Entwurf und Evaluierung mehrfädig superskalarer Prozessortechniken im Hinblick auf Multimedia

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

Von

Ulrich Sigmund

aus Freiburg i.Br.

Tag der mündlichen Prüfung: 23. Mai 2000

Erster Gutachter: Prof. Dr. Theo Ungerer

Zweiter Gutachter: Prof. Dr. Uwe Brinkschulte

Ich versichere wahrheitsgemäß, die Dissertation bis auf die dort angegebenen Hilfen selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Ulrich Sigmund

Inhaltsverzeichnis

Zusammenfassung

In meiner Dissertation wird ein mehrfädig superskalarer Prozessor im Hinblick auf Multimediaverarbeitung entworfen und mit einem Multimediaprogramm als Last getestet. Neu ist hierbei die Kombination von Multimediainstruktionen mit der mehrfädigen Prozessortechnik, einem zusätzlichen On-Chip-RAM, einer realen Last aus dem Multimediabereich (einem MPEG-2-Dekoder) sowie die Verwendung von handoptimiertem Assemblercode. Verschiedene Konfigurationen des Prozessors werden untersucht, um potentielle Flaschenhälse zu identifizieren, und schließlich zu eliminieren. Es zeigt sich, dass ein mehrfädiger Prozessor mit Multimediaerweiterungen in der Lage ist, einen um einen Faktor drei höheren IPC zu erreichen, als ein gleichwertiger Prozessor, der nur einen Kontrollfaden gleichzeitig ausführt.

1 Einleitung

Heutige Mikroprozessoren nutzen Befehlsebenenparallelität durch eine vielstufige Prozessor-Pipeline und durch die Superskalartechnik. Mit Multimediainstruktionen kommt häufig auch noch SIMD-Parallelität durch die Verwendung von Teilwortarithmetik zum Einsatz. Ein superskalarer Prozessor kann aus einem sequentiellen Befehlsstrom mehrere Befehle pro Takt den Verarbeitungseinheiten zuordnen und ausführen. Viele heutige Prozessoren können bis zu vier Befehle pro Takt zuordnen, in naher Zukunft wird bis zu achtfache Zuordnung erwartet. Die Befehlsebenenparallelität in einem sequentiellen Befehlsstrom ist jedoch insbesondere für nichtnumerische Anwendungen sehr beschränkt.

Neuere Studien zeigen die Grenzen der Prozessorleistung bereits bei heutigen Superskalarprozessoren. IPC-(instructions per cycle)-Werte zwischen 0,6 und 2,0 werden für SPEC95-Benchmark Programme auf einem PentiumPro-Prozessor berichtet. Messungen kommerzieller Datenbanktransaktionssysteme auf einem symmetrischen Multiprozessor mit vier PentiumPro-Prozessoren zeigen einen IPC von 0,29.

Diese geringe Auslastung ist auch in Multimediaanwendungen häufig zu sehen. Gründe dafür liegen in den häufigen Cache-Fehlzugriffen, im Ausnutzen vorhandener Parallelität durch die Multimediainstruktionen und in der Ungleichverteilung der Instruktionsgruppen im Programmablauf (rechenintensive Programmteile lassen sich kaum oder nur unter aufwendigen Codevervielfältigungen mit speicherintensiven Teilen überlappen).

Ein Lösungsansatz, um die Verarbeitungsleistung zu erhöhen, besteht jedoch darin, zusätzlich auch grobkörnige Parallelität auf Thread- und Prozeßebene zu nutzen. Dies kann durch Multiprozessor-Chips oder durch die mehrfädige Prozessortechnik geschehen. Ein mehrfädiger Prozessor speichert die Kontexte mehrerer Kontrollfäden in separaten Registersätzen auf dem Prozessor-Chip und kann Befehle verschiedener Kontrollfäden gleichzeitig (oder zumindest überlappt parallel) in der Prozessor-Pipeline ausführen. Ein mehrfädig superskalarer Prozessor (simultaneous multithreading) kann Befehle mehrerer Kontrollfäden sogar in einem Taktzyklus den Verarbeitungseinheiten zuordnen.

Mehrfädig superskalare Prozessoren wurden bereits mehrfach mit SPEC92- und SPEC95-Benchmarkprogrammen und kürzlich mit Lastprogrammen eines Datenbanktransaktionssytems evaluiert. Die Simulationen zeigen, dass die mehrfädig superskalare Prozessortechnik eine bis zu dreifache Performancesteigerung gegenüber vergleichbaren Superskalarprozessoren erreichen kann.

Die Evaluierung durch Multimedialastprogramme ist bisher nur in eingeschränkter Form erfolgt. Insbesondere wurden keine Multimediainstruktionen verwendet. Die Ergebnisse sind deshalb nicht als repräsentativ zu werten. Auch die Verwendung compilierter Benchmarks ist nur bedingt sinnvoll, da heutige Compiler kaum oder nicht zur Erzeugung von Routinen mit Multimediainstruktionen geeignet sind. In unserem Ansatz wurde deshalb ein in Assembler geschriebener und handoptimierter MPEG-2-Dekodierungsalgorithmus verwendet.

MPEG-2 ist derzeit das wichtigste Videokompressionsverfahren, und wird stellvertretend für diese Anwendungsklasse betrachtet. Der MPEG-2 Algorithmus ist im ISO/IEC 13818-2 Standard beschrieben und kommt für Digitales Fernsehen (DVB), DVD und HDTV zum Einsatz. MPEG-2 kombiniert stark rechenintensive Programmschritte wie die inverse diskrete Cosinustransformation (IDCT), speicherzugriffsorientierte Schritte in der Bewegungskompensation und irregulären Programmfluss in der Huffman-Dekodierung. Diese Mischung ist für viele Multimediaalgorithmen repräsentativ. Basierend auf unserer Analyse des MPEG-2-Dekodieralgorithmus, liegen nur etwa 15,5% der Anweisungen im nichtparallelisierbaren Teil der Dekodier-Pipeline. Somit ergibt sich eine durch Mehrfädigkeit maximal erreichbare Beschleunigung um einen Faktor 6,5 zu einem einfädigen Prozessor.

Ziel dieser Arbeit ist die Spezifikation und Evaluierung eines mehrfädigen Prozessors mit Multimediaerweiterungen. Dazu wird ein taktgenauer Simulator entwickelt und verschiedene Prozessorkonfigurationen entworfen und analysiert. In einem ersten Abschnitt der Simulationen wird mit auch unrealistischen Annahmen versucht, eine Prozessorkonfiguration mit maximaler Leistung zu ermitteln. Diese wird dann in einem zweiten Abschnitt durch die Verwendung realistischer Ressourcenzuordnungen zu einer realisierbaren Prozessorkonfiguration vereinfacht. Die Prozessorkonfigurationen werden jeweils mit verschieden großer Befehlszuordnungsbreite und Anzahl der Ausführungskontexte simuliert. Somit lassen sich die Gewinne durch die Mehrfädigkeit mit denen der die durch die Superskalarität des Prozessors entstehen vergleichen, und der Nutzen abwägen.

Weiterhin wird auch untersucht wie sich der simulierte Prozessor bei verschiedenen Varianten der Sprungvorhersage und des Cache-Managements verhält.

Die restlichen Kapitel dieser Arbeit sind folgendermaßen organisiert: Kapitel 2 und 3 beschreiben den Stand der Technik und Forschung in den Bereichen der Prozessoren mit Multimediaerweiterungen und der Mehrfädigkeit von Prozessoren. Kapitel 4 erläutert die Vor- und Nachteile eines mehrfädigen Prozessors mit Multimediaerweiterungen, und stellt den Prozessorentwurf dieser Arbeit vor. In Kapitel 5 wird die MPEG-2-Dekodierung als Last auf ihre potenzielle Parallelität untersucht, sowie Möglichkeiten zur Nutzung dieser Parallelität durch einen mehrfädigen Prozessor aufgezeigt. Der Aufbau der Untersuchten Prozessorfamilie wird im Kapitel 6 erläutert. Kapitel 7 beschreibt die Simulationslast, den mehrfädigen MPEG-2-Dekoder. Die Simulationsdurchführung und die daraus resultierenden Ergebnisse und Prozessorkonfigurationen werden im Kapitel 8 dargestellt. Kapitel 9 enthält schließlich eine Zusammenfassung der Ergebnisse, sowie einen Ausblick auf zukünftige Arbeiten in diesem Gebiet.

Die Arbeit wird durch einen Anhang ergänzt, der die Befehlsstruktur des simulierten Prozessors, sowie die Benutzung des entwickelten Simulators erläutert.

2 Prozessoren mit Multimediaerweiterungen

2.1 Überblick

In diesem Kapitel wird auf die Geschichte und den aktuellen Stand im Bereich der Prozessoren mit Multimediaerweiterungen und spezifischer Multimediaprozessoren[1] eingegangen. Es wird der Unterschied zwischen klassischen Prozessoren, DSP und dedizierten Multimedia Chips aufgezeigt und erläutert. Des weiteren werden die Prinzipien, die hinter den Multimediaerweiterungen dieser Prozessoren liegen, beschrieben. Das Kapitel wird durch einige Erläuterungen zur Verwendung dieser Prozessoren mit Hochsprachenübersetzern ergänzt.

2.2 Motivation

Unter dem Schlagwort „Multimedia“ versteht man heute meist die Bearbeitung von Bild- und Tondaten mit der Hilfe von Rechnern. Da dieser Begriff sehr unspezifisch gedeutet wird, seien hier einige Beispiele genannt.

- Kommunikation, Übertragung von Ton und Bild durch digitale Medien
  - Internet Telephonie, ISDN, Sprache über IP
  - Bildtelephon, Videokonferenzen
  - Digitales Fernsehen, HDTV
  - Video on Demand
- Visualisierung, Erzeugung von Bildern aus abstrakten Daten
  - Virtuelle Realität
  - Medizinische Anwendungen, 3D Darstellung von Schnittbildersequenzen
  - Entwurf von Maschinen, Gebäuden, virtuelles Durchschreiten
  - Computergenerierte Filmsequenzen
  - Computerspiele
- Bild- und Tonbearbeitung, die Veränderung oder Rekomposition von Bildern, Filmen oder Tondokumenten
  - Digitaler Filmschnitt, Nachkolorierung alter Filme
  - Bildbearbeitung in Verlagen, Werbeagenturen

- Restauration von Film- oder Tondokumenten
- Spezialeffekte in Kinofilmen

- Bild- und Tonarchivierung, die Speicherung und Wiedergabe von Bilder, Filmen und Tondokumenten auf und von digitalen Medien
  - Bildarchivierung in Verlagen
  - Medizinische Anwendungen, wie die Archivierung von Röntgenbildern
  - Speicherung und Wiedergabe von Kinofilmen oder Amateurfilmen auf digitalen Bändern und CDs

All diese Möglichkeiten werden durch digitale Multimedia-Hardware und Software ermöglicht. Dies findet sowohl im Bereich typischer Computersysteme, wie PCs oder Workstations als auch im Bereich eingebetteter Systeme, wie zum Beispiel Videokameras statt. In der Vergangenheit war dies aufgrund des enormen Datenaufkommens und der häufig vorhandenen Echtzeitbedingungen vor allem eine Domäne spezieller Bausteine und digitaler Signalprozessoren (DSP). Durch die Einführung sogenannter Multimediaprozessoren oder durch die Erweiterung bestehender Prozessorfamilien um Multimediaerweiterungen wird dies immer stärker ein Bereich, in dem auch Standardprozessoren diese Voraussetzungen erfüllen können.

2.3 Rück- und Umblick

2.3.1 Entwicklungsgeschichte

In der Vergangenheit wurden viele der im vorherigen Abschnitt genannten Aufgaben durch Spezialprozessoren erledigt, die als zusätzliche Komponente in den benutzten Systemen vorhanden sein mussten. Im Zuge der höheren Integration werden häufig Komponenten, die oft gemeinsam in einem System benutzt werden, zu neuen Standardkomponenten zusammengefügt. Dies konnte Anfang der 90er Jahre bei Bilddarstellungskomponenten beobachtet werden, die als zusätzliche Funktionen auch die einfachen Formen der Bildbearbeitung, wie sie für ein Fenstersystem benötigt werden, (Graphikbeschleuniger) übernahmen. Vorher wurde dies meist durch eine Darstellungskomponente und eine Bearbeitungskomponente erledigt (entweder ein dedizierter Graphikbeschleuniger wie z.B. TMS34010 [76] oder TMS34020 [77] oder durch den Systemprozessor selbst). Gegen Ende der 90er Jahre werden nun auch Elemente der dreidimensionalen, perspektivischen Bilddarstellung in Graphikchips übernommen. Ähnliche Entwicklungen sind auch bei allgemeinen Prozessoren zu beobachten, die mittlerweile neben dem eigentlichen Prozessorkern auch Caches, Speicherverwaltungseinheit und Fließkommaeinheit beinhalten.

[1] Da mittlerweile alle großen Prozessorfamilien über Multimediaerweiterungen verfügen, und reine Multimediaprozessoren fast nicht mehr existieren, wird der Begriff Multimediaprozessor und Prozessor mit Multimediaerweiterungen im folgenden Text häufig synonym verwendet.

