer ebenfalls blockiert, bis der Producer ein Element in die Warteschlange eingefügt hat und somit ein Element gelesen werden kann.

Ein Anwendungsbeispiel für die Modellierung eines Interfaces zur Hardware ist zum Beispiel die Modellierung einer seriellen Schnittstelle als Shared Object. Dieses würde seinen Klienten die beiden Methoden `send` und `receive` zur Verfügung stellen.

2.2.5 Remote Method Invocation

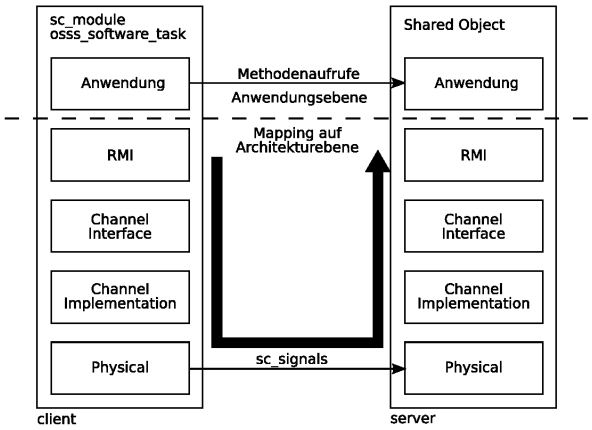

Abbildung 2.5: RMI-Schichtenarchitektur

Um die methodenbasierte Kommunikation der Klienten mit dem Shared Object von der Anwendungs- auf die Architekturebene zu übertragen, ist es nötig, eine Abbildung der methodenbasierten auf eine signalbasierte Kommunikation zu finden. Diese Aufgabe übernimmt in OSSS das so genannte RMI-Protokoll[GGS06].

Abbildung 2.5 zeigt die schichtenbasierte Kommunikation der Klienten mit dem Shared Object. Die unteren Schichten übertragen die Daten transparent für die obe-

ren Schichten, insofern ähnelt die eingesetzte Schichten-Architektur der des ISO[9]-OSI[10]-Referenzmodells. Die Umsetzung der methodenbasierten Kommunikation auf eine signalbasierte Kommunikation erfolgt mit Hilfe mehrerer Transaktoren. Für die Kommunikation auf der Anwendungsebene ist es durch diesen Ansatz unerheblich, wie die Kommunikation auf den unteren Ebenen gestaltet ist. Die Kommunikation kann auf physikalischer Ebene sowohl durch Punkt-zu-Punkt-Verbindungen als auch über Datenbusse erfolgen.

Für den Aufruf einer Methode auf dem Shared Object ist es nötig, sowohl die Parameter, als auch die Rückgabewerte zu übertragen. Da der Kommunikationskanal zwischen Klienten und Shared Object jedoch nur eine begrenzte Breite hat, müssen die zu übertragenden Daten auf der Senderseite serialisiert und auf der Empfängerseite wieder deserialisiert werden. Ein Shared Object kann mehrere Funktionen bereitstellen und auch mehrere Klienten haben, daher sind noch zusätzliche Informationen über die aufgerufene Methode und den anfragenden Klienten notwendig. Die benötigten ID[11]s und Adressen werden bei der Synthese durch das Synthesetool erzeugt und müssen nicht vom Entwickler vergeben werden.

2.2.6 Synthese eines Shared Objects

Shared Objects werden nach einem festen Schema synthetisiert[GGS06]. Abbildung 2.6 zeigt das Ergebnis der Synthese eines Shared Objects. Auf die Funktion der verschiedenen Blöcke soll in dem folgenden Abschnitt eingegangen werden.

2.2.6.1 Interface-Blöcke

Die Interface-Blöcke stellen die Verbindung zu den Klienten des Shared Objects über Datenbusse oder Punkt-zu-Punkt-Verbindungen her. Sie steuern in Verbindung mit dem Controller auch den Ablauf des RMI-Protokolls und generieren die Steuersignale für das *Argument-RAM*, in dem die Parameter der aufgerufenen Methode sowie deren Rückgabewerte abgelegt werden.

2.2.6.2 Guard Evaluator

Die von einem Shared Object bereitgestellten Funktionen können mit so genannten *Guard Conditions* versehen werden. Diese bestimmen, basierend auf dem inneren Zustand des Shared Objects, ob eine Funktion ausführbar ist. Eine Funktion ist nur

[9] International Organization for Standardization
[10] Open Systems Interconnection
[11] Identifier

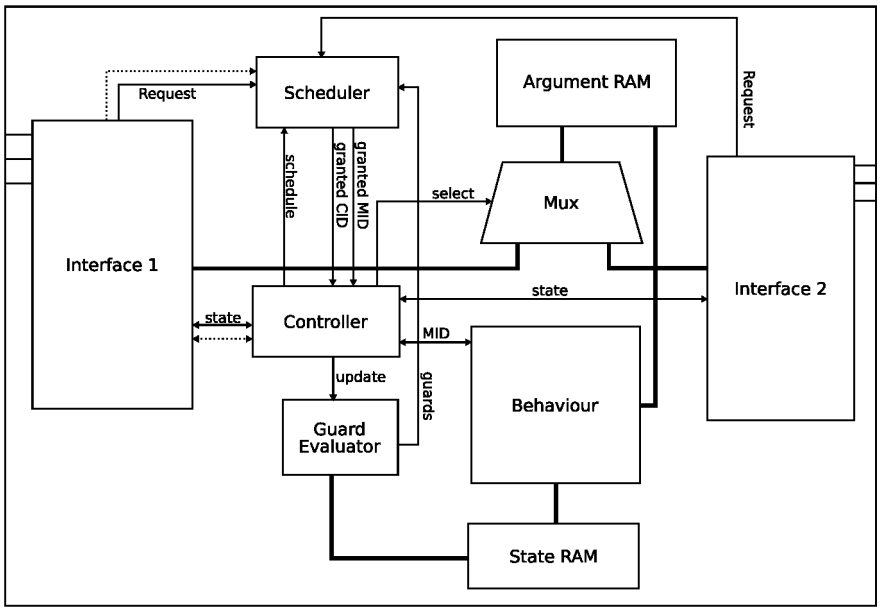

Abbildung 2.6: Shared Object nach der Synthese [GGS06]

dann ausführbar, wenn die *Guard Condition* „wahr“ ist. Die Auswertung dieser *Guard Conditions* erfolgt durch den *Guard Evaluator*.

2.2.6.3 Scheduler

Da mehrere Klienten konkurrierend auf ein Shared Object zugreifen können, ist es nötig, zu entscheiden, welchem Klienten als nächstes der Zugriff gewährt wird. Diese Aufgabe übernimmt der Scheduler-Block. Die OSSS-Bibliothek bietet eine Reihe vorgefertigter Scheduler, der Designer kann jedoch bei Bedarf auch eigene Scheduling-Algorithmen implementieren und anstelle der vorgegebenen verwenden.

Der Scheduler erhält von den Interfaces Informationen über die von den Klienten aufgerufenen Methoden. Der *Guard Evaluator* liefert dem Scheduler eine Liste von Funktionen, deren Guard-Bedingung erfüllt ist und die somit ausgeführt werden können. Basierend auf diesen Informationen gewährt der Scheduler einem der Klienten Zugriff auf eine der Methoden, indem er dem Controller die *ClientID* des Klienten und die *MethodID* der aufgerufenen Methode mitteilt.

2.2.6.4 Controller

Der Controller koordiniert zusammen mit den Interfaces den Ablauf des RMI-Protokolls sowie den Zugriff auf den Argument-RAM. Weiterhin steuert der Controller basierend auf den Informationen, die er vom Scheduler erhält, welche Methode ausgeführt wird und welchem Klienten der Zugriff ermöglicht wird.

2.2.6.5 Behaviour

Der Behaviour-Block enthält die eigentliche Funktionalität aller Methoden des Shared Objects. Der Controller bestimmt, welche Funktion ausgeführt werden soll. Der interne Zustand des Shared Object wird im *State-RAM*, die Eingabeparameter und Rückgabewerte im *Argument-RAM* gespeichert.

2.2.7 Integration in den Designflow für FPGAs

Ein weiteres Ziel von OSSS ist ein *seamless design-flow*, das heißt eine Integration der OSSS-Methodik in bestehende Designflows. Besonders bietet sich eine Integration in den Designflow für FPGAs an, da diese ein breites Anwendungsspektrum bieten. So bieten sich FPGAs für die Entwicklung von Prototypen und die Emulation des zu erstellenden Systems an[Ros97], da die Programmierung einfach verändert werden kann. Ein anderes Einsatzgebiet ist die Herstellung kleiner Serien, da für kleine Stückzahlen FPGAs kostengünstiger sind als ASICs, bedingt durch deren hohe Festkosten. Zur Integration wurde der Xilinx-FPGA-Designflow gewählt, wobei ermöglicht wird, das Xilinx Embedded Development Kit als Backend für die Systemsynthese zu verwenden. Die OSSS-Bibliothek enthält daher für viele der IP[12]-Komponenten, die das EDK[13] [Xil06a] bietet, Modelle, zum Beispiel Microblaze [Xilb] und Speichercontroller. Diese können somit für den Entwurf eines eingebetteten Systems in OSSS verwendet werden. Während der Synthese werden aus dem OSSS Design zusätzliche Informationen extrahiert, die in den folgenden Schritten genutzt werden, um die Synthese durch das EDK zu steuern.

Abbildung 2.7 stellt den Ablauf der Synthese eines OSSS-Entwurfs dar. Das Modell auf Anwendungsebene wird zu einem Modell auf Architekturebene verfeinert. Aus dem Architekturmodell erzeugen die OSSS-Synthesewerkzeuge verschiedene Zwischendateien, die als Eingabe für die weitere Synthese durch die Tools des EDK dienen. Auf die durch die OSSS-Synthese erzeugten Komponenten soll im Folgenden eingegangen werden.

[12] Intellectual Property
[13] Embedded Development Kit

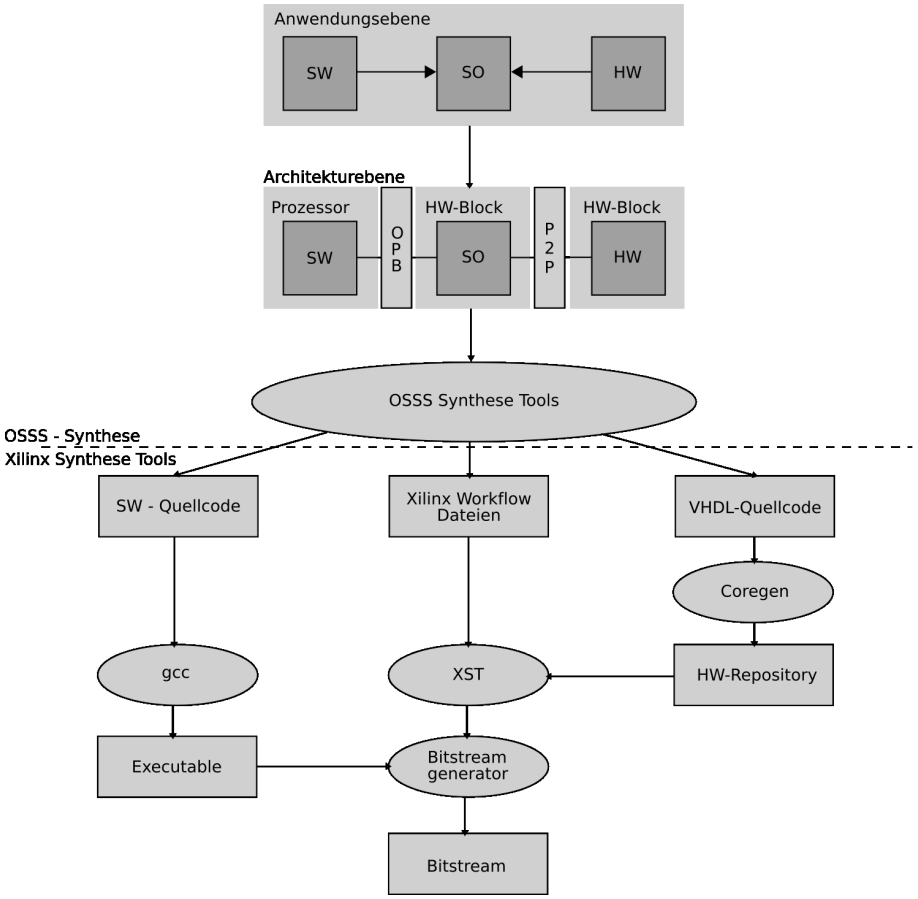

Abbildung 2.7: Der OSSS Synthese-Flow

Aus den in OSSS modellierten Software-Tasks wird C++-Sourcecode erzeugt, der neben dem eigentlichen Software-Task auch die Kommunikationsroutinen für das RMI-Protokoll enthält.

Aus den Hardwarekomponenten des Systems wird eine Beschreibung der Struktur des Gesamtsystems erzeugt. Komponenten, die auch vom EDK als Intellectual Property bereitgestellt werden, werden direkt in die Struktur des Systems eingefügt. Für Komponenten, für die keine entsprechenden Blöcke zur Verfügung stehen, wird während der OSSS-Synthese der SystemC-Sourcecode in VHDL-Sourcecode übersetzt. Der generierte Sourcecode wird, zusammen mit weiteren während der Synthese extrahierten Systeminformationen, dazu verwendet, automatisiert mit Hilfe der Werkzeuge aus dem EDK eine Komponentenbibliothek anzulegen. Diese Komponenten können dann, wie die vorgefertigten IP-Komponenten, in das Gesamtsystem eingefügt werden. Die so erzeugte Systembeschreibung wird dann durch das EDK zu einem Bitstrom synthetisiert. Der aus der Softwaresynthese entstandene Sourcecode wird kompiliert und mit Hilfe des *Bitstream initializers* in den Bitstrom eingefügt. Der so entstandene Bitstrom kann dann in ein FPGA programmiert werden.

2.3 MPEG-Audio

MPEG-Audio bezeichnet ein Verfahren zur verlustbehafteten Audiokompression, welches am Fraunhofer Institut für integrierte Schaltungen entwickelt wurde und 1992 als MPEG-1 (ISO/IEC 11172) [MPE92] standardisiert wurde. Dieser Abschnitt soll einen Überblick über die Funktionsweise eines MPEG-Audio-Dekoders geben [Ruc05].

2.3.1 Allgemeines

Die MPEG-Audio-Kompression basiert auf der Beobachtung, dass Audiosignale, die im Zeitbereich ein äußerst dynamisches Verhalten aufweisen, im Frequenzbereich häufig statischer sind. Dies lässt sich ausnutzen, indem nicht das Signal direkt komprimiert wird, sondern es zunächst in den Frequenzbereich transformiert wird, in dem es besser komprimiert werden kann.

MPEG-Audio bietet verschiedene Versionen, die sich in den verfügbaren Bit- und Samplingraten unterscheiden. Für die einzelnen Versionen definiert der Standard drei verschiedene *Layer*, die in ihrer Komplexität aufeinander aufbauen. Das heißt sowohl die Kompressionsrate als auch der Berechnungsaufwand steigt von *Layer 1* zu *Layer 3* an. Im Rahmen dieser Arbeit wird hauptsächlich *Layer 3* verwendet.

Der Berechnungsaufwand des Verfahrens ist asymmetrisch, das heißt der Aufwand für die Kodierung ist höher als für die Dekodierung. Dies ermöglicht auch auf weniger leistungsfähiger (und damit kostengünstigerer) Hardware eine Echtzeitdekompression.

Neben verschiedenen Bit- und Samplingraten sieht der Standard vor, sowohl einkanalige (Mono), als auch zweikanalige (Stereo) Audiodaten komprimieren zu können. Um im Falle von Stereodaten eine zusätzliche Verbesserung der Kompressionsrate zu erzielen, sind verschiedene Möglichkeiten der Stereoverarbeitung vorgesehen.

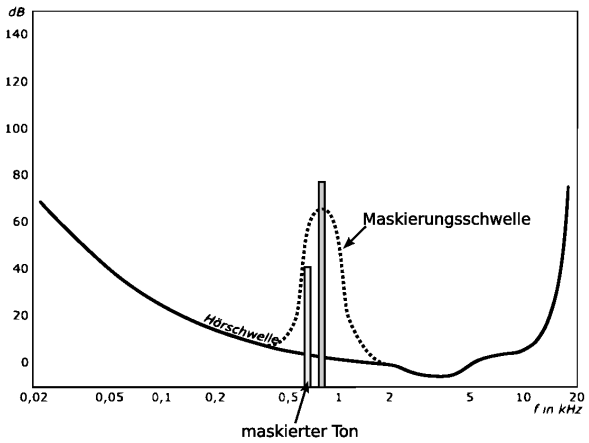

Abbildung 2.8: Schematische Darestellung der Maskierung während der Kodierung

Neben verschiedenen verlustfreien Verfahrensschritten kommen bei der MPEG-Audio-Kompression auch verlustbehaftete Verfahren zum Tragen. Diese werden verwendet, um Anteile aus dem Eingabedatenstrom herauszufiltern, die für den Höreindruck nicht benötigt werden. Hierzu wird bei der Kompression ein psychoakustisches Modell verwendet, welches die für das Hörempfinden nicht relevanten Signalanteile bestimmt. Diese Signalanteile werden dann bei der Kompression verworfen. Abbildung 2.8 zeigt am Beispiel der so genannten Maskierung, wie während der Kodierung nicht hörbare Anteile herausgefiltert werden. Da laute Töne leisere Töne, die im Frequenzbereich in ihrer Nähe liegen, überdecken, werden die überdeckten Töne nicht mit kodiert, da sie keinen Einfluss auf den Höreindruck haben.

2.3.2 Aufbau des Datenstroms

Der Datenstrom besteht aus einzelnen Frames. Jeder dieser Frames kodiert 36 Subbandsamples für jedes der 32 verwendeten Subbänder, also genau 1152 Ausgabesamples pro Kanal und Frame. Somit ist die Dauer, die die Ausgabe eines Frames benötigt abhängig von der Abtastrate. Bei einer Samplingrate von 44,1 kHz beträgt die Zeit für die Wiedergabe der 1.152 Samples circa 26,122 ms. Die einzelnen Frames bestehen wie-

derum aus einem Header, den so genannten Seiteninformationen sowie den eigentlichen komprimierten Daten.

2.3.2.1 Header

Der Header eines Frames besitzt eine feste Länge von 32 Bit. Tabelle 2.1 enthält die Bedeutungen der einzelnen Bits innerhalb des Headers.

Funktion	Anzahl Bit	Beschreibung
Synchronisationssequenz	11	Dient der Erkennung eines Frame-Anfangs
MPEG Version	2	00-MPEG2,5; 01-Nicht benutzt; 10-MPEG2; 11-MPEG1
Layer	2	00-nicht benutzt; 01-Layer3; 10-Layer2; 11-Layer1
Cyclic Redundancy Check	1	Wenn gesetzt, folgt auf den Header eine 16 Bit lange Prüfsumme
Bitrate	4	Bestimmt die Bitrate des Datenstroms
Sampling Freq.	2	Bestimmt die Sampling Frequenz
Padding	1	Wenn gesetzt, ist der Frame um genau ein Byte größer. Dient dem genauen Einhalten der gewählten Bitrate
Private Bit	1	undokumentiert
Stereo	2	Wählt zwischen Mono, Dualchannel und Stereo
Stereo Codierung	2	Bestimmt die Art der Stereokodierung
Copyright	1	Gesetzt, wenn es sich um urheberrechtlich geschütztes Material handelt
Original	1	Gesetzt, wenn der Datenstrom ein Original ist; nicht gesetzt bei einer Kopie
Emphasis	2	00-Keine; 01-50/15 ms; 10-nicht genutzt; 11-CCIT J.17

Tabelle 2.1: Aufbau des MPEG-Audio-Headers

Der Header ist die einzige Datenstruktur innerhalb der Frames, deren Bedeutung für alle *Layer* gleich ist. Die Bedeutung der anderen Datenstrukturen ist abhängig von dem gewählten *Layer*. Die Größe des Frames kann mit folgender Formel berechnet werden:

$$Size = \lfloor \frac{1.152}{Samplingrate} \frac{Bitrate}{8} + Padding \rfloor = \lfloor 144 \cdot \frac{Bitrate}{Samplingrate} + Padding \rfloor \quad (2.1)$$

So beträgt zum Beispiel die Größe eines Frames in Byte bei einer Samplingrate von 44,1 kHz, einer Bitrate von 320 kb/s sowie gesetztem *Padding-Bit*:

$$Size = \lfloor 144 \cdot \frac{320.000}{44.100} + 1 \rfloor = \lfloor 1.045,898 \rfloor = 1.045 Bytes$$

2.3.2.2 Seiteninformationen

Die Seiteninformationen enthalten *Layer*-spezifische Informationen, die für die Dekodierung der Framedaten benötigt werden. Die Seiteninformationen haben keine festgelegte Länge, sie variiert mit den kodierten Daten.

2.3.2.3 Datenteil

Nach den Seiteninformationen folgen die eigentlichen komprimierten Audiodaten. Da diese bei *Layer 3* Huffman-kodiert werden, ist es möglich, dass nicht alle zur Verfügung stehenden Bit benutzt werden. Dieser ungenutzte Platz bildet das so gennannte *bit reservoir*. Abbildung 2.9 zeigt die Verwendung des Bit-Reservoirs. Dieses kann mit den Daten des folgenden Frames gefüllt werden, falls für dessen Komprimierung mehr Bit benötigt werden, als bei der gewählten Bitrate pro Frame zur Verfügung stehen.

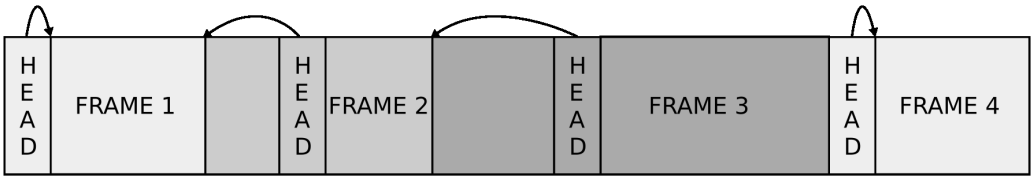

Abbildung 2.9: Verwendung des *bit reservoirs*

2.3.3 Ablauf der Dekodierung

Die Dekodierung eines MPEG-Audio-Datenstroms läuft in verschiedenen Schritten ab, die jeder komprimierte Frame durchläuft. Die Anzahl der Stufen ist abhängig von dem gewählten *Layer*. Abbildung 2.10 zeigt die einzelnen zur Dekodierung benötigten Schritte am Beispiel eines *Layer 3* kodierten Bitstroms. Im Folgenden sollen die einzelnen Schritte detaillierter beschrieben werden.

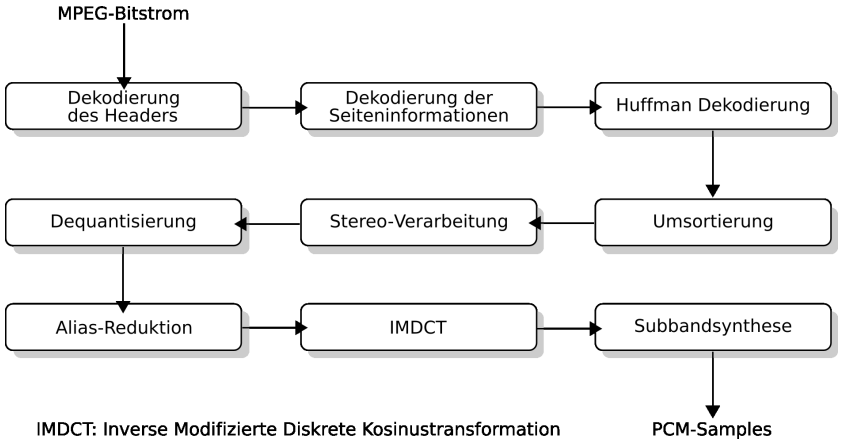

Abbildung 2.10: Ablauf der MPEG-Audio-Dekodierung

2.3.3.1 Dekodierung des Headers

Zunächst muss der Datenstrom synchronisiert werden, das heißt der Anfang eines Frames muss gesucht werden. Ein neuer Frame beginnt immer an einer Byte-Grenze, dadurch wird die Suche nach der Synchronisationssequenz am Frameanfang vereinfacht. Ist die Synchronisationssequenz gefunden, werden die darauf folgenden Header-Informationen dekodiert. Die Informationen aus dem Header werden benötigt, um zu bestimmen, welche Dekodierungsschritte für den jeweiligen Datenstrom nötig sind.

2.3.3.2 Verarbeitung der Seiteninformationen

Die Seiteninformationen enthalten unter anderem Informationen über die Benutzung des Bit-Reservoirs, Skalierungsfaktoren sowie Steuerinformationen für die Huffman-Dekodierung der in den *Main Data* abgelegten Frequenzdaten. Hierbei werden jedoch nicht die zur Kodierung verwendeten Huffman-Tabellen übertragen, sondern für jede Frequenzregion eine der im Standard definierten 32 statischen Tabellen ausgewählt.

2.3.3.3 Huffman-Dekodierung

Bei der Huffman-Kodierung handelt es sich um eine verlustfreie Entropiekodierung, die darauf basiert, dass häufig auftretenden Werten kurze Kodewörter zugeordnet werden und selten auftretende Werte mit langen Kodewörtern kodiert werden. Diese Kodierung wird in diesem Schritt rückgängig gemacht, indem in einer in den Seiteninformationen festgelegten Tabelle die dekodierten Werte gesucht werden.

2.3.3.4 Dequantisierung

Bei der MPEG-Audio-Kodierung werden die Frequenzlinien quantisiert, das heißt die kontinuierlichen Werte werden auf diskrete Werte abgebildet. Da das menschliche Gehör Lautstärkenunterschiede bei leisen Tönen besser differenzieren kann als bei höheren Lautstärken, wird eine nichtlineare Quantisierung verwendet. Bei der Dequantisierung wird dieser Schritt rückgängig gemacht.

2.3.3.5 Stereo-Verarbeitung

MPEG-Audio sieht verschiedene Möglichkeiten vor, Stereo-Audiodaten zu kodieren. Neben der Möglichkeit, diese als zwei getrennte Kanäle zu kodieren, bietet der MPEG-Standard die Möglichkeit, die beiden Kanäle vor der Kodierung zusammenzufassen. Hierbei wird ausgenutzt, dass sich die beiden Stereokanäle häufig nur wenig unterscheiden. Wurde bei der Kodierung ein so genanntes Joint-Stereo-Verfahren [MPE92] gewählt, so werden die beiden getrennten Stereokanäle in diesem Schritt wieder hergestellt.

