Architekturkonzepte für prozessorbasierte MPEG Videodecoder mit Schwerpunkt für mobile Anwendungen

von
Diplom-Ingenieur
Christian Benno Stabernack
aus Berlin

Von der Fakultät IV- Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Thomas Sikora
Gutachter:	Prof. Dr.-Ing. Hans-Ulrich Post
Gutachter:	Prof. Dr.-Ing. habil. Djamshid Tavangarian

Tag der wissenschaftlichen Aussprache: 17.12.2004

Berlin 2005
D83

Danksagung

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fraunhofer Institut Nachrichtentechnik Heinrich-Hertz-Institut in der Abteilung Bildsignalverarbeitung.

Ich danke Prof. Dr. Ing. Hans-Ulrich Post, der sich der Thematik meiner Arbeit angenommen und durch wertvolle Hinweise und Diskussionen zum Gelingen der Arbeit beigetragen hat. Des Weiteren danke ich Prof. Dr. Ing. Djamshid Tavangarian, der sich als Zweitgutachter zur Verfügung stellte und Prof. Dr. Ing Thomas Sikora für die Übernahme des Vorsitzes der Prüfungskommission.
Weiterhin möchte ich mich bei meinem Abteilungsleiter Dr. Ralf Schäfer für die Unterstützung meiner Bestrebungen und bei allen Kollegen der Abteilung für die hilfreichen Anregungen, die meine Arbeit begleitet haben, bedanken.

Besonderer Dank gilt meiner lieben Frau Carmen, die mir wesentlichen moralischen Beistand gewährt hat.

Zusammenfassung

Mobile Endgeräte basieren z. Zt. auf Systemen (System on a Chip), deren Hauptfunktionalität durch den Einsatz entsprechender Software auf eingebetteten Prozessoren gebildet wird.
Aufgrund der immer kürzeren Innovationszyklen kommt der softwarebasierten Umsetzung von Applikationen für eingebettete Systeme damit eine steigende Bedeutung zu.
Im Bereich der mobilfunkgestützten Anwendungen sind in zunehmendem Maße auch Multimediaapplikationen vorzufinden, die bis vor kurzem noch leistungsfähigen Systemen, wie z.B. Desktop-PCs oder dedizierten Implementierungen in Form spezieller ASICs, vorbehalten waren. Hierzu zählen auch Anwendungen, die auf entsprechenden Standards basierend, die echtzeitfähige Übertragung von Bewegtbilddaten unterstützen, wie z.B. Videotelefonie oder Broadcastdienste.
Dem hohen Rechenleistungsbedarf echtzeitkritischer Multimediaapplikationen stehen hierbei jedoch die relativ leistungsschwachen Prozessoren eingebetteter Systeme gegenüber.
Im Bereich der Videocodierung für Mobilfunkanwendungen hat sich der MPEG-4-Standard weitgehend etabliert und wird hier stellvertretend für alle weiteren MPEG-Videostandards ausführlich beschrieben und analysiert.
Die vorliegende Arbeit befasst sich daher im Kern mit dem Entwurf einer speziellen Befehlssatz- und Coprozessorerweiterung für MPEG-4-basierte Videodecoderalgorithmen, womit ein Performancegewinn von ca. 50% gegenüber einer reinen Softwareimplementierung erzielt wird. Die Erweiterungen sind in ihrer Definition generisch und daher nicht auf einen speziellen Prozessortyp zugeschnitten.
Im vorliegenden Falle wird eine für Mobilfunkterminals typische RISC-Architektur herangezogen, um die Leistungsfähigkeit unter Beweis zu stellen und den Einsatz in eingebetteten Systemen aufzuzeigen.
Die einzelnen Konzepte werden auf Basis einer Algorithmenanalyse hergeleitet, wobei erst eine Beschreibung der generischen Erweiterung erfolgt und anschließend die Integration in den verwendeten Prozessorkern unter Verwendung der Hardwarebeschreibungssprache VHDL beschrieben wird.
Für die Bemessung des Echtzeitverhaltens wird im Rahmen der Arbeit ein spezieller Profiler entworfen, der unter anderem auch die Untersuchung und Optimierung des Speicherzugriffsverhaltens gestattet.
Mit Hilfe des Profilers werden Messungen durchgeführt, die den Rechenzeitgewinn der jeweiligen Algorithmenteile unter Zuhilfenahme der implementierten Optimierungen aufzeigen. Ebenso wird ein Vergleich der Leistungsfähigkeit der vorgestellten Architektur mit gängigen Prozessorarchitekturen, wie Superskalare und VLIW-Prozessoren, vorgenommen.
Hierbei wird ermittelt, dass das entwickelte Konzept ähnliche Resultate erbringt wie die vergleichsweise komplexeren Prozessoren.
Neben der Leistungsfähigkeit steht auch die Ermittlung des Flächenbedarfs im Falle einer CMOS Gate-Array-basierten Implementierung zur Diskussion und wird ebenfalls für jede einzelne Erweiterung dargestellt.

Abkürzungsverzeichnis

3GPP	-	Third Generation Partnership Program
AGU	-	Address Generation Unit
API	-	Application Programming Interface
ASIC	-	Application Specific Integrated Circuit
ASIMD	-	Autonomous Single Instruction Multiple Data
AU	-	Access Unit
CAM	-	Content Addressable Memory
CMOS	-	Complementary Metal Oxide Semiconductor
CMV	-	Candidate Motion Vector
DCT	-	Discrete Cosine Transform
DMA	-	Direct Memory Access
DPCM	-	Differentielle Puls Code Modulation
DRAM	-	Dynamic Ransom Access Memory
DVB	-	Digital Video Broadcast
DVB-H	-	Digital Video Broadcast for Handhelds
DVD	-	Digital Versatile Disc
ETSI	-	European Telecommunication Standards Institute
EX	-	Execute
FIFO	-	First In First Out Memory
FIR	-	Finite Impulse Response
FLC	-	Fixed Length Coding
FLD	-	Fixed Length Decoding
fps	-	frames per second
gprof	-	GNU Profiler
GSM	-	Groupe Special Mobile
HDTV	-	High Definition Television
HW	-	Hardware
IDCT	-	Inverse Discrete Cosine Transform
IF	-	Instruction Fetch
IIR	-	Infinite Impulse Response
IP	-	Intellectual Property oder Internet Protocol
Iprof	-	Instruction profiler
IRQ	-	Interrupt Request
ISO	-	International Standardisation Organisation
ITU	-	International Telecommunication Union
LSB	-	Least Significant Bit
MAI	-	Multiple ARC Interface
MB	-	Makroblock
MIMD	-	Multiple Instruction Multiple Data
MISD	-	Multiple Instruction Single Data
MIPS	-	Million Instructions per Second
MMS	-	Multi Media Messaging
MMU	-	Memory Management Unit
MOPS	-	Million Operations per Second
MPEG	-	Motion Pictures Expert Group
MSB	-	Most Significant Bit

MV	-	Motion Vector
OF	-	Operand Fetch
Pel	-	Picture Element / Pixel
PLA	-	Programmable Logic Array
PSNR	-	Peak Signal to Noise Ratio
Q	-	Quantisierung
Q-1	-	Inverse Quantisierung
RISC	-	Reduced Instruction Set Computer
ROM	-	Read Only Memory
RTP	-	Real Time Transfer Protocol
RVLC	-	Reversible Variable Length Codes
SAD	-	Sum of Absolute Differences
SIMD	-	Single Instruction Multiple Data
SISD	-	Single Instruction Single Data
SOC	-	System on a Chip
SPMD	-	Single Program Multiple Data
SRAM	-	Static Random Access Memory
SW	-	Software
TDMA	-	Time Division Multiple Access
TFT	-	Thin Film Transistor
UDP	-	User Datagram Protocol
UMTS	-	Unified Mobile Telephone System
VHDL	-	Very High Definition Language
VL	-	Variable Length
VLC	-	Variable Length Coding
VLD	-	Variable Length Decoding
VLIW	-	Very Long Instruction Word
VLSI	-	Very Large Scale Integration
VO	-	Video Object
VOL	-	Video Object Layer
VOP	-	Video Object Plane
WB	-	Write Back

Inhaltsverzeichnis

1 Einleitung

Aufgrund der steigenden Digitalisierung von Übertragungskanälen und dem drastischen Anstieg zu übertragender Informationsmengen hat die Kompression von Daten jeglicher Art zunehmend an Bedeutung gewonnen. Im Besonderen gilt dies für die digitale Übertragung von Bild- und Toninformationen, die nach der Digitalisierung eine extrem hohe Datenrate aufweisen. Ohne entsprechende Kompressionsverfahren wäre somit eine digitale Übertragung unmöglich.
Durch den Einsatz von Verfahren zur Datenkompression kann jedoch die erforderliche Datenrate digitalisierter Informationen so weit reduziert werden, dass, im Verhältnis zu analogen Verfahren, die erforderliche Bandbreite nur noch einen Bruchteil beträgt.
Ein Beispiel hierfür ist der steigende Bedarf an Kapazitäten für Mobilfunkanwendungen, wodurch der Übergang von analogen zu digitalen Verfahren mit entsprechenden Sprachkompressionsmechanismen unabdingbar wurde und die Kapazität bestehender Kanäle nahezu verzehnfacht werden konnte.
Der Einsatz dieser Verfahren wurde erst durch die Fortschritte im Bereich der digitalen Signalverarbeitung und der Mikroelektronik ermöglicht. Das Aufeinandertreffen hochleistungsfähiger digitaler Signalprozessoren und die Entwicklung spezieller Verfahren zur digitalen Sprachkompression machten somit die zunehmend kostengünstigere Umsetzung der anwenderseitigen Gerätetechnik möglich.
Weitere Vorteile entstehen durch den verbesserten Fehlerschutz, der auf digitale Datenströme angewandt werden kann, und einen niedrigeren Stromverbrauch, der durch den Einsatz hochintegrierter Bausteine bei niedrigen Versorgungsspannungen erzielt wird.
Somit haben sich durch die Verwendung von Kompressionsverfahren, neben der anfänglich im Vordergrund stehenden ökonomischeren Nutzung bestehender Datenkanäle, weitere wesentliche Vorteile gegenüber den herkömmlichen analogen Verfahren eingestellt.
Eine weitere wichtige Anwendung von Kompressionsverfahren stellt die effiziente Speicherung auf entsprechenden Medien dar.
Hierfür wurden standardisierte Verfahren für Bild- und Toninformationen entwickelt, die es gestatten, Speichermedien, wie z.B. CDs oder DVDs, zu nutzen.
Die wohl bekanntesten und z. Zt. marktrelevanten Verfahren stellen dabei die von der ISO[1] veröffentlichten MPEG-Standards dar, die ihre Anwendung vornehmlich für die Speicherung von Multimediadaten auf DVDs und im Bereich des digitalen Fernsehens DVB[2] in Form des MPEG-2-Standards gefunden haben.
Neben den bereits erwähnten Applikationen gibt es eine Vielzahl von Nischenanwendungen, wie z.B. funkgebundene Überwachungskameras, Spiele mit hochqualitativer Grafikdarstellung und dgl., wo sich Kompressionsverfahren etabliert oder sogar neue Anwendungsbereiche ergeben haben. Hierfür sind als Beispiel MPEG1-Layer3 (MP3)-basierte Abspielgeräte zu nennen, die aufgrund der hohen erzielten Datenkompression ausschließlich mit Halbleiterspeichern ausgerüstet werden können.
Im Rahmen von internationalen Standardisierungsarbeiten haben sich im Wesentlichen zwei Gremien mit der Entwicklung von Standards zur Codierung von Bild- und Toninformationen auseinander gesetzt. Hierbei handelt es sich um die ISO, die mit der Arbeitsgruppe SC29WG11 die schon erwähnten MPEG-Standards ins Leben gerufen hat, und die ITU[3], die im Bereich der Telekommunikationsanwendungen ebenfalls eine Reihe von Standards zur

[1] ISO - International Standardisation Organisation
[2] DVB - Digital Video Broadcast
[3] ITU- International Telecommunication Union

Bild- und Toncodierung entwickelt hat. Dabei arbeiten beide Organisationen nicht getrennt voneinander, sondern es wird vielmehr auch hier versucht, Synergieeffekte zu nutzen.
Wie schon eingehend erwähnt, stellen Standards ein wichtiges Kriterium für die Einführung neuer Dienste dar. Daher wird auch die internationale Entwicklung in den Standardisierungsgremien stark von Vertretern der jeweiligen Unternehmen der Elektronikindustrie vorangetrieben.
Die neuesten standardisierten Verfahren der ISO zur Codierung von Bild- und Toninformationen finden sich im MPEG-4-Standard, der sich durch eine Vielzahl unterschiedlicher, flexibel einsetzbarer Verfahren auszeichnet und somit nicht mehr ausschließlich auf eine spezielle Zielapplikation ausgerichtet ist.
Neben den sog. offenen Standards existieren eine Reihe firmenspezifischer Verfahren, wie z.B. *Microsoft WindowsMedia 9* oder *Real Video,* auf die in der vorliegenden Arbeit jedoch nicht eingegangen wird, da diese nicht frei verwendet werden können.
Ein weiterer wichtiger Faktor für die breite Anwendung von Codierungsverfahren neben der Verfügbarkeit entsprechender Standards, stellt die kostengünstige Umsetzung der Geräte des Endanwenders dar.
Es ist schon bei der Standardisierung darauf zu achten, dass die hohe Leistungsfähigkeit bestimmter Verfahren nicht durch einen zu hohen Aufwand im Endgerät und letztendlich mit einem zu hohen Preis erkauft wird.
Des Weiteren ist beim Entwurf der jeweiligen Verfahren zu berücksichtigen, dass bei hoher Komplexität die Entwicklungszeiten stark ansteigen, da unter anderem der Aufwand für die Verifikation spezieller Bausteine unverhältnismäßig hoch ist. Dies trifft besonders auf Komponenten zu, die aufgrund hoher Realzeitanforderungen ausschließlich mit Hilfe dedizierter Logik aufgebaut werden können.
Dabei hat die Entwicklung der Mikroelektronik die Leistungsfähigkeit verfügbarer Designwerkzeuge weit überschritten, d.h. rein technologisch sind integrierte Bauelemente mit hoher Komplexität, wie sie für die Realisierung entsprechender Komponenten notwendig sind, umsetzbar. Das größte Problem stellt jedoch die Handhabbarkeit großer Designs dar, das als sog. Design-Gap bezeichnet wird.
Einen möglichen Ausweg aus dieser Situation zeigen die Bemühungen um die standardisierte Spezifikation einzelner chipinterner Komponenten, die, wie aus einem Baukasten, zu einem beliebigen System zusammengestellt werden können und somit die Wiederverwendung bereits entwickelter Komponenten ermöglichen (design reuse).
Unter Einsatz dieser als „System On a Chip“, kurz SOC, bekannt gewordenen Designmethodik kann schnell auf Marktbedürfnisse reagiert und z.B. schneller ein Bauelement für ein neues Verfahren kostengünstig realisiert werden.
Ein Ansatz, SOC-Systeme effizient umzusetzen, stellt die Verwendung von Prozessoren dar, die entsprechend programmiert, die gleiche Funktionalität wie eine bestimmte Logikkomponente auf einem Chip erfüllen. Der Vorteil besteht jedoch hier in der Verwendung einer bereits verifizierten Komponente, die beliebig oft und sehr flexibel für unterschiedliche Aufgaben eingesetzt werden kann. Die Verifikation einer programmierten Funktionalität kann hierbei wesentlich schneller mit Hilfe prozessorspezifischer Softwaresimulatoren vorgenommen werden. Innerhalb einer Produktfamilie können durch zeitsparende Änderungen softwareprogrammierter Funktionalitäten sehr kurze Entwicklungszeiten erzielt werden.

1.1 Motivation

Eine der stärksten treibenden Kräfte für den forcierten Einsatz von sog. eingebetteten Prozessoren stellt der Mobilfunksektor dar, der sich in der Entwicklung von Endgeräten durch immer kürzere Innovationszyklen auszeichnet. Gerade der gestiegene Funktionsumfang heutiger Mobiltelefone macht den Einsatz von hochleistungsfähigen Prozessoren bei gleichzeitig niedrigem Stromverbrauch unabdingbar.
Durch die Einführung neuer Dienste, die auch die zusätzliche Übertragung von Bildinformationen ermöglichen, wie z.B. MMS[4], kommt somit Systemprozessoren ein vollständig neues Anwendungsfeld zu. Jedoch scheidet deren Einsatz hierfür in den meisten Fällen aus verschiedenen Gründen aus.
Typischerweise sind Chipfläche und Verlustleistung Kriterien, die den Einsatz leistungsfähiger Prozessoren scheitern lassen und teilweise zu dedizierten Lösungen führen, die mit einem hohen Entwicklungsaufwand verbunden sind.
Aber gerade im Bereich der mobilen Bilddatenkommunikation lassen sich Prozessoren durch geeignete Optimierungsmaßnahmen für die Umsetzung von Bildsignalverarbeitungsverfahren erfolgreich einsetzen, ohne die hohen Anforderungen an Verlustleistung und Chipfläche unverhältnismäßig zu überschreiten.
Hierfür spricht die im Vergleich zu anderen typischen Bildcodierungsanwendungen, wie z.B. dem digitalen Fernsehen, geringere Bandbreite, die somit auch niedrigere Bildauflösungen impliziert und weniger Rechenleistungen für die Verarbeitung erfordert.
Die Hauptmotivation der vorliegenden Arbeit stellt daher die Optimierung von Prozessorarchitekturen für die effiziente Softwareumsetzung von Decoderalgorithmen zur Verarbeitung MPEG-4-komprimierter Bilddaten dar.
Dabei soll gezeigt werden, dass sich auch mit einfachsten Prozessorarchitekturen unter Einsatz entsprechender Optimierungsmaßnahmen auf der Basis spezieller Befehlssatz- und Coprozessorerweiterungen extrem leistungs- und wettbewerbsfähige Decodersysteme aufbauen lassen, die somit eine Schlüsselposition in zukünftigen mobilen Systemen einnehmen werden.

1.2 Gliederung

Der erste wesentliche Teil der Arbeit wird aus dem in **Kapitel 2** dargestellten Stand der Technik gebildet. Hier wird auf die aktuellen Trends im Bereich der Bildsignalverarbeitung und Codierung für Mobilfunkanwendungen eingegangen. Das Kapitel geht daher einleitend auf die Verfahren und Standards ein, bevor ein Überblick über die z. Zt. vorherrschenden Prozessorarchitekturen für Bildsignalverarbeitungsanwendungen gegeben wird. Hierbei werden einzelne Prozessorarchitekturen wie auch Gesamtsysteme und deren Verwendungen diskutiert und dargestellt. Die Messung und Bewertung der Leistungsfähigkeit von Prozessorsystemen stellt den Abschluss des Kapitels dar.
Kapitel 3 der vorliegenden Arbeit ist den einzelnen Verfahren und Algorithmen der Bildsignalverarbeitung gewidmet, die die Grundlage aller aktuellen Bildcodierungsstandards bilden.
Die Zusammenführung der Einzelverfahren zu einem hybriden Codierungsverfahren wird ebenfalls erläutert, wobei der prinzipielle Aufbau entsprechender Encoder wie auch Decoder dargestellt wird. Der zweite Teil des Kapitels geht auf die Umsetzung der vorhergehend dargestellten Einzelverfahren ein und stellt unterschiedliche Implementierungsvarianten der Kernalgorithmen eines Videodecoders für hybride Bildcodierungsverfahren dar. Hierzu zäh-

[4] MMS- Multi Media Messaging

len die Verfahren der Bitstromverarbeitung, der inversen Cosinustransformation und der Bewegungskompensation.
Aufbauend auf den dargestellten Grundlagen wird in **Kapitel 4** ein für die Videosignalverarbeitung optimiertes RISC-Prozessorsystem entworfen. Ausgehend von einer Einzelanalyse werden für echtzeitkritische Decoderalgorithmen spezielle Prozessorbefehle abgeleitet bzw. in Form eines Coprozessors implementiert. Die Integration der entwickelten Optimierungen in eine gegebene RISC-Prozessorarchitektur wird im zweiten Teil des Kapitels erläutert, wobei unter anderem auch auf die angewandte Designmethodik eingegangen wird.
Kapitel 5 ist der Verifikation und Analyse der umgesetzten Optimierungen gewidmet. Dabei wird einleitend eine quantitative Bestimmung der erzielten Taktzyklenreduktion für die einzelnen Optimierungen vorgenommen. Hierbei wird ebenfalls auf die Komplexität der einzelnen Konzepte eingegangen und ein Vergleich mit den in Kapitel 2 dargestellten Prozessorarchitekturen durchgeführt. Weiterhin werden die Parameter realer Umsetzungen des Systems wiedergegeben.
Der Abschluss der Arbeit wird durch **Kapitel 6** gebildet, das einen Ausblick auf weitere Arbeiten in dem diskutierten Themenkomplex bietet.

.

2 Bildcodierung für mobile Applikationen - Stand der Technik

Das nachfolgende Kapitel beschreibt den aktuellen Stand der Technik im Bereich der standardisierten Bildcodierungsverfahren und deren Umsetzung mit Hilfe von prozessorbasierten Lösungen für Mobilfunkanwendungen.
Hierbei wird eingangs kurz auf die Historie der Entwicklung unterschiedlicher Bildcodierungsstandards eingegangen und anschließend der MPEG-4-Standard mit seinen wesentlichen Funktionalitäten und Neuerungen gegenüber den älteren Verfahren vorgestellt.
Der zweite Teil des Kapitels wird durch die Vorstellung und Diskussion der am häufigsten anzutreffenden Prozessorarchitekturen für Anwendungen im Bereich der Bilddatenkompression gebildet.
Der dritte Teil des Kapitels geht auf Verfahren zur Rechenleistungsermittlung ein.
Das Kapitel wird mit einer zusammenfassenden Bewertung der vorgestellten Themenbereiche abgeschlossen.

2.1 Bildcodierung für mobile Anwendungen

Wie in der Einleitung schon erwähnt wurde, werden sich zukünftig immer mehr Anwendungen für Bildcodierungsverfahren im Bereich der mobilen Anwendungen finden, d.h. als Endgerät werden vornehmlich batteriegespeiste Terminals verwendet werden.
Zum einen handelt es sich dabei um Anwendungen, die einen Kommunikationskanal voraussetzen, wie z.B. Mobilfunknetze oder digitale Broadcastdienste. Auf der anderen Seite werden aber auch in zunehmendem Maße tragbare Abspielgeräte für audio-visuelle Daten interessant, die z.B. in Form von Handheldgeräten bzw. PDAs Verbreitung finden werden.
Im Rahmen der funkgebundenen Anwendungen können im Wesentlichen die sog. Conversational Services, d.h. die der bidirektionalen Kommunikation dienenden Dienste von reinen Broadcastdiensten unterschieden werden. Zum Zwecke der Standardisierung der mobilfunkbasierten Dienste (anfänglich nur GSM-Dienste, später auch GPRS, UMTS) wurde das *Third Generation Partnership Program (3GPP)* gebildet.
Folgende Dienste wurden dabei spezifiziert und beinhalten neben der genauen Beschreibung der Übertragungsverfahren auch die Spezifikation der verwendeten Videokompressionsstandards und Verfahren, die dort zur Anwendung kommen:

- H.320 Conversational
- 3GPP Conversational H.324/M
- H.323 Conversational Internet/best effort IP/RTP
- 3GPP Conversational IP/RTP/SIP
- Streaming Services
- 3GPP Streaming IP/RTP/RTSP
- Streaming IP/RTP/RTSP, wie z.B. in [1] beschrieben
- 3GPP Multimedia Messaging Services .

Sämtlichen Services ist gemeinsam, dass das vornehmlich eingesetzte Videocodierungsverfahren hier durch den MPEG-4-Standard gebildet wird. Für mobile digitale Broadcastanwendungen, d.h. die digitale Übertragung von Fernsehdiensten, wurde ursprünglich unter anderem der MPEG-2-basierte DVB-T-Standard (siehe hierzu [2]) entworfen.

Die anfängliche Euphorie, die durch die sehr guten Eigenschaften von DVB-T in Bezug auf Störunanfälligkeit in mobilen Umfeldern hervorgerufen wurde, dämpfte sich jedoch durch den hohen Stromverbrauch der Endgeräte. Hierbei ist in erster Linie das für die Kanaldecodierung verwendete Verfahren verantwortlich, welches eine ständige Bereitschaft des HF-Frontends erforderlich macht und z. Zt. zu Verlustleistungen für den Kanaldecoder von ca. 0,4W führt (siehe hierzu auch [4]) und damit batteriegespeiste Geräte ineffizient macht.
Um auf der Basis einer DVB-T-Infrastruktur trotzdem mobile Broadcastdienste zu realisieren, wird seit 09/2002 der DVB-H-Standard (erst DVB-M, dann DVB-X, seit 09/2003 DVB-H) entwickelt. Hierbei wird von wesentlich niedrigeren Datenraten ausgegangen, die im Bereich von 384 kBit/s liegen und eine nicht kontinuierliche Empfangsbereitschaft des Kanaldecoders ermöglichen.
Auf der Basis des sog. Time-Slicing wird ein DVB-T-Kanal in mehrere DVB-H-Kanäle aufgeteilt. Das HF-Frontend und der verwendete Kanaldecoder werden lediglich für die Zeiteinheit des zu decodierenden Time-Slice eingeschaltet, was zu einer Reduzierung der Stromaufnahme führt. Als Ziel für die Verlustleistung entsprechender Endgeräte wird momentan von 100 mW ausgegangen. Auch hier werden sämtliche Verfahren standardisiert, die im Rahmen der DVB-H-Dienste zur Anwendung kommen können, somit auch das verwendete Videocodierungsverfahren, das in diesem Falle ebenfalls Teil des MPEG-4-Standards sein wird (MPEG-4 Part 10/H.264 AVC). Mehr Informationen zu DVB-H finden sich in [3].
Allen vorhergehend beschriebenen Anwendungen gemein ist neben dem mobilen Einsatz und den damit einhergehenden niedrigen Verlustleistungen die nicht eindeutige Charakteristik des eigentlichen Endgerätes. Sämtliche Verfahren werden Einzug in Mobiltelefone finden, wie auch Empfangsgeräte für DVB-H höchstwahrscheinlich auf der Basis von Kleinstrechnern, z.B. PDAs, aufgebaut sein werden, die ebenso über Telefonfunktionen verfügen werden. Reine dedizierte Endgeräte werden sich daher nur im Bereich extrem kostensensitiver Anwendungen finden.
Die Architektur eines mobilen Endgerätes wird demnach nicht mehr auf eine spezielle Applikation ausgerichtet sein, sondern vielmehr die Form eines Multifunktionsgerätes für die beschriebenen Anwendungsszenarien haben.

2.2 Bildcodierungsstandards

Der erste Bildcodierungsstandard, der auf dem in Abschnitt 3.3 näher beschriebenen hybriden Codierungsverfahren beruhte und in größerem Maße eingesetzt wurde, ist der im Rahmen der ISO entwickelte MPEG-1-Standard. Der Grundgedanke bestand hier in der Schaffung eines genormten Verfahrens zur Speicherung von Audio- und Videoinformationen auf Compact-Discs. Da es sich vornehmlich nur um eine einzelne Hauptanwendung handelte, wurde dabei neben den Verfahren auch die nutzbare Bandbreite von alternativen Anwendungen eingeschränkt, indem z.B. die Bildauflösung und weitere Parameter fest vorgegeben wurden.
Die Unzulänglichkeiten des MPEG-1-Standards in Hinblick auf den Einsatz in Fernsehübertragungssystemen führte zur Entwicklung des MPEG-2-Standards (siehe [6]), der sich neben den bereits etablierten DVB-Systemen auch für die Speicherungen von Multimediadaten auf DVDs durchgesetzt hat und somit eine höhere Flexibilität in Bezug auf mögliche Anwendungen aufweist.
Die zunehmende Zahl von softwarebasierten Anwendungen auf PCs führte unter anderem zu einem steigenden Bedarf an Codierungsverfahren, die eine höhere Flexibilität aufweisen sollten als die bisher etablierten Verfahren. Als Antwort hierauf wurde der MPEG-4-Standard von der ISO ins Leben gerufen, dessen Entwicklung ca. 1997 aufgenommen wurde und sich

nach einigen Umorganisierungsmaßnahmen und Zwischenergebnissen noch in der Standardisierungsphase befindet.
Entgegen der ursprünglichen Planung, einen offenen Standard für Multimediaanwendungen zu schaffen, wurde aufgrund der Komplexität dieses Vorhabens kein endgültiger Termin für den Abschluss der Arbeiten definiert. Vielmehr handelt es sich um eine Basis von Standards, die ständig durch neue Unterstandards (Subparts) ergänzt wird.
Die wichtigsten Subparts bilden die Standards zur Video- und Audioquellencodierung (ISO 14496-2 und ISO 14496-3) und der Systemstandard (ISO 14496-1) zur Standardisierung von Terminalarchitekturen. Der neueste Subpart (ISO 14496-10 bzw. H.264/AVC) stellt wiederum einen völlig neuen Standard zur Videoquellencodierung dar, der aus dem ITU-H.26L-Standard hervorgegangen ist.
Neben den einzelnen Subparts des Standards wurden zusätzlich Versionen der einzelnen Parts etabliert, die sich aus dem Bedarf von speziellen Anwendungen ergeben haben. Der MPEG-4-Videostandard weist insgesamt vier Versionen auf, die mit den Versionen 1 und 2 universell einsetzbare Codierungsverfahren definieren und mit Version 3 (Studio-Profile) und Version 4 (Internet Streaming Profile) auf spezielle Applikationen ausgerichtet sind. In [8], [9] und [10] sind weiterführende Informationen zur Standardisierung zu finden.
Im Rahmen der vorliegenden Arbeit werden ausschließlich Verfahren der Versionen 1 und 2 verwendet, weshalb nicht näher auf die weiteren Versionen eingegangen wird.
Die Standards zur Bilddatenkompression spezifizieren in der Regel nicht die Arbeitsweise und den Aufbau des jeweiligen Encoders, sondern die Architektur eines Referenzdecoders und die Syntax des zu verarbeitenden Datenstroms. Der Vorteil dieser Vorgehensweise liegt in der Implementierungsfreiheit bei der Entwicklung des Encoders, womit der zu erreichenden Bildqualität nur Grenzen durch das Einhalten der Standardkonformität gesetzt sind.
Die ISO hat hierfür ein einheitliches Format für die Beschreibung der Bitstromsyntax festgelegt, das dem Aufbau einer Programmiersprache sehr ähnlich ist. Im Wesentlichen müssen für die Beschreibung Datenelemente mit fester (*FLC*) und variabler Länge (*VLC*) bedingte Sprünge und Schleifen darstellbar sein. Darüber hinaus wurden Datentypen definiert, die der eindeutigen Bestimmung des jeweiligen Syntaxelements dienen. Ein entsprechendes Beispiel hierzu findet sich in Tabelle 10 in Anhang G.2.

2.3 MPEG-4-Videocodierung

Die MPEG-4-Videocodierung (spezifiziert in [7]) basiert im Wesentlichen auf den Verfahren der hybriden Videocodierung, die in den Vorläufern des Standards, wie z.B. ISO-MPEG-1/-2 oder ITU-H.263, ebenfalls schon verwendet wurden.
Im Vordergrund der Entwicklung stand, neben der Steigerung der Codierungseffizienz, die Erschließung neuer Anwendungsszenarien. Daher wurde der Standard nicht nur in Hinblick auf eine bestimmte Applikation entwickelt, sondern es wurde Wert auf die Integration möglichst vieler effizienzsteigernder und flexibel einsetzbarer Verfahren gelegt.
Ein Novum stellt hierbei z.B. die Möglichkeit der Codierung von beliebig geformten Videoobjekten dar. Dabei wird nicht nur die eigentliche Bildinformation im klassischen Sinne codiert, sondern zusätzlich auch die Kontur eines beliebigen Objektes verarbeitet.
Der ursprüngliche Ansatz war der Gedanke, mit Hilfe einer speziellen, auf das Szenenobjekt angepassten Codierung, eine möglichst hohe Codiereffizienz erzielen zu können. Im Falle von natürlichen Videoszenen bedeutet dies, dass mittels eines Segmentierungsverfahrens einzelne Objekte einer Szene erkannt und damit ausschließlich deren Konturen und dazugehörigen Texturen codiert werden können (siehe Abbildung 1).

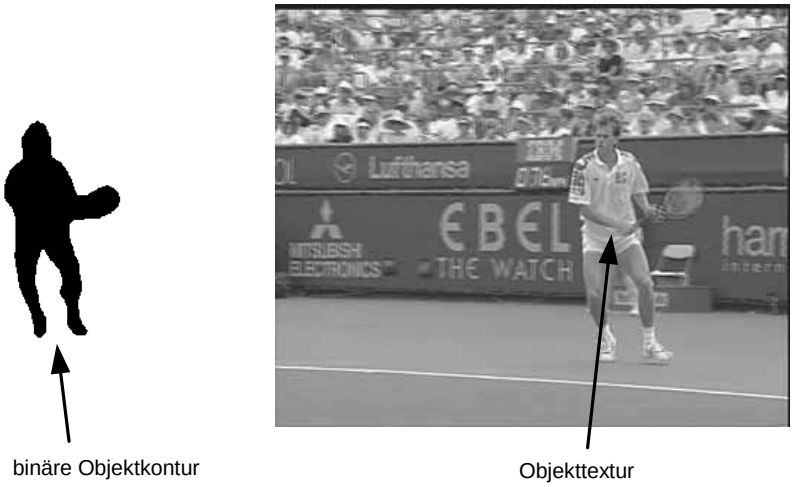


Abbildung 1 Segmentierungsmaske für objektbasierte Videocodierung

Aufgrund der Datenmenge, die für die Codierung der Objektkontur zusätzlich benötigt wird, hält sich jedoch der erzielte Codierungsgewinn in Grenzen. Da der MPEG-4-Videocodierungsstandard, wie seine Vorgänger, auf dem hybriden Codierungsverfahren beruht, basiert die Effizienzsteigerung hier hauptsächlich auf der Weiterentwicklung der bisherigen Verfahren, wie z.B. der Verbesserung der Prädiktions- und Bewegungskompensationsmechanismen. Weiterhin stellt die Segmentierung und die Objekterkennung ein bisher noch nicht vollständig gelöstes Problem dar. Die Codierung der Objektkontur wird mit Hilfe eines arithmetischen Coders vorgenommen, der im Falle einer binären Objektkontur die vollständige Maske für die zusätzlich übertragene Texturinformation codiert. Ähnlich der bewegungskompensierten Codierung der Texturinformation wird auch für die Konturinformation eine prädiktive Codierung angewandt. Da sich die Bewegungsänderung der Textur und der Kontur eines Objektes gleichermaßen ändern, wird dabei auf den gleichen Bewegungsvektorsatz zurückgegriffen. Komplette Szenen können mit Hilfe des MPEG-4-Systems-Standards beschrieben werden. Sie können dabei aus einer beliebigen Anzahl von Videoobjekten und weiteren synthetisch generierten Objekten bestehen. Abbildung 2 stellt beispielhaft die Kombination von synthetischen Videoobjekten, linkes Bild, und natürlichen Videoobjekten, Bild in der Mitte, zu einer Gesamtszene, Bild rechts, dar.

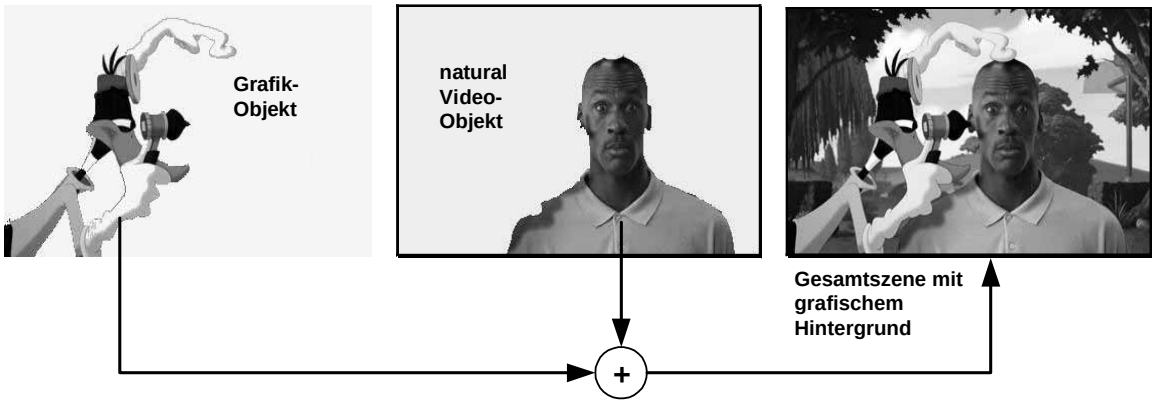


Abbildung 2 Beispiel zur objektbasierten Videocodierung

Weiterführende Informationen zur Konturcodierung können in [11] gefunden werden.

2.3.1 Fehlerschutzmechanismen

Ein weiterer wesentlicher Vorteil des MPEG-4-Standards stellen Fehlerschutzmechanismen auf Elementarstromebene dar, die bisher ausschließlich auf die Transportstromschicht angewandt werden konnten.
Die Möglichkeit, Videodatenströme mit einem eigenen, sog. inneren Fehlerschutz zu versehen, stellt eine der wesentlichen Bedingungen für den erfolgreichen Einsatz des Standards in Mobilfunksystemen dar. Der für die Datenübertragung angenommene Kanal wurde demnach immer als fehlerfrei vorausgesetzt. Für den Fall, dass der eigentliche Videodatenstrom einen Fehler aufweist, konnte keine Fehlerbehandlung vorgenommen werden, was zum Abbruch der Decodierung des aktuellen Bildes führte.
Im MPEG-4-Standard wurden daher erstmals Verfahren vorgesehen, die eine Fehlerbehandlung im Videodecoder ermöglichen, wodurch Datenübertragungsfehler in ihrer Fortpflanzung über den kompletten Bildbereich eingedämmt werden können. Das Grundkonzept ist dabei, einen Fehler im Datenstrom möglichst genau zu lokalisieren, um an einer definierten Stelle mit der Decodierung fortfahren zu können. Hierzu dienen im Wesentlichen folgende Verfahren, die einzeln oder in Kombination eingesetzt werden können:

- **Resync-Marker**

  In den Datenstrom werden in bestimmten Abständen spezielle Datenworte eingefügt, die der Neusynchronisation nach einem erkannten Fehler dienen sollen. Die Daten, die zwischen dem erkannten Fehler und dem sog. Resync-Marker liegen, können somit nicht mehr decodiert werden. Je kleiner der Abstand zwischen einzelnen Resync-Markern gewählt wird, desto besser lässt sich ein fehlerhafter Datenstrom rekonstruieren, um damit den visuellen Verlust an Bildinformation möglichst gering zu halten. Die Neusynchronisation kann im besten Falle für jeden einzelnen Makroblock vorgesehen werden.

- **Data-Partitioning**

  Mittels des Data Partitioning-Modes ist es möglich, die makroblockorientierte Struktur des Datenstroms so zu ändern, dass eine Trennung der Texturinformation von den Bewegungsvektoren vorgenommen wird. Dabei werden die Bewegungsvektoren eines Bildes zu einem Datensegment zusammengefasst und anschließend alle Datenelemente der Texturinformation übertragen. Somit wird eine spezielle Fehlerbehandlung ermöglicht, die es z.B. im Falle des Verlustes der Bewegungsinformation erlaubt, die eigentliche Bildinformation zu decodieren. Weiterhin besteht hierdurch die Möglichkeit, einen angepassten Fehlerschutz für die einzelnen Datensegmente zu verwenden.

- **Reversible VLC**

  Für eine exakte Lokalisierung eines Fehlers in den Texturdaten wurde ein spezieller VLC entworfen, der es erlaubt, nicht nur in normaler Abtastrichtung decodiert zu werden, sondern auch in umgekehrter Reihenfolge das gleiche Ergebnis nach der Decodierung liefert. Normalerweise führt ein Fehler in einem VLC-codierten Datenstrom zu einem Datenverlust, der nur durch das Auffinden des nächsten Synchronisationspunktes

abgefangen werden kann, wie durch einen Resync-Marker oder den Startcode des nächsten Bildes. Durch den Einsatz von RVLCs ist es jedoch möglich, den Datenfehler bitgenau einzugrenzen, indem nach einem erkannten Fehler der Datenstrom in umgekehrter Reihenfolge vom Ende des Datensegmentes ausgehend decodiert wird, bis wiederum ein Fehler auftritt. Das Konzept der Fehlereingrenzung mittels RVLCs ist in Abbildung 3 dargestellt.

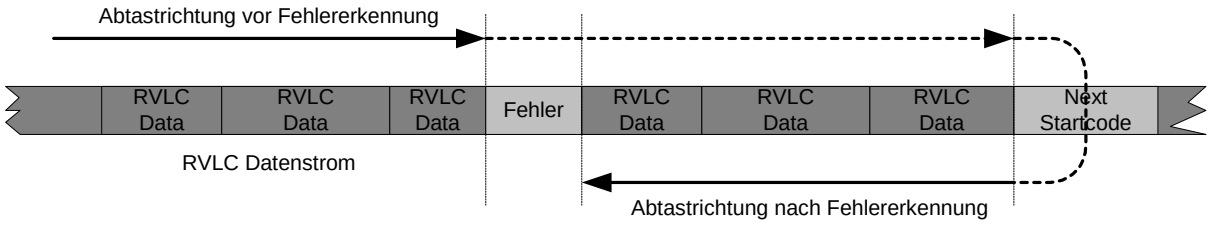


Abbildung 3 Fehlereingrenzung mittels RVLC

Bei den beschriebenen Verfahren handelt es sich um standardisierte Verfahren, den sog. Error-Resilience-Tools, die eine Fehlerbehandlung im Decoder ermöglichen. Die eigentliche Vorgehensweise der Fehlerbehandlung (Error-Concealment) ist hingegen nicht im Standard spezifiziert und kann frei implementiert sowie den jeweiligen Anforderungen angepasst werden. Weiterführende Informationen hierzu finden sich in [15].

2.3.2 MPEG-4-Anwendungsprofile

Um eine sinnvolle Zuordnung von Codierungsparametern und verwendeter Verfahren zu entsprechenden Applikationen zu ermöglichen, wurden im MPEG-4-Standard Definitionen eingeführt, die einerseits die verwendeten Verfahren und andererseits die zugehörigen Parameter, wie maximale Datenrate, Bildauflösung, Anzahl der Videoobjekte und die Bildwiederholfrequenz, festlegen. Die eingesetzten Verfahren (Tools) werden hierbei durch das sog. *Profile* definiert, während alle anderen übrigen Parameter, wie z.B. die verwendete Datenrate oder die Anzahl der Videoobjekte, aus der Wahl des *Levels* hervorgehen. Tabelle 1 gibt exemplarisch die im MPEG-4-Standard festgelegten Levels für das Visual Simple Profile mit den vorgegebenen Leistungsmerkmalen wieder.

Visual Profile	Level	Typical Visual Session Size	Max. MBs per VOP	Max. Objects	Max. number per type	Max. unique Quant Tables	Max. VMV buffer size (MB units)	Max. VCV buffer size (MB)	VCV decoder rate (MB/s)	Max. total VBV buffer size	Max. VBV buffer size	Max. bitrate (kbit/s)
Simple	L3	CIF	396	4	4 x Simple	1	792	396	11880	40	20	384
Simple	L2	CIF	396	4	4 x Simple	1	792	396	5940	40	20	128
Simple	L1	QCIF	99	4	4 x Simple	1	198	99	1485	10	5	64

Tabelle 1 Levels für das MPEG-4 Visual Simple Profile

Um eine höhere Flexibilität in Bezug auf standardkonforme Applikationen zu erzielen, wird z.B. statt der Festlegung von bestimmten Bildgrößen die maximale Anzahl von Makroblöcken eines Bildes bzw. eines VOP[5] angegeben. Eine weitere Neuerung, die für den Entwurf eines Decoders wesentlich ist, stellt die Angabe der Komplexität des jeweiligen Datenstroms durch die sog. *VCV[6] Decoder Rate* (mit der Einheit Makroblock/s) dar. Hiermit wird implizit eine bestimmte Verarbeitungsleistung von einem Decoder gefordert, der in der Lage sein soll, einen Datenstrom mit einer bestimmten Profile-/Level-Kombination echtzeitfähig zu decodieren.
Wird z.B. im Falle eines Datenstroms mit den Parametern „Simple Profile @ Level 2“ eine VCV Decoder Rate von 5940 angenommen, muss von einem echtzeitfähigen Decoder eine Verarbeitungsleistung gefordert werden, die es zulässt, 5940 Makroblöcke/s zu verarbeiten, wobei die Eingangsdatenrate nicht über 128 kBit/s liegen darf. Dies bedeutet z.B. bei einer angenommenen Bildgröße von 352 x 288 Bildpunkten eine maximal zulässige Bildwiederholfrequenz von 15 Bildern pro Sekunde.

2.4 Klassifizierung von Prozessorarchitekturen

Der nachfolgende Abschnitt soll der Klassifizierung von Prozessoren bzw. kombinierten Strukturen aus Prozessor und Coprozessor dienen, die den aktuellen Stand der Technik für die prozessorbasierte Umsetzung von Bildcodierungsverfahren darstellen. Dabei können grundsätzlich zwei Klassen von Datenverarbeitungsarchitekturen unterschieden werden.
Zum einen handelt es sich um Strukturen, die die parallele Verarbeitung von Daten erlauben. Hierzu zählen die Konzepte der SIMD[7]- und MIMD[8]-Prozessorarchitekturen, die eine parallele Verarbeitung mehrerer Datensätze mit einem Befehl (SIMD) oder mit mehreren Befehlen (MIMD) gleichzeitig ermöglichen. Aufgrund des teilweise hohen Grades an möglicher Datenparallelität bei Videoverarbeitungsalgorithmen bieten sich diese Architekturen besonders an.
Eine Variante dieses Konzeptes stellt die Sub-Word-Parallelität dar, bei der ein Datenwort für eine bestimmte Operation für mehrere einzelne Datensätze herangezogen wird. Hierzu zählen z.B. die um ISSE[9] erweiterten Intel Pentium-Prozessoren. Eine Darstellung dieses Konzeptes findet sich in Abschnitt 2.4.5 wieder.
Die zweite Art von Architekturen zeichnet sich durch Funktionsparallelität aus, indem entweder mit Hilfe mehrstufiger Pipelines oder durch den Einsatz sehr breiter Instruktionsworte ein hoher Befehlsdurchsatz ermöglicht wird. Die Funktionsparallelität mittels großer Instruktionswortbreiten wird im Falle der VLIW[10]- und Superskalaren Rechner angewandt, wobei hier auch Mischformen mit Pipelinestrukturen und SIMD-Erweiterungen auftreten können.
Grundsätzlich kann gesagt werden, dass sich in modernen General-Purpose-Prozessoren beide der zuvor genannten Architekturmerkmale wiederfinden und somit nicht immer eine eindeutige Zuordnung vorgenommen werden kann. So finden sich in einem Intel Pentium IV-Prozessor, der als grundlegendes Merkmal eine Superskalare Prozessorarchitektur darstellt, auch eine mehrstufige Pipeline und alle Merkmale einer SIMD-Architektur durch die ISSE2-Extensions, die eine parallele Verarbeitung auf Sub-Word-Ebene ermöglicht. Erweiterungen einfacher Prozessoren um Coprozessorfunktionalitäten können mit Hilfe der vorgestellten Klassifizierung als funktionsparallele Architekturen bezeichnet werden.

[5] VOP - Video Object Plane
[6] VCV - Video Complexity Verifier
[7] SIMD – Single Instruction Multiple Data
[8] MIMD – Multiple Instruction Multiple Data
[9] ISSE – Internet Streaming SIMD Extension
[10] VLIW – Very Long Instruction Word

Eine weitere Form von funktionsparallelen Architekturen stellt der Einsatz mehrerer Prozessoren dar, die in geeigneter Weise synchronisiert sind und über gemeinsam nutzbare Speicherstrukturen (shared memory) verfügen.
Eine Sonderform dieser Architektur stellt das von Intel in [17] vorgestellte Hyper-Threading-Konzept der Intel Pentium IV-Prozessoren dar. Hierbei wird ein Teil der Mikroarchitektur eines Prozessors zweifach ausgeführt, der sich somit dem Betriebssystem als Doppelprozessorsystem darstellt, real werden aber ALUs und Caches doppelt genutzt.

2.4.1 Reduced Instruction Set Computer

Die Entwicklung der RISC[11]-Prozessorarchitektur geht im Wesentlichen auf zwei Forschungsprojekte zurück. 1981 wurde an der Universität Stanford von Hennessy das MIPS-Projekt ins Leben gerufen, aus dem 1983 der erste als Chip gefertigte RISC-Prozessor und die Firma MIPS hervorging. Parallel hierzu startete 1980 an der Universität Berkeley das Berkeley-RISC-Projekt, von dem die Architektur der SUN-Sparc-Prozessoren abgeleitet wurde.
Bei Untersuchungen der bis dahin gängigen CISC[12]-Prozessorarchitekturen wurde festgestellt, dass statistisch gesehen nur ein Bruchteil aller verfügbaren Befehle und implementierten Adressierungsarten von gängigen Compilern auch wirklich genutzt werden.
Bei weniger häufig benutzten Befehlen handelte es sich jedoch meistens um sehr komplexe Befehle, die in hohem Maße auch die Chipfläche, die maximale Taktfrequenz und somit auch den Leistungsverbrauch des jeweiligen Prozessors bestimmten.
Ein weiterer Nachteil, der sich aus der Verwendung von CISC-Befehlen ergab, war die unterschiedliche Größe der einzelnen Instruktionen, was die effektive Abarbeitung in einer Pipeline erschwerte. Daher wurde aus den am meisten verwendeten Befehlen und Adressierungsarten eine Untermenge (*reduced instruction set*) gebildet und die nicht mehr vorhandenen Befehle durch entsprechende Softwareroutinen vom Compiler ersetzt, wodurch jedoch die Codegröße für äquivalenten Code im Vergleich zu CISC-Architekturen stieg.
Aufgrund der Beeinflussung der Prozessorarchitektur durch den Compilerbau wird bei RISC-Prozessoren daher auch von der Synergie aus Rechnerarchitektur und Compilerbau gesprochen. Die durch die Vereinfachung des Befehlsvorrats bedingte höhere Speicherbelastung wurde durch die Bereitstellung großer Registerbänke (Register-Files) teilweise kompensiert. Da der Zugriff auf den Speicher des Systems über Load- bzw. Store-Befehle ausgeführt, jedoch die eigentliche Verarbeitung ausschließlich mit Hilfe von Operandenregistern vorgenommen wird, werden RISC-Prozessoren auch als Load-/Store-Architekturen bezeichnet.
Durch die Beschränkung auf sehr wenige Befehle und der drastischen Einschränkung der Adressierungsarten konnte in der Regel, anstatt des normalerweise verwendeten Mikroprogrammrechenwerks, festverdrahtete Logik eingesetzt werden, wodurch sehr niedrige Ausführungszeiten für die einzelnen Prozessorbefehle erzielt werden konnten.
Die Kombination mit mehrstufigen Verarbeitungspipelines ermöglicht es somit bei einfachen RISC-Prozessoren, einen Befehl pro Taktzyklus auszuführen. In Abbildung 4 ist die interne Struktur des in der vorliegenden Arbeit eingesetzten Argonaut RISC-Cores mit seinen Verarbeitungsstufen dargestellt. Aufgrund der einfachen Architektur eines RISC-Prozessors spiegelt das dargestellte Blockschaltbild exemplarisch den grundsätzlichen Aufbau der meisten RISC-Prozessoren wider.

[11] RISC – Reduced Instruction Set Computer
[12] CISC - Complex Instruction Word Computer

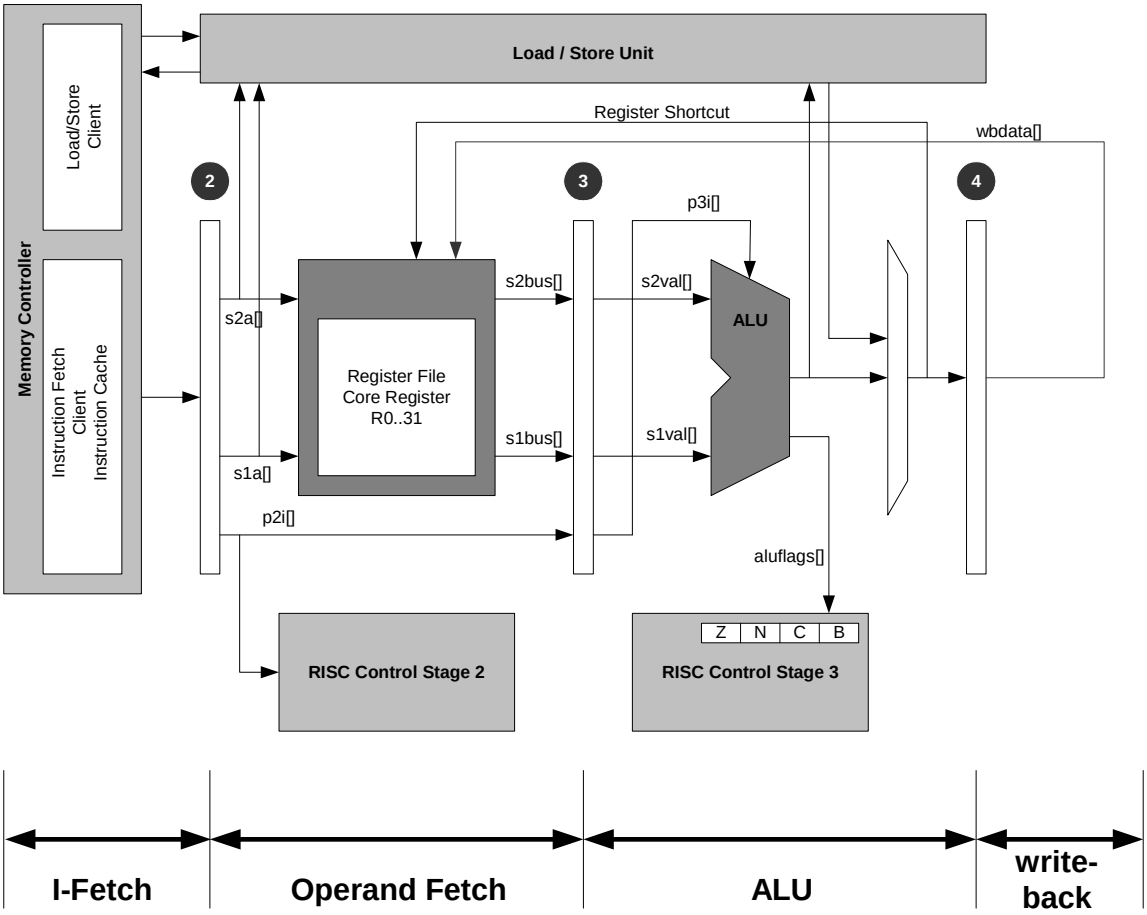


Abbildung 4 Blockschaltbild Argonaut RISC-Core mit Pipelinestufen

Die wesentlichen Merkmale sind dabei die vierstufige Befehlspipeline und eine durchgehende 32-Bit-Architektur mit 32 Bit breiten Daten- und Adresspfaden für Instruction-Fetch- und Load-/Store-Operationen. Der Prozessor verfügt in seiner Standardkonfiguration über ein Registerfile mit 32 Registern und eine ALU für arithmetische Operationen.
Aufgrund seiner Verfügbarkeit in Form des VHDL-Quelltextes kann der Prozessor durch benutzerdefinierte Befehle erweitert werden, wobei vom Hersteller bereits vordefinierte Befehlserweiterungen, wie Multiplikations- und Shift-Operationen, vorgegeben werden. Hierfür liegt eine entsprechende Hochsprachenunterstützung in Form von C- bzw. C++-Compilern vor. Bei der Verwendung anwenderdefinierter Instruktionen steht jedoch keine Compilerunterstützung zur Verfügung, sodass vollständig auf Assemblerprogrammierung zurückgegriffen werden muss.
Eine weitere Besonderheit des Prozessors stellt die Erweiterbarkeit des Registerfiles dar. Dabei steht ein eigenes Speicherinterface für den Anschluss eines externen Registerfiles zur Verfügung, auf das per Assemblerinstruktionen in Form von speziellen Registernamen zugegriffen werden kann. Weiterhin verfügt der Prozessor über einen direct-mapped, one-way associative Instruktions-Cache, der in seiner Größe an die Erfordernisse der Applikation angepasst werden kann.
Das nachfolgende Blockschaltbild (Abbildung 5) gibt die äußere Beschaltung des Prozessorkerns in seiner Grundkonfiguration wieder. Hierbei sind die unterschiedlichen Busse für Instruktionen (*if-client*) und Daten (*ld/st-client*) erkennbar, die auch auf einen gemeinsamen Speicher-Bus geführt werden können, falls eine lineare Speicherarchitektur vorliegt.

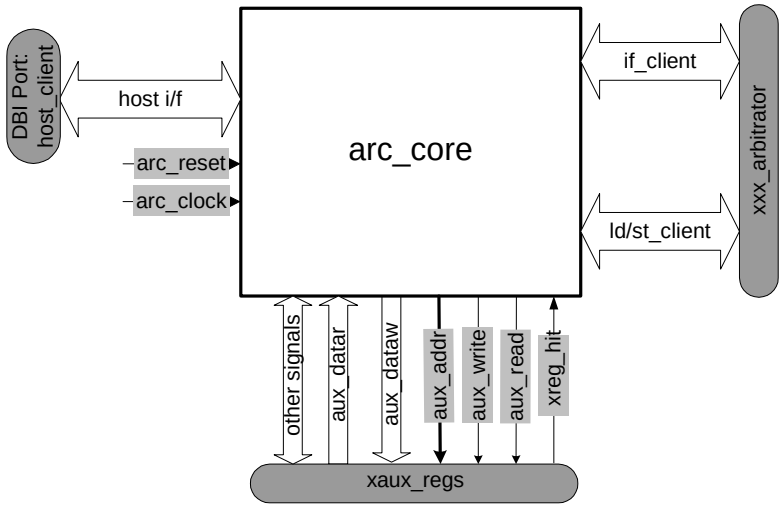


Abbildung 5 Top Level Entity Argonaut RISC-Core

Detaillierte Beschreibungen des Programmiermodells und der Basisoperationen sind in [42] zu finden. Durch den geringen Flächen- und niedrigen Leistungsbedarf haben sich RISC-Prozessoren stark im Bereich der eingebetteten Systeme etabliert. Hier werden in der Regel für einfache Kontroll- und Steuerungsaufgaben einfachste Prozessoren benötigt, die aber über eine Hochsprachenunterstützung verfügen müssen. Die größte Marktdurchdringung hat dabei die ARM RISC-Familie der Firma ARM erzielt.
In [18] wird von Hennessy die komplette Architektur eines dem MIPS-Prozessor sehr ähnlichen Prozessors (DLX32) exemplarisch hergeleitet, der die typischen Merkmale gängiger RISC-Prozessoren aufweist.

2.4.2 Very Long Instruction Word Computer

Im Gegensatz zu der im vorherigen Abschnitt beschriebenen RISC-Architektur können im Falle einer VLIW-Architektur pro Verarbeitungszyklus mehrere Befehle gleichzeitig ausgeführt werden. Um dies zu ermöglichen, beinhaltet ein einzelnes Instruktionswort nicht nur die Instruktion für eine Verarbeitungseinheit, sondern es werden mehrere Einheiten gleichzeitig und unabhängig voneinander angesprochen. Man spricht hierbei von Instruction-Level-Parallelism bzw. Parallelität auf Befehlsebene. Hierdurch entsteht ein hoher Datendurchsatz, der allerdings eine hohe Datenunabhängigkeit erfordert.
Um eine möglichst hohe Auslastung aller Einheiten des Prozessors zu ermöglichen, muss der Compiler bereits das Scheduling, d.h. die zeitliche Zuordnung der einzelnen Befehle auf die im Prozessor zur Verfügung stehenden Funktionseinheiten, vornehmen.
Problematisch für die maximale Gesamtauslastung aller Funktionseinheiten gestaltet sich die effiziente Verarbeitung von Sprungbefehlen. Sequentieller Programmcode, der über viele Abhängigkeiten bzw. bedingte Sprünge verfügt, eignet sich daher in der Regel nur sehr schlecht für VLIW-Prozessoren im Gegensatz zu eher regulären Konstrukten. In Analogie zu der vorhergehend beschriebenen RISC-Architektur kommt dem für den jeweiligen Prozessor entwickelte Hochsprachencompiler eine Schlüsselposition zu, da die Gesamtperformance des Prozessors im Wesentlichen durch die Fähigkeit des Compilers, parallelen Code zu generieren, bestimmt wird. Durch das Nichtvorhandensein eines Hardwareschedulers können somit

sehr kostengünstig Hochleistungsprozessoren realisiert werden, die durch ihren einfachen Aufbau hohe Taktfrequenzen ermöglichen.
In Abbildung 6 ist exemplarisch eine VLIW-Struktur mit sechs parallelen Einheiten, hier FU (Functional Units), dargestellt.

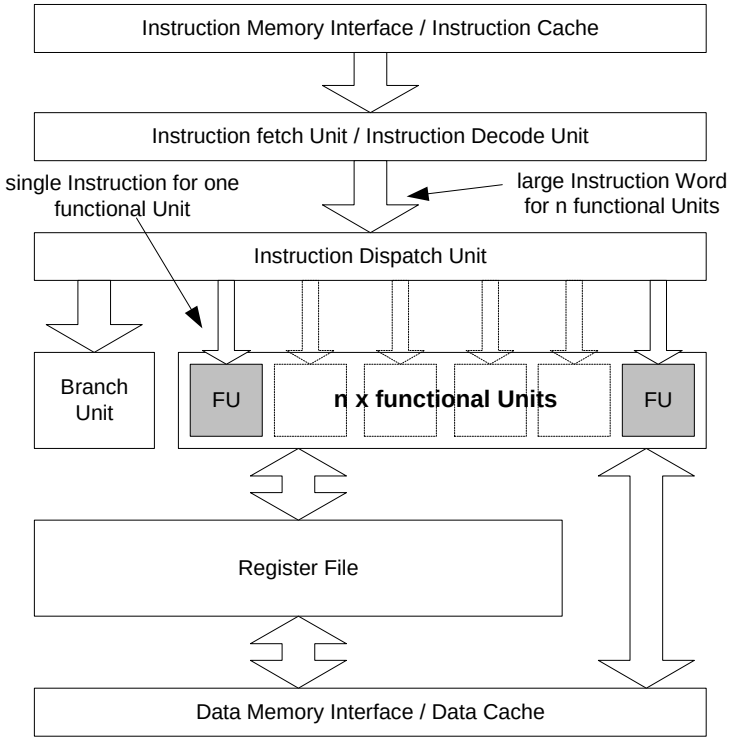


Abbildung 6 Grundprinzip einer VLIW-Architektur

Neben der Anwendung als Spezialprozessor in Signalverarbeitungsanwendungen haben sich VLIW-Prozessoren auch im Bereich der Multimediadatenverarbeitung und bei Spezialapplikationen mit einem hohen Parallelisierungsgrad etabliert. Zu den momentan aktuellen Prozessoren gehören folgende Typen:

- Intel IA-64, näher beschrieben in [20]
- Texas Instruments TMS320C6XXX Familie, näher beschrieben in [22] und [23]
- Philips Trimedia TM1XXX Familie .

Wie schon eingangs erwähnt wurde, können einzelne Prozessoren auch Mischformen der bisher beschriebenen Prozessorarchitekturen darstellen. Dies gilt im Speziellen für die Prozessoren Trimedia TM1XXX ([25]) und Equator MAP-CA ([27]), da es sich hierbei in erster Linie um VLIW-Architekturen handelt, die aber mit speziellen Erweiterungen zu Mediaprozessoren weiterentwickelt wurden (siehe Abschnitt 2.4.7).

2.4.3 Superskalare Rechnerarchitekturen

Der Nachteil der zuvor beschriebenen VLIW-Architektur, nur statisches Scheduling der Befehle mit Hilfe des Compilers vornehmen zu können, führte zu der Entwicklung Superskalarer Rechnerarchitekturen. Hierbei wird versucht, die in dieser Architektur ebenfalls vorhandenen

parallelen Funktionseinheiten während der Laufzeit mit Befehlen zu beschicken und damit eine optimale, gleichzeitige Auslastung zu erzielen.
Die Anzahl der parallelen Einheiten hängt dabei von der jeweiligen Architektur ab, besteht aber in der Regel aus zwei bis drei Rechenwerken für Integer-Operationen und ebenso vielen für Fließkomma-Berechnungen. Abbildung 7 zeigt ein stark vereinfachtes Blockschaltbild des Rechenkerns eines Intel Pentium III-Prozessors mit fünf bzw. sechs parallelen Einheiten, wobei zwei bzw. drei Rechenwerke vorhanden sind.

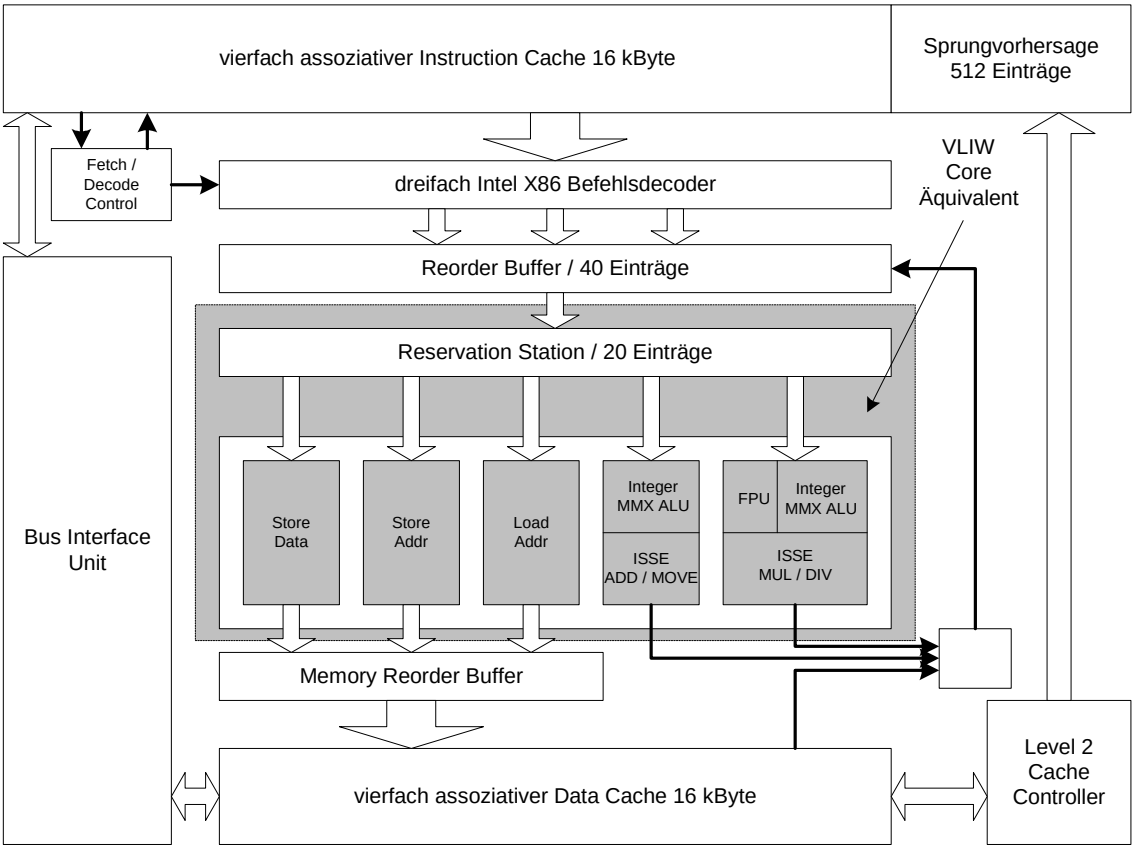


Abbildung 7 Blockschaltbild eines Intel Pentium III-Prozessors

Im Idealfall könnten diese Einheiten gleichzeitig arbeiten, jedoch wird aufgrund der Datenabhängigkeiten der jeweiligen Operationen und der speziellen Pipelinearchitektur des Prozessors in der Regel eine maximale Auslastung von zwei gleichzeitig arbeitenden Einheiten erreicht.
Je höher die Anzahl der parallelen Rechenwerke ist, desto höher ist die Wahrscheinlichkeit, dass durch die vorhandene Pipeline Hemmnisse entstehen, die aufgrund der Datenabhängigkeiten die parallele Befehlsausführung verhindern. Superskalare Rechnerarchitekturen werden häufig in Anlehnung an die Ursprungsarchitektur auch als dynamische VLIW-Architekturen bezeichnet.