Durch die Integration mehrerer Einheiten auf einem Chip ergibt sich nicht nur der Vorteil, dass die Anzahl der Systemkomponenten sinkt, häufig können die einzelnen kombinierten Elemente auch einfacher gehalten werden, da sie leichter zusammenarbeiten können. Meist werden auch komplexe Operationen in mehrere kleinere zerlegt, die dann flexibel kombiniert werden können. Dies wird durch eine bessere Verfügbarkeit der Komponenten für den Prozessor erreicht, so dass durch das Sinken des Verwaltungsaufwandes, der Durchsatz an sinnvollen Operationen ansteigt. Dies ist sehr deutlich bei Fließkommaeinheiten zu betrachten, bei denen in vielen Fällen im Zuge der Integration in den Hauptprozessor die transzendenten mathematischen Operationen aus dem Befehlssatz eliminiert und durch Software ersetzt wurden (MC68040). Da aber gleichzeitig (auch durch die Integration) die Geschwindigkeit bei den einfachen Befehlen um einen Faktor Fünf bis Sechs zunahm, wurden auch die transzendenten Operationen schneller durchgeführt.

Diese Gedanken und Erfahrungen standen Anfang der 90er Jahre Pate, als die ersten Prozessorbefehlssätze um spezielle Multimediabefehle erweitert wurden (MC88110 [49]). Diese Erweiterungen waren in erster Linie zur Beschleunigung der Videoausgabe auf einfachen Bilddarstellungssystemen ausgerichtet. Es fanden sich aber bereits viele der in aktuellen Multimediabefehlssätzen enthaltenen Erweiterungen, wie Teilwortarithmetik, Saturierungsarithmetik und Maskierung. Da die Graphikbeschleunigung aber viel günstiger und effizienter als Erweiterung der Darstellungskomponente erfolgen konnte, haben sich diese Erweiterungen zum damaligen Zeitpunkt nicht durchgesetzt.

Mit der starken Verbreitung von Multimediaanwendungen gegen Mitte der 90er Jahre, entstanden viele Hardwareerweiterungen [91][46] für PCs um die sehr aufwändigen Operationen zur Videokompression und Dekompression in Echtzeit erledigen zu können. Gleichzeitig stieg auch der Bedarf an Leistung im 3D-Visualisierungs- und Kommunikationsbereich an, so dass auch hier zusätzliche Hardwareerweiterungen entstanden. In der Mitte des Jahrzehnts wurden etwa zum gleichen Zeitpunkt von verschiedenen Herstellern sowohl multimediaspezifische Befehlserweiterungen für Standardprozessoren (INTEL MMX [2][3], HP MAX [44], SPARC VIS [24][39]), als auch allgemeine Multimediaprozessoren (Philips TriMedia [5][60], MPACT [18][14]) angekündigt. Der Gedanke hinter beiden Systemen war mehr Leistung zu geringerem Preis. Bei echten Multimediaprozessoren ergeben sich die geringeren Kosten dadurch, dass eine einzelne Erweiterung alle bisherigen Erweiterungen abdeckt (ist in diesem Prozessor auch noch eine weitere Komponente des Systems, wie z.B. das Bilddarstellungssystem integriert, entstehen kaum zusätzliche Kosten).

Diese Multimediaprozessoren bzw. um Multimediafunktionen erweiterte Standardprozessoren verfügen zusätzlich zu den Standardoperationen eines Prozessors und den zur Videobearbeitung benötigten Operationen auch über von digitalen Signalprozessoren übernommenen Funktionen zur Audiobearbeitung (Fixkommaarithmetik, „Multiply-Accumulate“, [28]) und zur dreidimensionalen, perspektivischen Bilderzeugung. Zusätzlich zu diesen Systemen entstanden auch aus dem Bereich der DSPs Prozessoren, die durch Übernahme von Konzepten wie Teilwortarithmetik und RISC Kern für Videobearbeitung in die Klasse der Multimediaprozessoren eingeordnet werden müssen (Siemens TriCore, Hyperstone). Als eine weitere Klasse von Chips, die in den Bereich der Multimediaprozessoren drängen, müssen auch Graphikbeschleuniger gesehen werden. Diese verfügen heute bereits über multi-

mediaspezifische Funktionseinheiten und in einigen Fällen auch über einen internen RISC Prozessor, der einen Großteil der Videooperationen übernehmen wird.

Eine andere Implementierungsmöglichkeit der meist sehr regulären Multimediaroutinen besteht in konfigurierbaren Logikbausteinen, die je nach benötigtem Algorithmus mit verschiedenen Konfigurationen geladen werden [23]. Dies erlaubt eine sehr hohe Ausführungsgeschwindigkeit, hat aber den Nachteil, dass es sich nur schlecht in die bestehende Prozessor- und Softwarestrukturen einfügen lässt.

War am Anfang der Entwicklung noch MPEG-1-Dekodierung die Meßlatte für diese Prozessoren, so hat sie sich in den letzten Jahren auf MPEG-2-Dekodierung erhöht. Das nächste zu erwartende Ziel dürfte sich im Rahmen von HDTV, sowie der MPEG-2-Videokompression und der dreidimensionalen, perspektivischen Bilddarstellung finden.

Analysen verschiedener Multimediaroutinen auf einem bestehenden Prozessor mit Multimediaerweiterungen haben gezeigt, dass sich durch die Verwendung von Multimediainstruktionen Leistungssteigerungen um einen Faktor von 1,1 bis 4,2 ergeben [59].

2.3.2 Vergleich bestehender Systeme

Bezeichnung	Jahr	Gruppe	Teilwort	Parallele Multiplikation	Bilddarstellung	Fließkomma	Saturierung	VLIW/Superskalar	Peak Operations per Cycle (mit mindestens acht Bit)	Max MCycles per Second	Spezielle 3D Erweiterungen	Spezielle Video Erweiterungen
TMS34020	1989	Graphik-beschl.	1,2,4,8,16,32 in 32	Nein	Nein	Nein	Ja	1x	32	28	Nein	Nein
MC88110	1991	µP+MM	4,8,16,32 in 64	Nein	Nein	Ja	Ja	2x SS	9	50	Nein	Nein
HP 712	1994	µP+MM	16 in 32	Nein	Nein	Ja	Ja	2x	8	80	Nein	Nein
TMS320C80	1994	DSP	8, 16 in 32	Nein	Nein	Ja	Ja	4x3+2	40	50	Nein	Ja
UltraSPARC™	1995	µP+MM	8 in 32 oder 16,32 in 64	Ja	Nein	Ja	Nein	4x SS	18	167	Nein	Ja
Intel P55C	1996	µP+MM	8,16,32 in 64	Ja	Nein	Ja	Ja	2x SS	16	233	Nein	Nein
Hyperstone												
PA 8000	1996	µP+MM	16 in 64	Nein	Nein	Ja	Ja	4x	18		Nein	Nein
MPACT 1	1996	MMP	9, 18, 36 in 72	Ja	Ja	Nein	Ja	5xVLIW	~350	62.5	Nein	Ja
MicroUnity MediaProcessor	1996	MMP	1, 2, 4, 8, 16, 32,64 in 64 oder 128	Ja	Nein	Nein	Ja	5xMT CbC[2]	40	200	Nein	Nein
INTEL PII	1997	µP+MM	8,16,32 in 64	Ja	Nein	Ja	Ja	3x SS	19	450	Nein	Nein
INTEL PIII	1999	µP+MM	8,16,32 in 64 32 in 128	Ja	Nein	Ja	Ja	3x SS				
MPACT 2	1997	MMP	9, 18, 36 in 72	Ja	Ja		Ja	5xVLIW		125	Ja	Ja
Philips TriMedia	1997	MMP	8, 16 In 32	Ja	Nein	Ja	Ja	5xVLIW	~50		Nein	Ja
ATI Rage PRO	1997	VGA+MM	8 in 64	Nein	Ja	Nein	Ja	?	?	?	Ja	Ja
RIVA DVD	1998	VGA+MM	8, 16 in 128	Ja	Ja	Nein	Ja	?	?	?	Ja	Ja

[41], [36], [31], [44], [90], [24], [39], [2], [3], [4], [1], [5], [60], [74], [73], [75]

2.4 Multimediabefehlserweiterungen

Der sichtbarste und deutlichste Unterschied zwischen klassischen Prozessoren (seien dies nun allgemeine Prozessoren oder DSPs) und Multimediaprozessoren ist durch die Erweiterung der üblichen Befehlssätze um spezifische Multimediabefehle und Ausführungseinheiten gegeben. Generell lassen sich hier zwei Arten von Erweiterungen unterscheiden, solche die für verschiedene multimediaspezifi-

[2] Dieser Prozessor ist mehrfädig angelegt. Von jedem Kontrollfaden kann bei jedem Takt maximal eine Instruktion gestartet werden. Da der Prozessor maximal fünf Kontrollfäden erlaubt, können pro Takt maximal fünf Instruktionen ausgeführt werden.

sche Anwendungen verwendet werden können und solche die auf genau eine Anwendung zugeschnitten sind.

2.4.1 Allgemeine Multimediaerweiterungen

Unter allgemeinen Multimediaerweiterungen seien hier Erweiterungen aufgeführt, die nicht nur eine einzige der multimediaspezifischen Funktionen abdecken, sondern die eine eher bausteinartige, allgemeine Erweiterung des klassischen Prozessormodells beinhalten. Diese Erweiterungen lassen sich deshalb für mehrere verschiedene Aufgaben einsetzen und sind nicht auf einen spezifischen Bereich beschränkt. Dieser Erweiterungstypus findet sich in den meisten Multimediaprozessoren und ist bei den Standardprozessoren mit Multimediaerweiterungen die einzige auftretende Form. Viele dieser Erweiterungen lassen sich durch kleine Ergänzungen zum bestehenden Prozessor erreichen, so dass sie vom Verhältnis zwischen Kosten und Nutzen wesentlich besser erscheinen als spezifische Erweiterungen, die meist völlig neue Ausführungseinheiten benötigen und auch nur beschränkt einsetzbar sind [44].

2.4.1.1 Saturierungsarithmetik

Im Bereich Multimedia wird meist mit geschlossenen Intervallen gearbeitet. Es werden also keine vollen Zahlenbereiche $[-\infty..\infty]$, sondern lediglich Unterbereiche (z.B. $[0..255]$) verwendet, obwohl die darzustellenden Größen meist kontinuierliche physikalische Größen mit beliebigem Wertebereich (z.B. Lichtenergie oder Schalldruck) repräsentieren. Dies hat seine Ursache in den beschränkten Auflösungen und Bereichen von existierenden Digital-zu-Analog- (DAC) und Analog-zu-Digital-Konvertern (ADC). Aufgrund des Aufbaus dieser Schaltungselemente sind diese Intervalle auch typischerweise durch Potenzen von zwei bestimmt. Aus Effizienzgründen werden diese Werte im Speicher auch meist in dieser Form abgelegt. Bei Bilddaten hat sich hier die Form von drei einzelnen acht Bit Werten pro Bildelement (Pixel) durchgesetzt, bei Tondaten werden meist 16 Bit pro Abtastwert verwendet. Wird nun mit diesen Werten gerechnet, so kann es sein, dass das Ergebnis der Rechnung nicht mehr im verwendeten Intervall liegt, sondern größer oder kleiner geworden ist. Bei normalen Prozessoren stehen hier nur zwei Möglichkeiten zur Verfügung: es wird eine Ausnahmebehandlungsroutine aufgerufen, oder aber der Zahlenbereich kippt durch das Weglassen des obersten, nicht mehr im Intervall vorhandenen Bits um. Es handelt sich also um eine Modulo-Arithmetik. Dies kann sehr störende Effekte für die Wahrnehmung der betroffenen Daten haben. Bei Tondaten ist eine deutliches Knacken zu vernehmen, bei Bilddaten kommt es zu störenden Punkten (weiße Punkte in schwarzem Bereich und umgekehrt). Bei der Saturierungsarithmetik wird der Ergebniswert auf das gültige Intervall beschränkt, indem er bei Über- bzw. Unterschreiten des Minimums bzw. Maximums des Intervalls dessen Wert annimmt.

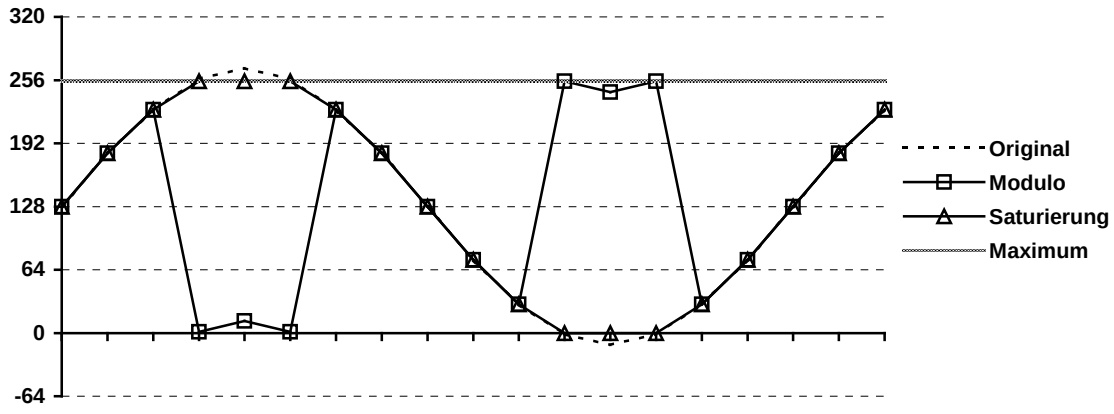


Abbildung 2-1: Auswirkung eines Überlaufes mit und ohne Saturierungsarithmetik

Ein störender Eindruck durch Saturierungsarithmetik ist im Allgemeinen nicht wahrnehmbar, solange sich die Über- bzw. Unterschreitungen des Wertebereiches in engen Grenzen halten. Eine Saturierung lässt sich auch durch eine entsprechende Ausnahmebehandlung erreichen, dies hat jedoch im Unterschied zur automatischen Saturierung einen negativen Effekt auf die Ausführungsgeschwindigkeit.