2.3.3.6 Umsortieren der Samples

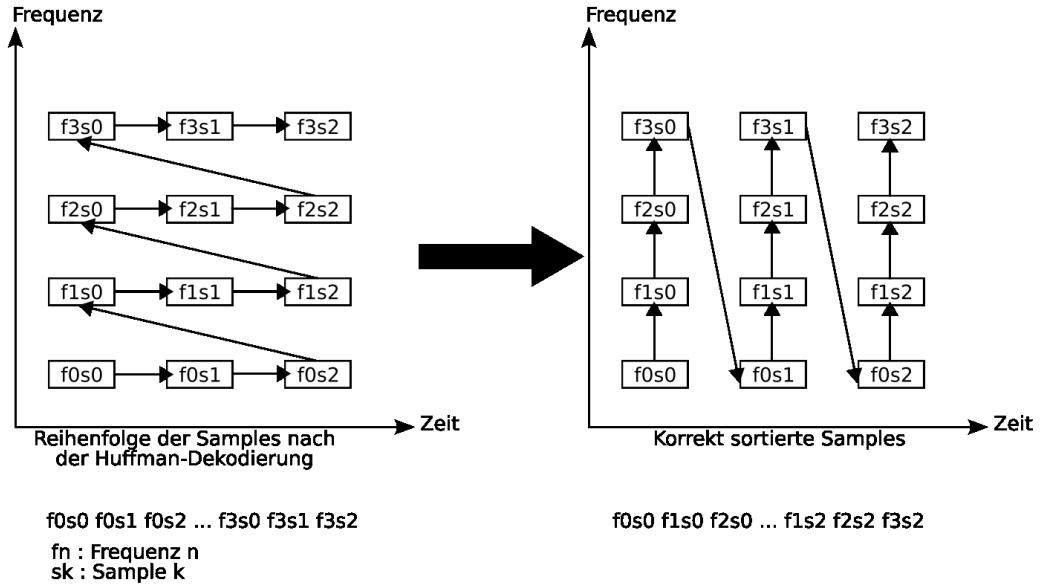

Abbildung 2.11: Umordnung der Samples

Die Huffman-Kodierung ist besonders dann effizient, wenn viele gleiche Werte auftreten. Dies ist besonders bei zeitlich aufeinanderfolgenden Samples der gleichen Frequenz zu

beobachten. Daher werden die Samples bei der Kodierung dementsprechend umsortiert. Bei der Dekodierung muss dieser Vorgang rückgängig gemacht werden. Abbildung 2.11 zeigt die Umsortierung der Samples in diesem Schritt. Nach der Huffman-Dekodierung folgen die verschiedene Subbandsamples gleicher Frequenz direkt aufeinander. Für die weiteren Schritte der Dekodierung ist es jedoch erforderlich, die Samples so anzuordnen, dass sie wieder zeitlich sortiert sind, das heißt Samples, die zu einem Abtastzeitpunkt gehören, zusammen gruppiert werden.

2.3.3.7 Alias-Reduktion

Bei der Kodierung werden die Audiodaten in 32 Subbänder aufgespalten. Dies erfolgt mittels 32 Bandpassfiltern in einer Polyphase-Filterbank. Da die Bandpassfilter nicht ideal sind, überlagern sich die einzelnen Frequenzbänder an den Seiten. Dies führt dazu, dass die Subbänder Frequenzen enthalten, die zu einem benachbarten Subband gehören. Dieses nennt man Aliasing. Der MPEG-Audio-Standard definiert eine Reihe von Berechnungen, die dieses Phänomen beseitigen.

2.3.3.8 Inverse Modifizierte Diskrete Kosinustransformation

Der nächste Schritt bei der Dekodierung ist die Berechnung der inversen MDCT[14]. Bei der Kodierung wird eine MDCT auf die Subbandsamples angewandt, um die Auflösung im Frequenzbereich zu erhöhen. Die inverse MDCT erzeugt aus je 18 der 576 Frequenzlinien ein Subbandsample, so dass nach diesem Schritt jeder Frame aus 36 Samples für jedes der 32 Subbänder besteht.

2.3.3.9 Subbandsynthese

Der letzte Schritt der Dekodierung dient dazu, aus den Subbandsamples wieder PCM[15]-Samples zu gewinnen. Hierzu ist es nötig, die bei der Kodierung durch die Polyphase-Filterbank erfolgte Filterung rückgängig zu machen. Eine effiziente Möglichkeit der Umsetzung dieses inversen Filters basiert auf einer 32-Punkt-DCT[16]. Die Effizienz der Implementierung ist von besonderer Bedeutung, da die Subbandsynthese, abhängig von der Samplingrate, pro Sekunde bis zu 3.000 mal (bei $f_{samp} = 48\ kHz$) ausgeführt wird.

[14] Modified Discrete Cosine Transformation
[15] Pulse Code Modulation
[16] Discrete Cosine Transformation

3 Entwurf

In diesem Kapitel soll näher auf die beim Design des Testsystems getroffenen Entscheidungen eingegangen werden. Zunächst soll das Ziel der Arbeit erläutert werden. Im Anschluss daran wird die zu erstellende Hard- und Software eingehender betrachtet.

3.1 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine Analyse und einen Vergleich der Effizienz eines automatisch generierten Interfaces zur Hardware-/ Softwarekommunikation, dem OSSS-RMI-Protokoll, mit einer manuell angefertigten Lösung durchzuführen. Hierzu ist es zunächst zweckmäßig, näher zu definieren, in welchem Sinne der Effizienzbegriff in dieser Arbeit verwendet werden wird.

Allgemein bezeichnet der Begriff Effizienz das Verhältnis der Größe einer erbrachten Leistung zu der Größe des eingesetzten Aufwands. Ist also der eingesetzte Aufwand niedrig und die erbrachte Leistung hoch, spricht man von einer hohen Effizienz. In der Informatik wird der Begriff Effizienz häufig im Zusammenhang mit der Komplexität von Algorithmen verwendet. Ein Algorithmus mit einem Berechnungsaufwand, der linear zur Größe der Eingabedatenmenge ist, ist effizienter als ein Algorithmus, dessen Berechnungsaufwand polynomiell oder exponentiell mit der Größe der Eingabedatenmenge steigt.

Bezogen auf den Entwurf eingebetteter Systeme lassen sich verschiedene Dimensionen der Effizienz unterscheiden. Einerseits die Effizienz des Workflows selbst, die so genannte Designeffizienz, andererseits die Effizienz des erstellten Systems. Die Effizienz des Systems lässt sich wiederum hinsichtlich verschiedener Zielgrößen bestimmen, zum Beispiel Flächenbedarf, Leistungsfähigkeit oder Energieverbrauch.

Die im Rahmen dieser Arbeit durchgeführte Effizienzanalyse soll primär im Hinblick auf die Effizienz des erstellten Systems erfolgen und die Designeffizienz des OSSS-Ansatzes nur als Nebenaspekt betrachten, da diese nur schwer objektiv messbar ist. Bei den möglichen Zielgrößen beschränkt sich die Betrachtung auf die Leistungsfähigkeit und den Flächenbedarf des Systems. Die Energieeffizienz des Systems soll in dieser Arbeit nicht betrachtet werden. Als Maß für die Effizienz soll der Vergleich eines automatisch generierten Verfahrens zur Hardware-Software-Kommunikation mit einem manuell erzeugten

dienen. Die Effizienz einer automatisch generierten Implementierung ist demnach hoch, wenn ihre Leistungsfähigkeit größer oder gleich, bzw. der Flächenverbrauch geringer oder gleich der manuell erstellten Referenzimplementierung ist.

3.2 Hardware

In diesem Abschnitt soll näher auf die Testumgebung eingegangen werden, die für die Effizienzanalyse verwendet wird. Als Beispielanwendung, die im Rahmen der Untersuchung verwendet wird, dient ein MPEG-Audio-Dekoder auf Basis einer Open Source C-Implementierung [Und].

3.2.1 Entwurf des Systems

Zunächst werden die an das System gestellten Anforderungen bestimmt, um daraus eine Spezifikation des Gesamtsystems zu entwickeln.

- Implementierung mit Hilfe eines Xilinx-FPGAs
- Verwendung eines Microblaze Soft-Core Prozessors
- Verwendung des ML-401 Development Boards

Um die Effizienz des OSSS-RMI-Protokolls bestimmen zu können, werden drei Versionen des MPEG-Audio-Dekoders angefertigt.

1. SW-Implementierung
2. Implementierung mit Hardware DCT und eigenem Kommunikationsprotokoll
3. HW/SW-Implementierung unter Verwendung des OSSS-RMI-Protokolls

Die SW-Implementierung dient als Basis für die beiden in Punkt 2 und 3 genannten Implementierungen. Die HW-Implementierung mit dem selbstentwickelten Kommunikationsprotokoll dient als Referenzimplementierung, um die Effizienz des automatisch generierten Kommunikationsinterfaces bestimmen zu können.

3.2.2 Auswahl der Komponenten

Das verwendete Development Board ML-401 von Xilinx verfügt über eine große Anzahl verschiedener Peripheriekomponenten und Schnittstellen:

- Xilinx Virtex-4 LX25 FPGA
- 64 MiB DDR-SDRAM[1]
- 8 MiB Flash-ROM
- 9 MiB SRAM[2]
- LC[3]-Display
- CompactFlash-Controller
- 24-Bit Video DAC[4]
- *AC97*[5] *Codec*
- verschiedene I/O Schnittstellen: USB, RS-232, JTAG[6]-Port, Ethernet, PS/2
- verschiedene Taster, Schalter und LEDs[7]

Für das als Testplattform verwendete Design werden jedoch nicht alle verfügbaren Komponenten benötigt. Die folgenden Komponenten finden in dem erstellten Design Verwendung:

- FPGA
- DDR-SDRAM
- CompactFlash
- *AC97 Codec*
- RS-232

[1] Double Data Rate Synchronous Dynamic Random Access Memory
[2] Static Random Access Memory
[3] Liquid Crystal
[4] Digital/Analog Converter
[5] Audio Codec '97
[6] Joint Test Action Group
[7] Light Emitting Diode

- JTAG

Das FPGA wird verwendet, um den Microblaze [Xilb] Prozessor sowie die für die Anbindung der sonstigen Systemkomponenten und Schnittstellen benötigten *glue-logic* aufzunehmen. Das auf dem Board vorhandene Virtex-4 FPGA [Xila] bietet neben 10.752 Slices, die für die Implementierung der benötigten Logikschaltungen verwendet werden können, zusätzlich noch interne RAM-Blöcke mit einer Kapazität von insgesamt 1.292 Kib.

Da die Größe der Software in Verbindung mit den für die Dekodierung benötigten Puffern die maximale Größe eines FPGA-internen Block-RAM von 64 KiB übersteigt, ist es nötig, einen externen Speicher zu verwenden. Der auf dem Development Board vorhandene DDR-SDRAM wird benutzt, da es sich um den größten auf dem Board verfügbaren Speicher handelt und somit Versuche mit verschiedenen Puffer-Szenarien möglich sind.

Da MPEG-Audio-Daten trotz Komprimierung noch einen Speicherbedarf von bis zu mehreren Megabyte pro Minute haben, ist es nötig, ein geeignetes Speichermedium auszuwählen. Möglich ist auch, den zu dekodierenden Datenstrom über eine der verfügbaren I/O Schnittstellen an das Design zu übertragen, die Daten also von einem Hostrechner an das Design zu *streamen*. Aufgrund der Bitraten des Datenstroms ist dies jedoch nur über die Ethernet- oder die USB-Schnittstelle möglich, über die serielle RS-232-Schnittstelle ist eine Übertragung der höheren Bitraten von bis zu 320 kb/s nicht möglich. Ein weiteres Problem bei diesem Ansatz ist, dass die Übertragung der Audiodaten auf die Testplattform für jeden Testlauf erneut durchgeführt werden müsste und die Entwicklung einer geeigneten Software für das Host-System notwendig machen würde. Weiterhin ist es, zum Beispiel zu Demonstrationszwecken, vorteilhaft, ein System zu entwickeln, das unabhängig von einem Host-System ist.

Daher wird ein nichtflüchtiger Speicher auf dem Development Board verwendet, um die Daten zu speichern. Da die Größe des auf dem Board vorhandenen Flash-ROMs mit 8 MiB nicht ausreichend ist, um Tests mit höheren Bitraten und über längere Zeit durchzuführen, und zudem die Programmierung des Flash-ROMs wiederum das Übertragen der Daten über die JTAG- oder serielle-Schnittstelle erfordern würde, wird die CompactFlash-Karte als Speichermedium verwendet. Diese bietet mit einer Speicherkapazität von 32 MB ausreichend Platz für verschiedene Testdatenströme und kann bei Bedarf einfach gegen ein Modell mit größerer Speicherkapazität ausgetauscht werden. Ein weiterer Vorteil ist, dass die Testdaten mit Hilfe eines so genannten USB-CardReaders auf die CompactFlash-Karte übertragen werden können, ohne dass hierfür spezielle Hard- oder Software auf der Testplattform zur Verfügung gestellt werden muss.

Die Ausgabe der dekodierten Daten soll in analoger Form über Kopfhörer erfolgen. Hierzu wird ein ebenfalls auf dem Board vorhandener Audio-Codec nach AC97-Standard[Nat] verwendet. Der Codec erlaubt die Aufnahme bzw. Wiedergabe analoger Stereo-Audiosignale mit einer Abtastrate von 48 kHz bei einer Genauigkeit von 16 Bit.

Für die Kommunikation des Testsystems mit dem Host-System sowie für die Ausgabe der Messdaten wird die serielle RS-232 Schnittstelle verwendet, da so die Daten auf der Seite des Host-Systems einfach in Dateien gespeichert werden können und die Ansteuerung der Schnittstelle auf Seiten des Testsystems über Bibliotheksfunktionen möglich ist.

Die JTAG-Schnittstelle dient dazu, neben der Konfiguration des FPGAs auch das auszuführende Programm auf das Testsystem zu übertragen. Weiterhin wird die JTAG-Schnittstelle dazu verwendet, eine Verbindung zwischen einem auf dem Host-System ausgeführten Debugger und dem zu debuggenden System aufzubauen, sowie die Ausführung des Programms zu kontrollieren.

3.2.3 FPGA-Design

Basierend auf den spezifizierten Anforderungen und den ausgewählten Hardwarekomponenten wird eine Auswahl von Systemkomponenten getroffen, die in dem FPGA implementiert werden sollen. Abbildung 3.1 zeigt hierzu einen grafischen Überblick über die Komponenten des zu erstellenden Systems. Diese Komponenten sind:

- Xilinx Microblaze
- DDR-SDRAM Controller
- SystemACE Controller
- AC97-Digital-Controller
- RS-232 UART[8]
- LMB[9] Block-RAM Interface Controller
- Block-RAM
- Debug-Modul
- Timer / Counter
- OPB
- verschiedene DCMs[10] und Inverter

[8] Universal Asynchronous Receiver Transmitter
[9] Local Memory Bus
[10] Digital Clock Manager

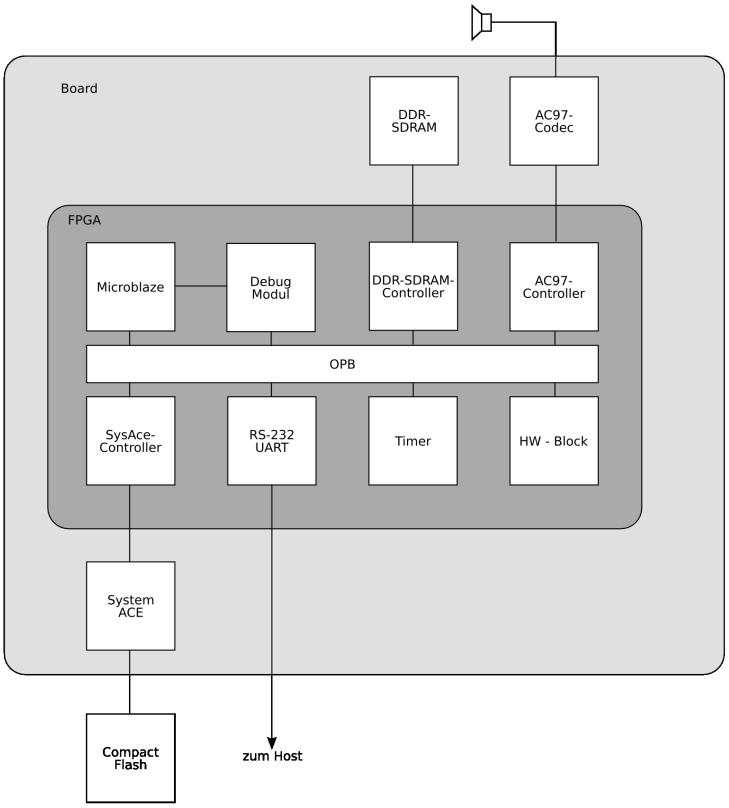

Abbildung 3.1: Schematische Darstellung des Testsystems

Im Folgenden soll näher auf die einzelnen verwendeten Systemkomponenten eingegangen werden.

3.2.3.1 Microblaze

Bei dem Xilinx Microblaze [Xilb] handelt es sich um einen 32-Bit Soft-Core, das heißt der Prozessor liegt als konfigurierbare Hardwarebeschreibung vor und ist im Gegensatz zu einem Hard-Core nicht fest in das Silizium des FPGAs integriert. Der Vorteil eines Soft-Cores ist, dass verschiedene Prozessorkonfigurationen einfach evaluiert werden können und eine Abwägung zwischen Flächenbedarf und benötigter Rechenleistung möglich ist.

Abbildung 3.2 zeigt den internen Aufbau des Microblaze. Der Microblaze verwendet eine Harvard-Architektur. Daten und Instruktionen werden also in verschiedenen Speichern abgelegt. Für die Adressierung der Speicher werden 32-Bit Adressen verwendet, so dass der verfügbare Adressraum eine Größe von 4 GiB hat. Der Zugriff auf Peripheriekomponenten erfolgt mittels *Memory Mapped I/O*, das heißt der Speicherbereich einer

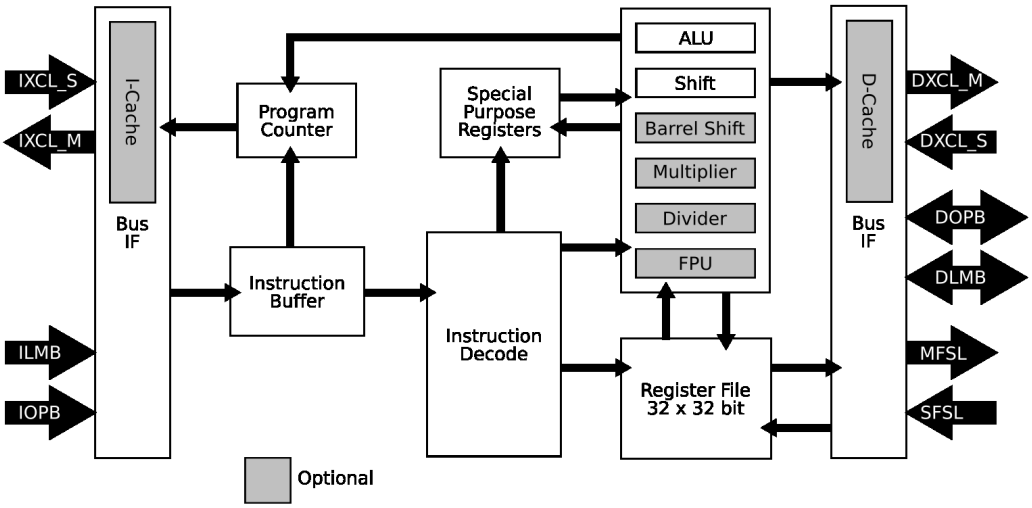

Abbildung 3.2: Microblaze Blockdiagramm

Komponente wird in den Adressraum des Microblaze eingeblendet und der Zugriff auf die Register einer Komponente erfolgt als Zugriff auf eine Speicheradresse.

Der Microblaze verwendet eine *RISC*[11]-Architektur. Der Befehlssatz verfügt also, verglichen mit einer *CISC*[12]-Architektur, nur über eine beschränkte Anzahl an Befehlen, die jedoch mit höherer Geschwindigkeit ausgeführt werden können. Der Microblaze verfügt über eine Pipeline mit fünf Stufen, das heißt fünf Befehle befinden sich gleichzeitig in verschiedenen Stufen der Abarbeitung. Dies führt zu einem verbesserten Befehlsdurchsatz und einer besseren Auslastung der einzelnen Funktionsblöcke innerhalb des Prozessors.

Die in Abbildung 3.2 grau hinterlegten Komponenten des Prozessors sind optional. Das heißt der Designer kann sie auf Kosten der Rechenleistung deaktivieren, um so den Flächenverbrauch des Prozessors zu reduzieren. Die optionalen Komponenten sind im Einzelnen folgende:

1. Instruktions- und Datencache
2. Barrel Shift
3. Multiplizierer
4. Dividierer

[11] Reduced Instruction Set Computing
[12] Complex Instruction Set Computing

5. Fließkommaeinheit

Die Instruktions- und Datencaches sind vor allem dann von Vorteil, wenn externe Speicher wie zum Beispiel der DDR-SDRAM als Programm- und Datenspeicher verwendet werden sollen, da durch das Caching die ansonsten bei Speicherzugriffen auftretendenen langen Latenzzeiten aufgefangen werden.

Die Barrel-Shift-Einheit ermöglicht Bitschiebeoperationen um beliebige Werte innerhalb eines Taktes. Ist diese nicht aktiviert, werden Schiebeoperationen um mehrere Bit durch mehrere Ein-Bit-Schiebeoperationen nachgebildet. Für Anwendungen, die Gebrauch von vielen Schiebeoperationen machen, kann es daher von Vorteil sein, die Barrel-Shift-Einheit zu aktivieren.

Ebenso kann durch die Aktivierung der Hardware-Multiplizierer, -Dividierer und Fließkommaeinheit die Verarbeitungsgeschwindigkeit für Anwendungen, die ausgiebig Gebrauch von solchen Operationen machen, erhöht werden. Sind die entsprechenden Funktionsblöcke nicht aktiviert, können die Operationen in Software emuliert werden, dies ist jedoch besonders bei der Fließkommaarithmetik mit einem deutlichen Performanceverlust verbunden. Für Anwendungen, die solche Operationen nicht oder nur in geringem Maße nutzen, kann der Flächenbedarf reduziert werden, indem die nicht verwendeten Funktionsblöcke deaktiviert werden. Für das im Rahmen dieser Arbeit zu erstellende System wird der Microblaze in einer Konfiguration mit aktivierten Caches, Barrel-Shift und Hardware-Multiplizierer verwendet. Dividierer und Fließkommaeinheit werden nicht benötigt und daher deaktiviert. In der verwendeten Konfiguration benötigt der Microblaze eine Fläche von 1.724 Slices auf dem FPGA.

Die Kommunikation des Microblaze mit den verschiedenen Speichern und anderen Peripheriekomponenten erfolgt über verschiedene Bussysteme und Punkt-zu-Punkt-Verbindungen. Die zentrale Kommunikationsstruktur bildet dabei der OPB[13]-Bus [IBM]. Dieser verbindet die Peripheriekomponenten mit dem Microblaze. Der Microblaze verfügt über zwei OPB-Interfaces (je eines für Daten- und Instruktionsbus), die mit unterschiedlichen Bussen, aber auch mit demselben Bus verbunden werden können. Der OPB-Bus überträgt die Daten mit der vollen Taktrate des Microblaze und erlaubt es, mehrere Busmaster-Komponenten an einem Bus zu betreiben. Zusätzlich zu den OPB Interfaces verfügt der Microblaze über ein LMB-Interface, über das die FPGA-Internen Block-RAMs angebunden werden können. Der LMB ist ein synchroner Bus, der über wenige Kontrollsignale verfügt und dazu dient, Zugriff auf die BRAMs innerhalb eines Taktes zu ermöglichen. Der LMB unterstützt, im Gegensatz zum OPB, jeweils nur eine Master-Komponente. Weiterhin verfügt der Microblaze über acht FSL[14]-Verbindungen, die dazu verwendet werden können, die Funktionalität des Microblazes zu erweitern. Der Instruktionssatz des Microblazes enthält Instruktionen, die ein Schreiben oder Lesen über die FSL-Verbindungen ermöglichen. Diese Verbindungen können

[13] On-Chip Peripheral Bus
[14] Fast Simplex Link

verwendet werden, Funktionen, deren Berechnungen durch eine Softwareroutine zu viel Rechenzeit benötigen würde, durch die Verwendung dedizierter Hardware zu beschleunigen. Die FSL-Verbindungen beinhalten FIFO[15]-Puffer konfigurierbarer Länge. Das Signalinterface der FSL-Verbindungen entspricht dem eines FIFO-Puffer, somit ist es möglich, Co-Prozessor-Komponenten mit dem Microblaze zu verbinden, ohne ein komplexes Busprotokoll implementieren zu müssen.

3.2.3.2 DDR-SDRAM-Controller

Da der auf dem Board vorhandene DDR-SDRAM zur Speicherung des Programms und der Daten verwendet wird, ist es nötig, eine Verbindung zum OPB-Bus herzustellen. Diese Aufgabe übernimmt der DDR-SDRAM-Controller[Xil06b]. Der von Xilinx als vorgefertigter IP-Block bereitgestellte Controller setzt die Buszugriffe auf die Ansteuersignale des DDR-SDRAMs um. Dazu werden aus den Adresssignalen die entsprechenden Zeilen- und Spaltenadressen erzeugt und der Ablauf der Zugriffe mit Hilfe eines Zustandsautomaten gesteuert. Da innerhalb von DRAM[16]-Bausteinen die Informationen in Form von Ladungen in einzelnen Kondensatoren gespeichert werden und diese Kondensatoren einer gewissen Entladung unterliegen, müssen die gespeicherten Daten periodisch ausgelesen und neu geschrieben werden, dieser Vorgang wird *refresh* genannt. Zusätzlich zu der Umsetzung der Adresssignale generiert der Controller die für diesen periodischen Refresh benötigten Signale.

3.2.3.3 CompactFlash-Karte

Um die zu dekodierenden Daten zur Verfügung zu stellen, wird eine CompactFlash-Speicherkarte verwendet. CompactFlash-Karten verwenden das gleiche Signalinterface wie ATA[17]-Festplatten. Auf diese Weise wird die komplexe Ansteuerung der Flash-Bausteine nach außen verborgen. Über das ATA-Interface kann auf mehrere interne Register zugegriffen werden, über die die Adressierung und die Datenübertragung erfolgt. Die Übertragung erfolgt dabei nicht taktsynchron, das heißt das Signalinterface beinhaltet keine Taktsignale. Die Synchronisation zwischen den Komponenten erfolgt über Kontrollsignale, die den Lese- bzw. Schreibzugriff steuern.

Die Anbindung erfolgt über den auf dem Board vorhandenen SystemACE Baustein[Xil02]. Der eigentliche Zweck des SystemAce Bausteins ist, ein angeschlossenes FPGA mit einem auf der CF-Karte abgelegten Bitstrom zu programmieren. Diese Funktion wird jedoch für das zu erstellende System nicht verwendet, stattdessen wird das SystemACE für die Anbindung der CF-Karte an den Microblaze verwendet. Die

[15] First In - First Out
[16] Dynamic Random Access Memory
[17] Advanced Technology Attachment

Ansteuerung des externen SystemACE erfolgt dabei über einen Controller innerhalb des FPGAs. Die hierfür von der Entwicklungsumgebung bereitgestellten Treiber enthalten Routinen für den Zugriff auf verschiedene Dateisysteme, um einen Zugriff auf die auf der CF-Karte abgelegten Dateien zu ermöglichen.