2.4.4 Digitale Signalprozessoren

Digitale Signalprozessoren stellen eine Spezialform der unterschiedlichsten Prozessorarchitekturen dar. Die eigentliche Bezeichnung *digitaler Signalprozessor* rührt von dem ursprüng-

lichen Ziel, eine Hardwareplattform zu schaffen, die in Lage ist, digitale Filter mit hohem Datendurchsatz für Realzeitanwendungen zu realisieren. Dabei wurde das Grundprinzip eines digitalen Filters der Hardwarestruktur zugrunde gelegt, das als prinzipielle Struktur in Form eines n-Tap FIR-Filters in Abbildung 8 dargestellt ist.

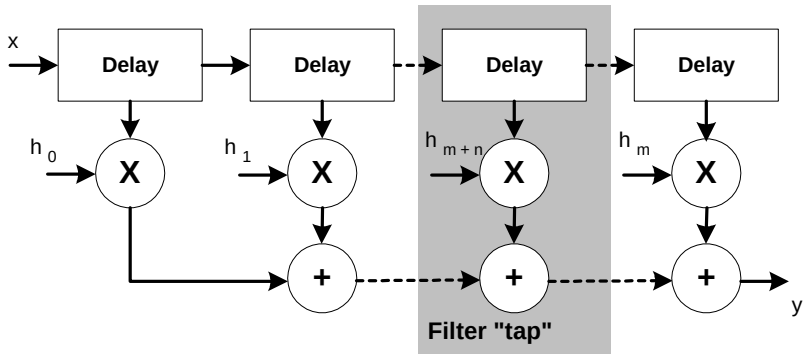


Abbildung 8 Grundprinzip eines n-Tap FIR-Filters

Hieraus ist ersichtlich, dass es sich um eine reguläre Struktur mit, je nach Anzahl der Filter-Taps, identischen Funktionsblöcken handelt. Die Hauptkomponenten eines Funktionsblocks stellen dabei ein Verzögerungsglied bzw. eine Speichereinheit, ein Multiplizierer und ein Addierer dar, wovon der Begriff der MAC[13]- Einheit abgeleitet wurde.
Die Architektur einer entsprechenden Verarbeitungseinheit ist in Abbildung 9 wiedergegeben.

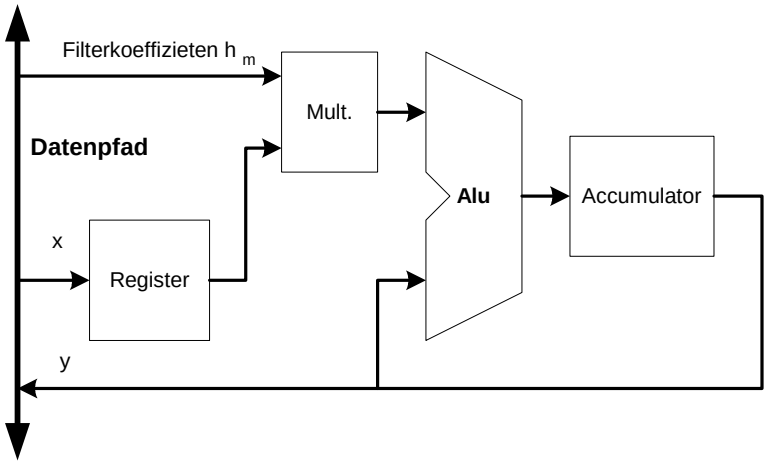


Abbildung 9 Grundprinzip einer Multiply and Accumulate-Einheit

Wird nun die MAC-Einheit um die prozessorspezifischen Komponenten, wie Adresszähler, Sequenzer, Befehlsdecoder etc., erweitert, erhält man die Grundstruktur eines digitalen Signalprozessors.
Als Prozessorarchitektur wurde ursprünglich die Harvard-Architektur aufgrund ihrer Teilung von Instruktionspfad und Datenpfad favorisiert, wobei eine Spezialform dieser Architektur, die nochmals eine Trennung des Datenpfads in zwei separate Pfade vorsieht, sich als beson-

[13] engl. MAC – Multiply and Accumulate

ders günstig für die Umsetzung des MAC-Befehls erweist und als erweiterte Harvard-Architektur bezeichnet wird (siehe hierzu [21]).
Typische Vertreter solcher Architekturen stellen die Prozessoren DSP56000 von Motorola und TMS320C40 von Texas Instruments dar, die neben den MAC-Befehlen über einen einfachen Befehlssatz verfügen.
Teilweise werden auch Prozessoren aufgrund ihrer hohen Leistungsfähigkeit in Bezug auf die parallele Verarbeitung von numerischen Daten als digitale Signalprozessoren bezeichnet, obwohl der charakteristische MAC-Befehl nicht vorhanden ist. Dabei dominieren die schon in Abschnitt 2.4.2 beschriebenen VLIW-Architekturen.

2.4.5 Splitted ALU / SIMD-Extensions

In der Regel arbeiten herkömmliche Prozessoren mit einer festen Datenpfadbreite, d.h. es werden alle Datenoperationen mittels der maximal zur Verfügung stehenden Busbreite ausgeführt. Prozessorinterne Registersätze und die Datenverarbeitungseinheiten des Prozessors, wie ALU etc., arbeiten dann ebenfalls mit gleicher Datenpfadbreite. In vielen Fällen der digitalen Bildsignalverarbeitung hat es sich aber herausgestellt, dass die maximal notwendige Datenpfadbreite, die sich aus der notwendigen Genauigkeit der Rechenoperationen ergibt, wesentlich kleiner ist als die meistens zur Verfügung stehende Wortbreite.
Auf einer typischen 32-Bit-Prozessorarchitektur wird z.B. für die Addition zweier 8-Bit-Werte jeweils ein 32-Bit-Register pro Operand benötig, wobei die restlichen 24 Bit ungenutzt bleiben. Die ausführende ALU wird ebenfalls nur zu 25% ausgelastet, da die restlichen 24 Bit der Operandenregister den Wert null beinhalten.
Aufgrund der Architektur einer ALU kann jedoch ohne wesentliche Änderungen eine Aufteilung in kleinere Einheiten mit kleinerer Datenpfadbreite vorgenommen werden. Werden nun ebenfalls die Operandenregister wie mehrere einzelne Register mit geringerer Bitbreite betrachtet, entsteht die sog. Splitted ALU-Architektur.
Abbildung 10 veranschaulicht nochmals die Grundidee einer Splitted ALU-Architektur (siehe hierzu auch [16]).

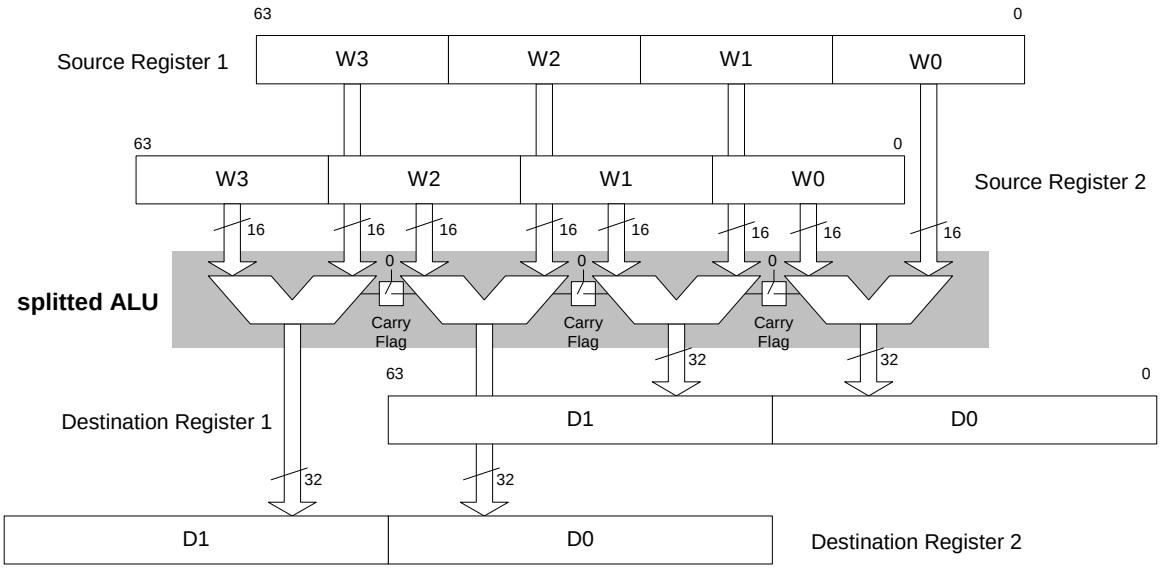


Abbildung 10 Grundprinzip einer Splitted ALU

Eine der wichtigsten realen Umsetzungen dieser Architektur stellen die MMX- bzw. ISSE-Erweiterungen der Intel Pentium-Prozessoren dar. Hierbei wird die Aufteilung der 64 Bit breiten MMX-Register in vier verschiedene Datentypen implizit durch die Wahl der jeweiligen MMX-Befehle vorgenommen. Dabei stehen folgende Datentypen für ein MMX-Register zur Verfügung:

- 8 x Packed Byte
- 4 x Packed Word
- 2 x Packed Double Word
- 1 x Quad Word .

Das hier angewandte Prinzip der parallelen Verarbeitung von mehreren Datensätzen wird auch allgemein als SIMD-Architektur (Single Instruction Multiple Data) bezeichnet. Die MMX-Architektur wurde von Intel zur ISSE- und ISSE2-Architektur weiterentwickelt, wobei es sich hauptsächlich um die Ergänzung weiterer Befehle und zusätzlicher Register handelt, die gegenüber der MMX-Architektur auf 128 Bit verbreitert sind und für die Nutzung der Splitted ALU-Befehle zur Verfügung stehen. Das grundsätzliche Konzept wurde jedoch beibehalten.
Je nachdem, ob die volle mögliche Datenbusbreite gewünscht ist oder ob auf den Einzeldatenpfaden gerechnet wird, werden die entsprechenden Schalter für die Weiterleitung des Übertrag-Flags gesetzt.
Neben den Prozessoren der Intel Pentium-Familie haben auch andere Prozessorhersteller ähnliche Konzepte verwirklicht, die auf der Erweiterung der Prozessorarchitektur um SIMD-Konzepte basieren. Die einzelnen Implementierungen unterscheiden sich dabei hauptsächlich in den unterschiedlich umfangreichen Befehlsvorräten, die teilweise direkt auf die Verarbeitung von Videodaten zugeschnitten sind. Hier sind folgende Prozessoren zu nennen:

- SUN VIS, Visual Instruction Set, näher beschrieben in [34]
- DEC Alpha 21264 SIMD integer Motion Video Instructions MV1
- Motorola MPC7400 (G4) 128 Bit SIMD AltiVec-Extension .

Weitere Informationen zur Leistungsfähigkeit der MMX-Architektur können in [19] gefunden werden.

2.4.6 Coprozessorerweiterung

Um die Umsetzung von speziellen Algorithmen auf einfachen Prozessoren zu ermöglichen, die durch ihren beschränkten Befehlsvorrat oder die Beschränkungen der Architektur eine effiziente Verarbeitung nicht gestatten, wird teilweise der Weg beschritten, dem Prozessor einen Coprozessor hinzuzufügen.
Der Coprozessor wird dabei meist als hochoptimierter Hardwarefunktionsblock umgesetzt, der dann ausschließlich in der Lage ist, eine spezielle Funktion mit einem Höchstmaß an Verarbeitungsgeschwindigkeit zur Verfügung zu stellen.
Die Kopplung des Coprozessors mit dem Hauptprozessor erfolgt in der Regel über gemeinsam genutzte Prozessorregister bzw. Speicherbereiche. Die Einbindung in den Befehlsvorrat kann über spezielle Coprozessorbefehle ausgeführt und damit in den regulären Programmfluss eingebunden werden.

Im Bereich der Bildsignalverarbeitung werden häufig die Funktionsblöcke DCT/IDCT als Coprozessoren angetroffen, aber auch andere Funktionen, wie z.B. Bitstromverarbeitung oder Bewegungsschätzung, können als Coprozessoren umgesetzt werden.
Ein Beispiel hierfür stellt der auf einer VLIW-Architektur basierende Mediaprozessor TM1300 von Philips dar. Aufgrund seiner Architektur ist die Umsetzung von stark irregulären Verarbeitungsvorgängen, wie die VLD-Decodierung, nur mit sehr schlechten Performanceeigenschaften möglich, da hier die massive Parallelität des Prozessors nicht ausgenutzt werden kann.

2.4.7 Media-Prozessorarchitekturen

Die Vertreter der Mediaprozessoren sind, ähnlich der Gruppe der digitalen Signalprozessoren, nicht mehr durch eine spezielle Architektur gekennzeichnet, sondern vielmehr nach ihrem typischen Anwendungsbereich der Multimediadatenverarbeitung benannt.
Dabei hat sich aufgrund der kostengünstigen Umsetzbarkeit die Gruppe der VLIW-basierten Prozessoren etabliert, die häufig noch durch spezielle Coprozessoren ergänzt werden. Neben den reinen architekturspezifischen Prozessorerweiterungen werden diese Prozessoren in der Regel mit einer Reihe von Schnittstellen ausgestattet, die es zulassen, den Prozessor als eigenständiges System einzusetzen und im Wesentlichen für die Ein- und Ausgabe von Audio- und Videodaten vorgesehen sind.
Der Bedarf nach speziellen Prozessoren für die Verarbeitung von Videodaten, unabhängig von leistungsfähigen Rechnerarchitekturen, wie z.B. Workstations und PCs, veranlasste Philips dazu, den der ersten kommerziell gefertigten Mediaprozessor zu entwickeln, der erstmals in [25], [26] und [35] vorgestellt wurde.
Der Prozessorkern besteht aus einem VLIW-Prozessor, der in seinem grundsätzlichen Aufbau der in Abbildung 6 bereits dargestellten Architektur entspricht. Die Anzahl der einzelnen funktionalen Einheiten beläuft sich hierbei auf 27, von denen 5 gleichzeitig durch eine VLIW-Instruktion adressiert werden können.
Der Prozessor verfügt neben den typischen Integer-ALU-Operationen auch über Einheiten für die Verarbeitung von Floating-Point-Daten, die den Prozessor ebenfalls für die Audiosignalverarbeitung geeignet machen. Sämtliche Operationen werden in mehreren Pipeline-Schritten bearbeitet, wodurch im Idealfall pro Prozessortakt 5 Befehle gleichzeitig ausgeführt werden können. Die VLIW-Befehlsworte werden mit variabler Länge im Speicher- bzw. Befehls-Cache des Prozessors abgelegt und können eine maximale Länge von 42 Bit aufweisen.
Für besonders registerintensive Rechenvorgänge wurde ein Registerfile implementiert, das über 128 32-Bit-Register verfügt.
Neben diesen Registern kann der Prozessor mit Hilfe seines 32 Bit breiten Adressbusses auf einen linearen Speicherbereich zugreifen. Der Befehls-Cache weist eine Größe von 32 kByte auf, während der Daten-Cache eine Größe von 16 kByte aufzeigt. Die einzelnen Befehle des Prozessors ähneln stark herkömmlichen RISC-Befehlen, wobei jeder Befehl in Abhängigkeit vom Zustand eines der General-Purpose-Register ausgeführt werden kann (conditional execution). Einige Befehle weisen die in 2.4.5 beschriebenen SIMD-Konzepte auf und ermöglichen somit einen nochmals gesteigerten Datendurchsatz.
Eine ähnliche Struktur zeigt die in Abbildung 11 dargestellte und in [27] und [28] näher beschriebene Architektur des MAP-CA[14] von Equator.

[14] Media-Accelerated-Processor for Consumer-Applications

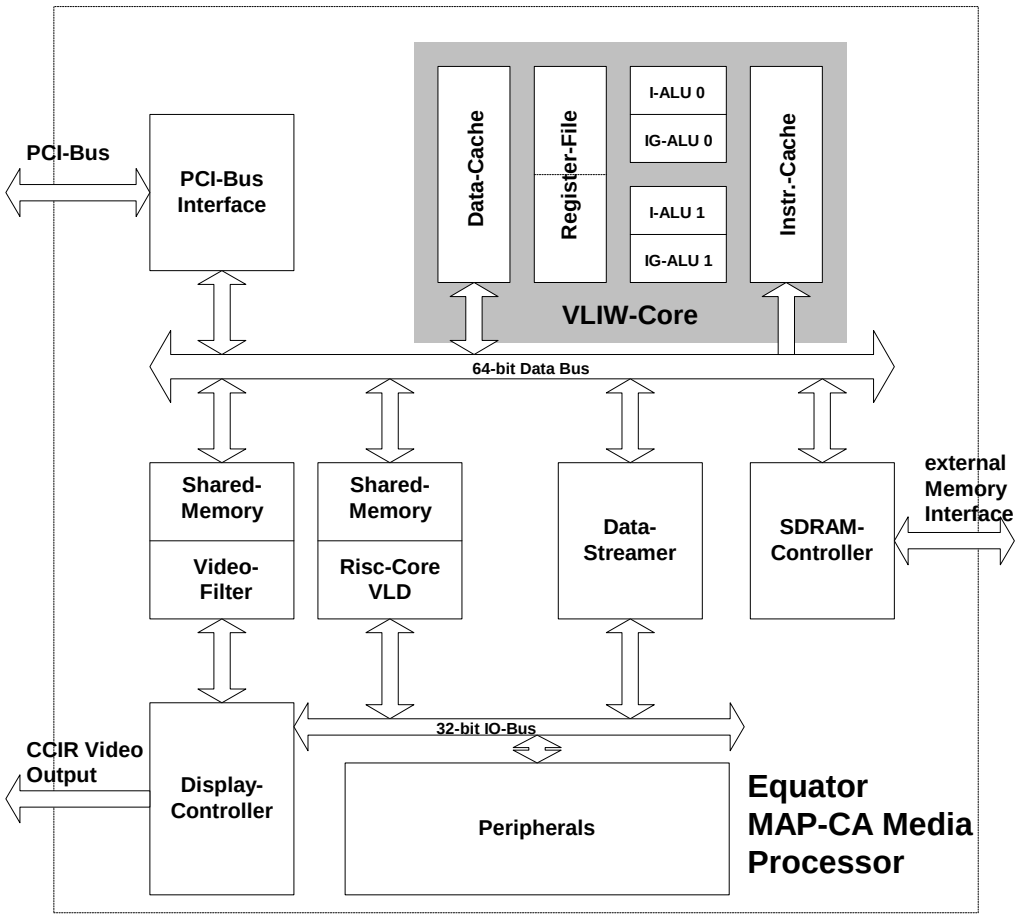


Abbildung 11 Blockschaltbild des Equator Mediaprozessors MAP-CA

Hierbei handelt es sich wiederum um einen VLIW-Prozessorkern, der um einige verarbeitungsspezifische Komponenten erweitert wurde. Die Architektur des Prozessorkerns unterscheidet sich jedoch von der des Trimedia, indem es sich um zwei identische VLIW-Verarbeitungseinheiten handelt, die zu einem Prozessorkern vereint wurden. Somit sind vier Opcodes parallel verarbeitbar. Beide Prozessorteile habe Zugriff auf ein eigenes Registerfile mit jeweils 64 32-Bit-Registern und können, ähnlich der Architektur des Trimedia, 32-Bit-Integer, 32- und 64-Bit Load-/Store- und Floating-Point-Operationen durchführen.
Ein VLIW-Datenwort hat eine Breite von 136 Bit und beinhaltet damit vier Opcodes mit jeweils 34 Bit Breite. Der Prozessorkern bietet ebenfalls die Möglichkeit, SIMD-Verarbeitung vorzunehmen, wobei von einer Maximalgröße von 64 Bit für ein Register ausgegangen wird, das in maximal acht 8-Bit-Datenpfade aufgeteilt werden kann.
Ein Novum stellt eine spezielle Verarbeitungseinheit für die Implementierung von FIR-Filtern dar, wobei ein besonderer Registersatz für die Speicherung fester Koeffizienten zur Verfügung steht. Ähnlich dem in Abschnitt 3.4.2 beschriebenen Konzept auf Basis der Distributed Arithmetic-Architektur kann die Berechnung von Partialprodukten erfolgen.
Neben dem Prozessorkern verfügt der Baustein über einen frei programmierbaren 16-Bit-RISC-Controller, der hauptsächlich für Kontrollflussaufgaben vorgesehen ist. Für einen effizienten Datenfluss sorgt der 64 Bit breite Datenbus, der externen SDRAM-Speicher und weitere Peripheriekomponenten mittels eines eigenen DMA-Controllers an den VLIW-Prozessorkern anbindet.

Neben den vorgestellten Prozessoren gibt es eine Reihe weiterer Prozessoren, die teilweise aus marktstrategischen Gründen des jeweiligen Herstellers der Gruppe der Mediaprozessoren zugeordnet werden. Hierbei sind in erster Linie die VLIW-Prozessoren der C6x-Serie von Texas Instruments (TI) zu nennen, die bei ihrer Markteinführung als High-Performance-DSPs vermarktet wurden und im Laufe der Zeit aufgrund ihrer hohen Rechenleistung von TI als Mediaprozessoren bezeichnet wurden.
Die Prozessoren der TMS320C6400-Reihe verfügen dabei über eine SIMD-Befehlssatzerweiterung. Eine detaillierte Beschreibung hierzu findet sich in [24]. Die Eignung der TI-C6x-Prozessoren als Mediaprozessoren wurde vom Verfasser anhand der Implementierung eines softwarebasierten MPEG-4-Decoders untersucht und in [73] näher beschrieben.
Ein ähnliches Konzept ist in [32] vorgestellt, bei dem ebenfalls ein VLIW-Prozessor mit einem SIMD-Befehlssatz ausgestattet wurde.
Weiterhin existieren eine Reihe akademisch motivierter Architekturen, wie z.B. die in [31] und [76] beschriebene Makroblock-Engine. Hierbei handelt es sich um einen programmierbaren Coprozessor, dessen Befehlssatz neben den typischen SIMD-Konzepten an den Belangen der Videosignalverarbeitung ausgerichtet ist und beispielsweise dedizierte Vektorverarbeitungsbefehle bietet.
Das in [33] vorgestellte Konzept verwendet eine Anzahl von einzelnen Prozessorelementen, die parallel wie auch seriell für die gemeinsame Verarbeitung von Multimediadaten verschaltet werden können.

2.4.8 Application Specific Instruction Processor ASIP

Im Gegensatz zu den vorhergehend beschriebenen Prozessorkonzepten, die in Bezug auf den implementierten Befehlssatz auch die Hauptmerkmale von General-Purpose-Prozessoren aufweisen, stellen sog. *Application Specific Instruction Processors* Implementierungen spezieller Prozessoren dar, die ausschließlich für ein begrenztes Anwendungsfeld konzipiert sind. Da die Anwendungsfälle sehr unterschiedlich sind und der Befehlssatz meist direkt durch das spezielle Problem definiert wird, werden ASIPs mit Hilfe spezieller Softwarewerkzeuge vom Designer selbst entworfen bzw. spezifiziert.
Hierfür gibt es zwei verschiedene Ansätze, die die Synthese entsprechender Prozessoren ermöglichen.
Zum einen kann vom Designer eine komplette Spezifikation erstellt werden, womit der Befehlssatz eindeutig definiert und anhand der festgelegten Parameter eine spezielle Prozessorarchitektur erstellt wird.
Die zweite Variante stellt die Anpassung einer gegebenen Architektur dar, die somit hauptsächlich in Bezug auf den implementierten Befehlssatz angepasst werden kann. Hierzu ist es meist nur notwendig, den Basisbefehlssatz mit den erforderlichen Parametern, wie z.B. Anzahl der Register und Operandenbreite und die grundlegenden Architekturmerkmale, wie die Anzahl der Pipeline-Stufen, anzugeben. Auf der Grundlage dieser Datenbasis wird anschließend ein synthetisierbarer Prozessorkern in Form einer Hardwarebeschreibungssprache erstellt. Nachfolgend sind zwei ASIP-Umsetzungen kurz umrissen:

- **Tensilica / Xtensa**

  Der Prozessorkern Xtensa LX der Firma Tensilica (siehe auch [29]) stellt ein kommerzielles Produkt zum Aufbau von Spezialprozessoren dar. Dabei wird auf der sog. *Xtensa Instruction Set Architecture* aufgebaut, die sich durch 16- bzw. 24-Bit-Instruktionsworte auszeichnet und somit die Codegröße für eingebettete

Anwendungen drastisch reduziert. Die Architektur bietet in ihrer Grundausbaustufe eine 32-Bit-ALU, 64 General-Purpose-Register und verfügt über ca. 80 Basisinstruktionen. Mit Hilfe der *Tensilica Instruction Extension Language* kann der Entwickler den Befehlssatz in Form einer speziellen Beschreibungssprache eigenständig erweitern und anschließend den eigentlichen Prozessor mit Hilfe eines Core-Generators erzeugen. Hierbei wird nicht nur der Prozessorkern in Form von Verilog- oder VHDL-Modellen erzeugt, sondern ebenfalls die komplette, auf den Prozessor angepasste Softwareentwicklungsumgebung bereitgestellt.

- **ASIP-Meister**

  ASIP-Meister stellt ein Softwarepaket zum Entwurf von Spezialprozessoren dar. Hiermit können sämtliche Parameter eines Prozessors per Eingabemaske einer grafischen Benutzeroberfläche in eine Datenbasis eingegeben werden. Die Architektur wird jedoch nicht auf eine Basisarchitektur, wie im Falle des Xtensa LX zurückgeführt, sondern ist frei definierbar.
  Nach Vervollständigung sämtlicher Daten inkl. der Beschreibung des Befehlssatzes lässt sich der entworfene Prozessor in Form von VHDL-Modellen für verschiedene Anwendungen synthetisieren. Hierzu zählen Verhaltensmodelle für die reine Simulation wie auch RTL-Modelle für die ASIC- bzw. FPGA-Synthese.
  Das im Rahmen eines Hochschulprojektes an der Osaka Universität in Japan entworfene Softwarepaket ist für Forschungszwecke frei verfügbar und in [30] näher beschrieben.

2.4.9 Eingebettete Systeme / SOCs für Mobilfunkanwendungen

Die bisher vorgestellten Architekturen stellen eigenständige Prozessoren dar, die jedoch erst durch die Erweiterung um verschiedene weitere Komponenten, wie z.B. Speichercontroller, Schnittstellen etc., in ein Gesamtsystem integriert werden können. Für Anwendungen in höchstintegrierten Systemen, wie sie in mobilen Telefonen vorzufinden sind, werden jedoch keine dedizierten Einzelbausteine eingesetzt. Die Integration der Einzelkomponenten findet hierbei auf der Chip-Ebene statt.
Die Miniaturisierung im Bereich der Mobilfunkanwendungen erfordert dabei einen immer höheren Grad an Systemintegration, wobei nicht nur der Prozessor und die dazugehörige Peripherie auf einem Chip untergebracht werden, sondern auch noch sämtliche Komponenten der Sprachsignalverarbeitung und Teile des HF-Front-Ends. Aus diesem Grunde wurden von verschiedenen Halbleiterherstellern sog. Applikationsprozessoren entwickelt, die auf der Basis einfacher RISC-Prozessoren aufgebaut werden und damit Single-Chip-Implementierungen mobiler Endgeräte zulassen.
Um die Leistungsaufnahme möglichst niedrig halten zu können, werden zum einen spezielle Halbleiterprozesse verwendet, aber auch spezielle Verfahren angewandt, um die Taktfrequenz (clock scaling) den jeweiligen Anforderungen in Bezug auf die erforderliche Rechenlast anpassen zu können. Folgende Prozessoren sind aktuell (02/2004) am Markt verfügbar bzw. befinden sich in der Produktvorphase.

Name / verfügbar	**Kern-takt in MHz**	**Prozessor-kern**	**Interner Speicher (Cache)**	**Coprozesor/ Takt in MHz**	**Peripherie**
AMD Alchemy **Au1100** Q1/2004	<330 <400 <500	MIPS32 RISC Integer 32 Bit LD/ST	16kB I-Cache 16kB D-Cache	MAC/ Divide Unit	LCD/TFT, SDRAM, Flash, CF, Smart Card
Motorola **I.MX21** Q1/2004	<200	ARM926 RISC Integer 32 Bit LD/ST	16kB I-Cache 8kB D-Cache		LCD/TFT, SDRAM, Flash, CF, Smart Card
Texas In-struments **OMAP1612** Q4/2003	<200	ARM926 RISC Integer 32 Bit LD/ST	16kB I-Cache 8kB D-Cache	DSP TMS320C55 x @200MHz	LCD/TFT, SDRAM, Flash, CF, Smart Card
Intel **PXA800 F** Q3/2003	<312	Xscale Core ARM V5 ISA Integer 32 Bit LD/ST		Intel MSA DSP Core @104 MHz	LCD/TFT, SDRAM, Flash
Infineon/Sci-Worx **Sci-Cos 3000** Q3/2003	<200	ARM926EJ RISC Integer 32 Bit LD/ST	16kB I-Cache 8kB D-Cache	Color Space Conversion MPEG-4 Motion Estimation @200MHz	LCD/TFT, SDRAM, Flash

Tabelle 2 Applikationsprozessoren für Mobilfunkanwendungen im Überblick

Wie Tabelle 2 zu entnehmen ist, werden ausschließlich RISC-Prozessoren als Hauptprozessoren eingesetzt, wobei die vorherrschende Architektur durch die *ARM V5 Instruction Set Architecture* gebildet wird. Neben den typischen Schnittstellen, die der Systemintegration und der Speicheranbindung dienen, finden sich in fast allen Systemen unterschiedliche Coprozessoren wieder, die der Anwendung entsprechend eine Unterstützung des Hauptprozessors in Bezug auf die Verarbeitung von Multimediadaten bieten sollen.
Auffällig sind die unterschiedlichen Coprozessortypen, die zur Anwendung kommen. Grundsätzlich kann hier zwischen dedizierten Komponenten und programmierbaren Einheiten in Form von speziellen Signalprozessoren unterschieden werden. Genaue Angaben über die erzielte Verlustleistung ist von keinem Hersteller in Erfahrung zu bringen, da es sich hierbei stets um stark applikationsabhängige Angaben handeln würde.
Eine aussagekräftige Bewertung kann somit nur auf der Grundlage eines direkten Vergleiches gleicher Applikationen bei identischen Anwendungsszenarien erfolgen. Typischerweise dürfen jedoch die Verlustleistungen eines Bausteins in Mobilfunkendgeräten nicht über 100 mW liegen.

2.5 Betriebssysteme und Programmiersprachen

Prozessorbasierte Systeme werden in der Regel nicht nur ausschließlich für eine spezielle Applikation entwickelt, sondern gerade wegen der geforderten Flexibilität in Bezug auf verschiedene Anwendungen eingesetzt. Im Falle der Mobilfunkanwendungen ist hier in erster Linie die gesamte Bandbreite der üblichen Telefonanwendungen zu nennen, die durch die Textdienste, wie SMS und später durch weitere Multimedia-Dienste, ergänzt wurden. Neben den eigentlichen Applikationen werden noch eine Reihe verschiedener Systemdienste von dem eingesetzten Prozessor ausgeführt, beispielsweise Stromsparfunktionen, Speichermanagement, Prozess-Scheduling und dgl. Aus der Vielzahl der Anwendungen ergibt sich der Bedarf nach anwendungsorientierten Betriebssystemen, die zum einen der begrenzten Leistungsfähigkeit eingebetteter Systeme Rechnung tragen, auf der anderen Seite jedoch den kompletten Umfang eines vollständigen Betriebssystems bereitstellen müssen. In aktuellen mobilen Systemen haben sich im Wesentlichen folgende Betriebssysteme etabliert:

- **Symbian OS**

  Symbian OS, ein Produkt des englischen Unternehmens Symbian, stellt das z. Zt. meist genutzte Betriebssystem für Mobiltelefone dar und beruht auf einer eigenständigen Implementierung, die ausschließlich für den Einsatz in Systemen mit eingeschränkten Leistungsmerkmalen entwickelt wurde. Das Unternehmen wurde 1998 von Ericsson, Nokia, Motorola und Psion gegründet, um spezielle Software, ausschließlich für Mobiltelefone, zu entwickeln.

- **WinCE / Pocket PC**

  Auf der Basis der Microsoft WIN32 APIs wurde von Microsoft ein stark vereinfachtes Betriebssystem entworfen, das vornehmlich für Taschencomputer (PDAs) entwickelt wurde, wovon jedoch auch Varianten für Set-Top-Boxen, Autoradios und Mobiltelefone abgeleitet wurden. Aufgrund lizenzrechtlicher Bestimmungen hat WinCE jedoch bislang nicht den gewünschten Verbreitungsgrad gefunden.

- **Embedded Linux**

  Linux stellt eine zunehmende Konkurrenz für die etablierten Betriebssysteme in eingebetteten Systemen dar. Gründe dafür sind die zunehmend leistungsfähigeren Systeme, die über die erforderlichen Hardwarevoraussetzungen, wie eine Memory Management Unit (MMU), verfügen und damit UNIX-ähnliche Betriebsysteme unterstützen, und der kostengünstigere Masseneinsatz. Wie WinCE, ist Linux aber ebenfalls der breite Massenmarkt bisher verwehrt geblieben.

Im Bereich der prozessorbasierten Umsetzung von Bildsignalverarbeitungsalgorithmen hat sich im Wesentlichen die Programmiersprache *C* bzw. *C++* etabliert, die aufgrund ihrer einfachen Konstrukte eine maschinennahe Programmierung zulässt.
Dies ist besonders wichtig, wenn hohe Ansprüche an die Realzeit-Performance des Systems gestellt werden. Problematisch stellt sich in der Regel die Nutzung der prozessorspezifischen Multimediaerweiterungen dar, wie die in Abschnitt 2.4.5 vorgestellten SIMD-Erweiterungen. Hierbei erfolgt meist keine direkte Verwendung der spezifischen Maschinenbefehle durch den

Hochsprachencompiler, da die explizite Verwendung unterschiedlicher Datentypen durch den Compiler nicht erkannt werden kann. Dies kann nur mittels spezieller Compiler-Anweisungen oder durch die direkte Verwendung des Maschinencodes erfolgen.
Für hochperformante Algorithmenteile wird deshalb häufig auf die direkte Programmierung mittels Assemblercode zurückgegriffen, die dann in Form von Bibliotheksfunktionen verwendet werden können. Einige Prozessorhersteller bieten für einen speziellen Prozessor optimierte Bibliotheksfunktionen an, wie z.B. die Intel Performance Libraries ([64]), die eine Reihe von Grundalgorithmen der Bildsignalverarbeitung enthalten und mit Hilfe von MMX- bzw. ISSE-Optimierungen umgesetzt wurden.

2.6 Methoden zur Performancebestimmung

Neben der Klassifizierung und der theoretischen Betrachtung von geeigneten Architekturen für die Videosignalverarbeitung stellt ein weiterer wichtiger Aspekt für die Bewertung von programmierbaren Systemen die Performanceanalyse dar, die der Bestimmung der Leistungsfähigkeit einer Prozessorarchitektur in Hinblick auf die erzielbare Verarbeitungsgeschwindigkeit ausgewählter Algorithmen dienen soll.
Für die Performancebestimmung haben sich verschiedene Konzepte etabliert, die zum einen auf speziellen Benchmarkprogrammen basieren oder zum anderen eine gezielte Untersuchung der eigentlichen Applikation auf dem Zielprozessor mit Hilfe von Profilingwerkzeugen ermöglichen.

2.6.1 Benchmarkprogramme

Die Untersuchung unterschiedlicher Prozessorarchitekturen mit Hilfe von Benchmarkprogrammen stellt einen Kompromiss aus der möglichst gut angenäherten Abbildung des jeweiligen Algorithmus auf die zu untersuchende Prozessorarchitektur und einer großen Anzahl von unterstützten Prozessoren dar. Hierbei wird in Anbetracht der unterschiedlichen Prozessorarchitekturen deutlich, dass aufgrund der oft fehlenden bzw. ungenügenden Compilerunterstützung die meisten Architekturoptimierungen nur mit Hilfe von dediziertem Assemblercode verwendet werden können. Benchmarkprogramme müssen hingegen in einer unabhängigen Programmiersprache erstellt sein, um möglichst viele Prozessoren unterstützen zu können, also um nahezu plattformunabhängig eingesetzt zu werden.
Ein weiteres Problem geeigneter Benchmarkprogramme stellt die Auswahl und Zusammenstellung der verwendeten Algorithmen dar, die für die Untersuchung herangezogen werden. Herkömmliche Benchmarkprogramme, wie z.B. SPECint92 oder SPECint95, setzen sich in der Regel aus Algorithmen zusammen, die die jeweilige Prozessorarchitektur mit einem möglichst breiten Spektrum unterschiedlicher Verfahren untersuchen, die naturgemäß ungeeignet sind, Aussagen über spezielle Eigenschaften treffen zu können.
Die ebenfalls häufig verwendeten Angaben der Prozessorperformance in der Einheit MIPS[15] bzw. MOPS[16] lassen in der Regel keine Aussage über die reale Prozessorperformance zu, da Operationen oder Instruktionen auf unterschiedlichen Architekturen unterschiedliche Effizienz haben können. Weitere Ausführungen hierzu finden sich in [36].
Eine ähnliche Problematik stellt sich für die Beurteilung von Signalprozessoren dar, wo häufig eine Angabe der Performance in der Einheit MACS[17] vorgenommen wird. Hieraus ist ebenfalls keine genaue Aussage über die Performance des Prozessors abzuleiten, da dabei

[15] MIPS - Million Instructions per Second
[16] MOPS - Million Operations per Second
[17] MACS - Multiply Accumulates per Second

vollständig unberücksichtigt bleibt, wie schnell nicht-MAC-basierte Operationen abgearbeitet werden oder ob z.B. Instruktionen parallel zu MAC-Befehlen ausgeführt werden können.
Um die Leitungsfähigkeit von Signalprozessoren trotzdem angemessen beurteilen zu können, wurden spezielle Benchmarkprogramme entwickelt, die sich aus einer Reihe der am häufigsten verwendeten Signalverarbeitungsalgorithmen zusammensetzen, wie z.B. FIR-Filter, die Fast Fourier-Transformation und dgl.
Für die Ermittlung aussagekräftiger Ergebnisse für Videoprozessorarchitekturen müssen jedoch videoverarbeitungsspezifische Algorithmen berücksichtigt werden, wie z.B. Bewegungskompensation oder Bewegungsschätzung. Ein Ansatz hierzu stellt der in [37] beschriebene Benchmark *Berkeley Multi Media Workload* dar, der sich aus einer Reihe von verschiedenen multimediaspezifischen Algorithmen bzw. kompletten Applikationen zusammensetzt, wie beispielsweise MPEG-2-Videocodierung, GSM- und MP3-Audiocodierung.
Vollständig unberücksichtigt bleibt in der Regel der Einfluss der Architektur des Systems in das der untersuchte Prozessor eingebettet ist, da meistens keine Aussage über die Art der Speicherzugriffe bzw. der möglichen Speicherbandbreite getroffen wird, d.h. Benchmarkprogramme untersuchen in der Regel ganze Systeme und nicht nur ausschließlich den verwendeten Prozessor.
Zusammenfassend ist zu sagen, dass der Einsatz von Benchmarkprogrammen für die Untersuchung der Eignung von Prozessorarchitekturen für videosignalverarbeitungsspezifische Anwendungen meist zu keinen repräsentativen Ergebnissen führt, sondern nur sehr ungenaue Resultate liefert.

2.6.2 Profiling

Eine weitere Variante, Prozessorarchitekturen zu beurteilen, stellt das sog. Profiling dar. Im Gegensatz zu den vorhergehend beschriebenen Benchmarkverfahren wird hier ausschließlich die Zielapplikation auf dem Zielprozessor untersucht. Dabei wird in der Regel mit Hilfe des verwendeten Hochsprachencompilers ein sog. Instrumentationcode in den eigentlichen Programmcode integriert, um z.B. Zeitmessungen zur Bestimmung der Verarbeitungszeiten starten und stoppen zu können. Je nach verwendetem Profilingverfahren wird der somit modifizierte Programmcode ausgeführt und generiert zur Ausführungszeit eine Zusammenfassung über die Laufzeit der einzelnen Funktionen des Programms, das sog. Laufzeitprofil.
Das beschriebene Konzept kann sowohl auf Programme angewendet werden, die unter der Kontrolle eines Betriebssystems ablaufen, wie auch auf Programmcode, der in einer Softwaresimulationsumgebung ausgeführt wird. Je nach Ausführungsplattform können dann die Ergebnisse zyklengenau oder auf eine Zeitbasis bezogen angegeben werden.
Der häufig verwendete Profiler *gprof* stellt einen typischen Vertreter dieser Art dar und ist im Rahmen der Open Software Foundation verfügbar (siehe hierzu [39]).
In [38] wird eine Methode vorgestellt, die mit Hilfe eines Hardware-Timers die Ausführungszeiten eines PC-basierten MPEG-2-Decoders misst, wobei der Instrumentationcode manuell in den Programmcode eingefügt wurde.
In [40] ist die Implementierung und Anwendung des auf dem *gprof* basierenden Instructionprofilers *iprof* beschrieben. Hierbei werden lineare Codesegmente ohne Sprungbefehle, sog. *Basic Blocks*, mit Hilfe eines Disassemblers auf ihre Zusammensetzung aus arithmetischen Operationen und Speicherzugriffsbefehlen untersucht. Anschließend wird im Rahmen der Laufzeituntersuchung gemessen, wie oft diese linearen Codesegmente ausgeführt wurden. Somit gelangt man zu einer Aussage über die Anzahl der Rechenoperationen und programmgesteuerten Speicherzugriffe. Unberücksichtigt bleibt hierbei die Zugriffscharakteristik auf den Programmspeicher des Systems wie auch die unterschiedliche Perfor-

mance bzw. Effizienz der einzelnen Operationen auf unterschiedlichen Prozessorarchitekturen.
Das Erstellen eines Laufzeitprofils mit Hilfe von automatisch eingefügtem Instrumentationcode birgt jedoch noch einige weitere Nachteile in Bezug auf die Aussagekraft der Ergebnisse. In der Regel schließen sich Compileroptionen für die Optimierung des Codes und die Generierung von Instrumentationcode gegenseitig aus, d.h. es kann meist nur unoptimierter bzw. schwach optimierter Code untersucht werden. Da jedoch gerade durch den bewussten Einsatz spezieller Compileroptimierungen teilweise hohe Laufzeitgewinne erzielt werden können, kann das somit erstellte Laufzeitprofil von unoptimiertem Code nur als Startpunkt für weitere Untersuchungen herangezogen werden.
Ein weiteres Problem stellt der Einfluss der verwendeten Zielplattform dar. Für den Fall der Ausführung des zu untersuchenden Programms unter der Kontrolle eines Betriebssystems muss mit starken Variationen zwischen mehreren Profilingläufen bei sonst gleichen Parametern gerechnet werden, da die Betriebssystemeinflüsse, wie z.B. Scheduling-Parameter, das Laufzeitverhalten des zu untersuchenden Programms stark verfälschen können.
Weiterhin kann keine Aussage über die Speicherzugriffscharakteristik getroffen werden, da nur ein Laufzeitprofil für die untersuchten Codesegmente angefertigt, jedoch keine Unterscheidung zwischen Speicherzugriffsoperationen, arithmetischen und Kontrolloperationen vorgenommen wird.

2.7 Schlussfolgerung

Die vorhergehenden Abschnitte haben den Stand der Technik im Bereich der Nutzung und Implementierung von mobilen Multimediadiensten näher beleuchtet.
Hierbei ist festzustellen, dass mobile Dienste in Zukunft in immer größerem Maße auch durch Videoinhalte geprägt sein werden.
Entsprechende Standards hierzu werden z. Zt. entwickelt oder sind bereits etabliert. Auch die hierbei verwendeten Videocodierungsstandards sind weitgehend festgelegt und im Wesentlichen auf die Standards ISO-MPEG-4 (inkl. MPEG-4 Part 10, H.264/AVC) bzw. ITU-H.263 zurückzuführen.
Für die echtzeitfähige Umsetzung dieser Standards kommen grundsätzlich verschiedene Varianten in Frage, die aber aufgrund unterschiedlicher Nachteile in Bezug auf die Spezifika mobiler Endgeräte ungeeignet sind.
Hierbei zeigt sich in erster Linie der hohe Stromverbrauch leistungsfähiger Prozessoren als größter Nachteil. Konzepte, wie z.B. VLIW-Prozessoren oder Superskalare Architekturen, sind hiervon in besonderem Maße betroffen, obwohl gerade sog. Mediaprozessoren auf diese Architekturkonzepte zurückgreifen.
Ebenfalls nachteilig wirkt sich die hohe Komplexität der angesprochenen Architekturen in Bezug auf den Flächenbedarf aus, der eine kostengünstige Umsetzung erschwert.
An dritter Stelle ist die notwendige Flexibilität zu nennen, die wiederum von verlustleistungsärmeren Implementierungen, z.B. in Form von dedizierten Komponenten, nicht erbracht wird.
Aufgrund des hohen Integrationsgrades werden in zunehmendem Maße immer mehr Funktionalitäten von eingebetteten Systemen gefordert, so beispielsweise Betriebssysteme und weitere, meist softwarebasierte Anwendungen.
Aus den genannten Gründen werden z. Zt. Applikationsprozessoren in erster Linie auf der Basis einfacher RISC-Prozessoren aufgebaut, die in Form von SOCs über weitere dedizierte Komponenten zur optimierten Verarbeitung von Multimediadaten verfügen.
Für eine effizientere Verarbeitung von Bildcodierungsverfahren stehen jedoch bereits weitere Ansätze zur Verfügung, die in Form von Befehlssatzerweiterungen auf der Basis von SIMD-Architekturen einen hohen Gewinn bezogen auf das Echtzeitverhalten versprechen, ohne die vorab geforderten Randbedingen in Frage zu stellen.
Hierbei muss eine Balance in Bezug auf Leistungsfähigkeit, Flexibilität und Verlustleistung zwischen ASIC, ASIP, GP-Prozessor und DSP gefunden werden, die somit die ideale Architektur für entsprechende Anwendungen zur Folge hat.
Um performante Architekturerweiterungen zu entwerfen, die ein Höchstmaß an Kosten-/Nutzenverhältnis erbringen, ist es erforderlich, die verwendete Applikationssoftware genau untersuchen zu können. Hiermit kommt den Profiling- und Benchmarkverfahren eine besondere Bedeutung zu, da sie in großem Umfang die Designentscheidung mit beeinflussen.

3 Grundlagen und Algorithmen der Bilddatenkompression

Das Kapitel gliedert sich in zwei Abschnitte, in denen die theoretischen Grundlagen für die folgenden Kapitel erläutert werden. Ausgehend von der Beschreibung der wesentlichen Einzelverfahren zur Bilddatenkompression wird auf die Funktionsweise hybrider Codierungsverfahren eingegangen. Der zweite Abschnitt wird aus einem Überblick über verschiedene Implementierungsvarianten der vorgestellten Algorithmen gebildet.

3.1 Entropiecodierung

Neben dem Einsatz in der Bildsignalverarbeitung stellt die Entropiecodierung eine der wichtigsten Verfahren zur verlustfreien Codierung von digitalen Signalen dar. Dabei wird das Quellensignal, das durch ein Quellenalphabet $A=\{a_1,a_2,a_3,a_4,...,a_n\}$ repräsentiert wird, mit Hilfe eines Repräsentationsalphabets $B=\{b_1,b_2,b_3,...,b_n\}$ übertragen bzw. codiert. Das Ziel der Codierung ist dabei, die Symbole des Quellenalphabets mit einer möglichst geringen Anzahl von Bits darzustellen, wobei Codes mit variabler Länge (VLC) entstehen. Hierbei wird jedes Symbol des Quellenalphabets auf mindestens ein Symbol des Repräsentationsalphabets abgebildet, um die verlustfreie Codierung des Quellensignals zu gewährleisten. Für die Zuordnung des Quellensymbols auf das entsprechende Symbol des Repräsentationsalphabets wird die Auftrittswahrscheinlichkeit des Quellensymbols herangezogen. Dabei gilt, dass ein häufig auftretendes Ereignis einen höheren Informationsgehalt besitzt als ein selten auftretendes Ereignis. Daraus ergibt sich, dass die minimale Rate, mit der das Quellensignal codierbar ist, dessen mittlerer Informationsgehalt, die Entropie $H(A)$, ist. Ziel ist es, annähernd mit der Rate der Entropie zu übertragen. Die Entropie für das Quellenalphabet A mit den Wahrscheinlichkeiten p der Symbole von A lautet dabei:

$$H(A) = -\sum_{i=1}^{n_a} p(a_i)\log_2(p(a_i)) \ . \tag{1}$$

Die mittlere Codewortlänge l_{avg} ergibt sich als gewichteter Mittelwert der Codewortlängen l_{CW} mit den Auftrittswahrscheinlichkeiten der Symbole des Quellenalphabets zu:

$$l_{avg} = \sum_{i=1}^{n_s} p(a_i) l_{cw}(i) \ . \tag{2}$$

3.1.1 Tabellenbasierte Codierung

Die einfachste Art der Entropiecodierung stellt der Einsatz von Codeworttabellen dar, womit nach bestimmten Verfahren die Zuordnung eines Quellenalphabetsymbols auf ein Repräsentationssymbol vorgenommen wird. Es muss jedoch eine umkehrbar eindeutige Zuordnung der Symbole gewährleistet sein, d.h. die Tabelle muss eineindeutig sein.
Eine der wichtigsten Umsetzungen der tabellenbasierten Entropiecodierung stellt die Huffmancodierung dar. Sie beruht auf einem Algorithmus für die Berechnung einer von Huffman entwickelten Codetabelle für Präfixcodes bei gegebenen Auftrittswahrscheinlichkeiten. Ein Präfixcode zeichnet sich durch die eindeutige Signalisierung des Codewortendes durch ein

Endbit aus. Für die Erstellung des Codebaums wird eine Liste der Wahrscheinlichkeiten des Auftretens der Symbole ($p(a_n)$) aufgestellt. Mittels eines speziellen Algorithmus wird nun jedem Symbol des Quellenalphabets eine Bitfolge zugeordnet. Typisches Merkmal eines Huffman-Codebaums ist dabei die Zuordnung der kürzesten Bitfolge, d.h. dem jeweiligen Symbol des Repräsentationsalphabets *B* wird das Symbol des Quellenalphabets mit der höchsten Auftrittswahrscheinlichkeit zugeordnet. Der Vorteil der Implementierung von Entropiecodern mit Hilfe von Huffman-Codes stellt die eindeutige Decodierung des Quellensymbols aus dem übertragenden Repräsentationssymbol dar, wobei jedes Symbol für sich eigenständig und eindeutig decodiert werden kann, ohne vorhergehend übertragende Symbole berücksichtigen zu müssen. Abbildung 12 stellt exemplarisch die Entwicklung eines Huffman-Codebaums für ein aus drei Symbolen bestehendes Quellenalphabet dar.

Symbol	VL-Code
A	1
B	01
C	0001

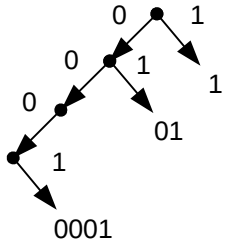


Abbildung 12 a: Huffman Codetabelle, b: Codebaum

Neben dem Huffman-Code gibt es eine Reihe anderer Verfahren, die die Erstellung von speziellen Codetabellen zur Grundlage haben. Für die Decodierung bieten sich verschiedene softwarebasierte Verfahren sowie auch hardwarebasierte Implementierungen und Assoziativspeicher oder Folgeadressdecoder an.

3.1.2 Arithmetische Codierung

Im Gegensatz zu tabellenbasierten Entropiecodierungsverfahren wird im Falle der arithmetischen Codierung keine statische Abbildung des Quellensymbols auf das Repräsentationsalphabet vorgenommen, sondern es wird einer komplett zu codierenden Symbolfolge, ähnlich der Codierung von Blockcodes, eine reelle Zahl im Intervall [0,1] zugeordnet.
Mit Hilfe der Intervallschachtelung wird das Codierungsintervall [0,1] mit jedem neuen Symbol, das in die Codierung einbezogen werden soll, weiter eingeschränkt. Dabei wird jedem Subintervall die Auftrittswahrscheinlichkeit eines bestimmten Symbols zugeordnet.
Mit jedem neuen Symbol wird damit das aktuelle Intervall weiter eingeschränkt und stellt somit den Gesamtbereich des nächsten zur Verfügung stehenden Teilintervalls dar. Der Vorgang wird nun für alle zu codierenden Symbole durchgeführt, um als Ergebnis die binäre Repräsentation des letzten Intervalls zu erhalten, wobei immer die kürzeste im Intervall liegende Binärzahl herangezogen wird. Die Decodierung der Sequenz findet in umgekehrter Reihenfolge zur Codierung statt. Die Grundvoraussetzung für die korrekte Decodierung ist die Kenntnis des zugrunde liegenden Modells im Decoder. Dabei wird die eingehende Sequenz wie eine Art Schablone über die einzelnen Codierungsintervalle gelegt und das resultierende Symbol des Quellenalphabets anhand des Teilintervalls selektiert, in dessen Zahlenbereich der Eingangswert liegt. Die folgende Abbildung 13 stellt die Codierung der Symbolfolge *B,A,C,C* mit den zugrunde liegenden Auftrittswahrscheinlichkeiten *p(A)=0,125, p(B)=0,375* und *p(C)=0,5* dar.

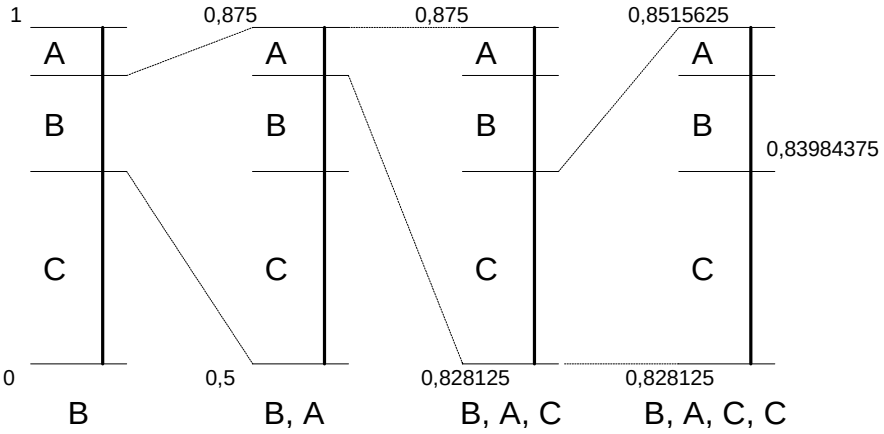


Abbildung 13 Arithmetische Codierung der Symbolfolge B, A, C, C

Die arithmetische Codierung wird z. Zt. im ISO JBIG-Standard für die Standbildcodierung und im MPEG-4-Standard für die Codierung von Objektkonturen eingesetzt. Der neueste Standard zur Bewegtbildcodierung H.264/AVC sieht den Einsatz eines adaptiven arithmetischen Coders für alle auftretenden Syntaxelemente vor.

3.2 Videocodierung

Die bisher beschriebenen Verfahren dienen der Codierung beliebiger digitaler Quellensignale und können über die statistische Bestimmung der Auftrittswahrscheinlichkeit einzelner Symbole angewandt werden. Die nachfolgend beschriebenen Verfahren basieren hingegen auf der gezielt angepassten Signalverarbeitung von Bilddaten. Hierbei können grundsätzlich die Verfahren zur Standbildcodierung und die der Bewegtbildcodierung unterscheiden werden. Die Kombination der beschriebenen Verfahren wird in Abschnitt 3.3 dargestellt und als hybrides Codierungsverfahren bezeichnet.
Allen hier erläuterten Konzepten ist jedoch die blockweise Aufteilung des Gesamtbildbereiches gemeinsam und stellt somit die Grundvoraussetzung aller Verfahren dar. Die gewählte Blockgröße kann dabei von Verfahren zu Verfahren variieren und ergibt sich aus der jeweiligen Anwendung. Die Größe eines sog. Makroblocks beträgt 16x16 Pixel und stellt damit in der Regel die größte wählbare Blockgröße dar, die dann vier 8x8 Blöcke enthält. Einige Codierungsverfahren definieren darüber hinaus Blockgrößen, die kleiner als 64 Pixel sind und fassen zum Teil mehrere Makroblöcke zu sog. Slices zusammen. Aus der makroblockweisen Aufteilung des Bildbereiches ergibt sich auch die Notwendigkeit, Bildauflösungen zu wählen, die nur Vielfache der gewählten Makroblockgröße betragen können.
Die zweite wichtige Grundlage stellt die Wahl des YUV-Farbraumes dar, der die Bildinformation für jeden Bildpunkt in Form eines Helligkeitswertes (Luminanz) und zweier, die Farbinformation bildenden Werte (Chrominanz) beschreibt. Eines der gebräuchlichsten Eingangsformate für Bildkompressionsverfahren stellt das sog. YUV 4:2:0-Format dar, das sich durch die örtliche Unterabtastung der Chrominanzinformation in vertikaler und horizontaler Richtung auszeichnet. Die Unterabtastung kann ohne deutlich sichtbare Artefakte eingesetzt werden, da visuelle Testreihen ergaben, dass das menschliche Auge Helligkeitsinformationen deutlicher wahrnimmt als Farbwertunterschiede. Neben dem hier hauptsächlich eingesetzten 4:2:0-Format werden auch noch das 4:2:2-Format mit Unterabtastung in vertikaler Richtung und das verlustfreie 4:4:4-Format für Bildcodierungsverfahren verwendet. Abbildung 14 zeigt die Makroblockstruktur eines 4:2:0-Makroblocks.

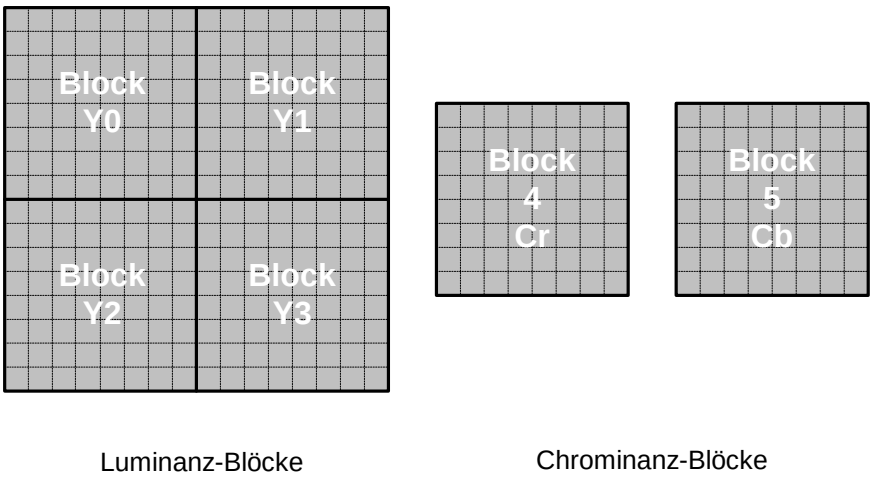


Abbildung 14 4:2:0 Makroblockstruktur für Luminanz- und Chrominanzinformation

3.2.1 Transformationscodierung

Für die im Rahmen der vorliegenden Arbeit vorgestellten hybriden Codierungsverfahren stellt die Transformationscodierung eines der zentralen Verfahren dar. Durch die Transformation der Bildinformation aus dem Ortsbereich in den Frequenzbereich mit Hilfe der diskreten Cosinus-Transformation (DCT) wird eine Analyse der transformierten Bildinformation möglich, die eine weitere komprimierende Codierung gewährleistet. Der Bildbereich einer 8x8 Bildpunktmatrix wird dabei mit Hilfe der zweidimensionalen DCT wie folgt transformiert:

$$G(f_x, f_y) = \frac{1}{4} C(f_x) C(f_y) \sum_{x=0}^{7} \sum_{y=0}^{7} g(x,y) \cos((2x+1) f_x \frac{\pi}{16}) \cos((2x+1) f_y \frac{\pi}{16}),$$

wobei

$$C(f) = \frac{1}{\sqrt{2}} \quad \text{, wenn } f = 0,$$
$$C(f) = 1 \quad \text{, wenn } f > 0, \tag{3}$$

mit

f_x, f_y : Ortsfrequenzen;
$G(f_x, f_y)$: DCT-Koeffizienten;
x, y : Ortskoordinaten;
$g(x,y)$: Bildsignal .

Eine statistische Auswertung der transformierten Blöcke ergibt dabei, dass in der Regel die Energien der Blöcke im Bereich der niederfrequenten Anteile höher sind als diejenigen der höherfrequenten Bereiche. Je höher also die Ortsfrequenzen f_x bzw. f_y gewählt werden, desto kleiner sind demnach meistens die Beträge korrespondierender Koeffizienten.
Weiterhin kann festgestellt werden, dass oftmals nach der Transformation ein Großteil der Koeffizienten den Wert null annehmen, d.h. grundsätzlich ist zu beobachten, dass niedrige Pixelwechselfrequenzen dominieren. Die folgende Abbildung 15 stellt exemplarisch die Transformation eines 8x8 Blockes aus dem Ortsbereich in den Frequenzbereich mittels DCT dar.

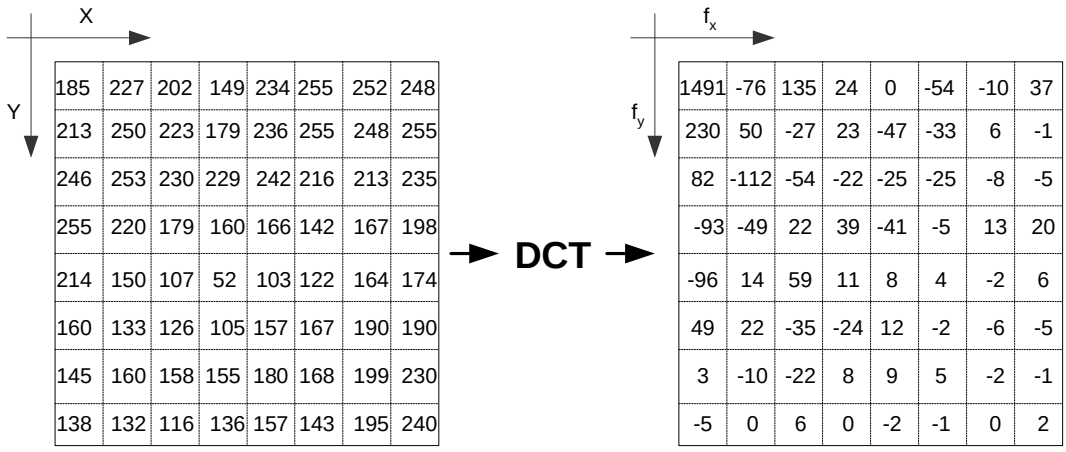


Abbildung 15 Transformation eines Blockes aus dem Ortsbereich in den Frequenzbereich

Neben der DCT gibt es noch eine Reihe weiterer Transformationen, die sich für die Verarbeitung von Bildsignalen eignen, jedoch hat sich die DCT aufgrund ihrer einfachen Umsetzbarkeit in fast allen Standards für die Bilddatenkompression etabliert.
Wie aus der vorhergehenden Abbildung 15 hervorgeht, stellt die Transformation der Bilddaten aus dem Ortsbereich in den Frequenzbereich noch keine eigentliche Komprimierung der Datenmenge dar.
Eine Kompression ergibt sich erst durch die Quantisierung (siehe Abbildung 16) der einzelnen Koeffizienten eines transformierten Bildblockes und anschließender Irrelevanz- und Redundanzcodierung. Es werden die gleichförmige und die gewichtete Quantisierung unterschieden, d.h. der Quantisierungsfaktor wird entweder für alle Koeffizienten eines Blockes gleich gewählt oder eine modifizierte Quantisierungsmatrix verwendet, die für jeden Koeffizienten einen eigenen Quantisierungsfaktor vorsieht.

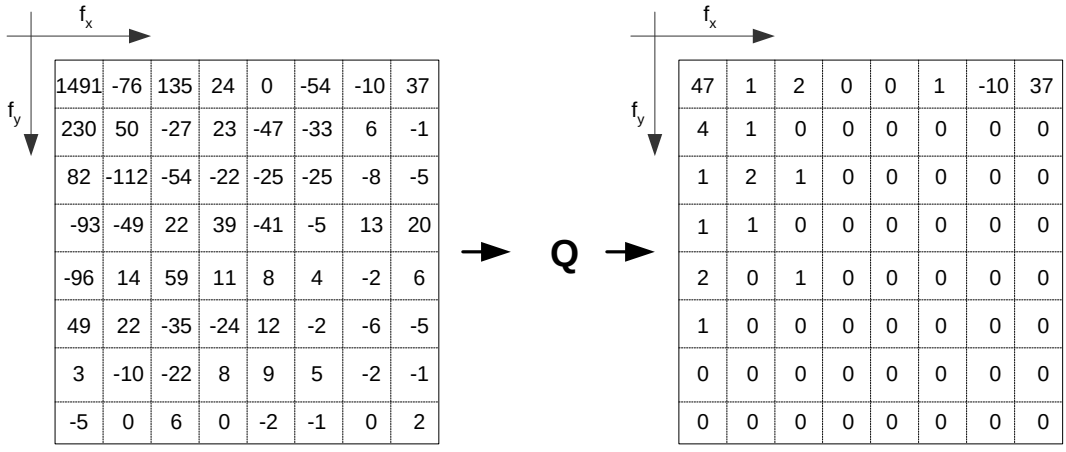


Abbildung 16 Quantisierung eines transformierten DCT-Blocks

Ziel der Quantisierung ist dabei die Steuerung der Datenrate im Encoder durch die gezielte Anpassung des Quantisierungsfaktors.

3.2.2 Redundanzcodierung

Wie in Abschnitt 3.2.1 schon erwähnt wurde, findet bei der Transformation der Bilddaten in den Frequenzbereich in der Regel eine Verteilung der Koeffizienten zu Gunsten der niederfrequenten Spektren statt, d.h. die Häufung der Koeffizienten, die ungleich null sind, werden sich hauptsächlich im Bereich um den DC-Koeffizienten anordnen. Weiterhin kann nach der Quantisierung des Blockes auf die Übertragung eines Großteils der Koeffizienten verzichtet werden, da das Resultat ihrer Quantisierung null ist. Die beschriebenen Eigenschaften werden dabei in den z. Zt. aktuellen Codierungsverfahren für die Redundanzcodierung des Blocks eingesetzt, indem die Reihenfolge der Bearbeitung der Koeffizienten entsprechend gewählt wird. Die einzelnen Koeffizienten werden dazu seriell bearbeitet, indem der Block zick-zack-förmig ausgelesen wird, beginnend mit dem Koeffizienten der niedrigsten Frequenz. Dabei wird eine Paarbildung aus zu codierendem Koeffizienten und einer Reihe von möglichen Null-Koeffizienten vorgenommen. Der Koeffizient wird dann in der Regel mit *Level*, die Anzahl der folgenden Null-Koeffizienten mit *Run* bezeichnet. Durch die Einführung des sog. *Last Flags*, das den letzten zu codierenden Koeffizienten des Blocks kennzeichnet, kann ein vorzeitiges Ende des Blocks angezeigt werden. Je nach Eigenschaft des Blockes hat sich jedoch herausgestellt, dass es durchaus sinnvoll sein kann, eine andere Abtastreihenfolge für das Auslesen des Blocks anzuwenden. Daher werden in den neueren Bildcodierungsstandards alternative Reihenfolgen spezifiziert, die, je nach Koeffizientenverteilung, bei Bedarf gewählt werden können. Abbildung 17 veranschaulicht die Bearbeitung eines Blockes nochmals.