Saturierung wird üblicherweise nicht auf Intervallen innerhalb eines Datentypwertebereiches, sondern nur auf dem gesamten Wertebereich der Datentypen implementiert. Meist ist jedoch eine Variante für vorzeichenlose und eine für vorzeichenbehaftete Zahlen vorhanden. Dies stellt keine Einschränkung dar, da die bei Multimediaoperationen verwendeten Datentypen entsprechend der vorhandenen Strukturen (z.B. DAC, ADC und Speicher) gewählt sind.

2.4.1.2 Datenparallele Instruktionen

Häufig werden bei Multimediaanwendungen viele Daten mit den gleichen Funktionen behandelt. Da Multimediadatentypen auch meist kleiner sind als die typischen Wortbreiten moderner Prozessoren, werden diese nur zu einem geringen Maße ausgelastet. Hier wurde nun bei vielen Multimediaprozessoren ein Ansatz gewählt, der dem eines SIMD-Rechners entspricht. Es wird die gleiche Funktion auf mehrere Elemente parallel angewendet. Dazu werden die Register und Ausführungseinheiten eines Multimediaprozessors in Teilworte zerlegt, in denen die Berechnung unabhängig stattfinden kann. Eine Zerlegung wird meist in 8, 16 oder 32 Bit-Teilwörtern durchgeführt, da dies die für Multimediadaten typischen Wortbreiten darstellen. Die Aufteilung der Register ist rein symbolisch, sie benötigt keine Repräsentation auf der Implementierungsebene des Prozessors. Auch die Änderungen an den Ausführungseinheiten sind im Allgemeinen minimal, da hier nur die Verbindungen zwischen den einzelnen Teilwörtern (meist dem Übertrag der Addition) unterbrochen werden müssen. Als Beispiel sei hier eine parallele Addition von vier 16 Bit-Datenworten in einem 64 Bit-Prozessorwort unter der Verwendung von Saturierungsarithmetik gezeigt.

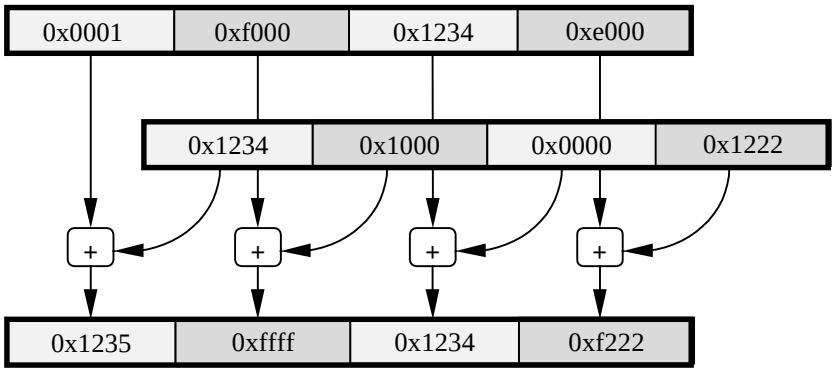


Abbildung 2-2: SIMD-Addition von vier Teilwörtern mit Saturierungsarithmetik

Es ist zu erkennen, dass kein Übertrag über die Teilwortgrenzen hinaus propagiert wird und somit die Teilwörter unabhängig voneinander bearbeitet werden. Neben den einfachen arithmetischen Anweisungen sind häufig auch Schiebeanweisungen auf Teilwortebene implementiert. Logische Anweisungen wie „Und“, „Oder“ oder „exklusives Oder“ benötigen keine spezielle Implementierung, da sie immer teilwortunabhängig ausgeführt werden.

Dieser Teilwortparallelismus erlaubt häufig eine Steigerung der Ausführungsgeschwindigkeit vieler Multimediaanwendungen um einen Faktor von Zwei bis Acht, je nach Größe der Datenwörter, und der Parallelisierbarkeit der Anwendung.

2.4.1.3 Zusätzliche arithmetische Anweisungen

Einige weitere arithmetische Anweisungen haben sich bei Multimediaanwendungen als nützlich erwiesen und wurden deshalb mehreren Multimediabefehlssätzen hinzugefügt. Es handelt sich meist um Befehle, die ohne großen Aufwand in bestehende Rechenwerke eingefügt werden können. Beispiele für diese Anweisungen sind Minimum, Maximum, Distanz oder Mittelwert. Ohne nähere Betrachtung erscheinen diese Anweisungen unnötig, da sie sich durch wenige bereits vorhandene Instruktionen ersetzen ließen. Der Vorteil ergibt sich aber aus der Datenwortbreite. Um zum Beispiel den Mittelwert zweier Zahlen mit n Bit zu berechnen, benötigt man ein Zwischenergebnis mit n+1 Bit. Da Multimediaprozessoren aber meist auf ganzen Teilern der Prozessorregister arbeiten (sprich auf 8, 16, 32 oder 64 Bit) hätte dies eine Halbierung der Parallelität zur Folge. Zusätzlich kommen außerdem noch Instruktionen zur Konvertierung zwischen dem Datenformat und einem erweiterten Format hinzu. Dies sei hier am Beispiel einer Teilroutine aus der MPEG Bewegungskompensation gezeigt. Dieses Codefragment bildet den Mittelwert aus je acht Pixel zweier Referenzbilder und speichert das Ergebnis im Zielbild, unter Verwendung von Intels MMX-Befehlssatz [2], sowie unter Hinzunahme einer Mittelwertanweisung „pavgusb“.

	Standard MMX Befehlssatz		Mit Mittelwertanweisung	
0	movq	mm0, [eax]	movq	mm0, [eax]
1	movq	mm2, [ebx]	pavgusb	mm0, [ebx]
2	pxor	mm7, mm7	movq	[edx], mm0
3	movq	mm1, mm0		
4	movq	mm3, mm2		
5	punpckhbw	mm1, mm7		
6	punpckhbw	mm3, mm7		
7	paddw	mm1, mm3		
8	punpcklbw	mm0, mm7		
9	punpcklbw	mm2, mm7		
10	paddw	mm0, mm2		
11	psraw	mm1, 1		
12	psraw	mm0, 1		
13	packsswb	mm0, mm1		
14	movq	[edx], mm0		

2.4.1.4 Auswahl durch Maskierungsanweisungen

Eine häufige Anwendung in Multimediaalgorithmen besteht in der Auswahl eines Wertes aus zwei Datenelementen, basierend auf einer datenabhängigen Bedingung. Im nichtdatenparallelen Fall kann dies durch eine bedingte Ausführung geschehen, im datenparallelen Fall muss dies durch ein anderes Verfahren erreicht werden. Bei klassischen SIMD-Architekturen wird dieses Problem meist durch ein Aktivierungsbit jedes Prozessors gelöst, so dass ein oder mehrere Prozessoren je nach Ergebnis einer vorherigen Vergleichsoperation die nachfolgenden Anweisungen ausführen oder nicht. Bei Multimediaprozessoren, in denen die datenparallele Ausführung nicht durch mehrere Prozessoren, sondern durch eine spezielle ALU erreicht wird, scheidet diese Möglichkeit aus. Stattdessen wird meist eine parallele Vergleichsoperation angeboten, die anstatt eines Bits im Statusregister eine Maske im Zielregister aufbaut. Dies sei hier anhand eines Vergleiches auf Gleichheit gezeigt.

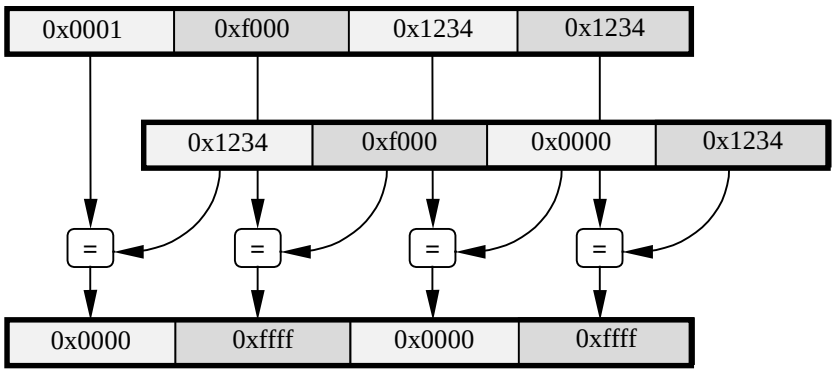


Abbildung 2-3: SIMD-Vergleichsoperation

Die erzeugte Maske kann dann mit Hilfe logischer Operationen benutzt werden, um aus zwei Quellworten ein Zielwort zu erzeugen. Meist ist zur Unterstützung eben dieser Auswahl eine zusätzliche logische Operation definiert, die die „Und"-Verknüpfung eines Quelloperanden mit dem 1-Komplement des zweiten Quelloperanden bildet. Formal ließe sich dies wie folgt beschreiben:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} := \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} cmpeq \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix}, \begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} := \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \wedge \neg \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} \vee \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \wedge \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

Auf den meisten Multimediaprozessoren lässt sich diese Operation in vier bis fünf Befehlen realisieren. Ein weiterer Vorteil dieser Anweisungsfolge gegenüber einer klassischen Vergleichs- und Sprungsequenz liegt in der Freiheit von bedingten Sprüngen und somit spekulationsfreien Ausführung.

2.4.1.5 Umordnungen und Formatkonvertierungen

Um von den datenparallelen Anweisungen der Multimediaprozessoren Gebrauch machen zu können, ist es häufig nötig zwischen den einzelnen Multimediadatentypen zu konvertieren, sowie die Teilworte in einem Prozessorwort umordnen zu können. Häufig wird die Umordnung durch die gleichen Befehle erreicht, die auch für die Formatkonvertierung verwendet werden. Bei der Formatkonvertierung lassen sich zwei Befehlsgruppen unterscheiden: diejenigen, welche von kleineren auf größere Datentypen entpacken und die, welche von größeren auf kleinere packen (größer und kleiner wird hier im Sinne von mehr bzw. weniger Bits verstanden). Beim Entpacken lässt sich zwischen vorzeichenlosen und vorzeichenbehafteten unterscheiden. Beim Packen kommt zusätzlich noch die Frage der Saturierung hinzu. Teilweise sind bei den Vorzeichen sogar Mischvarianten wünschenswert, so zum Beispiel bei Graphikanwendungen von 16 Bit vorzeichenbehaftet auf 8 Bit vorzeichenfrei. Da diese Anweisungen eine unterschiedliche Anzahl Quell- und Zielbits benötigen, passen sie nicht in das allgemeine Befehlskonzept des Prozessors. Um die typische Form von zwei Quelloperanden und einem Zieloperanden beizubehalten, werden die Packanweisungen meist in der Form realisiert, dass sie die Teilworte aus zwei Quellwörtern in einem Zielwort kombinieren. Bei den entpackenden Anweisungen wird im Allgemeinen nur ein Halbwort als Quelle akzeptiert, das dann auf ein ganzes Prozessorwort erweitert wird.

Die Umordnungsbefehle lassen sich in zwei Gruppen unterteilen, jene die die Teilwörter innerhalb eines Prozessordatenwortes umordnen und jene, die die Teilwörter aus zwei Prozessordatenworten zu einem neuen Prozessordatenwort mischen. Die erste Gruppe ist häufig als Spezialfall der zweiten implementiert, wobei beide Quellregister identisch sind. Die Umordnungsanweisungen sind entweder über ein Steuerwort für eine Multiplexerbank implementiert, so dass sich beliebige Permutationen erreichen lassen oder aber über verschiedene feste Permutationsanweisungen, durch deren Kombination sich das gewünschte Ergebnis erzielen lässt.

2.4.1.6 Multimedia-Multiplizierer

Die mächtigste Erweiterung für Multimediaanwendungen ergibt sich durch die Verwendung eines teilwortparallelen Multiplizierers. Da das Ergebnis einer Multiplikation doppelt soviel Bits einnimmt wie jeder der Quelloperanden, kann sie nicht analog zu den einfachen arithmetischen Operationen implementiert werden. Die einfachste Form wird hier durch Saturierung erreicht, bei der das Ergebnis in das Datenformat des Quelloperanden gezwungen wird. Eine andere mächtigere Variante ist gegeben, wenn ein Segment des Ergebnisses für den Zieloperanden ausgewählt werden kann. Dies ermöglicht eine einfache Implementierung von Festkommazahlen, da jeweils die Anzahl der geltenden Stellen des Ergebnisses nach der Multiplikation gewählt werden kann. Eine andere Möglichkeit besteht darin, jeweils zwei Ergebniswörter zu addieren, so dass sich nur die Hälfte der Teilwörter, aber mit doppelter Anzahl Bits, den Zieloperand teilen. Da eine Multiplikation mit nachfolgender Addition

eine sehr häufige Anwendung ist, kann diese Variante sehr oft erfolgreich eingesetzt werden. Weitere Möglichkeiten bestehen in der von klassischen DSPs bekannten Verwendung eines Multiplikationsüberlaufregisters oder eines Akkumulatorregisters für Multiplikations- und Additionssequenzen.