3.2.3.4 AC97 Digital Controller

Die AC97-Spezifikation von Intel sieht eine Aufteilung des Audio-Systems in zwei getrennte Komponenten vor. Einerseits den *AC97 Codec*, der die Digital / Analog-Wandlung sowie die analoge Signalverarbeitung ausführt, andererseits den *AC97 Digital Controller*, oder kurz DC97, über den die Anbindung des Codecs an den Systembus erfolgt. Die Kommunikation zwischen DC97 und *AC97 Codec* erfolgt dabei über eine *AC-Link*[Nat] genannte Verbindung.

Über diese Verbindung werden die Audiodaten bitseriell mit einer Taktrate von 12,288 MHz übertragen. Für die Übertragung kommt ein Zeitmultiplexverfahren zur Anwendung. Die Übertragung erfolgt frameweise, ein Frame enthält jeweils die zu einem Abtastzeitpunkt gehörigen Daten jedes Kanals. Insgesamt besteht ein Frame aus zehn Audiokanälen sowie drei Kanälen für Steuer- und Kontrollinformationen. Einer der Steuerkanäle hat eine Bitbreite von 16 Bit, die übrigen verwenden, ebenso wie die Kanäle für die Audiodaten, eine Bitbreite von 20 Bit pro Kanal. Daraus ergibt sich eine Länge der Frames von $16 + 12 \cdot 20 = 256\ Bit$. Bei einer Taktrate von 12,288 MHz folgt daraus eine maximale Abtastrate pro Kanal von $\frac{12.288.000\ Hz}{256} = 48.000\ Hz$. Die Spezifikation erlaubt jedoch die Abtastrate zu verdoppeln, indem zwei Kanäle gebündelt werden.

Da die Ausgabe der dekodierten Audiodaten über den auf dem Development Board vorhandenen *AC97 Codec* erfolgen soll, ist die Implementierung eines *AC97 Digital Controllers* innerhalb des FPGAs notwendig. Dieser verfügt über verschiedene Register zur Konfiguration des Codecs, sowie Puffer für Aufnahme und Wiedergabe. Der Controller bietet zwei konfigurierbare Interrupt-Signale, die verwendet werden können, um die Aufnahme- bzw. Wiedergabepuffer interruptgesteuert zu leeren, bzw. zu füllen.

3.2.3.5 Timer

Um genauere Aussagen über das Zeitverhalten der untersuchten Implementierungen zu ermöglichen, wird ein Timer in das Testsystem integriert. Als Timer wird eine von Xilinx als IP-Block bereitgestellte Komponente verwendet[Xil05]. Dieser Timer / Counter kann in verschiedenen Betriebsarten verwendet werden. Für Messzwecke im Testsystem wird der Timer zum Zählen der Takte verwendet. Hierzu wird der Timer in einen Modus versetzt, in dem er einen internen Zähler mit jedem Takt inkrementiert. Über Zugriffe

auf Register des Timers ist es möglich, den Timer zu starten, zu stoppen, den aktuellen Zählerstand auszulesen oder den Zählerstand auf null zu setzen.

3.2.3.6 Hardware DCT

Da insbesondere die HW / SW - Kommunikation analysiert werden soll, wird ein Teil der Dekodierung mit Hilfe dedizierter Hardware ausgeführt. Dazu ist es notwendig, einen Hardwareblock mit entsprechender Funktionalität zu entwickeln. Die Funktion, die ausgelagert werden soll, ist die während der Subbandsynthese verwendete 32-Punkt-DCT. Die Auslagerung der diskreten Kosinustransformation erscheint aus verschiedenen Gründen sinnvoll: Sie eignet sich relativ gut für eine parallele Berechnung und wird während der Subbandsynthese jedes Frames 36 mal pro Kanal ausgeführt. Daraus ergibt sich das Potential einer signifikanten Performancesteigerung verglichen mit einer reinen Softwareimplementierung.

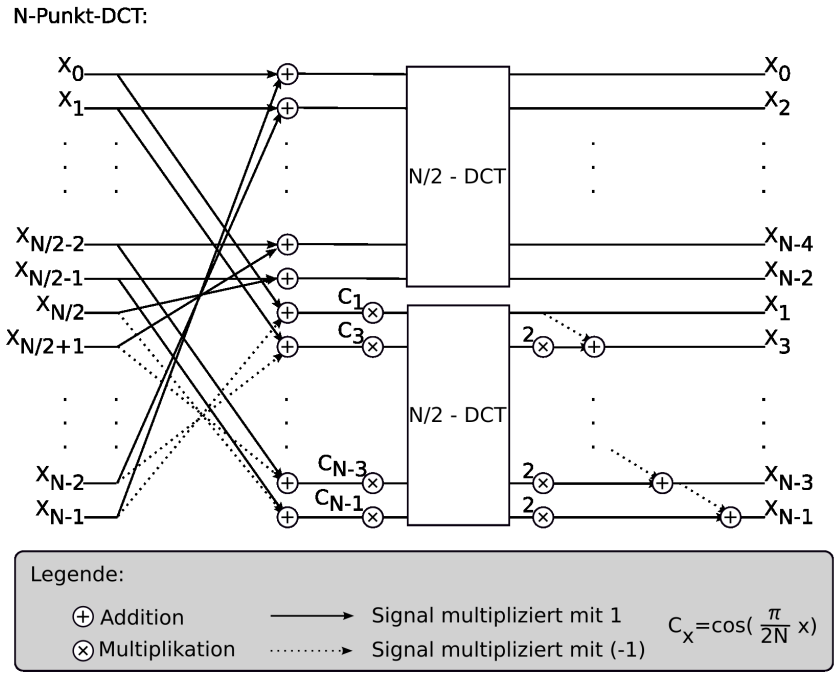

Abbildung 3.3: Berechnung einer N-Punkt-DCT durch zwei N/2-Punkt-DCT

Die DCT, die im Rahmen dieser Arbeit verwendet wird, ist für ein früheres Projekt[Grü04] entwickelt worden und basiert auf einer Hardwareumsetzung des rekursiven Algorithmus von Hou[Hou87] bzw. Lee [Lee84]. Abbildung 3.3 stellt den rekursiven Ablauf der Berechnungen dar. Der Algorithmus berechnet die N-Punkt-DCT durch die Berechnung zweier $\frac{N}{2}$-Punkt-DCTs und einige zusätzliche, so genannte Butterfly-Berechnungen. Im Vergleich zu einer naiven Implementierung der diskreten Kosi-

nustransformation mit einem Aufwand in der Größenordnung von $O(N^2)$ gelingt es, durch geschicktes Umordnen der Operationen, den Aufwand für die Berechnung auf $O(N \cdot logN)$ zu reduzieren.

Für diese bestehende Komponente wird ein OPB-Interface entwickelt, um die Hardware-DCT über den OPB-Bus mit dem Microblaze verbinden zu können.

Eine weitere Möglichkeit wäre einen vorgefertigten IP-Core zu verwenden. Jedoch sind die meisten verfügbaren DCT-IP-Cores für eine Verwendung innerhalb eines JPEG[18]-Enkoders optimiert und implementieren eine zweidimensionale 8x8-Punkt-DCT. Nur wenige IP-Cores implementieren eine eindimensionale 32-Punkt-DCT wie sie im Rahmen der Subbandsynthese verwendet wird. Xilinx stellt einen parametrisierbaren IP-DCT-Core zur Verfügung, der verwendet werden kann, um eine 32-Punkt-DCT zu implementieren, jedoch ist der Flächenverbrauch mit 9074 Slices zu hoch für eine Integration in das System, da das verwendete FPGA nur 10.752 Slices bietet und somit allein die DCT die verfügbaren Ressourcen zu circa 84 % auslasten würde.

3.3 Software

Neben dem Entwurf der Hardware ist es nötig, die Softwarekomponenten des Systems zu entwerfen. In diesem Abschnitt sollen die einzelnen Softwarekomponenten beschrieben werden.

3.3.1 MPEG-Audio-Dekoder

Als Anwendungsfall, der als Grundlage für die durchzuführenden Messungen dienen soll, wird ein MPEG-Audio-Dekoder verwendet. Es wird keine eigene Implementierung eines Dekoders angefertigt, sondern eine bestehende Open-Source-Implementierung verwendet. Die verwendete MAD[19]-Bibliothek [Und] wurde von der Firma Underbit in C entwickelt und bietet eine, sowohl im Hinblick auf Genauigkeit, als auch Performance stark optimierte Implementierung eines MPEG-Audio-Dekoders. Für die Dekodierung wird statt Fließkommazahlen eine Festkomma-Arithmetik verwendet. Dadurch ist sie besonders für Implementierungen auf Systemen mit begrenzter Rechenleistung geeignet.

Der modulare Aufbau der Bibliothek vereinfacht es, eine MPEG-Audio-Dekoder-Applikation zu entwickeln. Die Bibliothek bietet eine *Decoder*-Funktion, der verschiedene Callback-Funktionen als Parameter übergeben werden können. Diese Callback-

[18] Joint Photographic Experts Group
[19] MPEG Audio Decoder

Funktionen werden aufgerufen, um den Eingabepuffer zu füllen, einen Frame auszugeben oder die Fehlerbehandlung durchzuführen. Bei der Implementierung eines einfachen Dekoders reicht es aus, diese drei Callback-Funktionen zu implementieren und den Dekoder im Hauptprogramm zu initialisieren. Der gesamte Vorgang der Dekodierung wird dann von der Decoder-Funktion gesteuert. Für die im Rahmen dieser Arbeit verwendete Anwendung ist es ausreichend, die mit der Bibliothek mitgelieferte Beispielanwendung entsprechend zu modifizieren.

Die MAD-Bibliothek verwendet für die Subbandsynthese ebenfalls den rekursiven DCT-Algorithmus von Lee, jedoch in einer abgewickelten Variante. Die Subbandsynthese und die DCT sind innerhalb der Bibliothek gekapselt, dadurch ist es möglich, sie ohne größere Änderungen der Bibliothek in Hardware auszulagern.

Die Bibliothek verwendet für die interne Darstellung der Zahlen ein vorzeichenbehaftetes Festkomma-Format mit vier Bit für die Darstellung des Ganzzahlanteils und 28 Bit für die Darstellung der Nachkommastellen. Für die Darstellung negativer Zahlen wird die Zweierkomplement-Darstellung verwendet. Aufgrund der Verwendung von 28 binären Nachkommastellen beträgt die Wertigkeit des niederwertigsten Bit und damit die maximale Auflösung $2^{-28} \approx 3,725 \cdot 10^{-9}$. Der Wertebereich beträgt also: $80000000_{16} = -8_{10}$ bis $7FFFFFFF_{16} = +7,99999999627_{10}$.

Für die Implementierung auf dem Microblaze ist die Verwendung einer 32-Bit-Festkommaarithmetik gut geeignet, da die Register des Prozessors ebenfalls eine Breite von 32 Bit aufweisen.

3.3.2 Dateisystem

Da die zu dekodierenden Daten von der CompactFlash-Karte gelesen werden sollen, ist eine Unterstützung des verwendeten Dateisystems nötig. Die von Xilinx bereitgestellten Treiber für die SystemACE-Komponente bieten neben dem Low-Level-Routinen für den Zugriff auf die CompactFlash-Karte auch Funktionen für den Zugriff auf Dateiebene. Als Dateisystem für die CompactFlash-Karte wird FAT16[20] verwendet.

3.3.3 Audio-Ausgabe

Für die Ausgabe der dekodierten Audiodaten ist eine Ansteuerung des verwendeten AC97-Controllers nötig. Da der Controller einem Design-Beispiel, welches zum Lieferumfang des Development Boards gehörte, entnommen ist, sind auch die zur Ansteuerung verwendeten Software-Routinen an die in dem Beispiel verwendete Software angelehnt. Der Treiber stellt einen Puffer zur Verfügung, der durch den MPEG-Audio-Dekoder

[20] File Allocation Table

frameweise gefüllt wird. Die Ausgabe der Samples aus dem Puffer an den Controller erfolgt interruptgesteuert. Per Interrupt signalisiert der Controller dem Microblaze die Bereitschaft, weitere Daten zu empfangen. Der Microblaze führt die entsprechende Interrupt Service Routine aus und übergibt Werte an den Controller, solange dieser in der Lage ist, Daten zu empfangen. Der Controller übergibt die empfangenen Daten an den Codec, der diese dann in analoge Signale umwandelt.

3.4 HW/SW Kommunikation

Bei dem Design mit einer in Hardware ausgelagerten DCT ist es nötig, ein Kommunikationsprotokoll zu entwerfen, welches für den Datentransfer zwischen dem Microblaze und der DCT verwendet wird. Das verwendete Protokoll kann, da die DCT nicht konfigurierbar ist, einfach gestaltet sein. Es ist lediglich nötig, die Parameter an die DCT zu übertragen, die DCT zu starten und nach Abschluss der Berechnungen die Rückgabewerte zurückzulesen.

3.4.1 Implementierung eines einfachen Protokolls zur HW/SW-Kommunikation

Eine Möglichkeit für die Kommunikation ist, die internen Register für die Eingabeparameter linear in den Speicherbereich des Microblaze einzublenden, dieser schreibt die Eingabeparameter in einen fortlaufenden Speicherbereich und füllt so die Register. Die DCT wird durch Setzen eines Bits in einem Kontrollregister gestartet. Der Microblaze überprüft durch das Lesen eines Statusregisters, ob die Berechnung beendet ist und wartet gegebenenfalls. Nach Abschluss der Berechnung liest der Microblaze die ebenfalls in den Speicherbereich eingeblendeten Register mit den Rückgabewerten. Abbildung 3.4 zeigt eine grafische Darstellung der Speicherbereiche der DCT-Komponente.

Ein anderes Vorgehen wäre ebenfalls möglich: Der Microblaze könnte der DCT-Komponente zwei Speicheradressen mitteilen, zuerst die Adresse, ab der die Eingabeparameter gelesen werden sollen, danach die Adresse, ab der die Rückgabewerte geschrieben werden sollen. Über das Status- und Kontrollregister wird wiederum die Ausführung der DCT gestartet und der Zustand der DCT abgefragt. Diese Möglichkeit soll jedoch nicht weiter verfolgt werden, da die Komponente hierfür ein OPB-Master-Interface benötigen würde. Dies ist jedoch nicht gewünscht, da es zum einen die Komplexität der Komponente erhöhen würde und zum anderen zu konkurrierenden Zugriffen auf den OPB-Bus führen würde. Ist der Microblaze der einzige Master am OPB-Bus, so kann er den Bus belegen, ohne dass es zu Wartezyklen kommt, da der Bus durch andere Master belegt ist.

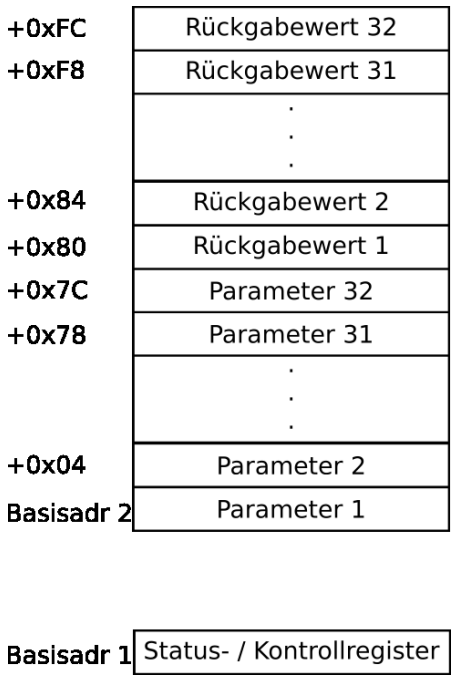

Abbildung 3.4: Speichermapping der DCT-Komponente

3.4.2 RMI - Protokoll

Im Folgenden soll der Ablauf der Kommunikation mit Hilfe des OSSS-RMI-Protokolls erläutert werden.

Abbildung 3.5 zeigt den Ablauf des RMI-Protokolls. Das Protokoll wird durch den entsprechenden Methodenaufruf gestartet. Zunächst übermittelt der Klient dem Shared Object die *MethodID* der aufgerufenen Methode. Daraufhin fragt der Klient die Guard-Bedingung ab. Sobald das Shared Object dem Klienten signalisiert, dass die Guard-Bedingung wahr ist, überträgt der Klient die Eingabeparameter der aufgerufenen Funktion oder Prozedur. Nach Abschluss der Übertragung der Parameter fragt der Klient so lange den Zustand des Shared Object an, bis dieses den Abschluss der aufgerufenen Funktion signalisiert. Falls eine Funktion des Shared Objects aufgerufen wurde und somit Rückgabewerte existieren, liest der Klient darauf folgend diese Rückgabewerte. Um den gesamten Transfer abzuschließen, überträgt der Klient noch einmal die *MethodID*.

Die Übermittlung der Parameter und Rückgabewerte erfolgt in serialisierter Form. Das heißt die Objekte, die als Parameter oder als Rückgabewert übermittelt werden, müssen in geeigneter Weise in einzelne Werte aufgeteilt werden, die über den Kommunikationskanal zwischen dem Klienten und dem Shared Object übertragen werden. Diese Serialisierung erfolgt automatisiert durch die OSSS Bibliothek. Daher ist es not-

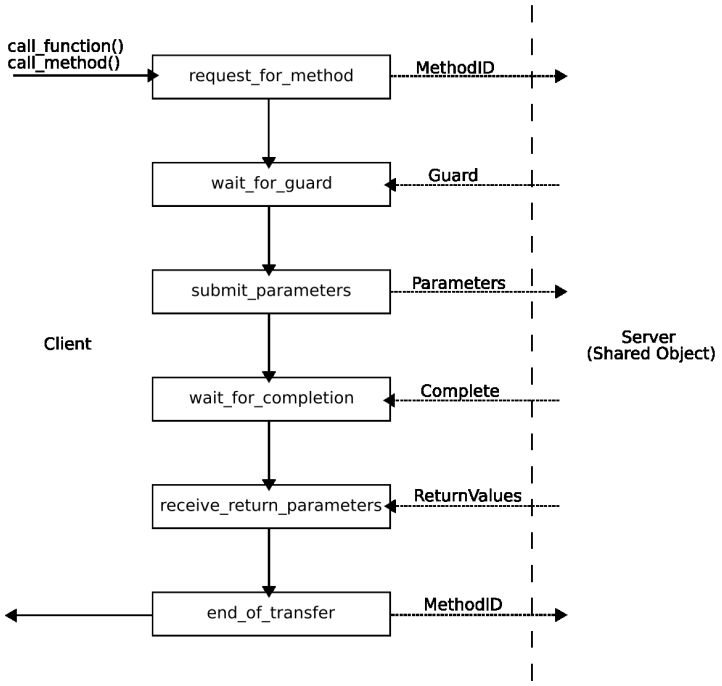

Abbildung 3.5: Ablauf des RMI-Protokolls

wendig, dass alle Parameter und Rückgabewerte von einem Typ sind, der von der Klasse `osss_serialisable_object` abgeleitet ist, und die Funktionen `serialise()` und `deserialise()` implementieren. Diese Funktionen werden aufgerufen, um die Serialisierung während des Sendens, bzw. die Deserialisierung beim Empfang, durchzuführen.

3.5 Testablauf

Das entworfene Testsystem soll dazu verwendet werden, verschiedene Messreihen durchzuführen. In diesem Abschnitt soll daher näher auf die verwendeten Testdaten sowie die Verfahren zur Messung eingegangen werden.

3.5.1 Testdaten

Um Messungen am Testsystem durchzuführen, ist es nötig, den Dekoder einen geeigneten Testdatenstrom dekodieren zu lassen. Im Rahmen der Tests werden verschiedene Datenströme als Testdaten für die Messungen verwendet. Zunächst werden verschiedene

Testbitströme verwendet, die der ISO-Standard für die Überprüfung der Einhaltung des Standards definiert[MPE95]. Diese ersten Tests dienen hauptsächlich der Überprüfung der Funktionalität des Dekoders.

Für die weiteren Tests werden verschiedene komprimierte Audio-Titel als Eingabedaten verwendet. Diese werden für die Tests mit Hilfe eines MPEG-Audio-Enkoders (lame 3.97 [LAM]) in verschiedenen Bitraten, von 32 kb/s bis 320 kb/s, erzeugt. Hierbei ist zu beachten, dass die Gesamtperformance des Systems zwar von der gewählten Bit- und Samplingrate abhängt, die Bitrate jedoch für die Messung der Performance der DCT unerheblich ist. Die Ursache dafür liegt darin begründet, dass die DCT unabhängig von der gewählten Bitrate pro Frame und Kanal jeweils 1.152 Werte verarbeitet.

Die Samplingrate hat im Gegensatz dazu insofern Einfluss auf das Messergebnis, als die Aufruffrequenz der DCT von der gewählten Samplingrate abhängt. Die von der DCT pro Frame erzeugten 1.152 Ausgabewerte ergeben bei einer Samplingrate von 44,1 kHz Audiodaten für circa 26,12 ms (etwa 38,28 Frames/s). Bei einer Samplingrate von 48 kHz hingegen ergibt die gleiche Anzahl Ausgabesamples eine Wiedergabedauer von 24 ms (etwa 41,67 Frames/s).

Für die Messung der Dauer der einzelnen DCT-Aufrufe ist die gewählte Samplingrate jedoch ebenfalls unerheblich, da sie nur die Frequenz beeinflusst, mit der die DCT-Funktion aufgerufen wird, die für die Berechnung der DCT benötigte Zeit und die Anzahl pro Aufruf transferierter Werte bleibt jedoch gleich.

Daher kann für die Testdatenströme die Samplingrate frei gewählt werden, ohne dass sich das Ergebnis der Messung des Kommunikationsoverheads für einen einzelnen DCT-Aufruf oder einen Frame verändern würde. Daher wird für die selbst erzeugten Testdatenströme eine Abtastrate von 44,1 kHz gewählt, da dies der bei der CD[21] verwendeten Abtastrate entspricht.

3.5.2 Profiling

In einem ersten Test wird die Funktion des verwendeten Open Source MPEG-Audio-Dekoders überprüft. Hierzu wird nicht das tatsächliche Testsystem verwendet, sondern die vom Xilinx EDK erstellte so genannte *Virtual Platform*. Bei der *Virtual Platform* handelt es sich um ein automatisch generiertes ausführbares Modell des Systems, das eine virtuelle Version der modellierten Hardware darstellt. Auf dieser virtuellen Hardware kann die zu dem System gehörige Software getestet werden. Hierzu bietet die *Virtual Platform* die Funktion eines *Instruction Set Simulators*, das heißt die virtuelle und die spätere physische Hardware sind zyklusäquivalent. Die für die Ausführung eines Programms benötigte Zeit ist zwar um Größenordnungen höher, als die auf der tatsäch-

[21] Compact Disc

lichen Hardware benötigte Zeit, da jedoch die Simulation zyklusäquivalent abläuft, ist es möglich, anhand der Zyklen, die die Ausführung benötigt, auf die Performance der späteren Hardware zu schließen. Auf diese Weise ist es schon zu einem relativ frühen Stadium des Designvorganges möglich, die Performance des Gesamtsystems abzuschätzen.

Bei den Tests, die mit Hilfe der *Virtual Platform* durchgeführt werden, werden die *compliance testing bitstreams*[MPE95], die in ISO 11172-part 4 definiert werden, als Eingabedaten verwendet. Dieser erste Test dient einerseits der Überprüfung der Funktionalität, andererseits dazu, die Designentscheidung hinsichtlich der Auslagerung der DCT in Hardware zu überprüfen. Dabei wird von den später verwendeten Mechanismen zur Ein- und Ausgabe der Daten abstrahiert. Konkret liegen die Eingabedaten als im Sourcecode des Programms fest kodierte Felder vor und die Ausgabedaten werden nach der Dekodierung verworfen. Somit wird der Aufwand, der sich aus der Daten-Ein- und Ausgabe ergibt, für die erste Abschätzung der Performance vernachlässigt.

Die *Virtual Platform* bietet verschiedene statistische Funktionen, um die Leistungsfähigkeit eines zu testenden Systems zu messen. Listing 3.1 zeigt die Ausgabe der statistischen Daten durch die *Virtual Platform*. Die Daten wurden durch die Dekodierung von 29 Frames mit einer Bitrate von 32 kb/s und einer Samplingrate von 44,1 kHz gewonnen. Den Daten ist zu entnehmen, dass die Dekodierung insgesamt 22.382.061 Taktzyklen benötigt, dies ergibt bei einer Frequenz von 100 MHz eine Zeit von circa 223,82 ms. Die für eine Dekodierung in Echtzeit zur Verfügung stehende Zeit beträgt für diese 29 Frames 757,55 ms. Es werden also nur 29,55 % der maximal zur Verfügung stehenden Zeit für die Dekodierung benötigt.

Neben den Daten zur Laufzeit werden noch weitere statistische Daten ausgegeben, so zum Beispiel die Anzahl ausgeführter Instruktionen, die Anzahl Multiplikationen und Divisionen oder auch die Anzahl an Cache-Treffern. Aus diesen Daten lässt sich beispielsweise erkennen, dass die Verwendung eines HW-Multiplizierers sinnvoll ist, da die Anzahl der Multiplikationsoperationen während der Dekodierung hoch ist.

Das Profiling liefert für die einzelnen Funktionen des ausgeführten Programms eine detaillierte Aufrufstatistik, welche sowohl die Anzahl der Aufrufe, als auch deren Dauer beinhaltet. Listing 3.2 zeigt das Ergebnis des Profilings für den Bitstrom, der auch für die Daten aus Listing 3.1 verwendet wurde. Wie zu erkennen ist, liefert das Profiling detaillierte Daten für die Ausführungszeit der einzelnen Funktionen. Den Daten ist zu entnehmen, dass 54,15 % der gesamten Laufzeit von der `dct32`-Funktion benötigt werden. Daher scheint es sinnvoll für eine Verbesserung der Performance des MPEG-Audio-Dekoders, die DCT-Funktion als dedizierte Hardware auszulagern.