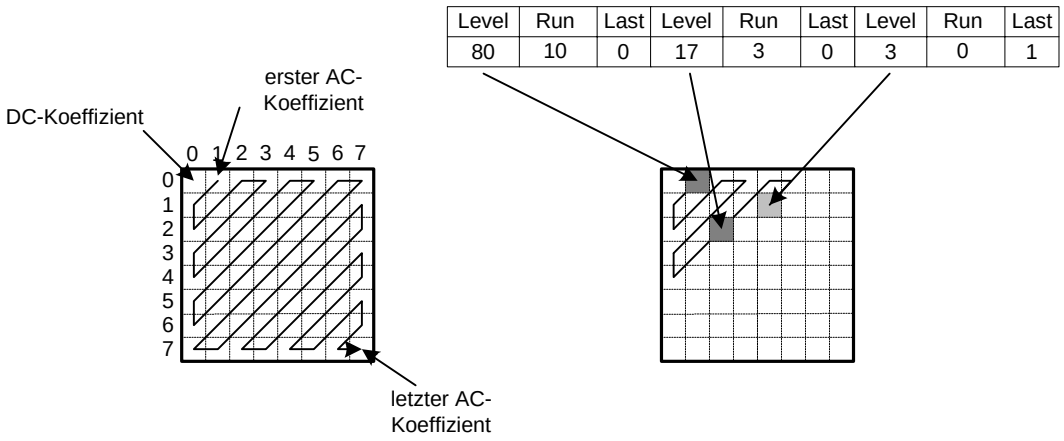


Abbildung 17 Abtastreihenfolge zur Redundanzcodierung eines Koeffizientenblocks

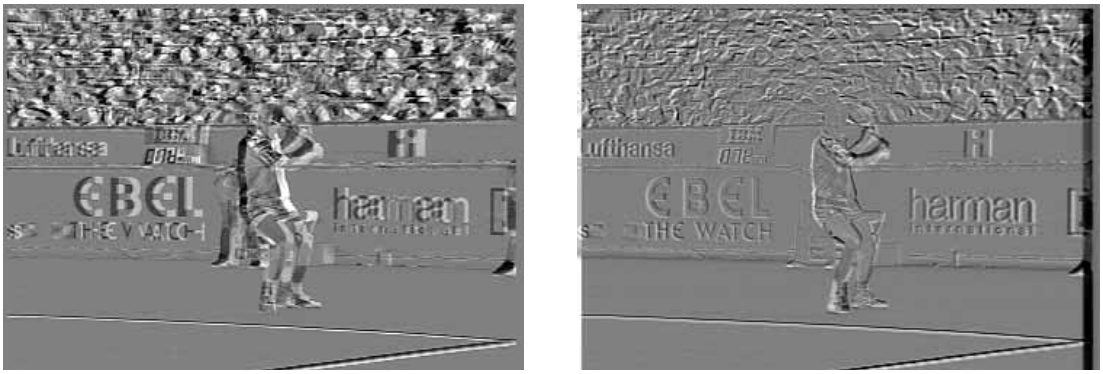
3.2.3 Intraprädiktion

Das MPEG-4-Videocodierungsverfahren verwendet, wie die Vorgängerverfahren ebenfalls, die Prädiktion von Bildinformationen zwischen zeitlich aufeinander folgenden Bildern. Die in Abschnitt 3.3 beschriebenen P- und B-Frame Prädiktionen können ebenfalls, je nach verwendetem Profile, eingesetzt werden. Neben der erwähnten inter-Frame Prädiktion ist jedoch nun auch die Möglichkeit gegeben, die Pixelinformationen eines Blockes von den bereits übertragenden Bilddaten der benachbarten Blöcke des gleichen Bildes zu prädizieren (örtliche Prädiktion). Dieses Verfahren wird in allen Profiles verwendet und ausschließlich auf intracodierte Frames angewandt. Dabei werden nicht nur die AC-Koeffizienten eines Blockes, sondern auch der DC-Koeffizient aus einem vorhergehend übertragenen Randblock abgelei-

tet. Die Auswahl des für die Prädiktion heranzuziehenden Blockes erfolgt über die Auswertung der DC-Koeffizienten benachbarter Blöcke. Da die umliegenden Blöcke mit unterschiedlicher Quantisierung rekonstruiert sein können, muss eine Normierung auf den aktuell verwendeten Quantisierungswert erfolgen, wodurch eine Division in der Berechnung der neuen Werte unvermeidbar wird. Weitere Informationen hierzu finden sich in [12], [13] und [14].

3.2.4 Bewegungskompensierte Prädiktion

Das zweite grundlegende Verfahren zur Bewegtbildcodierung stellt die bewegungskompensierte Prädiktion dar, wobei die Ähnlichkeit zeitlich aufeinander folgender Bilder einer Sequenz zur Datenkompression eingesetzt wird. Dabei kann im einfachsten Falle die Differenz zweier Bilder gebildet und codiert werden. Da sich jedoch die Bildänderung meist als Bewegung des Bildinhaltes ausdrückt, kann durch Berücksichtigung der Bewegung eine höhere Datenkompression bzw. eine Minimierung des zu codierenden Fehlersignals erreicht werden.
In Abbildung 18 sind die Ergebnisse der Differenzbildung zweier Bilder einer Sequenz mit und ohne Bewegungskompensation dargestellt. Im Falle der Differenzbildung ohne Bewegungskompensation (Abbildung 18a) sind deutlich größere Unterschiede zu erkennen, die sich in einem größeren Prädiktionsfehlersignal bemerkbar machen würden. Abbildung 18b hingegen weist wesentlich geringere Differenzen auf, wodurch nach der anschließenden Codierung eine niedrigere Datenrate als für das Vergleichsbild benötigt wird.



a *b*

Abbildung 18 a) Differenz der Bilder, b) Differenz der Bilder nach der Bewegungskompensation

Die Bewegungskompensation wird meist auf der Basis von Bildblöcken durchgeführt, die Größen von 16x16 bis zu 4x4 Bildpunkten (H.264/AVC) aufweisen können.
Um sich die Vorteile der Bewegungskompensation zu Nutze zu machen, ist es jedoch erforderlich, die Bewegung in einem Bild zu erkennen und diese einem Decoder neben dem Prädiktionsfehlersignal mitteilen zu können. Aufgrund der Aufteilung des Bildes in Einzelblöcke wird dabei in der Regel nicht die Bewegung für ein ganzes Bild bestimmt, sondern für jeden Bildblock (Referenzblock) die Verschiebung in einem Referenzbild im Vergleich zum aktuellen Bildbereich anhand von Bewegungsvektoren charakterisiert. Hierdurch ergibt sich eine genauere Annäherung an die verschiedenen Änderungen bzw. Bewegungsrichtungen in einem Bild. Für das Auffinden der Bewegungsvektoren wurden einige Verfahren entwickelt, wobei das Blockmatchingverfahren aufgrund seiner einfachen Umsetzbarkeit den größten Verbreitungsgrad gefunden hat. Hierbei wird für die Bestimmung der Bewegungsvektoren mindes-

tens der aktuelle Bildblock und der Suchbereich, der im einfachsten Fall das komplette Bild umfassen kann, benötigt. Anhand eines speziellen Kriteriums wird die Ähnlichkeit des Referenzblocks mit allen im Suchbereich befindlichen möglichen Bildblöcken ermittelt. Dazu wird der Referenzblock nach bestimmten Schemata über den Suchbereich pixelweise verschoben und anschließend mit Hilfe des ermittelten Kriteriums verglichen.
Der Bildblock im Suchbereich, dessen Kriterium am besten mit dem des Referenzblocks übereinstimmt, wird verwendet, um die Bewegungsvektoren zu ermitteln, in dem die Verschiebung in X- und Y-Richtung im Verhältnis zur Position des Referenzblocks bestimmt wird. Der Suchbereich kann im besten Fall, im Sinne der Bildcodierung, den kompletten Bildbereich umfassen, was jedoch aufgrund der hohen erforderlichen Rechenleistung in der Regel unpraktikabel ist.
In einer Sequenz von Bildern kann, neben der Einschränkung des Suchbereiches, auch die Wahl des Bildes, in dem der Suchbereich gewählt wird, variiert werden, wie z.B. im Falle der bidirektionalen Schätzung oder Multi-Frame-Schätzung.
Abbildung 19 stellt die Verschiebung der Blöcke aus Bild *B* im Verhältnis zum Vorgängerbild *A* dar. Die Größe der Blöcke entspricht hier nicht den tatsächlich verwendeten Blockgrößen, sondern dient lediglich der Veranschaulichung.
Neben der Variation des Suchbereiches, d.h. des verwendeten Bildausschnittes, kann auch die Suchgenauigkeit den Anforderungen des jeweiligen Coders angepasst werden. In den bisher standardisierten Verfahren werden Suchgenauigkeiten von ½- und ¼-Pel Genauigkeit verwendet, die durch Interpolation bzw. spezielle Filterung der Bildbereiche erzielt werden.

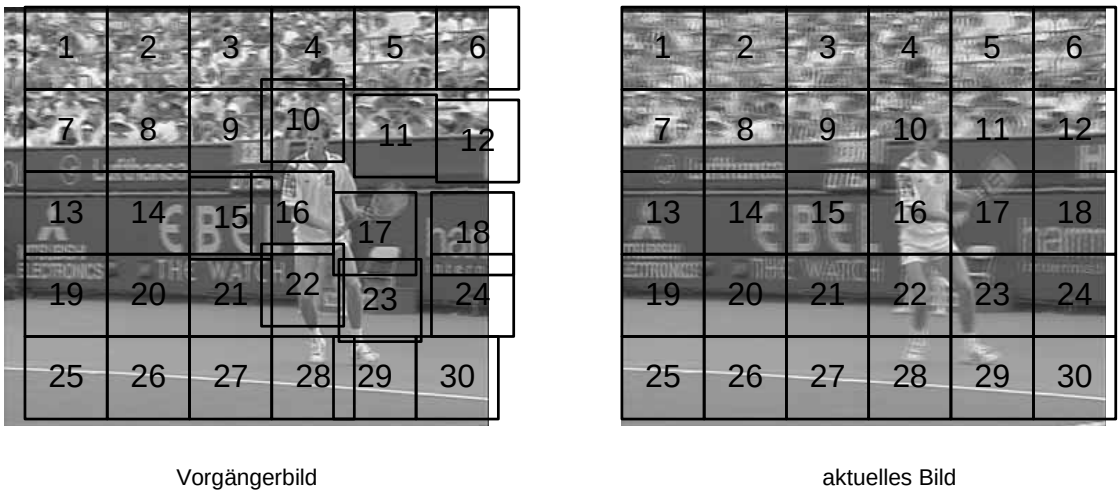


Abbildung 19 Blockmatching Bewegungsschätzung

Eine zusätzliche Erweiterung stellt der Gültigkeitsbereich der Bewegungsvektoren dar, die nicht nur Blöcke referenzieren können, die vollständig innerhalb des Referenzbildes, sondern auch Bildbereiche, die außerhalb des sichtbaren Bildbereiches liegen (*unrestricted motion vectors*). Die Grundlage für dieses Verfahren bildet die Erweiterung des Bildbereiches um einen zusätzlichen Rand, der im Encoder in die Bewegungsschätzung mit einbezogen werden kann. Der Randbereich wird dabei mit speziellen Bilddaten initialisiert und kann für die Prädiktion herangezogen werden. Die Initialisierung des Randbereiches muss daher in gleicher Art ebenfalls im Decoder erfolgen, um die gleichen Prädiktionsdaten für die Bewegungskompensation zur Verfügung zu stellen. Die Initialisierung der Randbilddaten wird als Padding bezeichnet und ist in ihrer Grundfunktion in Abbildung 20 dargestellt.

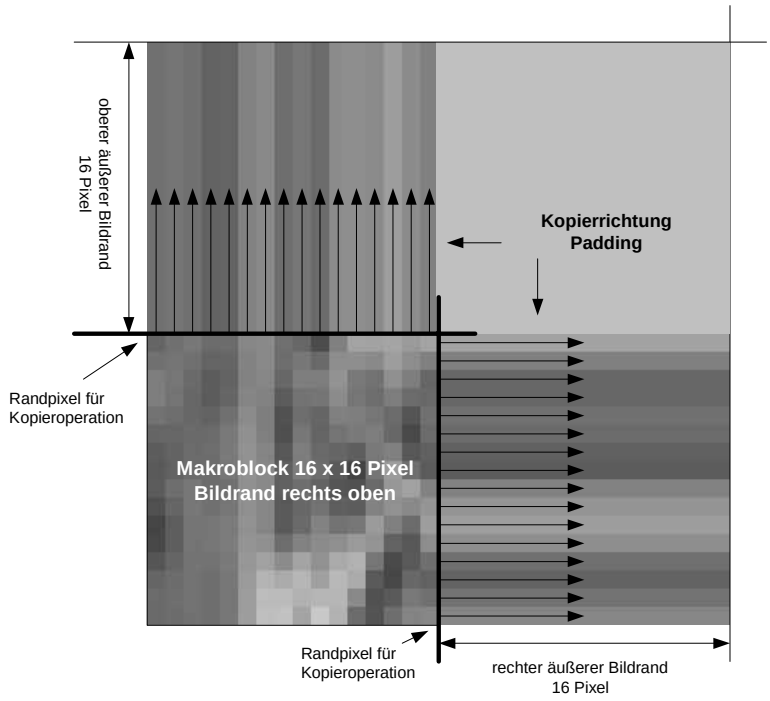


Abbildung 20 Beispiel einer Erweiterung des Bildrandes mittels Padding

Hierbei werden die Randpixel des jeweiligen Bildblockes auf Zeilen bzw. Spalten des entsprechenden Randblockes kopiert.
Neben der blockbasierten Bewegungskompensation wurde ein zweites Verfahren etabliert, das eine Bewegungskompensation auf der Basis globaler Bildbewegungen vornimmt. Hierbei wird für einen möglichst großen Bildbereich ein Bewegungsvektor übertragen, der somit die globale Bewegung des Bildinhaltes kennzeichnet. Ein typischer Anwendungsfall hierfür stellt z.B. ein Kameraschwenk dar, der im Falle der blockbasierten Bewegungskompensation in einer Reihe gleicher Bewegungsvektoren resultieren würde und mit Hilfe der globalen Bewegungskompensation lediglich einen Vektor benötigt.

3.3 Hybride Bewegtbildcodierung

Die bisher beschriebenen Verfahren zur Bilddatenkompression stellen Einzelverfahren dar, die in ihrer Kombination, d.h. der örtlichen Bearbeitung des Quellensignals und der Berücksichtigung des zeitlichen Verlaufs der zu codierenden Bildsequenz mit Hilfe der Bewegungskompensation, als hybrides Codierungsverfahren bezeichnet werden.
Die Eingangsdaten liegen dabei in Form einer YUV-Signalquelle vor, wobei, je nach verwendetem Codierungsstandard, ein Datenstrom mit standardkonformer Syntax am Ausgang des Encoders bereitgestellt wird.
Die Verarbeitung der Bilddaten wird in der Regel auf Bildebene synchronisiert, d.h. pro Eingangsbild wird ein Datenpaket am Ausgang des Encoders bereitgestellt.
Die interne Verarbeitung der Einzelbilder erfolgt dann auf Blockebene und kann wie folgt skizziert werden: Die Bilddaten gelangen vom Eingang des Encoders zeitgleich in die Bildspeicher der Bewegungsschätzung und den Eingangspuffer der DCT.
Im Falle des Anfangs einer Codierungssequenz muss als Prädiktionstyp für das komplette Bild *intra* angenommen werden, d.h. es kann keine zeitlich prädiktive Codierung angewandt

werden, da keine Vorgängerdaten für die Prädiktion zur Verfügung stehen. Hierbei wird lediglich die komplette Transformation und anschließende Quantisierung des kompletten Bildbereiches vorgenommen.
Die DCT-transformierten Bilddaten gelangen nach der Quantisierung jedoch zusätzlich in den Decoderpfad des Encoders, um die Prädiktionsdaten für den Einsatz der Bewegungsschätzung und -kompensation bereitzustellen. Dabei wird eine komplette Rückwandlung mit Hilfe der Dequantisierung und inversen Cosinustransformation vorgenommen und als Ergebnis in den Bildspeicher des Encoders kopiert. Für den Fall der bewegungskompensierten prädiktiven Codierung der Bilddaten (inter-Codierung) wird zuerst die Bewegungsschätzung vorgenommen. Die resultierenden Bewegungsvektoren werden in einem hierfür vorgesehenen Speicher gehalten und können bei Bedarf für jeden Bildblock herangezogen werden.
Abbildung 21 stellt das Blockschaltbild eines Encoders für die Codierung von Bewegtbildsequenzen mit Hilfe eines hybriden Codierungsverfahrens dar.

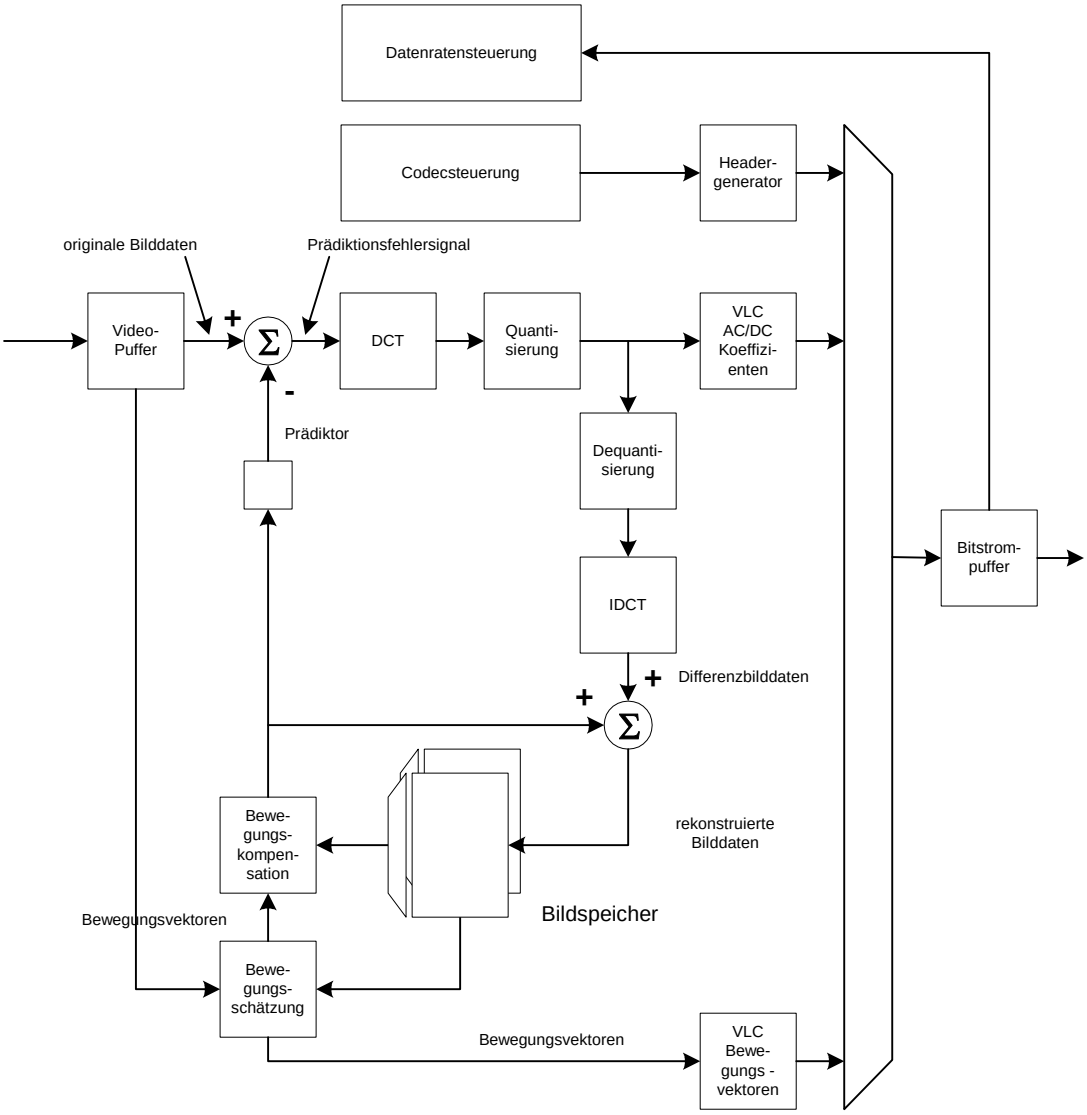


Abbildung 21 Prinzipschaltbild eines generischen Encoders für hybride Codierungsverfahren

Für die Erzeugung des Prädiktionsfehlersignals wird die Differenz aus den Originalbilddaten des aktuell zu codierenden Bildblocks und dem um die generierten Bewegungsvektoren verschobenen Referenzblock gebildet. Anschließend wird das so gewonnene Prädiktionsfehlersignal, wie im Falle der intra-Codierung, der Cosinustransformation und folgender Quantisierung unterworfen.
Die resultierenden Bildtypen werden als P-Frames (P steht für predictive) bzw. B-Frames (bidirectional predictive), falls zwei Bilder als Prädiktor zur Verfügung stehen, bezeichnet. Die Rücktransformation und Dequantisierung wird nun wiederum vorgenommen, wobei jedoch das entstehende Bild unter Berücksichtigung der verschobenen Bilddaten des Referenzbildes rekonstruiert und dem Bildspeicher für die Prädiktion des nächsten Bildes zur Verfügung gestellt wird.
Die internen Bildspeicher des Encoders enthalten somit immer die gleichen Bilddaten, die auch im Decoder für die Bewegungskompensation herangezogen werden.
Neben der in der Abbildung 21 dargestellten Codecsteuerung, die die Aufgabe besitzt, alle Abläufe des Encoders zu koordinieren, kommt der Datenratensteuerung eine wichtige Aufgabe zu. Es handelt sich hierbei um die Funktionseinheit, die eine konstante Ausgangsdatenrate des generierten Datenstroms gewährleisten soll. Hierfür wird der Ausgangspuffer des Encoders kontrolliert und mit Hilfe der Änderung des Quantisierers eine Steuerung der erzeugten Datenmenge herbeigeführt. Je nach Implementierung der Pufferkontrolle und des möglichen Zeitpunktes der Änderung des Quantisierers, können die resultierenden Datenraten starken oder weniger starken Schwankungen unterworfen sein.
Um eine schwankungsfreie Datenrate zu erhalten, ist es erforderlich, möglichst häufig Einfluss auf den Quantisierer nehmen zu können und somit eine feine Steuerung zu gewährleisten. Da der jeweils benutzte Wert des Quantisierers jedoch auch dem Decoder bekannt sein muss, werden der Steuerung durch den jeweils verwendeten Codierungsstandard Grenzen gesetzt. Meist ist die Änderung des Quantisierers nur auf Bildebene oder Makroblockebene, teilweise jedoch auch für jeden 8x8 Block, möglich.
Für Anwendungsfälle, bei denen eine konstante Datenrate nicht erforderlich ist, kann der verwendete Quantisierer auch einen festen Wert annehmen und für die Codierung der kompletten Sequenz beibehalten werden.
Ein Decoder muss prinzipiell in der Lage sein, den beschriebenen Vorgang der Codierung in umgekehrter Weise auszuführen, d.h. fast alle Komponenten des Encoders finden sich im Decoder als inverse Funktionen wieder. Das dargestellte Blockschaltbild (Abbildung 22) gibt hierbei die Struktur eines MPEG-4-Decoders wieder, dessen Funktionsweise sich nicht grundsätzlich von der beschriebenen Arbeitweise unterscheidet. Der grau unterlegte Teil des Blockschaltbildes zeigt hier die eigentliche Decoderarchitektur, die nicht MPEG-4-spezifisch ist.
Die von einer Datenquelle gelieferten Datenströme werden in der Regel dem Decoder als Access Unit, d.h. als einem Bild entsprechenden Datensatz, übergeben. Je nach verwendeter Syntax des zugrunde liegenden Codierungsstandards, findet ein Parsen des Datenstroms statt. Hierbei werden, meist auf Makroblockebene, die unterschiedlichen Datensegmente extrahiert und an die jeweiligen Komponenten weitergeleitet. Da in der Regel ein Großteil der Daten mit Hilfe von Variable-Length-Codes codiert wurde, wird als erster Schritt die Decodierung mit Hilfe hierfür vorgesehener Codetabellen durchgeführt. Dies gilt meist, neben den Koeffizienten der Texturinformation, auch für Bewegungsvektoren und diverse andere Steuerdatenworte. Im Falle der Texturdaten wird nun die Dequantisierung und anschließende Rücktransformation mit Hilfe der inversen diskreten Cosinustransformation vorgenommen.

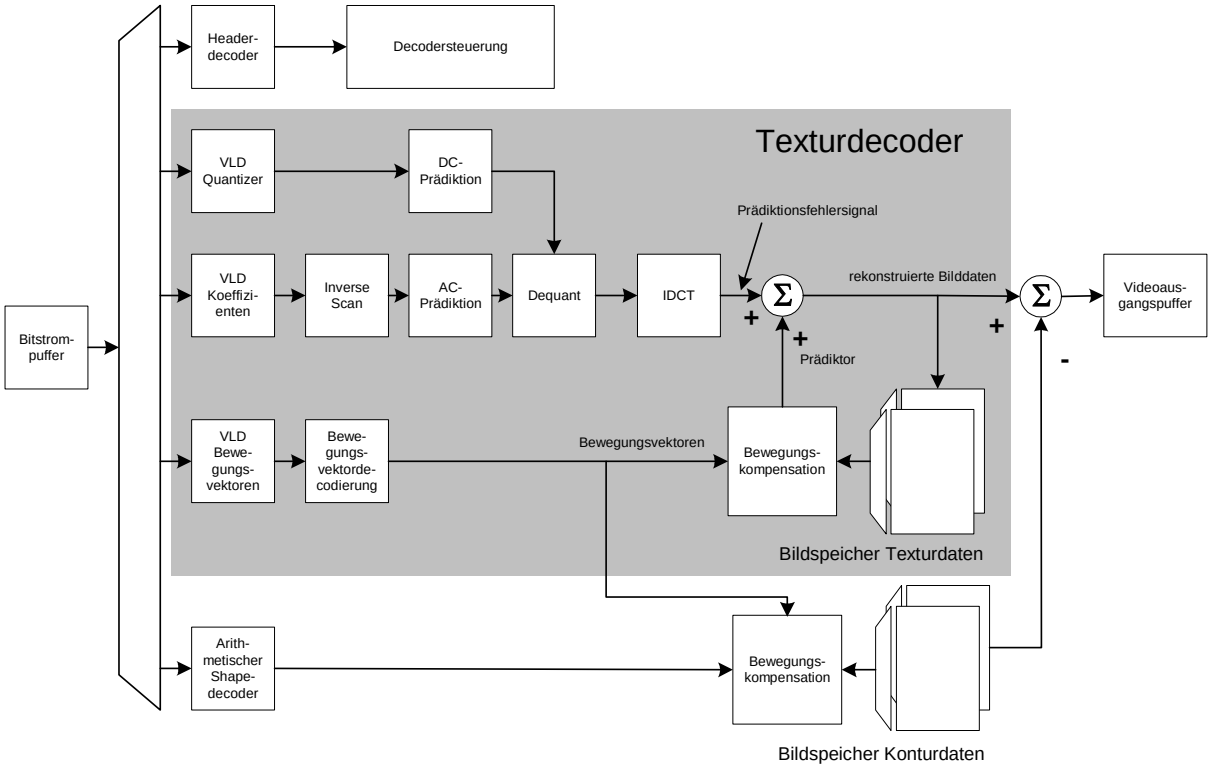


Abbildung 22 Prinzipschaltbild eines MPEG-4-Videodecoders

In Abhängigkeit vom jeweiligen Prädiktionstyp des aktuellen Bildes wird die Bewegungskompensation unter Verwendung der übertragenden Bewegungsvektoren durchgeführt oder es findet ein einfaches Kopieren der decodierten Blöcke in den aktuellen Bildspeicher im Falle eines intra-codierten Bildes statt. Nach der Decodierung aller übertragenden Blöcke wird der gesamte Bildspeicher an den Videoausgabepuffer übergeben und der aktuelle Bildspeicher als Prädiktionsspeicher für das nächste Bild bereitgestellt.
Weitere Informationen zu Grundlagen der Videocodierung können in [5] gefunden werden.

3.4 Implementierungskonzepte

Der nachfolgende Abschnitt soll einen Überblick über die Implementierungsvarianten der für die Videodecodierung benötigten Einzelverfahren geben. Dabei werden sowohl software- wie auch hardwarebasierte Umsetzungen beschrieben, die für die Implementierung prozessorbasierter Decoder aufgrund ihres hohen Rechenleistungsbedarfs relevant sind.

3.4.1 Bitstromverarbeitung

Die notwendigen Funktionalitäten zur Bitstromverarbeitung gliedern sich in die low-level Funktionalitäten zur Extrahierung von Fixed-Length-Codes (FLCs), Variable-Length-Codes (VLCs) und die darauf aufbauenden Funktionen zur syntaxgenauen Verarbeitung des Datenstroms durch einen Parser. Diese sind sehr implementierungsspezifisch und realisieren die Abarbeitung der im Anhang G.2 exemplarisch dargestellten Syntaxelemente eines Datenstroms. Im Falle von dedizierten Decoderarchitekturen werden in der Regel prozessorähnliche Strukturen eingesetzt, die auf Zustandsautomaten basieren, oder es wird auf einen ohnehin vorhandenen Prozessor zurückgegriffen. Aufgrund der zunehmenden Flexibilität der z. Zt.

etablierten Codierungsstandards ist eine dedizierte Implementierung von Parsern unter Einsatz dedizierter Logik als nicht mehr zeitgemäß anzusehen, da Implementierungs- und Verifikationsaufwand nicht mehr vertretbar sind. Die Verarbeitung von FLCs bzw. VLCs basiert hingegen auf dem Einsatz von Schieberegistern, die der Extrahierung einer definierten Anzahl von Bits aus dem aktuellen Datenstrom dienen. Die folgende Abbildung 23 gibt exemplarisch die Verarbeitung eines 12-Bit-Datenwortes wieder.

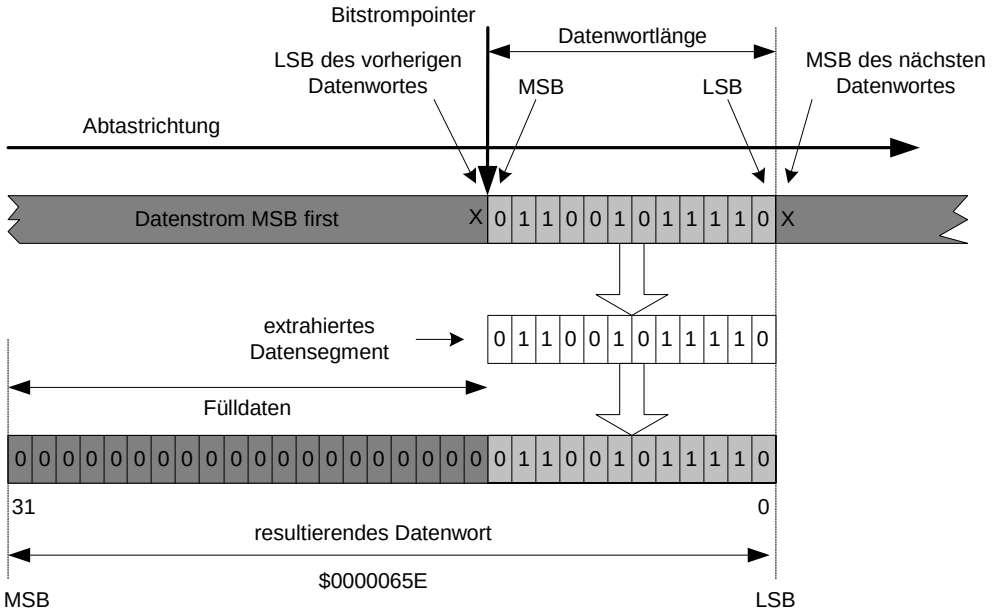


Abbildung 23 Prinzip der Verarbeitung von VL-Codes

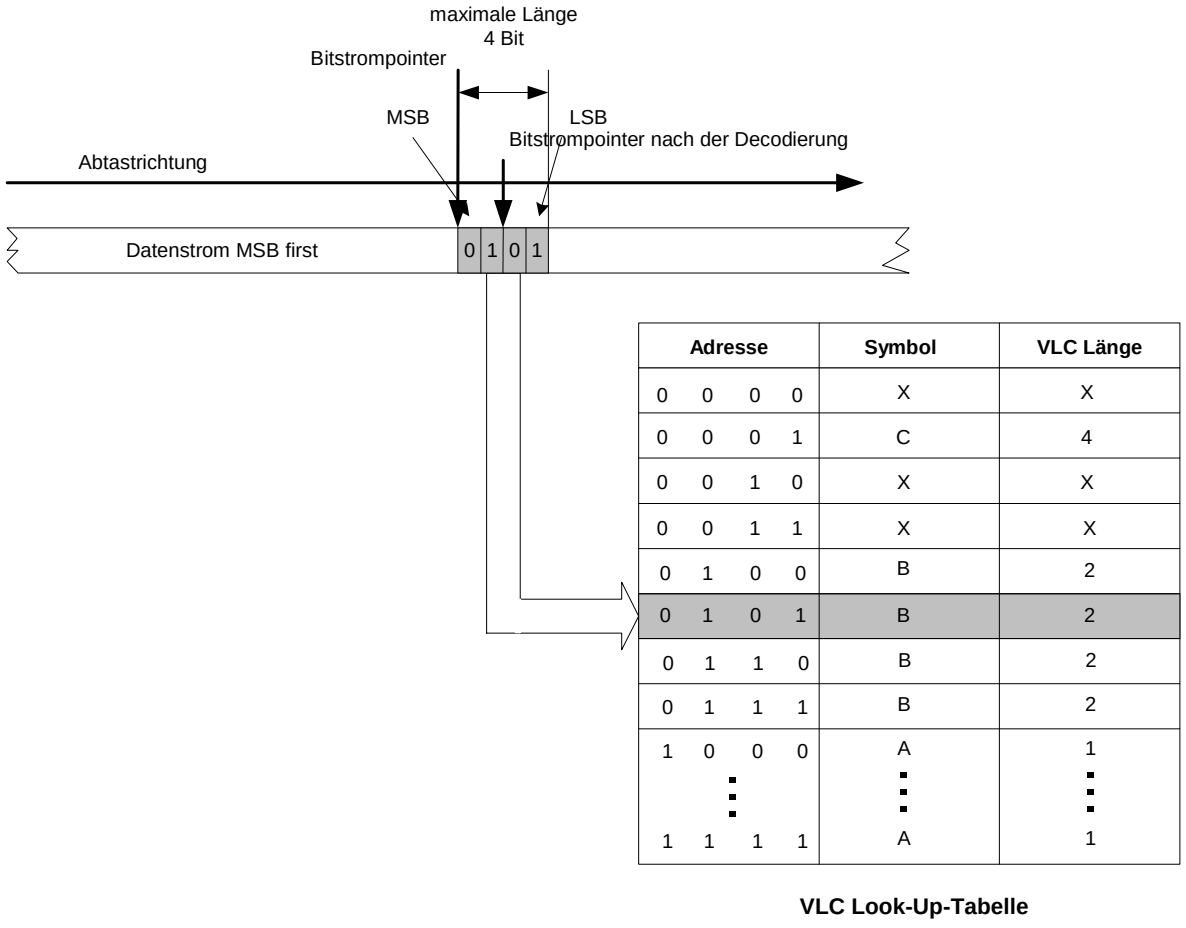
Im Falle von Syntaxelementen, die mit dem Datentyp *FLC* codiert sind, wäre somit kein weiterer Verarbeitungsschritt notwendig und der decodierte Wert kann direkt der jeweiligen Verarbeitungseinheit zugeführt werden. Der Zeiger auf die aktuelle Position im Datenstrom wird anschließend entsprechend verschoben und kennzeichnet somit den Anfang des nächsten Datenwortes. Wie dem Beispiel zu entnehmen ist, werden für die Verarbeitung von FLCs bzw. VLCs zwei Grundfunktionen benötigt, mit denen alle relevanten Funktionen umgesetzt werden können. In einem ersten Verarbeitungsschritt wird das Datenwort mit gewünschter Länge aus dem Datenstrom extrahiert, ohne den Zeiger auf den Datenstrom zu modifizieren. In einem zweiten Verarbeitungsschritt wird ausschließlich der Zeiger um die Länge des gelesenen Datenwortes verschoben. Die für die Verarbeitung notwendige Bitbreite des verwendeten Schieberegisters richtet sich nach der maximalen Länge der auftretenden Syntaxelemente des verwendeten Standards. Im Falle der beschriebenen MPEG-Standards ist mit einer maximalen Länge von 32 Bit zu rechnen. In [44] und [46] wird im Rahmen der Beschreibung einer Befehlssatzerweiterung für RISC-Prozessoren näher auf die Verarbeitung von FLCs eingegangen.

Die Verarbeitung von VLCs gestaltet sich hingegen aufwendiger, da hier die Anzahl der wirklich benötigten Datenbits nicht bekannt ist und erst nach der Decodierung neben dem eigentlichen Datenwort als Ergebnis geliefert wird. Für die Verarbeitung wird daher ein Datenwort mit der maximalen Länge des aktuellen VLCs aus dem Datenstrom gelesen. Anschließend wird mit Hilfe des extrahierten Symbols und eines VLC-Decoders das eigentliche Ergebnis-

datenwort und die Länge des VLCs ermittelt. In einem letzten Verarbeitungsschritt wird der Bitstromzeiger um die ermittelte Länge weitergeschoben. Im nachfolgenden Abschnitt werden die praxisrelevanten Implementierungsformen von VLC-Decodern beschrieben.

3.4.1.1 Tabellenbasierte VLC-Decodierung

Die einfachste Art der Decodierung von VLCs stellt die Implementierung mittels Look-Up-Tables dar, die sich sowohl für hardware- als auch für softwarebasierte Umsetzungen eignen. Hierbei wird ein Speicher verwendet, dessen maximaler Adressraum sich nach der möglichen maximalen Länge der VLCs richtet. In ihm sind alle möglichen Kombinationen aus der Zuordnung eines Codewortes zu dem resultierenden Codesymbol abgelegt. Die Größe des notwendigen Speichers beträgt somit $2^{\text{max. Codewortlänge}}$.
Die in Abbildung 24 dargestellte Tabelle weist damit eine Größe von 16 Einträgen auf, da die maximale Codewortlänge in dem gewählten Beispiel 4 Bit beträgt. Die Decodierung erfolgt mittels der Indizierung des Ergebnisdatensatzes über ein vom Datenstrom gelesenes Datenwort mit maximaler Länge, das als Adresszeiger auf den Look-Up-Speicher verwendet wird. Der Ergebnisdatensatz beinhaltet dabei das eigentliche decodierte Datenwort und die Länge des aktuell decodierten Symbols.



Adresse	Symbol	VLC Länge
0 0 0 0	X	X
0 0 0 1	C	4
0 0 1 0	X	X
0 0 1 1	X	X
0 1 0 0	B	2
0 1 0 1	B	2
0 1 1 0	B	2
0 1 1 1	B	2
1 0 0 0	A	1
⋮	⋮	⋮
1 1 1 1	A	1

VLC Look-Up-Tabelle

Abbildung 24 Prinzip einer tabellenbasierten VLC-Decodierung

Ein wesentlicher Nachteil einer einfachen tabellenbasierten Implementierung stellt die notwendige Speichergröße dar. Die im MPEG-4-Standard verwendeten maximalen Codewort-

längen betragen 16 Bit und mehr, wodurch bei einer vollen Abbildung des Ergebnisraumes ein Adressraum benötigt wird, der demnach größer als 65536 sein muss. Um die Speichergröße solcher Tabellen einzuschränken, bietet sich ein mehrstufiger Decodierungsansatz an, bei dem nicht die volle Codewortlänge in einem Vorgang decodiert wird, sondern in mehreren Einzelschritten. Als Datenspeicher können sowohl PLA- als auch ROM-Speicher eingesetzt werden, wobei die Implementierung mittels SRAM den Vorteil der Nachladbarkeit unterschiedlicher Codetabellen besitzt. In [48], [49], [50], [51] und [52] sind ASIC-Implementierungen PLA-basierter VLC-Decoder für verschiedene Codeworttabellen beschrieben.

3.4.1.2 Assoziativspeicher

Eine weitere Variante, VLC-Decoder zu implementieren, besteht in dem Einsatz inhaltsadressierter Speicher (CAM). Hierbei wird eine Speicherzelle nicht über eine Adresse ausgewählt, sondern es findet ein Vergleich eines Suchargumentes mit allen Einträgen des Speichers statt. Als Ergebnis wird ein Ergebnisvektor und ein Signal (hit oder match), das die erfolgreiche Suche im Speicher anzeigt, geliefert. Für die Realisierung eines VLC-Decoders mit Hilfe eines CAMs wird das vom Datenstrom gelesene Codewort auf den Eingang des CAMs gelegt, in dem alle auftretenden Codewörter gespeichert sind. Der resultierende Ergebnisvektor kann in Form einer Adresse auf einen weiteren Tabellenspeicher verwendet und somit als gewünschtes Ergebnis in Form des jeweiligen Codesymbols und der Codewortlänge ermittelt werden. Der Vorteil einer CAM-basierten Implementierung, wie sie in Abbildung 25 dargestellt ist, liegt in der geringeren Größe des verwendeten Speichers, da durch die direkte Zuordnung eines Codewortes zu einem Codesymbol keine Mehrfacheinträge verwendet werden müssen, wie im Falle einer tabellenbasierten Umsetzung. In [45] wird näher auf Architekturen für die Codierung und Decodierung von Codewörtern mit variabler Länge unter Einsatz inhaltsbasierter Speicher eingegangen.

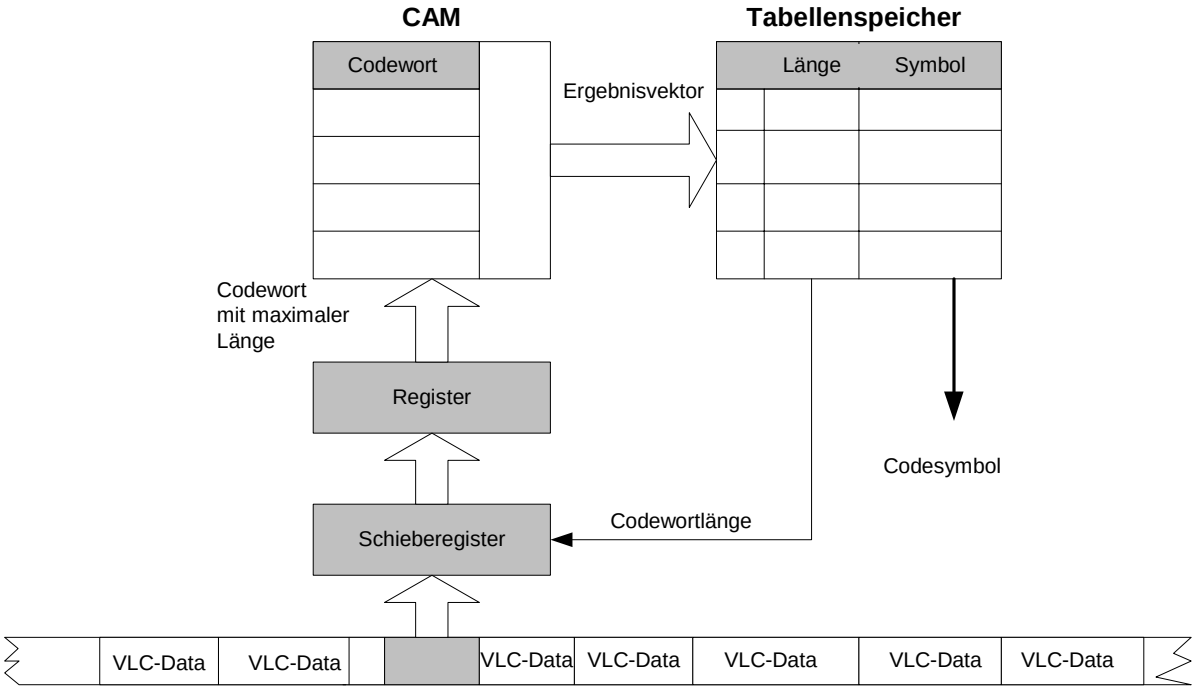


Abbildung 25 Prinzip eines CAM-basierten VLC-Decoders

3.4.2 Inverse Consinustransformation

Wie aus Abschnitt 3.2.1 hervorgeht, stellt die Transformation der Bildinformation aus dem Ortsbereich in den Frequenzbereich mittels der DCT nach Gleichung (3) eine der zentralen Funktionalitäten der hybriden Bewegtbildcodierung dar, die auch im Rahmen des MPEG-4-Standards verwendet wird. Um eine Rückwandlung der im Datenstrom übermittelten Koeffizienten aus dem Frequenzbereich *F(u,v)* in den Ortsbereich *f(x,y)* vornehmen zu können, kann die inverse Funktion durch Gleichung (4) beschrieben werden. Als Blockgröße wird nachfolgend ausschließlich von 8x8 Pixeln ausgegangen.

$$f(x,y) = \frac{2}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)\cdot C(v)\cdot F(u,v)\cdot \cos\frac{(2x+1)u\pi}{2N}\cdot \cos\frac{(2y+1)v\pi}{2N} \quad , \tag{4}$$

mit

f(x, y): Pixeldaten mit den Koordinaten x, y (x, y = 0,1,2,...., N-1) im Ortsbereich und
F(u, v): Koeffizienten mit den Koordinaten u, v (u, v = 0,1, 2 ..., N-1) im Frequenzbereich,

$C(0) = \frac{1}{\sqrt{2}}$, C(u) = C(v) = 1, wenn u,v $\neq 0$.

Eine für die Umsetzung der 8x8 IDCT wesentliche Eigenschaft stellt die Separierbarkeit in zwei eindimensionale Transformationen dar. Hierbei kann Gleichung (4) auch folgendermaßen dargestellt werden:

$$f(x,y) = \sqrt{\frac{2}{N}}\sum_{v=0}^{N-1} C(v)\cdot\left\{\sqrt{\frac{2}{N}}\sum_{v=0}^{N-1} C(v)F(u,v)\cos\frac{(2x+1)u\pi}{2N}\right\}\cdot\cos\frac{(2y+1)v\pi}{2N} \quad . \tag{5}$$

Der eingeklammerte Term kann durch *f '(x,y)* dargestellt werden:

$$f'(x,y) = \sqrt{\frac{2}{N}}\sum_{v=0}^{N-1} C(v)F(u,v)\cos\frac{(2x+1)u\pi}{2N} \quad . \tag{6}$$

Damit ergibt sich Gleichung (5) zu:

$$f(x,y) = \sqrt{\frac{2}{N}}\sum_{v=0}^{N-1} C(v)\cdot f'(x,y)\cdot\cos\frac{(2y+1)v\pi}{2N} \quad . \tag{7}$$

Für N = 8 erhält man daher für die eindimensionale IDCT aus (7) folgende Gleichung:

$$x_v = \frac{1}{2}\sum_{u=0}^{7} C(u)\cdot X_u\cdot\cos\frac{(2v+1)u\pi}{16} \tag{8}$$
$$(v = 0,1,2,3,4,5,6,7)$$

Hieraus folgt, dass eine zweidimensionale IDCT auch durch die Berechnung mehrerer eindimensionaler Transformationen (8) dargestellt werden kann. Dabei werden die in einem Eingangsspeicher liegenden Koeffizienten zuerst zeilenweise verarbeitet und anschließend in

einem Zwischenspeicher gehalten. Die weitere Verarbeitung findet anschließend spaltenweise statt.
Die folgende Abbildung 26 stellt den zweistufigen Verarbeitungsvorgang nochmals dar.

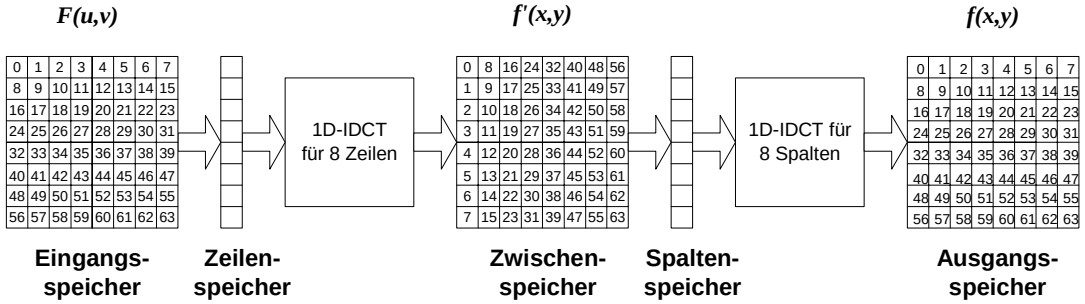


Abbildung 26 Bearbeitungsschema für eine 8x8 IDCT mit zwei eindimensionalen Transformationen

Ein entscheidender Vorteil ist dabei die zweifache Nutzung einer einzelnen Verarbeitungsstufe und dem damit einhergehenden geringeren Implementierungsaufwand im Verhältnis zu einer direkten Implementierung der zweidimensionalen IDCT.
In den nachfolgenden Beschreibungen wird daher auch nur auf die unterschiedlichen Umsetzungsvarianten der eindimensionalen IDCT für N=8 eingegangen, die sich für die Umsetzung der 8x8 IDCT nach Abbildung 26 eignen. In [56], [57], [58] und [59] wurden eine Reihe schneller Algorithmen für die 1D-IDCT vorgeschlagen. Allen Verfahren ist die Reduktion der benötigten Anzahl von Multiplikationen und Additionen gemeinsam, um die Verarbeitungsgeschwindigkeit und Komplexität des Verfahrens zu verbessern. Der bekannteste schnelle Algorithmus für die eindimensionale IDCT wurde in [54] von Chen et al. vorgestellt.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix}$$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} A & B & A & C \\ A & C & -A & -B \\ A & -C & -A & B \\ A & -B & A & -C \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix} - \frac{1}{2}\begin{bmatrix} D & E & F & G \\ E & -G & -D & -F \\ F & -D & G & E \\ G & -F & E & -D \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_3 \\ X_5 \\ X_7 \end{bmatrix} \tag{9}$$

mit

$$A = \cos\frac{\pi}{4}, B = \cos\frac{\pi}{8}, C = \sin\frac{\pi}{8}, D = \cos\frac{\pi}{16}, E = \cos\frac{3\pi}{16}, F = \sin\frac{3\pi}{16}, G = \sin\frac{\pi}{16}$$

Der Vorteil dieses Algorithmus liegt, neben der geringen Zahl arithmetischer Operationen, in der regulären Struktur, die sich somit besonders günstig für Hardwareumsetzungen eignet. Während die 1D-IDCT nach (8) noch N² Multiplikationen benötigt, kann nach Chen die An-

zahl der Multiplikationen durch Aufteilung in zwei Matrix-Vektor-Operationen (9) auf $N^2/2$ halbiert werden.
Durch weitere Optimierungen des Datenflussgraphen lässt sich die Anzahl der Multiplikationen auf 16 bzw. 13 zusätzlich reduzieren.
Ein weiterer schneller Algorithmus für die 1D-IDCT wurde von Loeffler et al. in [60] vorgeschlagen. Wie aus der Darstellung des hierzu gehörenden Datenflussgraphen in Abbildung 27 hervorgeht, konnte hier nochmals eine Reduktion der arithmetischen Operationen erzielt werden, wobei insgesamt nur 11 Multiplikationen für die Berechnung der 1D-IDCT benötigt werden.
Der dargestellte Algorithmus liefert somit die schnellste Umsetzung, ohne die von der IEEE in Form eines Standards in [55] definierten Genauigkeitsanforderungen algorithmenbedingt zu verletzen.

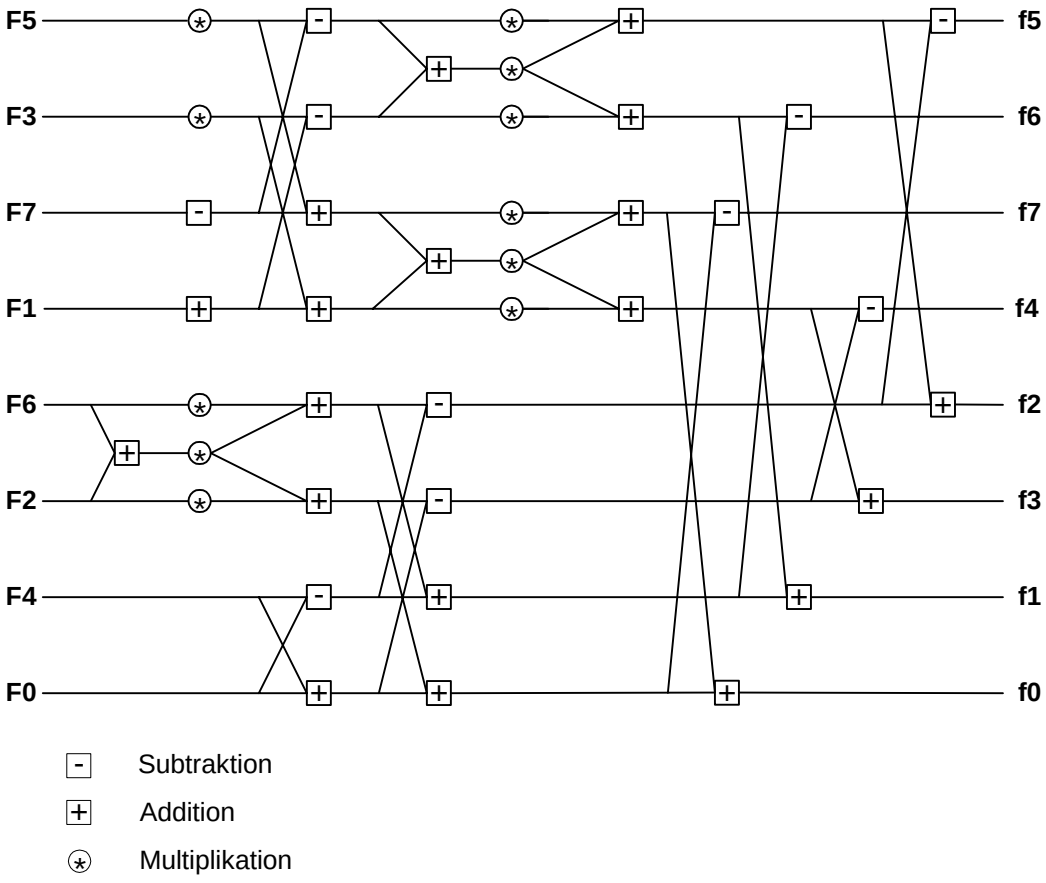


Abbildung 27 Datenflussgraph des Algorithmus zur eindimensionalen IDCT nach Loeffler

Allen Algorithmen ist dabei eine Kernoperation gemeinsam, die auf der Multiplikation einer Eingangsvariablen mit einem konstanten Koeffizienten und dem anschließenden Addieren mit einem zuvor berechneten Wert der gleichen Operation beruht. Die als MAC-Instruktion benannte Operation wurde bereits in Abschnitt 2.4.4 als Grundfunktion digitaler Signalprozessoren beschrieben.
Für die Implementierung der dargestellten Strukturen in Form einer dedizierten Hardware wird jedoch in der Regel nicht die Flexibilität gefordert, die ein Signalprozessor bietet. Die

nachfolgende Abbildung 28 gibt exemplarisch den Rechenkern einer MAC-Operation wieder mit der die in Abbildung 26 dargestellte Struktur umgesetzt werden kann. Diese Struktur verwendet dabei einen dedizierten Multiplizierer, einen Addierer inkl. Akkumulator und einen Butterflymechanismus, mit dem ein Umsortieren der Ausgangsdaten erfolgen kann.

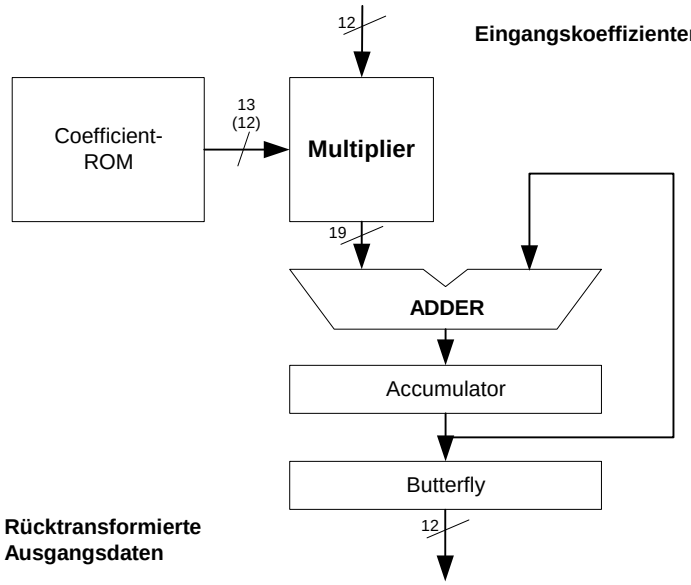


Abbildung 28 Blockschaltbild eines Rechenkerns für eindimensionale multipliziererbasierte IDCTs

Für die Speicherung der konstanten Koeffizienten wurde ein eigener Speicher in Form eines ROMs vorgesehen. Dabei kann aufgrund der erforderlichen Datenwortbreite und der Tatsache, dass die Multiplikation ausschließlich mit konstanten Koeffizienten durchgeführt wird, der Rechenkern hierfür optimiert werden. Im einfachsten Falle könnte der komplette Ergebnisraum der Multiplikation in Form eines Festwertspeichers ausgeführt werden, was jedoch aufgrund der zu erwartenden Größe des notwendigen Speichers unpraktikabel wäre.
Der in Abbildung 29 dargestellte Rechenkern verwendet stattdessen einen Distributed Arithmetic (DA)-Multiplizierer, der auf der Zerlegung der Multiplikation in einzelne Teilmultiplikationen und dem anschließenden Aufaddieren der Partialprodukte basiert. Die Partialprodukte werden in einem Festwertspeicher gehalten, dessen Einträge von den Eingangsdaten adressiert werden.
Das Aufaddieren der Partialprodukte wird mit Hilfe des Addierers vorgenommen, wobei durch ein Schieberegister für die richtige Wertigkeit der zu addierenden Einzelergebnisse gesorgt wird. Die weitere Verarbeitung und Einbettung in eine Gesamtstruktur entspricht dem des multipliziererbasierten Konzepts.
Nachteil eines DA-basierten Rechenkerns ist die höhere Zyklenanzahl, die im Wesentlichen durch die Bitbreite der Eingangsvariablen bestimmt wird. Bei einer Bitbreite von typischerweise 12 Bit für Transformationskoeffizienten werden somit mindestens 12 Zyklen benötigt, um ein Gesamtergebnis der Multiplikation der Eingangsvariablen mit dem Eingangskoeffizienten zu erhalten.
Für die Beurteilung eines Algorithmus in Bezug auf eine günstige Implementierbarkeit in Form einer speziellen Hardware bzw. einer Softwareumsetzung ist neben der Anzahl der notwendigen arithmetischen Operationen auch die Struktur des Algorithmus zu beachten. Im

Gegensatz des der Gleichung (9) zugrunde liegenden Algorithmus, benötigt die IDCT nach Abbildung 27 weniger Rechenoperationen, stellt jedoch eine unreguläre Struktur dar, die einen höheren Kontrollaufwand erfordert.
Weiterhin ist zu beachten, dass hierbei mehrere Multiplikationen pro Datenpfad notwendig sind. Somit eignet sich die in Abbildung 27 dargestellte Struktur eher für eine softwaremäßige Umsetzung, während der Algorithmus nach Chen eine weite Verbreitung in Form verschiedener Hardwareimplementierungen gefunden hat.

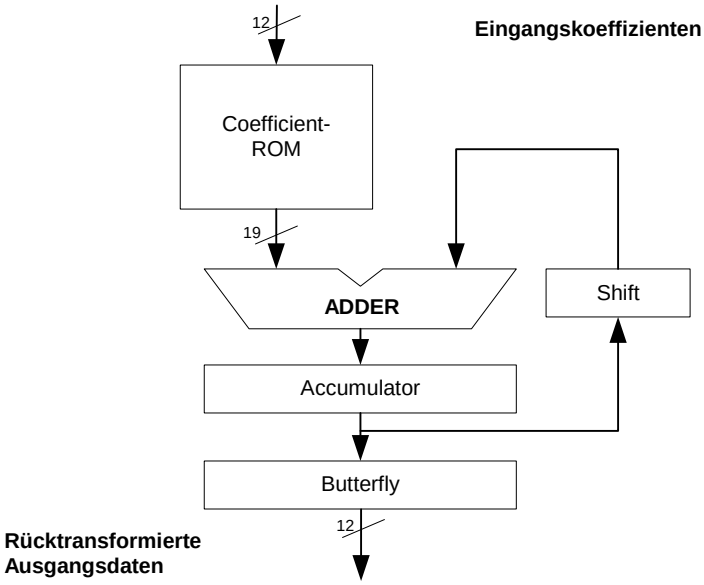


Abbildung 29 Blockschaltbild eines Rechenkerns für eindimensionale IDCTs auf Distributed Arithmetic-Basis

Eine beispielhafte Implementierung hierfür ist in [67] beschrieben, wo aufgrund der hohen Anforderungen bei der Decodierung von MPEG-2 HDTV-Datenströmen auf eine Hardwareimplementierung der IDCT zurückgegriffen wurde. In Anhang A.5 ist die Realisierung des Autors einer SIMD-basierten IDCT auf der Basis des Loeffler-Algorithmus wiedergegeben.
Weitere SIMD-Implementierungen sind in [63] und [66] beschrieben, die neben der Verwendung von MMX-Konstrukten auch Optimierungen auf Algorithmenebene beinhalten.
Hierbei wird z.B. eine Überprüfung der rücktransformierten Zeilen bzw. Spalten auf Null-Koeffizienten vorgenommen und die Berechnung auftretender Null-Produkte unterbunden. In [65] ist ein Konzept beschrieben, das nicht auf der üblichen Aufteilung der Berechnung in Zeilen und Spalten wie in Abbildung 26 basiert, sondern eine komplette 2D-IDCT auf eine Hardwarearchitektur abbildet. Informationen zum Einfluss der verwendeten Datenpfadbreiten und den resultierenden Berechnungsfehlern finden sich in [61] und [62].

3.4.3 Bewegungskompensation

Neben der Transformationscodierung stellt die bewegungskompensierte prädiktive Codierung die zweite wichtige Komponente eines hybriden Codierungsverfahrens dar.
Wie in Abschnitt 3.3 beschrieben wurde, werden dabei Bildinformationen blockweise aus einem Referenzbildspeicher mit Hilfe dazugehöriger Bewegungsvektoren adressiert und als Prädiktor mit dem im Datenstrom übertragenden Prädiktionsfehlersignal pixelweise addiert.

Im vorliegenden Falle handelt es sich um Blockgrößen mit 8x8 Bildpunkten, wobei die Verschiebungsvektoren eine Genauigkeit von einem halben Pixel aufweisen können.
Da die Bilddaten jedoch ausschließlich pixelweise vorliegen, muss daher eine Berechnung der ½-Pel Positionen vorgenommen werden. Der MPEG-4-Standard sieht hierfür die bilineare Interpolation vor.
Abbildung 30 stellt das angewandte Interpolationsschema dar, wobei die Interpolationsmöglichkeiten in horizontaler-, vertikaler- oder zweidimensionaler Richtung zu berücksichtigen sind.

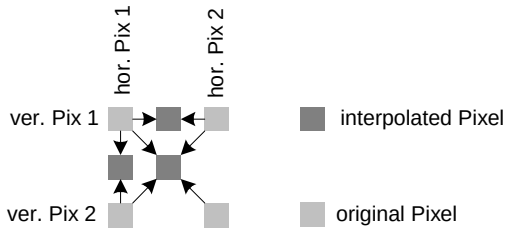


Abbildung 30 Schema der ½ Pel-Interpolation

Somit kann die Durchführung der Bewegungskompensation grundsätzlich in zwei Verarbeitungsprozesse unterteilt werden. Dabei handelt es sich um Speicherzugriffsoperationen, die den Quell- und Zielspeicher der Bilddaten adressieren und um die arithmetischen Operationen, die der Pixelinterpolation und der Addition des Prädiktors mit dem Prädiktionsfehlersignal dienen. Der Zugriff auf den Quellspeicher ist nicht deterministisch, da die Bewegungsvektoren im Quellspeicher auf beliebige Positionen zeigen können, wohingegen der Zielspeicher über die fortlaufenden Blockpositionen adressiert wird. In der Regel wird bei prozessorbasierten Systemen für den Speichertransfer auf einfache Load-/Store-Operationen oder auf parametrisierbare DMA-Controller zurückgegriffen. Die erforderlichen arithmetischen Operationen können wiederum in Pixelinterpolation und Aufsummierung des Prädiktors mit dem Fehlersignal unterschieden werden.
Da es sich um die einfache bilineare Interpolation handelt, kann die Berechnung auf Summations- und Schiebe-Befehle abgebildet werden.
Das folgende Codefragment gibt exemplarisch die Interpolation eines Bildpunktes für die ein- und zweidimensionale Interpolation wieder.

```
*des++ = (UInt8)((ver.Pix1 + ver.Pix2 + RC)>>1);
                eindimensionale Interpolation
*des++ = (UInt8)(( ver.Pix1 + ver.Pix2 + hor.Pix1 + hor.Pix2+ RC)>>2);
                zweidimensionale Interpolation
```

Das folgende Blockschaltbild in Abbildung 31 stellt nochmals den Datenfluss der einzelnen Verarbeitungsschritte der MPEG-4-spezifischen Bewegungskompensation dar und entspricht in seiner Funktionalität dem vorhergehend wiedergegebenen Programmauszug.
Neben der Pixelinterpolation wird hierbei jedoch noch die Summation von Prädiktor und Fehlersignal und die anschließende Begrenzung des Ergebniswertebereichs vorgenommen. Die Strukturen für horizontale und vertikale Interpolation sind identisch und könnten ggf. auch

einfach ausgeführt doppelt genutzt werden, wobei für ein Speichern der Zwischenergebnisse gesorgt werden müsste.
Da es sich um eine stufenweise Verarbeitung der Pixelwerte handelt, muss neben der Datenpfadimplementierung der Kontrollfluss der Komponente gesteuert werden.
Um eine Integration in ein Gesamtsystem vornehmen zu können, muss für die Generierung der Quell- und Zieladressen der Bilddaten gesorgt werden. Typischerweise wird in dedizierten Decoderarchitekturen nicht direkt auf den Bildspeicher zugegriffen, sondern über Pufferspeicher, ähnlich der Funktion eines Cache, geladen bzw. ausgelesen.

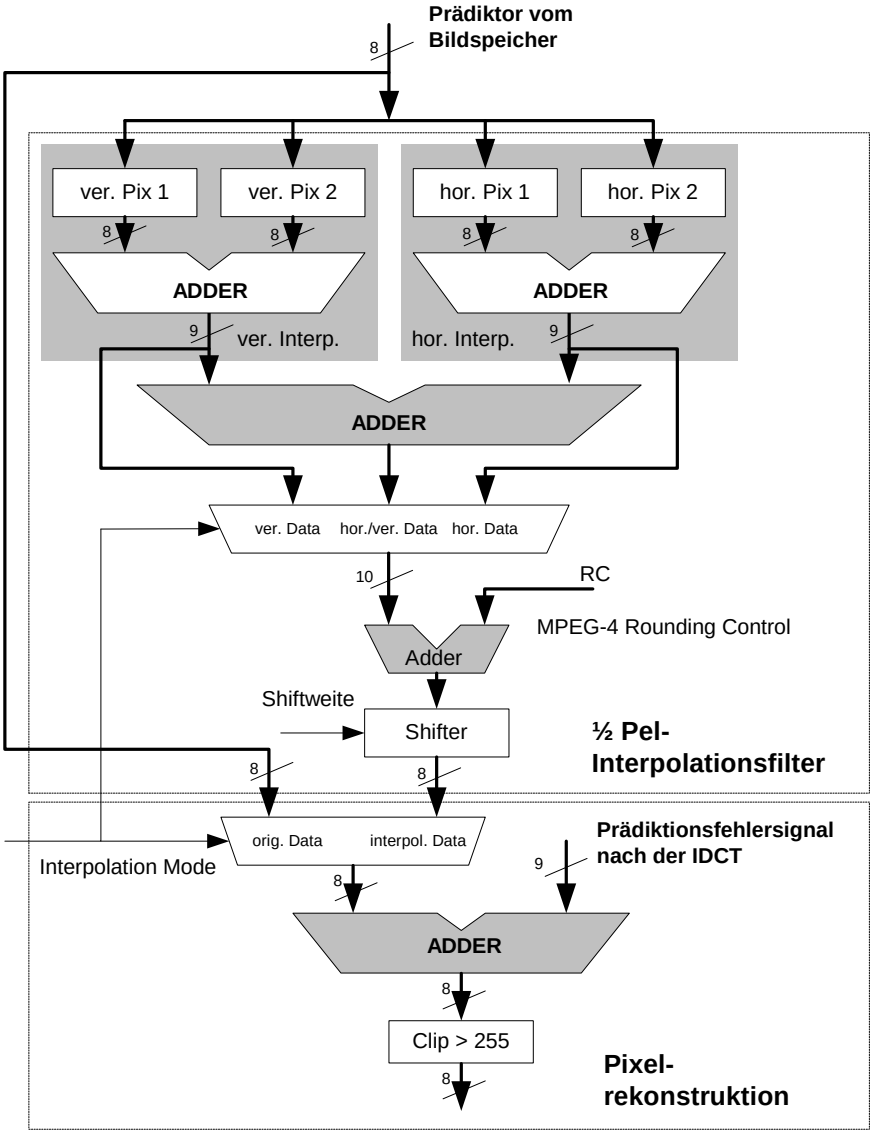


Abbildung 31 Blockschaltbild einer MPEG-4-Bewegungskompensation

Beispielhafte Implementierungen hierfür sind in [67] in Form einer dedizierten Hardware beschrieben. In [72] wird eine MMX-basierte SIMD-Implementierung eines Interpolationsfilters vorgestellt, das im Rahmen der Entwicklung eines kompletten Codecs entworfen wurde.
In Anhang A.2 ist die vollständige Implementierung der vom Verfasser umgesetzten Interpolationsroutinen aufgezeigt.