2.4.1.7 Lade- und Speicheroperationen

Multimediaanwendungen zeichnen sich häufig durch ein sehr hohes Datenaufkommen aus, wobei die Daten meist in Strömen oder Blöcken benötigt werden. Viele spezielle Multimediaprozessoren verfügen deshalb über zusätzliche Lade- und Speicherbefehle, um diesen Datenströmen gerecht zu werden. Da der Datenfluss bei diesen Anwendungen meist sehr deterministisch und schon recht früh bekannt ist, welche Daten benötigt werden, lassen sich Speicherlatenzen für Ladeanweisungen sehr gut durch spezielle Befehle zum Vorabladen von Datenelementen vermeiden. Bei den Schreibbefehlen lässt sich eine Geschwindigkeitssteigerung häufig durch eine Lockerung des Konsistenzmodells und durch eine Kombinierung von Schreibzugriffen erreichen. Ersteres hilft dabei Ladebefehlen, die den Ausführungsfluss hemmen, Speicherbefehle vorzuziehen, deren Zieladresse noch nicht bekannt ist. Letzteres ist besonders bei Speicherbereichen von Graphikkarten nützlich, da diese aufgrund ihrer Natur nicht in Caches gepuffert werden können, aber von konsekutiven Zugriffsfolgen stark in der Ausführungsgeschwindigkeit profitieren („write bursts“ durch „write combining“). Als nützlich kann sich auch ein maskierter Schreibzugriff bemerkbar machen, bei dem nur ein Teil eines Datenwortes wirklich in den Speicher geschrieben werden kann. Dies ist wiederum besonders für Graphikkarten interessant, in deren Speicher unregelmäßige oder durch Verdeckungsalgorithmen bestimmte Formen geschrieben werden sollen. Da diese Form des maskierten Schreibens durch die Hardware unterstützt wird [56], vermeidet man so Lese-/Modifizier-/Schreibzugriffe („read-modify-write“) und kann auch Nutzen durch das Kombinieren von Schreibzugriffen ziehen.

2.4.2 Spezielle Multimediaerweiterungen

Im Gegensatz zu den allgemeinen Multimediaerweiterungen dienen spezielle Multimediaerweiterungen nur einem einzelnen Zweck. Es handelt sich meist um zusätzliche Ausführungseinheiten, die eine einzige spezielle Operation sehr effizient ausführen können. Diese Anweisungen widersprechen deshalb dem Prinzip der eher allgemein ausgerichteten Multimediaprozessoren und stellen eine Annäherung an die Verwendung spezifischer Bausteine oder Einheiten zur Verarbeitung von Multimediadaten dar. Entsprechend dieser Einschränkung finden sich derartige Erweiterungen auch nicht bei den Standardprozessoren mit Multimediaerweiterungen, sondern bei den spezifisch zum Zweck der Bearbeitung von Multimediadaten entwickelten Prozessoren. Um so mehr ein Prozessor von diesen speziellen Einheiten Gebrauch macht, um so eher kann er der Gruppe der Spezial-DSPs, wie es sie schon lange gibt, zugeordnet werden. Je nach geschichtlicher Entwicklung und Anwendungsgebiet verfügen diese Prozessoren über unterschiedliche Erweiterungen.

Für diese Erweiterungen spricht die höhere Effizienz, mit der sie die entsprechende Aufgabe erfüllen. So kann ein einfacherer Prozessor mit einer zusätzlichen Ausführungseinheit bessere Leistungen erzielen als ein komplexer Prozessor ohne diese Einheit. Nachteilig ist, dass sich die spezielle Einheit nicht auf die Leistung anderer Anwendungen auswirkt. Da es sich meist um Einheiten handelt, die

auch in klassischen Spezial-DSPs vorhanden sind, können sie ohne hohen Entwicklungsaufwand in die neuen allgemeineren Prozessoren übernommen werden.

2.4.2.1 Bewegungsschätzung

Eine sehr beliebte spezielle Erweiterung ist die Verwendung eines Bewegungsschätzers für die MPEG-Kompression. Es handelt sich hierbei um eine sehr aufwändige, aber auch sehr reguläre Operation. Die Aufgabe besteht darin, einen Block im Referenzbild zu suchen, der dem aktuell bearbeiteten Block im Quellbild entspricht. Dies geschieht im Allgemeinen dadurch, dass Blöcke im Bereich des zu bearbeitenden Blocks im Referenzbild mit dem aktuellen Block verglichen werden und der Block, der zu diesem die geringste Abweichung zeigt, gewählt wird. Dazu muss ein Wert für die Blockdistanz gebildet werden, was meist durch die Verwendung der M_1-Norm geschieht.

$$D_{n,m} = \sum_{i=0}^{15} \sum_{j=0}^{15} \left| Q_{x+i,y+j} - R_{n+i,m+j} \right|$$

Diese Berechnung lässt sich mit einem regulären 16x16 systolischen Array sehr einfach durchführen, der Geschwindigkeitsgewinn ist sehr hoch, der Hardwareaufwand hält sich in Grenzen. Da diese Berechnung für jeden Makroblock sehr häufig stattfindet (etwa 260.000 mal für eine Sekunde Film) ist diese Operation diejenige, die den gesamten Enkodierungsablauf in der Rechenzeit dominiert. Durch Hinzufügen dieser Ausführungseinheit kann so mit einem einfachen Multimediaprozessor selbst die sehr hohe Anforderung für Echtzeit MPEG-2-Kompression erfüllt werden.

Eine allgemeinere Form dieser Erweiterung ist durch einen Befehl gegeben, der die Summe der Distanzen zwischen den Teilworten zweier Operanden bildet. Mit diesem Befehl lässt sich ein deutlicher Geschwindigkeitsgewinn erzielen, ohne dass zusätzliche aufwändige Spezialhardware erforderlich ist [90]. Der Nutzen dieser Erweiterung ist allerdings fraglich, da die hauptsächliche Last bei der Bewegungsschätzung im Speicherzugriff liegt.

2.4.2.2 Huffman-Kodierer und -Dekodierer

Huffman-Kodierung und -Dekodierung [64][29] ist ein sequentieller Prozess und gewinnt nicht oder nur kaum durch die Verwendung von Multimediaprozessoren. Da zusätzlich auch viele bedingte Sprünge und ungeordnete Speicherzugriffe benötigt werden, entscheidet hier die klassische Leistung eines Prozessors über die Ausführungsgeschwindigkeit. Ein Standardprozessor mit Multimediaerweiterungen, der für diese Art Algorithmus geschaffen und optimiert ist, kann diese Anforderung meist sehr gut erfüllen. Spezielle Multimediaprozessoren beziehen ihre Leistung meist durch die hohe Datenparallelität, lange Instruktionsworte mit mehreren Befehlen (VLIW) und eine lange Pipeline. Sie verfügen selten über eine hohe Taktfrequenz und gute Sprungvorhersage. Diese Ausrichtung sorgt für eine geringe Leistung bei der Bearbeitung von Huffman-kodierten Strömen. Da sich dies negativ auf die Gesamtlaufzeit der Kodierung und Dekodierung von Video- und Audiodaten auswirkt, besitzen einige spezifische Multimediaprozessoren eine eigene Bearbeitungseinheit zur Huffman-Kodierung und -Dekodierung.

2.4.2.3 Zweidimensionale Blockbearbeitungsfunktionen

Bei vielen Visualisierungsanwendungen werden häufig große Datenmengen in der Form von rechteckigen Graphikblöcken bewegt und bearbeitet. Standardaufgaben sind hier Skalierung, Filterung oder die klassische Form des Bit-Blits (Logische Kombination von drei Quellrechtecken zu einem Zielrechteck). In einer normalen PC-Umgebung wird diese Aufgabe meist durch den Graphikchip durchgeführt, da dieser den schnellsten Zugriff auf den Graphikspeicher besitzt[3]. Wird ein Multimediaprozessor nicht durch einen Graphikchip unterstützt oder übernimmt er selbst die Bilddarstellung, so wird meist eine spezielle Einheit zur Blockbearbeitung verwendet.

2.4.2.4 Dreidimensionale, perspektivische Bilderzeugung

Ein wichtiges Leistungsmerkmal bei PC-basierten oder für den Spielesektor ausgelegten Multimediasystemen ist die Erzeugung flüssiger dreidimensionaler Bildfolgen. Wie bei den zweidimensionalen Blockfunktionen, wird dies zu einem großen Teil in aktuellen PC-Systemen vom Graphiksubsystem selbst übernommen. Da dies von Spezialeinheiten durchgeführt wird, ist die Leistung mittlerweile so hoch, dass der Flaschenhals der Bilderzeugung in der Fließkommaverarbeitung des Systemprozessors liegt. Ein Multimediaprozessor, der mit diesen Graphikkarten konkurrieren muss, kommt bei den aktuellen Implementierungen nicht ohne eine dedizierte Erweiterung zur dreidimensionalen Bilderzeugung aus.

2.5 Einbindung von Multimediaerweiterungen in Hochsprachen

Für alle Multimediaerweiterungen stellt sich die Frage, wie diese Instruktionen in die vorhandenen Programmierumgebungen integriert sind. Wird, wie bei vielen DSPs üblich, in Assembler programmiert, so löst sich dieses Problem durch die Einführung neuer Prozessorbefehle und Adressierungsarten. Spezielle Erweiterungen werden, falls sie nicht auf Instruktionen abgebildet sind, meist als Peripherie abgebildet und über Ein-/Ausgabeanweisungen oder durch auf den Adressraum abgebildete Register angesprochen. Sollen die Multimediaerweiterungen in Hochsprachen eingebunden werden, so stellt sich das Problem, dass diese nicht für die neuen Instruktionen und Datentypen ausgelegt sind. Eine Einbettung in eine Hochsprache kann in drei Ebenen stattfinden, auf Quellcodeebene, durch Spracherweiterungen oder während der Codeerzeugung.

Zur Veranschaulichung der verschiedenen Varianten wird jeweils das folgende Codefragment implementiert, das einen 16x16-Block von vorzeichenlosen acht Bit-Zahlen mit einem ebenfalls 16x16-Block von vorzeichenbehafteten 16 Bit-Differenzwerten addiert, und das Ergebnis wieder einem Block vorzeichenloser acht Bit-Zahlen zuweist.

[3] Dies wird in modernen Graphikchips dadurch noch verbessert, dass diese über einen PCI- oder AGP-Schnittstelle einen direkten Zugriff auf den Hauptspeicher des Prozessors verfügen und somit Rechteckoperationen im gesamten Speicherraum des Systems ausgeführt werden können.

```
unsigned char * src, * dst;
int bpr;
short delta[16][16];
int x, y, s;

for(y=0; y<16; y++)
   {
   for(x=0; x<16; x++)
      {
      s = (int)(src[x]) + delta[y][x];
      if (s < 0) s = 0; else if (s > 255) s = 255;
      dst[x] = s;
      }
   src += bpr; dst += bpr;
   }
```

2.5.1 Einbettung auf der Quellcodeebene

Dies ist die einfachste Form der Einbettung in eine Hochsprache, da sie nur minimale Änderungen der Programmiersprache und des Übersetzers verlangen.

2.5.1.1 Eingefügte Assembleranweisungen „inline“

Verfügt ein Hochsprachenübersetzer über die Fähigkeit, Assemblerelemente auf Quelltextebene einzubeziehen, ist eine einfache Möglichkeit darin gegeben, dass, wie auch auf der reinen Assemblerebene, neue Instruktionen in den Instruktionssatz eingefügt werden. Dieser Weg ist besonders für Standardprozessoren mit Multimediaerweiterungen üblich, da hier meist bereits komplette Übersetzer und Entwicklungsumgebung existieren, die nur leicht verändert werden müssen. Nachteilig für diese Implementierungsvariante ist, dass zur Benutzung der Multimediafähigkeiten auf Assemblerprogrammierung zurückgegriffen werden muss, was natürlich den Aufwand zur Programmerstellung nicht unerheblich erhöht.

```
for(y=0; y <16; y++)
   {
   __asm {
         mov          eax, [src]
         mov          ecx, [y]
         mov          ebx, [delta]
         shl          ecx, 5
         add          ebx, ecx
         mov          edx, [dst]

         mov          ecx, -4
         pxor         mm7, mm7
      loop1:
         movd         mm0, [eax + 4 * ecx + 16]
         punpcklbw    mm0, mm7
         paddsw       mm0, [ebx + 8 * ecx + 32]
         packuswb     mm0, mm0
         movd         [edx + 4 * ecx + 16], mm0
         inc          ecx
         jne          loop1
         }
   src += bpr; dst += bpr;
   }
```

2.5.1.2 Erweiterung durch Bibliotheken

Eine noch geringere Veränderung der vorhandenen Übersetzer kann erreicht werden, wenn die Multimediaerweiterungen nicht direkt, sondern nur durch eingebundene Bibliotheken genutzt werden. Diese Variante hat den Nachteil, dass keine neuen Multimediafunktionen in Hochsprachen implementiert werden können, sondern neue Bibliotheken in Assembler erstellt werden müssen. Eine noch

stärkere Einschränkung ist gegeben, wenn diese Bibliotheken nur vom Hersteller der Multimediaprozessoren selbst erstellt werden können. In diesem Fall kann der Multimediaprozessor als Standardprozessor mit zusätzlicher Multimediahardware betrachtet werden.

```
for(y=0; y<16; y++)
   {
   AddUnsigned8BySigned16ToUnsigned8(src, delta[y], dst)

   src += bpr; dst += bpr;
   }
```

2.5.1.3 Erweiterung durch Pseudoprozeduren „intrinsic“

Diese Lösung stellt eine Mischung der beiden vorherigen Varianten dar: Es kann auf Assemblerprogrammierung verzichtet werden, es können aber beliebige Multimediaroutinen selbst programmiert werden. Jede Multimediaerweiterung wird wie eine Funktion der Hochsprache verwendet. Allerdings wird bei der Übersetzung kein Funktionsaufruf erzeugt, sondern der Befehl selbst. Dies ist in vielen Übersetzern bekannt und wird für häufige primitive Funktionen benutzt[4], die durch eine oder wenige Prozessorinstruktionen ausgeführt werden. Die Programmierung mit diesen Pseudofunktionen findet aufgrund der Nähe zu den eigentlichen Prozessorinstruktionen auf fast der gleichen Ebene wie die Programmierung in Assembler statt. Vorteilhaft ist, dass der Übersetzer die Registerallokation, Codeerzeugung und Optimierung selbst durchführt und somit den Programmierer von diesen Arbeiten entlastet.

```
for(y=0; y<16; y++)
   {
   for(x=0; x<16; x+=4)
      {
      store4(dst + x, packuswb(paddsw(pupcklbw(load4(src + x)), &(delta[y][x]))));
      }
   src += bpr; dst += bpr;
   }
```

2.5.2 Einbettung durch Spracherweiterung

Dies stellt eine deutlich stärkere Veränderung im Übersetzer und der verwendeten Hochsprache dar. Es wird versucht die zusätzlichen Befehle durch Änderungen der Hochsprachendefinition zu implementieren. Dies kann so weit gehen, dass sich Spezifikationen der Prozessorarchitektur in der Hochsprache wiederfinden. Dies widerspricht in gewisser Weise dem Prinzip einer allgemeinen Hochsprache.