Diese Art der Performanceanalyse bietet den Vorteil, dass das Zeitverhalten der Software, im Gegensatz zu einem Profiling auf der tatsächlichen Hardware, durch die Messung nicht beeinflusst wird. Die Erhebung der statistischen Daten erfolgt durch die Simulationsumgebung, daher ändert sich das Zyklusverhalten der Software nicht.

```
---------Simulation Statistics---------
               Cycles : 22382061
         Instructions : 7033278
                  CPI : 3.182
                Loads : 2070903
               Stores : 748313
      Multiplications : 749741
            Divisions : 470
        Barrel shifts : 482772
  Total # of branches : 355537
  # of taken branches : 269015 (75.664%)
                Calls : 17771
              Returns : 17771
   FP Adds (disabled) : 0
   FP Subs (disabled) : 0
   FP Muls (disabled) : 0
   FP Divs (disabled) : 0
   FP Cmps (disabled) : 0
      ICache Accesses : 7363343
          ICache Hits : 7356486 (99.907%)
      DCache Accesses : 2070332
          DCache Hits : 1524544 (73.638%)

----------Pipeline Stall Info-----------
    Total # of stalls : 17344977
        IFetch stalls : 340728 (1.964%)
 Memory Access stalls : 14993995 (86.446%)
    Multiplier stalls : 1499482 (8.645%)
Barrel Shifter stalls : 495732 (2.858%)
       Divider stalls : 15040 (0.087%)
           FPU stalls : 0 (0.000%)

--------------Branch Info---------------
Branches w/ delay slot  : 207268 (1.195%)
Branches w/o delay slot : 123494 (0.712%)
```

Listing 3.1: Ausgabe der statistischen Daten der *Virtual Platform*

```
Each sample counts as 1e-08 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 54.15      0.11     0.11     2088    54.02    54.02  dct32
 18.50      0.15     0.04       29  1329.01  5218.27  synth_full
  7.90      0.17     0.02      604    27.24    33.43  III_imdct_l
  7.34      0.18     0.02       29   527.46  1858.80  mad_layer_III
  3.82      0.19     0.01      108    73.71    73.71  III_aliasreduce
  1.79      0.19     0.00     1208     3.09     3.09  fastsdct
  1.23      0.20     0.00     1856     1.38     1.38  III_freqinver
  1.17      0.20     0.00     4743     0.51     0.51  mad_bit_read
  1.05      0.20     0.00      666     3.27     3.27  III_overlap
  0.72      0.20     0.00       62    24.31    24.31  III_imdct_s
  [...]
```

Listing 3.2: Ausschnitt aus der Ausgabe des Profilings

Im Gegensatz zu einem Profiling durch die *Virtual Platform*, verändert das Profiling auf der Hardware das beobachtete Verhalten. Einerseits ist es notwendig, zusätzliche Hardware, nämlich einen Timer, vorzusehen, der für die Messungen der Ausführungszeiten verwendet wird. Andererseits wird durch die Messung das Zeitverhalten der Software beeinflusst. Der Grad der Beeinflussung des Systems variiert mit der gewählten zeitlichen Auflösung des Profilings. Wird eine relativ grobe Auflösung verwendet, so ist die Zuordnung der verbrauchten Rechenzeit zu den einzelnen Funktionen ungenau, da Funktionen mit kurzen Ausführungszeiten dabei unter Umständen kein Rechenzeitverbrauch zugeordnet wird. Der Einfluss auf die Systemperformance ist dabei jedoch gering, da der durch die Messung verursachte Overhead klein ist. Wird eine feinere zeitliche Auflösung gewählt, so ist die Zuordnung der Zeiten genauer, jedoch verschlechtert sich die Gesamtperformance des Systems deutlich.

Diese Verschlechterung der Performance ist für das betrachtete System vor allem problematisch, da der Ablauf der Dekodierung maßgeblich von der Ausgabe der Audiodaten abhängt. Diese wird durch externe Komponenten getaktet, die von der Performanceverschlechterung durch das Profiling jedoch nicht betroffen sind. Daher kann das Zeitverhalten der Software bei aktiviertem Profiling auf der Hardware nicht realistisch bestimmt werden.

3.5.3 Messung der Systemperformance

Um trotz der Probleme mit dem Profiling Messungen durchführen zu können, ist es notwendig, ein anderes Messverfahren zu entwickeln. Da zunächst nur die gesamte Systemperformance gemessen werden soll, ist es nicht nötig, die detaillierten Informationen zu

gewinnen, die das Profiling liefern würde. Es sind mehrere verschiedene Varianten der Messung denkbar. Zwei dieser Varianten sollen im Folgenden näher betrachtet werden.

3.5.3.1 Messung ohne Timer

Während der Dekodierung eines Datenstroms kann es vorkommen, dass der zur Ausgabe der dekodierten Daten verwendete Puffer vollständig gefüllt ist. In diesem Fall kann der dekodierte Frame nicht in den Puffer geschrieben werden und der Dekoder muss warten, bis ein weiterer Frame komplett wiedergegeben wurde und der Puffer somit Platz für einen weiteren Frame bietet. Dazu überprüft der Dekoder in einer Schleife, ob der Puffer freien Platz für einen weiteren Frame enthält. Innerhalb dieser Schleife kann, ohne signifikante Änderung des Zeitverhaltens des Gesamtsystems, ein Zähler inkrementiert werden. Aus dem Stand des Zählers beim Verlassen der Schleife kann auf die in der Schleife verbrachte Zeit geschlossen werden.

Aus der in der Schleife verbrachten Zeit können Rückschlüsse auf die Auslastung des Gesamtsystems geschlossen werden. Dies ist möglich, da die für die Dekodierung des Frames benötigte Zeit gleich der gesamten für die Dekodierung zur Verfügung stehenden Zeit, abzüglich der in der Schleife verbrachten Zeit, ist. Ergibt die Messung beispielsweise, dass bei der Dekodierung eines Frames mit einer Abtastrate von 44,1 kHz, 2 ms innerhalb der Schleife verbracht wurden, so ergibt sich:

$$t_{frame} = \frac{1.152\ samples}{44.100\ samples/s} = 0,02612\ s$$

$$t_{loop} = 2\ ms = 0,002\ s$$

$$t_{used} = t_{frame} - t_{loop} = 0,02612\ ms - 0,002\ ms = 0,02412\ ms$$

$$=> Auslastung[\%] = \frac{t_{used}}{t_{frame}} \cdot 100 = \frac{0,02412}{0,02612} \cdot 100 \approx 92\ \%$$

Um von dem Stand des Zählers auf die verbrauchte Zeit schließen zu können, ist es notwendig, die Anzahl und Art der während eines Schleifendurchlaufs ausgeführten Instruktionen zu bestimmen und deren Ausführungszeiten zu addieren. Das folgende Fragment aus dem Assemblercode (Siehe Listing 3.3) enthält den zur Zählung der Schleifendurchläufe verwendeten Code.

Mit Hilfe der im *Microblaze Reference Manual* angegebenen Ausführungszeiten für die einzelnen Instruktionen kann somit die Laufzeit eines Schleifendurchlaufs bestimmt werden. Da die Schleife häufig durchlaufen wird, kann die Zeit für die Initialisierung der Zählvariablen sowie für die Speicherung des Zählerstands vernachlässigt werden. Innerhalb der Schleife werden zwei *load*-Operationen, eine Addition, ein Vergleich und eine *branch*-Instruktion ausgeführt. Die im *Microblaze Reference Manual*[Xilb] angegebenen Zeiten für diese Instruktionen betragen jeweils drei Takte für den bedingten Sprung und

```
      [...]
      addk r6,r0,r0          //initialisierung der Zählvariablen
$L44  lwi r4,r0,output_pos   //laden der derzeitigen Ausgabeposition
      lwi r3,r0,output_end   //laden der Endposition des Ausgabpuffers
      addik r6,r6,1          //inkrementieren des Schleifenzählers
      cmpu r18,r3,r4         //derzeitige Position = Ende?
      blti r18, $L44         //wenn ja, springe zu $L44
      [...]
```

Listing 3.3: Assemblercode der Zählschleife

jeweils einen Takt für die übrigen Instruktionen. Somit ergibt sich eine Ausführungszeit von acht Takten pro Schleifendurchlauf.

Bei diesem Verfahren werden jedoch die Effekte, die sich durch die Latenzzeiten bei Speicherzugriffen und durch die im System vorhandenen Caches ergeben, vernachlässigt. Somit wird die tatsächliche pro Schleifendurchlauf benötigte Zeit höher liegen, die Aussagekraft des ermittelten Wertes für die innerhalb der Schleife verbrachte Zeit ist also eingeschränkt. Der Wert kann jedoch verwendet werden, um die Auslastung des Systems nach oben abzuschätzen. Dies ist möglich, da für jeden Schleifendurchlauf mindestens acht Takte benötigt werden und somit, durch die Multiplikation der Anzahl der Durchläufe mit acht, die minimale Zeit bestimmt werden kann, die innerhalb der Schleife verbracht wurde.

3.5.3.2 Messung mit Timer

Ein Nachteil des oben vorgestellten Verfahrens zur Messung der Systemperformance ohne zusätzliche Hardware ist die Ungenauigkeit. Diese ergibt sich aus dem Verhalten der Caches und der externen Speicher, welches es schwierig macht, die Ausführungszeit eines Schleifendurchlaufs anhand der ausgeführten Instruktionen zu bestimmen. Daher soll im Folgenden ein Verfahren vorgestellt werden, das für die Messung einen Hardware-Timer verwendet und so die Genauigkeit der Messung erhöht.

Der verwendete Ansatz ist dem beim Profiling verwendeten ähnlich. Der Timer wird jedoch nicht dazu verwendet, periodisch die Ausführung der Software zu unterbrechen. Stattdessen wird ein Betriebsmodus für den Timer verwendet, in dem der Stand des Zählers mit jedem Takt inkrementiert wird. Bei diesem Verfahren wird die verbrauchte Rechenzeit direkt und nicht über den Umweg der freien CPU-Zeit gemessen. Die Messung wird bei Beginn der Dekodierung eines Frames gestartet, hierzu wird der Zählerstand auf null gesetzt und der Zähler gestartet. Der Zähler zählt die für die Dekodierung sowie die Ausgabe des Frames benötigten Takte, wobei die Zeit, die auf das Freiwerden eines Pufferplatzes gewartet wird, nicht mitgezählt wird. Während dieses Wartens wird der Zähler angehalten. Dieses Verfahren ermöglicht die Bestimmung der Gesamtperfor-

mance des Systems mit höherer Genauigkeit, als das vorherige Verfahren, ohne dabei jedoch die Leistungsfähigkeit in der selben Weise zu verschlechtern, wie das Profiling, das von der Xilinx-Laufzeitbibliothek angeboten wird.

3.5.4 Messung des Overheads

Für die eigentliche Analyse des Kommunikationsoverheads, die das Hauptziel dieser Arbeit darstellt, wird ein Messverfahren verwendet, das dem Verfahren zur Messung der Gesamtperformance mit Hilfe eines Timers ähnelt. Jedoch wird der Zähler nicht verwendet, um die für die gesamte Dekodierung eines Frames benötigte Zeit zu bestimmen, sondern lediglich, um die für die Ausführung eines DCT-Methodenaufrufs benötigten Takte zu zählen. Da die für die eigentliche Berechnung der DCT benötigten Takte bekannt sind, durch Simulation des VHDL-Modells, können Rückschlüsse auf die für die Kommunikation benötigten Zeiten gezogen werden. Die Messung wird vor dem Aufruf der DCT-Funktion gestartet und nach deren Abschluss wieder angehalten. Aus dem Zählerstand lässt sich somit direkt die Anzahl vergangener Takte und somit auch die vergangene Zeit bestimmen.

4 Implementierung

Im folgenden Kapitel soll näher auf die Details der Implementierung sowohl der Hardware, als auch der Software des Testsystems eingegangen werden. Die Entwicklung der Soft- und Hardware geschieht iterativ in mehreren Phasen.

4.1 Phase I - Entwicklung des Kernsystems und Virtual Platform

In einem ersten Schritt soll zunächst die reine Softwareimplementierung des MPEG-Audio-Dekoders erstellt werden. Diese dient als Basis für die weiteren Implementierungen.

4.1.1 Implementierung der Hardware

Da die Testhardware für alle Implementierungen, bis auf geringe Abweichungen, gleich ist, geschieht ein Großteil der nötigen Hardwareimplementierung in diesem ersten Schritt. Die Implementierung des Testsystems erfolgt dabei unter Verwendung des Xilinx Embedded Development Kits. Dieses ermöglicht es, ein System aus vordefinierten Komponenten (zum Beispiel Prozessoren, Speichercontroller etc.) mittels einer grafischen Bedienoberfläche zu erstellen. Hierzu wird zunächst ein neues Projekt mit Hilfe des *Base System Builders* erstellt. Der System Builder bietet nach Auswahl des verwendeten Development-Boards, die Möglichkeit, die verschiedenen auf dem Board vorhandenen Peripheriekomponenten dem System hinzuzufügen. Die Hardwarebeschreibung des so erstellten Systems wird in Form einer MHS[1]-Datei gespeichert, die eine strukturelle Beschreibung der Systemkomponenten beinhaltet. Zusätzlich wird ein so genanntes UCF[2] erzeugt. Diese Datei enthält eine textuelle Beschreibung von Nebenbedingungen, so genannter *Constraints*. Diese beinhalten einerseits die Zuordnung der Ein- und Ausgangssignale des Designs zu Gehäusepins des FPGAs sowie andererseits weitere Informationen, wie zum Beispiel Taktfrequenzen oder maximale Verzögerungszeiten.

[1] Microprocessor Hardware Specification
[2] User Constraints File

Der durch den *Base System Builder* erstellten Systembeschreibung können noch zusätzliche Hardwarekomponenten mit Hilfe der grafischen Benutzeroberfläche des XPS[3][Xil06a] hinzugefügt werden. Hierbei ist jedoch zu beachten, dass eventuell das *User Constraint File* manuell angepasst werden muss.

In einer initialen Version wurde ein Kernsystem bestehend aus dem Microblaze, der Anbindung der verschiedenen Speicher, sowie der seriellen Schnittstelle implementiert. Mit Hilfe des EDK wird aus dieser ersten Version eine *Virtual Platform* erzeugt, die für die Durchführung erster Messungen verwendet werden soll. Im Verlauf dieser ersten Tests wurde von Xilinx aktualisierte Versionen des Microblaze (Version 5.00.c) und anderer Peripheriekomponenten veröffentlicht. Diese neue Version beinhaltet einige Modifikationen des Prozessordesigns, die sich deutlich auf die Systemperformance auswirkten. Eine Änderung betraf die Länge der Pipeline des Prozessors. Diese wurde von drei auf fünf Stufen verlängert. Dadurch wurde eine Verringerung des CPI[4]-Wertes für komplexe arithmetische Operationen, wie zum Beispiel der Multiplikation, erreicht. Die für die Multiplikation benötigte Anzahl Takte sank von drei auf einen, dies entspricht somit einer Beschleunigung um den Faktor drei.

Da für die Dekodierung eines MPEG-Audio-Frames zahlreiche Multiplikationen notwendig sind, ergaben die ersten Tests, dass durch diese Performancesteigerung eine Echtzeitdekodierung durch eine reine Softwareimplementierung möglich ist.

Für die Verwendung der Virtual Platform entstehen, bedingt durch den ebenfalls aktualisierten SDRAM-Speichercontroller, jedoch zusätzliche Schwierigkeiten, da dieser in der aktuellen Version von der Virtual Platform nicht unterstützt wird. Dieses Problem wurde umgangen, indem der Speichercontroller wieder durch die Vorgängerversion ersetzt wurde. Hierzu ist jedoch eine manuelle Anpassung der MHS-Datei notwendig.

4.1.2 Implementierung der Software

Für eine erste Implementierung des Dekoders wurde die zur MAD-Bibliothek [Und] gehörende Player-Applikation modifiziert. Die Bibliothek bietet verschiedene plattformspezifische Optimierungen für verschiedene Prozessorfamilien. Für den im Rahmen dieser Arbeit eingesetzten Microblaze existierten jedoch keine speziellen Anpassungen. Da der Microblaze über keinerlei Instruktionen zum korrekten Multiplizieren von Zahlen in Festkommadarstellung bietet, wurden die generischen Multiplikationsroutinen verwendet.

Bei der Multiplikation zweier 32 Bit breiten Festkommazahlen ist zu beachten, dass das Ergebnis eine Bitbreite aufweist, die der addierten Bitbreite beider Operanden, also 64

[3] Xilinx Platform Studio
[4] Cycles per instruction

Bit, entspricht. Davon enthalten 8 Bit den Vorkomma- und 56 Bit den Nachkommaanteil. Um das Ergebnis wieder in die 32-Bit-Darstellung umzuwandeln, ist es nötig, das Ergebnis der Multiplikation um 28 Stellen nach rechts zu verschieben. Die niederwertigsten 32 Bit enthalten dann das Ergebnis in der gewünschten Darstellung.

Dieses Vorgehen ist für eine Verwendung auf dem Microblaze nicht möglich, da dieser als Ergebnis einer Multiplikation zweier 32-Bit-Zahlen nur die niederwertigsten 32 Bit des Ergebnisses zurückliefert. Somit ist es notwendig, die Operanden vor der Multiplikation passend zu skalieren. Dies ist jedoch mit einem Genauigkeitsverlust verbunden. Die in der Bibliothek verwendete Routine skaliert die beiden Operanden nicht gleichmäßig. Der erste Operand wird um 12 Bit, der Zweite um 16 Bit nach rechts verschoben, der Genauigkeitsverlust ist für den zweiten Operanden also höher als für den ersten. Aus den beiden Verschiebungen ergibt sich die insgesamt benötigte Verschiebung um 28 Bit.

Die für die ersten Tests mit Hilfe der Virtual Platform verwendeten MPEG-Datenströme wurden als Array im Sourcecode hart kodiert. Es wurden jeweils zehn Frames bei verschiedenen Bitraten getestet, um eine erste Abschätzung der Performance des Dekoders zu erhalten. Da hierbei sowohl von der Ein- als auch von der Ausgabe abstrahiert wurde, kann das Ergebnis jedoch nur als grobe Abschätzung dienen.

4.2 Phase II - Erstellen der Software-Implementierung

In einem zweiten Schritt wird der Entwurf aus dem ersten Schritt erweitert und die Software-Version des MPEG-Audio-Dekoders erstellt. Dies entspricht der ersten in Abschnitt 3.2.1 aufgezählten Implementierungen. Hierzu sind verschiedene Anpassungen sowohl an der Hard-, als auch an der Software notwendig.

4.2.1 Hardware

Das im vorherigen Schritt erstellte Hardwaresystem wird nun um Komponenten zur Ein- und Ausgabe erweitert. Für die Eingabe wird der SystemACE-Controller aus der Xilinx-Komponentenbibliothek dem System hinzugefügt, dieser bindet über einen externen Baustein die CompactFlash-Karte an und ermöglicht einen Zugriff auf die dort gespeicherten Dateien.

Für die Ausgabe der dekodierten Daten wird dem System der *AC97 Digital Controller* hinzugefügt, dieser ermöglicht die Ausgabe der Audiodaten über den *AC97 Codec*. Für den AC97-Controller existieren keine Komponenten in der Xilinx-Bibliothek, jedoch verwendet eines der zum Lieferumfang des Development-Boards gehörigen Designbeispiele den *AC97 Codec* zur Wiedergabe von Audiodaten. Die dort verwendete Implementierung des Controllers wird in leicht abgewandelter Form für die Wiedergabe der dekodierten

Audiodaten verwendet. Der Controller wird so konfiguriert, dass ein Interrupt-Signal generiert wird, sobald der für die Wiedergabe verwendete FIFO-Puffer weniger als halb gefüllt ist. Die Länge der innerhalb des Controllers verwendeten FIFO-Puffer wird von acht Einträgen auf sechzehn Einträge verlängert, um so die Frequenz, mit der die Unterbrechungen generiert werden, zu verringern. Da die Aufnahmefunktionalität des Codecs nicht verwendet wird, wird auch der Aufnahmeteil des Controllers deaktiviert, um so den Flächenverbrauch zu reduzieren.

Als weitere Komponente wird dem System ein Timer hinzugefügt, der für verschiedene Messzwecke benutzt wird.

4.2.2 Software

Um die dem System hinzugefügte Hardware verwenden zu können, ist es notwendig, die Software entsprechend zu modifizieren. Die Eingabe, die in der bisherigen Version der Software nur aus einem hartkodierten Feld bestand, wird durch einen Software-Puffer ersetzt.

Die Dekodierung der in diesem Puffer vorhandenen Daten erfolgt frameweise. Enthält der Puffer nicht mehr genügend Daten für die Dekodierung eines weiteren Frames, wird er über eine eigens dafür implementierte Callback-Funktion wieder gefüllt. Beim Füllen des Puffers ist zu beachten, dass noch unverarbeitete Daten innerhalb des Puffers vorhanden sein können. Ist dies der Fall, müssen diese an den Anfang des Puffers verschoben werden und der freie Platz mit Daten von der CompactFlash-Karte gefüllt werden. Der Zugriff auf das Dateisystem erfolgt über eine von Xilinx bereitgestellte Dateisystem-Bibliothek, die den Zugriff auf die auf der CF-Karte gespeicherten Dateien ermöglicht.

Für die Ansteuerung des AC97-Controllers wird die in dem Entwurfsbeispiel, dem auch der Controller selbst entstammt, verwendete Software modifiziert. In der ursprünglichen Version wird die Interrupt-Funktion des Controllers nicht verwendet, da diese jedoch benötigt wird, um eine Echtzeitdekodierung zu ermöglichen, wurde eine entsprechende Interrupt Service Routine implementiert. Diese Routine füllt den innerhalb des Controllers verwendeten Hardware-Puffer, sobald der Füllstand des Puffers auf weniger als die Hälfte sinkt.

Um eine Echtzeitwiedergabe zu ermöglichen, wird zusätzlich zu den Hardware-Puffern ein Software-Ringpuffer verwendet. Dieser wird vom Dekoder frameweise gefüllt und von der Wiedergabeinterrupt-Routine ausgelesen. Abbildung 4.1 zeigt, wie der Ringpuffer verwendet wird, um die Daten zwischenzuspeichern. Es findet also eine doppelte Pufferung statt. Der Ringpuffer wird vom Dekoder frameweise mit Daten gefüllt, die Interrupt-Routine liest die Daten aus dem Ringpuffer und schreibt sie wortweise in die für die Wiedergabe verwendeten FIFO-Puffer des Controllers.

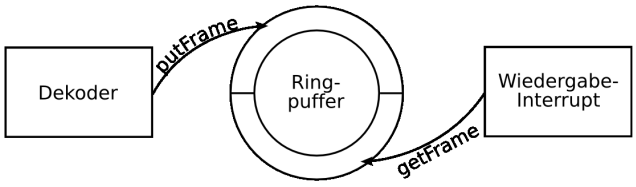

Abbildung 4.1: Verwendung des Ringpuffers während der Wiedergabe

Dieses Vorgehen hat den Vorteil, dass der Dekoder die Bearbeitung des nächsten Frames bereits beginnen kann, während der aktuelle Frame noch nicht vollständig ausgegeben wurde. Falls der Ringpuffer keinen weiteren Frame aufnehmen kann, wird der Dekoder blockiert bis die Ausgabe des aktuellen Frames beendet ist und somit der Puffer Platz für einen weiteren Frame bietet.

Für den Timer werden zunächst keine eigenen Software-Routinen implementiert, stattdessen soll die vom Xilinx EDK angebotene Profiling-Funktion verwendet werden, um statistische Daten über das Anwendungsprogramm zu gewinnen. Dabei ergeben sich jedoch verschiedene Probleme. Aufgrund verschiedener Ungenauigkeiten innerhalb der Dokumentation der Profiling-Funktion gelingt es, trotz aktivierter Profiling-Option, zunächst nicht, die Ausführungszeiten der einzelnen Funktionen zu messen, es wird lediglich die Anzahl der durchgeführten Funktionsaufrufe gezählt. Die eingehende Untersuchung dieses Verhaltens führt zu dem Ergebnis, dass der für die Messung verwendete Timer nicht korrekt initialisiert wird. Diese Initialisierung muss zu Beginn des Hauptprogramms durch den Aufruf der undokumentierten Funktion `_profile_init()`erfolgen. Nach erfolgter Initialisierung werden auch die Ausführungszeiten korrekt protokolliert.

Das schwerwiegendere Problem ergab sich jedoch durch die schon in 3.5.3.2 diskutierte Beeinflussung der Systemperformance durch das Profiling. Während des Profilings wird der Timer dazu verwendet, die Ausführung des eigentlichen Programms in konfigurierbaren Intervallen zu unterbrechen. Die Zeit, die zwischen zwei solchen Timerunterbrechungen vergangen ist, wird einer Funktion zugeordnet.

Wird eine grobe Zeitauflösung verwendet, so wird der Timerinterrupt selten ausgeführt und der Overhead ist somit niedrig, jedoch wird kurzen Funktionen oft keine Zeit zugeordnet. Wird im Gegensatz dazu eine feine Auflösung verwendet, so ist die Zuordnung der Ausführungszeiten genauer, der Overhead jedoch höher. Dies ist besonders problematisch, da die Geschwindigkeit der Ausgabe durch den *AC97 Codec* bestimmt wird und somit das System Echtzeitanforderungen unterliegt.

Sinkt die Systemperformance durch das aktivierte Profiling, so treten bestimmte Effekte nicht mehr auf, zum Beispiel erfolgt die Dekodierung der Frames nicht mehr in der zur Verfügung stehenden Zeit. Das heißt der Ausgabepuffer wird komplett geleert, bevor die Dekodierung des nächsten Frames abgeschlossen ist. Dies führt auch dazu, dass Effekte,

die durch Unterbrechungen durch den Wiedergabe-Interrupt verursacht werden, nicht mehr beobachtet werden können, da der Puffer bereits vollständig geleert wurde und keine weiteren Interrupts aufgerufen werden. Somit ist das Software-Profiling unter den gegebenen Umständen nicht geeignet, Aussagen über das System zu treffen.

Da für die reine Softwareimplementierung, wie sie in dieser Phase erstellt wurde, die detaillierten Informationen, die das Profiling liefern würde, nicht benötigt werden, können für eine Abschätzung der Systemperformance auch die anderen, in 3.5.3.1 und 3.5.3.2 dargestellten, Verfahren verwendet werden.

4.3 Phase III - Modifikation der HW-DCT

In diesem Schritt soll die vorher erstellte Softwareimplementierung durch eine Hardware-DCT erweitert werden. Hierzu ist es zunächst notwendig, die zu verwendende Implementierung der diskreten Kosinustransformation anzupassen. Dies ist vor allem daher nötig, da die Komponente entwickelt wurde, um auf einem Development-Board der Firma Altera verwendet zu werden, und daher spezifisch an dieses Board angepasste Komponenten verwendet. Unter anderem wird der auf dem Board vorhandene SRAM-Speicher zur Speicherung von Parametern und Zwischenergebnissen verwendet.

4.3.1 Analyse der bestehenden HW-DCT-Komponente

Zunächst erfolgt jedoch eine Analyse des Ablaufs der Berechnung, um die für die Ausführung der DCT benötigte Zeit zu bestimmen. Ergebnis der Analyse ist, dass die Hardwareimplementierung nur einen Bruchteil der Performance der Softwareimplementierung liefert. Obwohl die Zeit, die für die Berechnung der DCT benötigt wird, für die Bestimmung des Kommunikationsoverheads unerheblich ist, soll die Echtzeitfähigkeit des Systems erhalten bleiben. Somit sind, anders als ursprünglich vorgesehen, neben der Anbindung an den OPB-Bus noch weitere Anpassungen nötig, um die Performance zu verbessern.

Eine eingehende Analyse der Struktur der DCT ergibt zwei vielversprechende Ansatzpunkte für eine Verbesserung der Systemperformance. Besonders negativ wirkt sich die Verwendung des SRAMs auf die Leistungsfähigkeit des Systems aus. Durch eine Ersetzung des SRAMs durch einen Registersatz mit 64 Registern und drei Ports, zwei für den lesenden Zugriff und einer für den schreibenden, kann eine beträchtliche Steigerung der Leistungsfähigkeit erreicht werden. Dies liegt hauptsächlich darin begründet, dass die für den Zugriff auf die Speicherstellen benötigten Wartezyklen entfallen können.