4 Prozessorbasierte Bildsignalverarbeitung

Das nachfolgende Kapitel beschreibt die Herleitung der Architektur eines prozessorbasierten MPEG-4-Decoders. Die Grundlage des Systems bildet dabei der in Abschnitt 2.4.1 beschriebene RISC-Prozessorkern, der sich durch die Möglichkeit der Befehlssatzerweiterung und ein flexibles Speicherinterface auszeichnet, das die einfache Anbindung von Coprozessoren erlaubt. Ziel des Entwurfs ist dabei, die spezifische Verarbeitungsleistung des Prozessors in Bezug auf die Decodierung von MPEG-4-codierten Videodatenströmen durch geeignete Erweiterungen zu steigern. Eine wesentliche Grundlage für die zielgerichtete Optimierung und Erweiterung der bestehenden Prozessorarchitektur stellt dabei die Analyse der Leistungsfähigkeit der Ausgangsarchitektur dar.
Da sich die herkömmlichen Methoden im vorliegenden Fall als ungeeignet erwiesen haben, wurde im Rahmen der Arbeit ein eigenes Profilingverfahren entwickelt und umgesetzt, das anschließend vorgestellt wird. Auf der Grundlage der hiermit ermöglichten Analyse konnten bildsignalverarbeitungsspezifische Architekturerweiterungen definiert werden, die im zweiten Teil des Kapitels erläutert werden.
Der dritte Teil des Kapitels beschreibt die Implementierung der einzelnen Konzepte.

4.1 Architekturspezifische Analyse

Wie den Ausführungen in Abschnitt 2.6 zu entnehmen ist, sind die dort beschriebenen Verfahren nur eingeschränkt anwendbar, um eine genaue Analyse eines prozessorbasierten Systems vornehmen zu können. Dabei sind verschiedene Forderungen an die verwendeten Analyseverfahren zu stellen. In erster Linie muss der verwendete Prozessor zyklengenau in seinem Verhalten abgebildet werden können, um das Laufzeitverhalten in Bezug auf die Echtzeitfähigkeit exakt beurteilen zu können. Auf der Basis der zyklengenauen Simulation muss eine Zuordnung des Laufzeitverhaltens einzelner Codesegmente des ausgeführten Programms möglich sein, womit eine zielgerichtete Optimierung der Software bzw. Erweiterung der Architektur realisierbar wird. Die Simulation muss dabei ohne das Einfügen von Instrumentationcode durchgeführt werden können, um auch Einflüsse des verwendeten Compilers untersuchen bzw. auch reinen Assemblercode einsetzen zu können. Weiterhin muss die Möglichkeit bestehen, das Speicherzugriffsverhalten des Systems zu beurteilen, um somit eine Unterscheidung zwischen Rechen- und Speicherzugriffsoperationen durchführen zu können.
Durch die Differenzierung von Speicher- und Rechenoperationen kann ein Rückschluss auf die reine benötigte Rechenleistung des jeweiligen Programmcodes gezogen werden. Dies gilt in besonderem Falle für Architekturen, die über keinen speziellen Speicherkontroller verfügen und sämtliche Speicherzugriffsoperationen auf Prozessorbefehle abbilden. Auch hierbei muss ein direkter Rückschluss der benötigten Speicherzugriffe auf einzelne Algorithmenteile möglich sein.
Die Ausgangsbasis stellt hier ein zyklengenaues Simulationsmodell des Prozessorsystems dar, das eine Basisversion der Hauptapplikation ausführt. Mit Hilfe der Speicherzugriffs- und Zyklenanalyse wird eine Datenbasis geschaffen, die die laufzeitkritischen Algorithmenteile der Hauptapplikation charakterisiert. Auf Grundlage dieser Datenbasis können somit Optimierungen formuliert werden, die die Software oder die Architektur des Systems betreffen. Hardware- wie auch Softwareoptimierungen beeinflussen sich dabei gegenseitig, da z.B. im Falle einer Befehlssatzerweiterung nicht nur die Hardware des Prozessors, sondern auch die Software angepasst werden muss, um den Befehl gewinnbringend verwenden zu können.

Für die Realisierung einer architekturspezifischen Analyse können sowohl Prozessorsimulatoren wie auch Simulatoren für Hardwarebeschreibungssprachen eingesetzt werden. In der Regel wird jedoch bei softwarebasierten Prozessorsimulatoren nicht die Möglichkeit geboten, zyklengenaue Speichermodelle zu verwenden.
Eine Ausnahme hiervon stellt der Prozessorsimulator *Armulator* der Firma ARM dar, der es gestattet, eigene Speichermodelle in Form von C-Funktionen in den Simulator einzubinden. Simulationsmodelle für Speicherbausteine werden hingegen fast ausschließlich mittels einer Hardwarebeschreibungssprache umgesetzt und vom jeweiligen Halbleiterhersteller angeboten. Hierbei muss der verwendete Prozessor ebenfalls als Modell in der jeweiligen Hardwarebeschreibungssprache vorliegen, womit eine Simulation mittels spezieller Hardwaresimulatoren, wie z.B. *ModelSim* der Firma Modeltech, erforderlich wird. Problematisch stellt sich die notwendige Rechenzeit dar, die sich aus der Simulation von ausgeführter Software auf einem Prozessormodell und der eigentlichen Hardwaresimulation ergibt.
Weiterhin existieren auch Mischformen von Software- und Hardwaresimulatoren bzw. die Möglichkeit der Verwendung von laufzeitoptimierten C-Modellen in Verbindung mit Hardwaremodellen, wie z.B. das sog. Foreign Language Interface des Hardwaresimulators *ModelSim*.

4.1.1 Hardware-Profiler

Die vorhergehenden Betrachtungen betreffen das eigentliche Systemsimulationsmodell und die verwendete Simulationsumgebung. Hierbei wurde jedoch nicht auf die notwendigen Profilingmechanismen, d.h. die Generierung entsprechender Daten, eingegangen. Auf der Grundlage eines Hardwaremodells des verwendeten Prozessors wurde ein spezieller Profiler umgesetzt, der nachfolgend beschrieben wird.
Die in Abbildung 32 dargestellte Profilingplattform besteht im Wesentlichen aus den Komponenten Prozessorkern, hier Argonaut RISC-Core, einem speziellen Speicherinterface, dem eigentlichen Speichermodell und einer speziellen Profilingkomponente, die die Auswertung und Verwaltung der unterschiedlich anfallenden Daten vornimmt. Das Speichermodell kann in seinem Verhalten in Bezug auf die Anzahl der notwendigen Wartezyklen parametrisiert werden, jedoch auch gegen Speichermodelle mit beliebigem Verhalten ausgetauscht werden. Neben der Ermittlung der Anzahl der Speicherzugriffe ist auch die Erfassung der Anzahl der entstehenden pageübergreifenden Speicherzugriffe möglich, um eine Aussage über die Zugriffscharakteristik zu erhalten.
Dies ist besonders wichtig bei der Verwendung von Speicherarchitekturen, auf die burstweise zugegriffen werden muss, um den maximalen Datendurchsatz zu erzielen, wie z.B. im Falle von SDRAM-Speichern.
Das System wird dabei mittels der VHDL-Simulationsumgebung *ModelSim* ausgeführt und gesteuert. Um eine hohe Simulationsperformance in Bezug auf die Ausführungszeiten zu erzielen, wurde es nicht vollständig mittels VHDL implementiert, sondern teilweise in Form von C-Modellen umgesetzt und über das simulatorspezifische FLI[18] an die restliche VHDL-Umgebung angebunden. Neben dem Vorteil, eine höhere Verarbeitungsgeschwindigkeit zu erzielen, ist der Zugriff auf das Dateisystem des Simulationsrechners stark vereinfacht, was der späteren Verarbeitung der anfallenden Profilingdaten zugute kommt.
Weiterhin wurde ein in Abbildung 32 nicht dargestellter Bootloader implementiert, der dem Laden der auszuführenden Programme in den Programmspeicher des Systems dient. Mit Hilfe

[18] Foreign Language Interface

des Bootloaders kann auch der Zugriff auf decodierte Bilddaten erfolgen, womit eine universelle Schnittstelle für den Arbeitsspeicher des untersuchten Systems zur Verfügung steht.

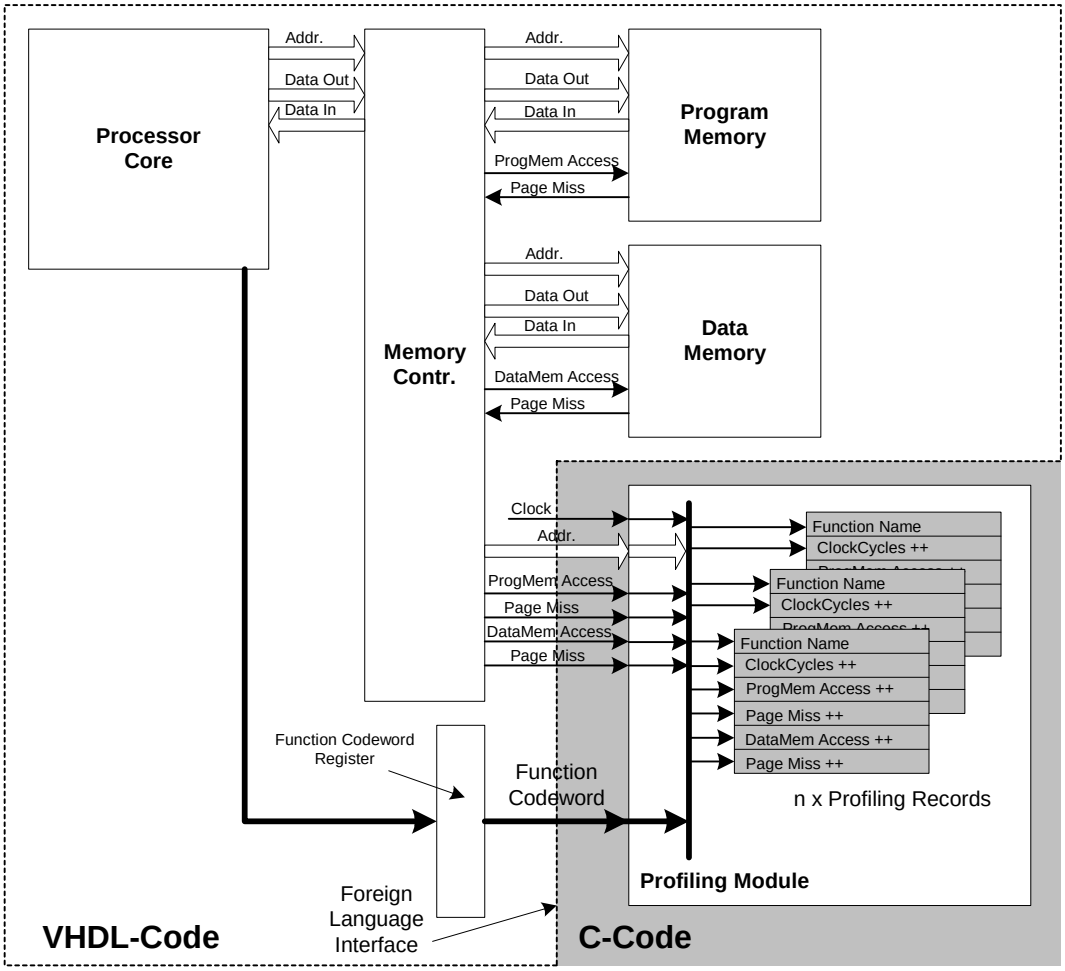


Abbildung 32 Blockschaltbild der Basisarchitektur mit Hardware-Profiler

Das Profiling-Modul ist an das modifizierte Speicherinterface des Prozessors gekoppelt, womit alle wesentlichen Vorgänge auf dem Speicherinterface protokolliert werden können, d.h. mit jedem Taktzyklus kann der Zustand des Speicherinterfaces aufgezeichnet werden. Um die anfallende Datenmenge in Grenzen zu halten, wurde die Möglichkeit der programmgesteuerten Aufzeichnung vorgesehen. Hiermit kann eine differenzierte Aufzeichnung von speziellen Profilinginformationen vorgenommen werden, indem einzelne Datensätze für unterschiedlich ausgeführte Programmsegmente verwendet werden können.
Somit ist es beispielsweise möglich, die notwendige Speicherbandbreite und die dazugehörigen Taktzyklen einer speziellen Funktion aufzuzeichnen. Der Beginn bzw. das Ende der Aufzeichnung werden durch spezielle Makros, die in den Programmcode integriert werden müssen, gesteuert. Mit Hilfe des Makros wird in ein spezielles Prozessorregister (ARC XAUX-Register) ein der Funktion zugeordnetes Codewort geschrieben, das somit einen bestimmten Aufzeichnungsdatensatz (Profiling-Record) referenziert.
In Abbildung 33 ist exemplarisch der Programmcode einer zu untersuchenden Funktion dargestellt. Hierbei wird durch das Makro *DEBUG_SIGN* mit dem Parameter *functi-*

on_entry_Decodevlc_DQAUNT die Aufzeichnung des Profiling-Records gestartet und durch *DEBUG_SIGN* (*function_exit_Decodevlc_DQAUNT)* beendet.

```
/*****************************************************************************
Proto:
UInt32      DecodeVlcDQUANT(bitstream* bits);
Author:     Benno Stabernack / HHI
Purpose:    Decodes the DQUANT code word from bitstream.
Abstract:   This function does the vlc decoding of the DQUANT code word
            from  bitstream.
            The bits are flushed. IP_Dquant_tab is used.
In:
InOut:
Out:         DQUANT value
Modified:
*****************************************************************************/
Int32 DecodeVlcDQUANT(bitstream* bits)

{
   UInt32 u_code;
   Int32 i_dquant;

   DEBUG_SIGN(function_entry_DecodeVlcDQUANT);

   u_code = GetBits(2,bits);
   i_dquant = IP_Dquant_Tab[u_code];
   PRN_VLD_VAL(" DecodeVlcDQUANT: u_dquant:%d ", ((Int16)i_dquant));

   DEBUG_SIGN(function_exit_DecodeVlcDQUANT);

   return i_dquant;
}
```


Abbildung 33 Beispiel der Steuerung des Hardware-Profilers

Die für die Ausführung der Steuermakros benötigte Prozessorrechenzeit kann vernachlässigt werden, da es sich hierbei nur um einfache Befehle handelt, die das externe Codewordregister mit dem jeweiligen Function-Codewort beschreiben. Das Ergebnis eines Profilinglaufs liegt in Form einer speziell formatierten Textdatei vor, in der sich für jede Funktion des untersuchten Programmcodes folgende Informationen für einen kompletten Programmdurchlauf befinden:

- calls - Anzahl der Funktionsaufrufe
- clock cycles - Anzahl der verbrauchten Taktzyklen
- ld/st - Anzahl aller Speicherzugriffe
- load 8/16 - Anzahl aller 8-/16-Bit-Lesezugriffe
- store 8/16 - Anzahl aller 8-/16-Bit-Schreibzugriffe
- page hit - Anzahl aller Speicherzugriffe innerhalb einer Speicher-Page
- page miss - Anzahl aller Page-Wechsel .

Wie schon eingangs erwähnt, ist die vorgestellte Architektur des Profilers speziell für den beschriebenen Prozessor entwickelt worden. Grundsätzlich kann jedoch das angewandte Verfahren für alle Prozessoren Verwendung finden, die in Form eines Hardware-Modells vorliegen. Hierbei ist es unwesentlich, ob es sich um ein reines Verhaltensmodell oder die syn-

thesefähige Beschreibung des Prozessors handelt. Voraussetzung ist dabei eine taktgenaue Abbildung des Verhaltens. In einem weiteren Projekt wurde vom Verfasser das gleiche Konzept auf das Simulationsmodell des ARM RISC-Prozessors angewandt, indem der hierfür vorhandene Softwaresimulator entsprechend erweitert wurde (siehe hierzu [68]) .

4.2 Algorithmenanalyse

Basierend auf der vorhergehend beschriebenen Profilingmethode kann der reine Rechenleistungsbedarf für die untersuchte Applikation ermittelt werden, indem die für die Verarbeitung bestimmter Algorithmenteile benötigte Anzahl von Taktzyklen gezählt wird. Hieraus kann eine genaue Analyse einzelner Programmsegmente erfolgen, um unterschiedliche Architekturerweiterungen oder auch Softwareoptimierungen definieren zu können. Für die grundlegende Untersuchung des Systems muss das Ergebnis unabhängig von der später eingesetzten Speicherarchitektur sein. Daher wurde als Zielarchitektur eine Speicherkonfiguration gewählt, die sich durch höchsten Datendurchsatz auszeichnet und keine Wartezyklen benötigt. Für die Analyse wurde eine vom Verfasser implementierte, unoptimierte Softwarevariante eines MPEG-4 Simple Profile-Decoders herangezogen und entsprechend Abbildung 33 erweitert. Auf diese Weise kann der maximale Rechenleistungsbedarf anhand der maximal benötigten Taktzyklenzahl bestimmt werden und über den zeitlichen Verlauf der benötigten Rechenleistung eine Aussage getroffen werden. Die Ergebnisse können dabei grob in die von den unterschiedlichen Bildcodierungstypen *I* und *P* hervorgerufenen Laufzeiten unterteilt werden. Das folgende Diagramm stellt diesen Sachverhalt exemplarisch dar.

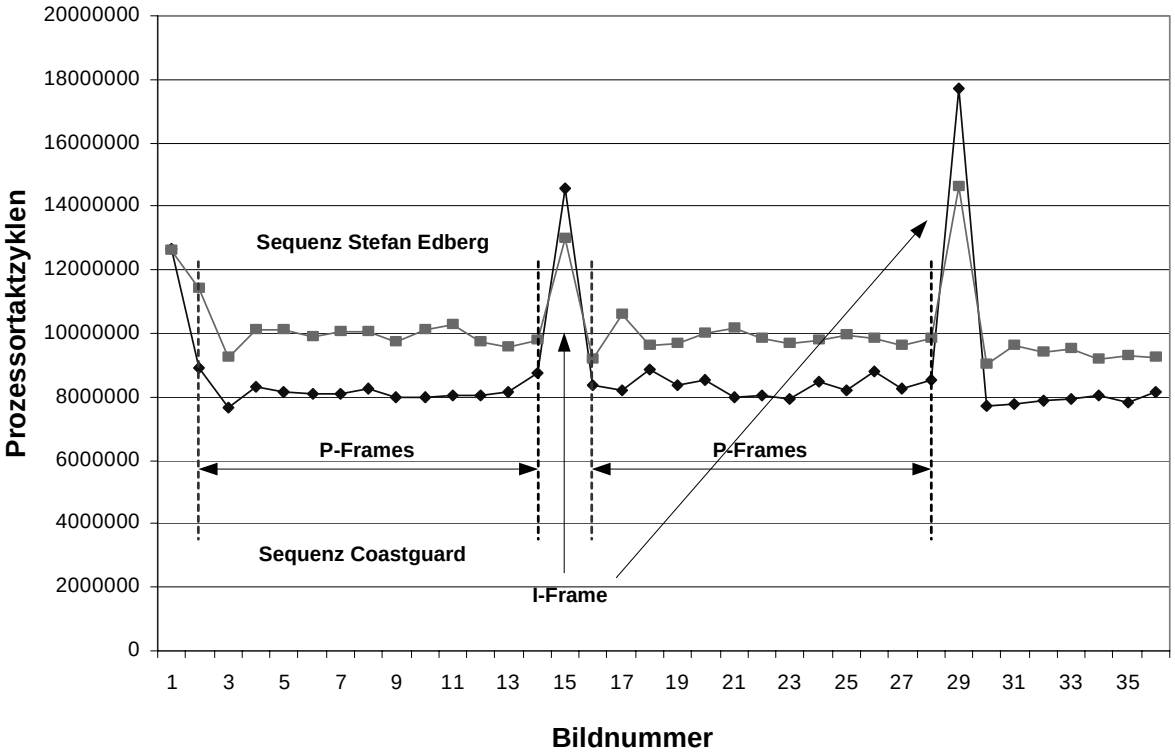


Abbildung 34 Zeitlicher Verlauf des Taktzyklenbedarfs

Dabei wurden zwei unterschiedliche MPEG-Testbildsequenzen (*Coastguard, Stefan Edberg,* siehe auch Anhang E) mit einer Datenrate von jeweils 384 kBit/s entsprechend MPEG-4 Simple Profile@Level 3 codiert.

Hierbei ist deutlich erkennbar, dass I-Frames eine höhere Anzahl an Taktzyklen benötigen, als dies für P-Frames erforderlich ist.
Die verwendete Bildsequenz nimmt ebenfalls Einfluss auf das Ergebnis der Laufzeitanalyse, da sich, je nach Bildinhalt, auch die Zusammensetzung der unterschiedlichen Blocktypen eines Bildes stark von anderen Sequenzen unterscheiden kann.
Im Falle von Sequenzen mit niedrigem Bewegungsanteil ist die Anzahl der intercodierten Blocktypen höher als bei Sequenzen mit hoher Bildänderung, da in diesem Falle oft auf intracodierte Blöcke zurückgegriffen werden muss, um eine Bildrekonstruktion zu gewährleisten. Hierbei ergeben sich nicht nur im Absolutwert unterschiedliche Laufzeiten, sondern auch die Zusammensetzung der Einzellaufzeiten der unterschiedlichen Algorithmenteile des Decoders unterscheidet sich signifikant.
Um eine Zuordnung der einzelnen Funktionsgruppen und Algorithmenteile des Decoders zu den generierten Profilingergebnissen herstellen zu können, wurden verschiedene Algorithmengruppen gebildet, die der folgenden Darstellung (Abbildung 35) entnommen werden können. Hier ist dem Flussdiagramm des Decoders mit den einzelnen Verarbeitungsschritten die entsprechende Funktionsgruppenzuordnung gegenübergestellt.

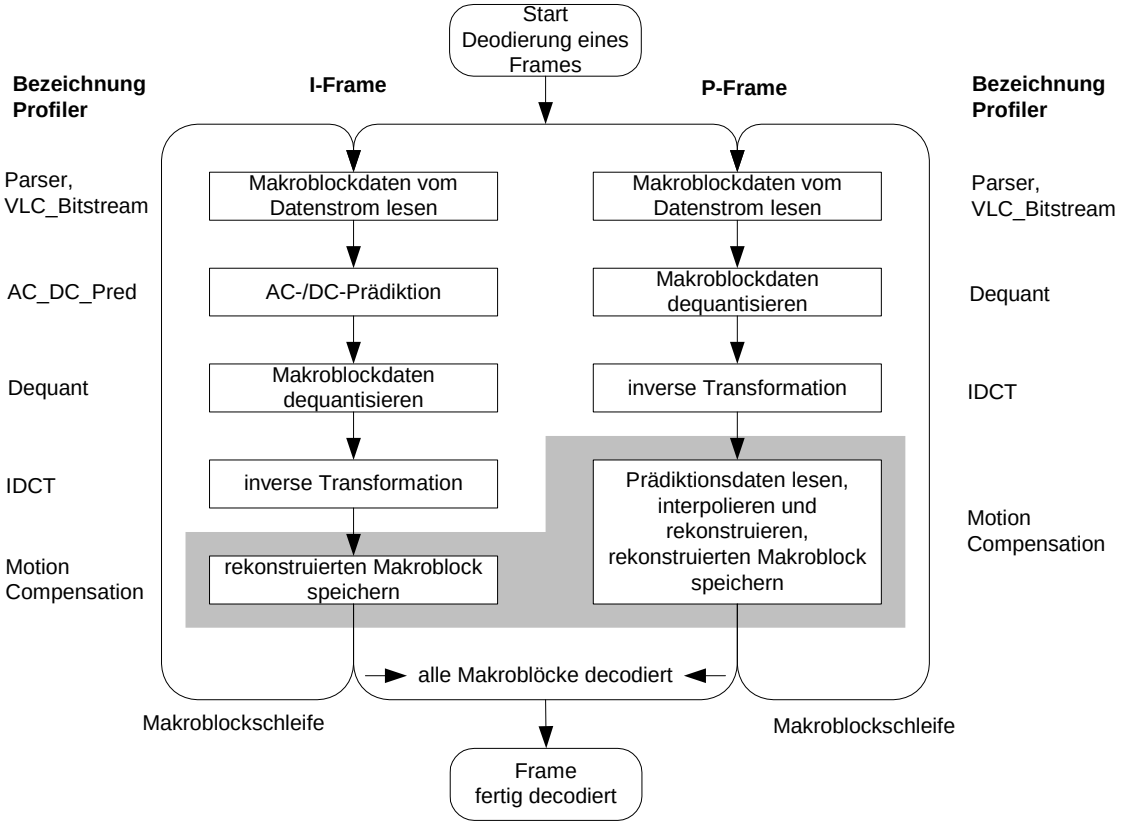


Abbildung 35 Vereinfachtes Ablaufdiagramm für die Decodierung eines MPEG-4-Video-Bildes

Eine Auswertung, wie sie im folgenden Diagramm dargestellt ist (Abbildung 36), zeigt im Detail die unterschiedliche Verteilung der benötigten Prozessortaktzyklen für die verschiedenen Bildtypen.

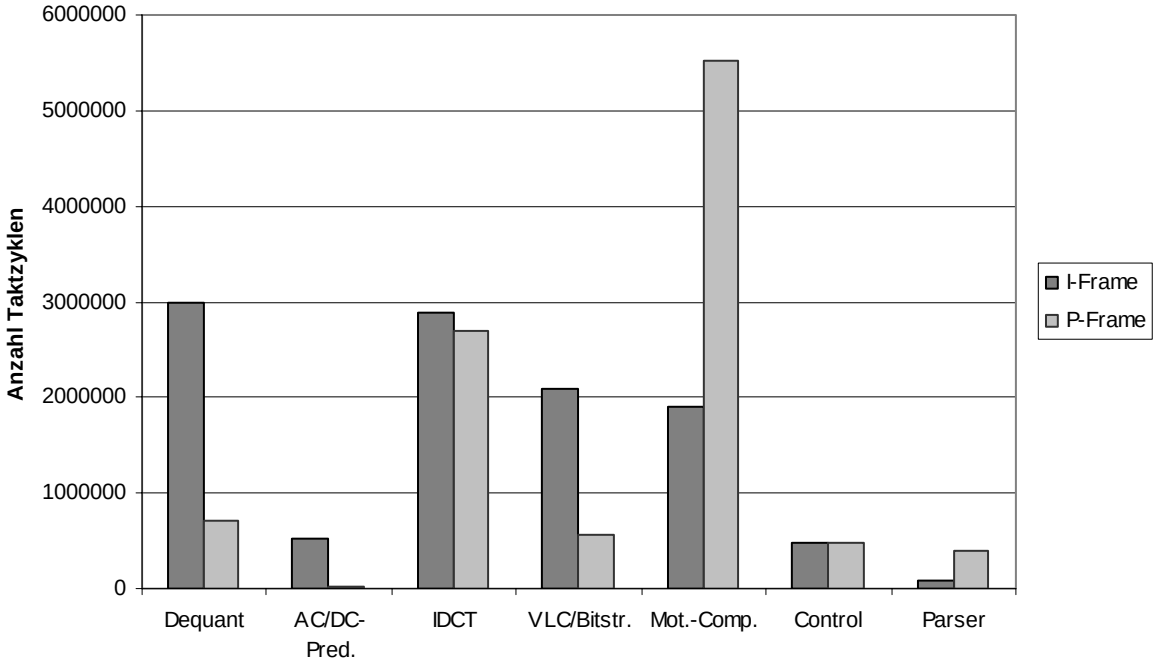


Abbildung 36 Taktzyklenbedarf in Abhängigkeit vom Bildprädiktionstyp

Hierbei zeigt sich, dass, je nach verwendetem Bildprädiktionstyp, unterschiedliche Rechenlastverteilungen auftreten, da im Falle eines I-Bildes der Anteil DCT-codierter Blöcke die maximale Anzahl annehmen kann und keine Bewegungskompensation stattfindet. Im Falle eines P-Bildes ist das Verhältnis genau gegenteilig. Auffällig ist weiterhin, dass sich der Taktzyklenbedarf für die Funktionen der Bitstromverarbeitung (*VLC/Bitstream*) nahezu verdreifacht, was auf die, im Vergleich zu P-Bildern, höhere Datenrate der I-Bilder zurückzuführen ist. Die in der Abbildung 36 zu erkennende Rechenlast für die Funktionsgruppe *Motion Compensation* im Falle intra-codierter Bildtypen (I-Frame) geht auf reine Speicheroperationen zurück, die die rücktransformierten Bilddaten in den Bildspeicher kopieren. Die hierzu notwendigen Funktionen wurden aus implementierungstechnischen Aspekten der Bewegungskompensation zugeordnet.
Somit können vier Hauptgruppen identifiziert werden, die für einen Großteil der Gesamtrechenlast verantwortlich sind. Dabei sind an erster Stelle die Bewegungskompensation und die inverse Transformation zu nennen, die in Abhängigkeit vom jeweils verwendeten Bildtyp die Hauptrechenlast hervorrufen. Die Funktionsgruppe *Dequant* ist einer einzelnen Funktion zuzuordnen und dient der Dequantisierung der rücktransformierten Koeffizienten auf der Basis einer Multiplikation pro codiertem Bildpunkt.
Eine weitere Algorithmengruppe stellen die Funktionen der Bitstromverarbeitung dar.
Da es sich hierbei um eine stark von der jeweils verwendeten Datenrate abhängige Rechenlast handelt, wirkt sich dieser Umstand als besonders nachteilig bei intra-codierten Bildtypen aus. Hier wird ohnehin eine hohe Rechenlast hervorgerufen und durch die höhere Datenrate im Verhältnis zu P-Bildern das Echtzeitverhalten nochmals nachteilig beeinflusst.
Das in Abbildung 36 dargestellte Diagramm bildet somit die Basis der weiteren Algorithmenanalyse, die sich mit Bezug auf die hervorgerufene Rechenlastverteilung auf die Algorithmengruppen *Motion Compensation*, *IDCT* und *VLC/Bitstream* konzentrieren wird.

Die genauere Analyse der Dequantisierung wird nachfolgend nicht vorgenommen, da aufgrund der Grundfunktion in Form einer Multiplikation keine Optimierungsmöglichkeit im Rahmen des verwendeten Prozessors besteht.

4.2.1 Bewegungskompensation

Wie aus Abschnitt 3.4.3 hervorgeht, setzt sich die Bewegungskompensation aus mehreren Einzelfunktionen zusammen, die auch im verwendeten Softwaredecoder in mehrere Einzelfunktionen aufgeteilt ist. Für eine hinreichende Untersuchung und Eingrenzung der echtzeitkritischen Algorithmenteile ist es daher erforderlich, eine getrennte Analyse bis auf die Funktionsebene durchzuführen. In gleicher Weise, in der auch schon der Gesamtdecoder in einzelne Funktionsgruppen unterteilt werden konnte, wird auch hierbei eine Aufteilung der Bewegungskompensation in die jeweiligen Teilalgorithmen vorgenommen. Grundsätzlich können dabei die Teilalgorithmen zur Pixelinterpolation (Interpolationsfilter) und der eigentlichen Bildrekonstruktion mittels Summation des Prädiktionsfehlersignals mit dem Prädiktor unterschieden werden. Das nachfolgende Flussdiagramm (Abbildung 37) stellt die zwei Algorithmengruppen in ihrem Ablauf dar.

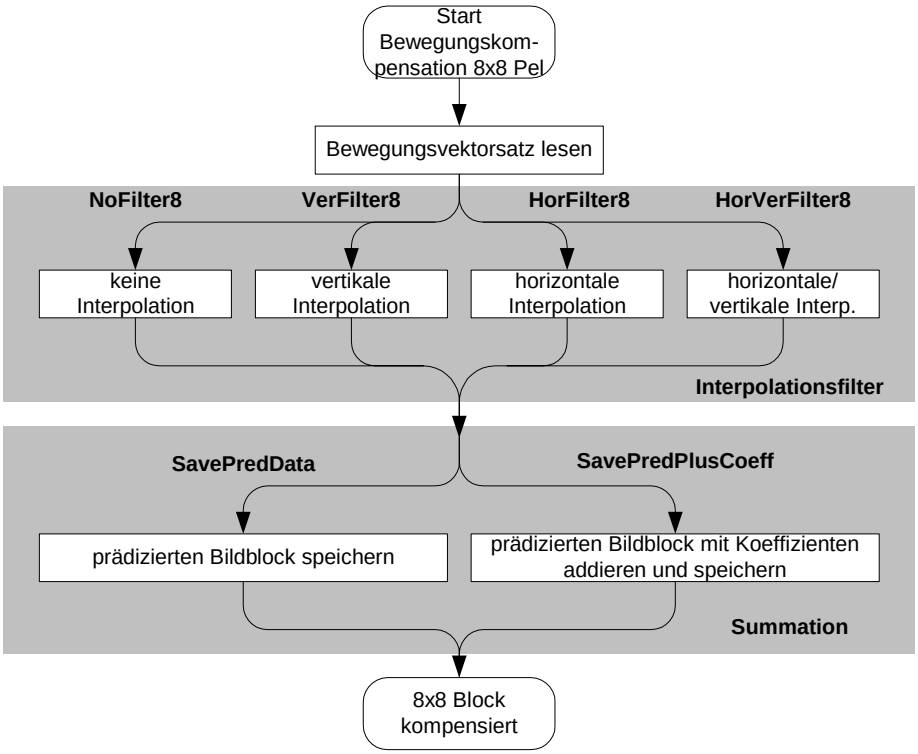


Abbildung 37 Ablaufdiagramm der Bewegungskompensation eines 8x8 Pixelblockes

Insgesamt sind an den dargestellten Algorithmenteilen der Bewegungskompensation 10 Funktionen beteiligt, die in ihrem Laufzeitverhalten wiederum genauer untersucht werden müssen, um die hierbei komplexen Programmteile identifizieren zu können. Hierzu bietet sich ebenfalls eine funktionsgenaue Analyse der verwendeten Funktionen an.
Abbildung 38 stellt das Ergebnis einer Analyse des Laufzeitverhaltens der einzelnen Funktionen bezogen auf die Decodierung eines P-Bildes dar. Die Betrachtung der Funktionsanalyse kennzeichnet die für die Verarbeitung von ½-Pel verschobenen Prädiktoren erforderlichen

Interpolationsfilter (*VerFilter8*, *HorFilter8* und *HorverFilter8*) und die Funktion zur Rekonstruktion der Bilddaten durch Summation von Prädiktor mit einem Prädiktionsfehlersignal (*SavePredPlusCoeff*) als besonders rechenintensiv.

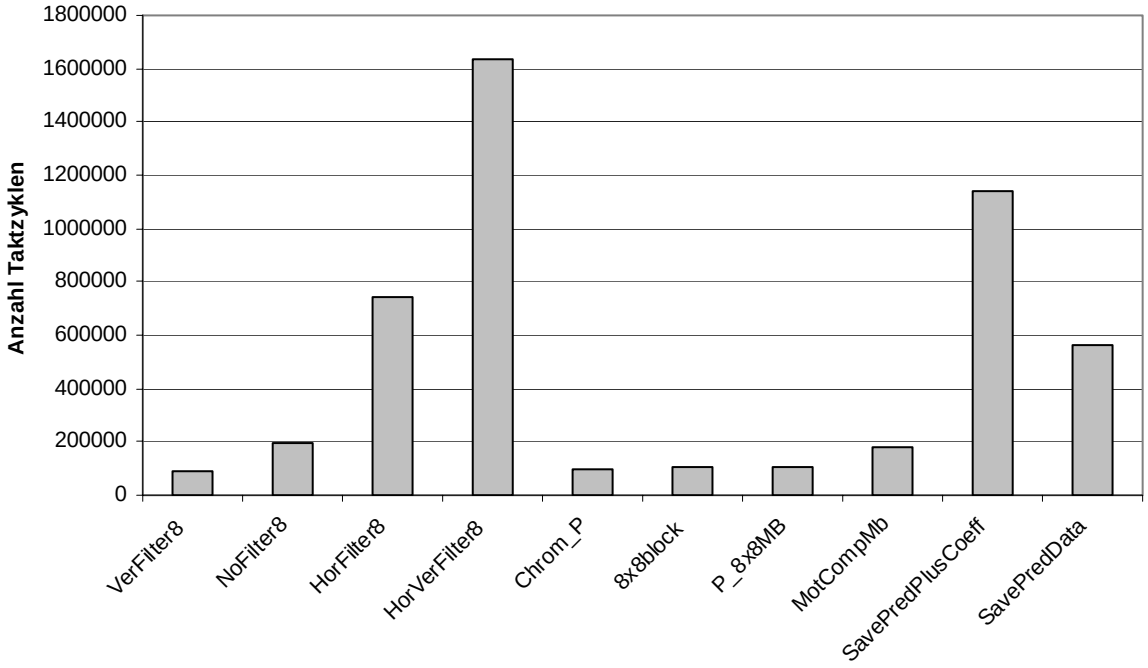


Abbildung 38 Beispielhafte Einzelanalyse der Funktionen der Bewegungskompensation bezogen auf die Decodierung von P-Bildern

Da es sich bei den Funktionen grundsätzlich um sehr speicherzugriffsintensive Funktionen handelt, ist durch die reine Betrachtung der benötigten Zyklenzahl keine hinreichende Beurteilung möglich. Sämtliche Speicheroperationen werden im vorliegenden Fall in Form von Prozessorinstruktionen (Load/Store) ausgeführt und nehmen somit direkten Einfluss auf die erforderliche Rechenzeit, auch wenn es sich nicht um reine Rechenoperationen handelt. Hierfür wurde auf der Basis der gleichen Testdaten eine Speicherzugriffsanalyse durchgeführt, die in Tabelle 3 wiedergegeben ist. Dabei wurde von der Möglichkeit Gebrauch gemacht, die verwendeten Datentypen und Schreib- bzw. Lesezugriffe unterscheiden zu können. Die Bezeichnung *load* kennzeichnet dabei die Anzahl aller Lesezugriffe. Hierbei bedeutet *load8/16* die Anzahl der Lesezugriffe, die einen 8- bzw. 16-Bit-Datentypen als Operanden verwenden. Die Angabe der Schreibzugriffe *store* erfolgt in analoger Weise.
Es ist zu erkennen, dass Funktionen mit hoher Taktzyklenzahl, wie z.B. *HorVerFilter8*, eine hohe Anzahl an 8-/16-Bit-Speicherzugriffen ausführen. Dies ist auf die Tatsache zurückzuführen, dass die Bilddaten im Speicher als 8-Bit-Datentypen angeordnet sind, um den Speicherbedarf niedrig halten zu können. Speicherzugriffsoperationen auf den Bildspeicher führen somit zu einer hohen Zyklenzahl, da für jedes zu lesende Pixel der Speicher byteweise adressiert wird, jedoch der eigentliche Speicherzugriff longwordweise erfolgt. Anstatt vier 8-Bit-Werte hiervon zu verarbeiten, wird jedoch nur ein Pixelwert verwendet. Auch bei der späteren Weiterverarbeitung der Bilddaten wird die byteweise Verarbeitung beibehalten und somit die maximale Datenpfadbreite des Prozessors nur zu 25% ausgenutzt.
Für die Beschleunigung der Funktionen der Bewegungskompensation steht somit die Optimierung der Speicherzugriffsoperationen und die daran angepasste Weiterverarbeitung mit Hilfe angepasster arithmetischer Operationen im Vordergrund.

Funktion	calls	clock cycles	ld/st	load	load 8/16	store	store 8/16
MotComb_Chrom_P_MB	792	99918	30612	19698	3786	10914	618
MotComb_P_8x8block	1584	107760	31680	20592	0	11088	0
MotComb_P_8x8MB	396	102960	32868	19800	3168	13068	0
MotCompMb	396	177935	55044	30492	3960	24552	0
NoFilter8	487	193908	32767	17091	0	15676	0
VerFilter8	115	89355	22885	15180	14720	7705	7360
HorFilter8	706	739182	100958	53656	50832	47302	45184
HorVerFilter8	1068	1637244	236028	161268	153792	74760	68352
SavePredPlusCoeff	852	1139124	280308	222372	163584	57936	54528
SavePredData	1524	563880	109728	54864	0	54864	0

Tabelle 3 Anzahl der Speicherzugriffe einzelner Funktionen der Bewegungskompensation

Die nachfolgenden Abschnitte geben die detaillierte Laufzeitanalyse und Optimierung auf der Basis entsprechender Befehlssatzerweiterungen der zur Bewegungskompensation gehörigen Funktionsgruppen *Interpolationfilter* und *Summation* wieder.

4.2.2 Interpolationsfilter

In Anlehnung an die Beschreibung der Interpolationsfilter in Kapitel 3.4.3 wurden für den hier verwendeten MPEG-4-Decoder unterschiedliche Funktionen für die Interpolation entworfen und implementiert. Da die Arbeitsweise der Bewegungskompensation auf eine blockweise Verarbeitung ausgerichtet ist, wird immer ein kompletter Datensatz von insgesamt 64 Pixelwerten verarbeitet. Der nachfolgend dargestellte Programmcode (Abbildung 39) gibt die Funktion der vertikalen Interpolation eines 8x8 Blockes wieder.

```
void VerFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
{
    UInt32 i;
    UInt8 CONST *p, *s;
    UInt32 a, b;

    rc = 1 - rc;
    s = (UInt8 CONST *) src;
    p = (UInt8 CONST *) src + src_stride;
    src_stride=src_stride-8;
    for( i = 0; i < 8; i++ )
    {
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        s+=src_stride;p+=src_stride;
    }
}
```

Interpolation eines Pixels

Interpolation eines 8x8 Blockes

Abbildung 39 C-Quelltext des Interpolationsfilters zur vertikalen Filterung

Die weiteren Funktionen zur horizontalen bzw. zweidimensionale Filterung lassen sich im Anhang A.2 der Arbeit finden. Die im vorliegenden Quelltext grau unterlegte Codezeile wird damit 64-mal pro Bildblock aufgerufen und stellt die eigentliche Kernoperation des Interpolationsfilters dar. Dabei wird die Summe der zu interpolierenden Pixel *a* und *b* gebildet, die im vorliegenden Falle immer durch einen 8-Bit-Wert repräsentiert werden. Durch ein spezielles Flag im VOP-Header des zu decodierenden Bildes wird optional die Addition einer Eins (rounding control) zur Pixelsumme angezeigt. Nach der Addition der erforderlichen Werte wird die Durchschnittswertbildung per Division durch zwei vorgenommen. Im vorliegenden Fall wird die Division als Schiebeoperation (right shift) ausgeführt, da der Prozessorkern nicht über eine Divisionseinheit verfügt und alternativ eine Softwareemulation der Division mit den daraus resultierenden Performanceeinbußen verwendet werden müsste. Die Variablen *a* und *b* werden über die Pointer **s* und **p* adressiert. Das Ergebnis der Operation wird in der Zieladresse abgelegt, die der Pointer **des* adressiert. Findet eine zweidimensionale Interpolation statt, wird diese somit insgesamt 128-mal ausgeführt. Um eine Befehlsoptimierung herleiten zu können, ist die Analyse des aus dem C-Code übersetzten Assemblercodes erforderlich, der nachfolgend (Abbildung 40) dargestellt ist. Dabei ist der C-Quellcode (links) den jeweiligen Assemblerinstruktionen (rechts) gegenübergestellt.

C- Quellcode → **ARC Assemblerinstruktionen**

```
a = *s++;                          → ldb  %r0, [%r13]   ; Pixel a in r0 laden indirekt
                                                          adressiert über R13
b = *p++;                          → ldb  %r5, [%r4]    ; Pixel b in r5 laden indirekt
                                                          adressiert über R4
                                     add  %r0, %r0, %r3 ; Pixel a + Rounding Control, Ergeb-
                                                          nis in R0
*des++ = (rc + a + b) >>1;           add  %r0, %r0, %r5 ; Pixel b + Ergebnis aus a +
                                                          Rounding Control
                                     asr  %r0, %r0      ; Interpolation durch shiften
                                     stb  %r0, [%r14]   ; Ergebnis speichern
```

Abbildung 40 C-Quelltext und entsprechender Assemblercode der Kernoperation des Interpolationsfilters

Für die Interpolation eines Pixels werden somit drei Speicherzugriffsoperationen (*ldb* und *stb*) und drei arithmetische Operationen (*add* und *asr*) ausgeführt, also insgesamt 384 Operationen pro 8x8 Block. Die nachfolgende Darstellung (Abbildung 41) des Befehlsflusses der Operationen innerhalb der Pipeline des Prozessors zeigt jedoch, dass nicht jeder Befehl innerhalb eines Taktes ausgeführt werden kann.

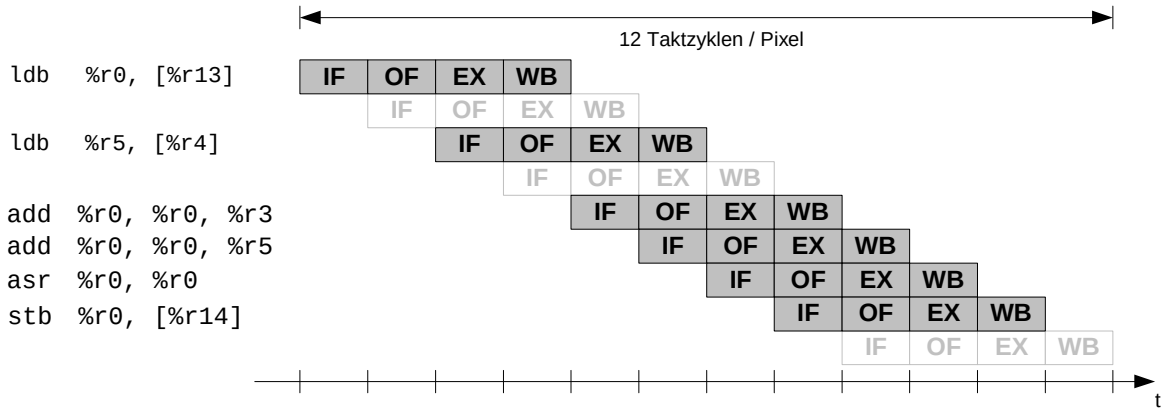


Abbildung 41 Pipeline-Befehlsfluss der Kernoperation des Interpolationsfilters

Hierfür sind die Speicherzugriffsoperationen (*ldb* und *stb*) verantwortlich, die bei der vorliegenden Prozessorarchitektur jeweils zwei Prozessorzyklen erfordern. Somit werden für die dargestellte Befehlsfolge insgesamt 12 Taktzyklen für eine Interpolation bzw. 768 Taktzyklen für die Interpolation eines kompletten 8x8 Blockes benötigt.
Da die Speicherzugriffsoperationen nicht optimierbar sind, steht nur die Befehlsfolge der Addition und anschließender Schiebeoperation für eine Optimierung zur Auswahl.
Der Datenfluss der Operation ist in der folgenden Abbildung 42 dargestellt. Dabei ist zu erkennen, dass die Operandenregister des Additionsbefehls (*add %r0, %r0, %r5*) nur mit jeweils 8 Bit genutzt werden, während die restlichen Bitpositionen der Register ungenutzt bleiben. Das Ergebnis der Addition kann dabei eine maximale Breite von 9 Bit aufweisen, jedoch wird durch die anschließende Schiebeoperation (*asr %r0, %r0*) wiederum ein 8-Bit-Ergebnis erzeugt.

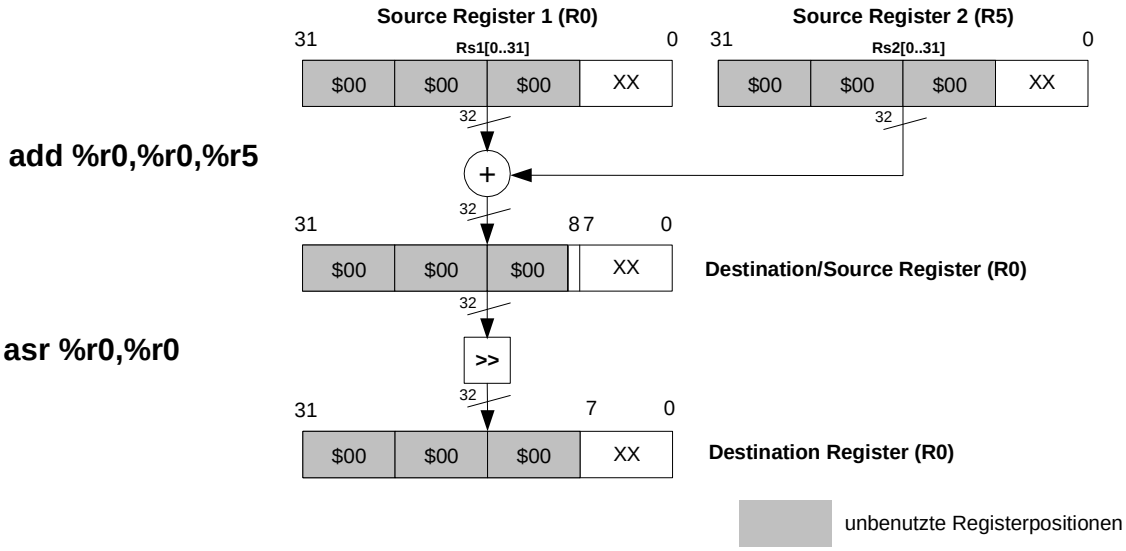


Abbildung 42 Datenfluss der Kernoperation des Interpolationsfilters

Durch die Zusammenfassung der Befehle *ADD* und *ASR* ließe sich die Befehlsfolge vereinfachen, indem ein spezieller Befehl hierfür implementiert würde. Allerdings würden auch hierbei 75% der Registerbandbreite der Operandenregister unbenutzt bleiben.
Eine Optimierung der Befehlsfolge kann jedoch unter Berücksichtigung der maximalen Datenwortbreite der Pixeldaten von 8 Bit erfolgen, indem eine gleichzeitige Verarbeitung von jeweils vier Pixeln ermöglicht wird.
Das Prinzip hierzu ist in Abbildung 43 dargestellt. Wie aus der Darstellung hervorgeht, werden vier Additionen parallel ausgeführt und die anschließende Division wiederum durch eine Schiebeoperation der Ergebnisse vorgenommen. Da die Shiftweite stets konstant um eine Bitposition durchgeführt wird, kann hier eine einfache Direktverdrahtung der jeweiligen Datenleitungen in Form eines 1-Bit-Barrelshifters vorgenommen werden, womit die Ausführungszeit der Operation nur noch von der Geschwindigkeit des Addierers abhängig ist und somit als Einzyklen-Befehl umgesetzt werden kann. Der Anwendung entsprechend wird der Befehl nachfolgend mit *INTERPOLATE* bezeichnet.

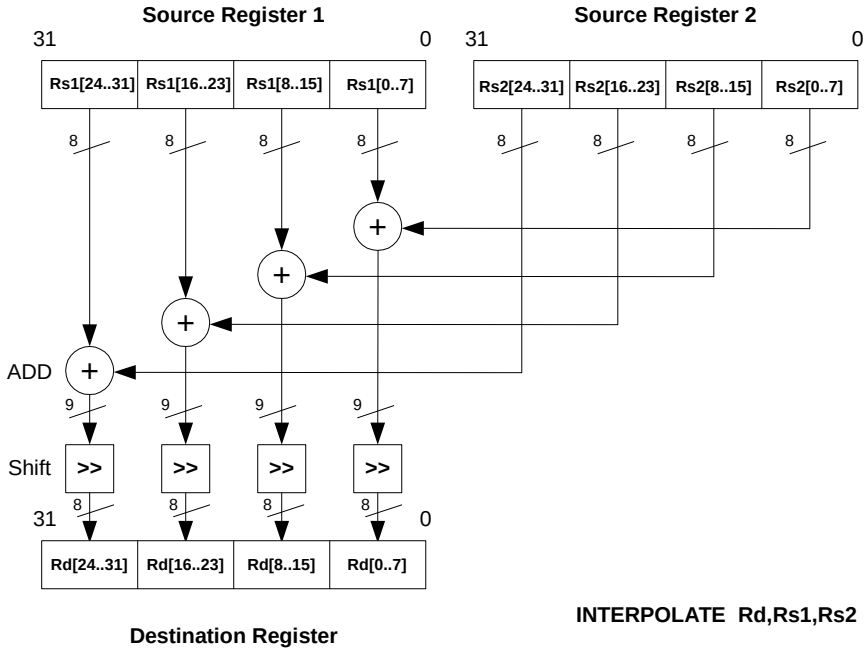


Abbildung 43 Prinzipschaltbild des Interpolationsbefehls INTERPOLATE

Bei der für die Berücksichtigung der für die MPEG-4-Rounding-Control erforderlichen Addition handelt es sich um die Addition eines Flags (0x01) zum Gesamtergebnis, dessen Verarbeitung in dem dargestellten Prinzipschaltbild nicht berücksichtigt ist, jedoch ebenfalls einfach parallel durchgeführt werden kann. Die Zuführung des Flags kann dabei auf verschiedene Weise erfolgen, wobei im vorliegenden Falle ein spezielles Prozessorregister verwendet wurde (siehe Abschnitt 4.4). Die Syntax und Semantik des Befehls sind nachfolgend dargestellt.

```
interpolate   Rd,Rs1,Rs2

          Rd [0..7]     :=    (Rs1[0..7] + Rs2[0..7] + E1) / 2
||        Rd [8..15]    :=    (Rs1[8..15]  + Rs2[8..15] + E1) / 2
||        Rd [16..23]   :=    (Rs1[16..23] + Rs2[16..23]+ E1) / 2
||        Rd [24..31]   :=    (Rs1[24..31] + Rs2[24..31] + E1) / 2
```

Die verwendeten Register sind hierbei:

```
Rs1       : Operandenregister 1 für vier Pixelwerte (a)
Rs2       : Operandenregister 2 für vier Pixelwerte (b)
Rd        : Ergebnisregister für vier interpolierte Pixel.
```

Durch die Einbettung des Befehls mit gleichzeitiger Verarbeitung von vier Pixeln in den Programmcode, stellt sich der optimierte Algorithmenkern folgend dar:

```
ld            %r0, [%r13]
ld            %r5, [%r4]
interpolate   %r0,%r0,r5
st            %r0, [%r14]    .
```

Für die Verarbeitung von jeweils vier Pixeln werden somit 4 Operationen benötigt bzw. 16 Operationen für einen 8x8 Block.
Unter der Annahme, dass der Interpolationsbefehl innerhalb eines Taktzyklus ausgeführt werden kann, ergibt sich eine Ausführungszeit von insgesamt 10 Taktzyklen (siehe Pipelinedarstellung in Abbildung 44) für jeweils vier Pixelwerte bzw. 160 Taktzyklen für die Kernoperation der eindimensionalen Interpolation eines 8x8 Blockes im Gegensatz zu 768 Taktzyklen bei der Verwendung einfacher, nicht optimierter Prozessorbefehle.

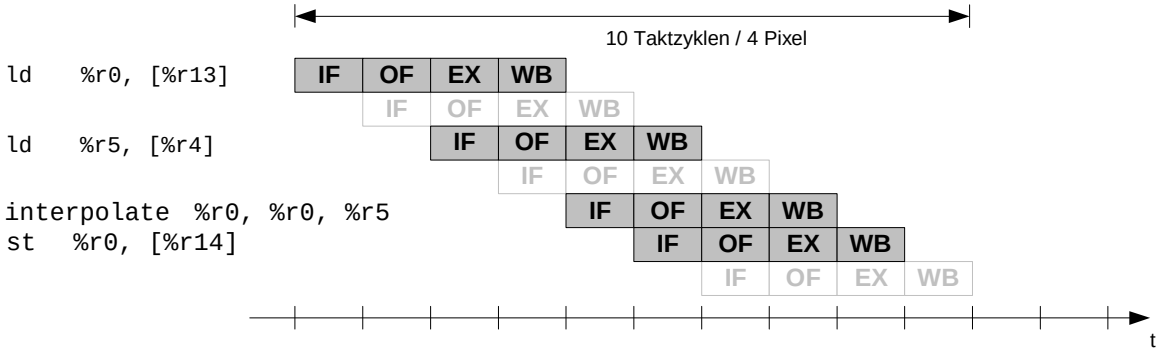


Abbildung 44 Pipeline-Befehlsfluss der Kernoperation des Interpolationsfilters mit Spezialinstruktion

Die zu erwartende Taktzyklenreduktion beträgt somit 79,2%. Unberücksichtigt bleiben hierbei der Einfluss der Kontrollflussoperationen und die Lage der Pixelwerte im Speicher, die im Falle eines nicht auf Long-Word-Grenzen adressierten Speicherzugriffs noch umsortiert werden müssen.

4.2.3 Summation

Die Summation des Prädiktors mit den rücktransformierten Koeffizienten stellt die zweite rechenintensive Funktion der Bewegungskompensation dar.
Für die Addition des Prädiktionsfehlersignals mit den Bilddaten des Referenzbildes (*SavePredPlusCoeff,* dargestellt in Abbildung 45) ist es erforderlich, einen 16-Bit-Wert mit einem 8-Bit-Wert zu verarbeiten. Der 16-Bit-Wert stellt dabei das dequantisierte und rücktransformierte Fehlersignal mit dem Wertebereich +/- 0-255 (2er Komplement) dar, das aus dem Datenstrom gelesen und decodiert wurde und in einem 16-Bit-Register mit dem Datentyp *Int16* zur Verfügung steht. Der 8-Bit-Wert entstammt dem Pixelspeicher des zuvor decodierten Bildes bzw. wurde mit Hilfe der vorhergehend beschriebenen Pixelinterpolation gewonnen.
Da es sich bei dem Ergebniswert wiederum um einen nicht vorzeichenbehafteten 8-Bit-Wert handelt, muss nach der Addition beider Werte eine Bereichsbegrenzung (Clipping) auf den Wertebereich 0-255 vorgenommen werden.
Die Kernoperation besteht hier wiederum aus zwei Teiloperationen, nämlich der eigentlichen Summation bzw. Addition und einer Bereichsbegrenzung, um zu einem 8-Bit-Wert zu gelangen. Die Kernoperation ist im folgend dargestellten C-Quellcode (Abbildung 45) wiedergegeben und grau unterlegt.
Da für die Wertebereichsbegrenzung kein spezieller Befehl zur Verfügung steht und eine Implementierung unter Zuhilfenahme entsprechender Abfragen eine höhere Rechenzeit verursachen würde, wurde die Operation mit Hilfe einer Look-Up-Table umgesetzt. Dabei wird das

Ergebnis der Addition als Adresse auf eine spezielle Clipping-Tabelle im Speicher verwendet. Der adressierte Wert stellt dann das Ergebnis der Wertebereichsbegrenzung dar.

```
void SavePredPlusCoeff(UInt8 *pred, Int16 *coef, Int32 stride, UInt8 *des)
{
    Int32 i,j;
    UInt8 l_u8;
    Int16 m_16;
    Int32 n_32;
    UInt8 CONST *p;
    Int16 CONST *c;

    DEBUG_SIGN(function_entry_SavePredPlusCoeff);
    p = pred;
    c = coef;
    stride-=8;;
    for (j=0;j<8;j++)
    {
        for( i = 0; i < 8; i++ )
        {
            l_u8 = *p++;
            m_16 = *c++;
            n_32 = (Int32) l_u8 + (Int32) m_16;
            *des++ = clip( n_32 );
        }
        des+=stride;
    }
    DEBUG_SIGN(function_exit_SavePredPlusCoeff);
}
```

Rekonstruktion eines 8x8 Blockes

Rekonstruktion eines Pixels

Abbildung 45 C-Quelltext der Funktion zur Rekonstruktion eines 8x8 Bildblockes

Auch hier basiert die Verarbeitung auf Bildblöcken mit 64 Pixeln. Der Algorithmenkern für die Verarbeitung eines Pixels ist im folgenden Programmauszug (Abbildung 46) dargestellt und wird 64-mal pro Funktionsaufruf ausgeführt. Hier ist der implementierte C-Code (links) dem hierzu korrespondierenden Assemblercode (rechts) gegenübergestellt.

C- Quellcode		**ARC Assemblerinstruktionen**
`l_u8 = *p++;`	→	`ldb %r1, [%r14]`
		`add %r14, %r14, 1`
`m_16 = *c++;`	→	`ldw.x %r0, [%r13]`
`n_32 = (Int32) l_u8 + (Int32) m_16;`	→	`add %r0, %r0, %r1`
		`ldb %r0, [%r2, %r0]`
`*des++ = clip( n_32 );`		`add %r13, %r13, 2`
		`stb %r0, [%r15]`

Abbildung 46 C-Quelltext und entsprechender Assemblercode der Kernfunktion zur Rekonstruktion eines 8x8 Bildblockes

Der Prädiktor (Bildpunkt aus dem Bildspeicher) wird mit Hilfe des Pointers **p* bzw. *R14* adressiert und als Ergebnis in der Variablen *l_u8* (Register R1) abgelegt. Das dequantisierte und rücktransformierte Prädiktionsfehlersignal (*m_16= *c++*) wird anschließend geladen. Nach der Durchführung der Addition erfolgt eine Wertebereichsbegrenzung durch den Zugriff auf die Clipping-Tabelle (**des++ = clip(n_32)*), die indirekt über den resultierenden Wert adressiert wird.

Für die Verarbeitung von 64 Pixeln bei sieben Operationen pro Pixel muss von mindestens 448 Operationen ausgegangen werden. Die Analyse der in Abbildung 47 dargestellten Pipelineverarbeitung zeigt jedoch, dass aufgrund der häufigen Speicherzugriffe eine wesentlich höhere Anzahl an realen Taktzyklen entsteht, die bei mindestens 14 Taktzyklen pro Bildpunkt bzw. 896 Takten für die komplette Blockverarbeitung liegt.

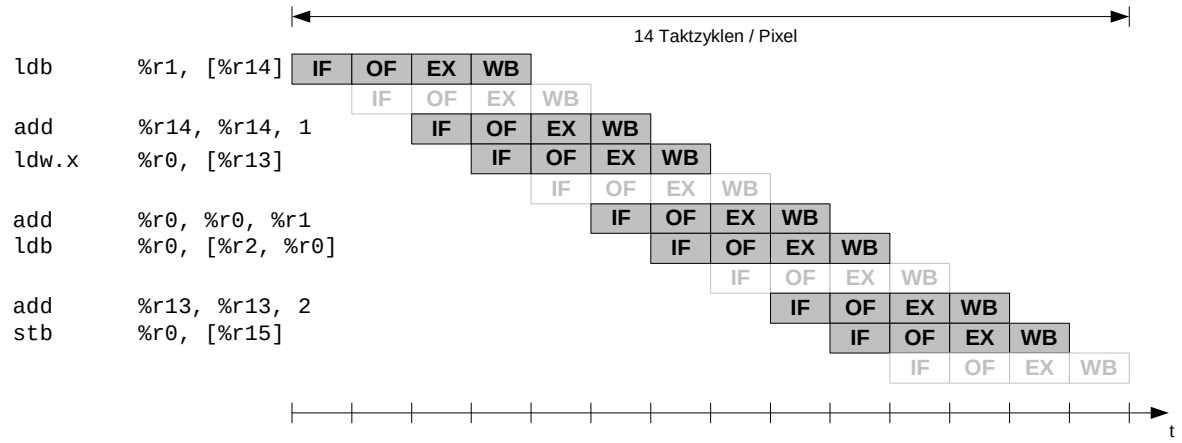


Abbildung 47 Pipeline-Befehlsfluss der Kernoperation des Interpolationsfilters

Die Darstellung des Datenflusses in Abbildung 48 zeigt auch hier wiederum, dass nur ein Teil der zur Verfügung stehenden Registerbreite verwendet wird. Die unbenutzten Registerpositionen sind zur Verdeutlichung grau unterlegt. Das Prädiktionsfehlersignal wird dabei als 16-Bit-Wert behandelt, bildet sich jedoch aus einem vorzeichenbehafteten 8-Bit-Wert.

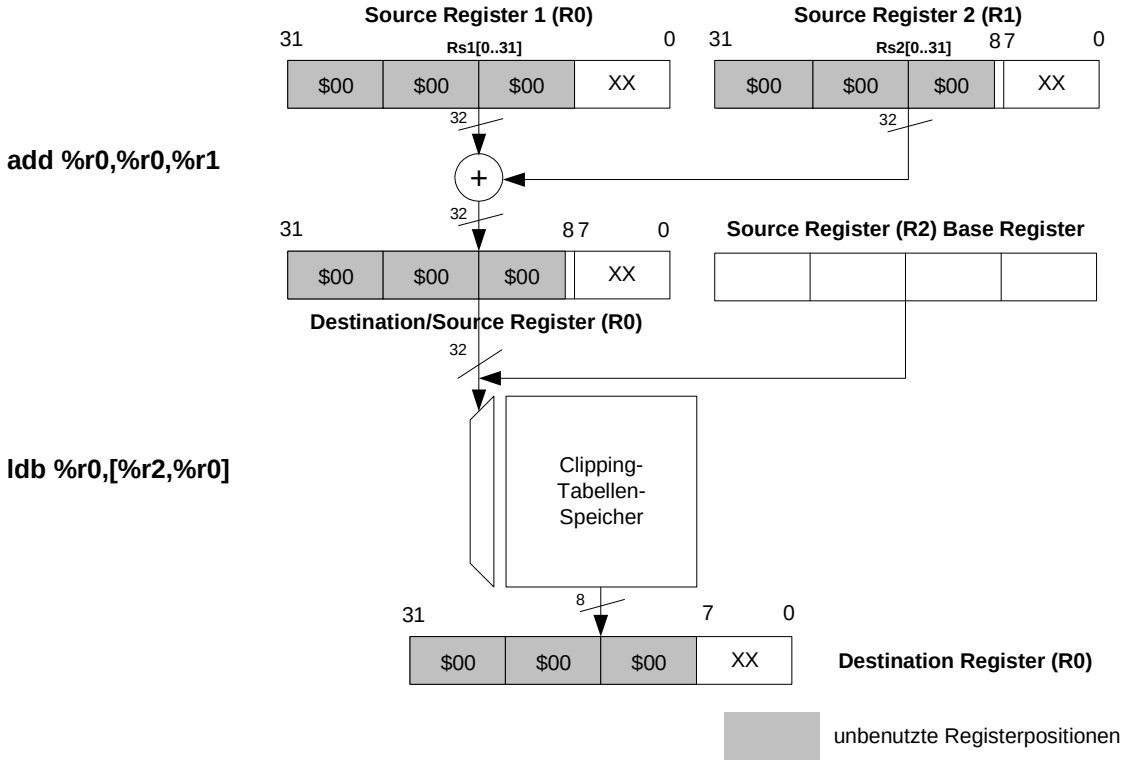


Abbildung 48 Datenfluss der Kernoperation der Pixelrekonstruktion

Auch in diesem Falle bietet sich eine parallele Verarbeitung der Daten an, da jedoch die Datenwortbreite des Prädiktionsfehlersignals 9 Bit beträgt, beschränkt sich die minimale Datenpfadbreite für einen Operanden damit auf 16 Bit. Somit kann eine parallele Verarbeitung von lediglich zwei Pixeln erfolgen. Die Basisoperation eines speziellen Befehls besteht nun aus der parallelen Addition zweier 16-Bit-Operanden mit jeweils zwei 8-Bit-Operanden und einer anschließenden Wertebereichsbegrenzung auf je ein 8-Bit-Resultat, wie in der folgenden Abbildung 49 wiedergegeben ist.