2.5.2.1 Zusätzliche Datentypen

Mit dieser Erweiterung kann vor allem die Saturierungsarithmetik für die Hochsprachenprogrammierer zur Verfügung gestellt werden. Denkbar wäre ein Typattribut wie zum Beispiel „saturated“, das im selben System wie „signed“ oder „unsigned“ verwendet wird.

[4] Am ehesten ist dies bei mathematischen Funktionen, wie zum Beispiel den trigonomischen Funktionen Sinus und Cosinus bekannt, die zwar nicht Elemente der Hochsprache, sondern Funktionen einer Bibliothek sind, aber vom Übersetzer gesondert behandelt werden.

```
unsigned saturated char * src, * dst;
int bpr;
short delta[16][16];
int x, y;

for(y=0; y<16; y++)
   {
   for(x=0; x<16; x++)
      {
      dst[x] = (short)(src[x]) + delta[y][x];
      }
   src += bpr; dst += bpr;
   }
```

Ebenso ließen sich Festkommazahlen in den Sprachumfang aufnehmen.

```
signed fixed(4) short block[16][16];
```

Eine Aufnahme der SIMD-Fähigkeiten in den Sprachumfang durch die Verwendung von gepackten Datentypen ist nur bedingt sinnvoll, da dies eigentlich schon durch die Array-Deklaration erreicht wird. Eine Möglichkeit wäre zum Beispiel durch ein „packed4“ oder „packed8“ Attribut.

```
unsigned saturated packed4 char * src, * dst;
int bpr;
short packed4 delta[16][4];
int x, y;

for(y=0; y<16; y++)
   {
   for(x=0; x<4; x++)
      {
      dst[x] = (short packed4)(src[x]) + delta[y][x];
      }
   src += bpr; dst += bpr;
   }
```

Ein weiterer Nachteil dieser Erweiterungsform liegt darin begründet, dass die Breite der Multimediaregister in die Sprachdefinition aufgenommen wird.

2.5.2.2 Zusätzliche Operatoren

Ein weiterer Weg, Multimediaelemente in den Sprachumfang einzubeziehen, ist die Verwendung zusätzlicher Operatoren. Durch diese lässt sich wie auch bei der Verwendung von neuen Typelementen Saturierungsarithmetik implementieren[5]. Sinnvoll wäre die Übernahme neuer Operatoren für Operationen, die nicht im Originalumfang von Hochsprachen vorhanden sind, sich aber nur schwer mit bestehenden Operatoren ausdrücken lassen, wie zum Beispiel des Mittelwertoperators.

Eine interessante Erweiterung wäre die Einführung von Operatoren für Felder. Dies würde es erlauben, sehr einfach die gewünschte SIMD-Parallelität zu spezifizieren, ohne die unterliegende Implementierung im Prozessor in die Sprachdefinition einzufügen.

[5] Es erscheint mir allerdings intuitiver, Saturierung eher als eine Eigenschaft eines Datentypen zu sehen, als eine Eigenschaft eines Operators.

```
unsigned saturated char * src, * dst;
int bpr;
short delta[16][16];
int x, y;

for(y=0; y<16; y++)
   {
   dst = ((short[16])src) +[16] delta[y];

   src += bpr; dst += bpr;
   }
```

Eine andere Variante könnte unter Modifikation des Indexoperators und Erweiterung des Additionsoperators etwa so aussehen:

```
unsigned saturated char * src, * dst;
int bpr;
short delta[16][16];
int x, y;

for(y=0; y<16; y++)
   {
   dst[0:16] = ((short[16])src[0:16]) + delta[y][0:16];

   src += bpr; dst += bpr;
   }
```

Die Verwendung der [x:y]-Indizierung würde hier für einen Bitfeldbereich im Datenwort beginnend bei x und der Größe y stehen. Dies würde es dem Übersetzer erlauben, SIMD-Operationen für beliebige Feldgrößen zu erzeugen.

2.5.2.3 Erweiterte Kontrollkonstrukte

Durch eine Erweiterung der Kontrollkonstrukte lassen sich Multimediaerweiterungen kaum in eine Hochsprache integrieren. Eine Möglichkeit wäre es, durch eine parallele Variante der Zählschleife dem Übersetzer zusätzliche Informationen über die Parallelität des Algorithmus zu geben. Dies würde ebenfalls Parallelisierungshemmnisse durch „aliasing“ vermeiden. Dieser Ansatz wurde für die datenparallelen Programmiersprachen „Modula-2*“ und HPF („High Performance Fortran“) gewählt[57][58].

```
unsigned saturated char * src, * dst;
int bpr;
short delta[16][16];
int x, y;

for(y=0; y<16; y++)
   {
   forpar(x=0; x<16; x++)
      {
      dst[x] = (short)(src[x]) + delta[y][x];
      }
   src += bpr; dst += bpr;
   }
```

2.5.3 Einbettung in der Codeerzeugung

Die für den Programmierer am einfachsten zu handhabende Variante ist es, wenn die Multimediaerweiterungen vom Übersetzer selbstständig während der Codeerzeugung im Quellprogramm erkannt und in den Zielcode eingefügt werden[6]. Dies ist allerdings die für den Autor eines Übersetzers die

[6] Ähnliches findet sich bei einigen Hochsprachenübersetzern, bei denen einge häufige einfache Ausdruckselemente nicht über die normalen Routinen optimiert werden, sondern die durch einen Mustererkenner komplett erkannt und ersetzt werden. Sehr häufig findet sich dies zum Beispiel bei der Übersetzung der Maximumsbildung, die meist als „x > y ? x : y“ geschrieben wird und somit leicht er-

aufwändigste Methode. Auch bleibt fraglich, ob der erzeugte Code von der für Multimedia notwendigen Güte ist.

Ein weiteres Problem dieses Ansatzes findet sich darin, dass der Aufwand, das Quellprogramm in einer Form zu schreiben, in der der Übersetzer diese Elemente findet, höher sein kann, als es die Verwendung eines spezifischen „intrinsics“ wäre. Besonders auffällig dürfte dies bei der Verwendung von Saturierungsarithmetik sein, bei der jedes mal wieder der Test auf eine Bereichsüberschreitung durchgeführt werden muss. Ungeklärt bleibt auch, ob der Übersetzer durch die zahlreichen Datentypen, die hierbei benutzt werden müssen, immer eine optimale Lösung finden.

Nützlich hingegen dürfte es sein, die vorhandene SIMD-Parallelität durch den Übersetzer finden zu lassen. Da sich dies meist in einfachen Zählschleifen findet, ist der zu erwartende Aufwand für den Autor des Übersetzers überschaubar.

2.5.4 Einbettungsmöglichkeiten in C++

Durch C++ ergibt sich die Möglichkeit, einige der oben genannten Ansätze recht einfach und ohne Änderung der Sprachdefinition in die Übersetzungsumgebung einzufügen [72]. Durch die Verwendung eigener Klassen für erweiterte Typen, sowie dem Überladen von Operatoren, lassen sich die Änderungen, die durch das Einfügen von Typen entstehen, aus der Sprachdefinition heraushalten. Als Implementierung der eigentlichen Funktionen bietet sich die Verwendung von „intrinsics“ an, die in „inline“-Methoden geschirmt werden.

```
class satrt_char
   {
   protected:
      char x;
   public:
      satrt_char(short s)
         {x = mmx_pack_saturated_signed_word_to_signed_char(s);}

      friend satrt_char operator+ (const satrt_char u, const satrt_char v)
         {return mmx_add_signed_saturated_byte(u, v);}

      friend satrt_char operator- (const satrt_char u, const satrt_char v)
         {return mmx_sub_signed_saturated_byte(u, v);}
   };
```

Das Erkennen von Parallelität kann dem Übersetzer überlassen werden oder durch eigene Feldklassen explizit gemacht werden.

kannt werden kann. Ein weiterer bekannter Fall ist auch eine Kopierschleife, die direkt in einen Kopierbefehl des Prozessors übersetzt wird.

```
template<int size>
class satrt_char_field
   {
   protected:
      char field[size];
   public:
      satrt_char_field(satrt_short_field<size> s)
         {
         int x;
         for(x=0; x<size; x+=4)
            {
            mmx_pack_saturated_signed_word4_to_signed_char4(s.field+x, field+x);
            }
         }

      friend satrt_char_field<size>& operator+= (const satrt_char_field<size> & u)
         {
         int x;
         for(x=0; x<size; x+=4)
            {
            mmx_add_signed_saturated_byte4(field+x, u.field+x, field+x);
            }
         }
   };
```

Problematisch bei diesem Ansatz ist, dass durch die Aufspaltung einer Operation über mehrere Schleifen sehr viele temporäre Variablen entstehen, die bei einem Mischen der Schleifenkörper nicht entstehen würden. Kann ein Übersetzer Datenparallelität erkennen, so kann er dies innerhalb einer Schleife deutlich besser als über mehrere Schleifen hinweg. Der obige Ansatz lohnt sich also nur, wenn der Übersetzer die Datenparallelität nicht selbst erkennen kann. Um möglichen Nachteilen aus geringer Parallelisierung und Optimierung durch den Übersetzer aus dem Weg zu gehen, könnte man viele verschiedene dieser Codefragmente als Methode bereitstellen. Man würde sich damit aber wieder dem Ansatz einer Erweiterung durch eine Bibliothek annähern, allerdings die Fähigkeit zur leichten Einbindung und Erweiterung nicht verlieren.

2.6 Simulation von Multimediaerweiterungen durch Standardinstruktionen

Viele Multimediaelemente lassen sich unter gewissen Laufzeitnachteilen auch mit Standardinstruktionen nachbilden. Dies hat dazu geführt, dass einige Prozessorhersteller bewusst auf zusätzliche Multimediabefehle verzichten. Speziell auf Prozessoren mit 64 Bit und mehr lässt sich auch mit Standardinstruktionen ein großer Teil der neuen Multimediafunktionen simulieren. Typischerweise ist aber mit einem zwei- bis achtfach so hohen Aufwand zu rechnen. Die aufwändigsten Simulationen ergeben sich für die parallele Multiplikation und bei Saturierungsrechnungen, speziell einer Teilwortgrößenänderung. Als Beispiel sei hier das Codefragment aus vorherigem Abschnitt verwendet:

```
movd         mm0, [eax + 4 * ecx + 16]
punpcklbw    mm0, mm7
paddsw       mm0, [ebx + 8 * ecx + 32]
packuswb     mm0, mm0
movd         [edx + 4 * ecx + 16], mm0
```

Da die Simulation der Entpack- und Packbefehle sehr aufwändig ist, wird ein anderes Verfahren verwendet. Das Wort aus acht Datenelementen zu je acht Bit wird in die geraden und ungeraden

Elemente zerlegt. Dies kann durch ein einfaches logisches „Und“ geschehen. Das Zusammenfügen wird entsprechend durch ein „Oder“ realisiert.

```
mov             [src], r4
and             r4, 0xff00ff00ff00ff00, r5
and             r4, 0x00ff00ff00ff00ff, r4
shr             r5, 8, r5
```

Um bei der nachfolgenden Operation keinen Unterlauf in das nächste Teilwort zu verursachen, wird ein Wächterbit eingefügt:

```
or              r5, 0x8000800080008000, r5
or              r4, 0x8000800080008000, r4
```

Danach kann der Korrekturterm addiert werden. Da sich die Teilworte aber nicht in der erwarteten Reihenfolge befinden, muss der Korrekturterm entsprechend der veränderten Reihenfolge gespeichert sein:

```
add             r4, [delta], r4
add             r5, [delta + 8], r5
```

Bei der Zusammenfügeoperation muss nun entsprechend ein eventueller Überlauf ausgeglichen werden. Dazu werden für den kleiner als 0 und den größer als 255-Fall entsprechende Masken gebildet:

```
and             r4, 0x8000800080008000, r6
shr             r6, 15, r6
or              r6, 0x8000800080008000, r6
sub             r6, 0x0001000100010001, r6
and             r4, r6, r4
and             r4, 0x0100010001000100, r6
shr             r6, 8, r6
or              r6, 0x8000800080008000, r6
sub             r6, 0x0001000100010001, r6
or              r4, r6, r4
and             r4, 0x00ff00ff00ff00ff, r4
```

Entsprechend wird dies auch für die zweite Teilwortgruppe durchgeführt. Danach können die Teilwörter wieder zusammengefügt werden.

```
shl             r5, 8, r5
or              r4, r5, r4
mov             r4, [dst]
```