Eine weitere Verbesserung der Leistung kann durch eine Modifikation der verwendeten Zustandsautomaten und der damit verbesserten Auslastung der verwendeten Addierer und Multiplizierer erreicht werden.

4.3.2 Optimierung der HW-DCT

Die ursprüngliche Implementierung verwendet einen Multiplizierer sowie einen Addierer für die eigentliche Berechnung der DCT. Die Möglichkeit zur parallelen Berechnung wurde jedoch nicht ausgenutzt. Durch eine Anpassung der Ausführungsreihenfolge und durch den Wegfall der für die Zugriffe auf das SRAM benötigten Wartezyklen kann die Auslastung der Addierer und Multiplizierer deutlich gesteigert werden.

Als letzte Maßnahme zur Steigerung der Performance kann zusätzlich noch eine Bit-Schiebeoperation, die für die weitere Verarbeitung der von der DCT erzeugten Ausgabedaten notwendig ist, durch die DCT-Komponente durchgeführt werden. Dies ist besonders vorteilhaft, da diese Schiebeoperation durch das Auswählen eines geeigneten Ausschnitts aus einem 64 Bit breiten Multiplikationsergebnis erfolgen kann, ohne dass eine tatsächliche Verschiebung der Bits in einem Register notwendig ist.

Die drei beschriebenen Maßnahmen führen dazu, dass die Berechnung der HW-DCT mit 1034 Takten circa 30 % schneller ist als die reine Softwareimplementierung. Für eine Abschätzung der Geschwindigkeit der Softwareimplementierung wurde der vom Kompiler erzeugte Assemblercode für die Berechnung der DCT analysiert und die einzelnen im *Microblaze Reference Manual*[Xilb] angegebenen Ausführungszeiten der Befehle addiert. Dies ist möglich, da innerhalb der DCT-Routine keinerlei Verzweigungen im Kontrollfluss existieren. Die Analyse des Assemblercodes ergab eine Ausführungszeit von etwa 1300 Takten. Diese Zeit basiert jedoch auf der Annahme, dass die Caches optimal arbeiten, das heißt der Microblaze muss die Pipeline nie auf Grund von Speicherzugriffen auf den Instruktions- oder Datenspeicher anhalten. Somit stellt diese Abschätzung den *best-case* dar. Die real zu messenden Ausführungszeiten für die Softwareimplementierung werden über dieser theoretisch bestimmten Ausführungszeit liegen.

4.3.3 Betrachtungen zur Leistungsfähigkeit der Komponente

Die gesamte Berechnung der DCT benötigt 1034 OPB-Bustakte bei einer Taktrate von 100 MHz, da die Komponente intern mit dem halben Bustakt betrieben wird, werden für die Berechnung 517 DCT-Takte benötigt. Davon entfallen 512 Takte auf die reine Berechnung und 5 auf die Initialisierung. Während der Berechnung werden 80 Multiplikationen, sowie 224 Additions- oder Subtraktionsoperationen durchgeführt. Daraus ergibt sich, dass der Multiplizierer nur in 15,625 % und der Addierer nur in 43,75 % aller Takte verwendet wird, die Auslastung also deutlich unter 50 % liegt.

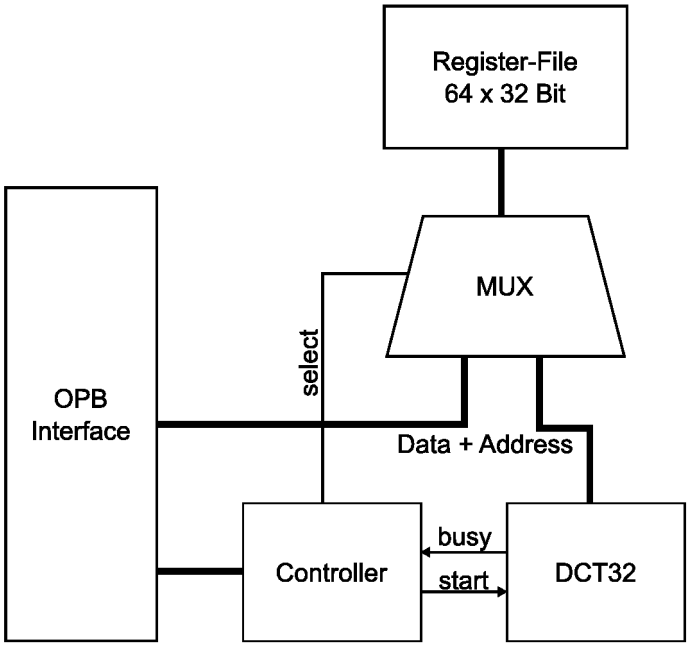

Abbildung 4.2: Schematische Darstellung der DCT-Komponente

Somit sind noch einige weitere Ansätze zur Steigerung der Leistung der DCT möglich: Beispielsweise kann durch Verwendung einer Variante des Multiplizierers die Pipelining unterstützt, die Taktrate und der Durchsatz erhöht werden. Eine weitere Möglichkeit ist, die Auslastung des verwendeten Addierers und Multiplizierers weiter zu verbessern und so eine Leistungssteigerung zu erzielen. Abschließend ist es möglich, die Leistungsfähigkeit durch Hinzufügen weiterer Addierer und Multiplizierer zu steigern.

Diese Ansätze zur weiteren Verbesserung der Leistung sollen jedoch nicht weiter verfolgt werden, da eine weitere Steigerung der Leistung keinerlei Vorteile für die Bestimmung des Kommunikationsoverheads hätte.

Als letzter Schritt bei der Modifikation der DCT-Komponente erfolgt die Implementierung eines OPB-Slave-Interfaces, das es ermöglicht, die Komponente über den OPB-Bus mit dem Microblaze zu verbinden. Abbildung 4.2 zeigt eine schematische Darstellung der Anbindung der DCT-Komponente an den OPB-Bus. Die so entstandene Komponente wird abschließend mit Hilfe eines VHDL-Simulators simuliert, um sowohl die Leistungsfähigkeit als auch die Funktion zu überprüfen.

4.4 Phase IV - Integration der HW-DCT und Messung

Die in Phase III fertiggestellte Hardware-DCT wird in diesem Schritt in das bestehende System eingefügt. Die daraus resultierende Implementierung mit der Berechnung der DCT durch eine Hardwarekomponente entspricht der zweiten in Abschnitt 3.2.1 genannten Implementierungen. Hierzu sind wiederum Anpassungen sowohl an der Hardware, als auch an der Software notwendig.

4.4.1 Änderungen der Hardware

Mit Hilfe des EDK wird die DCT-Komponente, die im vorherigen Schritt erstellt wurde, dem System hinzugefügt. Die Komponente wird hierzu mit dem OPB-Bus verbunden. Der Komponente werden zwei Adressbereiche aus dem Adressraum des Microblaze zugewiesen. Einerseits eine Adresse, an der das Status- und Kontrollregister eingeblendet wird, andererseits ein Adressbereich, in den die Register für die Parameter und Rückgabewerte eingeblendet werden.

4.4.2 Änderungen der Software

Um die DCT-Komponente zu verwenden, ist es nötig, die Software entsprechend anzupassen. Der Aufruf der DCT-Funktion innerhalb der MPEG-Audio-Bibliothek wird durch den Aufruf einer so genannten Callback-Funktion ersetzt. Dies erfolgt mittels eines Funktionszeigers, die dem Dekoder während der Initialisierung übergeben wird. Es wäre auch möglich, den Aufruf der DCT direkt in der Bibliothek so zu verändern, dass die Hardware-DCT verwendet wird, jedoch ist es für die folgenden Implementierungen unerlässlich, dass der Zugriff auf die Hardware aus dem Hauptprogramm heraus erfolgt. Aus der Verwendung einer Callback-Funktion ergeben sich noch weitere Vorteile. So ist es beispielsweise möglich, einfach zwischen Software-Implementierung und Hardware-Implementierung der DCT hin- und herzuwechseln. Hierzu ist es lediglich nötig, bei der Initialisierung einen Zeiger auf die gewünschte DCT-Funktion zu übergeben und das Hauptprogramm neu zu kompilieren. Die Bibliothek selbst muss somit nicht neu kompiliert werden.

Die bestehende Software DCT-Funktion wurde durch eine Funktion zur Ansteuerung der DCT-Komponente ersetzt. Bei einem Aufruf der DCT-Funktion werden die 32 Eingabewerte an die DCT-Komponente übertragen, indem sie in den Speicherbereich geschrieben werden, in den die internen Register der DCT eingeblendet sind. Nach der Übertragung der Parameter wird die Ausführung der DCT durch Setzen eines Bits im Kontroll- und Statusregister der DCT gestartet. Die DCT-Komponente beginnt die Berechnung und setzt das *busy flag* im Statusregister, um so anzuzeigen, dass eine Berechnung durchge-

führt wird. Die DCT-Funktion liest das Statusregister in einer Schleife kontinuierlich aus, bis das *busy flag* wieder gelöscht wird und somit die Berechnung der DCT abgeschlossen ist. Nach dem Abschluss der Berechnungen werden die Rückgabewerte ausgelesen und dem Dekoder für die weitere Verwendung zur Verfügung gestellt.

Die hier vorgestellte Anbindung der DCT nutzt die Möglichkeiten zur Parallelisierung, die sich durch eine Hardwareimplementierung der DCT ergeben, nicht effizient aus. Eine weitere Performancesteigerung ist möglich, indem der Prozessor während der Zeit, die die Berechnung der DCT benötigt, andere Teilaufgaben der Dekodierung berechnet. So ist es beispielsweise möglich, zunächst die gesamten Daten eines Frames in eine FIFO zu schreiben, aus der die DCT die jeweils für die Berechnung benötigten Werte selbstständig liest. Die DCT kann ihre Ausgabewerte dann ebenfalls in einen FIFO-Puffer schreiben, die ersten Rückgabewerte können dann bereits weiterverarbeitet werden, während die Berechnung der weiteren Daten noch nicht abgeschlossen ist. Da dies jedoch größere Anpassungen sowohl an der verwendeten Bibliothek, als auch an der Hardware-DCT zur Folge hat und die Performance des Gesamtsystems für die Bestimmung des Kommunikationsoverheads von geringer Bedeutung ist, soll dieser Ansatz nicht weiter verfolgt werden.

4.4.3 Messung der Systemperformance

Für die Messung der Performance und der für die Kommunikation mit der HW-DCT benötigten Zeit wird der im System vorhandene Timer verwendet. Es wurden mehrere Testläufe durchgeführt, bei denen verschiedene Testdaten mit einer Datenrate zwischen 32 kb/s und 320 kb/s verwendet wurden. Zunächst werden die für die Dekodierung eines jeden Frames benötigte Takte bestimmt. Hierbei ergibt sich, im Vergleich zur reinen Softwareimplementierung, eine Performanceverbesserung um vier Prozentpunkte sowohl in der durchschnittlichen Auslastung, als auch in der Spitzenauslastung. In weiteren Tests wird die Zeit gemessen, die für die Berechnung der DCT in Hardware, mit Übertragung der Daten, benötigt wird. Diese Daten dienten als Grundlage für den Vergleich und die Analyse der OSSS-Implementierung.

4.5 Phase V - Implementierung in OSSS und Simulation

In diesem Abschnitt soll die Realisierung der HW/SW-Implementierung des MPEG-Audio-Dekoders mit Hilfe der OSSS-Methodik beschrieben werden. Hierzu werden zwei verschiedene Konzepte von OSSS verwendet. Einerseits wird die auf dem Prozessor ausgeführte Software als OSSS-SW-Task modelliert, andererseits soll die Kommunikation der Software mit der ausgelagerten HW-DCT durch Verwendung des Shared Object-Konzepts erfolgen.

4.5.1 Überlegungen zum Shared Object

Bei der Erstellung der OSSS-Implementierung ist es zunächst notwendig, eine Zuordnung der Systemkomponenten zu Software- und Hardwarekomponenten zu finden. Diese Zuordnung wird für das konkret verwendete Beispiel aus den vorherigen Implementierungen übernommen. Für die Kommunikation zwischen der Softwarekomponente und der Hardware soll jedoch ein Shared Object verwendet werden.

Für die Implementierung des Shared Objects ergeben sich jedoch verschiedene Möglichkeiten, von denen zwei Implementierungsalternativen im Folgenden näher betrachtet werden sollen. Für die nachfolgende Synthese wird jedoch nur die erste Variante weiter verwendet.

4.5.1.1 Shared Object als Interface zur Hardware

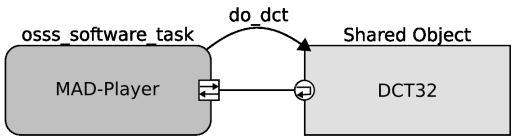

Abbildung 4.3: Das Shared Object als Interface zur Hardware

Eine Möglichkeit, das Shared Object-Konzept im Rahmen des Designs zu verwenden, ist in der Form eines methodenbasierten Interfaces zur Hardware, wie in Abbildung 4.3 dargestellt. Hierbei erfolgt der Aufruf einer von einer Hardwarekomponente bereitgestellten Funktion aus Sicht der Software wie ein lokaler Funktionsaufruf. Das Protokoll, welches für die Kommunikation zwischen der Software und der Hardwarekomponente verwendet wird, wird dabei von OSSS erzeugt. Diese Umsetzung entspricht dem in 4.4 verwendeten Ansatz zur Kommunikation mit der HW-Komponente.

4.5.1.2 Shared Object als geteilte Ressource

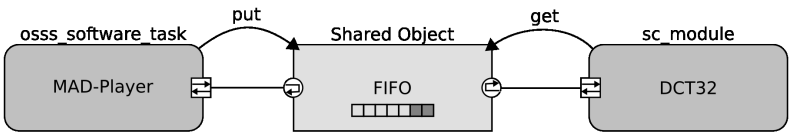

Abbildung 4.4: Das Shared Object als geteilte Ressource

Ein weiteres mögliches Vorgehen ist es, nicht die HW-DCT als Shared Object zu modellieren, sondern eine geteilte Ressource, wie zum Beispiel einen FIFO-Puffer so wie in

Abbildung 4.4 gezeigt. Bei dieser Implementierung stellen sowohl die Software, als auch die HW-DCT Klienten des Shared Objects dar. Dieser Ansatz entspricht dem zweiten in Abschnitt 4.4 vorgestellten Ansatz.

Für eine Simulation werden beide Ansätze implementiert. Da Simulationen sowohl auf OSSS-Anwendungsebene, als auch auf Architekturebene durchgeführt werden, ergeben sich vier verschiedene Simulationsmodelle.

4.5.2 Einbindung der MAD-Library

Eine erste Schwierigkeit bei der Erstellung der OSSS-Implementierung ergab sich aus der Tatsache, dass die verwendete MPEG-Audio-Bibliothek in C implementiert ist, wohingegen OSSS als Erweiterung von SystemC auf C++ basiert. Ohne weitreichende Modifikationen der Bibliothek wäre es nicht möglich, diese mit einem C++-Kompiler zu kompilieren. Die Hauptprobleme, die eine einfache Überführung des Quellcodes von C nach C++ verhinderten, sind fehlende Typkonvertierungen sowie die häufige Verwendung statischer Initialisierer für *structs* und *unions*, die von C++ nicht in dieser Form unterstützt werden. Daher wird nur das eigentliche Hauptprogramm nach C++ portiert, die verwendete MPEG-Audio-Bibliothek wird mit Hilfe eines C-Kompilers übersetzt und aus dem Objektcode eine statische Bibliothek erzeugt. Diese wird dann beim Linken des Hauptprogramms eingebunden. Für diesen Ansatz ist es von Vorteil, dass bei der Erstellung der vorherigen Implementierung der Aufruf der DCT, mit Hilfe von Funktionszeigern und Callback-Funktionen, in das Hauptprogramm verlagert wurde.

4.5.3 Portierung des Hauptprogramms

Um das Hauptprogramm als `osss_software_task` zu modellieren, ist es nötig, die Software nach C++ zu übertragen und in eine eigene Klasse zu kapseln. Diese Klasse wird von der Basisklasse `osss_software_task`, die von OSSS zur Verfügung gestellt wird, abgeleitet. Durch die Verwendung einer C-Bibliothek aus einem C++-Hauptprogramm ergeben sich jedoch zusätzliche Schwierigkeiten. So ist es nicht möglich, Funktionszeiger auf Member-Funktionen zu erhalten. Diese werden jedoch für die dem Dekoder übergebenen Callback-Funktionen benötigt. Dieses Problem kann gelöst werden, indem für jede verwendete Callback-Funktion ein statischer Funktions-Wrapper angelegt wird (siehe Listing 4.1).

Für statische Funktionen ist es möglich, Funktionszeiger zu erzeugen, die auf die entsprechende Funktion zeigen. In den statischen Wrapperfunktionen erfolgt dann der Aufruf der entsprechenden Memberfunktionen. Hierzu wird ein Zeiger auf eine Instanz der Player-Klasse benötigt, der über eine globale Variable, die während der Erzeugung des Player-Objektes gesetzt wird, zur Verfügung gestellt wird. Bei diesem Ansatz ist es nur

```
[...]
class player;

static void dct_wrapper(mad_fixed_t const in[32], unsigned int slot,
                        mad_fixed_t lo[16][8], mad_fixed_t hi[16][8]);
[...]
player *theObject;

class player : public osss_software_task
{
  [...]
  void dct32(mad_fixed_t const in[32], unsigned int slot,
             mad_fixed_t lo[16][8], mad_fixed_t hi[16][8])
  {
    [...]
  }
  [...]
};

static void dct_wrapper(mad_fixed_t const in[32], unsigned int slot,
                        mad_fixed_t lo[16][8], mad_fixed_t hi[16][8])
{
  theObject->dct32(in, slot,lo, hi);
}
[...]
```

Listing 4.1: Einfügen statischer Funktions-Wrapper für Memberfunktionen in C++

möglich, eine einzige Instanz eines Players zu erzeugen, dies bedeutet für die im Rahmen dieser Arbeit erstellte Beispielapplikation jedoch keine Einschränkung.

4.5.4 Erstellen des OSSS-Anwendungsebenen-Modells

Neben der Umsetzung des Hauptprogramms nach OSSS sind noch weitere Modifikationen nötig. Für die Kommunikation des Software-Tasks mit dem Shared Object ist es notwendig, dem Software-Task einen Port hinzuzufügen (siehe Listing 4.2), an den das Shared Object gebunden wird. Der Port stellt dabei einen Stellvertreter, oder auch *proxy*, für das Shared Object dar und stellt dessen Funktionen der Software zur Verfügung. Die Software kann die bereitgestellten Funktionen aufrufen, als wären sie lokale Funktionen des Ports. Durch dieses Verfahren wird der Kommunikationsmechanismus, über den die Kommunikation mit der Hardware erfolgt, vor der Software verborgen.

Die Bindung des Ports des Software-Tasks an das Shared Object entspricht der Port / Interface-Bindung von SystemC (siehe Listing 4.3). Dazu wird das Methodeninterface des Shared Objects als abstrakte Klasse realisiert, die dann bei der Deklaration des Ports als Template-Parameter übergeben wird.

Um die Daten für die Übertragung zwischen dem Software-Task und der Hardware-Komponente zu kapseln, wird eine neue Klasse `dct_data` eingeführt. Die Klasse ist abgeleitet von der Klasse `osss_seriablisable_object` und implementiert die Funktionen `serialise` und `deserialise`. Dies ist notwendig, um Instanzen dieser Klasse für den Transport zwischen dem Software-Task und dem Shared Object vorzubereiten.

Da die DCT für die spätere Hardware-Implementierung nicht aus dem SystemC-Modell synthetisiert werden soll, wird eine nicht-synthetisierbare Implementierung der DCT innerhalb des Modells verwendet. Es werden die beiden oben vorgestellten Implementierungsvarianten erstellt und simuliert.

Bei der Simulation des OSSS-Anwendungsebenen-Modells wird sowohl von der Art des Kommunikationskanals zwischen Software-Task und Shared Object, als auch von dem Prozessor, auf dem die Software ausgeführt wird, abstrahiert. Das Zeitverhalten wird somit nicht modelliert, dass Modell stellt also ein rein funktionales Modell des Systems dar.

Durch eine Verfeinerung des Modells ist es möglich, in der Simulation auch das Zeitverhalten zu modellieren. Für die Modellierung des Zeitverhaltens der Software ist es jedoch nötig, die Ausführungszeiten der Software durch die Verwendung so genannter EET-Makros zu annotieren.

```
class player : public osss_software_task
{
public :
  osss_port_to_shared< dct_if > dct_port; //Deklaration des Ports
  OSSS_SW_CTOR(player)                    //Konstruktor des Software-Tasks
  {
  }
[...]
void dct32(mad_fixed_t const in[32], unsigned int slot,
           mad_fixed_t lo[16][8], mad_fixed_t hi[16][8])
{
  int i = 0;
  dct_data input, output;
  //Vorbereiten des Datenobjekts
  for(i = 0; i < 32; i++){
    input.setData(i, in[i]);
  }

  //Aufruf der Funktion des Shared Objects
  output = dct_port->do_dct(input);

  //verarbeiten der Rückgabewerte
  for(i = 0; i < 16; i++){
    hi[15-i][slot] = output.getData(i);
    lo[i][slot] = output.getData(i+16);
  }
}
[...]
```

Listing 4.2: Implementierung des Software-Tasks

```
SC_MODULE(Top)
{
  sc_in<bool>   Clk;
  sc_in<bool>   Reset;

  player *player1;

  osss_shared< dct > *dct_inst;

  SC_CTOR( Top )
  {
    dct_inst = new osss_shared<dct>("dct1");
    dct_inst->clock_port(Clk);
    dct_inst->reset_port(Reset);

    player1 = new player( "player1" );
    player1->clock_port(Clk);
    player1->reset_port(Reset);
    player1->dct_port(*dct_inst);
  }
};
```

Listing 4.3: Implementierung der *Top-Level-Entity* auf Anwendungsebene

4.5.5 Erstellen des OSSS-Architekturebenen-Modells

Die beiden auf Anwendungsebene erstellten Entwürfe werden in einem weiteren Schritt zu Modellen auf Architekturebene verfeinert. Die dafür nötigen Änderungen sind für beide Designs identisch, daher sollen sie hier nur einmal beschrieben werden.

```
[...]
class player : public osss_software_task
{
public :
  osss_port<osss_rmi_if<dct_if > > dct_port;
[...]
```

Listing 4.4: Hinzufügen des RMI-Ports zum Software-Task

Das Modell wird bei der Verfeinerung um einen Prozessor und einen OSSS-Channel erweitert (Siehe Listing 4.5). Der Software-Task wird dem Prozessor zugeordnet und der Channel sowohl an den Prozessor, als auch an das Shared Object gebunden. Die Implementierung des OSSS-Channels kann nach den jeweiligen Anforderungen der Anwen-

```
[...]
#define chan_type osss_rmi_channel<xilinx_opb_channel<false, false> >

SC_MODULE(Top)
{
  sc_in<bool>   Clk;
  sc_in<bool>   Reset;

  player *player1;

  chan_type *channel;

  osss_object_socket <osss_shared< dct > > *dct_inst;

  osss_processor* m_processor;

  SC_CTOR( Top )
  {
    CREATE_OSSS_CHANNEL( channel, chan_type, "channel" );
    channel->clock_port(Clk);
    channel->reset_port(Reset);

    dct_inst = new osss_object_socket< osss_shared<dct> >();
    dct_inst->clock_port(Clk);
    dct_inst->reset_port(Reset);
    dct_inst->bind(*channel);

    player1 = new player( "player1" );
    player1->dct_port(*dct_inst);

    m_processor = new xilinx_microblaze("my_processor");
    m_processor->clock_port(Clk);
    m_processor->reset_port(Reset);
    m_processor->rmi_client_port(*channel);
    m_processor->add_sw_task(player1);
  }
};
```

Listing 4.5: Implementierung der *Top-Level-Entity* auf Architekturebene

dung gewählt werden. Für die beiden hier erstellten Entwürfe wird eine Implementierung des OPB-Busses als OSSS-Channel gewählt.

Die Simulation auf Architekturebene berücksichtigt neben der Ausführungszeit der Software auch die für die Kommunikation benötigte Zeit. Die Güte der Simulation der Softwareausführungszeiten ist jedoch abhängig von der Genauigkeit der mittels der EET-Makros annotierten Zeiten. Die Simulation der Kommunikation erfolgt zyklusgenau.

Auch mit den Architekturebenen-Modellen werden verschiedene Testläufe durchgeführt, bei denen verschiedene Bitraten getestet werden. Die Ergebnisse der Simulation sollen jedoch im Rahmen dieser Arbeit nicht weiter betrachtet werden, da die durch die Simulation gewonnenen Daten für das vorliegende Modell nur eine geringe Aussagekraft haben. Dies ist dadurch bedingt, dass es aufgrund der Struktur des Software-Tasks, welcher aus C++-Callbackfunktionen und einer C-Bibliothek für die MPEG-Audio-Dekodierung besteht, nicht möglich war, die geschätzten Ausführungszeiten korrekt zu annotieren. Bei einer anderen Struktur des Software-Tasks wäre es möglich, durch Angeben der geschätzten Ausführungszeiten das Zeitverhalten der Software bei der Simulation zu berücksichtigen.

Die in diesem Schritt realisierte Implementierung entspricht der Eingabe des OSSS-Synthese-Flows. Es ist nicht nötig, dass der Entwickler eine weitere Verfeinerung des Modells, insbesondere der HW/SW-Kommunikation vornimmt. Die weitere Verfeinerung des Modells erfolgt automatisch durch das Synthesewerkzeug. Durch dieses Vorgehen wird dem Designer die aufwändige Verfeinerung der Kommunikation abgenommen. Besonders die Implementierung komplexer Busprotokolle wird dem Designer durch die automatische Synthese abgenommen. Die ermöglicht, im Vergleich zur herkömmlichen *low-level* Implementierung des Kommunikationsinterfaces, einfachere Änderungen des Designs. So ist es beispielsweise möglich, zusätzliche HW-Komponenten hinzuzufügen, oder den für die Kommunikation verwendeten Mechanismus auszutauschen. Dadurch wird der Designvorgang effizienter und die Produktivität des Designers gesteigert.

Da die Werkzeuge für eine automatische Synthese zum Zeitpunkt dieser Arbeit noch nicht zur Verfügung stehen, ist es im nächsten Schritt nötig, die Synthese manuell durchzuführen, um zu einer Implementierung auf Hardware-Ebene zu gelangen.

4.6 Phase VI - Umsetzung des RMI-Protokolls

In diesem Abschnitt erfolgt die Implementierung der dritten in 3.2.1 erwähnten Implementierungen. Da das OSSS-Synthesewerkzeug nicht zur Verfügung steht, wird die Synthese des Architekturmodells manuell durchgeführt. Die Implementierung des Shared Objects wurde dabei nicht aus dem OSSS-Modell erzeugt, sondern es wurde die in den vorherigen Phasen verwendete HW-DCT Komponente so modifiziert, dass sie dem

Ergebnis einer Synthese des OSSS-Modells entspricht. Hierzu wird jedoch eine vereinfachte Version des Shared Objects angenommen, bei dem der Scheduler entfällt, da das Shared Object lediglich einen Klienten besitzt. Weiterhin entfällt der *Guard Evaluator*, da die Guard-Bedingung der bereitgestellten Funktion immer „wahr“ ist. Diese Optimierungen sind auch automatisiert durch das Synthesewerkzeug geplant, somit sollten sie die Betrachtungen nicht verfälschen. Diese Modifikationen betreffen jedoch nur das Interface zwischen dem Microblaze und der DCT, der Funktionsblock, der die eigentliche Berechnung der DCT durchführt, bleibt unverändert.