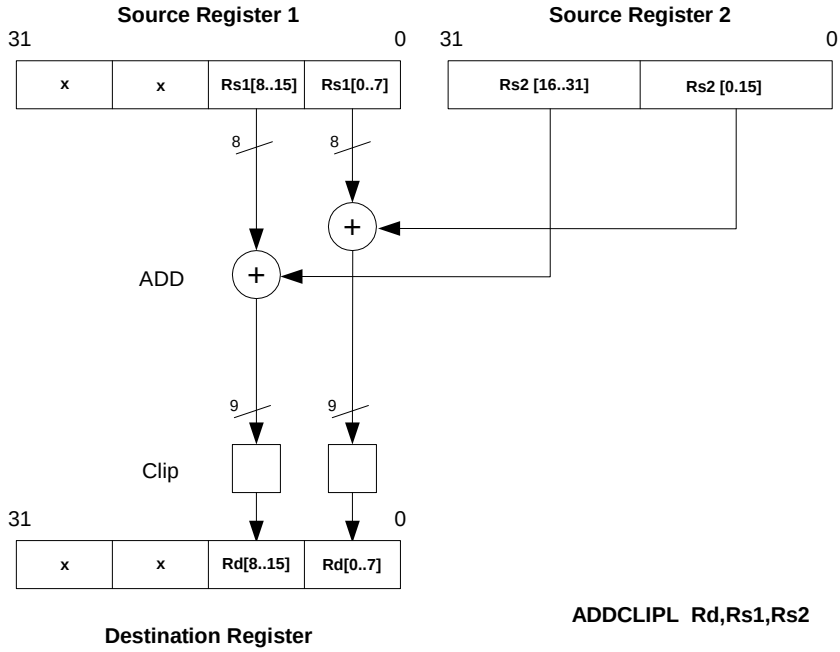


Abbildung 49 Prinzipschaltbild des Summationsbefehls ADDCLIPL

Um unnötige Speicherzugriffsoperationen zu vermeiden, können zwei unterschiedliche Befehle implementiert werden, die jeweils die oberen zwei Byte des Prädiktor-Quellregisters bzw. die unteren zwei Byte für die Addition verwenden. Die Ergebnisse der Operation werden ebenfalls in den unteren bzw. oberen Datenbereichen des Ergebnisregisters abgelegt, um eine einfache Weiterverarbeitung zu ermöglichen. Die Syntax und Semantik des Befehls, nachfolgend mit *ADDCLIP* bezeichnet, haben dementsprechend die folgende Form:

```
addclipl  Rd,Rs1,Rs2

          Rd [0..7]     :=   Clip(Rs1[0..7] + Rs2[0..15])
||        Rd [8..15]    :=   Clip(Rs1[8..15] + Rs2[16..31])

addcliph  Rd,Rs1,Rs2

          Rd [16..23]   :=   Clip(Rs1[16..23] + Rs2[0..15])
||        Rd [24..31]   :=   Clip(Rs1[24..31] + Rs2[16..31])  ,
```

mit den verwendeten Registern:

```
Rs1       : Operandenregister 1 für vier Pixelwerte (Prädiktor)
Rs2       : Operandenregister 2 für zwei Pixelwerte (Fehlersignal)
Rd        : Ergebnisregister für zwei rekonstruierte Pixel    .
```

Die Verarbeitung für jeweils vier Pixel kann mit der beschriebenen Instruktion folgendermaßen umgesetzt werden:

```
ld          %r1, [%r14]       ; lade vier 8-Bit-Prädiktoren in R1
ld          %r0, [%r13]       ; lade zwei 16-Bit-Fehlersignale
add         %r13, %r13, 1     ; erhöhe Speicherpointer R13 um 1
addclipl    %r0, %r0, %r1     ; addiere und clippe unter 16 Bit
ld          %r0, [%r13]       ; lade zwei 16-Bit-Fehlersignale
addcliph    %r0, %r0, %r1     ; addiere und clippe obere 16 Bit
stb         %r0, [%r15]       ; speichere Ergebnis        .
```

Der dargestellte Programmcode rekonstruiert vier Pixel mit Hilfe von 7 Instruktionen und benötigt für die Bearbeitung eines vollständigen Blockes, bestehend aus 64 Pixeln, demnach insgesamt 116 Operationen. Die Pipelineanalyse (siehe Abbildung 50) zeigt jedoch auch hier, dass aufgrund der Speicherzugriffe, die für das Laden der Operandenregister notwendig sind, eine höhere Anzahl an Taktzyklen resultiert.

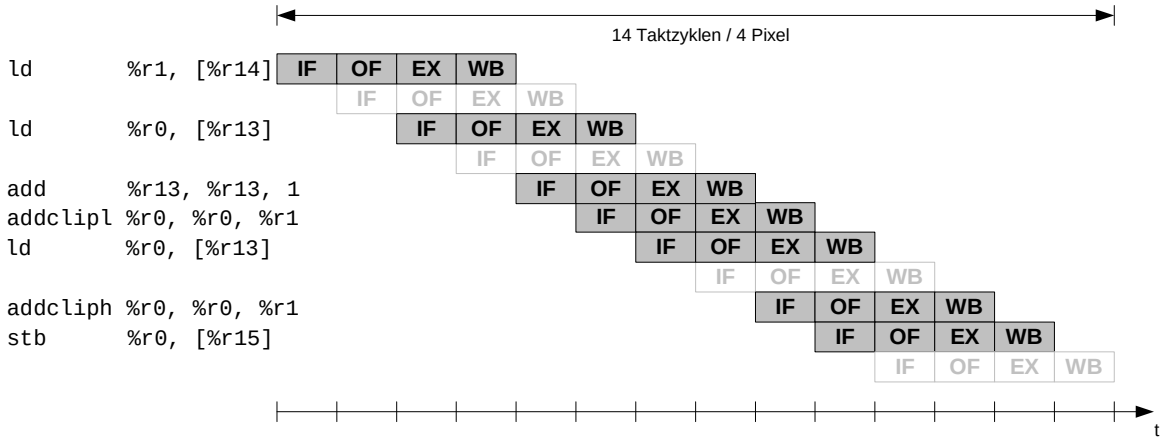


Abbildung 50 Pipeline-Befehlsfluss der Kernoperation zur Rekonstruktion eines Pixels mit der Spezialinstruktion ADDCLIP

Die Summation eines Blockes benötigt unter Verwendung des ADDCLIP-Befehls im dargestellten Kontext 224 Taktzyklen im Gegensatz zu einer Implementierung unter Verwendung der Standardprozessorbefehle mit 896 Taktzyklen. Der Realzeitgewinn für die Kernoperation kann somit mit 75% ermittelt werden.

4.2.4 Formatierung von Pixeldaten

Wie aus der Beschreibung des Befehls *INTERPOLATE* hervorgeht, müssen die Pixeldaten in der zu verarbeitenden Reihenfolge in hierfür vorgesehenen Quellregistern abgelegt sein. Da es sich jedoch bei einem Zugriff auf den Bildspeicher immer um 32-Bit-Zugriffe handelt, die Bilddaten jedoch als 8-Bit-Werte im Speicher abgelegt werden, kann es zu Speicherzugriffen kommen, die nicht auf Long-Word-Grenzen liegen. Hierbei ist maximal ein zweifacher Zugriff notwendig, um die erforderlichen vier Pixeldaten zu erhalten. Nachdem der Speicherzugriff erfolgt ist, kann jedoch nicht die direkte Weiterverarbeitung durch den Interpolationsbefehl ausgeführt werden, da die verwendeten Pixeldatenworte erst in ihrer Position innerhalb des Quellregisters angeordnet werden müssen. Um die vorhergehend beschriebenen Befehle effizient nutzen zu können, muss demnach eine Vorverarbeitung durchgeführt werden, die aus

maximal zwei Quellregistern mit insgesamt 8 Byte die Daten für das verwendete 4-Byte-Operandenregister zusammenstellt. Die folgende Darstellung in Abbildung 51 verdeutlicht das Problem nochmals. Hier wird aufgrund eines entsprechenden Bewegungsvektors das grau unterlegte Datenwort adressiert, das sich jedoch nicht auf einer Long-Word-Adresse befindet. Hierbei müssen demnach zwei Speicherzugriffe erfolgen. Wie dem dazu verwendeten Programmcode zu entnehmen ist, werden neben den zwei Speicherzugriffsbefehlen (*ld %r0, [%r10], ld %r1,[%r11]*) noch weitere 5 Instruktionen benötigt, um das adressierte Datenwort in einem Operandenregister anzuordnen.

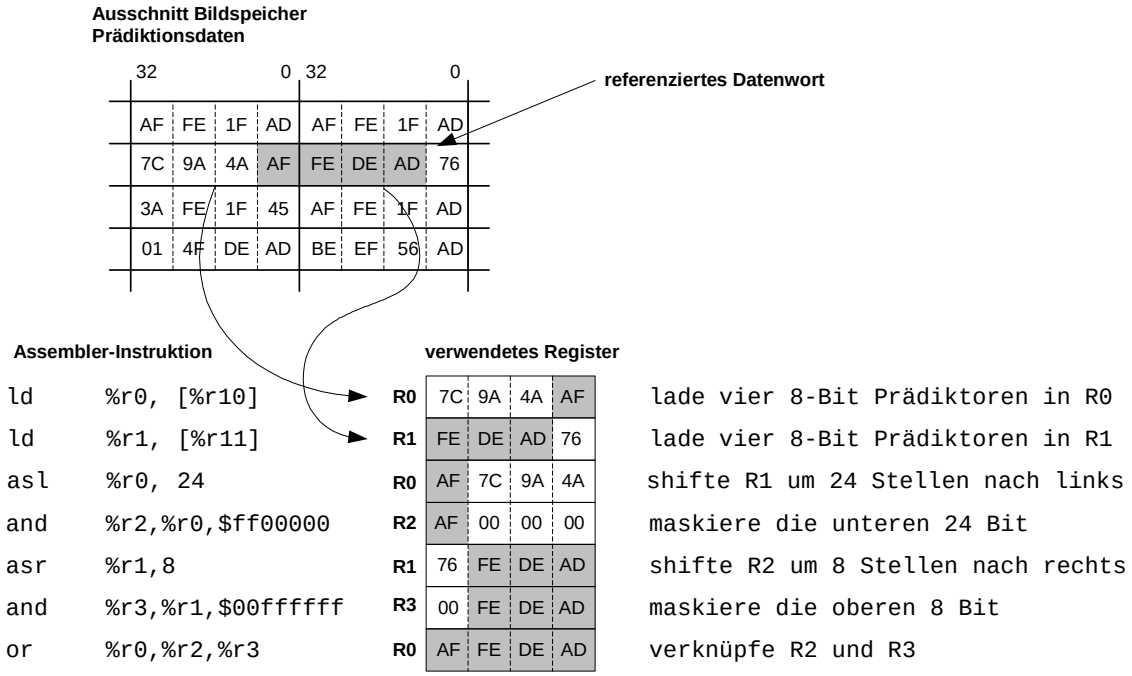


Abbildung 51 Assembler-Befehlssequenz für ungeraden Speicherzugriff

Wie die Darstellung der Verarbeitungsschritte (Abbildung 52) anhand der Prozessorpipeline wiedergibt, führen, neben den Speicherzugriffsoperationen, sämtliche weiteren Befehle zu Ausführungszeiten von jeweils einem Taktzyklus.

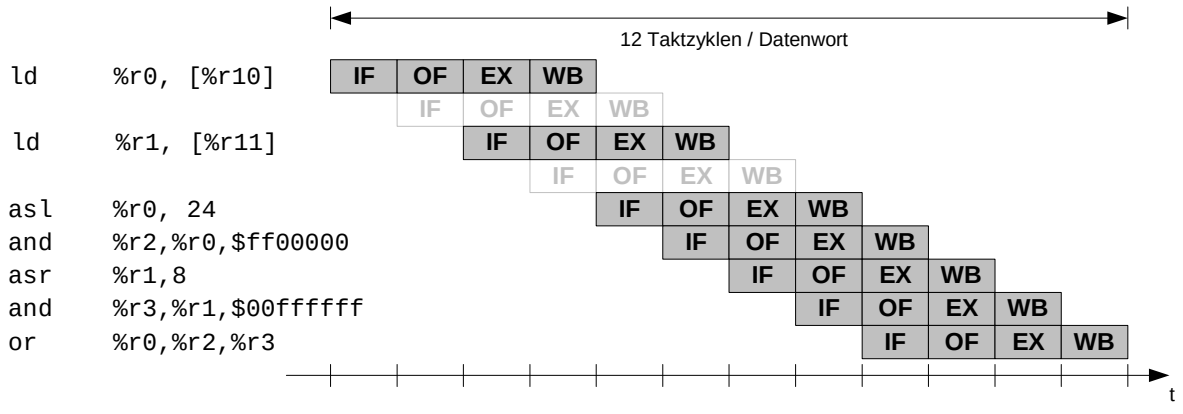


Abbildung 52 Pipeline-Befehlsfluss zur Verarbeitung ungerader Speicherzugriffe

Für jeden ungeraden Speicherzugriff müssten somit zwölf zusätzliche Taktzyklen durchgeführt werden, um das zu verarbeitende Datenwort zu formatieren.
Da die Anzahl der ungeraden Speicherzugriffe auf den Prädiktionsspeicher nicht deterministisch ist, kann es hierbei zu einer hohen Anzahl von zusätzlichen Operationen kommen.
Um die notwendige Vorverarbeitung zu vereinfachen, wird daher eine weitere Spezialinstruktion vorgeschlagen, die in der folgenden Abbildung 53 dargestellt ist. Hierbei werden zwei Operandenregister durch feste Zuordnung der Datenpfade in einem Zielregister zusammengefasst.

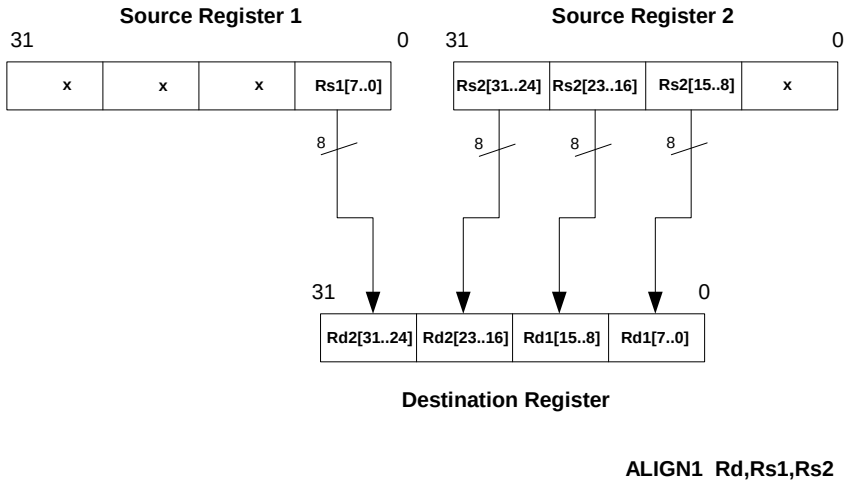


Abbildung 53 Prinzipschaltbild des Formatierungsbefehls ALIGN1

Da sich maximal drei verschiedene Kombinationen von ungeraden Adressierungen ergeben können, müssen drei verschiedene Befehle implementiert werden, die der Verschiebung entsprechend gewählt werden.
Aufgrund der geringen Komplexität der Instruktionen ist auch hier von Einzyklen-Befehlen auszugehen, deren Syntax und Semantik nachfolgend dargestellt sind:

```
align1 Rd,Rs1,Rs2

Rd [31..0]    :=   Rs1[7..0], Rs2[31..24], Rs2[23..16], Rs2[15..8]

align2 Rd,Rs1,Rs2

Rd [31..0]    :=   Rs1[15..8], Rs1[7..0], Rs2[31..24], Rs2[23..16]

align3 Rd,Rs1,Rs2

Rd [31..0]    :=   Rs1 [23..16], Rs1[15..8], Rs1[7..0],Rs2[31..24],
```

mit den verwendeten Registern:

```
Rs1      : Operandenregister 1 für vier Pixelwerte
Rs2      : Operandenregister 2 für vier Pixelwerte
Rd       : Ergebnisregister für vier kombinierte Pixelwerte  .
```

Inklusive der notwendigen Speicherzugriffsoperationen wird damit die Anzahl der benötigten Taktzyklen von 12 auf 5 Zyklen reduziert.

4.2.5 Bitstromverarbeitung

Die bereits vorgestellte Betrachtung der Laufzeituntersuchungen (Abbildung 36 auf Seite 67) der Decodierung einer kompletten Bildsequenz zeigt einen nicht konstanten Taktzyklenbedarf, der auf die unterschiedlichen Bildtypen zurückzuführen ist.
Das nachfolgend dargestellte Diagramm (Abbildung 54) zeigt jedoch noch eine weitere Abhängigkeit der pro Bild benötigten Anzahl von Prozessortaktzyklen. Hierbei ist zu erkennen, dass auch mit steigender Datenrate, bei sonst gleichen Codierungsparametern, die Anzahl der benötigten Taktzyklen pro Bild steigt. Die Steigung ist im Falle eines I-Bildes dabei stärker als im Falle der P-Bilder des gleichen Datenstroms. Im vorliegenden Falle wurde die Sequenz *Stefan* (352x288 Bildpunkte) untersucht, die mit fünf unterschiedlichen Datenraten mit einem Encoder mit Ratensteuerung codiert wurde.

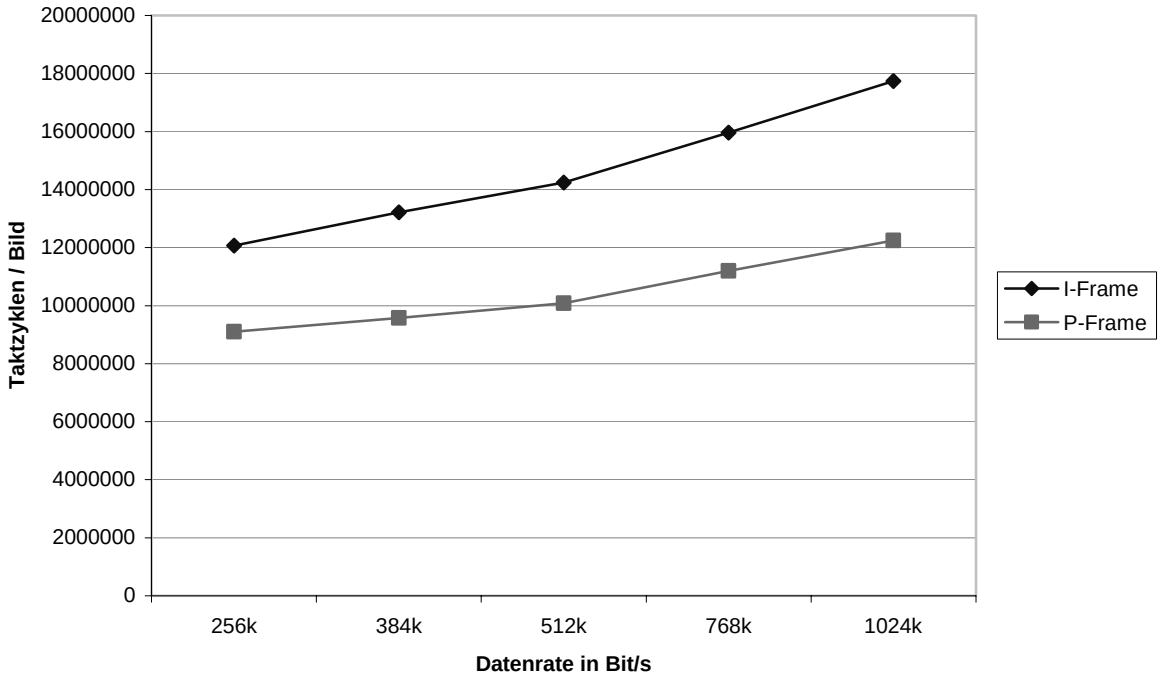


Abbildung 54 Anzahl der benötigten Prozessortaktzyklen pro Bild in Abhängigkeit von der Datenrate

Um die Echtzeitfähigkeit eines Decoders nicht nur für die Decodierung einer Anzahl von Bildern zu sichern, sondern die Einhaltung der in Tabelle 1 wiedergegebenen Echtzeitparameter (Makroblöcke/s) zu gewährleisten, muss besonderes Augenmerk auf eine effiziente Bitstromverarbeitung gelegt werden. Eine Auswertung der in Abbildung 36 dargestellten Ergebnisse ergibt, dass ca. 18% des Gesamtzyklenbedarfs für die Decodierung eines I-Bildes auf die reine Datenstromverarbeitung zurückzuführen sind. Die Analyse der hierzu zählenden Funktionen zeigt wiederum, dass ca. 91% der Rechenzeit in diesem speziellen Fall auf drei Einzelfunktionen entfallen. Dabei handelt es sich um die Funktionen zur bitgenauen Verarbeitung der Datenströme *ShowBits*, *FlushBits* und *GetBits*. Alle weiteren Funktionen dienen der tabellenbasierten Decodierung der entropiecodierten Datenstromsymbole.
Wie schon in Kapitel 3.4.1 erläutert wurde, wird mit Hilfe der Funktionen die Extrahierung von Datenworten mit variabler Bitlänge aus dem Datenstrom durchgeführt. Die Funktion *ShowBits* stellt dabei die Basisfunktion dar, mit der ein Datenwort mit gewünschter Länge ab einer bestimmten Position im Datenstrom zur Verfügung gestellt wird. Die Funktion *Flush-*

Bits dient der Modifikation des Zeigers auf den aktuellen Datenstrom, wobei *GetBits* die Kombination der vorher genannten Funktionen darstellt. Die Grundfunktion der Datenstrombearbeitung wird anhand der folgenden Abbildung 55 verdeutlicht, wobei ein Datenwort mit variabler Länge aus einem 32-Bit-Register von einer beliebigen Position extrahiert werden muss.

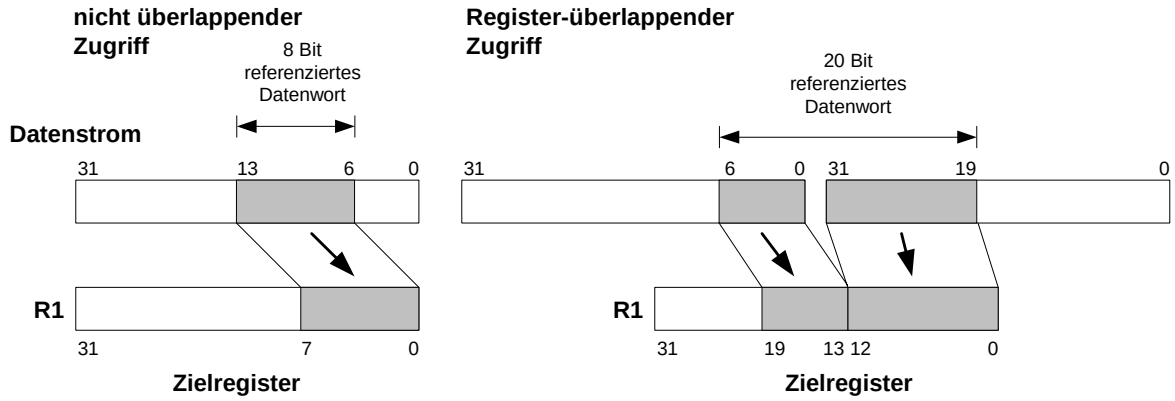


Abbildung 55 Grundoperation der Bitstromverarbeitungsfunktionen

Da der Datenstrom im Speicher in Form einer Folge von 32-Bit-Datenworten abgelegt ist und ein longwordübergreifender Zugriff stattfinden kann, müssen somit zwei Basisoperationen unterschieden werden.
Die entsprechenden Implementierungen hierzu sind im folgenden Quellcodeauszug (Abbildung 56) wiedergegeben. Der erste Teil der Funktion (dunkelgrau unterlegt) stellt dabei die Bearbeitung unter Berücksichtigung nur eines Registerzugriffs dar. Für den Fall, dass auf weitere Daten des nächsten 32-Bit-Datenwortes zugegriffen werden muss, wird der zweite Teil der Funktion (hellgrau unterlegt) ausgeführt.

C- Quellcode

```
UInt32  viShowBits(UInt32 u_length, stream * bits)
{
      UInt32 u_word ;

      u_word = bits->u32_curword;

      if (u_length<=bits->bits_valid)
      {
            u_word = u_word >> (32-u_length);
      }
      else        /* second word */
      {
            u_word = bits->u32p_buff[0];
            u_word=sw(u_word);
            bits->u32_curword |= (u_word >> bits->bits_valid);
            u_word = bits->u32_curword >> (32-u_length);
      }

      return u_word;
}
```

nicht überlappender Zugriff

überlappender Zugriff

ARC Assemblerinstruktionen

```
st    %blink, [%sp, 4]
st    %fp, [%sp]
mov   %fp, %sp
sub   %sp, %sp, 20
st    %r13, [%sp, 16]
mov   %r13, %r0
ld    %r3, [%gp, u32_curword@sda]
ld    %r2, [%gp, bits_valid@sda]
sub   %r1, 32, %r13
sub.f 0, %r13, %r2
lsr.ls      %r13, %r3, %r1
bls   .LN47.6
ld    %r0, [%gp, u32p_buff@sda]
ld    %r0, [%r0]
lsr   %r0, %r0, %r2
or    %r0, %r0, %r3
st    %r0, [%gp, u32_curword@sda]
lsr   %r13, %r0, %r1
.LN47.6:
mov   %r0, %r13
ld    %blink, [%fp, 4]
ld    %r13, [%sp, 16]   ;
j.d   [%blink]
ld.a  %fp, [%sp, 20]    ; delay
```

Abbildung 56 C-Quelltext und entsprechender Assemblercode der Kernfunktion zur Bitstromverarbeitung

Auch hier kann aufgrund der Analyse der Pipelineverarbeitung des Codes die hervorgerufene Anzahl von Taktzyklen ermittelt werden. Im Falle des einfachen Registerzugriffs (siehe folgende Abbildung 57) sind dabei 12 Zyklen zu verzeichnen.

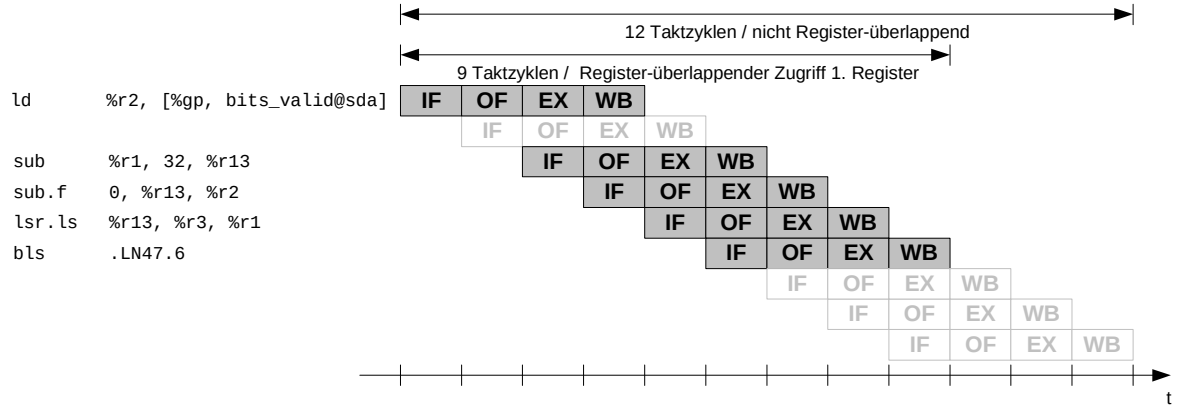


Abbildung 57 Pipeline-Befehlsfluss der Verarbeitung von Bitstromdaten bei einfachem Registerzugriff

Muss die Funktion aufgrund eines überlappenden Zugriffs weiter ausgeführt werden, sind weitere 12 Taktzyklen zur Gesamtausführungszeit hinzuzurechnen. Dieses ist in Abbildung 58 dargestellt.

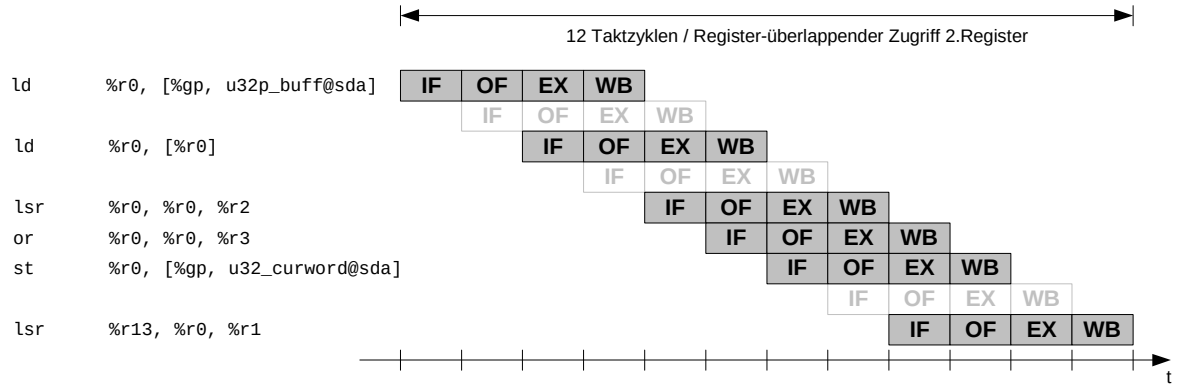


Abbildung 58 Pipeline-Befehlsfluss der Verarbeitung von Bitstromdaten mit registerüberlappendem Zugriff

Im Gegensatz zu den bisherigen Befehlen, die sich an den Konzepten einer SIMD-Architektur orientieren, kann im vorliegenden Falle keine parallele Verarbeitung von Daten vorgenommen werden, da es sich hier um einen streng sequentiellen Ablauf handelt. Hierbei wird im Falle der Verarbeitung eines VL-codierten Datenelementes erst nach der vollständigen Decodierung festgestellt, welche Länge das aktuelle Wort besitzt bzw. wo die Startposition des nächsten Datenwortes innerhalb des Datenstromes zu finden ist.

Weiterhin bietet es sich an, Befehle zu implementieren, die in ihrer Verwendung dem Funktionsaufruf der C-Funktionen entsprechen. Folgende Prozessorbefehle werden daher implementiert:

- **showbits** Rd1,Rs1
- **flushbits** Rs1 .

Hier wird jeweils *Rs1* als Operandenregister verwendet, in dem die gewünschte Datenwortbreite gespeichert ist und *Rd1* als Zielregister für das extrahierte Datenwort benutzt. Die Verarbeitung von VL-Codes kann dabei mit Hilfe des folgenden Programmauszugs umgesetzt werden:

```
ld       %r0, [%r10]        ; hole maximale Länge des VLC
showbits %r0,%r1            ; extrahiere Datenwort mit maximaler Länge
ld       %r1, [%r11,%r0]    ; lade Ergebnisdatenwort aus VLC-Tabelle
ld       %r2, [%r11,%r3]    ; lade Symbollänge aus VLC-Tabelle
flushbits%r2                ; modifiziere Zeiger auf Datenstrom .
```

Aufgrund des notwendigen Kontrollflusses, der bei der Abarbeitung der Befehle zwangsläufig entsteht, muss für den Zyklenbedarf dieser Befehle von drei Takten ausgegangen werden. Notwendigerweise muss dabei Einfluss auf die Pipeline des Prozessors genommen werden, um eine Verzögerung (stall) der Befehlsverarbeitung herbeizuführen.
Die eigentliche Datenquelle kann in der vorliegenden Form der Befehle durch eine implizite Verwendung eines Pufferspeichers erfolgen, der durch den Programmcode nicht beeinflusst wird. Hierfür bieten sich FIFO-Speicher an, die den Datenstrom in Form von 32-Bit-Daten seriell der Verarbeitungseinheit zuführen. Ebenso kann aber auch ein im Speicher linear adressierbarer Bereich unter Benutzung eines speziellen Pointers Verwendung finden.
Legt man den Assemblercode der in Anhang D.3 wiedergegebenen Bitstromroutinen zugrunde, werden durchschnittlich für den Aufruf der Routine *ShowBits* 37 Takte und 32 Takte für die Funktion *FlushBits* benötigt, womit durch die vorgeschlagenen Befehle ein Taktzyklengewinn von ca. 85%-90% zu erzielen ist.

4.2.6 Inverse Cosinustransformation

Die Cosinustransformation stellt ein zentrales Verfahren zur hybriden Videocodierung dar. Daher ist auch im vorliegenden Fall mit einem hohen Anteil an benötigter Rechenleistung für die inverse Cosinustransformation zu rechnen. Unanhängig vom jeweils verwendeten Bildcodierungstyp entsteht eine hohe Rechenlast, die auf Basis der vorgestellten Untersuchungen mit ca. 25 % im Verhältnis zum Gesamtrechenbedarf eines Bildes zu bemessen ist. Im Zuge des Einsatzes der vorhergehend beschriebenen Befehlssatzerweiterungen, ist der resultierende Anteil noch höher anzusetzen.
Die vorliegende Softwareimplementierung, des auf dem Loeffler-Algorithmus basierenden Verfahrens, benötigt für die Decodierung eines kompletten 8x8 Blockes maximal 1936 Taktzyklen unter der Annahme, dass alle Koeffizienten eines Blockes codiert sind. Für den Fall, dass eine geringere Zahl von Koeffizienten codiert ist, verfügt der im Anhang A.4 zu findende Programmcode über Abbruchbedingungen, die eine minimale Anzahl von 1410 Prozessortaktzyklen ermöglichen.
Die Analyse des aus dem C-Quelltext übersetzten Assemblercodes der IDCT zeigt aufgrund der einfachen Struktur des verwendeten Algorithmus einen nahezu optimalen Code, der sich durch den Einsatz von dedizierten Assemblerkonstrukten nur in sehr geringem Maße optimieren lässt.

Für die Laufzeitoptimierung der IDCT bieten sich daher nur folgende Varianten an:

- **Implementierung von Spezialinstruktionen**

  Durch die Umsetzung eines speziellen *Multiply and Accumulate*-Befehls (MAC), der mit 16-Bit-Operanden eine ähnliche Struktur wie die bereits vorgestellten SIMD-Befehle aufweisen könnte, lässt sich die eigentliche Signalverarbeitung der IDCT beschleunigen, da ein Großteil der Basisoperationen auf eine Multiplikation mit anschließender Addition zurückzuführen ist. Eine Parallelverarbeitung, wie sie SIMD-Instruktionen aufweisen, lässt sich jedoch aufgrund der beschränkten Registerbreite des Prozessors nicht realisieren, da das Zielregister einer MAC-Operation für den vorliegenden Anwendungsfall eine Breite von 32 Bit aufweisen müsste und somit nur ein Ergebnis gespeichert werden könnte.

- **Vergrößerung des Registerfiles**

  Aufgrund der auf 32 begrenzten Zahl an General-Purpose-Registern kommt es im Falle der vorliegenden Softwareimplementierung zu einer Vielzahl von zusätzlichen Speicherzugriffen. Insbesondere die algorithmenbedingte Transposematrix erfordert eine hohe Zahl von Speicherzugriffen für die Umsortierung bzw. den Zugriff auf die Koeffizienten. Eine Erhöhung der Anzahl nutzbarer Register kann hier zu einer Taktzyklenreduktion führen, wenn die Register vom Compiler vollständig genutzt bzw. per Assemblerinstruktion direkt zugewiesen werden. Dabei sind mindestens 64 zusätzliche Register für die maximale Zahl von Koeffizienten eines Blockes vorzusehen.

- **Coprozessorimplementierung**

  Beide vorgestellten Varianten setzen eine Modifikation des Prozessors voraus und verwenden dabei in geeigneter Weise den bestehenden Code weiter. Eine weitere Variante, die vollständig unabhängig von der implementierten Software realisiert werden kann, stellt die Umsetzung eines dedizierten IDCT-Coprozessors dar. Hierbei sind verschiedene Formen denkbar, wie die Implementierung eines Coprozessors für die 1D-Transformation, wodurch auch die vollständige 8x8 Rücktransformation verwirklicht werden kann.

Wie aus den Ausführungen zum Loeffler-Algorithmus in Kapitel 3.4.2 zu entnehmen ist, werden für die eindimensionale IDCT 11 Multiplikationen und 29 Additionen bzw. Subtraktionen benötigt, d.h. nur für die Abarbeitung der arithmetischen Operationen sind 40 Takte nötig. Insgesamt sind demnach für einen 8x8 Block 640 Taktzyklen erforderlich. Unter der Annahme, dass ein MAC-Befehl (8-mal im vorliegenden Fall) optimal eingesetzt werden kann, kann die Zahl arithmetischer Operationen auf 32 Taktzyklen (3*MUL[19], 8*MAC, 18*ADD/SUB) und damit auf 512 Taktzyklen für einen 8x8 Block reduziert werden.
Hinzuzuziehen sind die notwendigen Operationen, um einen Koeffizientenblock vollständig in den hierfür benötigten Registersatz zu laden. Da Speicherzugriffsoperationen mit 2 Taktzyklen zu veranschlagen sind, werden somit nochmals 128 Taktzyklen erforderlich. Unter Verwendung der dargestellten Prozessoroptimierungen kann somit eine softwarebasierte

[19] 2 Takte pro Multiplikation

Implementierung des Algorithmus mit einem Zyklenbedarf von mindestens 640 Taktzyklen realisiert werden.
Da die IDCT in der vorliegenden Form eine singuläre Funktion darstellt, die sich aufgrund der eindeutigen Abhängigkeiten zwischen Programm- und Datenfluss besonders einfach aus dem Verarbeitungsprozess des Decoders herauslösen lässt, bietet sich jedoch eine Implementierung in Form eines Coprozessors an.
Der Funktionsprototyp der implementierten IDCT hat die nachfolgend dargestellte Form und stellt als Schnittstelle einen 16-Bit-Adresszeiger zur Verfügung, der für den Zugriff auf Eingangs- wie auch Ausgangsdaten Verwendung findet.

```
void Idct(Int16 * ai16_dct);
```

Das Einschreiben der dequantisierten Koeffizienten in den Blockspeicher erfolgt nach dem in Abbildung 17 dargestellten Scan-Schema, wobei jedoch ausschließlich die Koeffizienten, die einen Wert ungleich null besitzen, in die jeweiligen Speicherpositionen übertragen werden und daher von einem bereits gelöschten Speicherblock ausgegangen werden muss.
Eine dem Funktionsprototypen entsprechende Implementierung verfügt demnach über einen Eingangs- bzw. Ausgangspuffer und ermöglicht damit eine lose Coprozessoranbindung an den Prozessorkern.
Für die Steuerung des Coprozessors wird nur ein Startsignal benötigt, mit dem der Beginn der Verarbeitung eingeschriebener Koeffizienten signalisiert werden kann. Das Ende der Verarbeitung wird dem Prozessor mit Hilfe eines Statusregisters oder eines Interrupts angezeigt, wonach ein Auslesen der rücktransformierten Koeffizienten erfolgt. Die folgende Abbildung 59 stellt die vorgeschlagene Architektur einer IDCT-Anbindung in Form eines Coprozessors dar.

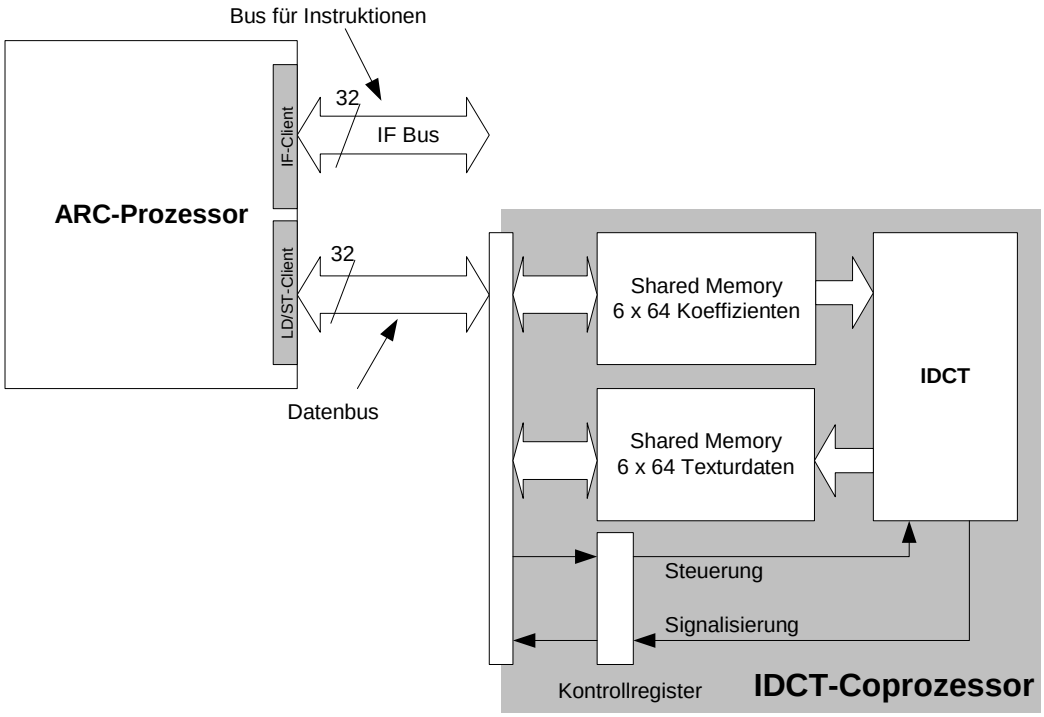


Abbildung 59 Blockschaltbild der Kopplung eines IDCT-Coprozessors

Um die erforderliche Entkopplung zwischen Hauptprozessor und Coprozessor zu erreichen, stellt ein von beiden Komponenten gemeinsam adressierbarer Speicherbereich (shared memory) die beste Variante dar.
Für die Steuerung des Coprozessors und die Signalisierung der Betriebszustände ist in Abbildung 59 ein eigenes Register vorgesehen, das sowohl vom Prozessor als auch vom Coprozessor schreibend und lesend angesprochen werden kann.
Wie schon erläutert wurde, werden die dequantisierten DCT-Koeffizienten in einen gelöschten Speicherbereich eingeschrieben. Das Löschen des Eingangsspeichers stellt damit eine weitere Funktionalität dar, die von einem IDCT-Coprozessor unterstützt werden muss, um eine möglichst lose Kopplung zu erzielen.
Aufgrund der damit möglichen Entkopplung des Hauptverarbeitungsprozesses von der Verarbeitung des Coprozessors kann eine zeitlich parallele Verarbeitung beider Decoderprozesse erfolgen, wie in Abbildung 60 dargestellt ist.

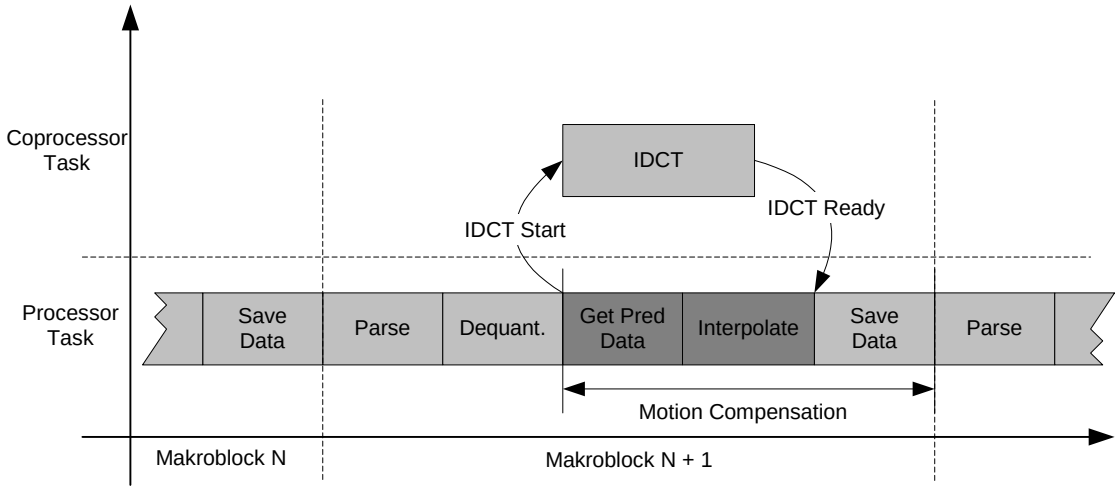


Abbildung 60 Verarbeitungsabfolge eines Makroblocks mit IDCT-Coprozessor

Hierbei ist davon auszugehen, dass der parallele Verarbeitungsprozess des Softwaredecoders eine höhere Zyklenanzahl benötigt als die Rücktransformation der DCT-Koeffizienten durch den Coprozessor.
Um den Kommunikationsbedarf zwischen Prozessorkern und Coprozessor möglichst niedrig zu halten und somit Latenzzeiten in der parallelen Verarbeitung zu minimieren, ist eine Kopplung nicht auf Blockebene, sondern auf Makroblockebene anzustreben.

4.3 Implementierung

Das zentrale Thema dieses Abschnitts stellt die Implementierung der vorhergehend beschriebenen Konzepte dar.
Die Grundlage hierzu bildet die Erweiterung eines RISC-Prozessorkerns mit speziellen Befehlen zur Videodatenverarbeitung und einem IDCT-Coprozessor.
Einführend soll ein Überblick über die verwendeten Entwurfshilfsmittel gegeben werden.
Da das umgesetzte Design eine Mischform aus Hardware- und Softwarekomponenten darstellt, wird auf die Implementierung spezieller Hardware und der dazugehörenden Software eingegangen.
Die Systemintegration des Prozessors zu einem Gesamtsystem wird ebenfalls dargestellt.

4.3.1 Designflow

Das im Rahmen der vorliegenden Arbeit entstandene System setzt sich aus einer Vielzahl unterschiedlicher Komponenten zusammen, die mit verschiedenen Entwurfshilfsmitteln entwickelt wurden. Im Wesentlichen können hier Software- und Hardwarekomponenten unterschieden werden, für die jeweils spezielle Werkzeuge zur Verfügung stehen.
Die Ausgangsbasis für den Entwurf stellt die Entwicklung eines standardkonformen MPEG-4-Softwaredecoders dar, der in einem ersten Schritt auf einer PC-Plattform umgesetzt und verifiziert wurde. Hierfür wurde der verfügbare Compiler (*Intel Performance Suite*) für die Entwicklung von Applikationen unter Microsoft WIN32-Betriebssystemen verwendet. Der Code wurde ausschließlich C-basiert implementiert. Der Einsatz plattformspezifischer Bibliotheksfunktionen wurde vermieden, um eine spätere Portierung zu erleichtern.
Die Verifikation des Decoders wurde mit Hilfe der von der ISO zur Verfügung gestellten Referenzmodelle und der ebenfalls frei verfügbaren Testdatenströme durchgeführt.
Anhand des verifizierten Decoders konnten erste Anforderungen in Bezug auf Speicherbedarf und Rechenleistung definiert werden. Weiterhin wurden schon spezielle Optimierungen durchgeführt, die der Minimierung des Speicherbedarfs dienten und somit ebenfalls im Zuge der Verifikation komfortabel mit Hilfe der PC-basierten Entwurfswerkzeuge getestet werden konnten.
Die Portierung des verifizierten Codes wurde mit Hilfe einer ARC-spezifischen Entwicklungsumgebung (*Metaware ARC Dev-Suite*) durchgeführt. Hierzu zählen C-Compiler, Assembler und ein spezieller Prozessorsimulator, der es gestattet, den entwickelten Code testen zu können, ohne über eine Hardwareimplementierung des Prozessors verfügen zu müssen.
Mit Hilfe des Simulators wurde die grundsätzliche Funktionsweise des Decoders verifiziert und eine Co-Simulation unter Verwendung der PC-basierten Implementierung durchgeführt, womit die Konformität zu den Referenzmodellen sichergestellt werden konnte.
Da die Simulation des Prozessorkerns mit dem zur Verfügung stehenden Softwaresimulator zu hohen Rechenzeiten führen kann, wurde ein Teil der Entwicklung mit Hilfe einer Gate-Array-Implementierung des Prozessors durchgeführt. Dabei besteht die Möglichkeit, den verwendeten Softwaresimulator als Front-End für den Hardwaredebugger des Systems einzusetzen und somit sämtliche Funktionen des Simulators auf einer reellen Hardware auszuführen. Die folgende Abbildung 61 zeigt die dabei verwendete Hardware.

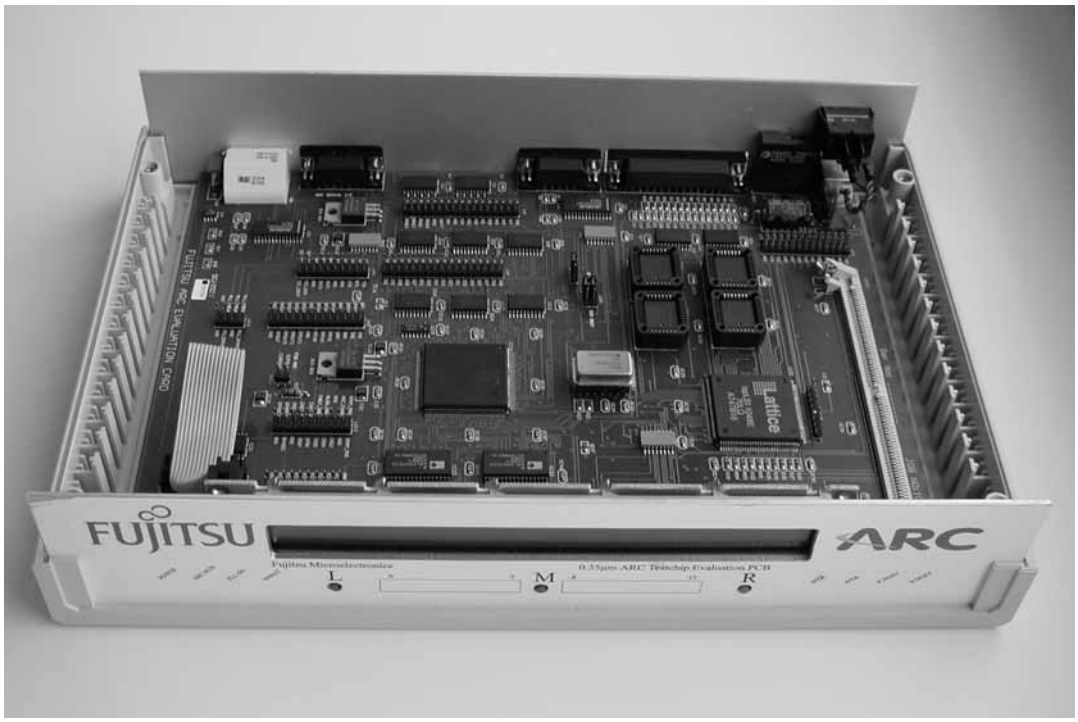


Abbildung 61 ARC-Testsystem zur Softwareverifikation

Für den Entwurf der Hardwareerweiterungen standen somit eine standardkonforme Softwareimplementierung und eine plattformspezifische Portierung des Decoders zur Verfügung.
Die Entwicklung der Hardwarekomponenten wurde auf der Basis der Hardwarebeschreibungssprache VHDL vorgenommen und der hierfür zur Verfügung stehende Simulator *ModelSim* zur Simulation und Verifikation eingesetzt. Für die Co-Simulation des Prozessorkerns mit dem zuvor entwickelten Softwaredecoder standen mehrere Varianten zur Auswahl. Eine Kopplung des *Seecode-Debuggers* der Firma Metaware mit dem verwendeten VHDL-Simulator ist zwar technisch möglich, schied jedoch aufgrund extrem hoher Ausführungszeiten aus. Hierbei wird der PC-basierte Softwaredebugger dazu verwendet, den mittels *ModelSim* simulierten Prozessorkern zu kontrollieren und den zu untersuchenden Code auszuführen. Die Kopplung beider Systeme wird über eine spezielle Softwareschnittstelle realisiert, die die eigentliche Kommunikation auf TCP/IP-Protokolle abbildet.
Die im Rahmen dieser Arbeit angewandte Methode besteht in der Verwendung des in Abschnitt 4.1.1 erläuterten Profilingsystems, das neben der Kontrolle des ausführenden Codes auch den initialen Ladevorgang der Programme in den Arbeitsspeicher des Prozessorsystems vornimmt. Auf dieser Grundlage konnte eine weitere Verifikation durchgeführt werden, die dem Test des Softwaredecoders auf dem VHDL-Modell des Prozessors diente. Auch hierbei wurden hohe Simulationszeiten erzielt, die auf der verwendeten Hardware zu Rechenzeiten von bis zu 2 Stunden pro Bild einer Sequenz (QCIF, 176x144 Pixel) führten. Für die Implementierung und Verifikation der nachfolgend beschriebenen Hardwareerweiterungen wurde daher ein spezieller Testcode entworfen, der ausschließlich dem Test der Einzelfunktionen diente. Eine Gesamtsimulation mit einer eigens hierfür geänderten Softwarevariante des Decoders wurde erst nach erfolgreicher Einzelsimulation der jeweiligen Erweiterung vorgenommen. Die notwendigen Änderungen der Software wurden wiederum mit Hilfe der PC-Simulationsumgebung durchgeführt und für die Befehlssatzerweiterungen spezielle platt-

formunabhängige Funktionen entworfen, die die Funktion der Spezialbefehle nachbilden. Somit konnte zu jedem Zeitpunkt der Entwicklung eine Co-Simulation der PC-basierten Softwareumgebung mit der VHDL-Simulationsumgebung durchgeführt werden. Im Anschluss an die Gesamtverifikation des Systems wurde eine Synthese des Prozessorkerns vorgenommen, um eine Aussage über den Flächenbedarf und die kritischen Pfade des Systems zu erhalten. Hierfür wurde die dem Verfasser zur Verfügung stehende ASIC-Bibliothek *CE81* der Firma Fujitsu verwendet und mit Hilfe der Synthesewerkzeuge Synopsys *Design Compiler DC* synthetisiert.

4.4 Befehlssatzerweiterung

Das Prozessorsystem basiert auf dem schon in 2.4.1 beschriebenen Prozessorkern, der jedoch noch neben den nachfolgend beschriebenen Befehlssatzerweiterungen durch einen Barrel-Shifter, einen 32-Bit-Multiplizierer und einen 2 kByte großen Instruction-Cache ergänzt wurde. Diese speziellen Funktionseinheiten des Prozessors sind jedoch Bestandteil des Design Kits des Prozessors und können vom Anwender frei konfiguriert werden. Die eigentliche Befehlssatzerweiterung kann relativ einfach vorgenommen werden, da die Architektur des Instruktionsdecoders hierzu 16 freie Stellen für Zwei-Operandenbefehle und 55 Stellen für Ein-Operandenbefehle im Adressraum des Instructionsets bietet.
Abbildung 62 zeigt den schon beschriebenen Prozessorkern, jedoch ergänzt um die für die Befehlssatzerweiterung erforderlichen Datenpfade und Komponenten (hier dunkelgrau unterlegt) sowie einen separaten Instruktionsdecoder (*Extension Instruction Decoder*) und die eigentliche Verarbeitungseinheit (*Extension Instruction ALU*). Durch die parallele Decodierung der Instruktionsworte wird somit der Instruktionssatz des Prozessors ausgedehnt. Die dazu zur Verfügung stehende Verarbeitungseinheit wird ebenfalls parallel zur ursprünglichen ALU des Prozessors betrieben.

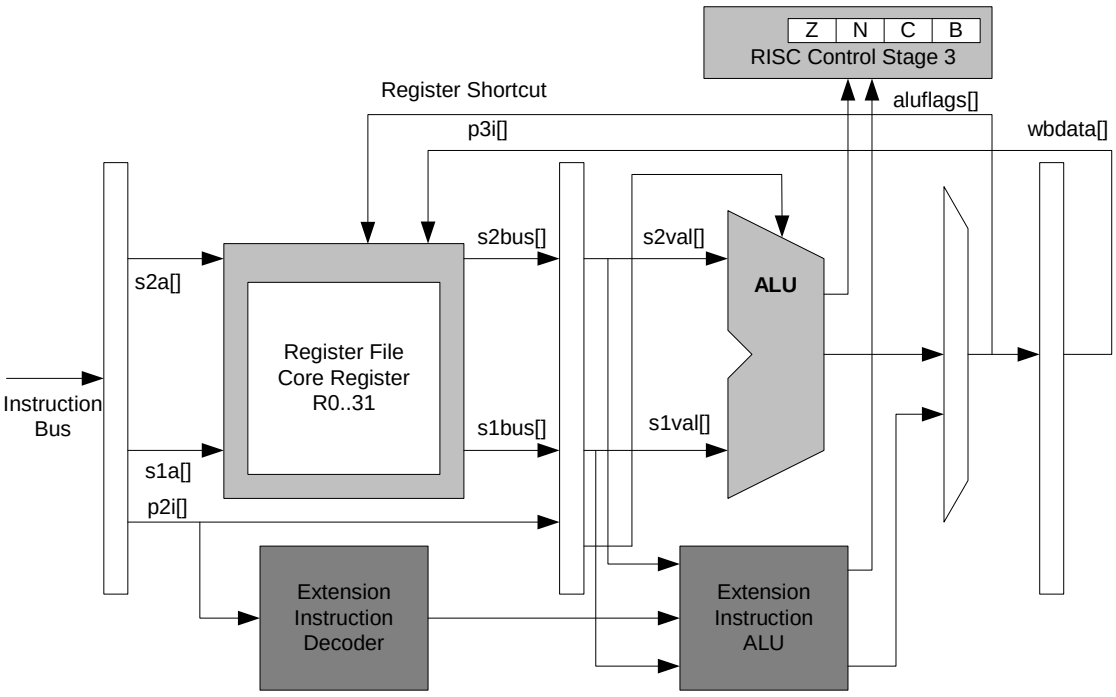


Abbildung 62 Blockschaltbild der Befehlssatzerweiterung ARC Risc-Core

Hierzu werden im Falle von Zwei-Operandenbefehlen die Ausgänge der adressierten Prozessorregister *s1val[]* und *s2val[]* an die jeweilige Komponente herangeführt.
Der Ausgang wird auf einen gemeinsamen Multiplexer geführt und anschließend über den Datenpfad *wbdata[]* für die Übernahme in das durch den Befehl adressierte Zielregister geschrieben. Für Rückwirkungen auf die Prozessorflags besteht die Möglichkeit, die bestehenden Flags zu modifizieren oder aber auch spezielle Extension-Flags zu setzen, die auch als Bedingung für Programmsprünge eingesetzt werden können. Für ALU-Operationen, die mehr als einen Taktzyklus benötigen, kann die Befehlspipeline des Prozessors angehalten (stall) werden.
Die Implementierung des in Abschnitt 4.2.1 erläuterten Befehls *ADDCLIP* (siehe Abbildung 63) wurde in der beschriebenen Form umgesetzt und stellt eine typische ALU-Erweiterung mit Hilfe eines Zwei-Operandenbefehls dar. Die Parallelität auf Sub-Word-Ebene wurde durch die Umsetzung zweier paralleler Addierer realisiert, deren Eingänge durch die Registerbusse *s1val[]* und *s2val[]* gebildet werden.
Die Bereichsbegrenzung auf einen 8-Bit-Wert wird für jedes berechnete Datenwort einzeln vorgenommen und bei Bedarf durch Schalten eines Multiplexers mit einem fest codierten Wert realisiert. Die Ausgangsdaten werden auf den gemeinsamen Bus *XResult* geführt und können somit in das adressierte Zielregister übernommen werden. Die entsprechende VHDL-Implementierung des Befehls ist in Anhang B.2 wiedergegeben.

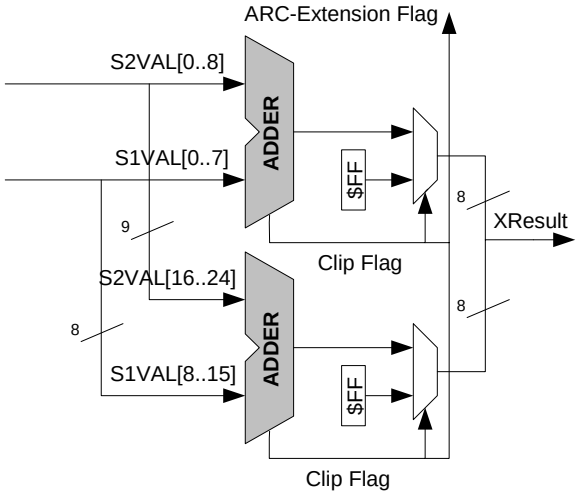


Abbildung 63 Architektur der Extension Instruction ALU für den Befehl ADDCLIP

Für die Implementierung des Befehls *INTERPOLATE* wurde eine ähnliche Struktur gewählt, die wiederum einen Zwei-Operandenbefehl zur Grundlage hat. Hierbei werden, wie aus Abbildung 64 hervorgeht, vier Sub-Words parallel verarbeitet, indem jeweils die Summe aus zwei 8-Bit-Werten gebildet wird.
Die anschließende Schiebeoperation wird durch die feste Zuordnung der Resultate in Form eines 1-Bit-Barrelshifters realisiert. Die Einbeziehung des Rounding-Control-Flags wird mit Hilfe des Extension-Flags *E1* vorgenommen, auf das über eigene Prozessorbefehle separat zugegriffen werden kann.

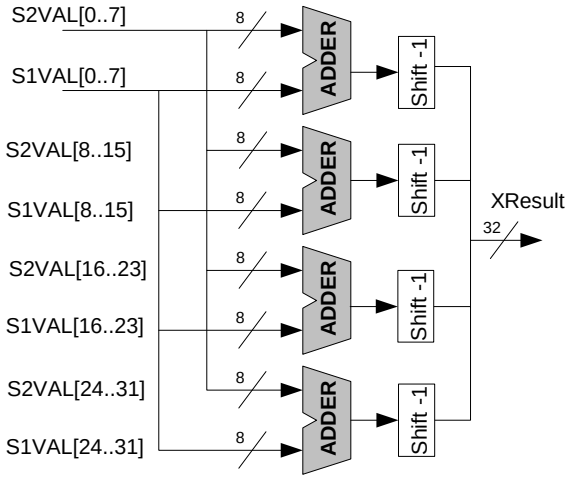


Abbildung 64 Architektur der Extension Instruction ALU für den Befehl INTERPOLATE

Die für die Vorformatierung der zu interpolierenden Pixeldaten konzipierten *ALIGN*-Befehle konnten auf einfache Art implementiert werden, da es sich hier nur um die unterschiedliche Verschaltung einzelner Datenpfade des Eingangs- und des Ausgangsregisters handelt. Für jeden *ALIGN*-Befehl wurde daher eine einfache Buszuordnung vorgenommen, die in Form des nachfolgend wiedergegebenen VHDL-Codes veranschaulicht werden kann:

```
i_align1  <= s2val(7 downto 0)  & s1val(31 downto 8);  -- B.S. ALIGN1
i_align2  <= s2val(15 downto 0) & s1val(31 downto 16); -- B.S. ALIGN2
i_align3  <= s2val(23 downto 0) & s1val(31 downto 24); -- B.S. ALIGN3.
```

Alle bisher dargestellten Befehle können aufgrund ihrer einfachen Struktur als 1-Zyklenbefehle in die Verarbeitungspipeline des Prozessors eingebunden werden.
Für die in Abschnitt 4.2.5 entworfene Befehlssatzerweiterung zur Verarbeitung serieller Datenströme wurde die in der nachfolgenden Abbildung 65 dargestellte Architektur umgesetzt.

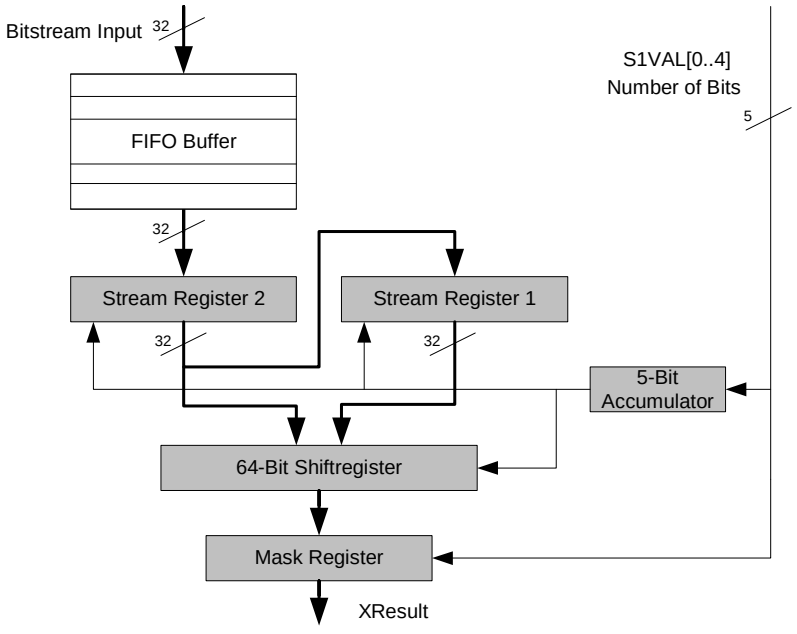


Abbildung 65 Blockschaltbild der Verarbeitungseinheit für serielle Datenströme

Hierbei gelangen die Bitstromdaten in Form von 32-Bit-Datenworten in einen Eingangspuffer, der als Quelle für die Bitstromregister (Stream-Register 1, 2) dient. In beiden Registern befindet sich nach einem initialen Ladevorgang die parallele Repräsentation des Datenstroms. Mit Hilfe eines 64-Bit-Barrel-Shifters kann nun aus diesem Datenstrom ein beliebiges Datenwort mit einer maximalen Länge von 32 Bit ausgelesen und am Ausgang des Registers einem Maskierungsregister zur Verfügung gestellt werden. Bei einem Überschreiten der Registergrenzen muss ein weiterer Nachladevorgang durchgeführt werden, der durch das Akkumulieren der bereits verwendeten Bitbreiten angezeigt wird. Durch die Einzelsteuerung der Funktionen des Akkumulators und der Shiftweite des Schieberegisters können die Einzelbefehle *FlushBits* und *ShowBits* umgesetzt werden.
Die Befehlserweiterung wurde als Ein-Operanden-Befehl umgesetzt und steuert aus dem adressierten Register die Schiebebreite des Schieberegisters und die Positionen des Maskierungsregisters. Der dazugehörige VHDL-Code kann in Anhang B.4 eingesehen werden.

4.5 IDCT-Coprozessor

Wie auch die zuvor beschriebenen Befehlssatzerweiterungen, handelt es sich im Falle des IDCT-Coprozessors um eine Komponente, die aufgrund einer universellen Schnittstelle auch in anderen prozessorbasierten Systemen eingesetzt werden kann.
Der Coprozessor besteht dabei aus einem Kommunikationsspeicher, der über eine, in diesem Falle angepasste Schnittstelle an ein Speichersystem angeschlossen werden kann, dem eigentlichen IDCT-Verarbeitungskern und einer Ablaufsteuerung, die sämtliche Kontroll- und Steuerungsaufgaben des Coprozessors bereitstellt. Die IDCT-Kernkomponente stellt eine spezielle Implementierung einer doppelt genutzten eindimensionalen IDCT mit Transpose-Speicher dar. Das Konzept der Nutzung zwei eindimensionaler Transformationen wurde schon in Abschnitt 3.4.2 vorgestellt, wobei in der vorliegenden Implementierung jedoch nur ein Rechenwerk aufgebaut wurde, das jedoch zweifach genutzt werden kann. Die eigentliche Umsetzung des Rechenwerks wurde mit Hilfe der ebenfalls schon beschriebenen DA-Architektur (siehe hierzu Abschnitt 3.4.2) realisiert. Der hier verwendete IDCT-Kern entstand nicht im Rahmen dieser Arbeit, sondern lag schon in Form eines IPs[20] als Ergebnis eines vorangehenden Projektes vor, wo er in einem MPEG-2-HDTV-Decoder (siehe [69] und [70]) eingesetzt wurde. Aufgrund der doppelten Nutzung eines Rechenkerns weist die Gesamtkomponente eine Latenzzeit von 128 Taktzyklen auf, die zu Beginn der Verarbeitung eines 8x8 Blockes zu berücksichtigen ist. Nach Überbrückung der Latenzzeit wird pro Taktperiode ein Eingangsdatenwort übernommen und liefert am Ausgang ein Ergebnisdatenwort. Um einen möglichst hohen Datendurchsatz zu erzielen, ist es daher erforderlich, die Eingangsdaten ohne Unterbrechung der IDCT zu übergeben. Weiterhin kann die Latenzzeit minimiert werden, indem anstatt nur eines Blockes die Daten eines kompletten Makroblockes, d.h. 6 x 64 Koeffizienten, übergeben werden.
Da die Eingangsdaten der IDCT jedoch nicht kontinuierlich durch den Prozessor bereitgestellt werden können, ist die Verwendung eines Pufferspeichers erforderlich, der alle Koeffizienten eines Makroblocks speichern und taktsynchron bereitstellen kann. Hierfür wurde ein statischer Speicher als Kommunikationsspeicher vorgesehen, der als Single-Port-Implementierung ausgeführt wurde. Der Speicher wird dabei während der Decodierungsphase der Bitstromdaten dem Prozessor über das Speicherinterface zugänglich gemacht und dient nach Abschluss des Einschreibens der Eingangsdaten der IDCT als Arbeitspeicher.

[20] IP – engl. Intellectual Property

Um eine lose Kopplung zwischen beiden Systemen zu erzielen, wurde auch für die Ausgangsdaten, d.h. für die rücktransformierten Koeffizienten, ein eigener Speicher vorgesehen, der in gleicher Weise durch den Prozessor angesprochen wird.
Die Größe der Pufferspeicher wurde dabei an der Wortbreite der Eingangskoeffizienten ausgerichtet, die 12 Bit beträgt. Für die Speicherung von 384 Koeffizienten werden somit 768 Byte benötigt. In der vorliegenden Implementierung wurden zwei jeweils 1024 Byte fassende Speicher (in Abbildung 85 in Anhang C.4 mit *ramin* und *ramout* bezeichnet) verwendet, die neben der Pufferung der Koeffizienten auch als schneller Speicher für den Prozessor verwendet werden.
Die Steuerung des Coprozessors wird mit Hilfe einer Zustandsmaschine mit sieben Zuständen vorgenommen, deren Aufgabe im Wesentlichen aus der Generierung aufeinander folgender Adressen für die Pufferspeicher besteht. Die nachfolgende Abbildung 66 gibt die Zustandsmaschine der Coprozessorsteuerung wieder.