Auch Multimediamultiplikationen können durch entsprechende Instruktionen aus einem Standardbefehlssatz simuliert werden. Als Beispiel sei hier graphisch die Multiplikation zweier komplexer Zahlen mit je 16 Bit imaginär und reell gezeigt. Problematisch ist hierbei natürlich die Vorzeichenbearbeitung, die in einem zweiten Schritt korrigiert werden muss.

$$s_r = x_r y_r - x_i y_i ; s_i = x_r y_i + x_i y_r$$

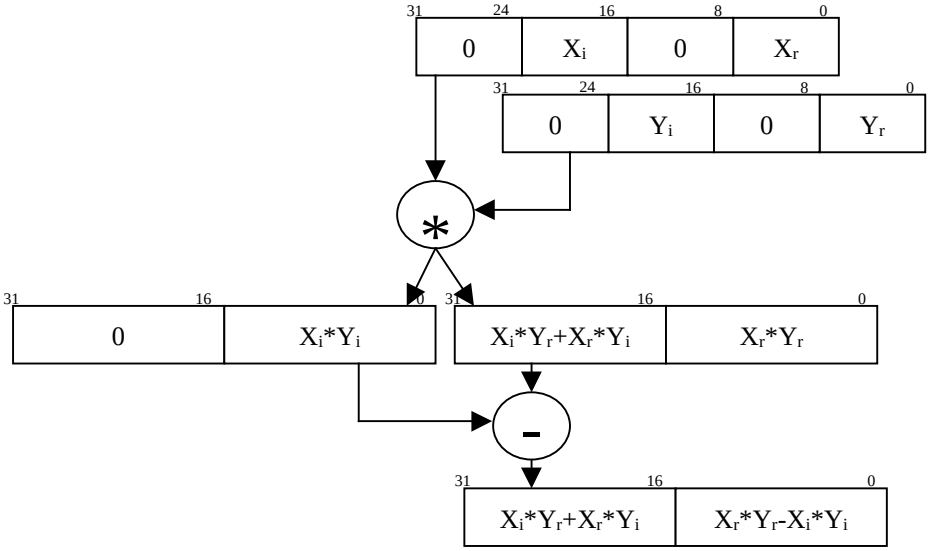


Abbildung 2-4: Simulation einer SIMD-Multiplikation durch nicht SIMD Befehle

Basierend auf dieser Zerlegung können auch andere Multiplikationen simuliert werden. Allerdings ist der Nutzen geringer als bei diesem Spezialfall.

2.7 Ausblick

Multimediaerweiterungen gehören mittlerweile zum Standardbefehlsumfang der meisten Prozessoren. Dies wird sich in der Zukunft weiter verstärken. Dies geschieht vor allem durch breitere Register (z.B. 128 Bit) und durch die Unterstützung von SIMD-Fließkommaoperationen. Weiterhin wird sich der Trend zu weiterer Integration verstärken. Systemprozessoren werden zusätzliche Elemente der Systeme in sich aufnehmen („System on a Chip“) wie zum Beispiel die Bilddarstellung. Auch dadurch wird sich der Bedarf nach leistungsstarken Multimediafunktionen noch verstärken, da die Unterstützung für zwei- und dreidimensionale Graphikerzeugung in den Systemprozessor verschoben wird.

Der Schwachpunkt der aktuellen Multimediaprozessoren und Erweiterungen liegt in ihrer geringen Einbettung in Hochsprachen. Alle verfügbaren Entwicklungssysteme benutzen entweder eine direkte Kodierung in Assemblerschreibweise oder bieten prozessorspezifische „intrinsic“-Funktionen. Dies verhindert einerseits die einfache Umsetzung eines Programms von einem Multimediaprozessor auf einen anderen (oder auch nur von einer Generation zur nächsten), andererseits wird die Implementierung des Prozessors in die Ebene der Programmierung gespiegelt, was die Programmierung dieser Chips nicht unwesentlich erschwert. Es bleibt abzuwarten, in welcher Form sich Multimediainstruktionen in zukünftigen Übersetzergenerationen wiederfinden und ob sich diese Instruktionen in die allgemeine Programmform einfinden oder ob sie eine getrennte Anweisungsfamilie bleiben, die nur von Spezialisten genutzt werden können.

3 Mehrfädige Prozessoren

3.1 Einführung

Mehrfädige Prozessoren werden seit mehreren Jahren in der Forschungsgemeinschaft diskutiert und sind das Ziel verschiedener Studien und Simulationen. Neben einigen frühen Implementierungen in parallelen Hochleistungsrechnern erscheint erst in jüngster Zeit eine Verwendung dieses Verfahrens für Standardprozessoren realisierbar.

3.2 Motivation

Der grundlegende Gedanke eines mehrfädigen Prozessors findet sich darin, dass mehrere Programmfäden quasi gleichzeitig auf einem Prozessor ausgeführt werden[85]. Dies geschieht zum einen, um Latenzen, wie sie bei Speicherzugriffen entstehen zu überbrücken[7], zum anderen, um die Ausnutzung der Ausführungseinheiten eines superskalaren Prozessors zu verbessern. Studien haben gezeigt, dass die Leistungsfähigkeit von superskalaren Prozessoren durch die Verwendung mehrerer Kontrollfäden mit moderatem Hardwareaufwand deutlich gesteigert werden kann.

Betrachtet man eine typische Pipelineauslastung eines Prozessors, so erkennt man, dass viele Ausführungsmöglichkeiten nicht genutzt werden:

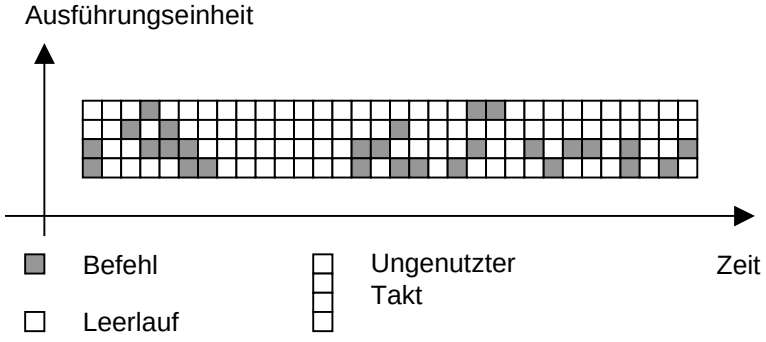


Abbildung 3-1: Pipelineauslastung in einfädigem Prozessor

Man kann zwei Arten unbenutzter Ausführungsmöglichkeiten erkennen:

- Ein kompletter Takt wird nicht benutzt. Dies entsteht meist durch eine Anweisung mit langer Latenz, zum Beispiel einem Fehlzugriff im Daten-Cache oder einer Fehlspekulation der Sprungvorhersage[61]

[7] Ein anderes bekanntes Verfahren zur Überbrückung dieser Latenzen wird durch ein Vorabladen der benötigten Daten erreicht. Hierbei wird ein Ladeanforderungsbefehl für die benötigten Daten weit vor dem Befehl der diese Daten benötigt ausgeführt. Dieses Verfahren kann allerdings nur bei sehr regelmäßigen Speicherstrukturen genutzt werden[50].

- Eine Ausführungseinheit wird in einem Takt nicht benutzt. Dies entsteht meist durch Datenabhängigkeiten oder Ressourcenauslastung

Der hier betrachtete Lösungsansatz versucht nun die vorhandenen Leertakte durch Instruktionen eines anderen Kontrollfadens zu füllen. Dies würde zu einer besseren Auslastung der Prozessorressourcen ohne deutlich gesteigerten Hardwareaufwand führen.

3.3 Rückblick

Werden Kontrollfäden nur taktweise verschränkt, stehen drei verschiedene Kontextwechselstrategien bereit[47]:

- Taktweise : Die verfügbaren Kontrollfäden können jeweils einen Takt nutzen und geben dann das Nutzungsrecht an den nächsten Kontrollfaden weiter.
- Blockweise : Ein Kontrollfaden kann den Prozessor solange nutzen, bis er keine Instruktionen mehr ausführen kann, zum Beispiel durch einen Cache-Zeilenfehlzugriff.
- Prioritätsgesteuert : Ein Kontrollfaden kann nur dann Befehle ausführen, wenn kein Kontrollfaden mit einer höheren Priorität Befehle zur Ausführung bereit hat.

Ursprünglich wurden mehrfädige Prozessoren eingesetzt, um die langen Latenzen durch einen entfernten Speicherzugriff in einem hochparallelen Prozessor mit verteiltem Speicher auszugleichen [9][43]. Wenn ein Prozessor auf ein Datenelement von einem anderen Prozessor warten muss, wird ein anderer Kontrollfaden gestartet [6]. Dies erlaubt es, die langen Latenzen und damit verbundenen Totzeiten eines Prozessors mit sinnvollen Instruktionen zu überbrücken. Bei diesen Prozessoren entstehen durch den Kontextwechsel meist mehrere Leerlauftakte, bis die Pipeline wieder gefüllt ist. Ein Kontextwechsel lohnt sich also erst dann, wenn eine Wartezeit auftritt, die deutlich höher ist als die Leerlaufzeit des Wechsels. Der Wechsel wird bei diesem „Block-multithreading“ entweder implizit durch das auftretende Ereignis (wie zum Beispiel Fehlzugriff im Daten-Cache) verursacht oder explizit durch eine Prozessoranweisung.

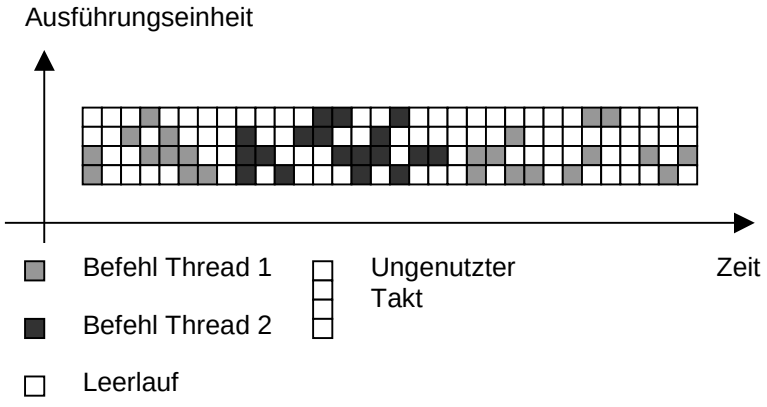


Abbildung 3-2: Pipelineauslastung in mehrfädigem Prozessor mit Blockverschränkung

Der zusätzliche Hardware-Aufwand ist relativ gering. Lediglich ein zweiter Registersatz, der über eine Auswahllogik selektiert wird, wird benötigt.

Durch die sich immer weiter öffnende Schere zwischen Prozessorleistung und Speicherzugriffszeit wird dieses Verfahren auch für nicht verteilte Systeme interessant, da auch die Latenzen, die in einem normalen Einprozessorsystem durch einen Fehlzugriff auf den Daten-Cache entstehen, bereits heute mehrere hundert Takte betragen können.

Eine andere Variante der Mehrfädigkeit wird durch Taktverschränkung („Cycle by Cycle interleaving“) erreicht. Hierbei werden in jedem Takt die Anweisungen eines anderen Kontrollfadens ausgeführt [27][70]. Dies ermöglicht es, auch kurze Latenzen, wie sie z.B. bei einer lokalen Ladeoperation oder einer Fließkommaberechnung entstehen, zu überbrücken. Es können hier zwei Varianten unterschieden werden: solche, die in jedem Takt einen Kontextwechsel durchführen müssen und solche, die einen Kontextwechsel durchführen können. Im ersten Fall entstehen ungenutzte Takte in der Pipeline, falls der Kontrollfaden, der für diesen Takt vorgesehen war, in diesem keine Instruktion starten bzw. ausführen kann. Nachteilig ist hier, dass bei einer geringer parallelen Last ein einzelner Kontrollfaden keinen Gewinn aus dem komplexen Prozessor ziehen kann.

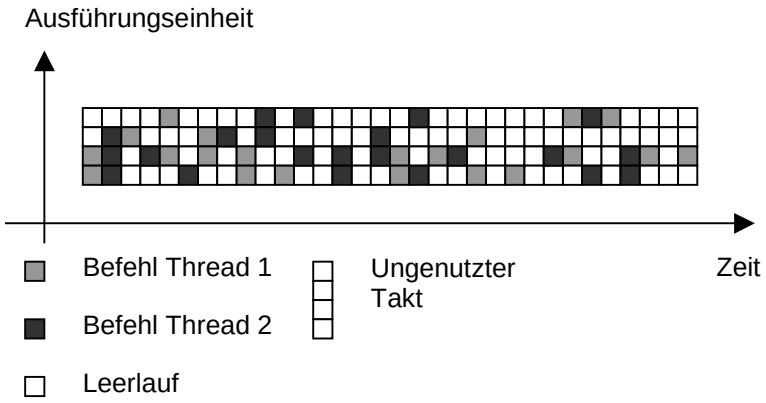


Abbildung 3-3: Pipelineauslastung in mehrfädigem Prozessor mit Taktverschränkung

Diese beiden Verfahren können allerdings nur komplett ungenutzte Takte durch andere Kontrollfäden überbrücken. Einzelne ungenutzte Ausführungseinheiten bleiben ungenutzt.

3.4 Mehrfädig Superskalar „Simultaneous Multithreading“

Dieses Modell entsteht aus einem superskalaren Prozessor mit dynamischer Instruktionszuordnung durch das Hinzufügen mehrerer Kontrollfäden. Die Ausführungseinheiten werden von allen Kontrollfäden gemeinsam genutzt, die Zuordnung einer Anweisung zu einer Ausführungseinheit wird dynamisch unter den bereiten Anweisungen aller Kontrollfäden getroffen. Dieser Ansatz verbindet die Vorteile eines superskalaren Prozessors mit denen eines mehrfädigen. Auch ein einzelner Kontrollfaden kann die zusätzlichen Ausführungseinheiten nutzen, wenn er genug eigene Parallelität besitzt.