Die Software wird derart angepasst, dass statt des eigenen Protokolls zur Kommunikation mit der DCT, das OSSS-RMI-Protokoll verwendet wird. Hierbei soll darauf geachtet werden, dass das Ergebnis der „manuellen Synthese“ möglichst identisch zu dem der automatischen ist.

4.6.1 Anpassung der HW-DCT

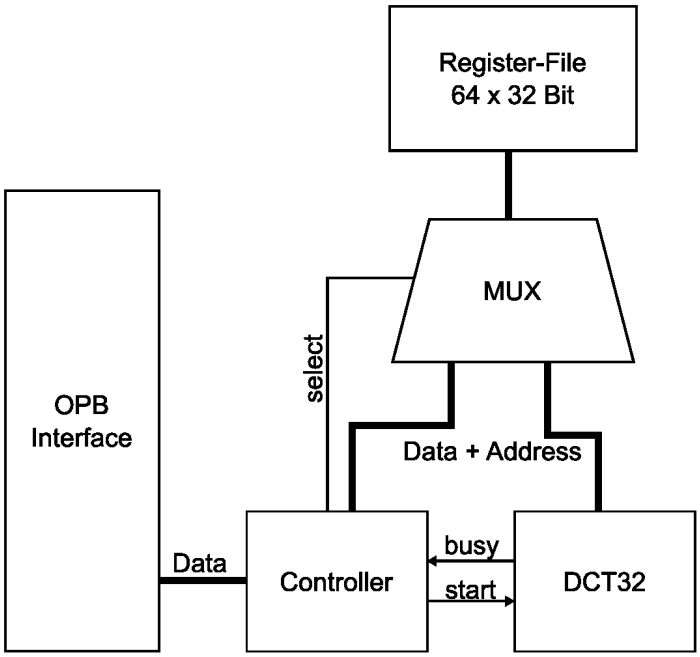

Abbildung 4.5: Schematische Darstellung der RMI-DCT-Komponente

Zunächst wird das für den Zugriff auf die DCT verwendete Protokoll an das in 3.4.2 vorgestellte RMI-Protokoll angepasst. Hierzu ist es nötig, dem Interface einen Zustandsautomaten hinzufügen, der einerseits den Ablauf der verschiedenen Protokollschritte koordiniert und andererseits die Serialisierung und Deserialisierung der zu übertra-

genden Daten durchführt. Abbildung 4.5 zeigt eine Darstellung der veränderten DCT-Komponente. Die für die Adressierung der aufgerufenen Methode benötigte eindeutige *MethodID* wird beliebig gewählt. Diese würde während der Synthese automatisch durch das Synthesewerkzeug erzeugt werden. Die OSSS-Implementierung der HW-DCT stellt jedoch nur eine einzige Methode zur Verfügung, daher wird die übertragene *MethodID* vom Shared Object nicht ausgewertet. Die modifizierte Komponente wird wiederum mit Hilfe eines VHDL-Simulators simuliert.

4.6.2 Anpassung der Software

Auch an der Software sind verschiedene Anpassungen notwendig. Aus dem Modell des Systems werden die zur Modellierung der Hardware benötigten Komponenten sowie die eingefügten EET-Makros entfernt. Der Software-Task wird als Hauptprogramm übernommen. Der Port wurde durch ein Objekt ersetzt, welches die Abarbeitung des OSSS-RMI-Protokolls auf der Klientenseite steuert. Hierzu wird weitestgehend der gleiche Ablauf für die Serialisierung der zu übertragenden Daten verwendet, wie auch bei der Simulation auf Architekturebene. Im Folgenden soll dieser Ablauf näher betrachtet werden.

Der Ablauf des Protokolls beginnt, sobald eine Interface-Methode des Shared Objects aus dem Hauptprogramm aufgerufen wird. Zunächst wird der Aufruf der Methode auf dem Shared Object vorbereitet. Hierzu wird unter anderem die *ObjectID* des Shared Objects und die *MethodID* der aufgerufenen Methode bestimmt. Daraufhin werden die Methodenparameter serialisiert.

Abbildung 4.6 stellt beispielhaft die Serialisierung eines Objektes mit vier Datenmembern sowie die Übertragung der serialisierten Daten über einen 32 Bit breiten OPB-Bus dar. Zunächst wird aus den Datenmembern ein Bitvektor erzeugt, indem die Daten bitweise in einen Vektor geschrieben werden. Dieser Bitvektor wird in so genannte *Data Chunks* unterteilt. Die Bitbreite dieser *Data Chunks* entspricht der Breite des minimal über den Kommunikationskanal übertragbaren Einheit. Diese entspricht für den OPB-Bus einem Byte, somit werden aus dem Bitvektor mehrere acht Bit breite *Data Chunks* erzeugt. Um den Durchsatz über den OPB zu erhöhen, werden mehrere *Data Chunks* zusammengefasst, um für jede Übertragung die volle Breite des Datenbusses auszunutzen. Für das dargestellte Beispiel beträgt die Breite des Busses 32 Bit, somit werden jeweils vier *Data Chunks* zu einem zu übertragenden Wert zusammengefasst. Sollte die Anzahl *Data Chunks* nicht durch vier teilbar sein, so wird die Anzahl, durch Hinzufügen von *Data Chunks* mit dem Wert null, zu einer durch vier teilbare Anzahl ergänzt.

Die Kommunikation und die Übertragung der Daten an das Shared Object erfolgt in mehreren Schritten. Zunächst sendet der Klient, im konkreten Fall der Software-Task, die *MethodID* der gewünschten, vom Shared Object bereitgestellten, Funktion. Dies erfolgt über einen Schreibzugriff auf eine Adresse des Shared Objects. Im Falle von Bus-

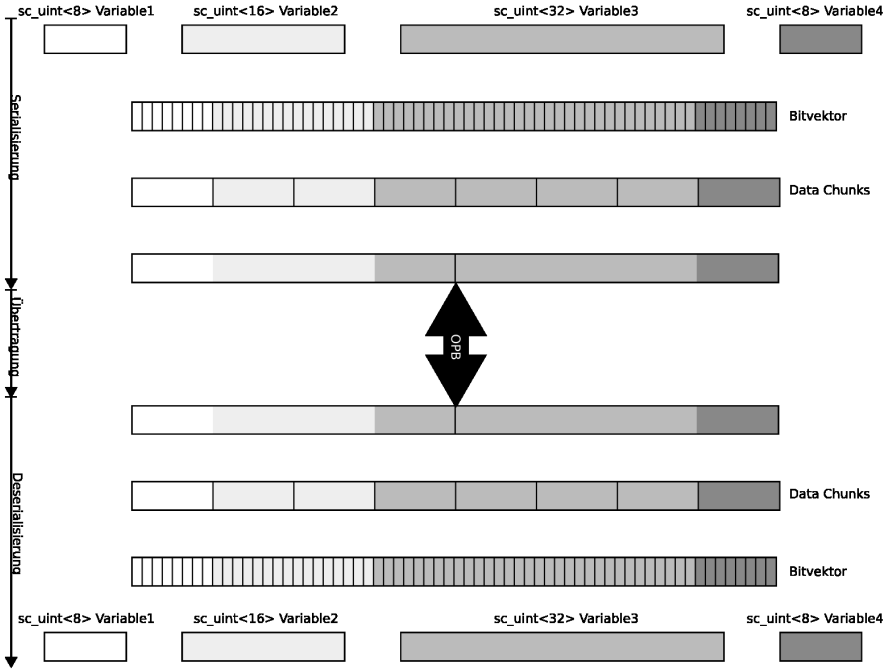

Abbildung 4.6: Beispiel für die Serialisierung eines Objekts

sen, über die mehrere Klienten mit einem Shared Object kommunizieren, erhält jeder Klient eine andere Zieladresse für die Zugriffe auf das Shared Object. Dies ist nötig, damit das Shared Object die Klienten eindeutig identifizieren kann. Nach der Übertragung der *MethodID* überprüft das Shared Object, ob die angefragte Methode verfügbar ist, also noch keinem anderen Klienten ein Zugriff auf die Methode gestattet wurde, und evaluiert, soweit vorhanden, die *Guard Conditions*. Der Klient liest so lange von der ihm zugewiesenen Speicheradresse, bis das Shared Object ihm den Zugriff auf die angefragte Methode gestattet. Sobald der Zugriff erlaubt wurde, beginnt der Klient mit der Übertragung der Daten, hierzu wird zunächst die Länge des folgenden Datenblockes und im Anschluss daran das serialisierte Objekt an das Shared Object übertragen. Nach Abschluss der Übertragung der Parameter beginnt das Shared Object mit der Ausführung der angefragten Funktion. Während die Funktion ausgeführt wird, fragt der Klient den Status des Shared Objects kontinuierlich ab. Signalisiert das Shared Objekt die Fertigstellung der aufgerufenen Methode, beginnt der Klient mit dem Zurücklesen der Rückgabewerte. Die Übertragung der Rückgabewerte und deren Deserialisierung erfolgt analog zur Übertragung der Parameter, nur in umgekehrter Richtung. Nachdem die Rückgabewerte übertragen wurde, beendet der Klient den Transfer, in dem er wiederum die *MethodID* an das Shared Object sendet. Das Ergebnis des Methodenaufrufs steht dem Software-Task dann zur weiteren Verarbeitung zur Verfügung.

Das durch die beschriebenen Anpassungen von Soft- und Hardware erzeugte System wird wiederum verwendet, um verschiedene MPEG-Audio-Datenströme zu dekodieren.

Wie schon in der Implementierung mit dem eigenen Kommunikationsprotokoll wird der Timer verwendet, um die für die Kommunikation der Software mit der Hardware benötigte Zeit zu messen. Die Messung zeigt, dass die Verwendung des RMI-Protokolls eine drastische Reduktion der Systemperformance verursacht (Siehe 5.3).

4.7 Phase VII - Implementierung eines optimierten RMI-Protokolls

Da die Verwendung des RMI-Protokolls zu einem drastischen Performanceverlust führt, soll bestimmt werden, ob der Verlust durch das Protokoll selbst bedingt ist, oder die Serialisierung für die Leistungsverschlechterung verantwortlich ist. Zu diesem Zweck wurde das RMI-Protokoll auf Seiten des Software-Tasks angepasst und optimiert. Da das angepasste Verfahren nicht dazu dienen soll, das bisherige von OSSS verwendete Serialisierungsverfahren zu ersetzen, ist es möglich, dass Verfahren speziell an den vorliegenden Anwendungsfall anzupassen. Ein Schlüsselaspekt bei der Reimplementierung ist der Verzicht auf die in der *Standard Template Library* definierten Datenstrukturen, wie zum Beispiel Vektoren.

Für das konkret verwendete MPEG-Audio-Beispiel bedeutet dies, dass die Daten, ohne weitere Vorverarbeitung, über den Datenbus übertragen werden können, da sie schon in einem Datenformat, das der Busbreite entspricht, vorliegen. Somit ist lediglich eine Übertragung der Werte in der richtigen Reihenfolge nötig.

Auch für den so modifizierten Entwurf wird wieder, mit Hilfe des Timers, die für die Kommunikation verwendete Zeit bestimmt. Die Auswertung der Testergebnisse ergibt, dass der Performanceverlust hauptsächlich durch das verwendete Serialisierungsverfahren und nur zu einem kleinen Teil durch das Kommunikationsprotokoll verursacht wird.

5 Auswertung

In diesem Kapitel erfolgt die Auswertung der durch die durchgeführten Tests gewonnen Daten. Ziel der Auswertung soll sein, den durch die Verwendung eines automatisch generierten Kommunikationsinterface verursachten Overhead zu bestimmen. Die hierzu verwendeten Messverfahren wurden bereits in Kapitel 3.5 näher beschrieben.

5.1 SW-Implementierung

Zunächst sollen die Ergebnisse der reinen Softwareimplementierung betrachtet werden. Es wurden zwei verschiedene Messverfahren verwendet, um die Daten für die Auswertung zu gewinnen. Diese wurden bereits in 3.5.3.1 und 3.5.3.2 eingehend beschrieben, so dass an dieser Stelle nur die Ergebnisse der Messung präsentiert werden sollen.

5.1.1 Messung ohne Timer

Zu Beginn sollen die Ergebnisse der Messung ohne Timer betrachtet werden, bei der die ungenutzte CPU-Zeit gemessen wird, um so auf die Auslastung zu schließen. Es fällt auf, dass die Ergebnisse der Messung zwischen zwei Testläufen, die unter identischen Bedingungen gestartet wurden, leicht variieren. Die gemessenen freien Zeiten pro Frame variieren im Mittel um 0,19 %. Die Ursache für diese nichtdeterministischen Effekte liegt einerseits in der Verwendung verschiedener Taktdomänen, da der Prozessor und die Ausgabe über den *AC97 Codec* mit verschiedenen nicht synchronisierten Takten betrieben werden und es dadurch zu Drift-Effekten kommen kann. Ein weiterer Grund liegt in der Verwendung der CF-Karte. Diese arbeitet intern ebenfalls mit einem eigenen Takt, der nicht mit dem Takt des Prozessors synchronisiert ist. Die für identische Zugriffe auf die CF-Karte benötigte Zeit variiert daher auch um einen geringen Wert, so dass es zwischen zwei identischen Testläufen zu leichten Variationen in der gemessenen Auslastung kommen kann. Da diese Schwankungen jedoch mit durchschnittlich weniger als einem Prozent des Messwertes ohne bedeutende Auswirkungen auf das Messergebnis sind, werden diese Variationen nicht weiter betrachtet.

Abbildung 5.1 zeigt exemplarisch die Darstellung einer Messung mit einer verwendeten Datenrate von 32 kb/s sowie einer Datenrate von 320 kb/s. Dargestellt ist die Auslas-

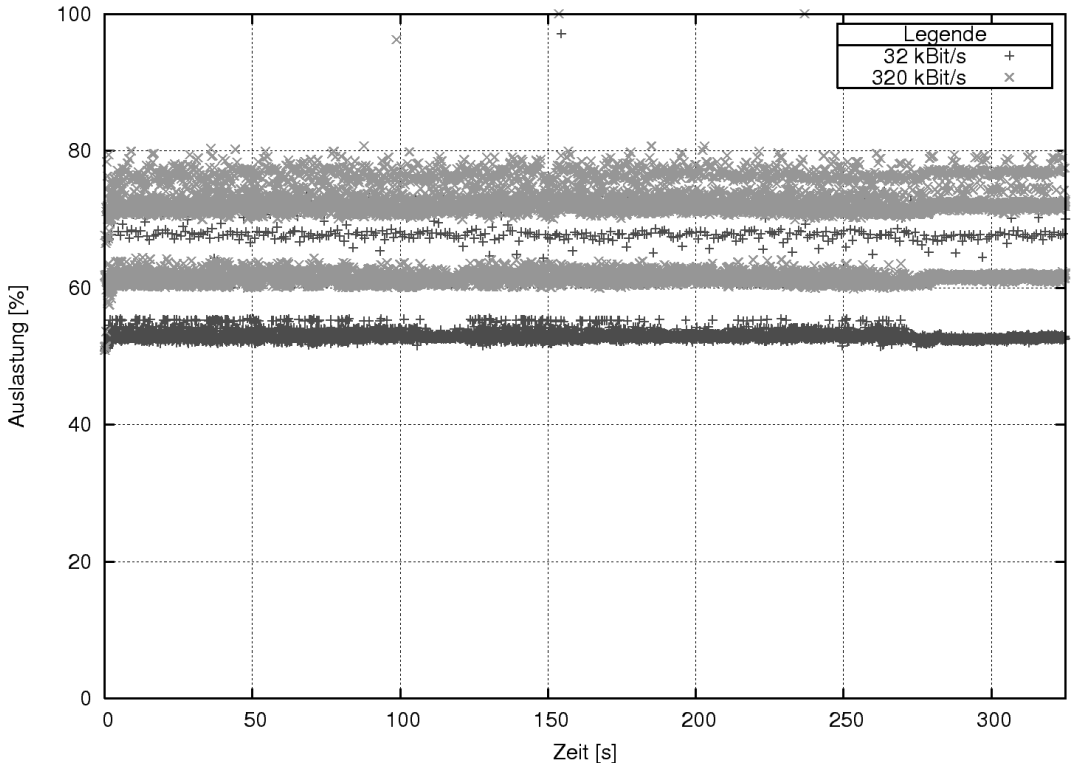

Abbildung 5.1: CPU-Auslastung für die Dekodierung (SW-DCT / Messung ohne Timer)

tung des Microblaze für jeden einzelnen Frame. Formel 5.1 wurde verwendet, um die Auslastung für jeden Frame zu bestimmen:

$$usage = \frac{t_{frame}}{t_{used}} \cdot 100 \tag{5.1}$$

Wobei t_{used} die für die Dekodierung des Frames benötigte Zeit und t_{frame} die für die Dekodierung maximal zur Verfügung stehende Zeit ist.

Die beiden betrachteten Bitraten wurden gewählt, da sie das Minimum bzw. das Maximum der im *Layer 3* definierten Bitraten darstellen. Die Prozessorauslastung ist erwartungsgemäß bei einer Bitrate von 320 kb/s höher, als bei einer Bitrate von 32 kb/s. Für zwei Frames liegt die Auslastung über 100 %, bei diesen Frames benötigte die Dekodierung also mehr als die zur Verfügung stehende Zeit. Durch die Verwendung von Ausgabepuffern geeigneter Größe ist dennoch eine Echtzeitdekodierung möglich. So ist es in dem betrachteten Beispiel ausreichend, einen Puffer für zwei Frames vorzusehen, um die auftretenden Zeitüberschreitungen abzufangen.

Tabelle 5.1 stellt die durchschnittliche sowie die maximale Auslastung für die beiden betrachteten Bitraten dar. Die mit diesem Verfahren gemessene durchschnittliche Aus-

Bitrate	durchschnittliche Auslastung	maximale Auslastung
32 kb/s	52,387 %	97,084 %
320 kb/s	67,099 %	>100 %

Tabelle 5.1: Ergebnis der Messung ohne Hardware-Timer

lastung liegt bei ca. 67,1 % für einen mit einer Datenrate von 320 kb/s kodierten MPEG-Audio-Datenstrom. Bei der Messung zeigt sich auch eine weitere Schwäche des Messverfahrens. Liegt die Auslastung für einen Frame bei 100 % oder höher, so kann der genaue Wert der Auslastung nicht bestimmt werden. Dies liegt darin begründet, dass in diesem Fall die Schleife, in der das Zählen durchgeführt wird, nicht durchlaufen wird, als Wert für die ungenutzte Zeit also null geliefert wird. Dies ermöglicht jedoch keinerlei Rückschluss auf die Zeitspanne, um die die für die Dekodierung verfügbare Zeit überschritten wurde.

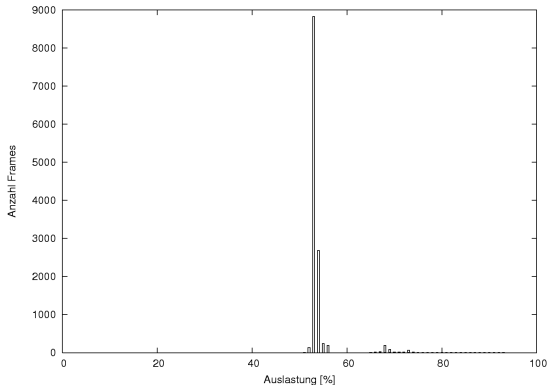

Abbildung 5.2: Histogramm der CPU-Auslastung für eine Datenrate von 32 kb/s

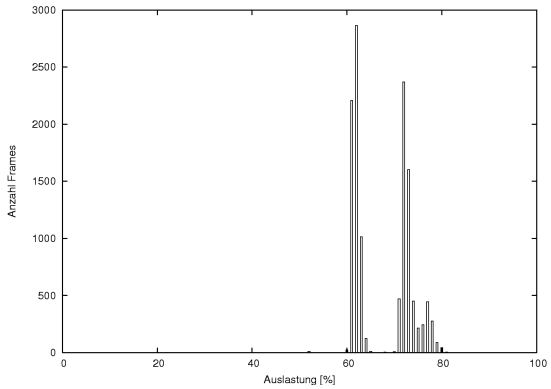

Abbildung 5.3: Histogramm der CPU-Auslastung für eine Datenrate von 320 kb/s

Abbildungen 5.2 und 5.3 stellen die Histogramme der Auslastung für die verschiedenen Bitraten dar. In der Histogrammdarstellung ist deutlich zu erkennen, dass die auftretenden Auslastungen nicht über einen Bereich gleichmäßig verteilt sind, sondern sich in

drei Gruppen aufteilen. Aufgrund der Achsenskalierung sind diese für den Datenstrom mit 320 kb/s besser zu erkennen, als für den Datenstrom mit 32 kb/s.

Im Histogramm für den Datenstrom mit 320 kb/s sind deutlich die Spitzen bei einer Auslastung von 62 %, 72 % sowie 77 % zu erkennen, wobei sich die Bereiche der beiden Gruppen mit höherer Auslastung überschneiden. Dem Histogramm ist weiterhin zu entnehmen, dass die Anzahl der Frames in der linken Gruppe in etwa der addierten Anzahl der Frames in den anderen beiden Gruppen entspricht. Die Einteilung in drei Gruppen und die Anzahl der jeweils zu den Gruppen gehörenden Frames ergibt sich aus der Pufferung der Eingabedaten.

In der Gruppe um 62 % liegen Frames, für die es nicht nötig war, den Eingabepuffer des Dekoders neu zu füllen, bei denen also keine Lesezugriffe auf die CF-Karte nötig waren. Die Größe der Frames kann mit Formel 2.1 bestimmt werden und beträgt für den Datenstrom entweder 1.044 oder 1.045 Bytes, je nach gesetztem Padding-Bit.

Da der Eingabepuffer des Dekoders eine Größe von 3.072 Bytes hat, muss der Puffer jeden zweiten Frame wieder aufgefüllt werden. Dies liegt darin begründet, dass nach dem Auffüllen des Puffers der *Header* des nächsten Frames immer am Beginn des Eingabepuffers liegt. Somit sind nach dem Wiederauffüllen zwei komplette Frames innerhalb des Puffers gespeichert, da die addierte Länge zweier Frames bei einer Bitrate von 320 kb/s 2.088, 2.089 oder 2.090 Bytes entspricht. Für die Dekodierung des dritten Frames, der nur teilweise innerhalb des Puffers gespeichert ist, ist es notwendig, Daten von der CompactFlash-Karte nachzuladen.

Diese Frames bilden die zweite Gruppe innerhalb des Histogramms bei einer Auslastung von ca. 72 %. Die Auslastung ist höher, da es für das Auffüllen des Frames zunächst nötig ist, eventuell noch innerhalb des Puffers vorhandene Daten an den Beginn des Puffers zu verschieben und den freien Platz innerhalb des Puffers durch Lesezugriffe auf das Dateisystem wieder zu füllen.

Die für den Zugriff auf das Dateisystem verwendete Bibliothek verwendet intern ebenfalls einen Puffer von konfigurierbarer Größe. In der Voreinstellung beträgt die Größe dieses zweiten Puffers 10 KiB. Die Frames, für die es sowohl nötig war, den Dekoderpuffer wie auch den Eingabepuffer wiederaufzufüllen, bilden die dritte Gruppe innerhalb des Histogramms bei einer Auslastung von ca. 77 %. Da das Größenverhältnis der beiden Puffer in etwa eins zu drei beträgt, enthält auch die zweite Gruppe circa dreimal so viele Frames wie die Gruppe mit der höchsten Auslastung. Da der Dateisystempuffer etwa dreimal so groß ist wie der Dekoderpuffer, ist es nur nötig, diesen für ein Drittel der Frames wieder aufzufüllen.

Auch im Histogramm des Bitstroms mit der niedrigeren Bitrate ist die Teilung in drei Gruppen zu erkennen, da jedoch die Größe der Frames nur ein Zehntel der Größe der Frames bei 320 kb/s beträgt, ist die erste Gruppe entsprechend größer. Bei einer Datenrate von 32 kb/s enthält der voll gefüllte Puffer jeweils 29 komplette Frames. Für die Deko-

dierung des 30. Frames ist es notwendig, den Puffer wiederaufzufüllen, daher beträgt die Anzahl der Frames in der ersten Gruppe in etwa das 29-fache der addierten Anzahl der Frames in den beiden anderen Gruppen. Das Zahlenverhältnis der beiden anderen Gruppen zueinander beträgt wiederum drei zu eins.

Die Messungen zeigen, dass eine Dekodierung eines MPEG-Audio-Datenstroms in Echtzeit auch bei einer reinen Software-Implementierung des Dekoders möglich ist. Es ist jedoch nötig, Ausgabepuffer mit einer Größe von mehreren Ausgabeframes vorzusehen, um die Ausgabe fortsetzen zu können, falls die Dekodierung des nächsten Frames nicht innerhalb der für die Ausgabe eines Frames zur Verfügung stehenden Zeit durchgeführt werden kann.

Die in diesem Abschnitt ausgewerteten Messungen dienten also nur der Überprüfung der Funktion des Dekoders und einer ersten Abschätzung hinsichtlich der Performance des Gesamtsystems. Sie stellen also ein Nebenergebnis dieser Arbeit dar.

5.1.2 Messung mit Timer

Um die Genauigkeit der Messungen zu verbessern und nicht nur die Gesamtauslastung bestimmen zu können, sondern auch die für die Berechnung der DCT benötigte Zeit, wurde für weitere Messungen ein Timer verwendet. Dieser ermöglicht es, genauere Messungen durchzuführen, als es mit dem vorherigen Messverfahren möglich gewesen wäre, und auch für Frames, bei denen die Auslastung über 100 % lag, diese genau zu bestimmen. Ein weiterer Vorteil ist, dass die Messergebnisse nicht mehr von der Annahme abhängig sind, dass die Ausführungszeiten einer Schleife oder Funktion durch Aufsummieren der Ausführungszeiten der einzelnen Assemblerinstruktionen bestimmt werden können. Somit dient die Messung mit einem Hardwaretimer auch der Überprüfung dieser Annahme.

Für die weiteren Messungen werden an dieser Stelle nur noch die Messergebnisse der Versuche mit einem Datenstrom mit einer Bitrate von 320 kb/s betrachtet, da die beobachteten Ergebnisse für die anderen Bitraten analog waren (Siehe Abschnitt 3.5.1).