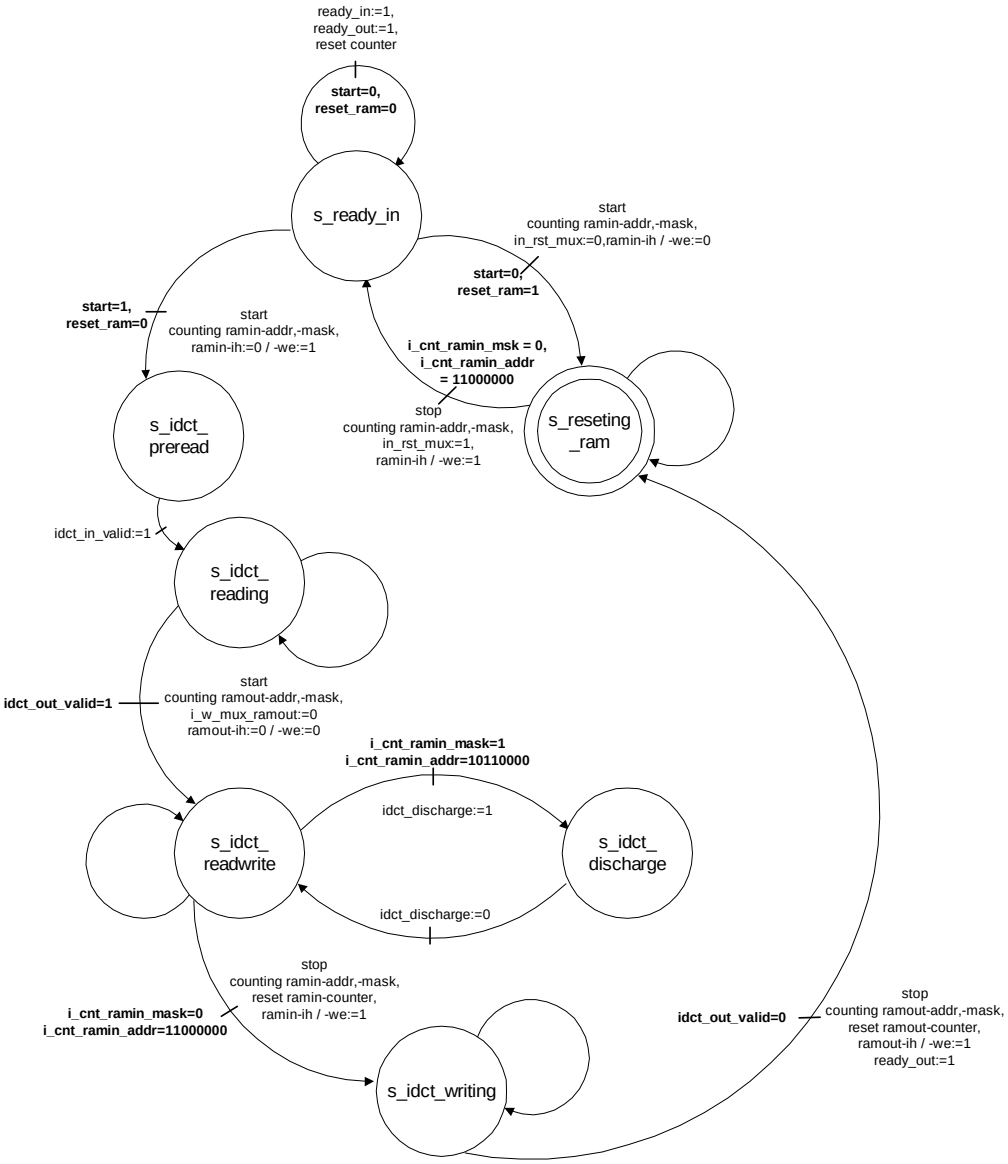


Abbildung 66 Zustandsmaschine der Coprozessorsteuerung

Wie Abbildung 66 zu entnehmen ist, wird die initiale Steuerung des Coprozessors stets durch den Hauptprozessor vorgenommen, wobei zwei Vorgänge ausgelöst werden können. Durch Schreiben des Command-Registers durch den Hauptprozessor lässt sich das Löschen (Zustand *s_reseting_ram*) des Eingangsspeichers und die Verarbeitung der eingeschriebenen Koeffizienten (Zustand *s_idct_preread*) herbeiführen.
Der Status des Coprozessors kann mittels Lesen eines speziellen Statusregisters abgefragt werden, womit die Fertigstellung eines Transformationsvorgangs angezeigt wird.
Die Steuerung der IDCT aus dem Programmcode wird durch den folgenden Programmauszug verdeutlicht.

```
#ifdef HARDWARE_IDCT
      // all of the coded Macroblock data has been written to the IDCT-RAM
      // now the ICDT can start
#ifdef SIMULATE_HARDWARE_IDCT
      // Idct for 6 blocks 8x8
      for (i=0; i<6; i++) Idct((Int16 *)IDCT_RAM_IN + i*64);
      // copy results to IDCT_RAM_OUT
      for (i=0; i<192; i++) {
      IDCT_RAM_OUT[i] = IDCT_RAM_IN[i];
      //     ((Int8 *) IDCT_RAM_OUT)[4*i]  =  ((Int8 *) IDCT_RAM_IN)[4*i+3];
      //     ((Int8 *) IDCT_RAM_OUT)[4*i+1]=  ((Int8 *) IDCT_RAM_IN)[4*i+2];
      //     ((Int8 *) IDCT_RAM_OUT)[4*i+2]=  ((Int8 *) IDCT_RAM_IN)[4*i+1];
      //     ((Int8 *) IDCT_RAM_OUT)[4*i+3]=  ((Int8 *) IDCT_RAM_IN)[4*i];
            }
#else  // real Hardware_IDCT
      *(int *)IDCT_Command_Register = IDCT_START;

      // while (!( *(int *)IDCT_Status_Register & IDCT_READY_OUT))
            *(int *)IDCT_Status_Register=0;
#endif

#ifdef SIMULATE_HARDWARE_IDCT
      for (i=0; i<192; i++) IDCT_RAM_IN[i]=0;  // RAM clearing
#endif
```

Der Programmauszug beinhaltet dabei auch Programmcode, der für die Simulation ohne die Verwendung einer Hardware-IDCT eingesetzt wurde.
Ein weiteres Merkmal des Coprozessors ist die Möglichkeit der eigenständigen Initialisierung des Eingangspuffers. Das Löschen des Speichers kann dabei entweder programmgesteuert durch den Prozessor eingeleitet werden oder aber nach vollständiger Abarbeitung aller Koeffizienten erfolgen.

4.6 Gesamtsystem

Aufbauend auf dem erweiterten Prozessorkern wurde ein Gesamtsystem (nachfolgend als VPU bezeichnet) entwickelt, das als eigenständiger Videodecoder ohne weitere Komponenten, wie auch als Zusatzkomponente in größeren Systemen, so z.B. als SOCs[21] in Abbildung 84 in Anhang C.3 dargestellt, verwendbar ist.
Der Decoder stellt nach außen Schnittstellen zur Verfügung, die es gestatten, die internen Speicherbereiche des Systems zu beschreiben bzw. auszulesen, um codierte bzw. decodierte Daten ein- und auslesen zu können. Neben den Daten gelangt auch der eigentliche Programmcode des Decoders über die Speicherschnittstelle in den Programmspeicher des Systems.
Nach erfolgreicher Initialisierung des Programmspeichers wird der Start des geladenen Programms mit Hilfe eines externen Resets ausgelöst.
Weiterhin verfügt das System über die Möglichkeit, mit Hilfe eines externen Debugginginterfaces (*DBI-Port*), den Programmcode des Prozessors im Einzelschrittbetrieb unter Zuhilfenahme eines PCs und eines speziellen Debugmonitors zu untersuchen.
Für die Signalisierung verschiedener Betriebszustände stehen eine GPIO-Schnittstelle und mehrere Hardwareinterruptsignale zur Verfügung, die programmgesteuert vom Prozessorkern des Systems gesteuert und abgefragt werden können.
Die interne Struktur der VPU ist in Abbildung 67 dargestellt und kann grob in die drei funktionalen Blöcke Prozessorsystem, Systeminterface und Speichersystem unterteilt werden.
Das Prozessorsystem (*arc_core*) wird gebildet durch den befehlssatzerweiterten RISC-Prozessor, dessen Speicherinterface (*ld/st_client, if_client*) mit dem Speichersystem verbunden ist. Das Speichersystem besteht aus den Komponenten Arbitrator (*vpu_arbitrator*), Multiplexer (*vpu_arb_mux*) und den verwendeten Speicherblöcken inklusive des IDCT-Coprozessors.
Für die Ausgabe der Videodaten wurde ein spezielles Speicherinterface entworfen, das sich als sog. Shadow-RAM (*shadow RAM Port*) in das System einbettet. Hiermit besteht die Möglichkeit, Videodaten unabhängig von den jeweiligen Speicherzugriffen des Prozessors auszulesen und an eine Videoausgabeschnittstelle zu übergeben. Somit können wirkungsvoll die durch den Speicherzugriff des Prozessors verlangsamten Ausgabevorgänge beschleunigt werden. Die Einbindung in ein Gesamtsystem erfolgt über das Slave-Port, das sich wie eine Peripheriekomponente in ein bestehendes Speichersystem einbinden lässt.
Die restlichen Komponenten dienen der Steuerung und Systemintegration der VPU in ein Gesamtsystem und werden im Rahmen der Arbeit nicht weiter diskutiert.

[21] SOC – System on a Chip

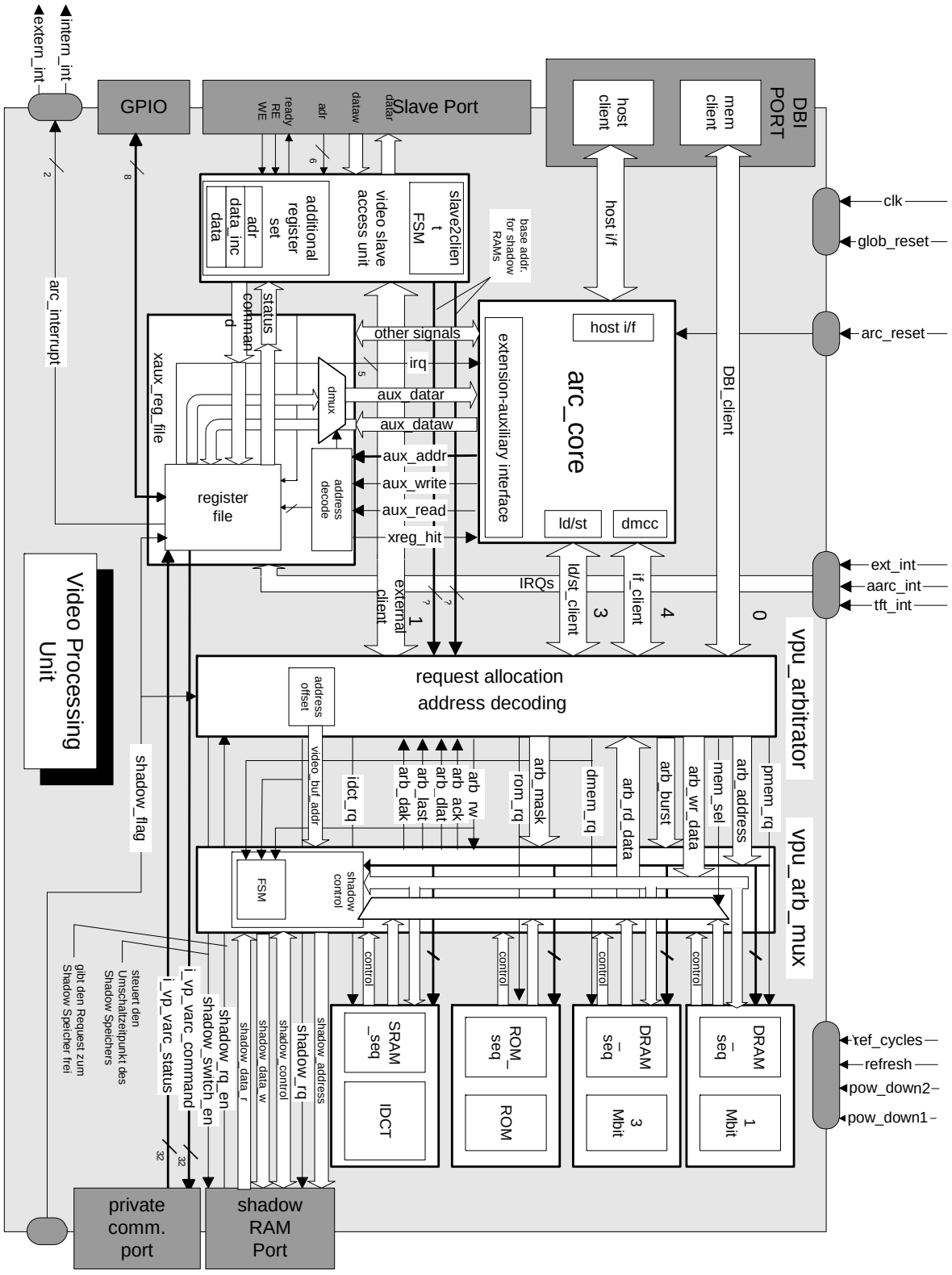


Abbildung 67 Blockschaltbild des MPEG-4-Videodecoders

4.6.1 Speichersystem

Das Speichersystem des Decoders kann wiederum grob in drei Funktionsgruppen unterteilt werden. An erster Stelle sind die eigentlichen Speicherkomponenten zu nennen, die für unterschiedliche Datenstrukturen vorgesehen sind und aufgrund ihrer Zugriffscharakteristik mit Hilfe unterschiedlicher Speicherarchitekturen umgesetzt werden.
Neben den Speichern für Programmcode und Daten stehen Speicherbereiche für Tabellen und Bilddaten zur Verfügung. Die Ankopplung des IDCT-Coprozessors ist ebenfalls über einen eigenen Speicherbereich realisiert.
Die zweite Komponente stellt der bereits erwähnte Multiplexer dar, der alle Speicherkomponenten zu einem linearen Adressbereich zusammenfasst.
Aufgrund der speziellen Architektur des verwendeten Prozessors und einer Anzahl weiterer Komponenten, die aktiven Zugriff auf den gebildeten Gesamtspeicherbereich haben müssen, ist es erforderlich, einen speziellen Arbitrator zu implementieren, der den Zugriff auf das Bussystem des Multiplexers und somit auf die einzelnen Speicherbereiche steuert.
Hierbei müssen die unterschiedlichen Busprotokolle der jeweiligen Clients an das Protokoll des Multiplexers angepasst werden. Weiterhin ist es erforderlich, den Zugriff der einzelnen Clients durch ein Priorisierungsschema derart zu regeln, dass ein gleichzeitiger Mehrfachzugriff in einer bestimmten Reihenfolge abgearbeitet wird. Hierfür wurde ein Arbiter entworfen, der für die unterschiedlichen Clients Prioritätsstufen vorsieht. Folgende Clients werden vom Arbiter unterstützt:

- **Debug Client**
  Mit Hilfe des Debug Clients steht ein Debugginginterface zur Verfügung, das unter Zuhilfenahme einer speziellen externen Hardware den Einzelschrittbetrieb des Prozessors unter Steuerung eines PCs ermöglicht. Hiermit wird die Fehlersuche in der auf dem Prozessor ablaufenden Software stark vereinfacht. Da der Zugriff auf den Speicher des Systems zu jedem Zeitpunkt gewährleistet sein muss, wurde dem Debugginginterface die höchste Priorität (0) zugeordnet.

- **System Client**
  Neben dem Debug Client stellt der System Client die zweite Möglichkeit dar, extern auf den Gesamtspeicher des Systems zuzugreifen. Hiermit erfolgen die komplette Steuerung und der Programmladevorgang unter Einsatz eines Systemprozessors. Um den bevorzugten Zugriff auf den Speicher vor dem Prozessor zu gewährleisten, wurde die Prioritätsstufe 1 gewählt.

- **Load / Store Client**
  Der Zugriff des Prozessors auf die Datenbereiche erfolgt mit Hilfe des LD/ST Clients, der die Prioritätsstufe 2 erhält.

- **Instruction Fetch Client**
  Das Laden der auszuführenden Befehle aus dem Programmspeicher in den prozessorinternen Programmcache wird mit Hilfe des Instruction Fetch Clients realisiert. Da die Zugriffscharakteristik des IF Clients aufgrund des ansteuernden Cachecontrollers hauptsächlich von Burstzugriffen geprägt wird, wurde hierfür die Prioritätsstufe 3 gewählt.

Ein detailliertes Blockschaltbild des Arbitrators ist in Abbildung 83 in Anhang C.2 dargestellt.

Die Zuordnung der vom Adressgenerator des Prozessors erzeugten Adressen zu den einzelnen Speicherbereichen des Systems wurde mit Hilfe eines Adressvergleichers und eines Multiplexers realisiert, die zusammen eine Einheit bilden und in Anhang C.1 als Blockschaltbild wiedergegeben sind. Speicherzugriffe werden damit auf die jeweilige Komponente umgeleitet, indem durch die Signalisierung des Adressvergleichers der Datenpfad der zu selektierenden Komponente auf den Daten- und Adressbus gelegt wird.

4.7 Softwarearchitektur

Die grundsätzliche Softwarearchitektur des Videodecoders wird durch die Verwendung in einer Master-/Slave-Konfiguration geprägt, d.h. dem Decoder wird nach einer Initialisierungsphase eine AU[22] übergeben, die nach Aufforderung durch einen Master decodiert wird. Anschließend werden die decodierten Videodaten in einen definierten Adressraum des Decoders geschrieben.
Die Fertigstellung eines Bildes wird dem Master durch Setzen eines Flags im Register-File des Slave-Ports signalisiert. Anschließend ist der Decoder wieder bereit, neue komprimierte Daten in Empfang zu nehmen und wiederum zu decodieren.
Der eigentliche Videodecoder ist also in ein Hauptprogramm eingebettet, das eine Endlosschleife bildet und über einen externen Master gesteuert werden kann. Das entsprechende Flussdiagramm hierzu findet sich in Anhang F.1.
Der Funktionsaufruf des eigentlichen Decoders wird entweder im Initialisierungsmodus oder im Decodermodus verwendet.
Im Falle des Initialisierungsmodus wird dem Decoder der MPEG-4-VOL-Header des Videodatenstroms übergeben, anhand dessen der Decoder die notwendigen internen Speicherstrukturen für den Decodierungsvorgang initialisiert.
Da der Decoder über keine eigenen systemspezifischen Speicherverwaltungsmechanismen verfügt, muss die Bereitstellung der notwendigen Speicherbereiche außerhalb der Decoder-API erfolgen. Hierfür gibt der Decoder nach dem Initialisierungsvorgang die für den aktuellen Datenstrom notwendige Speichergröße zurück, womit über einen Adresszeiger dem Decoder ein entsprechend großer Arbeitsspeicherbereich angezeigt wird.

Initialisierungsmodus:

```
Boolean DecodeVideo (   dec_struct * dec_str,
                        UInt8 *AUbuf,
                        UInt32 buffersize,
                        TRUE);
```

Decodermodus:

```
Boolean DecodeVideo (   dec_struct * dec_str,
                        UInt8 *AUbuf,
                        UInt32 buffersize,
                        FALSE);
```

Die Übergabe des Bitstrompuffers, in dem sich die aktuelle AU befindet, erfolgt in Form eines eigenen Adresszeigers **AUbuf*. Die Kommunikation der restlichen Parameter vom

[22] AU- Access Unit

Hauptprogramm zum Decoder und umgekehrt wird über eine spezielle Speicherstruktur (*dec_str*) realisiert, über deren Basisadresse Hauptprogramm und Decoder verfügen.
Die nächste Aufrufstufe unterhalb der beschriebenen API unterscheidet lediglich zwischen dem Initialisierungsmode bzw. dem Decodierungsmode. Die Unterscheidung erfolgt über eine boolesche Variable im Funktionsaufruf.
Im Gegensatz zum Initialisierungsfall verbirgt sich hinter dem Decodierungsaufruf eine tiefe Funktionshierarchie aller für den Decodierungsprozess notwendigen Unterfunktionen. Dabei wird die Verarbeitung eines Bildes in drei Aufrufebenen unterteilt. Auf höchster Ebene sind alle Funktionen wiederzufinden, die für die Verarbeitung des Bildes auf VOP-Ebene notwendig sind. Dies sind z.B. Funktionen, die globale Adresszeiger der Bildbereiche initialisieren oder generelle Bildvorbereitungsfunktionen beinhalten, wie z.B. die Unterscheidung der Bildprädiktionstypen (siehe Flussdiagramm in Anhang F.2).
Das in Anhang F.3 dargestellte Flussdiagramm stellt die nächste Aufrufebene innerhalb des Decoders dar, die alle makroblockverarbeitenden Funktionen beinhaltet, wie unter anderem auch die komplette Bewegungskompensation eines Makroblockes.
Die nächst niedrigere Funktionsebene stellen die Blockverarbeitungsfunktionen dar, die im Wesentlichen aus den Dequantisierungsfunktionen und der IDCT gebildet werden.
Die niedrigste Funktionsebene bilden die Bitstromverarbeitungsroutinen, die neben den elementaren Verarbeitungsfunktionen sämtliche VLD-Routinen beinhalten.

4.7.1 Softwareoptimierungen

Neben der Verwendung der beschriebenen Hardwareerweiterungen kommen auch spezielle, ausschließlich auf die entwickelte Software bezogene Optimierungsmaßnahmen zum Tragen, um die erzielbare Echtzeitperformance zu steigern.
Basierend auf den in Kapitel 4.1 vorgestellten Laufzeituntersuchungen des unoptimierten Decoders wurden einzelne Funktionen auf ihre Optimierbarkeit untersucht.
Dabei konnte festgestellt werden, dass der verwendete Hochsprachencompiler, in der höchsten Optimierungsstufe betrieben, nahezu optimalen Code generiert. Dies ist in erster Linie auf den relativ einfachen Prozessorkern und den stark eingeschränkten Befehlsumfang zurückzuführen.
In wenigen Fällen konnte jedoch durch den gezielten Einsatz einfacher Assemblerfunktionen ein Performancegewinn im Verhältnis zu der funktional äquivalenten Hochsprachenimplementierung erzielt werden. Dies war in der Regel auf das Unvermögen des Compilers zurückzuführen, in diesen speziellen Fällen die vierstufige Verarbeitungspipeline des Prozessors vollständig zu füllen.
Weiterhin mussten die Speicherkopierfunktionen der Laufzeitbibliothek des ARC durch eigene Routinen ersetzt werden.
Weitere Optimierungen betrafen die Verwendung von Adresszeigern im C-Code. Durch die Referenzierung von Speicherbereichen über komplexe Pointer, die eine eigene Adressrechnung benötigen, kann teilweise eine hohe Rechenlast hervorgerufen werden.
Bei der Implementierung der Software wurde daher darauf geachtet, innerhalb von Funktionen die notwendige Pointerberechnung nur einmalig durchzuführen und die resultierende Adresse in einer lokalen Variablen zu speichern. Weitere Maßnahmen stellen Loop-Unrolling und Function-Inlining dar.
Zusammenfassend kann jedoch der erzielbare Rechenzeitgewinn durch reine Softwareoptimierungen mit weniger als 2% angegeben werden, der damit deutlich niedriger als der durch die vorgestellten Hardwareerweiterungen erzielte durchschnittliche Gewinn ist.

5 Analyse, Ergebnisse und Beispiele

Basierend auf der vorgestellten Simulationsumgebung (siehe Kapitel 4) sollen die architekturspezifischen Optimierungsansätze verifiziert werden. Dabei werden alle Verfahren einzeln bewertet und zu einem Gesamtergebnis zusammengefasst.
Um eine qualitative und quantitative Einordnung der entworfenen Architektur zu ermöglichen, wird ein Vergleich mit typischen Vertretern alternativer Prozessorarchitekturen vorgenommen. Abschnitt 5.5 stellt Anwendungsbeispiele der Architektur vor.

5.1 Quantitative Performanceanalyse

Nachfolgend wird ein Überblick über die erzielten Performanceeigenschaften des Decoders gegeben. Hierzu werden die einzelnen implementierten Konzepte mit Hilfe entsprechender Messergebnisse qualitativ und quantitativ bestätigt und in den nachfolgenden Abschnitten dargestellt und erläutert. Dabei wird das implementierte Verfahren jeweils einer reinen Softwareumsetzung gegenübergestellt und der erzielte Rechenzeitgewinn ermittelt. Der verwendete Softwaredecoder wurde mit der höchsten Optimierungsstufe übersetzt und stellt damit das erreichbare Maximum in Bezug auf Echtzeitperformance dar.
Da bei einigen Verfahren eine Abhängigkeit von der Änderung des Bildinhaltes und der Datenrate zu erwarten ist, wurden drei unterschiedliche MPEG-Testsequenzen (siehe Anhang E) mit fünf differierenden Datenraten codiert. Hierfür wurden jeweils gleiche Codierungsparameter gewählt. Wie z.B. I-Framerate, Framerate und Bildauflösung bzw. Anzahl codierter Makroblöcke.
Die Messungen wurden mit Hilfe des in Abschnitt 4.1.1 beschriebenen Hardware-Profilers durchgeführt und resultieren in einer taktzyklengenauen Angabe der benötigten Rechenzeit, die für jedes Bild einer Sequenz ermittelt wurde. Neben der Gegenüberstellung der durchschnittlich erzielten Taktzyklenreduktion wurde auch eine Auswertung des auf die Decodierung eines Bildes bezogenen Rechenzeitgewinns dargestellt, sofern dieser stark von den Durchschnittswerten abwich.

5.1.1 Interpolationsfilter

Die Befehle zur Implementierung der Interpolationsfilter *INTERPOLATE* und *ALIGN* werden nur im Falle von P-Blocktypen verwendet und weisen daher eine starke Abhängigkeit von der Charakteristik der codierten Sequenz in Bezug auf den erzielten Taktzyklengewinn auf. Der ermittelte durchschnittliche Gewinn liegt zwischen 5,7% und 30%.
Wie aus Abbildung 68 zu ersehen ist, kann ein hoher Gewinn bei Sequenzen mit niedrigen Datenraten und hohem Anteil an Bildänderung erzielt werden.
Da die Interpolationsfilter ausschließlich bei P-Blöcken mit ½-pixelgenauen Vektoren zur Anwendung kommen, ist hierbei auf einen großen Anteil entsprechender Blöcke im Verhältnis zu Pel-genauen Blockverschiebungen zu schließen.
Bei höheren Datenraten führt wiederum die größere Anzahl von intra-codierten Blöcken zu einem niedrigeren Rechenzeitgewinn, der dann aber immer noch bei mindestens 5% liegt.
Das Diagramm (Abbildung 68) verdeutlicht die aufgeführten Aussagen nochmals. Bei der Sequenz *Container* handelt es sich um Bildinhalte mit geringer zeitlicher und örtlicher Änderung, die sich gut mit der gegebenen Datenrate codieren lassen. Der Anteil bewegungskompensierter Bildblöcke ist demnach relativ gering. Ein hoher Anteil verwendeter Bild-

blöcke wird als sog. *not_coded* Block übertragen, womit fast vollständig auf die Einbeziehung eines Prädiktionsfehlersignals verzichtet wird.
Je höher die Datenrate gewählt wird, desto mehr Blöcke können daher direkt in Form intra-codierter Blöcke übertragen werden und erfordern somit ebenfalls keine Bewegungskompensation. Dieser Zusammenhang gilt ebenso für Sequenzen mit hoher Bildänderung, wie z.B. *Stefan* und *Foreman*, jedoch besteht hier ein im Verhältnis höherer Anteil prädiktiv codierter Bildblöcke.

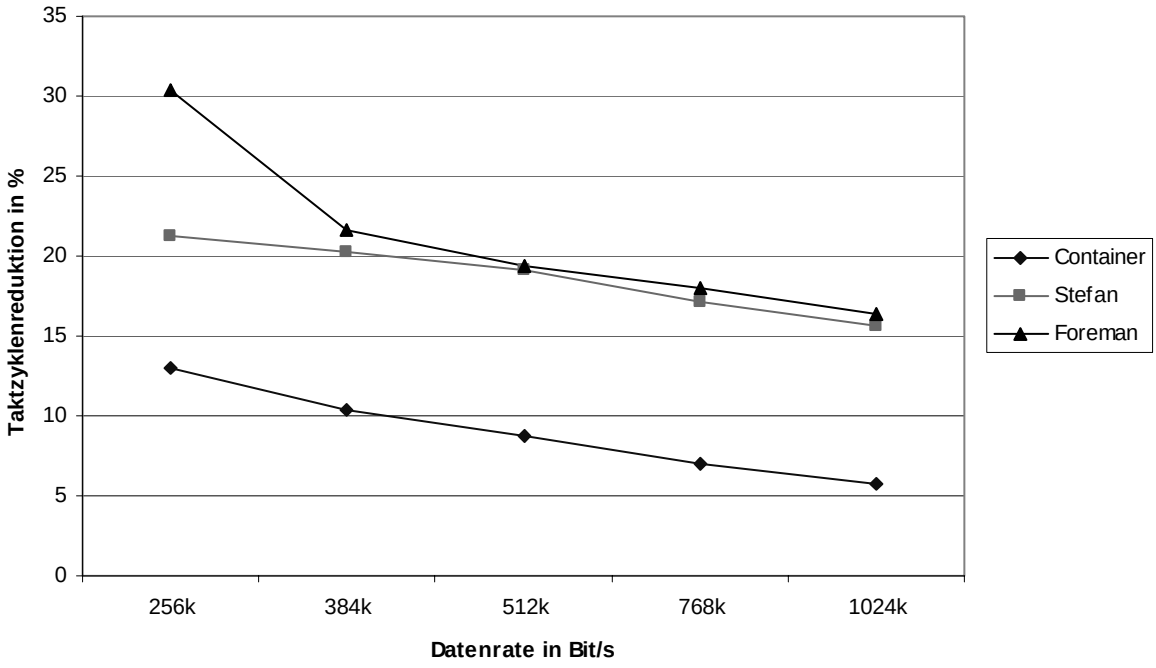


Abbildung 68 Taktzyklenreduktion mit Befehlssatzerweiterung für Interpolationsfilter

5.1.2 Summation mit Clipping

Ein nahezu proportionaler Zusammenhang zwischen Datenrate und Taktzyklenreduktion lässt sich für den Befehl *ADDCLIP* erkennen, wobei eine Abhängigkeit von der jeweiligen Charakteristik der codierten Bildsequenz (*Foreman*, starke Bildänderung) zu beobachten ist.
Das folgende Diagramm in Abbildung 69 stellt die hiermit durchschnittlich erzielte Taktzyklenreduktion dar, die zwischen 2% und 7% liegt.
Hierbei ist jedoch zu beachten, dass der Befehl ausschließlich bei der Decodierung von P-Bildtypen zur Anwendung gelangt und somit bei Sequenzen mit einer hohen Anzahl von P-Blocktypen besonders gute Resultate erzielt.
Die Zunahme der Taktzyklenreduktion bei steigender Datenrate ist auf eine hohe Anzahl codierter Blöcke zurückzuführen, die in Form eines Prädiktionsfehlersignals im Datenstrom übertragen werden. Mit steigender Datenrate steht dem Encoder eine entsprechend höhere Datenmenge pro Bildeinheit zur Verfügung, die somit zu einer höheren Zahl von DCT-Koeffizienten führt, die aufgrund hoher Quantisierungswerte nicht weggefallen sind.

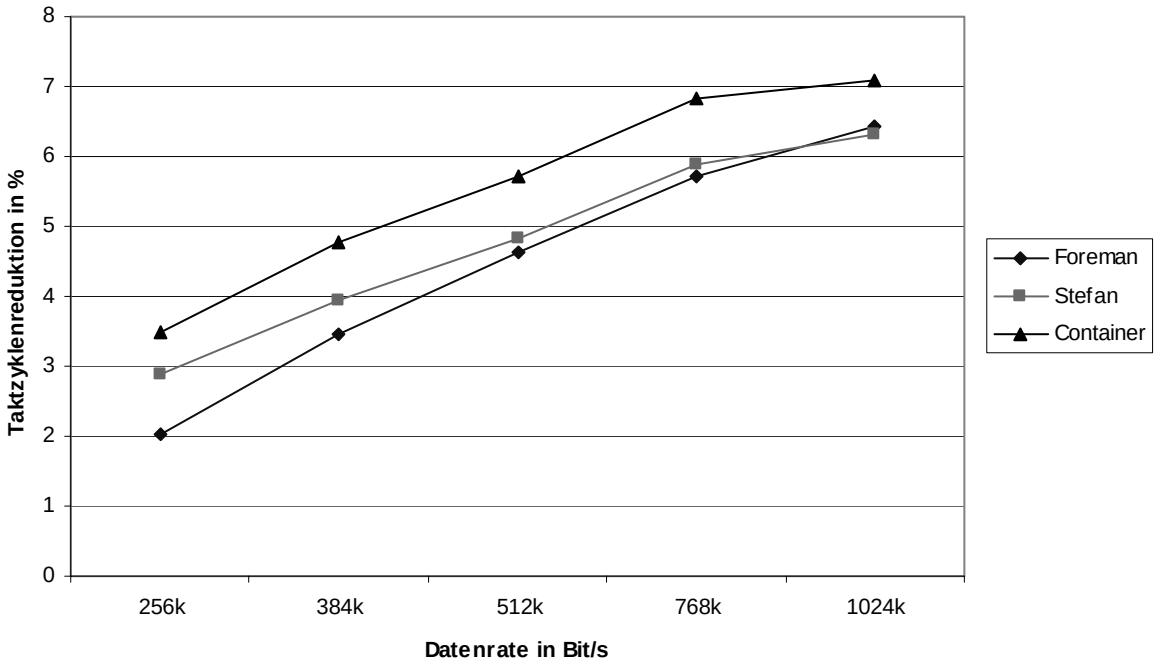


Abbildung 69 Taktzyklenreduktion mit Befehlssatzerweiterung für Pixelsummation

Abbildung 70 gibt exemplarisch den Zusammenhang zwischen Bildtyp und Taktzyklenreduktion wieder. In diesem Falle ist nur eine Sequenz (*Stefan*) in Bezug auf das zeitliche Verhalten dargestellt.
Deutlich ist zu erkennen, dass eine Taktzyklenreduktion nur für P-Bilder zu verzeichnen ist (Bildernummern 2..15, 17..30), wohingegen intra-codierte Bilder (1,16) keine Zyklenreduktion aufweisen.

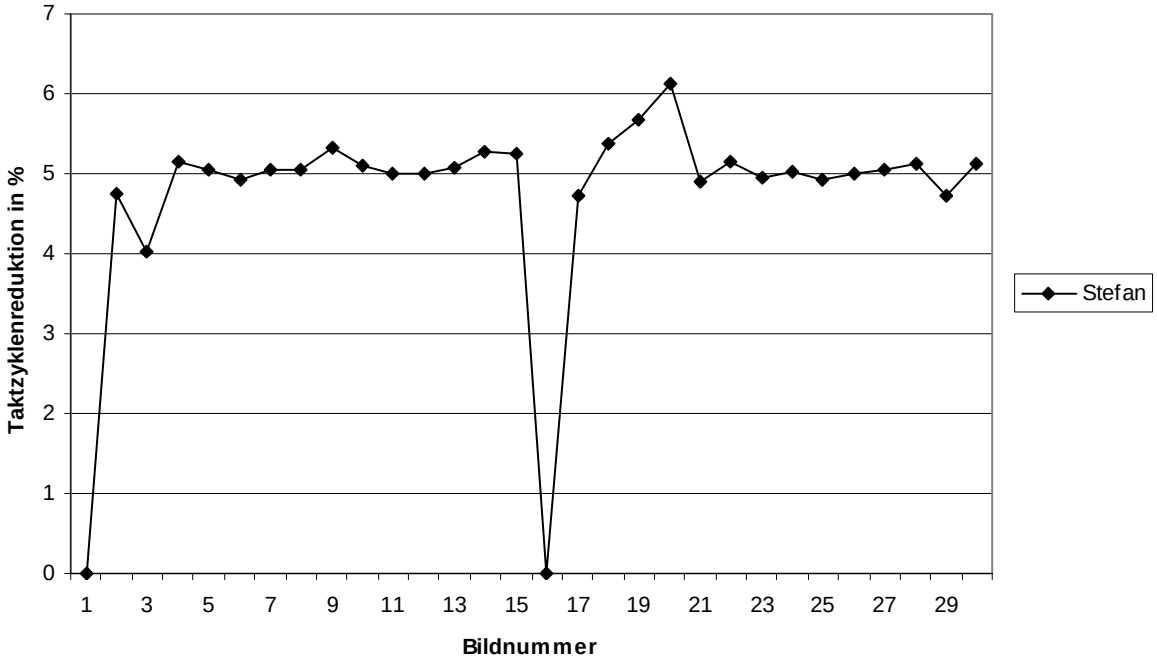


Abbildung 70 Taktzyklenreduktion pro Bild mit Befehlssatzerweiterung für Save/Clipping

5.1.3 Bitstromverarbeitung

Abbildung 71 gibt die prozentuale Taktzyklenreduktion für die in Abschnitt 4.4 beschriebenen Befehle zur Bitstromverarbeitung wieder. Hierbei wurden die benötigten Taktzyklen für die Decodierung der Sequenzen unter Verwendung der Befehle *GetBits*, *FlushBits* und *ShowBits* gemessen und der hierzu korrespondierenden Softwareimplementierung gegenübergestellt. Wie der Abbildung 71 zu entnehmen ist, verläuft die erreichbare Taktzyklenreduktion nahezu proportional zur Datenrate des verwendeten Datenstroms und liegt zwischen 1,8 % und 4,4 % für die untersuchten Datenraten bis 1024 kBit/s.
Die Charakteristik des jeweiligen Datenstroms bei gleicher Datenrate hat nur einen sehr geringen Einfluss auf die Taktzyklenreduktion und bewegt sich im Bereich von ca. 0,2 %. Aufgrund der hohen Datenratenschwankungen zwischen I- und P-Bildern und des linearen Zusammenhangs zwischen Datenrate und Zyklenreduktion kann der resultierende Gewinn in Bezug auf die Decodierung eines einzelnen I-Bildes einer Sequenz entsprechend höher ausfallen.

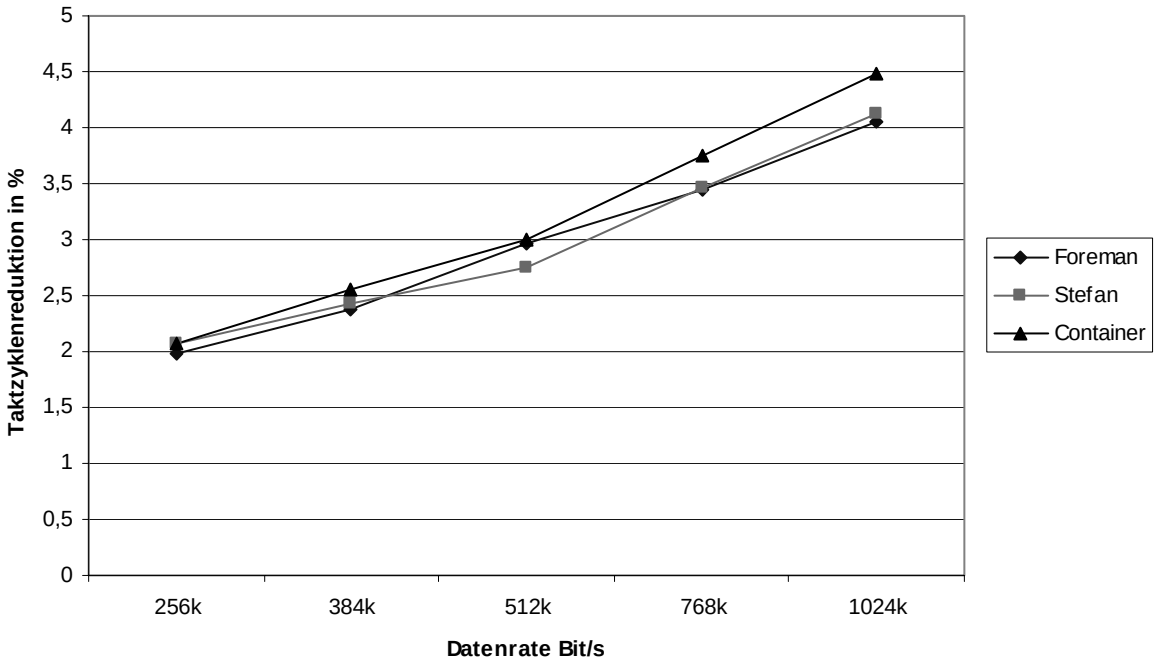


Abbildung 71 Prozentuale Taktzyklenreduktion bei Verwendung von Bitstromverarbeitungsbefehlen

Das nachfolgende Diagramm in Abbildung 72 gibt die prozentuale Taktzyklenreduktion pro Bild einer Sequenz (*Container*) bei 384 kBit/s wieder.
Wie hieraus hervorgeht, liegt die Taktzyklenreduktion für I-Bilder bei ca. 9% und kann damit aufgrund des proportionalen Verhältnisses zur Eingangsdatenrate auch unabhängig von der jeweiligen Bildsequenz angenommen werden, da I-Bilder nicht in der codierten Datenmenge durch die Merkmale der prädiktiven Codierung beeinflusst werden können.

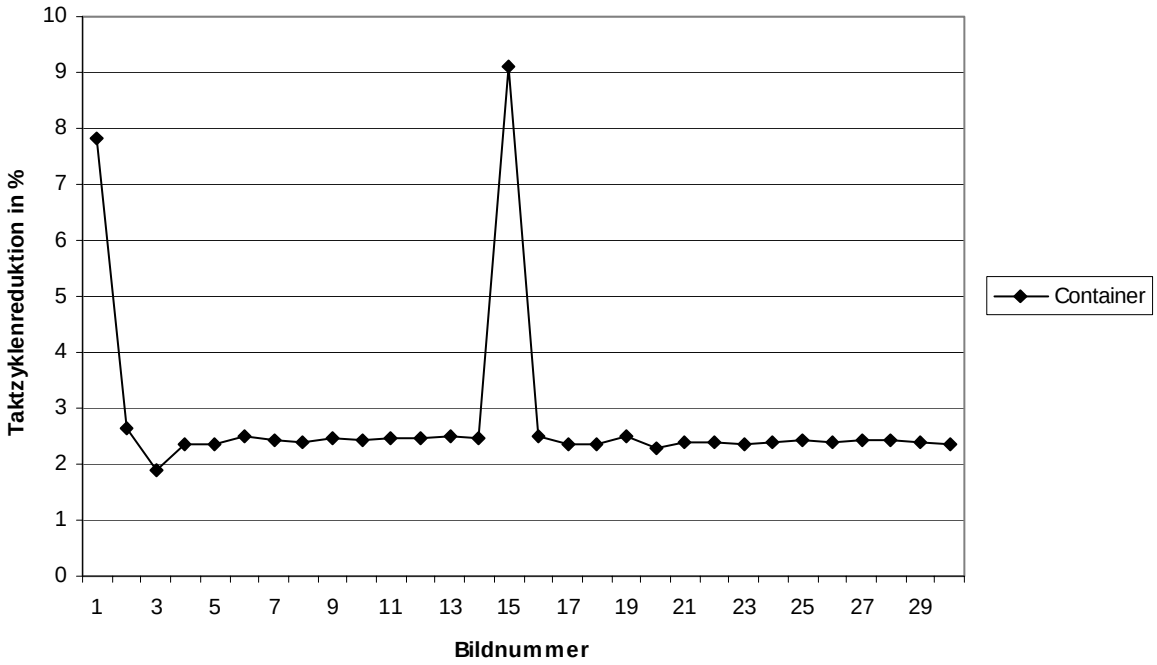


Abbildung 72 Prozentuale Taktzyklenreduktion pro Bild einer Sequenz unter Verwendung von Bitstromverarbeitungsbefehlen

Die umgesetzte Befehlssatzerweiterung führt somit zu einem durchschnittlich niedrigeren Gewinn, kann jedoch sehr effektiv für die Herabsetzung der Decodierungslatenzzeit eingesetzt werden, die durch Datenratenschwankungen in Abhängigkeit vom Bildprädiktionstyp hervorgerufen wird.

5.1.4 IDCT-Coprozessor

Der in Abbildung 73 dargestellte Rechenzeitgewinn für die Verwendung des IDCT-Coprozessors weist einen relativ hohen Wert auf, der zwischen 30% und 35% liegt. Dies ist auf die Tatsache zurückzuführen, dass die DCT eine der zentralen Codierungsfunktionalitäten für die übertragenden Texturdaten für I- wie auch P-Blocktypen darstellt. Hierbei wird immer von einer konstanten Anzahl von Transformationskoeffizienten pro codiertem Block ausgegangen, die unabhängig von der Charakteristik der Textur eine feste Rechenzeit für die Rücktransformation erfordert. Selbst im Falle einer sehr geringen Anzahl übertragender Prädiktionsfehlersignale wird bei einem Block, der z.B. nur über einen Transformationskoeffizienten verfügt, eine komplette 8x8 IDCT durchgeführt.
Bei Sequenzen mit geringer Bildänderung und hoher Datenrate fällt der Rechenzeitgewinn demnach höher aus, da weniger bewegungskompensierte Bilddaten und eine höhere Anzahl von I-Blöcken verwendet werden.
Diese Aussage lässt sich auch hier wiederum anhand der verwendeten Bildsequenzen belegen. Der Ausgangspunkt für alle drei untersuchten Sequenzen stellt eine Datenrate dar, bei der der Anteil DCT-codierter Blöcke für alle Sequenzen, aufgrund der begrenzt zur Verfügung stehenden Datenmenge pro Bild, gleich ist.
Mit steigender Datenrate kann im Falle von Sequenzen mit geringer Bildänderung die zur Verfügung stehende Datenmenge pro Bild für die vollständige intra-Codierung eines Blockes herangezogen werden, da der Anteil der benötigten Datenmenge für Bewegungsvektoren bzw.

Prädiktionsfehlersignale kleiner ausfällt. Für Sequenzen mit hoher Bildänderung stellt sich das Verhältnis in umgekehrter Weise zu Ungunsten intra-codierter Blöcke und zu Gunsten entsprechender P-Blöcke dar.

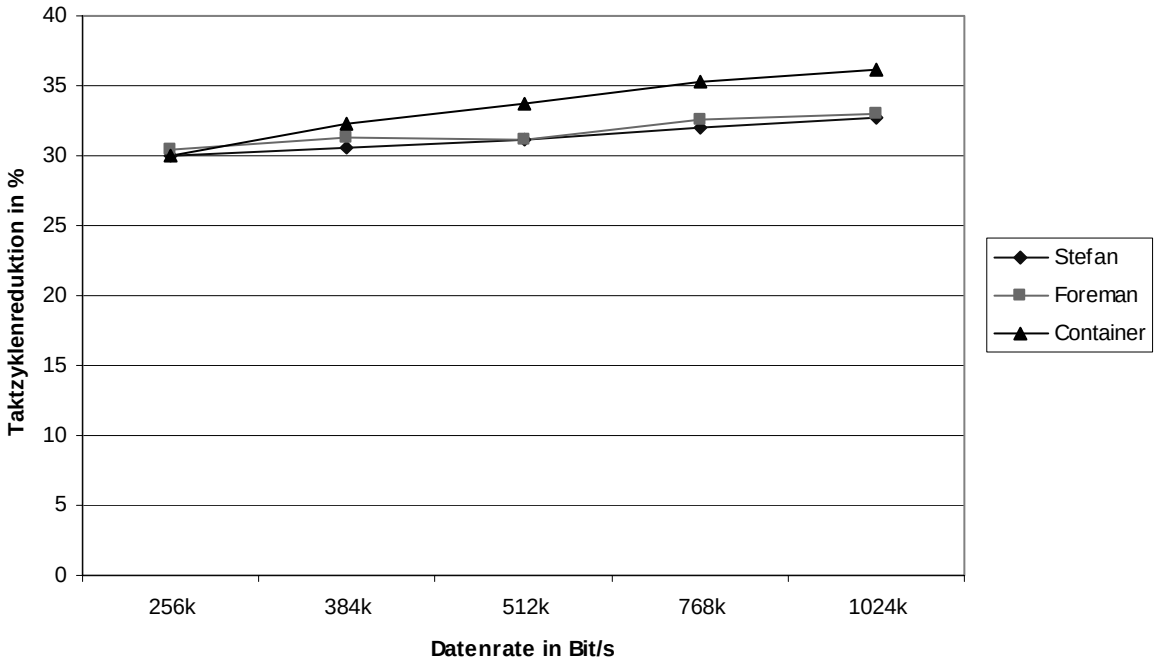


Abbildung 73 Taktzyklenreduktion mit dem IDCT-Coprozessor

5.1.5 Kombination aller Optimierungen

Wie aus den vorhergehenden Ausführungen zu entnehmen ist, führen alle implementierten Verfahren zu einem von der Datenrate und dem verwendeten Bildmaterial abhängigen Rechenzeitgewinn. Somit lässt sich jedes Verfahren einzeln einsetzen, ist jedoch damit stark von dem jeweiligen Einsatzumfeld abhängig.
Es können somit auf ein spezielles Anwendungsfeld zugeschnittene Umsetzungen vorgenommen werden, die nicht notwendigerweise sämtliche Verfahren in sich vereinen.
Da sich die erzielten Rechenzeitgewinne jedoch in ihrer Gesamtheit gegenseitig beeinflussen, ist es nahe liegend, eine Analyse der Kombination sämtlicher Verfahren durchzuführen.
In Abbildung 74 soll daher ein Überblick über den erzielten Rechenzeitgewinn bei gleichzeitiger Verwendung aller vorhergehend beschriebenen Optimierungsmaßnahmen im Verhältnis zu einer reinen Softwareimplementierung aufgezeigt werden.
Wie dem Diagramm entnommen werden kann, ist in diesem Falle die Taktzyklenreduktion nahezu konstant und unabhängig von der verwendeten Datenrate der jeweiligen Bildsequenz. Dies ist darauf zurückzuführen, dass die Befehlssatzerweiterungen für die Bitstromverarbeitung und Summation einen entgegengesetzten Rechenzeitgewinn zu den Befehlen für die Umsetzung der Interpolationsfilter hervorrufen. Die Befehlssatzerweiterungen führen somit in ihrer Gesamtheit schon zu einem nahezu von der Gesamtscharakteristik der verwendeten Datenströme unabhängigen Rechenzeitgewinn. Weiterhin ist zu beobachten, dass bei Bildsequenzen mit niedriger Bildänderung eine geringere Taktzyklenreduktion herbeigeführt wird, die auf eine reduzierte Anzahl verwendeter P-Blöcke zurückzuführen ist.

Insgesamt liegt der erzielte Rechenzeitgewinn zwischen 40% und 50% für die verwendeten Sequenzen unter Einbeziehung aller Optimierungsmaßnahmen.

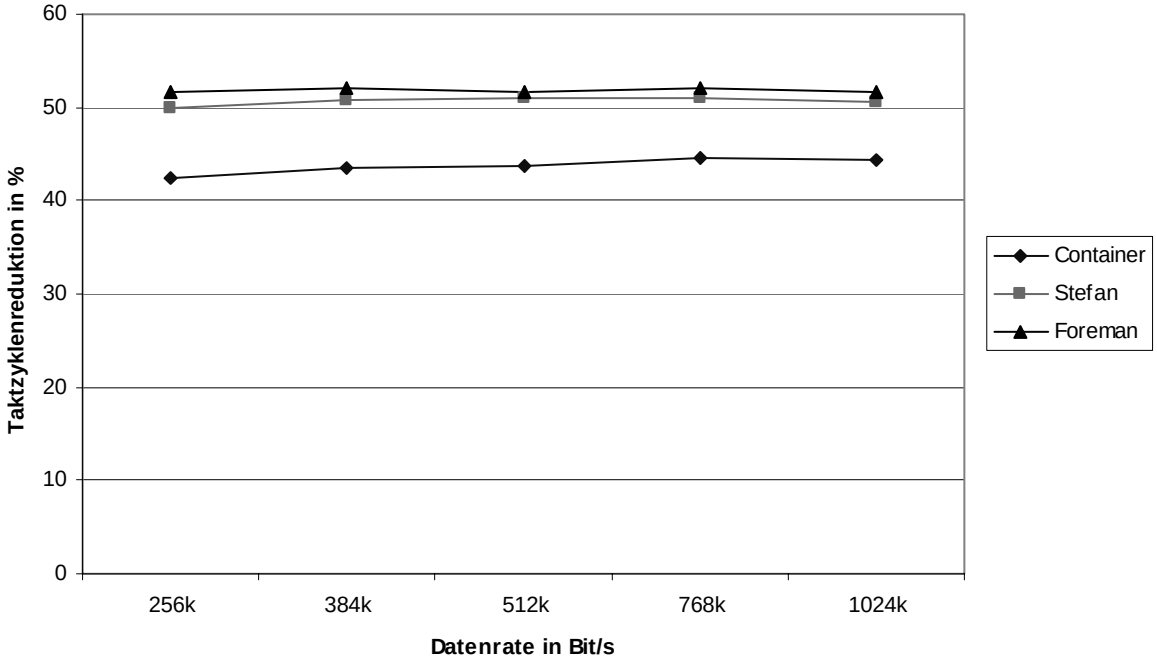


Abbildung 74 Taktzyklenreduktion unter Berücksichtigung aller Optimierungen

5.2 Syntheseergebnisse

Um eine Aussage über die Komplexität der Implementierung in Bezug auf eine Umsetzung für eine bestimmte Halbleitertechnologie zu erhalten, wurde die Architektur mit der am Heinrich-Hertz-Institut zur Verfügung stehenden ASIC-Bibliothek *CE81* der Firma Fujitsu synthetisiert. Es handelt sich hierbei um eine CMOS Embedded-Array-Technologie[23] mit einer Strukturbreite von 0,18 µm, die sich durch die Kombinationsmöglichkeit mit vordefinierten Makros (Mega-Cells), wie z.B. Speicherstrukturen und entsprechender Gate-Array-Strukturen, auszeichnet.

Die Technologie weist folgende Parameter auf:

- 0,18µm Silizium CMOS Gate-Array mit 3-5 Verdrahtungsebenen
- 1,1- 1,8 V Versorgungsspannung
- Gate Delay Time (t_{pd}) = 12 ps bei 1,8 V, Inverter bei F/O = 1
- $P_{dynamic}$ = 8nW/MHz/Gate bei 1,1V 2-NAND-Gate, F/O =1 .

Weitere Informationen über die eingesetzte Technologie können dem Datenblatt des Herstellers Fujitsu ([71]) entnommen werden.
Die Synthese der VHDL-RTL-Sourcen erfolgte mit dem Synthesewerkzeug *Design Compiler DC* der Firma Synopsys, wobei ausschließlich der eigentliche Prozessorkern und die Logikeinheiten des Coprozessors synthetisiert wurden. Die erforderlichen Speicherkomponenten für Prozessorcache und Triple-Port-Register-File wurden nicht berücksichtigt, da hierfür spezielle

[23] Embedded Array auch Structured Gate Array

Makros Verwendung finden. Die Optimierung des Systems wurde nach Fläche, d.h. Anzahl verwendeter Zellen, vorgenommen. Das Design wurde nicht hierarchisch synthetisiert, sondern auf eine sog. Flat-Netzliste zurückgeführt.
Als Basissystem wurde ein ARC-Core inkl. Barrelshifter, Multiplizierer und aller XALU-Erweiterungen gewählt. Die Größe der Einzelkomponenten wurde durch inkrementelles Synthetisieren des Gesamtsystems ermittelt. Für die Bestimmung der resultierenden Gatterzahlen wurde ein NAND-Gatteräquivalent mit 40 CE81-Zellen angenommen und auf die einzelnen Zellenanzahlen umgerechnet.
Die Größe des IDCT-Coprozessors wurde eigenständig ermittelt, da er nicht fester Bestandteil des ARC-Cores ist, sondern als lose gekoppelte Komponente in Form einer separaten Einheit zu betrachten ist. Die Komplexität wurde jedoch dem Prozessorkern gegenübergestellt, da er im Falle des Nichtvorhandenseins des Coprozessors für die Berechnung der IDCT zur Verfügung stehen müsste. Als oberste Taktfrequenz wurden ca. 120 MHz ermittelt, die durch verschiedene Simulationen bestätigt werden konnten.
Das folgende Diagramm in Abbildung 75 gibt einen Überblick über die Komplexität der synthetisierten Komponenten des Systems in Bezug auf ihren Flächenbedarf bei Verwendung aller Erweiterungen.

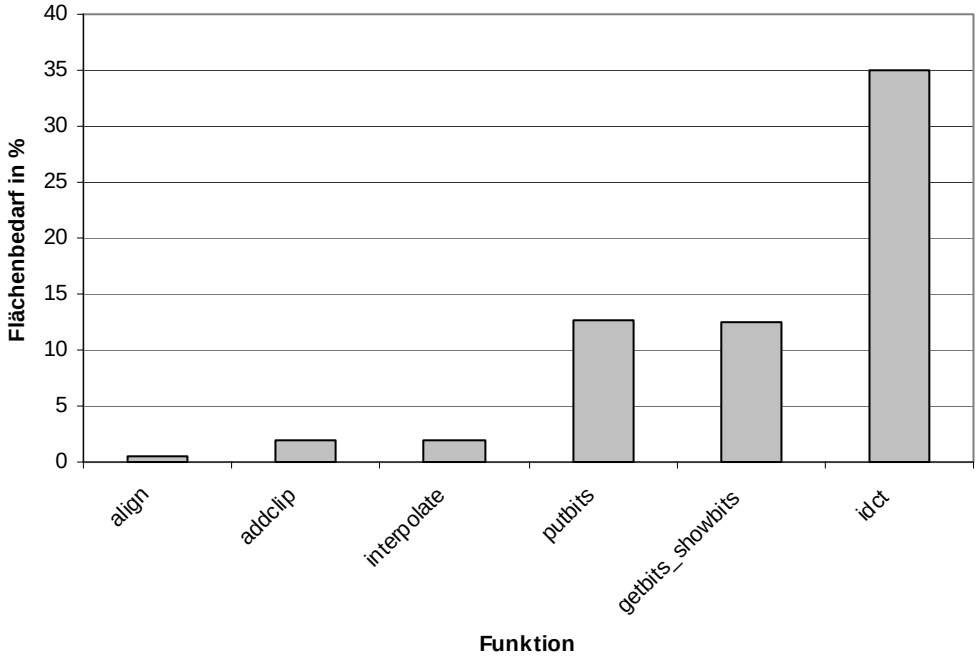


Abbildung 75 Vergleich des Flächenbedarfs einzelner Erweiterungen im Verhältnis zum Gesamtflächenbedarf

Die IDCT stellt die Komponente mit dem höchsten Flächenbedarf dar, gefolgt von den Befehlssatzerweiterungen zur Bitstromverarbeitung und den eigentlichen Bildsignalverarbeitungsbefehlen. Neben der Betrachtung des Flächenbedarfs ist die Gegenüberstellung der erzielten Leistungsparameter der einzelnen Optimierungsverfahren im Verhältnis zum benötigten Flächenbedarf von besonderem Interesse, da hiervon das Kosten-/Nutzen-Verhältnis abgeleitet werden kann.
Eine entsprechende Gegenüberstellung ist der folgenden Tabelle 4 zu entnehmen. Im Gegensatz zum vorhergehend dargestellten Diagramm wird hier der Flächenbedarf einer einzelnen

Komponente im Verhältnis zum Flächenbedarf des Prozessors aufgezeigt. Somit ist ein direkter Vergleich mit den in Abschnitt 5.1 gewonnenen Ergebnissen möglich.
Hierbei ist deutlich zu erkennen, dass die in Form einer dedizierten Komponente umgesetzte IDCT den größten Rechenzeitgewinn erzielt. Dabei liegt jedoch der benötigte Flächenbedarf bei 94% des Flächenbedarfs des reinen Prozessorkerns.
Weiterhin kann festgestellt werden, dass die in Form eines Befehls umgesetzten Optimierungen mit hohem Flächenbedarf einen vergleichsweise niedrigen Rechenzeitgewinn hervorrufen. Dies ist in erster Linie darauf zurückzuführen, dass hierbei keine Parallelität auf Befehlsebene realisiert werden konnte, wie z.B. im Falle der Bitstromverarbeitungsbefehle.
Ein Gegenbeispiel hierzu stellen die Befehle zur Pixelinterpolation dar, wo ein überdurchschnittlich hoher Gewinn mit einem sehr geringen Flächenbedarf erzielt werden konnte, was der konsequenten Umsetzung des SIMD-Konzeptes zuzuschreiben ist.

	Flächen-bedarf in %	Anzahl CE81 Cells	Anzahl Gates	Performance-gewinn in % Min.	Performance-gewinn in % Max.
ARC Core inkl. XALU		1808120	45203		
Bitstreampro-cessing	25,25	456560	11414	1,8	4,4
ALIGN	0,52	9480	237		
INTERPOLATE	1,95	35360	884	5,7	30
ADDCLIP	1,90	34360	859	1,8	7,0
IDCT	94,78	1713760	42844	30,0	35,0

Tabelle 4 Übersicht des Flächenbedarfs und Performancegewinns einzelner Erweiterungen

5.3 Verlustleistungsbetrachtung

Auf der Grundlage der im vorhergehenden Abschnitt 5.2 vorgestellten Syntheseergebnisse kann die zu erwartende Verlustleistung des Prozessorkerns und der Erweiterungen ermittelt werden. Hierzu dienen die Angaben der jeweiligen Anzahl verwendeter CMOS-Gates und die dynamische Verlustleistung $P_{dynamic}$ pro Gate, die von Fujitsu wie folgt angegeben wird:

$$P_{dynamic} = 8nW/MHz/Gate \text{ bei } 1{,}1V \text{ 2-NAND-Gate, F/O} = 1 \ .$$

Die nachfolgend dargestellte Tabelle 5 gibt einen Überblick über die zu erwartende Verlustleistung für die einzelnen Komponenten des Systems. Die Angaben sind dabei auf mW/MHz normiert, d.h., um den tatsächlichen Wert zu ermitteln, muss die Taktfrequenz der jeweiligen Implementierung als Faktor berücksichtigt werden.

	ARC Core inkl. XALU	Bitstream-processing	ALIGN	INTER-POLATE	ADDCLIP	IDCT	∑
Anzahl Gates	45203	11414	237	884	859	42844	101441
P_d / MHz in mW	0,36	0,09	0,002	0,007	0,007	0,34	0,81

Tabelle 5 Übersicht der Verlustleistungen einzelner Erweiterungen

Unberücksichtigt bleiben hierbei die Einflüsse der restlichen Komponenten einer vollständigen SOC-Umsetzung, wie z.B. Peripherie und Speicher. Weiterhin ist in der Regel von Systemen auszugehen, die über mehrere Versorgungsspannungen verfügen bzw. in der Lage sind, die Versorgungsspannung der benötigten Taktfrequenz des Prozessors anzupassen (siehe hierzu auch Abschnitt 5.5.1).

5.4 Vergleich mit anderen Prozessorarchitekturen

Neben der quantitativen Analyse der implementierten Prozessorerweiterungen ist der Vergleich mit anderen etablierten Prozessorarchitekturen von Interesse. Dabei sollten typische Vertreter der in Abschnitt 2.4 vorgestellten Architekturen für einen Vergleich herangezogen werden. Im Wesentlichen kommen z. Zt. hierfür die Intel X86 Prozessorarchitektur und die Mediaprozessoren Philips Trimedia TM1300 und TI TMS320C6x in Frage.
Ein Vergleich mit den in Abschnitt 2.4.4 beschriebenen Signalprozessoren schied aufgrund der Nichtverfügbarkeit entsprechend leistungsfähiger Prozessoren aus. Der Vergleich mit anderen RISC-Prozessoren wurde ebenfalls nicht im Rahmen diese Arbeit vorgenommen, da die am Markt verfügbaren Prozessoren eine sehr ähnliche Architektur, wie die des hier eingesetzten Prozessors, aufweisen und somit zu fast identischen Ergebnissen führen würden.
Aufgrund der sehr unterschiedlichen Prozessormerkmale in Bezug auf Prozessortakt und Systemeinbindung wurde eine taktzyklengenaue Ermittlung der benötigten Rechenleistung angestrebt, um somit einen taktfrequenzbereinigten Vergleich mit der entwickelten Architektur zu gewährleisten. Dabei wurden prozessoreigene Taktzyklenzähler bzw. zyklengenaue Softwaresimulatoren des jeweiligen Prozessors verwendet.
Im Gegensatz zur quantitativen Analyse der Architektur in Abschnitt 5.1 wurden für die nachfolgend beschriebenen Vergleiche praxisbezogene Parameter bei der Wahl der für die Untersuchung eingesetzten Datenströme verwendet. Hierfür wurden Datenströme codiert, die den in Tabelle 1 wiedergegebenen Konformitätskriterien für MPEG-4 Profiles und MPEG-4 Levels entsprechen.
Um wiederum die unterschiedlichen Charakteristika in Bezug auf Bildänderungen zu berücksichtigen, wurden für jedes untersuchte Level die drei Sequenzen *Container*, *Stefan* und *Foreman* (siehe Anhang E) verwendet. Aus den Einzelergebnissen wurde der durchschnittliche Taktzyklenbedarf für die Decodierung eines Bildes ermittelt und in einem Diagramm dargestellt.
Ein Überblick über die Systemparameter aller untersuchten Architekturen findet sich in folgender Tabelle 6.

	TMS320C6201	TM1300	Motorola Power PC G4 MPC7450	Intel Pentium IV	Intel Pentium III
Prozessor-takt in MHz	200	166	667	2423	700
Cache	64kB L1 Inst.	32kB L1 Inst. 16 kB L1 Data	32 kB L1 Inst, 32 kB L1 Data, 256 kB L2	8kB L1 Data Execution Trace 12k Words 512 kB L2	16 kB L1 Inst, 16 kB L1 Data, 256 kB L2
Speicher-system	SRAM 200 MHz	SDRAM 133 MHz	SDRAM 133 MHz	DDRAM 533 MHz	SDRAM 100 MHz
Compiler	TI Code Com-poser Studio V2.1	Philips tmcc V5.6.8	gcc (GCC) 3.1 20020420 (pre-release)	Intel C++ 7.0	Intel C++ 7.0
Optimie-rungsstufe	O3	O5	O3	O2	O2
Betriebs-system	-	-	Darwin Kernel Version 6.5	Windows 2000 Professional V 5.0.2195 Ser-vice Pack 3	Windows 2000 Professional V 5.0.2195 Ser-vice Pack 3
Messme-thode	Zyklenzähler / Simulator	Zyklenzähler / Hardware	Zyklenzähler / Hardware	Zyklenzähler / Hardware	Zyklenzähler / Hardware

Tabelle 6 Systemparameter untersuchter Prozessoren

5.4.1 Superskalare Architekturen

Eine der am häufigsten anzutreffenden prozessorbasierten Architekturen für die Decodierung und Wiedergabe von Multimediadaten stellen die Intel X86-basierten PC-Systeme dar. Trotz der hohen Taktfrequenzen, die diese Prozessoren in der Regel aufweisen, wurde mit Hilfe der in Kapitel 2.4.5 beschriebenen SIMD-Konzepte für eine möglichst effiziente Verarbeitung entsprechender Algorithmen gesorgt. Der Vergleich beschränkt sich daher nicht nur auf die Verwendung des softwarebasierten Decoders unter Verwendung unterschiedlicher Plattformen, sondern es wurden auch verfügbare SIMD-Befehle verwendet, um optimierte Algorithmen umzusetzen und zu vergleichen.
Im vorliegenden Falle wurden die in Anhang A.5 wiedergegebene MMX-Implementierung der IDCT und das MMX-optimierte Interpolationsfilter verwendet, um den sonst zum ARC-Decoder vollständig identischen Code des Decoders zu optimieren.
Der Code wurde darüber hinaus mit dem Intel Compiler V7.0 mit der höchsten Optimierungsstufe übersetzt. Die Ermittlung der benötigten Taktzyklen wurde mit Hilfe des Intel Pentium-Prozessorbefehls *RDTSC* durchgeführt, der den prozessoreigenen Zyklenzähler ausliest und in einem Zielregister zur Verfügung stellt (siehe hierzu auch [74]).
Um betriebssystembedingte Seiteneffekte bei der Ausführung des Decoders unter Windows2000 zu minimieren, wurde die Priorität des Decoderthreads mit Hilfe der Funktionen *SetPriorityClass* und *SetThreadPriority* auf die Werte *REALTIME_PRIORITY_CLASS* und

THREAD_PRIORITY_TIME_CRITICAL gesetzt. Die Messungen wurden mehrmals wiederholt, um ein Höchstmaß an Genauigkeit zu erzielen.
Abbildung 76 gibt die ermittelten Ergebnisse der Messungen für Intel Pentium IV- und Intel Pentium III-Prozessoren im Vergleich zur vorliegenden Architektur wieder.