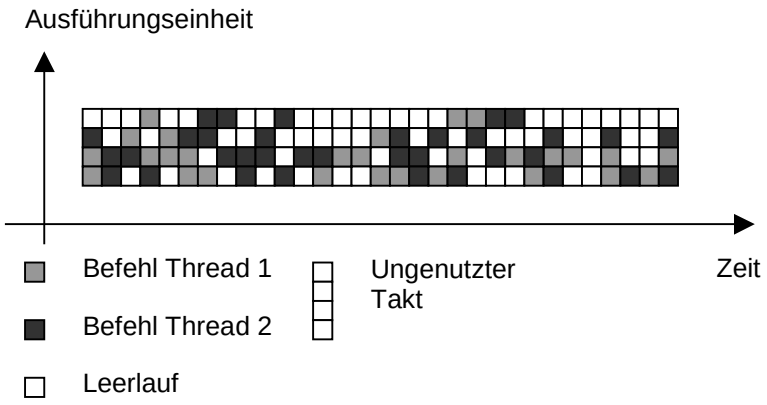


Abbildung 3-4: Pipelineauslastung in mehrfädigem Prozessor mit Simultaneous Multithreading

Untersuchungen mit Simulatoren zeigen, dass diese Variante eines mehrfädigen Prozessors in der Lage ist, die genutzte Parallelität eines Prozessors deutlich zu erhöhen [81][82][45][15][42]. Der zusätzlich entstehende Hardwareaufwand ist signifikant geringer als bei einem System mit mehreren Prozessoren auf einem Chip [11], darf aber nicht völlig vernachlässigt werden, da durch zusätzliche Register und Auswahllogik längere Gatterlaufzeiten entstehen, die den einfädigen Fall deutlich benachteiligen können [13].

3.5 Softwareunterstützung

Ein mehrfädiger Prozessor kann seine Mehrfädigkeit auf mehrere Arten nach außen repräsentieren:

- Der Prozessor simuliert mehrere komplette einzelne Prozessoren. Das Betriebssystem muss hierfür nicht oder nur in geringem Maße angepasst werden (sofern es bereits SMP unterstützt). Nachteilig ist hier, dass viele Steuerungs- und Kontrollbefehle wie zum Beispiel zur Steuerung des Daten-Caches für jeden Kontrollfaden getrennt existieren.
- Der Prozessor stellt sich dem Betriebssystem als einfädiger Prozessor mit zusätzlichen Erweiterungen zur Erzeugung weiterer Kontrollfäden dar. Dies verlangt eine Anpassung des Betriebssystems, da die Unterstützung mehrerer Kontrollfäden auf einem Prozessor üblicherweise nicht vorhanden ist. Vorteilhaft ist, dass das Betriebssystem die volle Kontrolle über die einzelnen logischen Kontrollfäden besitzt und diese bei einer zusätzlichen SMP-Lösung besser auf die einzelnen Prozessoren verteilen kann. Diese Lösung kann auch benutzt werden, um die Mehrfädigkeit dem Programmierer oder Hochsprachenübersetzer direkt zur Verfügung zu stellen.
- Der Prozessor stellt sich dem Betriebssystem als einfädiger Prozessor dar. Zusätzliche Kontrollfäden werden vom Hauptkontrollfaden automatisch abgespalten, ohne dass eine Softwareunterstützung nötig wäre. Dies könnte zum Beispiel bei einem spekulativen Sprung passieren, so dass beide Anweisungssequenzen parallel ausgeführt werden, und bei der Bestätigung des Sprunges wird ein Kontrollfaden eliminiert [62], [40].

Die Mehrfädigkeit kann dem Programmierer auf mehrere Arten bereitgestellt werden.

- Die physikalischen Kontrollfäden werden auf Systemkontrollfäden abgebildet. Dies erlaubt es, die Mehrfädigkeit in bereits vorhandenen Anwendungen einzusetzen.
- Die Mehrfädigkeit wird dem Programmierer direkt in der Form leichtgewichtiger Kontrollfäden zur Verfügung gestellt [19]. Um die Mehrfädigkeit in diesem Fall zu nutzen, muss Software neu erstellt werden, allerdings wird hierdurch der maximale Nutzen aus einem mehrfädigen Prozessor erzielt.
- Die Mehrfädigkeit wird durch den Compiler genutzt, der eine automatische Parallelisierung des Programms durchführt [20]. Dies erleichtert die Entwicklung mehrfädiger Programme, ist allerdings nur bei einer begrenzten Klasse von Programmen möglich. Eine verwandte Lösung wäre die spekulative Ausführung von Programmteilen durch einen zweiten Kontrollfaden [71] („Space-Time Computing“ [78]), [79], [83].

Diese Lösungen lassen sich natürlich frei kombinieren, so dass sowohl Betriebssystem, Anwendungsprogramm als auch Übersetzer Kontrollfäden erzeugen und kontrollieren können.

3.6 Ausblick

Der Weg zu mehr grobkörniger Parallelität in einem Prozessorchip wird zur Zeit vor allem in zwei Richtungen verfolgt, einmal die Implementierung mehrerer mehr oder weniger unabhängiger Prozessoren auf einem Chip, wie z.B. mit dem SUN MAJC[25][78] oder dem IBM POWER 4[35] Prozessor und einem Simultaneous Multithreading wie im Compaq Alpha EV8[16]. Zudem finden sich mehrfädige Anwendungen auch vermehrt in Spezialprozessoren wie im Intel IXP2000[26]. Dieser Prozessor besitzt nicht nur sechs unabhängige Prozessoren („Microengine“), jeder Prozessor besitzt dazu auch noch vier Ausführungskontexte, die bei Bedarf gewechselt werden („Blockmultithreading“).

4 Ein mehrfädiger Multimediaprozessor

4.1 Einführung

Durch die Kombination eines Prozessors mit Multimediaerweiterungen mit einem mehrfädigen („Simultaneous Multithreading“) Prozessor entsteht ein neuer ungleich mächtigerer Prozessortyp. In diesem Kapitel werde die grundlegenden Prinzipien und Hintergründe dieses Prozessorkonzepts aufgezeigt.

4.2 Motivation

Multimediaanwendungen bieten aufgrund ihrer Struktur sehr viel Parallelität, sowohl auf der Instruktionsebene, als auch auf der Anwendungsebene. Durch die Verwendung von Multimediainstruktionen kann die Parallelität auf der Instruktionsebene zu einem sehr großen Umfang genutzt werden. Hierdurch sinkt allerdings die für die superskalare Ausführung benötigte Parallelität ab, so dass ein Prozessor mit parallelen Ausführungseinheiten nur einen Bruchteil seiner maximalen Leistung erbringen kann. Dies wird noch durch die enorme Speicherbelastung und die ungleiche Verteilung der verschiedenen Anweisungstypen in einer Multimediaanwendung erschwert. Durch einen mehrfädigen Prozessor könnte die auf Anwendungsebene vorhandene Parallelität auf die Instruktionsebene gebracht und genutzt werden.

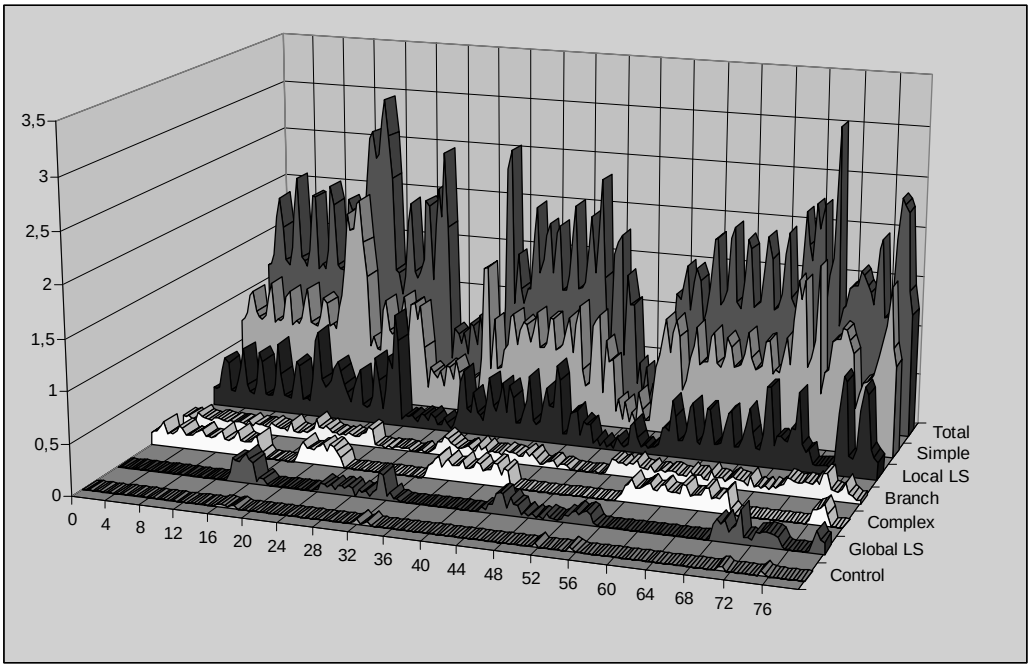


Abbildung 4-1: Instruktionsmischung in Multimediaroutine

In diesem Diagramm sind die durchschnittliche Anzahl der Instruktionen pro Takt (IPC) für verschiedene Anweisungstypen während der Ausführung einer Multimediaroutine angegeben (die untere Achse ist in jeweils 1000 Instruktionen skaliert). Man erkennt deutlich, wie zu unterschiedlichen Zeiten der Ausführung verschiedene Anweisungstypen überwiegen.

Als weiterer Faktor kommen hier auch die Echtzeitanforderungen eines Multimediasystems hinzu. Werden Elemente, die bisher als einzelne unabhängige Komponenten realisiert waren, auf einem einzelnen Prozessor ausgeführt, so entstehen deutlich stärkere Echtzeitvoraussetzungen als zuvor. Diese könnten durch einen mehrfädigen Prozessor in idealer Weise erfüllt werden, da er durch seinen verlustfreien Kontextwechsel ideale Bedingungen für ein Echtzeitbetriebssystem erfüllt.

4.3 Mehrfädigkeit in bestehenden/geplanten Multimediaprozessoren

Einige der am Markt verfügbaren oder geplanten Multimediaprozessoren verfügen bereits über ein Konzept zur Nutzung der Mehrfädigkeit. Dies ist nicht immer vollständig ausgebaut, könnte aber richtungsweisend für zukünftige Implementierungen dieser Familien sein.

4.3.1 TMS 320C8X

Bei diesem Prozessor handelt sich es nicht um einen mehrfädigen Prozessor, vielmehr sind fünf unabhängige Prozessoren auf einem Chip kombiniert. Diese internen Prozessoren kommunizieren über einen gemeinsamen Speicher, der durch ein Verteilungsnetzwerk („Crossbar“) für alle verfügbar ist. Die fünf Prozessoren teilen sich in einen RISC Prozessor und vier (in einer späteren Variante nur noch zwei) DSPs auf. Die Aufgabe des RISC Prozessors ist es, die DSPs zu koordinieren und gleichmäßig mit Arbeit zu versorgen.

Durch die heterogene Struktur des Prozessors ist eine Programmentwicklung deutlich erschwert, da die Aufteilung der Anwendung auf die verschiedenen Teilprozessoren manuell erfolgen muss. Nachteilig ist weiterhin, dass ein Programmteil, der nicht parallelisiert werden kann, keine Beschleunigung durch die anderen Prozessoren erhält.

4.3.2 MicroUnitys MediaProzessor

In diesem Prozessor können bis zu fünf unabhängige Kontrollfäden ausgeführt werden. Es wird eine taktweise Verschränkung („Cycle by Cycle Interleaving“) verwendet. Jeder Kontrollfaden kann pro Takt nur eine Instruktion starten. Es ergibt sich also eine maximale Instruktionszahl von Fünf pro Takt. Leider sind genauere Daten über diesen Prozessor nicht öffentlich verfügbar. Nachteilig ist hier natürlich, dass ein einzelner Kontrollfaden nicht durch die Mehrzahl an Ausführungseinheiten gewinnt. Dies ist insbesondere störend, da bei entsprechender Ausnutzung der parallelen Elemente die Ausführungszeit durch die nicht parallelen Elemente bestimmt wird („Amdahls Gesetz“).

4.3.3 SUN MAJC-5200 [25][78]

Es handelt sich hier nicht eigentlich um einen mehrfädigen Prozessor, sondern um einen Multiprozessor. Zwei fast unabhängige Prozessoren teilen sich einen gemeinsamen Daten-Cache. Zusätzlich dazu verfügt der Prozessor auch über eine Art „Thread-Cache“. Durch einen expliziten Befehl kann die Ausführung zu einem anderen Kontrollfaden gewechselt werden. Es handelt sich also um explizites Block-Multithreading. Die Ähnlichkeit zu einem „Simultaneous Multithreading“ Prozessor ist vor allem durch den gemeinsamen Daten-Cache gegeben, so dass sich Ergebnisse, die sich auf Cacheoptimierungen beziehen, durchaus zwischen den beiden Modellen austauschen lassen.

4.3.4 Cradle Technology „Universal Microsystem“

Dies ist zwar kein direkter Multimediaprozessor in der Form, dass es sich um einen Prozessor mit Multimediabefehlssatz handelt, doch liegt ein wichtiger Zielbereich dieses Prozessors in der Bearbeitung von Multimediadaten.