Abbildung 5.4 zeigt die grafische Darstellung der durch die Messung mit dem Hardware-Timer gewonnenen Messdaten. Die durchschnittliche Auslastung beträgt 56,138 %, die maximale gemessene Auslastung 109,651 %. Für die Messungen wurde der Eingabepuffer auf 4.096 Byte vergrößert. Dies hat zur Folge, dass jeweils drei komplette Frames innerhalb des Puffers gespeichert sind. Das Verhältnis der Frames, bei denen kein Auffüllen des Puffers nötig ist, zu Frames, bei denen der Puffer aufgefüllt werden muss, beträgt somit zwei zu eins. Die Anzahl der Frames in der ersten Gruppe steigt also entsprechend an, was dem in Abbildung 5.5 dargestellten Histogramm zu entnehmen ist.

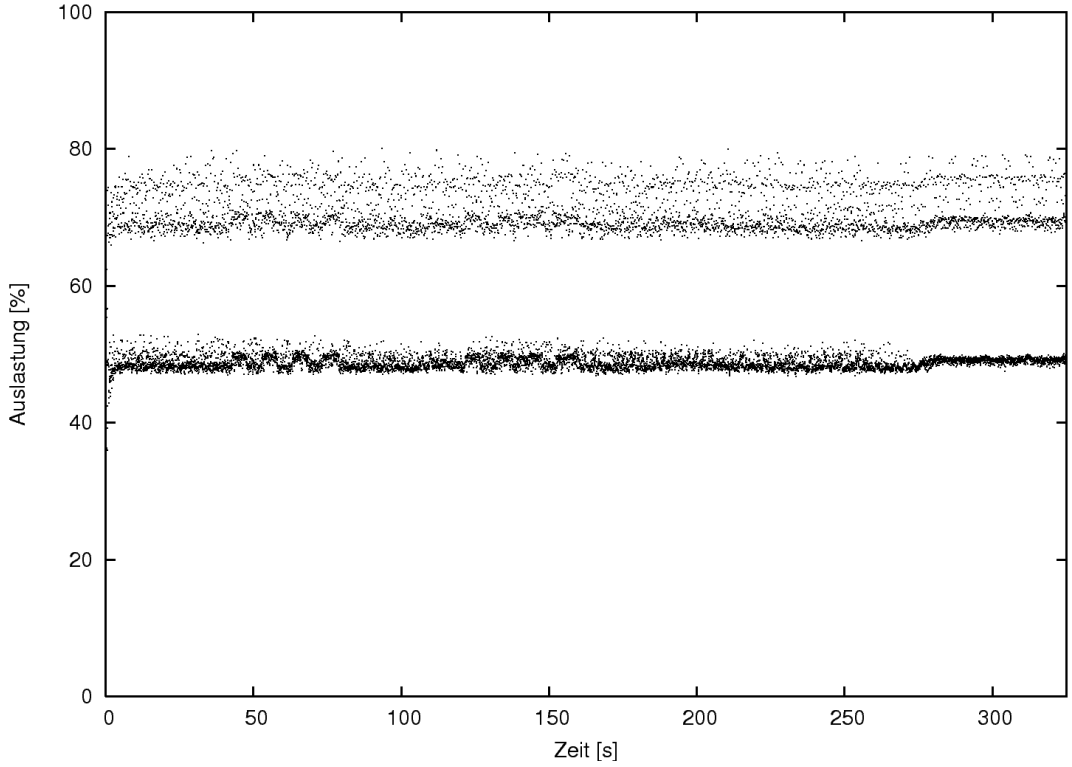

Abbildung 5.4: CPU-Auslastung für die Dekodierung (SW-DCT / Messung mit Timer)

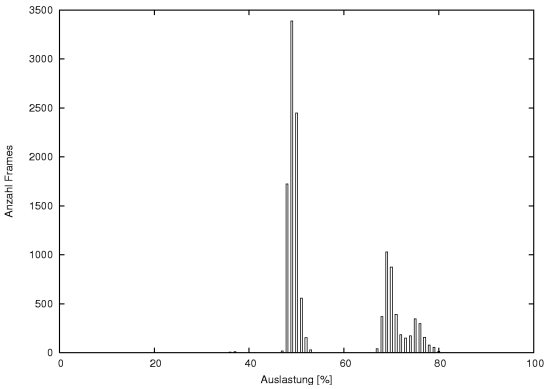

Abbildung 5.5: Histogramm der Auslastung für eine Datenrate von 320 kb/s

Der Vergleich der in den Abbildungen 5.3 und 5.5 dargestellten Histogrammen zeigt weiterhin, dass das Zentrum der Gruppen mit der geringsten Auslastung bei der Messung mit dem Hardware-Timer nach links verschoben ist. Bei der Messung ohne Timer liegt es bei einer Auslastung von 62 %, bei der Messung mit Timer liegt es bei einer Auslastung von 49 %. Da in dieser Gruppe jeweils die Frames liegen, für die es nicht notwendig war, den Eingabepuffer wiederaufzufüllen, kann die Verschiebung nicht auf die Veränderung der Größe des Eingabepuffers zurückgeführt werden.

Die Ursache für die Verschiebung liegt stattdessen in der Ungenauigkeit des ersten Messverfahrens. Dieses basierte auf der Annahme, dass es möglich ist, die innerhalb der Schleife verbrachte Zeit zu bestimmen, indem die Ausführungszeiten der Instruktionen pro Schleifendurchlauf aufsummiert werden.

Der Vergleich der Ergebnisse ergibt, dass statt der berechneten acht Takte pro Schleifendurchlauf im Durchschnitt circa 10,5 Takte pro Schleifendurchlauf benötigt werden. Daraus folgt auch, dass die mit dem ersten Messverfahren gemessene freie CPU-Zeit zu niedrig und damit die Auslastung zu hoch angenommen wird. Aufgrund der ungenauen Ergebnisse, die durch das Zählen der Schleifendurchläufe gewonnen werden, soll dieses Verfahren für die weiteren Messungen nicht mehr verwendet werden, stattdessen wird für die weiteren Messungen immer der Timer verwendet.

Um zu überprüfen, ob das Überschreiten der für die Dekodierung zur Verfügung stehenden Zeit bei einigen Frames durch die Dekodierung selbst, oder durch die Dateisystemzugriffe verursacht wird, wurde eine weitere Messung durchgeführt. Bei dieser Messung wurde der zu dekodierende Datenstrom vor dem Beginn der Dekodierung vollständig in den Speicher geladen. Somit ist es nicht notwendig, während der Dekodierung den Speicher wiederaufzufüllen.

Abbildung 5.6 zeigt die Ergebnisse der Messung. Wie zu erkennen ist, liegt die Auslastung niedriger als in den vorangegangenen Versuchen. Die auftretenden Auslastungen sind auf einen sehr viel schmaleren Bereich begrenzt, die maximale Auslastung beträgt 50,445 %, die durchschnittliche Auslastung 45,743 %. Die Auslastung variiert also kaum in Abhängigkeit vom zu dekodierenden Datenstrom. Lediglich die gewählte Bit- und Samplingrate haben einen größeren Einfluss auf die Auslastung des Dekoders.

Dies hat zur Folge, dass die beobachteten Überschreitungen der für die Dekodierung zur Verfügung stehenden Zeit durch das Lesen der Daten von der CompactFlash-Karte bedingt sein müssen. Die später durchgeführten Tests deuten auf eine Abhängigkeit von der eingestellten Größe des Dateisystem-Puffers hin. Dies scheint für einen kleinen Teil der Lesezugriffe reproduzierbar dazu zu führen, dass die Zugriffe auf die Datei signifikant länger benötigen, als in den übrigen Fällen.

Eine weitere Möglichkeit, die durch die Verwendung des Timers geschaffen wird, ist es, nicht nur die Gesamtperformance des Systems zu messen, sondern direkt die für die

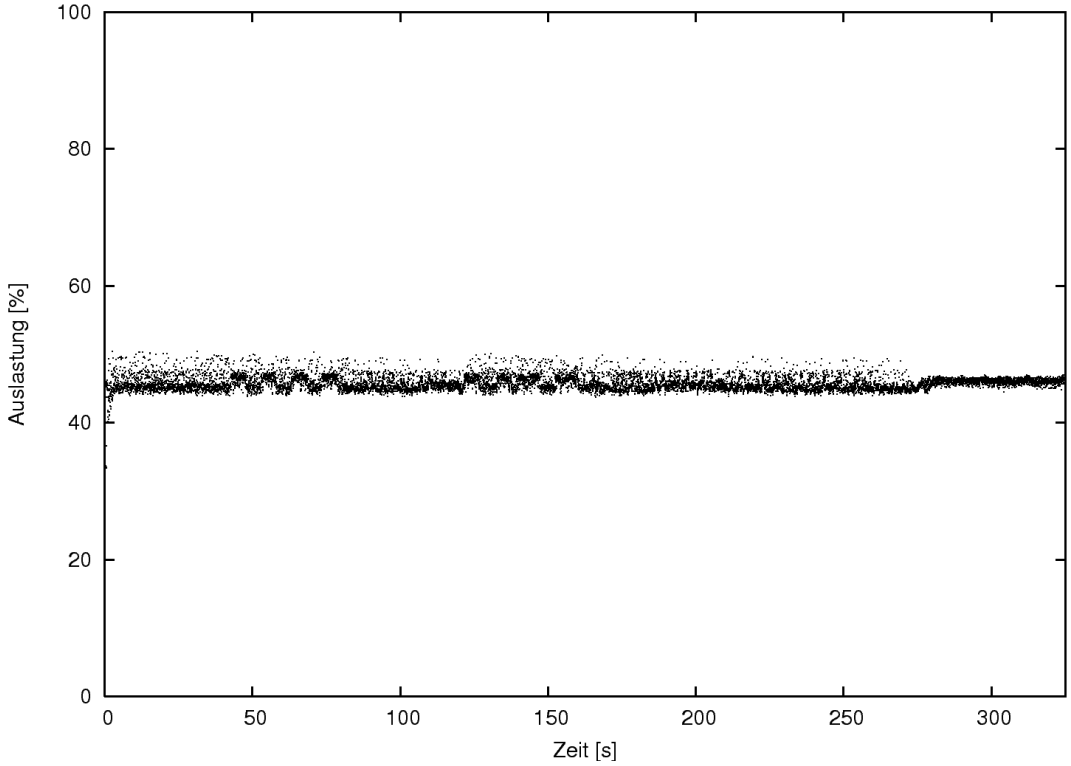

Abbildung 5.6: CPU-Auslastung bei vollständiger Speicherung im DDR-SDRAM

Ausführung der DCT benötigten Zeit. Hierzu wird der Timer direkt vor dem Aufruf der DCT gestartet und nach Beendigung des Funktionsaufrufs wieder gestoppt.

Die Messungen für 12.500 Frames ergaben eine minimale Ausführungszeit von 35,98 μs, sowie eine maximale Zeit von 39,83 μs. Die durchschnittliche Dauer, die ein DCT-Aufruf benötigt, beträgt 37,88 μs. Diese Werte werden als Vergleichswerte für die Implementierungen mit Hardware-DCT und OSSS-RMI-Protokoll verwendet.

5.1.3 Ressourcenbedarf

Die innerhalb des FPGAs implementierten Komponenten benötigen jeweils einen Teil der durch das FPGA bereitgestellten Ressourcen. Im Folgenden soll der Ressourcenbedarf der einzelnen Komponenten eingehender betrachtet werden. Da für die Software-Implementierung keine zusätzliche dedizierte Hardware verwendet wird, ist der Ressourcenbedarf dieser Implementierung geringer, als der der anderen Implementierungen.

Komponente	**Instanz**	**belegte Slices**
microblaze	microblaze_0	1.724
opb_v20	mb_opb	168
lmb_v10	ilmb	1
lmb_v10	dlmb	1
opb_mdm	debug_module	67
lmb_bram_if_cntlr	dmlb_cntlr	3
lmb_bram_if_cntlr	ilmb_cntlr	3
opb_uartlite	rs232_uart	52
opb_gpio	LEDs_positions	38
opb_gpio	Push_Buttons_Position	29
opb_sysace	SysACE_CompactFlash	135
mch_opb_ddr	DDR_SDRAM_64Mx32	947
opb_ac97_controller_ref	opb_ac97_controller_ref_0	121
opb_timer	opb_timer_0	271
opb_intc	opb_intc_0	87
lmb_bram	bram_block	0
util_vector_logic	sysclk_inv	1
util_vector_logic	clk90_inv	1
util_vector_logic	ddr_clk90_inv	1
dcm_module	dcm_0	0
dcm_module	dcm_1	1
—	Gesamt:	3.651

Tabelle 5.2: Ressourcenverbrauch der einzelnen verwendeten Komponenten

Tabelle 5.2 zeigt den Ressourcenverbrauch der einzelnen Komponenten. Wie zu erkennen ist, belegt das Design 3.651 von 10.752 Slices. Das entspricht 33,956 % der auf dem FPGA verfügbaren Slices.

5.2 Implementierung mit HW-DCT

Nachdem in den vorangegangenen Tests jeweils die Softwareimplementierung der DCT verwendet wurde, soll im Folgenden die Variante mit einer Hardwareimplementierung der DCT betrachtet werden. Für die Messungen wird dabei wiederum der Timer verwendet, um die Ausführungszeiten der DCT sowie die Gesamtauslastung des Systems zu bestimmen.

5.2.1 Vorüberlegungen

Die vorangegangenen Messungen ergaben, dass die Ausführung der Software-Implementierung der DCT durchschnittlich 37,88 μs, also 3.788 Takte bei einer Taktfrequenz von 100 MHz, benötigt. Die Simulation der Hardware-DCT-Komponente ergab eine Ausführungszeit von 1.034 Takten. Hierbei ist jedoch die für den Transport der Daten benötigte Zeit nicht enthalten. Somit verlängert sich die Zeit für den gesamten Aufruf der DCT im optimalen Fall um weitere 64 Takte, da der Transport der 32 Parameter und Rückgabewerte jeweils mindestens einen Takt pro Wert benötigt.

Von diesen Werten ausgehend kann eine erste Abschätzung der zu erwartenden Systemperformance erfolgen. Für die reine Softwareimplementierung lag die durchschnittliche Systemauslastung bei circa 56,138 %. Daraus ergibt sich, dass im Mittel die Dekodierung eines Frames 1.466.462 Takte benötigt.

$$cycles = f_{CPU} \cdot t_{frame} \cdot usage = 100\ MHz \cdot \frac{1.152}{44.100\ Hz} \cdot 0,56138 = 1.466.462$$

Aus den Messergebnissen der Softwareimplementierung und den Simulationsergebnissen der Hardware-DCT lässt sich so der Performancegewinn abschätzen. Die DCT wird pro Frame und Kanal 36 mal ausgeführt, daraus ergeben sich für die von der Software-DCT benötigten Takte pro Frame $3.788 \cdot 72 = 272.736$ Takte. Die HW-DCT benötigt pro Frame $(1.034 + 64) \cdot 72 = 79.056$ Takte. Dieser Wert stellt jedoch den *best-case* dar, da er eventuell durch den Speicherzugriff auftretende Latenzen nicht berücksichtigt. Der tatsächlich zu messende Wert wird also höher liegen.

Aus den berechneten Werten ergibt sich eine zu erwartende Verbesserung von $272.736 - 79.056 = 193.680$ Takten. Daraus lässt sich die Gesamtauslastung des Systems herleiten.

$$usage = \frac{cycles}{f_{CPU} \cdot t_{frame}} = \frac{1.466.462 - 193.680}{100.000.000 \cdot \frac{1.152}{44.100}} = 0.48724$$

Zu erwarten ist also eine Gesamtauslastung von circa 48,724 %. Dieser Wert würde eine mittlere Verbesserung im Vergleich zu der Softwareimplementierung von 13,21 % bedeuten.

5.2.2 Gesamtauslastung

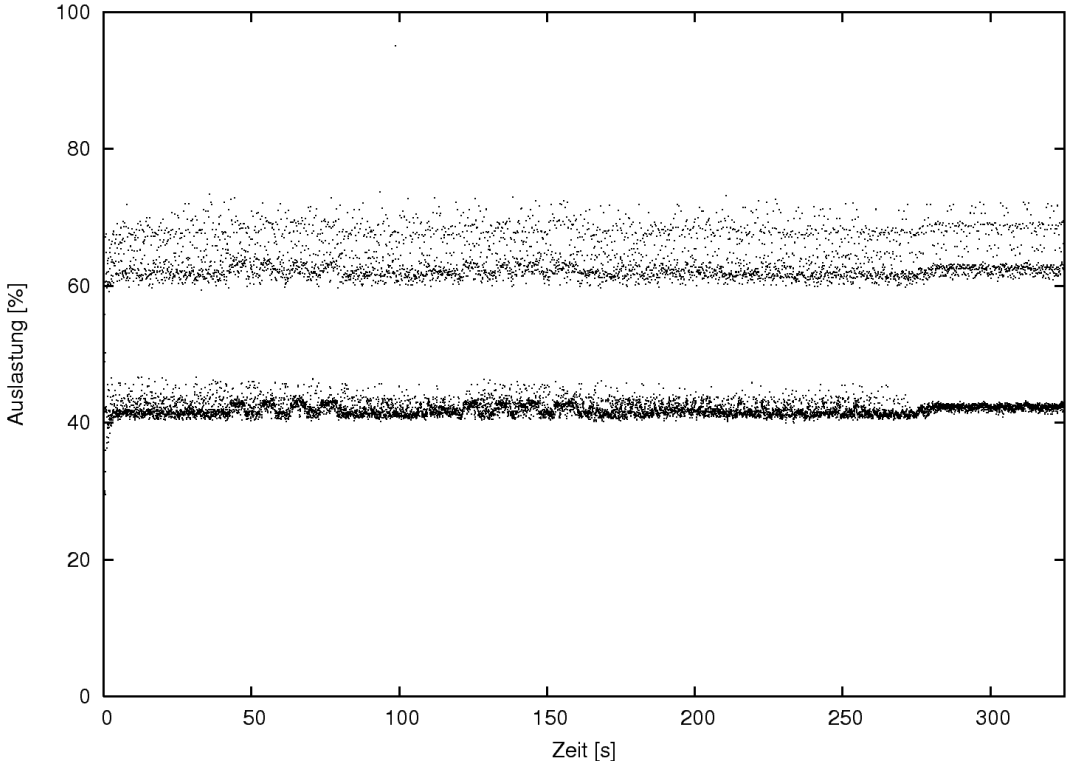

Abbildung 5.7: Auslastung für jeden Frame mit verwendeter HW-DCT

Nachdem die zu erwartende Verbesserung der Systemperformance bestimmt wurde, sollen nun die Ergebnisse der Messung ausgewertet werden. Abbildung 5.7 zeigt die Auslastung des Systems für jeden einzelnen Frame. Die Ergebnisse entsprechen in weiten Teilen denen der Versuche mit der Software-DCT. Die Auslastung ist jedoch erwartungsgemäß geringer, da die für die Berechnung der DCT benötigte Zeit bei der Hardware-Implementierung geringer ist als bei der Software-Implementierung. Die durchschnittliche Auslastung beträgt 49,322 %, die maximale Auslastung 102,536 %. Dies bedeutet sowohl für die durchschnittliche wie für die maximale Auslastung eine Verbesserung von circa sieben Prozentpunkten. Auch mit Verwendung der Hardware-DCT ist also eine

Echtzeitdekodierung aller Frames nicht möglich. Erst die Verwendung eines Ausgabepuffers entsprechender Größe ermöglicht eine störungsfreie Wiedergabe.

Durch Anpassung der Puffergröße in der verwendeten Dateisystem-Bibliothek ist es jedoch möglich, die Auslastung auf unter 100 % zu senken. Bei einer Verdoppelung der Puffergröße auf 20 KiB sinkt die maximale Auslastung auf 94 %, die durchschnittliche Auslastung sinkt gleichzeitig auf 46 %. Tabelle 5.3 zeigt die verschiedenen Auslastungen sowohl für die HW-DCT, als auch für die SW-DCT. Wie aus der Tabelle ersichtlich ist hat die gewählte Puffergröße einen deutlichen Einfluss auf die Systemperformance. Da die Strategie, nach der die verwendete Dateisystem-Bibliothek den Puffer verwendet, unbekannt ist, konnte die genaue Ursache für das beobachtete Verhalten nicht geklärt werden. Für die dieser Arbeit zugrunde liegende Fragestellung ist dieses Problem jedoch nicht von Bedeutung, da es nicht durch die Dekodierung selbst verursacht wird, sondern durch die verwendete Dateisystem-Bibliothek und somit auch keine Auswirkungen auf die Messung des Kommunikationsoverheads hat.

Puffergröße	SW-DCT		HW-DCT	
	Maximum	Durchschnitt	Maximum	Durchschnitt
1 KiB	106,783 %	54,434 %	99,761 %	47,352 %
10 KiB	109,651 %	56,138 %	102,536 %	49,322 %
20 KiB	100,641 %	52,995 %	94,009 %	46,245 %

Tabelle 5.3: Durchschnittliche und maximale Auslastungen in Abhängigkeit der gewählten Dateisystem-Puffergröße

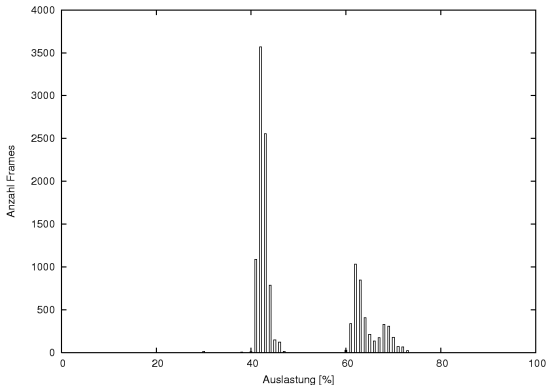

Abbildung 5.8: Histogramm der Auslastung bei Verwendung der HW-DCT

Abbildung 5.8 zeigt wiederum die Histogrammdarstellung der gemessenen Daten. Auch diese weist eine starke Ähnlichkeit zu Abbildung 5.5 auf. Im Histogramm ist ebenfalls die Aufteilung in die drei Gruppen zu erkennen, diese sind jedoch leicht nach links verschoben, da die Auslastung geringer ist. Die Verschiebung ist hierbei für alle Gruppen gleich, wie zu erwarten war, da die Verringerung der für die Dekodierung des Frames benötigten Zeit für jeden Frame, unabhängig von der Auslastung, gleich ist.

5.2.3 Messung der Ausführungszeit der HW-DCT

Es ist zu erkennen, dass die theoretisch vorausberechnete Reduktion der Auslastung nicht vollständig erreicht wurde. Dies liegt darin begründet, dass für den Datentransfer mehr Takte als angenommen benötigt werden. Im Folgenden soll mit Hilfe des Timers bestimmt werden, wieviel Zeit für die Ausführung der HW-DCT inklusive Datentransfer benötigt wird.

Hierzu wurde sowohl eine Messung der reinen Berechnungsdauer, wie auch eine Messung der Dauer der Berechnung inklusive der Datenübertragung vorgenommen. Somit ist es möglich, den Overhead für die Kommunikation zu bestimmen.

Die Messungen der Ausführungszeit ohne Kommunikation ergaben, dass die Komponente durchschnittlich 1.110 Takte für die Berechnung der DCT benötigt. Die Abweichung zwischen der durch die Simulation des VHDL-Modells bestimmten Ausführungszeit und der gemessenen Ausführungszeit ergibt sich einerseits durch die für das Starten und Anhalten des Timers benötigte Zeit und andererseits aus den Verzögerungen, die durch die Abfrage des DCT-Status über den OPB-Bus bedingt sind.

Die in der Simulation gemessenen Werte erfassen nur die Zeit vom Starten der DCT bis diese durch Löschen des *busy flag* im Statusregister den Abschluss der Berechnungen anzeigt. Im realen System vergehen jedoch zwischen zwei Register-Lesevorgängen einige Takte, so dass die Änderung des Registerinhalts erst beim nächsten Auslesen des Registers sichtbar wird.

In einer weiteren Messung wurde die benötigte Zeit für die Ausführung der DCT inklusive der Datenübertragung zwischen Microblaze und DCT-Komponente bestimmt. Die Messung ergab, dass durchschnittlich 1.948 Takte für die Berechnung der DCT inklusive Datenübertragung benötigt werden. Dieser Wert liegt um 850 Takte über dem für den *best-case* angenommenen Wert. Die Zeit, die für die Kommunikation, also die Übertragung der Daten zwischen dem Microblaze und der DCT-Komponente, benötigt wird, kann aus den gemessenen Werten bestimmt werden. Sie beträgt somit $1.948 - 1.110 = 838$ Takte für die Übertragung der jeweils 36 Parameter- und Rückgabewerte. Die Übertragung jedes einzelnen Wertes benötigt folglich circa zwölf Takte. Dies ist darauf zurückzuführen, dass für die Speicherung der Laufzeitvariablen der DDR-SDRAM verwendet wird und die Adressierung von Speicherstellen mit einer gewissen Latenz verbunden ist. Außerdem ist es immer dann, wenn ein Wert, der an die DCT gesendet werden soll, nicht im Cache des Microblaze enthalten ist, nötig, die Daten mehrmals über den OPB-Bus zu übertragen. Der Microblaze liest in einem solchen Fall zunächst das Datum aus dem Speicher, und überträgt es dann an die Komponente. Der gleiche Vorgang wiederholt sich mit den Rückgabewerten in umgekehrter Richtung.

5.2.4 Ressourcenverbrauch der HW-DCT

Da das System um eine Hardware-Komponente für die Berechnung der DCT erweitert wurde, vergrößert sich auch der Ressourcenbedarf des Designs. Die verwendete DCT-Komponente belegt 2.674 Slices. Ein großer Teil dieses Ressourcenverbrauchs wird durch die Speicherung der Parameter und Zwischenergebnisse in Registern verursacht, hierfür werden allein 2.048 Flipflops benötigt. Durch die Verwendung der RAM-Blöcke innerhalb des FPGAs könnte der Flächenbedarf der Komponente reduziert werden. Da dies jedoch keine Auswirkungen auf den Kommunikationsoverhead hätte, wurde von einer Optimierung hinsichtlich der belegten Fläche Abstand genommen.

Die Gesamtbelegung der verfügbaren Fläche beträgt somit für die Implementierung mit HW-DCT 6.325 der 10.752 Slices. Dies entspricht einer Auslastung von 58,826 %.

5.3 OSSS

Die Ergebnisse der vorherigen Auswertungen sollen nun verwendet werden, um einen Vergleich zwischen dem automatisch generierten Interface zur Hardware-Software-Kommunikation und der eigenen Implementierung durchzuführen. Hierzu wird wiederum der MPEG-kodierte Datenstrom dekodiert und die für die Dekodierung jedes Frames benötigte Zeit gemessen. Zusätzlich zu der gemessenen Zeit für die gesamte Dekodierung wird auch die Zeit gemessen, die für die Berechnung der DCT sowie den Transport der Daten benötigt wird.

5.3.1 Messung der Systemperformance

Während der Messungen zeigte sich, dass es aus Zeitgründen nicht möglich war, die Dekodierung der gesamten MPEG-Audio-Datei durchzuführen, da die Dekodierung jedes Frames circa 2,93 s dauerte. Dies hätte eine Gesamtdauer der Dekodierung von über zehn Stunden bedeutet. Die Dekodierung benötigt folglich mehr als das 100-fache der pro Frame zur Verfügung stehenden Zeit.