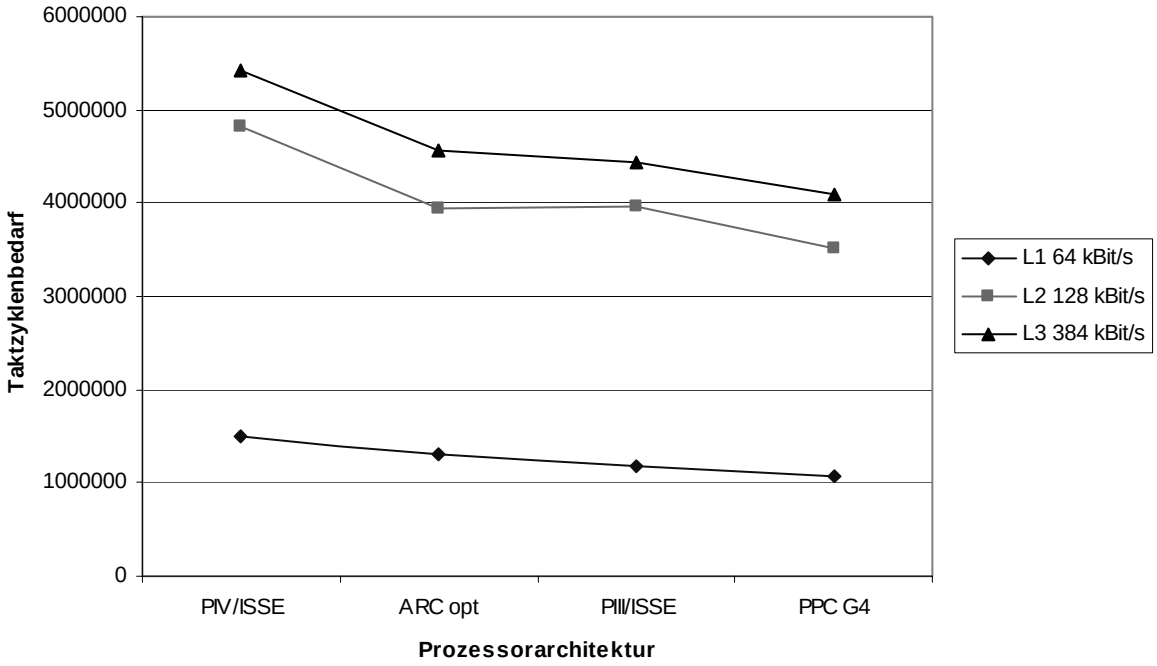


Abbildung 76 Vergleich des durchschnittlichen Taktzyklenbedarfs für PC-Prozessorarchitekturen mit der entworfenen Architektur

Neben den X86-basierten Prozessoren wurde ebenfalls die von Motorola und IBM entwickelte PowerPC-Architektur in Form des PPC7450 (PowerPC G4) untersucht, für die jedoch kein speziell optimierter Code verwendet wurde.
Bezogen auf den reinen Taktzyklenbedarf kann zusammenfassend festgestellt werden, dass die im Rahmen der Arbeit entworfene Architektur gleiche und bessere Ergebnisse erzielt als die untersuchten, wesentlich komplexeren, Superskalaren Prozessoren. Dies ist jedoch im Wesentlichen auf das Vorhandensein eines dedizierten IDCT-Coprozessors zurückzuführen, dem eine MMX-Softwareumsetzung gegenübersteht.
Der durchschnittlich höhere Taktzyklenbedarf im Falle des Intel Pentium IV-Prozessors lässt sich auf die im Verhältnis zum Intel Pentium III größere Pipelinetiefe zurückführen, die sich auf die Verarbeitung irregulärer Softwarestrukturen, wie z.B. die Bitstromverarbeitung, nachteilig auswirkt. Diese These lässt sich mittels der Untersuchung des prozentualen Rechenzeitgewinns durch den Einsatz MMX-optimierten Codes belegen.
Abbildung 77 gibt einen Überblick über die erzielten Rechenzeitgewinne auf den unterschiedlichen Zielprozessoren wieder, bei denen die prozentuale Taktzyklenreduktion im Falle des Pentium IV am höchsten ausfällt.
Für den Vergleich wurde zusätzlich noch die X86-kompatible AMD-K6-Architektur herangezogen, die über eine, dem Pentium III ähnliche, Befehlspipeline und eine MMX-kompatible SIMD-Befehlssatzerweiterung verfügt.

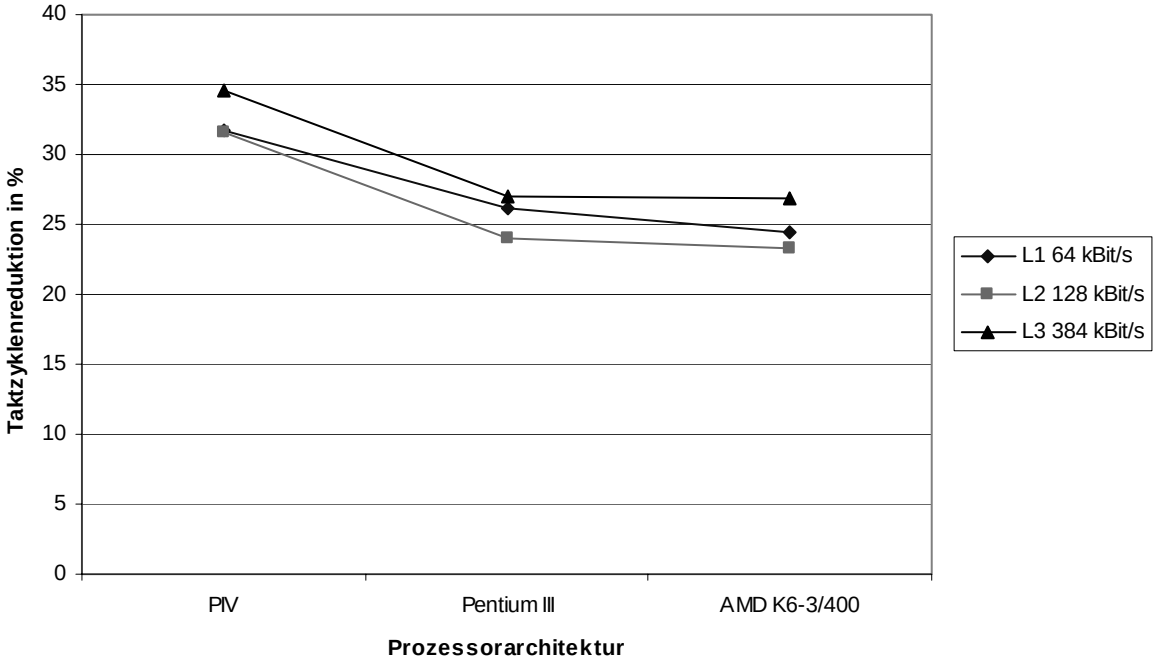


Abbildung 77 Erzielbare Taktzyklenreduktion durch den Einsatz spezieller SIMD-Befehle

5.4.2 VLIW / Mediaprozessor

Die zweite wichtige Architektur für die Umsetzung prozessorbasierter Decoder stellen die in Abschnitt 2.4.7 beschriebenen VLIW-basierten Mediaprozessoren dar.
Da hier die Auswahl an verfügbaren Prozessoren z. Zt. beschränkt ist, wurden für die vergleichenden Tests zwei Prozessoren mit dem größten Verbreitungsgrad ausgewählt. Die Messungen wurden unter Einsatz eines mit 166 MHz getaktetem Trimedia TM1300 und eines Prozessorsimulators für den von TI hergestellten TMS320C6201 durchgeführt.
Auf beiden Prozessoren wurde ausschließlich der Decoder ausgeführt, sodass es zu keinerlei Verfälschungen der Echtzeitperformance durch ein Betriebssystem kommen konnte. Für beide Prozessoren standen keine prozessorspezifischen Assembleroptimierungen zur Verfügung, womit die Messungen sich ausschließlich auf den compileroptimierten C-Code beziehen. Auch hierbei wurde die jeweils höchste wählbare Optimierungsstufe für die Übersetzung des Codes gewählt. Die Ermittlung der Taktzyklen wurde im Falle des TM1300 mit Hilfe des prozessorinternen 64 Bit-Zyklenzählers *CCOUNT* vorgenommen, dessen Zählerstand mit den Befehlen *CYCLES* und *HIGHCYLCES* ausgelesen werden kann. Im Falle des TMS320C6201 wurde der Zyklenzähler des Softwaresimulators verwendet.
Hierbei zeigt sich, dass die vorgestellte Architektur nahezu gleichwertige Ergebnisse im Vergleich zum TM1300 liefert, wobei allerdings mit in Betracht gezogen werden muss, dass die Codebasis des hier verwendeten Decoders nicht über assemblergestützte Optimierungsmaßnahmen verfügt.
Gestützt auf den Erfahrungen, die mit Assembleroptimierungen auf der Basis der SIMD-Befehle für X86-Prozessoren gemacht wurden, kann für die Trimedia-Architektur von einem Optimierungspotential von ca. 20% ausgegangen werden.
Die höheren Zyklenzahlen des TMS320C6201-Prozessors sind auf die schlechteren Ergebnisse des optimierenden Compilers zurückzuführen. Der Assemblerquelltext des übersetzten Decoders zeigte hierbei stellenweise einen sehr niedrigen Grad an Parallelität, obwohl durch-

aus ein hohes Optimierungspotential durch den C-Quelltext gegeben war. Da sich beide VLIW-Architekturen ähneln, ist mit einer ähnlichen Performance für den TMS320C6201 zu rechnen. Abbildung 78 gibt die Ergebnisse der Messungen wieder.

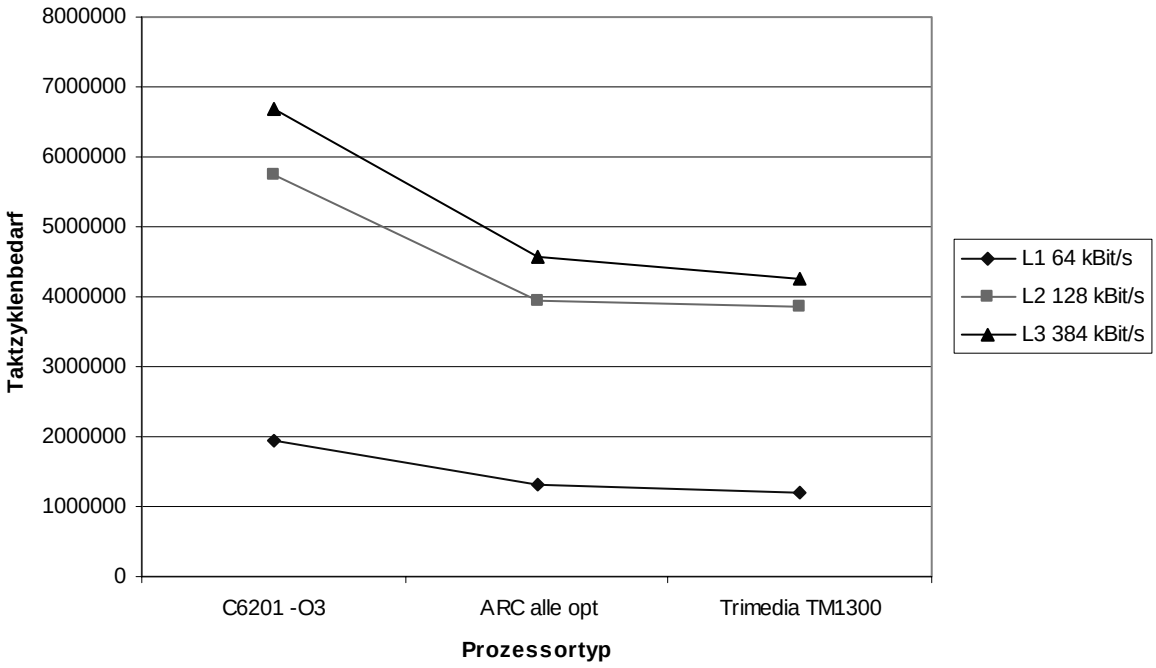


Abbildung 78 Vergleich des durchschnittlichen Taktzyklenbedarfs für VLIW-Architekturen mit der entworfenen Architektur

5.5 Anwendungsbeispiele

Aus der ermittelten Leistungsfähigkeit der Architektur lassen sich die Parameter für die Umsetzung prozessorbasierter Decoder für spezielle Anwendungen ableiten. In erster Linie können die benötigte Speichergröße und der benötigte Systemtakt ermittelt werden.
Tabelle 7 gibt die Systemparameter eines auf der vorgestellten Architektur aufbauenden Decoderkerns für die jeweilige Profile- und Level-Kombinationen wieder. Die angegebenen Taktraten wurden aus den Untersuchungen der benötigten Taktzyklen pro Bild ermittelt und stellen den jeweiligen erforderlichen Spitzenwert dar.
Der Speicherbedarf wurde aus den Anforderungen der Software ermittelt, wobei für die Speicherzugriffszeit der den Untersuchungen zugrunde gelegte Speicher mit null Waitstates angenommen wurde.

	Systemtakt	Speicher-bedarf kByte	Speicher-zugriffszeit	Bildgröße typisch	Makroblöcke /s	P_d Core
SP@L1	ca. 25 MHz	113603 Byte	40 ns	176 x 144	1485	20 mW
SP@L2	ca. 100 MHz	453447 Byte	10 ns	352 x 288	5940	80 mW
SP@L3	ca. 200 MHz	453447 Byte	5 ns	352 x 288	11880	160 mW

Tabelle 7 Übersicht der Systemparameter Profile-/Level-konformer Decoder

Die Angabe der dynamischen Verlustleistung P_d ergibt sich aus den in Abschnitt 5.3 ermittelten Parametern und dem errechneten maximalen Systemtakt.
Wie schon erwähnt, stellen diese Werte Spitzenwerte dar, die die höchste auftretende Rechenlast widerspiegeln. Durch geeignete Maßnahmen können diese Spitzenwerte jedoch noch herabgesetzt werden, indem bestimmte Einschränkungen in Bezug auf das Echtzeitverhalten gemacht werden.
Als Beispiel soll hierfür MMS[24] als eine der Hauptanwendungen der Videodatenübertragung zukünftiger Mobilfunkdienste genannt werden. Hierbei werden Videodaten nicht unter Echtzeitbedingungen, sondern in Form eines Dateitransfers dem Empfänger übersandt. Die Darstellung erfolgt erst nach abgeschlossener Übertragung. Die harte Echtzeitbedingung, die Decodierung eines Bildes sofort vornehmen zu müssen und innerhalb eines Zeitintervalls abgeschlossen zu haben, um nicht einen Datenverlust herbeizuführen, bestehen hierbei nicht. Aufgrund dieser Bedingungen kann die Decodierungslatenzzeit dahingehend verändert werden, dass die Spitzenrechenlast auf zwei Zeiteinheiten verteilt wird, indem für die Ausgabe des Bildes ein entsprechender Puffer zur Verfügung gestellt wird. Die notwendige Grundlage für dieses Prinzip stellt die stark unterschiedliche Decodierungszeit für die Bildtypen I und P dar. In der Regel folgt auf ein I-Bild mit der daraus resultierenden höheren Decodierungszeit ein P-Bild, das eine niedrigere Decodierungszeit benötigt, als für das jeweilige Zeitintervall zur Verfügung steht.
Das beschriebene Konzept kann allerdings nur in begrenztem Maße angewandt werden, da mit zunehmender Latenz die verwendeten Pufferspeicher jeweils entsprechend vergrößert werden müssen. Dies gilt besonders für die MPEG-4 Simple Profile Level 2 und 3, wo eine im Verhältnis zu Level 1 viermal größere Bildauflösung verwendet werden kann.
Dem Einsatz der Architektur für höhere Bildauflösungen sind aufgrund der Systemparameter technologische Grenzen gesetzt. Geht man von der linearen Skalierbarkeit des Systems aus, so sind bei einer Bildauflösung von 640 x 480 Pixeln und einer Bildwiederholfrequenz von 30 Hz bzw. 47520 Makroblöcken/s ein Systemtakt von 800 MHz und eine Speicherzugriffszeit von 1,25ns zugrunde zu legen.
Ein weiterer Nachteil stellt die Speicherzugriffscharakteristik dar, die den Einsatz von SDRAM-Speicherarchitekturen erschwert. Da jedoch auch der Speicherbedarf mit der Bildauflösung steigt, ist der Einsatz des hier verwendeten statischen Speichers aufgrund des Flächenbedarfs nahezu ausgeschlossen.
Prozessorbasierte Implementierungen, wie in der vorliegenden Arbeit behandelt, werden daher vorläufig nicht für die Umsetzung von Decodern mit höherer Bildauflösung in Frage kommen.

5.5.1 Verfahren zur verlustleistungsoptimierten Decodierung

Wie schon im vorhergehenden Abschnitt angedeutet wurde, kann die unterschiedlich benötigte Rechenzeit pro Bild für Realzeitoptimierungsmaßnahmen genutzt werden. Problematisch stellt sich dabei die Vorhersage der benötigten Rechenzeit pro Bildeinheit dar, um zu jedem Zeitpunkt die geforderten Echtzeitparameter einhalten zu können.
Das nachfolgende Diagramm in Abbildung 79 stellt die maximal notwendige Obergrenze der Taktfrequenz eines Prozessorsystems in Bezug auf die Decodierung eines Bildes anhand des zeitlichen Verlaufs dar.
Wie hieraus deutlich zu ersehen ist, können starke Schwankungen in der benötigten Prozessortaktfrequenz, im vorliegenden Falle zwischen 17 MHz und 32 MHz, auftreten.

[24] MMS – Multi Media Messaging

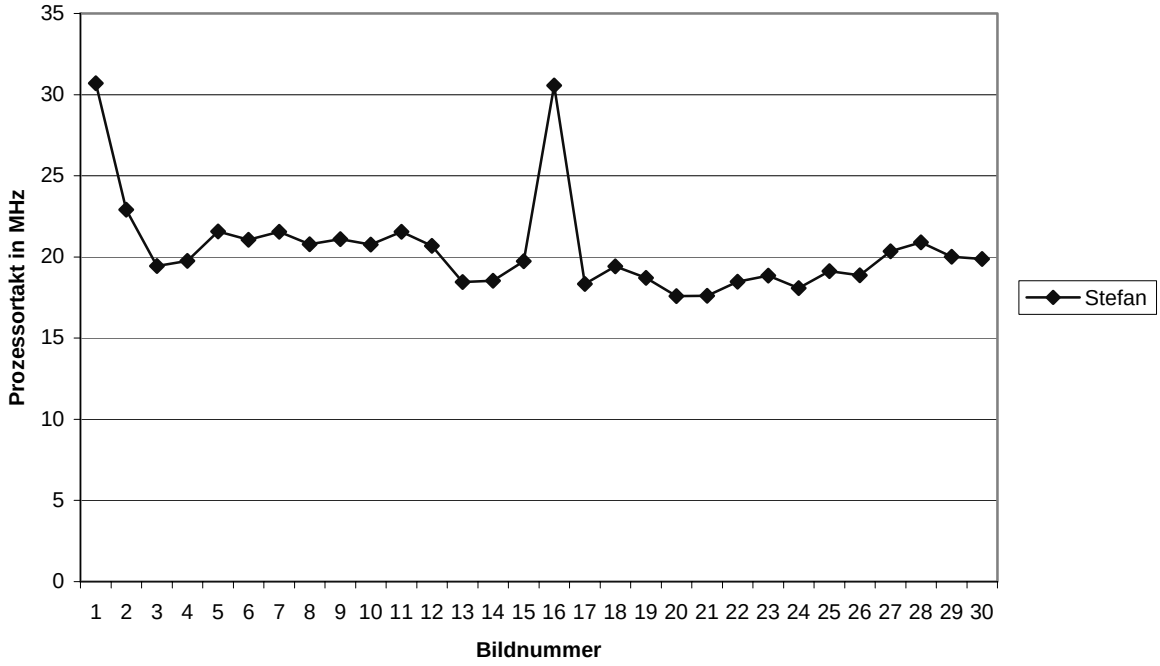


Abbildung 79 Maximal benötigter Prozessortakt in MHz pro Bild

Dementsprechend ist auch die benötigte Verlustleistung für jedes Bild different unter der Voraussetzung, dass der Prozessortakt an die jeweils benötigte Obergrenze angepasst werden kann, da nach Gleichung (9) ein linearer Zusammenhang für CMOS-Schaltungen zwischen dem dynamischen Leistungsverbrauch $P_{dynamic}$ und der verwendeten Taktfrequenz f_k besteht.

$$P_{dynamic} = \sum_{k=1}^{M} C_k \cdot f_k \cdot V_{DD}^{\ 2} \tag{9}$$

Weiterhin kann durch eine Verminderung der Taktfrequenz eine Absenkung der Versorgungsspannung V_{DD} (voltage scaling) vorgenommen werden, die quadratisch in die Gleichungen (9) und (10) eingeht (siehe hierzu [43]).

$$P = C \cdot f \cdot V_{DD}^{\ 2} \tag{10}$$

Stehen den jeweiligen Systemen Mechanismen zur Verfügung, zur Laufzeit die notwendige Rechenleistung zu bestimmen, kann somit eine signifikante Herabsetzung der Verlustleistung herbeigeführt werden. Der Prozessortakt und somit auch die Versorgungsspannung werden hierbei an die Realzeiterfordernisse angepasst.
Hierfür wurde vom Verfasser ein Verfahren entworfen, das eine a priori Detektion der Codierungsparameter anhand der quantitativen Ermittlung der auftretenden Blocktypen zulässt. Das Verfahren wurde zum Patent angemeldet und ist in [75] näher beschrieben. Eine weitere Variante zur Komplexitätsbestimmung von Videodatenströmen stellen die in [47] vorgestellten MPEG Complexity Profiles dar.

5.5.2 Verwendung für andere Codierungsverfahren

Die vorgestellten Prozessorerweiterungen können prinzipiell für alle bisher etablierten Videocodierungsstandards verwendet werden, da ein großer Überdeckungsgrad bei den eingesetzten Algorithmen herrscht. Der zu erwartende Rechenzeitgewinn wird hierbei deutlich höher ausfallen, da der im Rahmen der Arbeit verwendete MPEG-4-Standard die höchste Komplexität aufweist im Verhältnis zu den älteren Codierungsstandards.
Eine Ausnahme hiervon stellt der H.264/AVC Standard dar, der eine höhere Komplexität aufzeigt. Darüber hinaus wird hier nicht auf eine 8x8 Cosinustransformation zurückgegriffen, womit eine Verwendung des IDCT-Coprozessors ausgeschlossen ist.
Weiterhin können sämtliche Codierungsverfahren von den Bitstromverarbeitungsbefehlen profitieren, die auf Entropiecodierungsverfahren beruhen. Hierzu zählen auch die Audiocodierungsverfahren, wie z.B. MPEG-1 Layer 3 (MP3) oder MPEG-4 Celp.
Firmenspezifische Verfahren für die Videocodierung, wie z.B. Real Video, weisen in der Regel ähnliche Algorithmen auf und könnten somit ebenfalls von den vorgestellten Optimierungen profitieren.
Ein Überblick über die mögliche Verwendung der Befehlssatzerweiterungen und des Coprozessors gibt die folgende Tabelle 8:

	IDCT	INTERPOLATE	ADDCLIP	ALLIGN	Bitstream
ITU H.263	✓	✓	✓	✓	✓
ISO 11172-2 MPEG-1 Video	✓	✓	✓	✓	✓
ISO 13818-2 MPEG-2 Video	✓	✓	✓	✓	✓
ITU H.264 Baseline		✓	✓	✓	✓
ISO 11172-2 MPEG-1 Audio (u.a. MP3)				✓	✓
ISO 14496-3 MPEG-4 Audio AAC, Celp				✓	✓

Tabelle 8 Verwendung für weitere Codierungsverfahren

5.6 Optimierungsansätze

Wie aus dem vorhergehenden Abschnitt zu entnehmen ist, ist die Decodierungsperformance noch sehr stark von den jeweils verwendeten Bildcodierungsparametern abhängig und führt zu starken Schwankungen in der benötigten Rechenzeit. Dies ist im Speziellen bei dedizierten Systemen nachteilig, die ausschließlich für die Decodierung von Videodaten implementiert werden, da hier immer der Worst-Case-Takt zugrunde gelegt werden muss. Es ist daher anzustreben, eine möglichst ausgeglichene Rechenlast zu erzielen. Nach eingehender Analyse der entwickelten Architektur kann dies jedoch nur über die Reduzierung der jeweils benötigten Speicherbandbreite vorgenommen werden.
Das vorliegende System verwendet für Speichertransfers Load-/Store-Operationen, d.h. alle Speichertransfers werden programmgesteuert ausgeführt. Durch die Verwendung spezieller Speichercontroller kann die Prozessorlast, die auf Speicheroperationen entfällt, gesenkt werden. Hierzu sind allerdings nur sequentielle Speicherzugriffe geeignet, da der hervorgerufene Kontrollaufwand sonst überproportional steigen würde.
Neben der Operandenbreite stellt die Erhöhung der Anzahl an verfügbaren Registern eine weitere Optimierungsmöglichkeit der vorgestellten Architektur dar.
Eine zusätzliche Erweitungsvariante besteht in der Implementierung eines anwendungsspezifischen Speichercontrollers, der neben reinen Speicherkopieroperationen auch komplexe Operationen, wie z.B. Teile der Bewegungskompensation, ausführen könnte. Dazu würden z.B. alle blockbasierten Kopier- wie auch Filteroperationen gehören, wodurch eine erhebliche Steigerung der Leistungsfähigkeit des vorliegenden Systems erzielt werden könnte.
Eine weitere Problematik ergibt sich aus der Architektur des eingesetzten RISC-Prozessors, die im vorliegenden Fall auf eine Operandenbreite von 32 Bit limitiert ist.
Wie die Analyse der implementierten Befehle und auch schon die Überlegungen zum Entwurf gezeigt haben, ist in vielen Fällen eine Parallelverarbeitung in Form von Instruction Level Parallelism durch die Struktur der Algorithmen gegeben. In den meisten Fällen wurde jedoch der erzielbare Gewinn der umgesetzten Befehle durch die Operandenbreite von 32 Bit begrenzt. Dies wird besonders deutlich im Vergleich zu den ebenfalls untersuchten SIMD-Implementierungen auf X86-Basis. Hier kann auf Grundlage der 128-Bit-Operanden ein höherer Grad an Parallelverarbeitung erzielt und der Rechenzeitgewinn maximiert werden.

6 Ausblick

Grundsätzlich kann gesagt werden, dass das im Rahmen dieser Arbeit vorgestellte Konzept zur Realisierung eines prozessorbasierten MPEG-4-Decoders den Stand der Technik der nächsten Jahre darstellen wird. Im Speziellen gilt dies für mobile Endgeräte, die auf der Basis eingebetteter Prozessoren in Form von SOCs umgesetzt werden.
Ein weiterer Trend, der das hier vorgestellte Konzept in seiner zukunftsweisenden Implementierung bestätigt, ist die Tatsache, dass die eingesetzten Prozessoren immer mehr zu Systemprozessoren migrieren, d.h. ein Prozessor in einem mobilen Endgerät wird in Zukunft nicht nur Coprozessoren für dedizierte Anwendungen verwenden, sondern vielmehr selbst wesentliche Aufgaben der Codierung bzw. Decodierung übernehmen.
Diese Prozessoren zeichnen sich durch einen sehr hohen Integrationsgrad in Bezug auf die Erfordernisse eines Betriebssystems aus. Dabei sind in der Regel MMU, Display-Schnittstelle und alle weiteren notwendigen Schnittstellen bereits auf einem Chip integriert, womit hochkompakte Systeme aufgebaut werden können.
Typische Vertreter solcher Systemprozessoren für mobile Endgeräte sind die Intel StrongArm- und Intel XScale-Prozessoren, Texas Instruments OMAP und die von AMD angekündigten MIPS-basierten Embedded-System-Prozessoren, die jedoch z. Zt. über keine der im Rahmen dieser Arbeit vorgestellten Erweiterungen verfügen.
Mit den hier vorgestellten Konzepten zur Erweitung einfacher RISC-Prozessoren können somit Systeme entwickelt werden, die für die Verwendung als Systemprozessor mit einem Betriebssystem gleichermaßen geeignet sind wie für die Verarbeitung von Multimediadaten.
Eine konsequente Weiterführung des Konzeptes der Befehlssatzerweiterung stellt die Erweiterung auf den H.264 Codierungsstandard dar, wie sie vom Verfasser in [78] beschrieben wird.
Eine weitere Tendenz stellt die Erweiterung von Terminals zur bidirektionalen Multimediakommunikation dar. Dabei werden in verstärktem Maße nicht nur Multimediadaten angefordert, sondern mit Hilfe entsprechender Encoder im Endgerät generiert.
Um solche Systeme effizient mit Hilfe der bereits erwähnten Systemprozessoren umsetzen zu können, wäre also eine Erweiterung der Beschleunigerfunktionalitäten auf encoderspezifische Funktionen erforderlich. Als Beispiel hierfür kann die ebenfalls vom Autor mitentwickelte Architektur eines prozessorbasierten MPEG-4-Videocodecs gelten, der in Abschnitt 6.1 beschrieben wird.
Ein weiteres zukünftiges Anwendungsgebiet der vorgestellten Architektur stellt der Einsatz in Consumer-Endgeräten, wie z.B. MPEG-4-fähige DVD-Player, dar.
Eines der wesentlichen Argumente, die zum Einsatz eines prozessorbasierten Decoders führen, spiegelt die Notwendigkeit wider, multistandardfähige Geräte zu entwerfen, die in der Lage sein müssen, einen Großteil der am Markt etablierten Kompressionsverfahren zu verarbeiten.
Da es jedoch unwirtschaftlich erscheint, in einem Gerät mehrere physikalisch getrennte Decoder zu betreiben und eine Umsetzung eines Multistandarddecoders mit Hilfe dedizierter Logik nicht effizient realisierbar ist, stellt eine prozessorbasierte Implementierung die zukünftige Architektur hierfür dar. Sinnvoll erscheint der Einsatz prozessorbasierter Strukturen auch unter dem Aspekt der Unterstützung neuer Standards, wobei durch Softwaretausch ein bereits bestehendes System leicht an ein neues Verfahren angepasst werden kann, sofern die hierfür notwendige Rechenleistung ausreichend ist.
Für den Einsatz prozessorbasierter Decoder in Geräten zur Decodierung und Darstellung höchstaufgelöster Bilddaten sind jedoch grundsätzlich technologische Grenzen zu erkennen, die sich durch den benötigten Bedarf nach größeren Speichern ergeben. Hier sind aufgrund

des Flächenbedarfs z. Zt. ausschließlich dynamische Speicherarchitekturen einsetzbar, die sich wegen ihrer Zugriffseigenschaften ungünstig auf die Speicherbandbreite in Prozessorsystemen auswirken. Der wahlfreie, nichtdeterministische Zugriff, der sich durch softwarebasierte Implementierungen ergibt, führt zu hohen Schwankungen in den Zugriffszeiten der dynamischen Speicherarchitekturen, die für blockweisen Zugriff entworfen sind. Abschnitt 6.2 umreißt die Grundmerkmale einer Architektur für Multistandard-Decoderarchitekturen für höhere Bildauflösungen.

6.1 Prozessorbasierter Videocodec

Im Rahmen dieser Arbeit wurde eine optimierte Prozessorarchitektur vorgestellt, die im Hinblick auf die wesentlichen Algorithmen zur MPEG-Decodierung erweitert wurde. Der Hauptanwendungsbereich prozessorbasierter Videodecoder ist vorerst hauptsächlich im Bereich der Mobilkommunikation zu sehen, wo jedoch auch in zunehmendem Maße die bidirektionale, visuelle Kommunikation eine Rolle spielen wird. Es ist daher nahe liegend auch an die Umsetzung von prozessorbasierten Videoencodern bzw. Videocodecs zu denken. Wie aus der Beschreibung der grundsätzlichen Funktion von Videocodierungsalgorithmen in Abschnitt 3.3 zu entnehmen ist, besteht bei hybriden Codierungsverfahren der Encoder zu einem Teil aus zum Decoder identischen Funktionalitäten.
Ein Decoder stellt somit eine Unterfunktion eines Encoders dar. Die bereits für den Decoder entworfenen Optimierungen sind damit in ihrer Vollständigkeit auch für die Umsetzung eines Encoders zu verwenden. Hinzu kommen die Funktionsteile der Bewegungsschätzung, Entropiecodierung und Cosinustransformation. Aufgrund der zusätzlichen Funktionalitäten verschlechtert sich jedoch der erzielte Performancegewinn gegenüber einer Decoderimplementierung. Das nachfolgende Diagramm in Abbildung 80 stellt die Rechenlastverteilung eines MPEG-4 SP@L1-Videoencoders auf einem ARM7TDMI RISC-Prozessor dar.

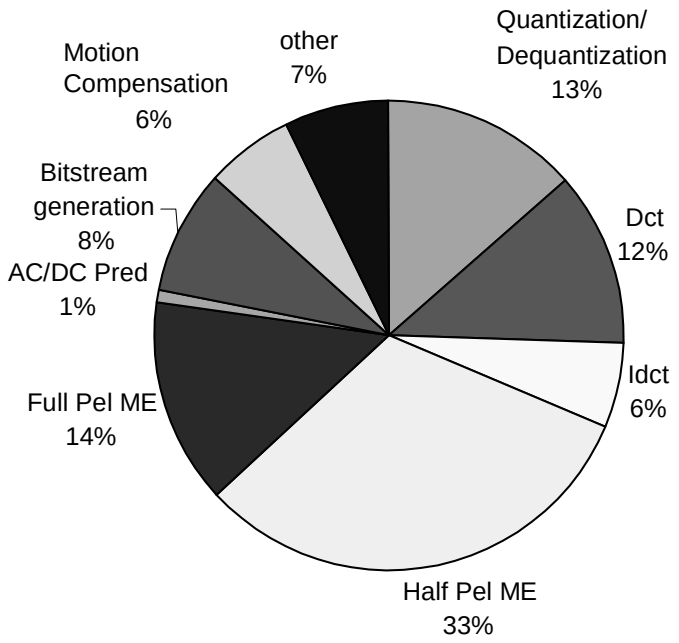


Abbildung 80 Prozessorauslastung eines MPEG-4-Videoencoders auf ARM7TDMI-Basis

Für die Untersuchung wurde eine reine Softwareimplementierung herangezogen, die in analoger Weise zu der im Rahmen dieser Arbeit verwendeten Decodersoftware vom Verfasser entwickelt wurde.
Wie der Darstellung zu entnehmen ist, benötigen die decoderspezifischen Algorithmen nur noch ca. 20% der Gesamtrechenleistung und stellen somit nicht mehr die wichtigsten Kandidaten für eine Optimierung dar.
Der größte Teil der Rechenleistung von ca. 47% (Full Pel ME und Half Pel ME) wird hingegen von der Bewegungsschätzung benötigt, die zum Auffinden der Bewegungsvektoren bei prädiktiv codierten Bildern dient. Der erste Ansatz für eine Optimierung zielt somit auf die Analyse der Algorithmen der Bewegungsschätzung ab. Da es sich hierbei um eine einfach entkoppelbare Funktion handelt, erscheint eine Coprozessorerweiterung hierfür günstig. Für alle weiteren Algorithmenteile des Encoders bieten sich die gleichen Optimierungsmaßnahmen wie für den beschriebenen Decoder an. Die entworfenen Befehlssatzerweiterungen könnten bei Einsatz des ARC in gleicher Weise verwendet werden und würden den in Bezug auf die Gesamtrechenleistung anzusetzenden Gewinn erbringen. Im vorliegenden Falle wurde jedoch der schon erwähnte Prozessorkern ARM7TDMI eingesetzt, der keine Befehlssatzerweiterungen zulässt, weshalb ausschließlich eine Bewegungsschätzung als Coprozessor umgesetzt wurde. Eine genauere Beschreibung des umgesetzten Videocodecs kann in [77] gefunden werden.
Der Encoder erreicht mit der beschriebenen Erweiterung Realzeitfähigkeit für MPEG-4 SP@L1-Codierungsparameter bei ca. 100 MHz und ist für Anwendungen im Bereich der Mobilkommunikation konzipiert worden.

6.2 Multistandard-Decoder

Wie aus den Ausführungen in Abschnitt 3 hervorgeht, basieren die etablierten Standards z. Zt. auf sehr ähnlichen Konzepten und sind somit immer auf blockbasierte Verfahren zurückzuführen, die Transformationscodierung und Bewegungskompensation in sich vereinen.
Neben den unterschiedlichen Anwendungsszenarien, die zu der Entwicklung neuer standardisierter Verfahren geführt haben, ist die Steigerung der Codierungseffizienz der wesentliche Aspekt. Der steigende Codierungsgewinn ist bei genauerer Betrachtung der MPEG- (MPEG-1,-2,-4) bzw. ITU-Standards (H.263, H.264) auf den zunehmenden Gebrauch von Prädiktionsverfahren und deren Verfeinerung zurückzuführen, wie z.B. die Verkleinerung der Blockgrößen und die Erhöhung der Genauigkeit der Bewegungsvektoren.
Aufgrund der verbesserten Codierungseffizienz bieten sich für bereits bestehende Infrastrukturen, wie z.B. DVB-T, neue Anwendungsmöglichkeiten, die den Einsatz neuer Standards beschleunigen könnten. Um den Übergang auf neue Codierungsverfahren zu ermöglichen, sieht die dem DVB-Standard zugrunde liegende MPEG-2-Spezifikation vor, den spezifizierten Transportstrom auch für zukünftige Codierungsverfahren verwenden zu können.
Ein ähnlicher Prozess hat schon bei den Audiocodierungsverfahren stattgefunden, wo nach der Einführung des MPEG-2-Standards auch Dolby AC3 als alternatives Codierungsverfahren zunehmend mehr Verwendung gefunden hat.
Entsprechende Endgeräte werden sich in Zukunft daher durch die Fähigkeit auszeichnen, unterschiedliche Codierungsstandards zu unterstützen. Eine Implementierung auf der Basis verschiedener einzelner Decoder stellt dabei eine unwirtschaftliche Lösung dar, da ein großer Überdeckungsgrad bei den verwendeten Konzepten besteht.
Eine nahe liegende Lösung dieses Problems ist die softwarebasierte Umsetzung der einzelnen unterschiedlichen Decoder auf einem leistungsfähigen Prozessor, wobei die Möglichkeit eines

Softwareupgrades gegeben wäre. Die Leistungsfähigkeit des Prozessors muss dabei an der Komplexität des aufwendigsten Verfahrens ausgerichtet werden.
Bei der Implementierung der einzelnen z. Zt. aktuellen Verfahren hat sich jedoch gezeigt (siehe hierzu auch Abschnitt 5.5), dass, speziell im Falle von prozessorbasierten Decoderimplementierungen, die notwendige Systemspeicherbandbreite mit steigender Codierungseffizienz zunimmt. Dies ist auf den intensiveren Einsatz von Prädiktionsmechanismen zurückzuführen, die die schon im Decoderspeicher vorhandenen Daten teilweise mehrfach für die Rekonstruktion der aktuellen Bilddaten heranziehen. Das gilt in besonderem Falle für die Bewegungskompensation, wo bei steigender Genauigkeit der Bewegungsvektoren eine größere Menge an Pixeln für die Rekonstruktion des adressierten Bildpunktes verwendet werden muss. Weiterhin ist die Prädiktion im Ortsbereich zu nennen, die ebenfalls zu einer wesentlichen Erhöhung der erforderlichen Speicherbandbreite führt.
Um diesen Zusammenhang zu verdeutlichen, stellt das folgende Diagramm in Abbildung 81 den Vergleich der reinen Speicherzugriffe auf den Arbeitspeicher eines MPEG-4- (Simple Profile @ Level 2) und eines H.264-Decoders (Baseline Profile) dar.

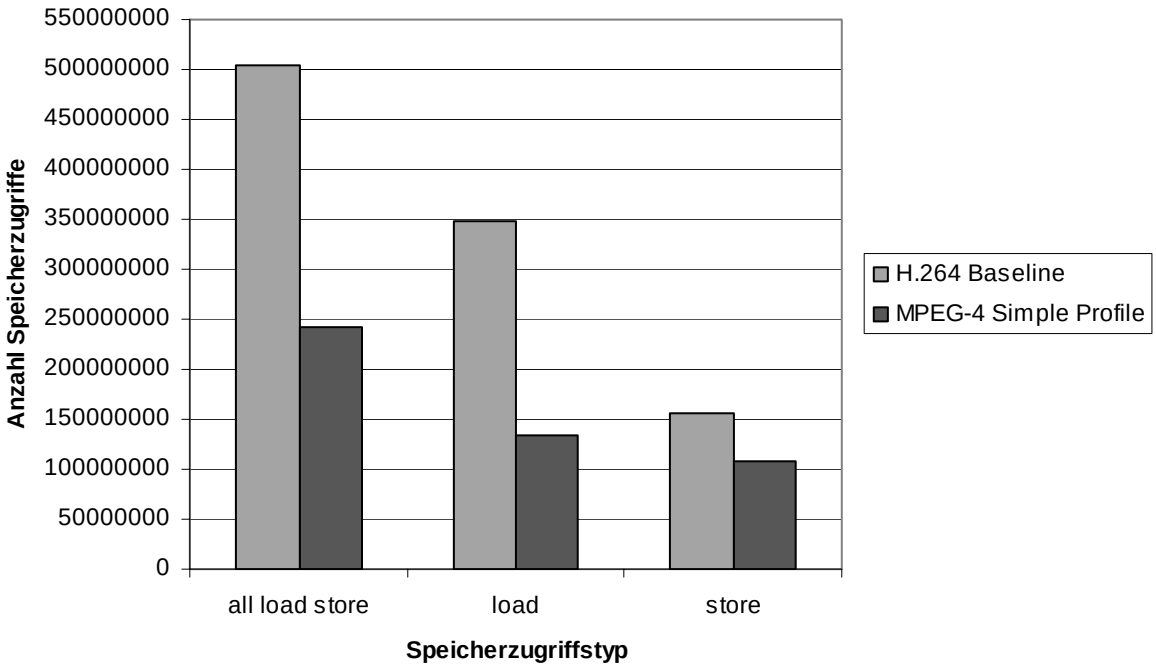


Abbildung 81 Vergleich der Anzahl der Speicherzugriffe bei MPEG-4 und H.264

Beide Decoder haben dabei die gleiche Bildsequenz (300 Bilder) bei gleicher Datenraten und Bildauflösung (128 kBit/s, CIF, 352x240) zu verarbeiten.
Die Decoder wurden als reine Softwareimplementierungen mit dem im Rahmen dieser Arbeit vorgestellten Profiler untersucht, wobei die reinen Speicherzugriffe ohne Instruction-Fetch-Operationen betrachtet wurden. Deutlich ist der Unterschied in der absoluten Zahl der Speicherzugriffe (*all load store*) zu erkennen, die bei H.264 etwa doppelt so hoch ausfällt und zu einem großen Teil auf Lesezugriffe auf den Prädiktionsspeicher zurückzuführen ist.
Als Resümee aus den Entwicklungen der vorliegenden Arbeit ist jedoch unter anderem zu entnehmen, dass Speicherzugriffe mit hohen Datenraten nur sehr ungünstig mit Prozessorarchitekturen abgebildet werden können.

Eine Lösung dieses Problems stellt die Trennung der standardspezifischen Algorithmenteile von den gemeinsam nutzbaren Verfahren dar. Dabei kann für die Implementierung eines entsprechenden Decoders eine hardwarebasierte, generische Bewegungskompensation umgesetzt werden, die den Speicherzugriff mit hohen Datenraten gewährleistet.
Neben den reinen Speicherzugriffsoperationen führt die gestiegene Datenrate jedoch auch zu einem wesentlich höheren Anteil arithmetischer Operationen, die durch den verwendeten Prozessorkern ausgeführt werden müssen. Hier fällt die Anzahl benötigter Operationen für das H.264 Baseline Profile ca. dreimal höher aus als für das MPEG-4 Simple Profile.
Wie der vorliegenden Arbeit als Ergebnis zu entnehmen ist, können durch die konsequente Umsetzung des SIMD-Prinzips leistungsfähige Prozessoren entworfen werden. Die Voraussetzung hierfür ist jedoch eine große zur Verfügung stehende Operandenbreite, die somit einen hohen Grad an Parallelverarbeitung gewährleistet. Vom Verfasser wird daher z. Zt. ein Prozessorkern entworfen, der an die speziellen Bedürfnisse von Multistandardarchitekturen angepasst und in [78] näher beschrieben ist.

Anhang A Programmauszüge

A.1 Bitstromverarbeitung

ShowBits, FlushBits, GetBits

```
/*****************************************************************************
Proto:        Int32           viShowBits( UInt32 length);
Author:
Purpose:
Abstract:
In:           length
InOut:
Out:
Modified:
*****************************************************************************/
UInt32  viShowBits(UInt32 u_length, stream * bits)
{
      UInt32 u_word ;

      u_word = bits->u32_curword;

      if (u_length<=bits->bits_valid)
      {
            u_word = u_word >> (32-u_length);
      }

      else        /* second word */
      {
            u_word = bits->u32p_buff[0];
            u_word=sw(u_word);
            bits->u32_curword |= (u_word >> bits->bits_valid);
            u_word = bits->u32_curword >> (32-u_length);

      }

      return u_word;
}

/*****************************************************************************
Proto:          void viFlushBits(UInt32 length);
Author:
Purpose:
Abstract:
In:
InOut:
Out:
Modified:
*****************************************************************************/

void     viFlushBits( UInt32 u_length, stream * bits)
{

      Int32 d;
      d=u_length-bits->bits_valid;

      if (d<=0)
      {
            bits->u32_curword <<= u_length;
            bits->bits_valid-=u_length;
      }
      else        /* second word */
      {
            bits->u32_curword = bits->u32p_buff[0];
            bits->u32_curword=sw(bits->u32_curword);
               bits->u32p_buff++;
            bits->u32_curword <<= d;
            bits->bits_valid= 32-d;

      }
```

```
}

/*****************************************************************************
Proto:
UInt32          viGetBits( UInt32 length);
Author:    Benno Stabernack / HHI
Purpose:
Abstract:
In:                     length
InOut:
Out:
Modified:   Optimization -> unroll getbits from using showbits and flush
seperately
*****************************************************************************/

UInt32 viGetBits( UInt32 u_length, stream * bits)
{
      UInt32 word ;
      word = viShowBits(u_length,bits);
      viFlushBits(u_length,bits);
      return word;
}
```

A.2 Interpolationsfilter

VerFilter8, HorFilter8, HorVerFilter8

```
/************************************************************************
Proto:
void VerFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
Author:    Benno Stabernack / HHI
Purpose:   Gets prediction data with vertical half-pixel interpolation.

Abstract:  The texture data of the reference vop is vertical interpolated and
           copied to the outputbuffer (des).

In:        src: pointer to texture data, height: vertical blocksize, src_stride: stride of
            texture data,
           rc: rounding control
InOut:
Out:       des: pointer to output buffer
Modified:
************************************************************************/
void VerFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
{
      UInt32 i;
      UInt8 CONST *p, *s;
      UInt32 a, b;

    rc = 1 - rc;
    s = (UInt8 CONST *) src;
    p = (UInt8 CONST *) src + src_stride;
    src_stride=src_stride-8;

 for( i = 0; i < 8; i++ )
  {
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        a = *s++;b = *p++;*des++ = (UInt8)((rc + a + b) >>1);
        s+=src_stride;p+=src_stride;
  }

}
/************************************************************************
Proto:           void HorFilter8(UInt8 *src,
                                 Int32 height,
                                 Int32 src_stride,
                                 UInt8 rc,
                                 UInt8 *des)
Author:          Benno Stabernack / HHI
Purpose:         Gets prediction data with horizontal half-pixel interpolation.

Abstract:        The texture data of the reference vop is horizontal interpolated
                 and copied to the outputbuffer (des).

In:              src: pointer to texture data, height: vertical blocksize,
                 src_stride: stride of texture data,
                 rc: rounding control
InOut:
Out:             des: pointer to output buffer
Modified:
************************************************************************/

void HorFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
{
    Int32 i, x;
    UInt32 l, m;

    rc = 1 - rc;
    src_stride-=8;
```

```
    for(i=0; i<height; i++)
    {
        l = *src;
        for( x = 0; x < 8; x++)
        {
            src++;
            m = *src;
            *des++ = (UInt8)((l + m + rc)>>1);
            l = m;
        }
        src+=src_stride;
    }

}
/*****************************************************************************
Proto:            void HorVerFilter8(     UInt8 *src,
                                          Int32 height,
                                          Int32 src_stride,
                                          UInt8 rc,
                                          UInt8 *des)
Author:           Benno Stabernack / HHI
Purpose:          Gets prediction data with horizontal and vertical half-pixel
                  interpolation.

Abstract:         The texture data of the reference vop is horizontal and vertical
                  interpolated and copied to the outputbuffer (des).

In:               src: pointer to texture data, height: vertical blocksize,
                  src_stride: stride of texture data,
                  rc: rounding control
InOut:
Out:              des: pointer to output buffer
Modified:
*****************************************************************************/
void HorVerFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
{
    Int32 i, x;
    UInt32 l,m,n,o;
    UInt8 CONST *p;

    p= src + src_stride;
    src_stride-=8;
    rc = 2 - rc;
    for(i=0; i<height; i++)
    {
        l = *src;
        n = *p;
        for( x = 0; x < 8; x++ )
        {
           src++;
           p++;

           m    = *src;
           o    = *p;
           *des++ = (UInt8)((l + m + n + o + rc)>>2);
           l    = m;
           n    = o;
        }
        src+=src_stride;p+=src_stride;
   }

}
```

A.3 Bewegungskompensation

```
/*****************************************************************************
Proto:            void SavePredPlusCoeff(UInt8 *pred, Int16 *coef, Int32 stride, UInt8 *des)
Author:           Benno Stabernack / HHI
Purpose:          Saves the clipped combination of prediction and inverse transformed texture
                  values to the frame buffer.
Abstract:         The inverse transformed texture values are added to the prediction data,
                  clipped and stored in the appropriate frame buffer.

In:               pred: pointer to prediction data, src: pointer to texture data, stride:
                  stride of texture data
InOut:
Out:              des: pointer to frame buffer
Modified:
*****************************************************************************/
/* save the clipped combination of prediction and dct data */
void SavePredPlusCoeff(UInt8 *pred, Int16 *coef, Int32 stride, UInt8 *des)
{
    Int32 i,j;
    UInt8 l_u8;
    Int16 m_16;
    Int32 n_32;
    UInt8 CONST *p;
    Int16 CONST *c;

    DEBUG_SIGN(function_entry_SavePredPlusCoeff);
    p = pred;
    c = coef;
    stride-=8;;
    for (j=0;j<8;j++)
    {
        for( i = 0; i < 8; i++ )
        {
            l_u8 = *p++;
            m_16 = *c++;
            n_32 = (Int32) l_u8 + (Int32) m_16;
            *des++ = clip( n_32 );
        }
        des+=stride;

    }

    DEBUG_SIGN(function_exit_SavePredPlusCoeff);

}
```

A.4 Inverse DCT C-Quellcode

```
/*****************************************************************************
Proto:
void            Idct(Int16 *ai16_dct);
Author:         B.S., HHI
Purpose:        Performs a block inverse DCT.
Abstract:       The inverse DCT is of the current blcok is performed.
                The accuracy fits the standard requirements.
                        It works with interm results on 16 bits
                between 1st and second pass.This has been checked
                        (by the way 15 bits are sufficients ! )
In:
InOut:          ai16_dct
Out:                    none
Modified: 14.10.97
          04.09.02 converted access to idct field from Int16 to Int32
*****************************************************************************/


void  Idct(Int16 * ai16_dct)
{

            Int32 *ptr32;
            Int16 *ptr;
      Int32 i,u0,u1,u2,u3,u4,u5,u6,u7,u8;

      ptr32 = (Int32*)ai16_dct;
      for (i = BLOCK_SIZE; i; i--, ptr32 += 4)
      {
            /* do a row idct */

            /* load ac coefficients */
            /*
            u1 = 4<<11
            u2 = 6
            u3 = 2
            u4 = 1
            u5 = 7
            u6 = 5
            u7 = 3
            */
            u0 = ptr32[0];            /* [0]<<16|[1] dep.on endianess (LE 1,0) */
            u3 = ptr32[1];            /* [2]<<16|[3] dep. on endianess (LE 3,2) */
            u1 = ptr32[2];            /* 4,5 dep. on endianess (LE 5,4) */
            u2 = ptr32[3];            /* 6,7 dep. on endianess (LE 7,6) */

#ifdef LITTLE_ENDIAN
            u8 = u0 & 0xffff0000;         /* AC1 LE */
#else
            u8 = u0 & 0xffff;       /* AC1 BE */
#endif
            u8 |= (u3 | u1 | u2);         /* quick double word check */
            if ( u8 == 0)
            {
#ifndef LITTLE_ENDIAN
                  u0>>=16;          /* BE: [0]<<16|[1] --> [0] */
#endif
                  u0 <<= 3;
                  u0  &= 0xffff;
                  u0  |= u0 << 16;  /* u0 << (16+3) | u0<<3 */

                  /* zero ac coefficients so just extend dc along */
                  ptr32[0] = ptr32[1] = ptr32[2] = ptr32[3] = u0;
            }
            else
            {
#ifdef LITTLE_ENDIAN
                  u4 = u0 >> 16;          /* LE 1,0 -> 1 */
                  u0 <<= 16;        /* move lower word upwards to gain correct sign */
                  u0 >>= (16-11);         /* move down -> sign bit extended */

                  u7 = u3 >> 16;          /* LE 3,2 -> 3 */
                  u3 <<= 16;
```

```
                u3 >>= 16;

                u6 = u1 >> 16;
                u1 <<= 16;
                u1 >>= (16 - 11);

                u5 = u2 >> 16;
                u2 <<= 16;
                u2 >>= 16;
#else
                u4 = u0 << 16;          /* BE [0,1] -> [1,0x0000] */
                u4 >>= 16;        /* [1] sign extended */
                u0 >>= 16;
                u0 <<= 11;

                u7 = u3 << 16;
                u7 >>= 16;
                u3 >>= 16;

                u6 = u1 << 16;
                u6 >>= 16;
                u1 >>= 16;
                u1 <<= 11;

                u5 = u2 << 16;
                u5 >>= 16;
                u2 >>= 16;
#endif
                u0 += 128;

                /* first stage */
                u8 = CC7 * (u4 + u5);
                u4 = u8 + (CC1 - CC7) * u4;
                u5 = u8 - (CC1 + CC7) * u5;
                u8 = CC3 * (u6 + u7);
                u6 = u8 - (CC3 - CC5) * u6;
                u7 = u8 - (CC3 + CC5) * u7;

                /* second stage */
                u8 = u0 + u1;
                u0 -= u1;
                u1 = CC6 * (u3 + u2);
                u2 = u1 - (CC2 + CC6) * u2;
                u3 = u1 + (CC2 - CC6) * u3;
                u1 = u4 + u6;
                u4 -= u6;
                u6 = u5 + u7;
                u5 -= u7;

                /* 3rd and 4th stage and store */
                ComputeAndStore32(0, 1, 2, 3, 4, 5, 6, 7, 8);
          }
     }


     for (ptr = ai16_dct, i = BLOCK_SIZE; i; i--, ptr++)
     {
          /* do a column idct */

          /* load ac coefficients */
          LoadCoefs(4 * BLOCK_SIZE, 6 * BLOCK_SIZE, 2 * BLOCK_SIZE, 1 * BLOCK_SIZE,
                7 * BLOCK_SIZE, 5 * BLOCK_SIZE, 3 * BLOCK_SIZE, 8);

          if ((u1 | u2 | u3 | u4 | u5 | u6 | u7) == 0)
          {
                /* zero ac coefficients so just extend dc along */
                ptr[0] = ptr[1 * BLOCK_SIZE] = ptr[2 * BLOCK_SIZE] = ptr[3 * BLOCK_SIZE] =
                      ptr[4 * BLOCK_SIZE] = ptr[5 * BLOCK_SIZE] = ptr[6 * BLOCK_SIZE] =
                      ptr[7 * BLOCK_SIZE] = ((ptr[0] + 32) >> 6);
          }
          else
          {
                u0 = (ptr[0] << 8) + 8192;

                /* first stage */
```

```
                u8 = CC7 * (u4 + u5) + 4;
                u4 = (u8 + (CC1 - CC7) * u4) >> 3;
                u5 = (u8 - (CC1 + CC7) * u5) >> 3;
                u8 = CC3 * (u6 + u7) + 4;
                u6 = (u8 - (CC3 - CC5) * u6) >> 3;
                u7 = (u8 - (CC3 + CC5) * u7) >> 3;

                /* second stage */
                u8 = u0 + u1;
                u0 -= u1;
                u1 = CC6 * (u3 + u2) + 4;
                u2 = (u1 - (CC2 + CC6) * u2) >> 3;
                u3 = (u1 + (CC2 - CC6) * u3) >> 3;
                u1 = u4 + u6;
                u4 -= u6;
                u6 = u5 + u7;
                u5 -= u7;

                /* 3rd and 4th stage and store */
                ComputeAndStore(0, 1 * BLOCK_SIZE, 2 * BLOCK_SIZE, 3 * BLOCK_SIZE,
                      4 * BLOCK_SIZE, 5 * BLOCK_SIZE, 6 * BLOCK_SIZE,
                      7 * BLOCK_SIZE, 14);
            }
        }
}

#endif /* matches #ifndef __MMX_IDCT__ */
```

A.5 Inverse DCT MMX Version

```
/*****************************************************************************
Proto:
void            MMX_Idct(Int16 *ai16_dct);
Author:         HHI
Purpose:        Performs a block inverse DCT using the mmx instruction set for PII.
Abstract:
In:
InOut:
Out:
*****************************************************************************/

void  Idct(Int16 * ai16_dct)
{
    int i,j;
    __int16 *tmp = intermediate;
    __int16 *out = ai16_dct;

    /*****************************************************/
    /* row transform                                     */
    /*****************************************************/
    mmx_one_row_idct (ai16_dct, tmp, coeff_0_4);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_1_7);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_2_6);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_3_5);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_0_4);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_3_5);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_2_6);
    ai16_dct += 8;
    tmp += 8;
    mmx_one_row_idct (ai16_dct, tmp, coeff_1_7);

    /*****************************************************/
    /* column transform                                  */
    /*****************************************************/

    mmx_column_idct (intermediate, out);

    /*****************************************************/
    /* scaling                                           */
    /*****************************************************/

    mmx_copy_int16_sc (out);

}


void mmx_one_row_idct (__int16* input, __int16* output, __int16 *coeff)
{

    __asm {

        ;;push        eax
        ;;push        ebx
        ;;push        ecx
        ;;push        esi

        mov         eax, input            ; pointer to coefficients
        mov         ebx, output
        mov         ecx, coeff
```

```
        ;; are there any non-zero coefficients ?? ;;;;;;;;;;;;;;;;;;;;;;

        movq        mm0, qword ptr [eax]           ; 0 ; x3 x2 x1 x0
        movq        mm1, qword ptr [eax + 8]       ; 1 ; x7 x6 x5 x4
        pxor        mm7, mm7                       ; 7 ;  0  0  0  0
        movq        mm6, mm0
        por         mm6, mm1                       ;   ;
        movq        mm2, mm6                       ; 2 ;
        psrlq       mm6, 32
        por         mm2, mm6
        movd        edi, mm2
        cmp         edi, 0
        jne         transform
        movq        qword ptr [ebx], mm7
        movq        qword ptr [ebx+8], mm7
        jmp         end

transform:
        ;; transform of one row ;;;;;;;;;;;;;;;;;;;;;;
        movq        mm2, mm0                       ; 2 ; x3 x2 x1 x0
        movq        mm3, qword ptr [ecx]           ; 3 ; w06 w04 w02 w00
        punpcklwd   mm0, mm1                       ;   ; x5 x1 x4 x0
        movq        mm5, mm0                       ; 5 ; x5 x1 x4 x0
        punpckldq   mm0, mm0                       ;   ; x4 x0 x4 x0
        movq        mm4, qword ptr [ecx + 8]       ; 4 ; w07 w05 w03 w01
        punpckhwd   mm2, mm1                       ;   ; x7 x3 x6 x2
        pmaddwd     mm3, mm0                       ;   ; x4*w06+x0*w04 x4*w02+x0*w00
        movq        mm6, mm2                       ; 6 ; x7 x3 x6 x2
        movq        mm1, qword ptr [ecx + 32]      ; 1 ; w22 w20 w18 w16
        punpckldq   mm2, mm2                       ;   ; x6 x2 x6 x2
        pmaddwd     mm4, mm2                       ;   ; x6*w07+x2+w05 x6*w03+x2*w01
        punpckhdq   mm5, mm5                       ;   ; x5 x1 x5 x1
        pmaddwd     mm0, qword ptr [ecx + 16]      ;   ; x4*w14+x0*w12 x4*w10+x0*w08
        punpckhdq   mm6, mm6                        ;   ; x7 x3 x7 x3
        movq        mm7, qword ptr [ecx + 40]      ; 7 ; w23 w21 w19 w17
        pmaddwd     mm1, mm5                       ;   ; x5*w22+x1*w20 x5*w18+x1*w16
        paddd       mm3, qword ptr round_inv_row   ;   ;
        pmaddwd     mm7, mm6                       ;   ; x7*w23+x3*w21 x7*w19+x3*w17
        pmaddwd     mm2, qword ptr [ecx + 24]      ;   ; x6*w15+x2*w13 x6*w11+x2*w09
        paddd       mm3, mm4                       ; 4 ; a1=sum(even1) a0=sum(even0)
        pmaddwd     mm5, qword ptr [ecx + 48]      ;   ; x5*w30+x1*w28 x5*w26+x1*w24
        movq        mm4, mm3                       ; 4 ; a1 a0
        pmaddwd     mm6, qword ptr [ecx + 56]      ;   ; x7*w31+x3*w29 x7*w27+x3*w25
        paddd       mm1, mm7                       ; 7 ; b1 =sum(odd1) b0=sum (odd0)
        paddd       mm0, qword ptr round_inv_row   ;   ;
        psubd       mm3, mm1                       ;   ; a1-b1 a0-b0
        psrad       mm3, qword ptr shift_inv_row   ;   ; y6 = a1-b1 y7 = a0-b0
        paddd       mm1, mm4                       ; 4 ; a1+b1 a0+b0
        paddd       mm0, mm2                       ; 2 ; a3=sum(even3) a2=sum(even2)
        psrad       mm1, qword ptr shift_inv_row   ;   ; y1=a1+b1 y0=a0+b0
        paddd       mm5, mm6                       ; 6 ; b3=sum(odd3) b2=sum(odd2)
        movq        mm4, mm0                       ; 4 ; a3 a2
        paddd       mm0, mm5                       ;   ; a3+b3 a2+b2
        psubd       mm4, mm5                       ;   ; a3-b3 a2-b2
        psrad       mm0, qword ptr shift_inv_row   ;   ; y3=a3+b3 y2=a2+b2
        psrad       mm4, qword ptr shift_inv_row   ;   ; y4=a3-b3 y5=a2-b2
        packssdw    mm1, mm0                       ; 0 ; y3 y2 y1 y0
        packssdw    mm4, mm3                       ; 3 ; y6 y7 y4 y5
        movq        mm7, mm4                       ; 7 ; y6 y7 y4 y5
        psrld       mm4, 16                        ;   ;  0 y6 0  y4
        pslld       mm7, 16                        ;   ; y7 0 y5 0
        movq        qword ptr [ebx], mm1           ; 1 ; save y3 y2 y1 y0
        por         mm7, mm4                       ; 4 ; y7 y6 y5 y4
        movq        qword ptr [ebx + 8], mm7       ; 7 ; save y7 y6 y5 y4
        ;; end of transform of one row ;;;;;;;;;;;;;;;;;;;;;;
end:
        emms
        ;;pop     esi
        ;;pop     ecx
        ;;pop     ebx
        ;;pop     eax

    }
}
```

Anhang B VHDL Code-Auszüge

B.1 Extension ALU-Implementierung ALIGN-Befehl

```
-- B.S. ================================= ALIGN Instruc-
tions==================================

  i_align1  <= s2val(7 downto 0)  & s1val(31 downto 8);  -- B.S. ALIGN1
  i_align2  <= s2val(15 downto 0) & s1val(31 downto 16); -- B.S. ALIGN1
  i_align3  <= s2val(23 downto 0) & s1val(31 downto 24); -- B.S. ALIGN1
```

B.2 Extension ALU-Implementierung ADDCLIP-Befehl

```
-- B.S. ================================= ADDCLIP Instructions
================================

  i_add1 <= "00" & s1val(7 downto 0)   + s2val(9 downto 0);   -- B.S. ADD first Pixel
  i_add2 <= "00" & s1val(15 downto 8)  + s2val(25 downto 16); -- B.S. ADD second Pixel
  i_add3 <= "00" & s1val(23 downto 16) + s2val(9 downto 0);   -- B.S. ADD third Pixel
  i_add4 <= "00" & s1val(31 downto 24) + s2val(25 downto 16); -- B.S. ADD fourth Pixel

  i_addclipl( 7 DOWNTO  0) <= (others => '0') WHEN i_add1(9)='1' else -- B.S. Clipping nega-
tive to 0
                              (others => '1') WHEN i_add1(9 DOWNTO 8) = "01" ELSE  -- B.S.
Clipping overflow to 'FF'
                              i_add1(7 DOWNTO 0);                     -- B.S.
  i_addclipl(15 DOWNTO  8) <= (others => '0') WHEN i_add2(9)='1' else -- B.S. Clipping nega-
tive to 0
                              (others => '1') WHEN i_add2(9 DOWNTO 8) = "01" ELSE  -- B.S.
Clipping overflow to 'FF'
                              i_add2(7 DOWNTO 0);                     -- B.S.
  i_addclipl(31 DOWNTO 16) <= (others => '0');                        -- B.S.

  i_addcliph(15 DOWNTO  0) <= (others => '0');                        -- B.S.
  i_addcliph(23 DOWNTO 16) <= (others => '0') WHEN i_add3(9)='1' else -- B.S. Clipping nega-
tive to 0
                              (others => '1') WHEN i_add3(9 DOWNTO 8) = "01" ELSE  -- B.S.
Clipping overflow to 'FF'
                              i_add3(7 DOWNTO 0);                     -- B.S.
  i_addcliph(31 DOWNTO 24) <= (others => '0') WHEN i_add4(9)='1' else -- B.S. Clipping nega-
tive to 0
                              (others => '1') WHEN i_add4(9 DOWNTO 8) = "01" ELSE  -- B.S.
Clipping overflow to 'FF'
                              i_add4(7 DOWNTO 0);                     -- B.S.
```

B.3 Extension ALU-Implementierung INTERPOLATE-Befehl

```
-- B.S. ======================  Interpolate Instructions ================================

  i_intl1 <= ('0' & s1val( 7 DOWNTO  0)) + ('0' & s2val( 7 DOWNTO  0)); -- B.S. add first
Pixel
  i_intl2 <= ('0' & s1val(15 DOWNTO  8)) + ('0' & s2val(15 DOWNTO  8)); -- B.S. add second
Pixel
  i_intl3 <= ('0' & s1val(23 DOWNTO 16)) + ('0' & s2val(23 DOWNTO 16)); -- B.S. add third
Pixel
  i_intl4 <= ('0' & s1val(31 DOWNTO 24)) + ('0' & s2val(31 DOWNTO 24)); -- B.S. add fourth
Pixel

  i_interpolatel( 7 DOWNTO  0) <= i_intl1(8 DOWNTO 1); -- B.S. div first Pixel
  i_interpolatel(15 DOWNTO  8) <= i_intl2(8 DOWNTO 1); -- B.S. div first Pixel
  i_interpolatel(23 DOWNTO 16) <= i_intl3(8 DOWNTO 1); -- B.S. div first Pixel
  i_interpolatel(31 DOWNTO 24) <= i_intl4(8 DOWNTO 1); -- B.S. div first Pixel

  i_inth1 <= ('0' & s1val( 7 DOWNTO  0)) +  ('0' & s2val( 7 DOWNTO  0)) + "000000001"; -- B.S.
add first Pixel + 1
  i_inth2 <= ('0' & s1val(15 DOWNTO  8)) +  ('0' & s2val(15 DOWNTO  8)) + "000000001"; -- B.S.
add second Pixel + 1
  i_inth3 <= ('0' & s1val(23 DOWNTO 16)) +  ('0' & s2val(23 DOWNTO 16)) + "000000001"; -- B.S.
add third Pixel + 1
  i_inth4 <= ('0' & s1val(31 DOWNTO 24)) +  ('0' & s2val(31 DOWNTO 24)) + "000000001"; -- B.S.
add fourth Pixel + 1

  i_interpolateh( 7 DOWNTO  0) <= i_inth1(8 DOWNTO 1); -- B.S. div first Pixel
  i_interpolateh(15 DOWNTO  8) <= i_inth2(8 DOWNTO 1); -- B.S. div first Pixel
  i_interpolateh(23 DOWNTO 16) <= i_inth3(8 DOWNTO 1); -- B.S. div first Pixel
  i_interpolateh(31 DOWNTO 24) <= i_inth4(8 DOWNTO 1); -- B.S. div first Pixel
```

B.4 Extension ALU-Implementierung Bitstream-Befehle

```
-- B.S. ======================  GetBits and ShowBits Instructions ======================


i_bitptrnew(5 DOWNTO 0) <= ('0' & i_bitptr(4 DOWNTO 0)) + ('0' & s1val(4 DOWNTO 0)); -- B.S.
always calculate next possible bit pointer

i_bitflags(3 downto 2)  <=  aluflags(3) & aluflags(2); -- B.S.
i_bitflags(1)           <=  '1' WHEN i_bitptrnew(5)='1' OR s1val(5)='1' ELSE -- B.S.
                            '0';            -- B.S. Carry Flag if new pointer points
i_bitflags(0)           <=  aluflags(0);    -- B.S. to bitsnew (2nd longword in FIFO)


bitpointer : PROCESS(ck,clr)            -- B.S.  Handle the bitpointer
                                        -- B.S.  (used with GetBits)
  BEGIN                                 -- B.S.

    IF clr = '1' THEN                   -- B.S.
        i_bitptr <= (others => '0');    -- B.S.

    ELSIF (ck'EVENT AND ck = '1') THEN
        IF en3='1' AND p3iv='1' AND p3condtrue='1' AND p3i = getbits THEN           -- B.S.
check if GetBits command is in stage 3
        -- B.S. wait if stage 3 is had been allowed to complete to actually latch
        -- B.S. new bit pointer (i.e. internal write stage 4)
          i_bitptr <= i_bitptrnew(4 DOWNTO 0);   -- B.S. use new bit pointer
        ELSE                            -- B.S.
          i_bitptr <= i_bitptr  ;       -- B.S.         use old bit pointer
        END IF;                         -- B.S.

    END IF;                             -- B.S.

END PROCESS;                            -- B.S.
show_bits:process(i_bitptr,s1val,i_bitsold,i_bitsnew,i_bits_shifted) -- B.S.

  BEGIN -- B.S.

    -- B.S. shift operation
      case i_bitptr(4 DOWNTO 0) IS -- B.S.
          when    "00000" =>      i_bits_shifted <= i_bitsold;
-- B.S. 0
          when    "00001" =>      i_bits_shifted <= i_bitsold(30 DOWNTO 0) & i_bitsnew(31);
-- B.S. 1
          when    "00010" =>      i_bits_shifted <= i_bitsold(29 downto 0) & i_bitsnew(31
downto 30);     -- B.S. 2
          when    "00011" =>      i_bits_shifted <= i_bitsold(28 downto 0) & i_bitsnew(31
downto 29);     -- B.S. 3
          when    "00100" =>      i_bits_shifted <= i_bitsold(27 downto 0) & i_bitsnew(31
downto 28);     -- B.S. 4
          when    "00101" =>      i_bits_shifted <= i_bitsold(26 downto 0) & i_bitsnew(31
downto 27);     -- B.S. 5
          when    "00110" =>      i_bits_shifted <= i_bitsold(25 downto 0) & i_bitsnew(31
downto 26);     -- B.S. 6
          when    "00111" =>      i_bits_shifted <= i_bitsold(24 downto 0) & i_bitsnew(31
downto 25);     -- B.S. 7
          when    "01000" =>      i_bits_shifted <= i_bitsold(23 downto 0) & i_bitsnew(31
downto 24);     -- B.S. 8
          when    "01001" =>      i_bits_shifted <= i_bitsold(22 downto 0) & i_bitsnew(31
downto 23);     -- B.S. 9
          when    "01010" =>      i_bits_shifted <= i_bitsold(21 downto 0) & i_bitsnew(31
downto 22);    -- B.S. 10
          when    "01011" =>      i_bits_shifted <= i_bitsold(20 downto 0) & i_bitsnew(31
downto 21);    -- B.S. 11
          when    "01100" =>      i_bits_shifted <= i_bitsold(19 downto 0) & i_bitsnew(31
downto 20);    -- B.S. 12
          when    "01101" =>      i_bits_shifted <= i_bitsold(18 downto 0) & i_bitsnew(31
downto 19);    -- B.S. 13
          when    "01110" =>      i_bits_shifted <= i_bitsold(17 downto 0) & i_bitsnew(31
downto 18);    -- B.S. 14
          when    "01111" =>      i_bits_shifted <= i_bitsold(16 downto 0) & i_bitsnew(31
downto 17);    -- B.S. 15
          when    "10000" =>      i_bits_shifted <= i_bitsold(15 downto 0) & i_bitsnew(31
downto 16);    -- B.S. 16
```

```
          when    "10001" =>      i_bits_shifted <= i_bitsold(14 downto 0) & i_bitsnew(31
downto 15);    -- B.S. 17
          when    "10010" =>      i_bits_shifted <= i_bitsold(13 downto 0) & i_bitsnew(31
downto 14);    -- B.S. 18
          when    "10011" =>      i_bits_shifted <= i_bitsold(12 downto 0) & i_bitsnew(31
downto 13);    -- B.S. 19
          when    "10100" =>      i_bits_shifted <= i_bitsold(11 downto 0) & i_bitsnew(31
downto 12);    -- B.S. 20
          when    "10101" =>      i_bits_shifted <= i_bitsold(10 downto 0) & i_bitsnew(31
downto 11);    -- B.S. 21
          when    "10110" =>      i_bits_shifted <= i_bitsold(9  downto 0) & i_bitsnew(31
downto 10);    -- B.S. 22
          when    "10111" =>      i_bits_shifted <= i_bitsold(8  downto 0) & i_bitsnew(31
downto 9);    -- B.S. 23
          when    "11000" =>      i_bits_shifted <= i_bitsold(7  downto 0) & i_bitsnew(31
downto 8);    -- B.S. 24
          when    "11001" =>      i_bits_shifted <= i_bitsold(6  downto 0) & i_bitsnew(31
downto 7);    -- B.S. 25
          when    "11010" =>      i_bits_shifted <= i_bitsold(5  downto 0) & i_bitsnew(31
downto 6);    -- B.S. 26
          when    "11011" =>      i_bits_shifted <= i_bitsold(4  downto 0) & i_bitsnew(31
downto 5);    -- B.S. 27
          when    "11100" =>      i_bits_shifted <= i_bitsold(3  downto 0) & i_bitsnew(31
downto 4);    -- B.S. 28
          when    "11101" =>      i_bits_shifted <= i_bitsold(2  downto 0) & i_bitsnew(31
downto 3);    -- B.S. 29
          when    "11110" =>      i_bits_shifted <= i_bitsold(1  downto 0) & i_bitsnew(31
downto 2);    -- B.S. 30
          when    others  =>      i_bits_shifted <= i_bitsold(0)           & i_bitsnew(31
downto 1)    ;-- B.S. 31
      end CASE;

    -- B.S. cut upper bits
      case s1val(4 DOWNTO 0) IS          -- B.S. s1val contains length
          when    "00000" =>    IF s1val(5)='0' THEN
                                  i_bits <= (others => '0');
-- B.S. 0
                                ELSE
                                  i_bits <= i_bits_shifted(31 downto 0);
-- B.S. 32
                                END IF ;
          when    "00001" =>      i_bits <= "00000000000000000000000000000000" &
i_bits_shifted(31);              -- B.S. 1
          when    "00010" =>      i_bits <= "00000000000000000000000000000000" &
i_bits_shifted(31 downto 30);   -- B.S. 2
          when    "00011" =>      i_bits <= "00000000000000000000000000000000" &
i_bits_shifted(31 downto 29);  -- B.S. 3
          when    "00100" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 28); -- B.S. 4
          when    "00101" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 27);-- B.S. 5
          when    "00110" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 26);    -- B.S. 6
          when    "00111" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 25);   -- B.S. 7
          when    "01000" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 24);  -- B.S. 8
          when    "01001" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 23); -- B.S. 9
          when    "01010" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31
downto 22);-- B.S. 10
          when    "01011" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto
21);    -- B.S. 11
          when    "01100" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto
20);    -- B.S. 12
          when    "01101" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto
19);   -- B.S. 13
          when    "01110" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto
18);  -- B.S. 14
          when    "01111" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto
17);    -- B.S. 15
          when    "10000" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto 16);
-- B.S. 16
          when    "10001" =>      i_bits <= "00000000000000000000000000000000" & i_bits_shifted(31 downto 15);
-- B.S. 17
```

```
          when    "10010" =>      i_bits <= "00000000000000" & i_bits_shifted(31 downto 14);
-- B.S. 18
          when    "10011" =>      i_bits <= "0000000000000" & i_bits_shifted(31 downto 13); --
B.S. 19
          when    "10100" =>      i_bits <= "000000000000" & i_bits_shifted(31 downto 12);
-- B.S. 20
          when    "10101" =>      i_bits <= "00000000000" & i_bits_shifted(31 downto 11);    -
- B.S. 21
          when    "10110" =>      i_bits <= "0000000000" & i_bits_shifted(31 downto 10);    --
B.S. 22
          when    "10111" =>      i_bits <= "000000000" & i_bits_shifted(31 downto 9);    --
B.S. 23
          when    "11000" =>      i_bits <= "00000000" & i_bits_shifted(31 downto 8);    --
B.S. 24
          when    "11001" =>      i_bits <= "0000000" & i_bits_shifted(31 downto 7);    --
B.S. 25
          when    "11010" =>      i_bits <= "000000" & i_bits_shifted(31 downto 6);    -- B.S.
26
          when    "11011" =>      i_bits <= "00000" & i_bits_shifted(31 downto 5);    -- B.S.
27
          when    "11100" =>      i_bits <= "0000" & i_bits_shifted(31 downto 4);    -- B.S.
28
          when    "11101" =>      i_bits <= "000" & i_bits_shifted(31 downto 3);    -- B.S. 29
          when    "11110" =>      i_bits <= "00" & i_bits_shifted(31 downto 2);    -- B.S. 30
          when    others  =>      i_bits <= "0" & i_bits_shifted(31 downto 1);    -- B.S. 31
      end CASE;

END PROCESS;                            -- B.S.
```

B.5 Opcode Definition für Befehlssatzerweiterung

```
-- BS
-- BS Additional Instructions:
-- BS ADDCLIPL, ADDCLIPH, INTERPOLATEL, INTERPOLATEH, ALIGN1, ALIGN2, ALIGN3

 constant addclipl    : std_ulogic_vector(opcodsz downto 0):= "10110"; -- BS  ADDCLIPL
 constant addcliph    : std_ulogic_vector(opcodsz downto 0):= "10111"; -- BS  ADDCLIPH
 constant interpolatel: std_ulogic_vector(opcodsz downto 0):= "11000"; -- BS  INTERPOLATEL
 constant interpolateh: std_ulogic_vector(opcodsz downto 0):= "11001"; -- BS  INTERPOLATEH
 constant align1      : std_ulogic_vector(opcodsz downto 0):= "11010"; -- BS  ALIGN1
 constant align2      : std_ulogic_vector(opcodsz downto 0):= "11011"; -- BS  ALIGN2
 constant align3      : std_ulogic_vector(opcodsz downto 0):= "11100"; -- BS  ALIGN3
 constant getbits     : std_ulogic_vector(opcodsz downto 0):= "11101"; -- BS  GETBITS
 constant showbits    : std_ulogic_vector(opcodsz downto 0):= "11110"; -- BS  SHOWBITS
 constant putbits     : std_ulogic_vector(opcodsz downto 0):= "11111"; -- BS  PUTBITS
```

Anhang C Blockschaltbilder

C.1 Blockschaltbild ARC-Multiplexer

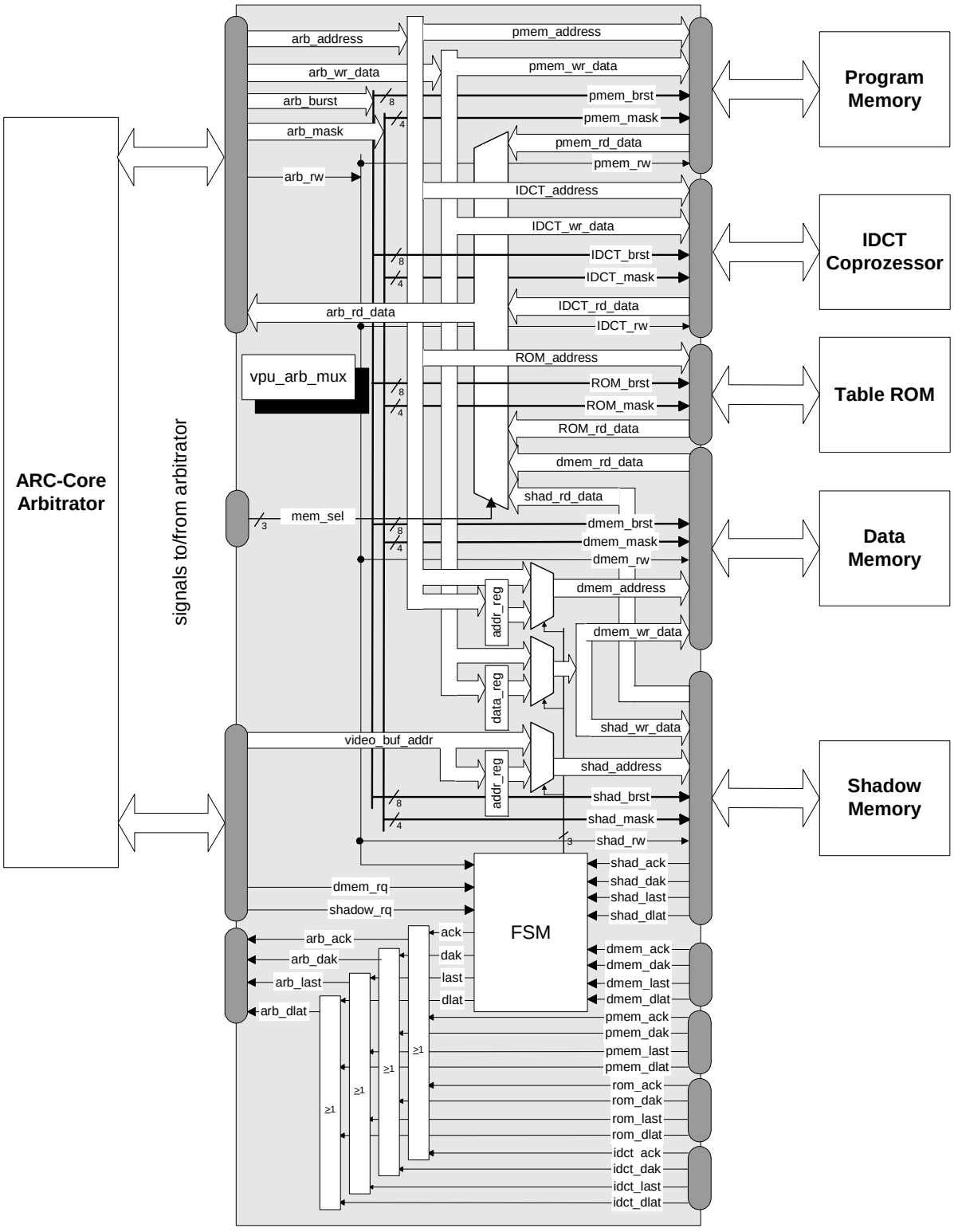


Abbildung 82 Blockschaltbild ARC-Multiplexer

C.2 Blockschaltbild ARC-Arbitrator

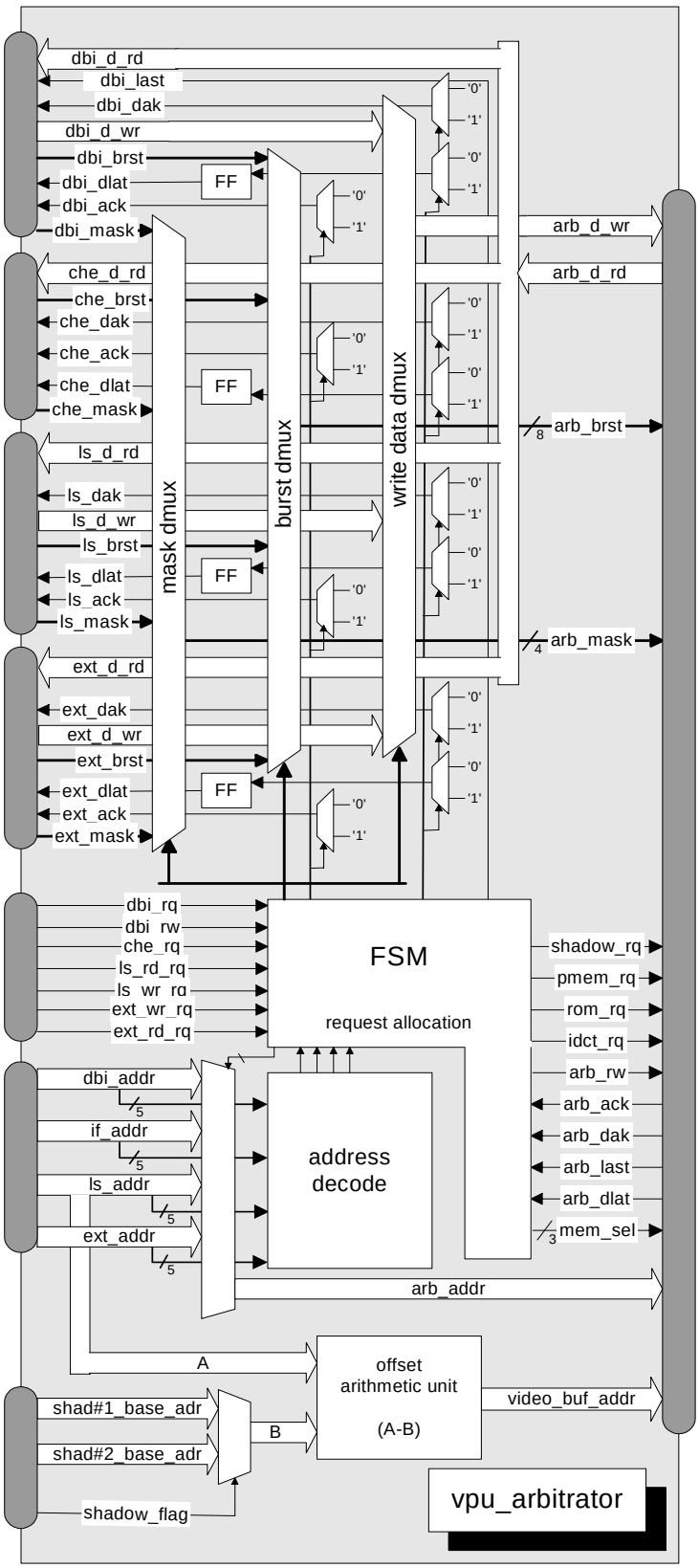


Abbildung 83 Blockschaltbild ARC-Arbitrator

C.3 Blockschaltbild Gesamtsystem

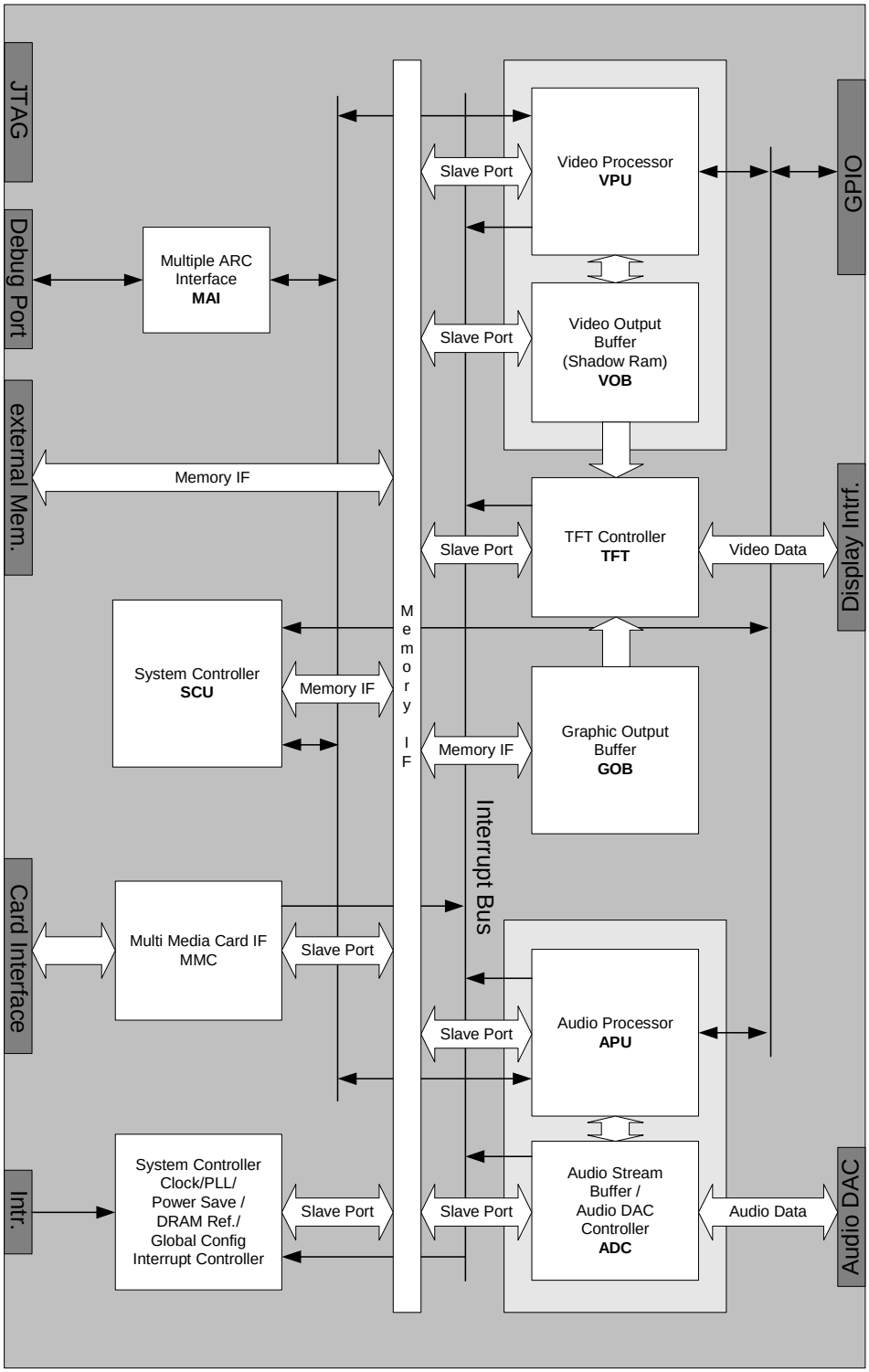


Abbildung 84 Blockschaltbild eines SOCs für mobile Multimediaanwendungen

C.4 Blockschaltbild IDCT-Coprozessor

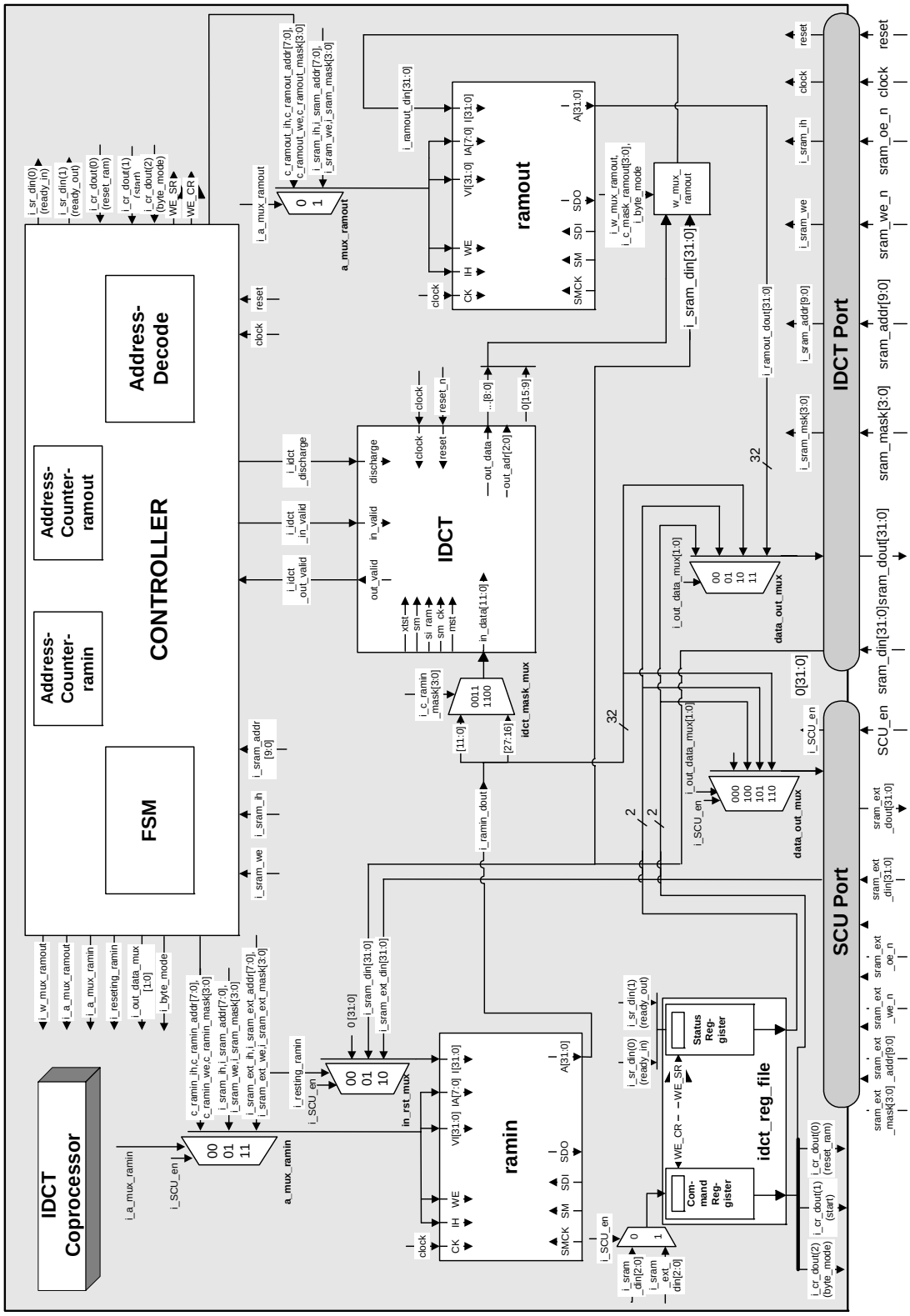


Abbildung 85 Blockschaltbild des IDCT-Coprozessors

Anhang D Assembler Quellcode Argonaut RISC-Core

D.1 Quelltext Interpolationsfilter

```
;-------------| VerFilter8 |-------------------
      .global     VerFilter8
VerFilter8:
;-254:/*****************************************************************************
;-255:Proto:
;-256:void VerFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
;-257:Author:    HHI
;-258:Purpose:   Gets prediction data with vertical half-pixel interpolation.
;-259:
;-260:Abstract:  The texture data of the reference vop is vertical interpolated and
;-261:           copied to the outputbuffer (des).
;-262:
;-263:In:        src: pointer to texture data, height: vertical blocksize, src_stride: stride
of texture data,
;-264:           rc: rounding control
;-265:InOut:
;-266:Out:       des: pointer to output buffer
;-267:Modified:
;-268:*****************************************************************************/
;-269:void VerFilter8(UInt8 *src,Int32 height,Int32 src_stride,UInt8 rc,UInt8 *des)
;-270:{
      st    %blink, [%sp, 4]
      st    %fp, [%sp]
      mov   %fp, %sp
      sub   %sp, %sp, 32
      st    %r13, [%sp, 16]
      st    %r14, [%sp, 20]
      st    %r15, [%sp, 24]
      st    %r16, [%sp, 28]
      mov   %r13, %r0
      mov   %r15, %r2
      mov   %r16, %r3
      mov   %r14, %r4
;-271:   Int8 i;
;-272:   UInt8 CONST *p, *s;
;-273:   UInt8 a, b;
;-274:
;-275:    DEBUG_SIGN(function_entry_VerFilter8);
; -------- BEGIN _Asm inline code
 ; 0x2000063 ist eine Konstante
 sr 0x2000063, [0x13]
; -------- END _Asm inline code
;-276:
;-277:    rc = 1 - rc;
      sub   %r0, 1, %r16
      extb  %r3, %r0
;-278:    s = (UInt8 CONST *) src;
;-279:    p = (UInt8 CONST *) src + src_stride;
      add   %r4, %r15, %r13
;-280:    src_stride=src_stride-8;
      sub   %r1, %r15, 8
;-281:
;-282: for( i = 0; i < 8; i++ )
      sub   %r2, %r2, %r2
.LN099.4:
;-283:  {
;-284:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13]
      ldb   %r5, [%r4]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14]
;-285:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 1]
      ldb   %r5, [%r4, 1]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
```

```
      stb   %r0, [%r14, 1]
;-286:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 2]
      ldb   %r5, [%r4, 2]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14, 2]
;-287:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 3]
      ldb   %r5, [%r4, 3]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14, 3]
;-288:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 4]
      ldb   %r5, [%r4, 4]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14, 4]
;-289:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 5]
      ldb   %r5, [%r4, 5]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14, 5]
;-290:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 6]
      ldb   %r5, [%r4, 6]
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14, 6]
;-291:        a = *s++;b = *p++;*des++ = (rc + a + b) >>1;
      ldb   %r0, [%r13, 7]
      add   %r6, %r13, 8
      ldb   %r5, [%r4, 7]
      add   %r4, %r4, 8
      add   %r0, %r0, %r3
      add   %r0, %r0, %r5
      asr   %r0, %r0
      stb   %r0, [%r14, 7]
      add   %r14, %r14, 8
;-292:        s+=src_stride;p+=src_stride;
;-293:  }
      add   %r0, %r2, 1
      extb  %r2, %r0
      sub.f 0, %r2, 8
      add   %r4, %r4, %r1     ;
      blt.d .LN099.4
      add   %r13, %r6, %r1    ; delay
;-294:
;-295:    DEBUG_SIGN(function_exit_VerFilter8);
; -------- BEGIN _Asm inline code
 ; 0x3000063 ist eine Konstante
 sr 0x3000063, [0x13]
; -------- END _Asm inline code
;-296:}
      ld    %r13, [%sp, 16]
      ld    %r14, [%sp, 20]
      ld    %r15, [%sp, 24]
      ld    %blink, [%fp, 4]
      ld    %r16, [%sp, 28]   ;
      j.d   [%blink]
      ld.a  %fp, [%sp, 32]    ; delay

      .type VerFilter8, @function
      .size VerFilter8, . - VerFilter8
```

D.2 Quelltext Bewegungskompensation

```
;-39:/*****************************************************************************
;-40:Proto:
;-41:void SavePredPlusCoeff(UInt8 *pred, Int16 *coef, Int32 stride, UInt8 *des)
;-42:Author:    HHI
;-43:Purpose:   Saves the clipped combination of prediction and inverse transformed texture
values to the frame buffer.
;-44:
;-45:Abstract:  The inverse transformed texture values are added to the prediction data,
clipped
;-46:           and stored in the appropriate frame buffer.
;-47:
;-48:In:        pred: pointer to prediction data, src: pointer to texture data, stride: stride
of texture data
;-49:InOut:
;-50:Out:       des: pointer to frame buffer
;-51:Modified:
;-52:*****************************************************************************/
;-53:/* save the clipped combination of prediction and dct data */
;-54:void SavePredPlusCoeff(UInt8 *pred, Int16 *coef, Int32 stride, UInt8 *des)
;-55:{
      st    %blink, [%sp, 4]
      st    %fp, [%sp]
      mov   %fp, %sp
      sub   %sp, %sp, 32
      st    %r13, [%sp, 16]
      st    %r14, [%sp, 20]
      st    %r15, [%sp, 24]
      st    %r16, [%sp, 28]
      mov   %r14, %r0
      mov   %r13, %r1
      mov   %r16, %r2
      mov   %r15, %r3
;-56:    Int32 i,j;
;-57:    UInt8 l_u8;
;-58:    Int16 m_16;
;-59:    Int32 n_32;
;-60:    UInt8 CONST *p;
;-61:    Int16 CONST *c;
;-62:
;-63:    DEBUG_SIGN(function_entry_SavePredPlusCoeff);
; -------- BEGIN _Asm inline code
 ; 0x200005f ist eine Konstante
 sr 0x200005f, [0x13]
; -------- END _Asm inline code
;-64:    p = pred;
;-65:    c = coef;
;-66:    stride-=8;;
      sub   %r4, %r16, 8
;-67:    for (j=0;j<8;j++)
      ld    %r2, [%gp, clip_zero@sda]
      mov   %r3, 8
.LN095.4:
;-68:    {
;-69:        for( i = 0; i < 8; i++ )
      mov   %lp_count, 8
      ; ZD LOOP:
      lp    .LP095.8
;-70:        {
;-71:            l_u8 = *p++;
      ldb   %r1, [%r14]
      add   %r14, %r14, 1
;-72:            m_16 = *c++;
      ldw.x %r0, [%r13]
;-73:            n_32 = (Int32) l_u8 + (Int32) m_16;
      add   %r0, %r0, %r1
;-74:            *des++ = clip( n_32 );
      ldb   %r0, [%r2, %r0]
      add   %r13, %r13, 2     ;
      stb   %r0, [%r15]
      add   %r15, %r15, 1
;-75:        }
.LP095.8:
```

```
;-76:        des+=stride;
;-77:
;-78:    }
      sub.f %r3, %r3, 1
      add   %r15, %r15, %r4   ;
      bpnz  .LN095.4
;-79:
;-80:    DEBUG_SIGN(function_exit_SavePredPlusCoeff);
; -------- BEGIN _Asm inline code
 ; 0x300005f ist eine Konstante
 sr 0x300005f, [0x13]
; -------- END _Asm inline code
;-81:
;-82:}
      ld    %r13, [%sp, 16]
      ld    %r14, [%sp, 20]
      ld    %r15, [%sp, 24]
      ld    %blink, [%fp, 4]
      ld    %r16, [%sp, 28]   ;
      j.d   [%blink]
      ld.a  %fp, [%sp, 32]    ; delay

      .type SavePredPlusCoeff, @function
      .size SavePredPlusCoeff, . - SavePredPlusCoeff
```

D.3 Quelltext Bitstromroutinen

```
;-68:UInt32           ShowBits( UInt32 u_length)
;-69:{
      st    %blink, [%sp, 4]
      st    %fp, [%sp]
      mov   %fp, %sp
      sub   %sp, %sp, 20
      st    %r13, [%sp, 16]
      mov   %r13, %r0
;-70:    UInt32 u_word ;
;-71:
;-72:    DEBUG_SIGN(function_entry_ShowBits);
; -------- BEGIN _Asm inline code
 ; 0x200002d ist eine Konstante
 sr 0x200002d, [0x13]
; -------- END _Asm inline code
;-73:
;-74:    u_word=u32_curword;
      ld    %r3, [%gp, u32_curword@sda]
;-75:    if (u_length<=bits_valid)
      ld    %r2, [%gp, bits_valid@sda]
      sub   %r1, 32, %r13
      sub.f 0, %r13, %r2
;-76:    {
;-77:        u_word = u_word >> (32-u_length);
      lsr.ls      %r13, %r3, %r1    ;
      bls   .LN47.6
;-78:    }
;-79:    else        /* second word */
;-80:    {
;-81:        u_word = *u32p_buff;
      ld    %r0, [%gp, u32p_buff@sda]
      ld    %r0, [%r0]
;-82:        u32_curword |= (u_word >> bits_valid);
      lsr   %r0, %r0, %r2
      or    %r0, %r0, %r3
      st    %r0, [%gp, u32_curword@sda]
;-83:        u_word = u32_curword >> (32-u_length);
      lsr   %r13, %r0, %r1
.LN47.6:
;-84:
;-85:    }
;-86:
;-87:    DEBUG_SIGN(function_exit_ShowBits);
; -------- BEGIN _Asm inline code
 ; 0x300002d ist eine Konstante
 sr 0x300002d, [0x13]
; -------- END _Asm inline code
;-88:    return u_word;
;-89:}
      mov   %r0, %r13
      ld    %blink, [%fp, 4]
      ld    %r13, [%sp, 16]   ;
      j.d   [%blink]
      ld.a  %fp, [%sp, 20]    ; delay

      .type ShowBits, @function
      .size ShowBits, . - ShowBits



;-------------| FlushBits |-------------------
      .global     FlushBits
FlushBits:
;-90:/*****************************************************************************
;-91:Proto:          void FlushBits(UInt32 length);
;-92:Author:
;-93:Purpose:
;-94:Abstract:
;-95:In:
;-96:InOut:
;-97:Out:
```

```
;-98:Modified:
;-99:*************************************************************************/
;-100:
;-101:void     FlushBits(UInt32 u_length)
;-102:{
      st    %blink, [%sp, 4]
      st    %fp, [%sp]
      mov   %fp, %sp
      sub   %sp, %sp, 20
      st    %r13, [%sp, 16]
      mov   %r13, %r0
;-103:    Int32 d;
;-104:
;-105:    DEBUG_SIGN(function_entry_FlushBits);
; -------- BEGIN _Asm inline code
 ; 0x200002f ist eine Konstante
 sr 0x200002f, [0x13]
; -------- END _Asm inline code
;-106:
;-107:    d=u_length-bits_valid;
      ld    %r2, [%gp, bits_valid@sda]
      sub.f %r1, %r13, %r2
;-108:
;-109:    if (d<=0)
      nop   ;
      bpnz  .LN49.5
;-110:    {
;-111:       u32_curword <<= u_length;
      ld    %r0, [%gp, u32_curword@sda]
      asl   %r0, %r0, %r13
      st    %r0, [%gp, u32_curword@sda]
;-112:       bits_valid-=u_length;
      b.d   .LN49.7
      sub   %r0, %r2, %r13    ; delay
.LN49.5:
;-113:    }
;-114:    else        /* second word */
;-115:    {
;-116:       u32_curword = *u32p_buff;
      ld    %r0, [%gp, u32p_buff@sda]
      ld    %r2, [%r0]
;-117:       u32p_buff++;
      add   %r0, %r0, 4
      st    %r0, [%gp, u32p_buff@sda]
;-118:       u32_curword <<= d;
      asl   %r0, %r2, %r1
      st    %r0, [%gp, u32_curword@sda]
;-119:       bits_valid= 32-d;
      sub   %r0, 32, %r1
.LN49.7:
      st    %r0, [%gp, bits_valid@sda]
;-120:
;-121:    }
;-122:
;-123:    DEBUG_SIGN(function_exit_FlushBits);
; -------- BEGIN _Asm inline code
 ; 0x300002f ist eine Konstante
 sr 0x300002f, [0x13]
; -------- END _Asm inline code
;-124:
;-125:}
      ld    %blink, [%fp, 4]
      ld    %r13, [%sp, 16]   ;
      j.d   [%blink]
      ld.a  %fp, [%sp, 20]    ; delay

      .type FlushBits, @function
      .size FlushBits, . - FlushBits
```

Anhang E MPEG-4-Testsequenzen

Bezeichnung:	Container
Bildgröße:	352 x 288
Farbformat:	YUV 4:2:0
Quantisierung:	8 Bit
Länge der Sequenz:	300 Frames
Bildänderung:	niedrig

Bezeichnung:	Foreman
Bildgröße:	352 x 288
Farbformat:	YUV 4:2:0
Quantisierung:	8 Bit
Länge der Sequenz:	300 Frames
Bildänderung:	mittel

Bezeichnung:	Stefan
Bildgröße:	352 x 288
Farbformat:	YUV 4:2:0
Quantisierung:	8 Bit
Länge der Sequenz:	300 Frames
Bildänderung:	hoch

Anhang F Flussdiagramme MPEG-4-Decodersoftware

F.1 Decoder Hauptschleife

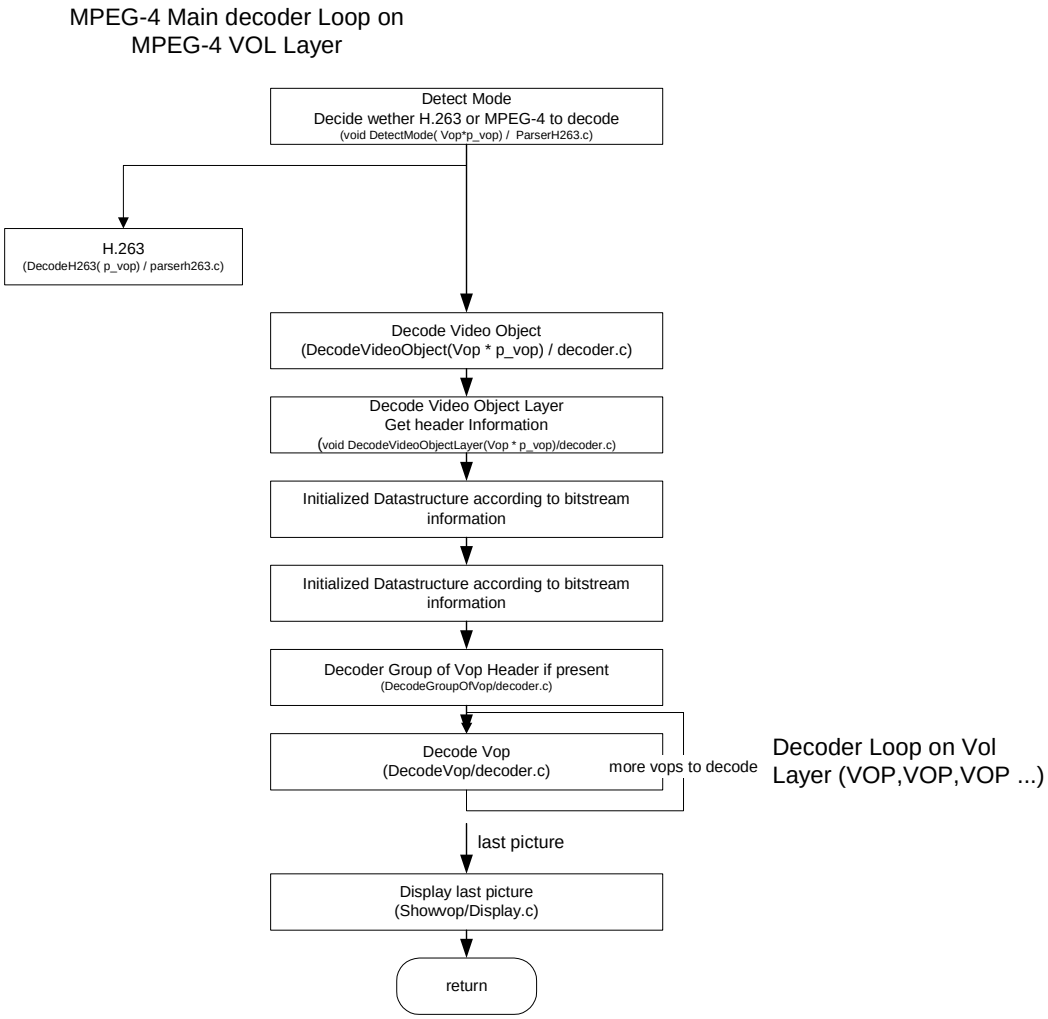


Abbildung 86 Flussdiagramm der MPEG-4-Decoder-Hauptschleife

F.2 Decoderschleife Bilddecodierung

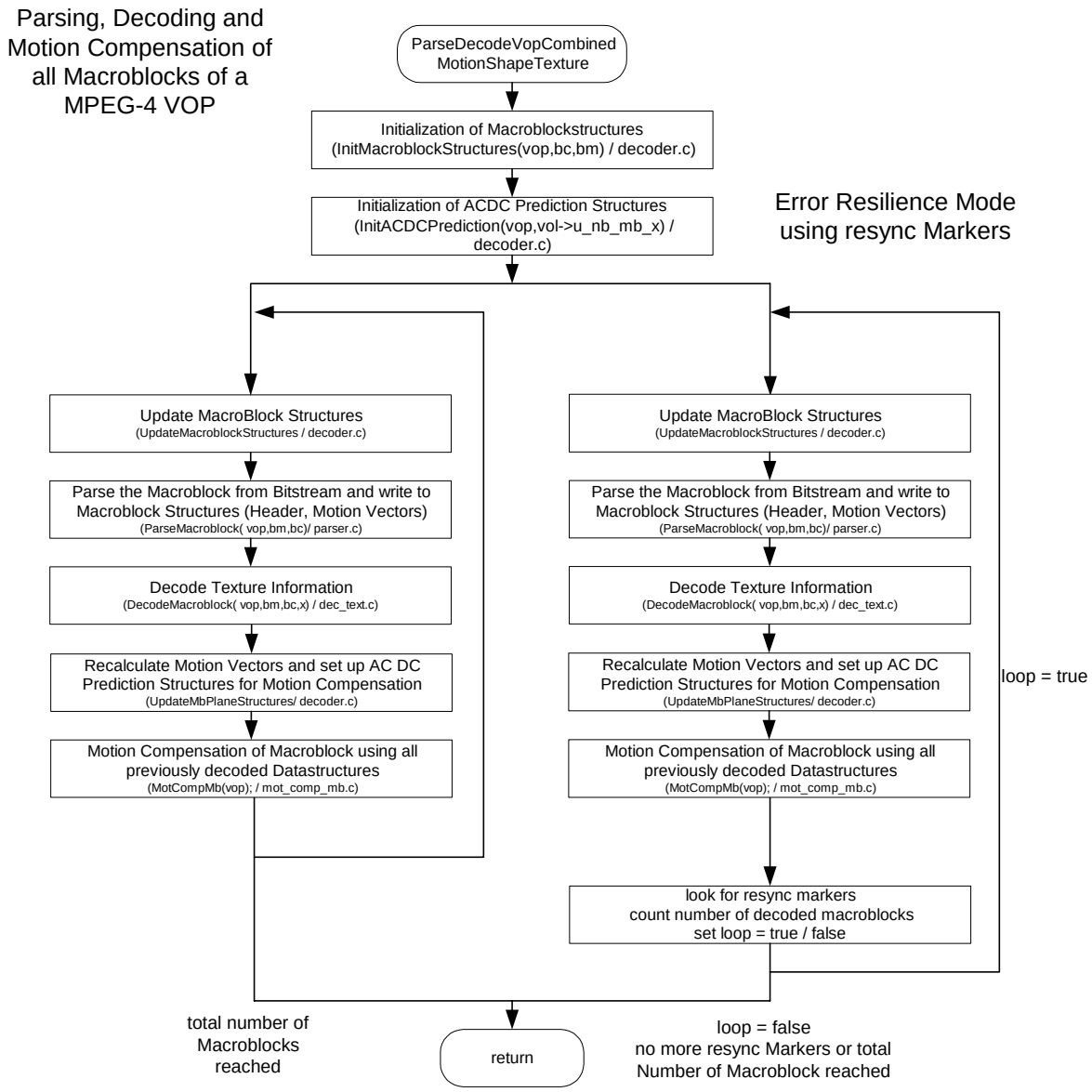


Abbildung 87 Flussdiagramm MPEG-4-Decoder, Decodierung eines Bildes

F.3 Decodierung Makroblockheader

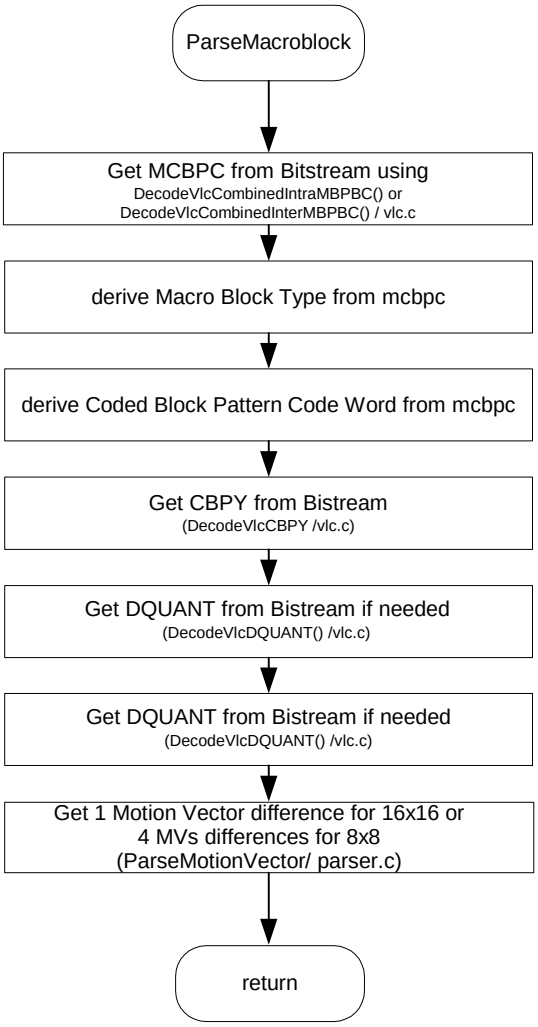


Abbildung 88 Flussdiagramm MPEG-4-Decoder, Makroblockverarbeitung

F.4 Decodierung Texturinformation eines Makroblocks

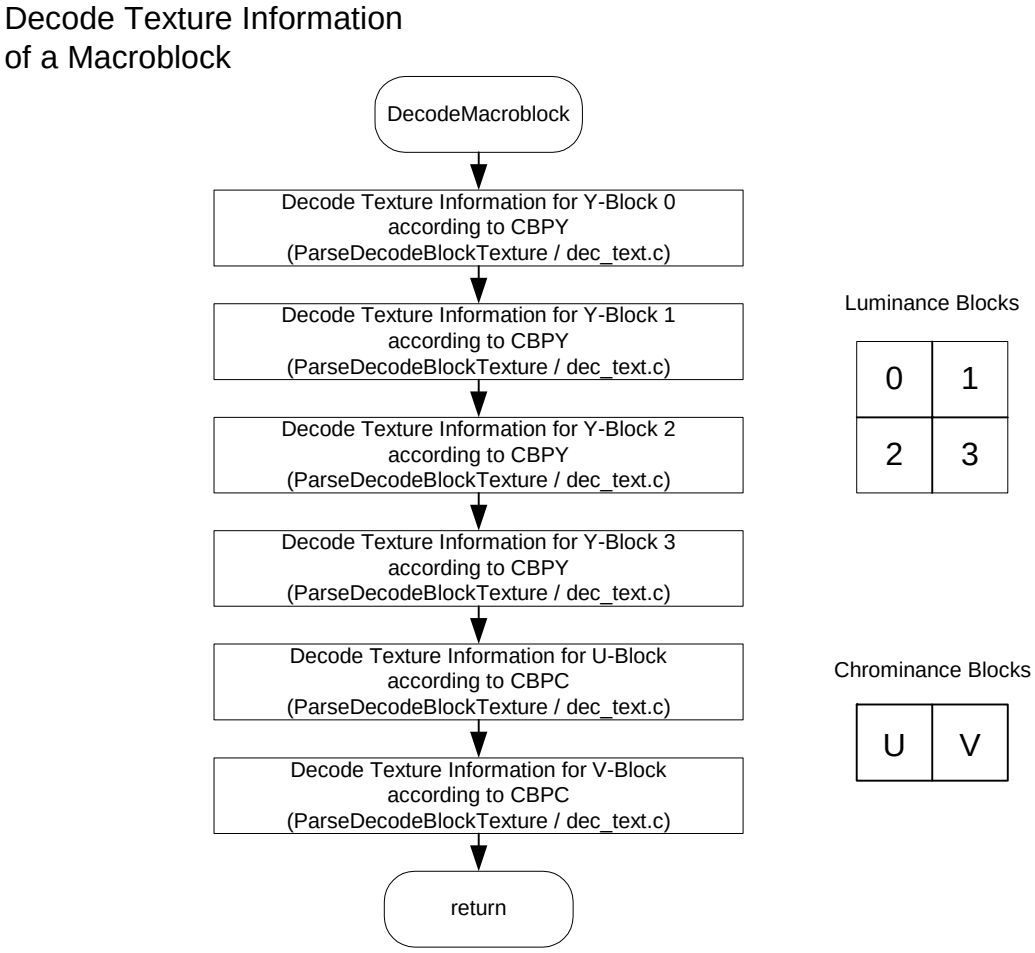


Abbildung 89 Flussdiagramm MPEG-4-Decoder, Texturdecodierung eines Makroblocks

Anhang G Tabellen

G.1 MPEG-4 Visual Profiles

Visual Tools \ Profiles	Simple	Core	Main	Simple Scalable	N-bit
Basic • I-VOP, P-VOP • AC/DC Prediction • 4-MV, Unrestricted MV	✓	✓	✓	✓	✓
Error resilience • Slice Resynchronization • Data Partitioning • Reversible VLC	✓	✓	✓	✓	✓
Short Header	✓	✓	✓		✓
B-VOP		✓	✓	✓	✓
P-VOP with OBMC (Texture)					
Method 1/Method 2 Quantization	✓		✓		✓
P-VOP based temporal scalability • Rectangular • Arbitrary Shape		✓	✓		✓
Binary Shape		✓	✓		✓
Grey Shape			✓		
Interlace			✓		
Sprite			✓		
Temporal Scalability (Rectangular)				✓	
Spatial Scalability (Rectangular)				✓	
N-Bit					✓

Tabelle 9 Features der verschiedenen MPEG-4 Visual Profiles

G.2 Standardisierte Bitstrombeschreibung exemplarisch

block(i) {	**No. of bits**	**Mnemonic**
last = 0		
if(!data_partitioned && (derived_mb_type == 3 || derived_mb_type == 4)) {		
if(short_video_header == 1)		
intra_dc_coefficient	8	uimsbf
else if (use_intra_dc_vlc == 1) {		
if (i<4) {		
dct_dc_size_luminance	2-11	vlclbf
if(dct_dc_size_luminance != 0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_luminance > 8)		
marker_bit	1	bslbf
} else {		
dct_dc_size_chrominance	2-12	vlclbf
if(dct_dc_size_chrominance !=0)		
dct_dc_differential	1-12	vlclbf
if (dct_dc_size_chrominance > 8)		
marker_bit	1	bslbf
}		
}		
}		
if (pattern_code[i])		
while (! last)		
DCT coefficient	3-24	vlclbf
}		

Tabelle 10 Exemplarischer Auszug aus dem ISO 14496-2-Standard

Literaturverzeichnis

[1] B. Stabernack, H. Richter, „A Multi Media Streaming Framework for Mobile Applications, A first Approach", 12 th International Packetvideo Workshop (PV2002), April 24th-26 in Pittsburgh PA, USA

[2] ETSI (European Telecommunication Standards Institute), DVB-T, EN 300 744

[3] ETSI (European Telecommunication Standards Institute), DVB- H System Specification normative draftEN 302 304

[4] „Performance analysis and low power VLSI implementation of DVB-T receiver", http://www.signal.uu.se/Courses/Semabstracts/ofdm2.pdf

[5] Ohm J.-R., Bildsignalverarbeitung für Multimediasysteme, Vorlesungsskript Bildsignalverarbeitung, TU Berlin, Oktober 1999

[6] ISO/IEC 13818-2: Information Technology - Generic Coding of Moving Pictures and Associated Audio - Part 2: Video, 1994

[7] ISO/IEC FDIS 14496-2: Coding of Audio-Visual Objects - Part 2: Visual, Oct.1998

[8] J. Mitchell, W. Pennebaker, C. Fogg, D. LeGall, „MPEG video compression standard", Chapman & Hall, 1996

[9] R. Schäfer, T. Sikora, „Digital video coding standards and their role in video communications", Proceedings of the IEEE, Vol. 83, No. 6, June 1995, pp. 907-924

[10] Atul Puri et al., „Multimedia Systems, Standards and Networks", Marcel Dekker Inc., 2000

[11] N. Brady, "MPEG-4 Standardized Methods for the Compressions of Arbitrarily Shaped Video Objects", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 9, No. 8, December 1999

[12] G.R. Hu and T.K. Tan, Panasonic Singapore Laboratories (Matsushita), „Simplification of the DC/AC Inverse Prediction and Inverse Quantization for MPEG-4 Texture Decoding", ISO/IEC JTC1/SC29/WG11, MPEG97/M3623

[13] Hong Jiang, Minhua Zhou, Weiping Li, and Xuemin Chen, „Complexity Analysis of MPEG-4 AC Prediction", ISO/IEC JTC1/SC29/WG11 MPEG00/M5668, March 2000

[14] Weiping Li, Optivision Inc, „Experiment on Coding Efficiency Gain of AC Prediction", ISO/IEC JTC1/SC29/WG11 MPEG00 / M5671 March, 2000

[15] R. Talluri, „Error-Resilient Video Coding in the ISO MPEG-4 Standard", IEEE Communications Magazine, Vol.36 No. 6 June 1998,pp.112-119

[16] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, Andrew Wolfe, „Challenges to Combining General-Purpose and Multimedia Processors", IEEE Computer,Vol. 30, No. 12, December 1997, pp. 33-37

[17] Yen-Kuang Chen et al., „Media Applications on Hyper-Threading Technology", Intel Technology Journal Q1, 2002, Vol.6 Issue 1

[18] J. L. Hennesy, D. A. Patterson, „Computer Architecture. A Quantitative Approach", Second Edition, Morgan Kauffmann Publishers, Inc., 1996

[19] R. Bhargava, L.K. John, B.L. Evans, R. Radhakrishnan, „Evaluating MMX Technology Using DSP and Multimedia Applications", Proc. 31st IEEE Intl. Symp. On Microarchitecture (MICRO-31), Dallas, Texas, 1998, pp. 37-46

[20] Ronald E. Curry, „IA-64 Architecture delivers new Levels of Performance and Scalability", Intel Corp., 1998.

[21] Motorola, „DSP56300 Family Manual", Motorola, 1995

[22] N. Seshan, „High velociTI processing", IEEE Signal Processing Magazine, March 1998, pp. 86-101

[23] Texas Instruments, „TMS320C62X/C67X Technical Brief", Texas Instruments Inc.., April 1998, Literature Number SPRU197B.

[24] Texas Instruments, „TMS320DMP642 Digital Media Processor Product Preview", Texas Instruments INC., July 2002, Literature Number SPRS200.

[25] Gerrit A. Slavenburg, S. Rathnam, H. Dijkstra, „The Trimedia TM-1 PCI VLIW Media Processor", Proceedings Hot Chips 8, August 1996

[26] F. Sijstermans, „Trimedia CPU64 Architecture", Proceedings Microprocessor Forum 98, San Jose, California, October 12-15 1998

[27] C. Basoglu, W. Lee, J. O`Donnell, „The MAP1000A VLIW Mediaprocessor", IEEE Micro, Vol.20, pp.48-59, 2000

[28] C. Basoglu, W. Lee, J. O`Donnell, „The Equator MAP-CA DSP: An End-To-End Broadband Signal Processor VLIW", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 12, No. 8, August 2002

[29] Tensilica Inc. Corp., „Tensilica Unveils Next-Generation Xtensa LX Processor Core", Global Signal Processing Times, June 2004

[30] Masaharu Imai, Yoshinori Takeuchi, Akichika Shiomi, Jun Sato, Akira Kitajima, „An Application Specific Processor Development Environment: ASIP Meister", Technical Report in IEICE, ICD2002-113, vol. 102, No. 401, pp. 39-44, October. 2002

[31] H.-J. Stolberg, M. Berekovic, P. Pirsch, H. Runge, „Implementing the MPEG-4 Advanced Simple Profile for Streaming Video Applications", Proceedings International Conference on Multimedia and EXPO (ICME2001), August 2001, pp. 297-300

[32] Sang-Joon Nam et. al., „DIVA: Dual Issue VLIW Architecture with Media Instructions for Image Processing", IEEE Transactions on Consumer Electronics, Vol. 45, No.1 February 1999

[33] F. Charot, G. Le Fol, P. Lemmonier, C. Wagner, R. Barzic, C. Bouville, „Toward Hardware Building Blocks for Software-Only Real-Time Video Processing: The MOVIE Approach", IEEE Transaction for Circuits and Systems for Video Technology, Vol. 9, No. 6, September 1999

[34] C.G. Zhou, L. Khon, D. Rice, I. Kabir, A. Jabbi, X.-P. Hu, „MPEG Video Decoding with the UltraSPARC Visual Instruction Set", Digest of Papers COMPCON Spring 95, p470 - 475, IEEE, March 1995

[35] S. Rathnam, G. Slavenberg, „An Architectural Overview of the Programmable Multimedia Processor, TM-1," Proc. of COMPCON '96, Santa Clara, Calif., 1996, pp. 319-326

[36] Thomas M Conte, Wen-mei W. Hwu, „Benchmark Characterization", IEEE Computer, Vol.24, No1, January 1991, pp 48-56

[37] N.T. Slingerland, A.J. Smith, „Design and Characterization of the Berkeley Multimedia Workload", University of California at Berkeley Technical Report CSD-00-1122, December. 2000

[38] Paul C. H. Lee, „Performance Analysis of an MPEG-2 Audio/Video Player", IEEE Transactions on Consumer Electronics, Vol. 45, No. 1, February 1999

[39] Jeffrey Osier, Free Software Foundation, „The Gnu Profiler", Free Software Foundation, Inc. January. 1993

[40] P. M. Kuhn and W. Stechele, „Complexity Analysis of the Emerging MPEG-4 Standard as a Basis for VLSI Implementation", Vol. SPIE3309 VisualCommun. Image Process., San Jose, Jan. 1998, pp. 498–509

[41] M. Mattavelli and S. Brunetton, „Implementing Real-time Video on Multimedia Processors by Complexity Prediction Techniques", IEEE Transactions on Consumer Electronics, Vol. 44, No. 3, Aug. 1998, pp. 760–767.

[42] Argonaut RISC Assembler Programming Users Manual

[43] M. Takahashi et al., „A 60 mW MPEG4 video codec using clustered voltage scaling with variable supply voltage scheme", ISSCC98 International Solid State Circuits Conference, 1998, pp. 36-37

[44] M. Berekovic, G. Meyer, Y. Guo and P. Pirsch, „A Multimedia RISC Core for efficient Bitstream Parsing and VLD", Multimedia Hardware Architectures, Proceedings of SPIE, January. 1998, pp. 479–488.

[45] R. Hoffer, „Architekturen für die Codierung mit variablen Codewortlängen auf der Basis inhaltsbasierter Speicher", Fortschrittsberichte VDI Reihe 10 Nr. 471, Düsseldorf VDI Verlag 1997

[46] M. Berekovic, H.-J. Stolberg, M.B. Kulaczewski, P. Pirsch, H. Möller, H. Runge, J. Kneip, and B. Stabernack, „Instruction Set Extensions for MPEG-4 Video," J. VLSI Sign. Process. Syst., Vol. 23, No. 1, Oct. 1999, pp. 27–50.

[47] B. Stabernack, et al., „Set Of Complexity Profiles For MPEG-4 VM-8- And VCD-V08 Compliant Video Decoder Implementations", M3182, Contribution to ISO/MPEG

[48] R.Saito et al., „VLSI Implementaions of a variable-Length Decoding Processor for Real Time Video", Proc. IEEE Workshop on Visual Signal Processing Conference, S. 87-90, 1991

[49] M.T. Sun, „VLSI Architecture and Implementation of a High-Speed Entropy Decoder", Proc, IEEE Int. Sympsosium on Circuits an Systems, Vol. 1, S. 200 –203, 1991

[50] Seung P. Kim, „A Concurrent Memory-Efficient VLC Decoder for MPEG Applications", IEEE Transactions on Consumer Electronics, Vol 42, No. 3, August 1996

[51] Shaw-Min Lei, „An Entropy Coding System for Digital HDTV Applications",IEEE Transactions Circuit Syst., Vol 1 März 1991

[52] M. T. Sun, S.M. Lei, „A High Speed Entropy Decoder für HDTV", Proc. IEEE Custom Integrated Circuits Conference, S 26.1.1-26.3.4,1992

[53] Y. Zhang, Kai K. Ma Qindong Yao, „A Software/Hardware Co-Design Methodology for Embedded Microporcessor Core Design", IEEE Transactions on Consumer Electronics, Vol. 45, No. 4, November 1999

[54] Chen, W.-H., Smith, C. H., and Fralick, S.C., „A fast computational algorithm for the discrete cosine transform", IEEE Transactions Communications. Vol COM-25 No. 9, Sept. 1977

[55] IEEE, „Standard Specifications for the Implementation of 8x8 Inverse Discrete Cosine Transformation", IEEE Std 1180-1090, 1991

[56] Z. Wang, „Fast Algorithms for the Discrete W-Transform and for the Discrete Fourier Transform", IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-32, No.4, August 1984, pp.803-816

[57] Byeong Lee, „An New Algorithm to Compute the Discrete Cosine Transform", IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-32, No.6, Dezember 1984, pp.1243-1245

[58] N. Suehiro and M Hatori, „Fast Algorithms for the DFT and Sinusoidal Transforms", IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-34, No.3, June 1986, pp.642-644

[59] H.S. Hou, „A Fast Recursive Algorithm for Computing the Discrete Cosine Transform", IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-35, No.10, October 1987, pp.1455-1466

[60] Christoph Loeffler, Adriaan Ligtenberg and George S. Moschytz, „Practical Fast 1-D DCT Algorithms with 11 Multiplications", Preceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP '89),1989, pp.988-991

[61] Seehyun Kim and Wonyong Sung, „Optimum Wordlength Determination of 8x8 IDCT Architectures Conforming to the IEEE Standard Specifications", Twenty-Ninth Annual Asilomar Conference on Signals, Systems, and Computers, Oct. 1995

[62] Seehyun Kim, Wonyong Sung, „Finite wordlength effects analysis and wordlength optimization of a multiplier-adder based 8 X 8 2D-IDCT architecture", ISCAS '96

[63] Intel Corp., „A fast precise implementation of 8x8 discrete co-sine transform using the streaming SIMD extensions and MMX instructions", MMX Technology Application Notes AP-922., http://developer.intel.com

[64] Intel Corp., „Integrated Performance Primitives for Intel Architecture Reference Manual", http://developer.intel.com

[65] Thou-Ho Chen, „A Cost-Effective 8x8 2-D IDCT Core Processor with folded Architecture", IEEE Transactions on Consumer Electronics, Vol. 45, No. 2, May 1999

[66] E. Murata, M. Ikekawa, and I. Kuroda, „Fast 2D IDCT Implementation with Multimedia Instructions for a Software MPEG-2 Decoder", Proc. IEEE Int. Conf. Acoust. Speech Sign. Process. (ICASSP), Vol. 5, May 1998, pp. 3105–3108.

[67] T. Masaki, Y. Morimoto, T. Onoye, I.Shirakawa, „VLSI Implementation of Inverse Discrete Cosine Transformer and Motion Compensator for MPEG2 HDTV Video Decoding", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 5, October 1995

[68] Heiko Hübert, Benno Stabernack, Henryk Richter, „Tool-Aided Performance Analysis and Optimization of an H.264 Decoder for Embedded Systems", The Eighth IEEE International Symposium on Consumer Electronics (ISCE 2004), Reading, England, September 2004

[69] H.Krahn, B.Stabernack, „HDTV durch die digitale Hintertür", Zeitschrift Elektronik, Heft 18/1997, S. 58 ff.

[70] W. Yan, et al, „Dokumentation zur Inversen Diskreten Cosinus Transformation (IDCT)", Heinrich Hertz Institut, März 1998

[71] Fujitsu Semiconductor, „Semicustom CMOS Embedded Array CE81 Series Datasheet DS06-20110-4e", Fujitsu Semiconductor Datasheet.

[72] B. Erol, F. Kossentini, H. Alnuweiri, „Efficient Coding and Mapping Algorithms for Software-Only Real-Time Video Coding at Low Bit Rates", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 10, No. 6, September 2000

[73] B.Stabernack, M.Talmi, „Real time implementation of an DSP-based MPEG-4 Videodecoder", ICSPAT, Oktober 1999, Orlando, FL USA

[74] Intel Corp., „Using the RDTSC Instruction for Performance Monitoring", Intel Pentium II Processor Application Notes, http://developer.intel.com

[75] B.Stabernack, „Verfahren und Anordnung zur Ermittlung der Decodierungskomplexität von blockbasiert codierten Videodatenströmen sowie Verwendung dieses Verfahrens und ein entsprechendes Computerprogramm-Erzeugnis zur Regelung der Taktfrequenz", DE-Patenanmeldung, amtl. Aktz. 103 13 149.3

[76] M. Berekovic, H.-J. Stolberg, P. Pirsch, „Multi-Core System-On-Chip Architecture for MPEG-4 Streaming Video", Transactions on Circuits and Systems for Video Technology (CSVT), Vol. 12, No. 8, August 2002, pp. 688-699.

[77] B. Stabernack, Martin Köhler, Matthias Reißmann, Gerd von Cölln, „A processor based System on a Chip Design for Mobile Multi Media applications", IEEE International Symposium on Consumer Electronics ISCE 02, TU-Illmenau, 24 –26 September 2002

[78] B. Stabernack, H. Richter, „Instruction Set extensions for Mobile Video Applications on Embedded RISC Processors", IEEE International Conference on Consumer Electronics ICCE 2005, Januar 2005