Im Durchschnitt über 2.350 Frames, also 169.200 Aufrufe der DCT, beträgt die benötigte Zeit für die Ausführung der DCT 4.077.637 Takte, also bei einer Taktfrequenz von 100 MHz etwa 40,78 ms. Die maximale Ausführungszeit beträgt 40,87 ms. Daraus ergibt sich, im Vergleich zu der SW-Implementierung eine um den Faktor 1.076 höhere Ausführungszeit. Verglichen mit der HW-DCT beträgt die Ausführungszeit sogar das 2.093-fache.

5.3.2 Ressourcenverbrauch der OSSS-Implementierung

Da die Interface-Logik für die durch die manuelle „Synthese“ erzeugte Implementierung des Shared Objects komplexer ist, als die der ursprünglichen Hardware-Implementierung, ist zu erwarten, dass der Flächenbedarf höher ist. Tatsächlich wird der Flächenbedarf vom EDK mit 2.678 Slices angegeben. Dies entspricht einer Zunahme von vier Slices im Vergleich zu der ersten HW-Implementierung der DCT.

Die Ursache für die geringe Zunahme des Flächenbedarfs liegt darin, dass durch die Änderung des Businterfaces nicht mehr zwei Speicherbereiche in den Adressraum des Microblaze eingeblendet werden, sondern die Komponente nur noch als ein Register erscheint. Dadurch kann der Flächenbedarf des Businterfaces verkleinert werden. Somit werden die Änderungen, die sich durch die komplexere Zustandslogik ergeben, fast vollständig kompensiert.

Die Größe des gesamten Designs vergrößert sich somit auf 6.329 Slices, dies entspricht einer Änderung im Vergleich zu der Implementierung mit HW-DCT um 0,063 %. Insgesamt sind daher 58,863 % der auf dem FPGA verfügbaren Slices belegt.

5.4 RMI-Protokoll mit optimierter Serialisierung

Aufgrund der schlechten Performance des automatisch generierten Kommunikationsinterfaces wurden die Ursachen hierfür näher bestimmt. Es kommen drei verschiedene Gründe für die Verschlechterung der Leistung in betracht:

1. Die Verwendung von C++ anstelle von C für die Implementierung der Callbackfunktionen und des Hauptprogramms.

2. Die Serialisierung und Deserialisierung der zu übertragenden Daten.

3. Das verwendete Kommunikationsprotokoll.

Um auszuschließen, dass die Verwendung von C++ zu der Verschlechterung führt, wurde ein Vergleichstest durchgeführt, bei dem statt der OSSS-RMI-Implementierung wieder die Software-Implementierung verwendet wurde.

Die Auswertung ergab, dass sich bei Verwendung von C++ statt C die Ausführungszeit für einen Aufruf der DCT-Routine von durchschnittlich 3.788 Takten auf 4.281 Takte erhöht. Dies entspricht einer Verlängerung um 13,015 %. Da der beobachtete Performanceverlust jedoch um einige Größenordnungen höher ist, kann dieser nicht auf die Verwendung von C++ zurückgeführt werden.

Um zu überprüfen, ob die Verwendung des Protokolls für die Verschlechterung der Performance verantwortlich ist, wurde ein weiterer Test durchgeführt. Bei diesem wurden die für die Serialisierung und Übertragung der Daten verwendeten Routinen ausgetauscht. Die neu implementierten Routinen sind insofern vereinfacht, als für die Serialisierung nicht mehr auf Konstrukte der C++ Standard Template Library, wie zum Beispiel Vektoren, zurückgegriffen wird. Für das Hauptprogramm und die Callback-Funktionen wird jedoch weiterhin C++ verwendet. Das Kommunikationsprotokoll zwischen Prozessor und DCT-Komponente bleibt unverändert.

Die Messung ergab, dass sich die für den Aufruf der DCT benötigte Zeit von 1.948 Takten auf 2.339 Takte verlängerte. Dies entspricht einer Steigerung von 20,072 %. Diese enthält, neben dem durch die Verwendung von C++ bedingten Overhead, auch den durch das verwendete Protokoll verursachten Overhead.

Verwendete DCT	mittlere Ausführungsdauer [in Takten]	relativ zur SW-DCT
SW	3.788	100 %
HW	1.948	51,426 %
OSSS-RMI	4.077.637	107.646,172 %
SW-C++	4.281	113,015 %
HW-C++	2.339	61,748 %

Tabelle 5.4: Ausführungszeiten der verschiedenen DCT-Implementierungen

Wie Tabelle 5.4 zu entnehmen ist, verlängert sowohl die Verwendung von C++, wie auch die Verwendung des RMI-Protokolls die Ausführungszeiten der DCT-Routinen. Verschlechterungen in der Größenordnung, wie sie beim Einsatz des OSSS-Serialisierungsverfahrens beobachtet wurden, sind jedoch nicht gemessen worden. Daraus folgt, dass die Ursache für die Verschlechterung der Performance in dem verwendeten Serialisierungsverfahren liegt. Der signifikanteste Unterschied zwischen dem ursprünglichen OSSS-Serialisierungsverfahren und der optimierten Variante ist die intensive Verwendung von Vektor-Datenstrukturen im Serialisierungsverfahren von OSSS.

Die gewonnenen Ergebnisse weisen darauf hin, dass die Verwendung von Datenstrukturen aus der STL den größten Anteil an der Verschlechterung der Performance bei der Verwendung des OSSS-RMI-Protokolls hat. Die Ergebnisse der optimierten Implementierung der Serialisierung zeigen, dass die Verwendung des RMI-Protokolls nur einen relativ geringen Einfluss auf die Effizienz des Gesamtsystems hat. Das Serialisierungsverfahren hingegen bedarf einer Überarbeitung.

In einer Variante, die auf die Verwendung der Vektor-Datenstruktur verzichtet, könnte das automatisierte Verfahren zum Erzeugen des Kommunikationsinterfaces den zeitaufwändigen manuellen Entwurf ersetzen. Die gesteigerte Designeffizienz würde den relativ geringen Verlust an Systemperformance kompensieren.

6 Fazit

In diesem Kapitel werden die Ergebnisse der durchgeführten Analyse zusammengefasst und die daraus gewonnenen Erkenntnisse dargelegt. Hierzu werden die Ergebnisse der Arbeit zunächst in Hinblick auf die drei Zielgrößen, Designeffizienz, Flächenbedarf und Effizienz des entworfenen Systems, betrachtet. Im Anschluss daran wird ein Gesamtfazit gezogen werden.

6.1 Designeffizienz

Da die Designeffizienz nur schwer quantifizierbar ist, kann die Bewertung der Designeffizienz im Rahmen dieser Arbeit nur in sehr eingeschränktem Umfang erfolgen. Für eine genauere Untersuchung wäre es notwendig, eine größere Anzahl verschiedenartiger Designs und die für deren Umsetzung benötigte Zeit zu betrachten. Dadurch wäre es möglich, statistische Aussagen über die Effizienz der von OSSS postulierten Designmethodik im Vergleich zu anderen Ansätzen zu treffen.

Aufgrund der Verwendung von OSSS als einheitliche Sprache sowohl für den Entwurf der Soft-, als auch der Hardware bietet der OSSS Ansatz jedoch besonders im Vergleich zum traditionellen Entwurf, bei dem Hard- und Software in verschiedenen Sprachen entwickelt werden, klare Vorteile. Es entfällt beispielsweise die zeitaufwändige und fehlerträchtige Reimplementierung der Hardwarekomponenten während des Übergangs von einer ausführbaren Systembeschreibung zu einem ersten Hardwaremodell.

Auch im Vergleich zu einem auf SystemC basierenden Entwurf bietet OSSS Vorteile, da der Entwickler bei der komplexen Aufgabe der Verfeinerung des Entwurfs durch die automatische Generierung der Kommunikationsinterfaces unterstützt wird. Weiterhin ermöglicht OSSS verglichen mit SystemC die automatisierte Synthese des Systems von einer höheren Abstraktionsebene, so dass es nicht notwendig ist, das System manuell so weit zu verfeinern, wie dies bei SystemC der Fall ist.

Durch die Verwendung eines automatisch generierten Kommunikationsinterfaces ist es außerdem möglich, mit geringem Aufwand die Implementierung des für die Kommunikation verwendeten Kanals auszutauschen, um so verschiedene Kommunikationsstrukturen zu evaluieren. Das Konzept der Shared Objects bietet dem Entwickler zudem eine einfa-

che Möglichkeit, Funktionen in Hardware auszulagern, ohne dass dies große Änderungen an der Software bedingen würde.

Somit kann abschließend festgestellt werden, dass OSSS den Designvorgang für den Entwickler vereinfacht und daher auch eine Steigerung der Produktivität und der Designeffizienz zu erwarten ist.

6.2 Ressourcenverbrauch

Im Folgenden sollen die in Kapitel 5 gewonnenen Daten zum Ressourcenverbrauch des Designs zusammengefasst werden und die Auswirkung der Verwendung von OSSS auf den Verbrauch an FPGA-Ressourcen und Speicher betrachtet werden. Dies kann jedoch auch nur mit eingeschränkter Gültigkeit erfolgen, da die „Synthese“ des Shared Objects manuell erfolgte und somit die Implementierung nicht vollständig einer automatisch generierten entspricht. Weiterhin handelt es sich bei dem im Rahmen dieser Arbeit betrachteten Shared Object um eine vereinfachte Version ohne Scheduler und *Guard Evaluator*. Diese beiden im allgemeinen Falle eines Shared Objects vorhandenen Blöcke können also nicht in die Betrachtung des Ressourcenverbrauchs mit einbezogen werden.

6.2.1 Auswirkungen auf den Flächenbedarf

Die während der Synthese gewonnenen Daten zum Ressourcenverbrauch ergeben für die Implementierung mit dem eigenen Kommunikationsinterface einen Flächenverbrauch auf dem FPGA von 6.325 Slices. Im Vergleich dazu steigt die von der OSSS-Implementierung belegte Fläche auf 6.329 Slices. Dies entspricht einer Zunahme von vier Slices oder 0,063 % in Bezug zu der ursprünglichen Implementierung. Somit zeigt sich, dass der durch die OSSS-Implementierung verursachte zusätzliche Flächenverbrauch gering ist.

6.2.2 Auswirkungen auf die Größe des ausführbaren Programms

Die Untersuchung der Größe des ausführbaren Programms ergibt, für die Implementierung in C eine Größe von 93,129 KiB. Bei der Verwendung von C++ steigt die Größe des Programms auf 393,345 KiB. Diese Zunahme ist zu einem Teil durch die Verwendung der C++-Bibliotheken zur Standardein- und ausgabe bedingt.

Für die OSSS-Implementierung steigt die Größe des Programms auf 1.755,055 KiB. Die Änderung der Größe des ausführbaren Programms ist mit einer Zunahme von 1.784,542 %, im Vergleich zu der C-Implementierung, und 322,473 %, im Vergleich

zur C++-Implementierung, groß. Diese Zunahme ist auf die Verwendung der OSSS- und SystemC-Bibliotheken zurückzuführen. Durch die Verwendung einer optimierten Bibliothek könnte die Größe des Programms jedoch reduziert werden.

Festzustellen ist folglich, dass die Verwendung von OSSS einen höheren Verbrauch an Ressourcen, sowohl bezogen auf die Fläche, als auch auf die benötigten Kapazität des Programmspeichers hat. Bezogen auf die Fläche ist dieser zusätzliche Ressourcenverbrauch jedoch relativ gering. In Bezug auf den Speicherbedarf ist dieser zusätzliche Speicherverbrauch relevant, könnte jedoch durch den Einsatz einer optimierten OSSS-SW-Bibliothek verringert werden.

6.3 Effizienz des Systems

In diesem Abschnitt soll die Bewertung der in Kapitel 5 ausgewerteten Messdaten erfolgen. Wie dort schon diskutiert verursacht die Verwendung des von OSSS benutzten Verfahrens einen drastischen Rückgang der Systemperformance. Verglichen mit der nicht auf OSSS-basierenden Version verlängert sich die für einen Aufruf benötigte Zeit um mehr als den Faktor 2.000. Die daraus resultierende Verringerung der Gesamtperformance des Systems ist nicht akzeptabel.

Die angestellten Untersuchungen ergaben, dass das Problem nicht durch die Verwendung des OSSS-Kommunikationsprotokolls selbst hervorgerufen wird, sondern durch das von OSSS verwendete Verfahren zur Serialisierung der zu übertragenden Objekte. Insbesondere die Verwendung der Datenstrukturen aus der C++ *Standard Template Library* während der Serialisierung verursacht den starken Performance-Rückgang.

Die Ergebnisse aus Abschnitt 5.4 zeigen, dass bei der Verwendung eines optimierten Serialisierungsverfahrens der Kommunikationsoverhead drastisch reduziert werden kann. Die Messung ergab eine Verlängerung der für die Berechnung der DCT-benötigten Zeit um circa 20 % im Vergleich zu der Implementierung ohne das RMI-Interface. Diese gemessene Verlängerung der Zugriffszeit beinhaltet zusätzlich zu dem durch das OSSS-RMI-Protokoll verursachten den durch C++ verursachten Overhead[ISO06]. Im Vergleich zu einem C++-Design fällt die Verringerung der Performance also geringer aus.

Dieses Ergebnis zeigt, dass das RMI-Verfahren, wie es von OSSS verwendet wird, nicht generell untauglich für die Verwendung beim HW/SW-Co-Design ist, sondern die beobachteten Probleme auf die gewählte Implementierung zurückzuführen sind.

6.4 Weitere Erkenntnisse

Ein weiteres Ergebnis der im Rahmen dieser Arbeit angestellten Untersuchungen war, dass es möglich ist, eine C-Software-Bibliothek in ein OSSS-Design zu integrieren, ohne dass die Bibliothek mit einem C++-Kompiler übersetzt werden muss. Dies ist jedoch aufgrund der Objektorientierung von C++ mit bestimmten Schwierigkeiten (siehe Kapitel 4.5.3) verbunden. Die Möglichkeit zur Weiterverwendung bestehender C-Bibliotheken ist besonders für so genannten *legacy code*, also Bibliotheken die nicht mehr gewartet werden und nicht im Quelltext vorliegen, von Bedeutung.

Eine weitere Erkenntnis ist, dass der Microblaze mit der reinen Dekodierung des Datenstroms nur etwa zur Hälfte ausgelastet ist, eine Echtzeitdekodierung somit prinzipiell auch bei einer reinen Softwareimplementierung möglich ist. Dies war für ein datenflussdominiertes System wie einen MPEG-Audio-Dekoder nicht zu erwarten gewesen. Möglich wurde die Dekodierung jedoch erst durch die gesteigerte Effizienz des in den Microblaze integrierten Multiplizierers, da so die Latenzzeit für Multiplikation um den Faktor drei gesenkt wurde.

6.5 Abschlussbetrachtung

Als abschließendes Fazit dieser Arbeit kann festgestellt werden, dass die Verwendung von OSSS den Designvorgang eines eingebetteten Systems vereinfacht und ihn somit effizienter gestaltet. Das automatisch generierte Interface zur HW/SW-Kommunikation selbst verursacht nur einen relativ geringen Overhead sowohl im Hinblick auf die Performance, als auch den Flächenbedarf. Das für die Kommunikation verwendete Serialisierungsverfahren ist jedoch so rechenzeitaufwändig, dass es in der derzeitigen Form nicht geeignet ist, für die Implementierung eines Designs verwendet zu werden.

Ein weiteres Problem stellt die nicht unerhebliche Vergrößerung des ausführbaren Programms dar. Die Vergrößerung des Programms kann durch die Ersetzung der verwendeten SystemC- und OSSS-Bibliotheken durch speziell für die Verwendung in Software-Tasks angepasste Versionen reduziert werden.

Somit ergibt sich als Fazit aus der durchgeführten Auswertung, dass die Steigerung der Designeffizienz durch den Einsatz der OSSS-Methodik den durch das automatisch generierte Kommunikationsinterface verursachten Overhead nicht rechtfertigt. In einer optimierten Fassung der OSSS-Bibliothek, in der der Aufwand für die Serialisierung reduziert wird, würde der Overhead auf ein durch die gesteigerte Designeffizienz gerechtfertigtes Maß reduziert werden. Das Ziel für zukünftige Versionen sollte also sein, einerseits den Aufwand für die Serialisierung zu reduzieren und andererseits die Größe der Software-Bibliothek zu verkleinern.

Anhang A

Verwendete Werkzeuge und Komponenten

A.1 Software

Software	Version
Xilinx EDK	8.2.02i
Xilinx ISE	8.2.03i
Lame	3.97

A.2 Bibliotheken

Bibliothek	Version
MAD-Library	0.15.1b
OSSS	2.0
SystemC	2.1.v1
standalone	1.00.a
xilfatfs	1.00.a

A.3 IP-Komponenten

Komponente	Version
Microblaze	5.00.c
opb_v20	1.10.c
lmb_v10	1.00.a
opb_mdm	2.00.a
lmb_bram_if_cntlr	2.00.a
opb_uartlite	1.00.b
opb_gpio	3.01.b
opb_sysace	1.00.c
mch_opb_ddr	1.00.c
opb_ac97_controller_ref	1.00.a
opb_timer	1.00.b
opb_intc	1.00.c
bram_block	1.00.a
util_vector_logic	1.00.a
dcm_module	1.00.a

Glossar und Abkürzungen

AC97 Audio Codec '97, Spezifikation einer Audioschnittstelle

AC-Link Bezeichnung für die Verbindung zwischen AC97-Codec und AC97 Digital Controller

ASIC Application Specific Integrated Circuit

ATA Advanced Technology Attachment

BRAM Block RAM

CD Compact Disc

CISC Complex Instruction Set Computing, Designphilosophie für Prozessoren

Co-Simulation Eine gemeinsame Simulation heterogener Systeme, zum Beispiel Software und Hardware

CPI Cycles per Instruction

DAC Digital Analog Converter

DC97 AC97 Digital Controller

DCM Digital Clock Manager

DCT Discrete Cosine Transform

DRAM Dynamic Random Access Memory

EDK Embedded Development Kit, Eine von Xilinx angebotene Werkzeugsammlung für den Entwurf eingebetteter Systeme

EET Estimated Execution Time

FAT File Allocation Table, Name eines von Mircosoft verwendeten Dateisystems

FIFO First In - First Out

FPGA Field Programmable Gate Array

FSL Fast Simplex Link

ICODES Interface and Communication based Design of Embedded Systems

ID Identifier

IEC International Electrotechnical Commission

IP Intellectual Property

ISO International Organization for Standardization

JPEG Joint Photographic Experts Group

JTAG Joint Test Action Group

Lame Rekursives Akronym für „LAME Ain't an MP3 Encoder“, ein Open Source MPEG-Audio-Layer3 Enkoder

LCD Liquid Crystal Display

LED Light Emitting Diode

LMB Local Memory Bus

MAD MPEG Audio Decoder, Name einer Dekoderbibliothek

MDCT Modified Discrete Cosine Transform

MHS Microprocessor Hardware Specification

MPEG Motion Picture Experts Group

MUX Multiplexer

OPB On-Chip Peripheral Bus

OSI Open Systems Interconnection

OSSS Oldenburg System Synthesis Subset

PCM Pulse Code Modulation

PS/2 Personal System/2, eine Produktreihe von IBM

RET Required Execution Time

RISC Reduced Instruction Set Computing, Designphilosophie für Prozessoren

RMI Remote Method Invocation

RS-232 eigentlich ANSI/EIA/TIA-232-F-1997, ein Standard für eine serielle Schnittstelle

RTL Register Transfer Level

Slice Konfigurierbarer Block innerhalb eines FPGAs, bestehend aus zwei Look-Up-Tables und zwei Flipflops

SO Shared Object

SoC System on Chip

SRAM Static Random Access Memory

STL Standard Template Library

SystemC Eine auf C++ basierende Hardwarebeschreibungssprache

UART Universal Asynchronous Receiver Transmitter

UCF User Constraints File

USB Universal Serial Bus

Verilog Eine Hardwarebeschreibungssprache

VHDL VHSIC Hardware Description Language, eine Hardwarebeschreibungssprache

VHSIC Very High Speed Integrated Circuit

XPS Xilinx Platform Studio, eine integrierte Entwicklungsumgebung für eingebettete Systeme

Literaturverzeichnis

[CG03] Cai, Lukai und Daniel Gajski: *Transaction level modeling: an overview.* In: *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Seiten 19–24, New York, NY, USA, 2003. ACM Press. 6

[Gaj96] Gajski, Daniel D.: *Principles of digital design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 3, 4

[GGS06] Grabbe, Cornelia, Kim Grüttner und Thorsten Schubert: *Software Synthesis Technology for Hardware/Software Interfaces & Hardware Synthesis Technology for Hardware/Software Interfaces.* ICODES Deliverable D27&D28, August 2006. XIII, 8, 11, 12, 13

[GLMS02] Grötker, Thorsten, Stan Liao, Grant Martin und Stuart Swan: *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002. 3

[GNOS03] Grimpe, Eike, Wolfgang Nebel, Frank Oppenheimer und Thorsten Schubert: *Object-Oriented Hardware Design and Synthesis Based on SystemC 2.0.* In: *SystemC : Methodologies and Applications. - Boston u.a.: Kluwer*, Seiten 217–246, 2003. 9

[GO03] Grimpe, Eike und Frank Oppenheimer: *Extending the SystemC synthesis subset by object-oriented features.* In: *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Seiten 25–30, New York, NY, USA, 2003. ACM Press. 6

[Grü04] Grüttner, Kim: *HW/SW-Kommunikation zwischen einem MPEG Audio Decoder(MAD) und einer PCI-FPGA-Karte*, Februar 2004. 33

[GZD97] Gajski, Daniel, Jianwen Zhu und Rainer Dömer: *Essential Issues in Codesign.* In: Staunstrup, J. und W. Wolf (Herausgeber): *Hardware / Software Co-Design - Principles and Practice*, Kapitel 1, Seiten 1–45. Kluwer Academic Publishers, 1. Auflage, 1997. 6

[GZD+00] Gajski, Daniel D., Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer und Shuqing Zhao: *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000. 9

[Hou87] Hou, Hsieh: *A fast recursive algorithm for computing the discrete cosine transform*. IEEE Transactions on Acoustics, Speech, and Signal Processing, 35(10):1455 – 1461, Oktober 1987. 33

[IBM] IBM: *On-Chip Peripheral Bus, Architecture Specifications, Version 2.1*. (`http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/9A7AFA74DAD200D087256AB30005F0C8/$file/OpbBus.pdf`). 30

[ICO] *ICODES Project Description*. (`http://icodes.offis.de/`). 1

[ISO06] ISO/IEC: *TR 18015. Information technology - Programming languages, their environments and system software interfaces - Technical Report on C++ Performance*. Technischer Bericht, International Organization for Standardization, 2006. 87

[JRV+97] Jerraya, A.A., M. Romdhani, C.A. Valderama, Ph. Le Marrec, F. Hessel, G.F. Marchioro und J.M. Daveau: *Languages for System-Level Specification and Design*. In: Staunstrup, J. und W. Wolf (Herausgeber): *Hardware / Software Co-Design - Principles and Practice*, Kapitel 7, Seiten 235–262. Kluwer Academic Publishers, 1. Auflage, 1997. 6

[KOSK+01] Kuhn, T., T. Oppold, C. Schulz-Key, M. Winterholer, W. Rosenstiel, M. Edwards und Y. Kashai: *Object oriented hardware synthesis and verification*. In: *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, Seiten 189–194, New York, NY, USA, 2001. ACM Press. 9

[KRK98] Kuhn, T., W. Rosenstiel und U. Kebschull: *Object Oriented Hardware Modeling and Simulation Based on Java*. In: *International Workshop on IP Based Synthesis and System Design, Grenoble, France*, 1998. 9

[LAM] LAME Project, The: *LAME MP3 Encoder*. (`http://lame.sourceforge.net/index.php`). 39

[Lee84] Lee, Byeong G.: *A new algorithm to compute the discrete cosine Transform*. IEEE Transactions on Acoustics, Speech, and Signal Processing, 35(6):1243– 1245, Dezember 1984. 33

[MPE92] *ISO/IEC IS11172-3. Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s-Part 3: Audio.*, 1992. 16, 21

[MPE95] *ISO/IEC IS11172-4. Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s-Part 4: Compliance testing.*, 1995. 39, 40

[MWE01] Micheli, Giovanni De, Wayne Wolf und Rolf Ernst: *Readings in Hardware/Software Co-Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 4

[Nat] National Semiconductor: *LM4550 - AC'97 Rev 2.1 Multi-Channel Audio Codec with Stereo Headphone Amplifier, Sample Rate Conversion and National 3D Sound.* (`http://cache.national.com/ds/LM/LM4550.pdf`). 26, 32

[Nie98] Niemann, Ralf: *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 1998. XIII, 4

[ODE] *Homepage of the ODETTE-Project.* (`http://odette.offis.de/`). 1

[Ope] Open SystemC Initiative: *Open SystemC Initiative Homepage.* (`http://www.systemc.org`). 3

[Pan01] Panda, Preeti Ranjan: *SystemC: a modeling platform supporting multiple design abstractions.* In: *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, Seiten 75–80, New York, NY, USA, 2001. ACM Press. XIII, 5, 6

[Ros97] Rosenstiel, Wolfgang: *Prototyping and Emulation.* In: Staunstrup, J. und W. Wolf (Herausgeber): *Hardware / Software Co-Design - Principles and Practice*, Kapitel 3, Seiten 75–112. Kluwer Academic Publishers, 1. Auflage, 1997. 14

[Ruc05] Ruckert, Martin: *Understanding MP3.* SpringerVerlag, 2005. 16

[SKWS+04] Schulz-Key, C., M. Winterholer, T. Schweizer, T. Kuhn und W. Rosenstiel: *Object-oriented modeling and synthesis of SystemC specifications.* In: *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, Seiten 238–243, Piscataway, NJ, USA, 2004. IEEE Press. 9

[Str00] Stroustrup, Bjarne: *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 3

[Und] Underbit Technologies, Inc.: *MAD: MPEG Audio Decoder.* (`http://www.underbit.com/products/mad/`). 24, 34, 48

[Xila] XILINX, INC.: *Virtex-4 Family Overview*. (http://direct.xilinx.com/bvdocs/publications/ds112.pdf). 26

[Xilb] XILINX, INC.: *MicroBlaze Processor Reference Guide, Embedded Development Kit 8.2i*. (http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf). 14, 26, 28, 43, 53

[Xil02] XILINX, INC.: *System ACE - CompactFlash Solution*. (http://direct.xilinx.com/bvdocs/publications/ds080.pdf), April 2002. 31

[Xil05] XILINX, INC.: *OPB Timer/Counter (v1.00b)*. (http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_timer.pdf), Dezember 2005. 32

[Xil06a] XILINX, INC.: *Embedded System Tools Reference Manual - Embedded Development Kit, EDK 8.2i*. (http://www.xilinx.com/ise/embedded/edk82i_docs/est_rm.pdf), Juni 2006. 14, 48

[Xil06b] XILINX, INC.: *OPB Double Data Rate (DDR) Synchronous DRAM (SDRAM) Controller (v2.00b)*. (http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_ddr.pdf), März 2006. 31