Dieser Prozessor ist auf eine hohe Anzahl von Teilprozessoren hin in einer hierarchische Struktur aufgebaut.

Jeweils vier MSP („Multi-Stream Prozessor“) Elemente bilden zusammen mit lokalem Daten- und Instruktionsspeicher ein Prozessorelement. Ein einzelner Chip kann wiederum aus mehreren dieser „Quads“ bestehen. Ein MSP selbst gliedert sich in einen 32bit RISC Prozessor und zwei DSEs („Digital Signal Engine“), die mit eigenem Programmspeicher und Registern DSP Instruktionen ausführen. Man kann diesen Prozessor als eine Sammlung von vielen Prozessoren der Art eines TMS320C8X sehen.

Aufgrund dieser sehr komplexen und heterogenen internen Struktur ist eine Programmentwicklung erwartungsgemäß sehr aufwändig und eher mit dem Entwurf einer entsprechenden Schaltung zu vergleichen.

4.3.5 Simulation der Mehrfädigkeit in nicht mehrfädigen Prozessoren

Durch die große Zahl Register und die hohe Registerbreite von teilweise bis zu 128Bit entsteht bei vielen Multimediaprozessoren ein ungewöhnlich hoher Aufwand für einen Kontextwechsel. Dies kann dadurch abgefangen werden, dass ein Teil der Register für spezielle sehr häufig ausgeführte Kontrollfäden verwendet wird. Dies vermindert zwar deutlich die Kontextwechselzeit, reduziert aber auch die Anzahl der verfügbaren Register pro Kontrollfaden.

Geht man bei der Aufteilung der Register noch einen Schritt weiter und teilt diese fest auf mehrere Kontrollfäden auf, so lässt sich bei sehr regulären Anwendungen eine Mehrfädigkeit dadurch erreichen, dass unabhängige Kontrollfäden auf der Instruktionsebene miteinander gemischt werden. Dies ist natürlich nicht möglich, wenn der Kontrollfluss eines Ausführungsfadens sehr ungeordnet ist und zahlreiche bedingte Ausführungssequenzen enthält. Begünstigt wird dieses Verweben mehrerer Kontrollfäden durch das Vorhandensein bedingter Instruktionen, da so die Häufigkeit von Kontrollfluss

verändernden Anweisungen stark gesenkt werden kann. Auch Instruktionen zum expliziten Vorabladen von Cache-Zeilen können sich hier als nützlich erweisen, da sie es erlauben, die Grenze der Überlappung der Zeit zwischen Speicherzugriffstart und Datennutzung mit anderen Programmteilen über die Größe des Befehlrückordnungspuffers hinaus schieben können.

Diese Variante der Mehrfädigkeit ist natürlich sehr statisch und kann kaum auf variierende Lasten optimiert werden. Auch nimmt der Codeumfang deutlich zu, da mehrere Varianten jedes Kontrollfadens als Code vorhanden sein müssen. Weiterhin nachteilig ist, dass diese Form der Mehrfädigkeit Latenzen durch falsch vorhergesagte Sprünge nicht ausgleichen kann, da alle Kontrollfäden dadurch gleichermaßen betroffen sind.

4.4 Vor- und Nachteile der Mehrfädigkeit in einem Multimediaprozessor

Die Verwendung eines mehrfädigen Prozessorkerns bringt einige spezielle Vorteile für Multimediaprozessoren, wirft aber auch einige Probleme auf, die in einem einfädigen Prozessor bei Multimediaanwendungen nicht, oder nicht so stark auftreten.

Als Vorteile zeichnen sich ab:

- Multimediaanwendungen zeichnen sich durch eine sehr unausgeglichene Instruktionsmischung aus. Durch die Verwendung der Mehrfädigkeit kann dieses Ungleichgewicht ausgeglichen werden. Die Instruktionsmischung wird über die Programmausführung verwischt, so dass eine gleichmäßigere Ausnutzung der Prozessorressourcen erreicht wird.
- Durch die große Speicherraumnutzung einer Multimediaanwendungen entstehen ungleich höhere Cachefehlzugriffsraten als in einem konventionellen Programm. Durch die Fähigkeit eines mehrfädigen Prozessors die dadurch entstehenden hohen Latenzen auszugleichen, kann die sonst ungenutzte Prozessorzeit sinnvoll genutzt werden.
- Einzelne Multimediaalgorithmen lassen sich meist sehr leicht parallelisieren, da sie entweder große unabhängige Speicherbereiche bearbeiten, oder aber eine relativ grobe Fließbandstruktur aufweisen. Dies kommt der Anforderung eines mehrfädigen Prozessors nach der gleichzeitigen Ausführung mehrere Kontrollfäden sehr entgegen.
- Gesamte Multimediaanwendungen oder -systeme bestehen meist aus unabhängigen Datenströmen. Diese müssen im allgemeinen gleichzeitig bearbeitet und synchron präsentiert werden. Dies kann durch einen mehrfädigen Prozessor in idealer Weise erreicht werden, da dieser aufgrund seiner Struktur mehrere Lasten zugleich ausführen kann.
- In Multimediaanwendungen treten durch die Bearbeitung verschiedener Datenströme mit meist hohen Datenraten sehr viele Programmunterbrechungen zur Ansteuerung externer Systemelemente wie zum Beispiel Laufwerken oder Graphikbeschleunigern auf. Durch diese Programmunterbrechungen entsteht meist ein hoher Kontextwechselaufwand, da der Prozessorzustand während der Ausführung der Unterbrechung gesichert sein muss. Dies kann von einem mehrfä-

digen Prozessor recht günstig implementiert werden, wenn ein Kontrollfaden explizit für die Ausführung dieser Programmunterbrechungen genutzt wird [37].

- Bei einigen Multimediaalgorithmen (zum Beispiel dreidimensionale, perspektivische Darstellung) treten Fließkommaoperationen als hauptsächlicher Befehlstypus in den Vordergrund der Programmlast. Da diese Befehle meist eine Latenz mehrerer Takte aufweisen kommt dies der Fähigkeit des mehrfädigen Prozessors zur Latenznutzung entgegen.
- Durch die häufige Benutzung von SIMD Anweisungen in vielen Multimediaroutinen werden Möglichkeiten zur Nutzung der Parallelität auf Anweisungsebene genommen. So lassen sich zum Beispiel Schleifen häufig nicht mehr nutzbringend ausrollen, da durch die gleichzeitige Ausführung mehrerer Schleifendurchläufe mit Hilfe der SIMD Anweisungen, die Anzahl der Schleifendurchläufe bereits stark vermindert wurde. Ein mehrfädiger Prozessor kann die dadurch entstehenden zahlreichen ungenutzten Ausführungsmöglichkeiten durch andere Kontrollfäden nutzen.

Neben diesen Vorteilen stehen auch einige durch die Mehrfädigkeit verursachte Probleme dem Konzept entgegen.

- Durch die Verwendung mehrere Kontrollfäden kann die Ausführungszeit einzelner Kontrollfäden undeterministisch verändert werden. Dies steht einer Verwendung in einer Echtzeitanwendung entgegen. Durch die Verwendung prioritätsgesteuerter Prozessoreinheiten kann dieser Effekt umgangen werden. Eine weitere Möglichkeit die Ausführungsgeschwindigkeit vorhersehbarer zu gestalten besteht in einer Beschränkung der maximalen Ressourcenzuordnung an einzelne Kontrollfäden, so dass für jeden Ausführungskontext eine minimale Ressourcenmenge garantiert ist. Bei extremen Echtzeitanforderungen einzelner Programmteile könnte, falls diese nur zu einem geringen Anteil an der Gesamtausführungszeit beteiligt sind, während der Ausführung dieser Programmteile auf die parallele Ausführung anderer Kontrollfäden verzichtet werden, und somit ein dem einfädigen Prozessor gleichwertiges Verhalten erreicht werden.
- Mehrfädige Prozessoren haben eine deutlich schlechtere Vorhersehbarkeit des Daten-Caches. Dies kann zu ebenfalls dazu führen, dass das Programmverhalten deutliche Schwankungen der Ausführungsgeschwindigkeit zeigt. Durch die Verwendung geteilter Caches oder der Implementierung prioritätsgesteuerter Ersetzungsstrategien könnte dieses Problem umgangen werden.
- Multimediaanwendungen sind meist sehr speicherlastig. Dies könnte dazu führen, dass eine Mehrfädigkeit des Prozessors keinen Nutzen bringt, da der Speicherbus, der nicht vervielfältigt wird, bereits den Flaschenhals des Systems bildet. Durch die Verwendung längerer Speicherzugriffsfolgen, oder zusätzlichen Speichers auf dem Prozessorchip, kann dieser Flaschenhals entschärft werden.

Werden diese Nachteile bzw. Limitierungen bei einem Prozessorentwurf beachtet und durch entsprechende Gegenmaßnahmen entschärft, so können ihre Auswirkungen gemildert oder sogar eliminiert werden. Somit stünden einer großen Zahl an Vorteilen nur geringe Nachteile entgegen.

4.5 Beschreibung des simulierten Prozessors

Der in dieser Arbeit untersuchte Prozessor (genauer die „Prozessorfamilie“) vereint das „Simultaneous Multithreading“ mit Multimediaerweiterungen.

Im Multimediabereich unterstützt der Prozessor SIMD Operationen, Saturierungsarithmetik und erweiterte Multiplikationsbefehle. Als Ergänzung zum Datencache stehen noch mehrere KB an lokalem Speicher zur Verfügung, der mit einer festen Latenz gelesen und geschrieben werden kann. Die Verwendung von zusätzlichem lokalem Speicher ist eine bei DSPs häufig anzutreffende Erweiterung, um die Ausführungsgeschwindigkeit bei benötigter Echtzeitfähigkeit zu erhöhen[8] und Maximalzeiten zu garantieren.

Jeder Kontrollfaden des Prozessors besitzt 32 unabhängige 32-Bit Register. Die Kontrollfäden teilen sich die Befehlslade- (Fetch), Befehlsdekodier- (Decode), Befehlszuordnungs- (Issue) und Befehlsrückordnungs- (Completion) Einheiten sowie die Ausführungseinheiten (Execution), Umordnungspuffer (Reservationstations) und Umordnungsregister (Rename Register).

Zur Kontrollfadensteuerung steht eine Klasse von Kontrollanweisungen und eine zugehörige Ausführungseinheit bereit. Die Mehrfädigkeit wird durch einen sehr leichtgewichtigen „Nanokernel“ auf beliebig viele virtuelle Kontrollfäden erweitert. Diese Erweiterung auf virtuelle Kontrollfäden erlaubt es, eine beliebig mehrfädige Last auf verschiedenen Prozessorkonfigurationen zu testen.

[8] Ein Cache kann dieses nicht leisten, da er aufgrund der Möglichkeit eines Fehlzugriffes keine verbesserte maximale Zugriffszeit bietet, sondern lediglich die durchschnittliche Zugriffszeit vermindert. Er kann sogar die maximale Zugriffszeit verlängern, falls zur Belegung einer neuen Cache-Zeile zuerst noch der Inhalt einer alten Cache-Zeile gesichert werden muss.

5 Parallelisierbarkeit von Multimediaanwendungen am Beispiel MPEG-2-Dekodierung

5.1 Überblick

MPEG-2 ist der aktuell in der Industrie verwendete digitale Kompressionsstandard für kombinierte Audio- und Videodaten. Er findet Verwendung in DVD[9], DVB[10]sowie zahlreicher Spezialanwendungen (z.B. Überwachungsanlagen, Kioskanwendungen etc.) Der MPEG-2-Standard wurde aus dem MPEG-1-Standard entwickelt und ist mit kleinen Ausnahmen aufwärtskompatibel zu diesem. Es kamen einige Erweiterungen hinzu:

- Bildauflösungen mit mehr als 4096 Bildpunkten Breite/Höhe
- Unterstützung von Bildmaterial im Zeilensprungverfahren („Interlace“)
- „Extensibility“ : Ein Videostrom kann aus einer Basisebene sowie Erweiterungsebenen bestehen, die falls sie verwendet werden, die Qualität erhöhen können.
- Mehrkanalton (MPEG-1 unterstützt hier lediglich Stereo)

Der MPEG-2-Standard definiert die Kompression und Dekompression von Video- und Audiodaten[11], sowie das Verschränken („Multiplexing“) und die Synchronisierung dieser Daten.

5.2 Aufschlüsselung des MPEG-2-Video Dekodierungsprozesses

5.2.1 Überblick

Der MPEG-2-Kompressionsstandard definiert ein Verfahren zur verlustbehafteten Komprimierung von bewegten Bilddaten. Er basiert auf dem älteren MPEG-1-Standard und bietet Erweiterungen speziell für die Komprimierung von Halbbilddaten, wie sie im Fernsehbereich üblich sind. Die Kompressionsrate der Daten hängt von dem verwendeten Ausgangsmaterial, sowie der gewünschten

[9] DVD : „Digital Versatile Disc“, eine Erweiterung des CD Standards für Datenträger mit bis zu 20-Gbyte, meist für Videodaten genutzt.
[10] DVB : „Digital Video Broadcast“, das Übertragen von Fernsehsendungen nicht als Analogsignal, sondern in der Form komprimierter Videodaten. Dieses findet sowohl über Sateliten, durch Kabel aus auch terrestrisch statt.
[11] Obwohl die MPEG 1 und 2 Standards auch die Audiokompression beinhalten, wird im amerikanischen Raum meist das proprietäre Dolby Digital (AC 3) verwendet.

Qualität ab. Typische Kompressionsraten[12] liegen im Bereich von 1 zu 80 bis 1 zu 16. Diese Kompression wird durch mehrere Schritte erreicht.

- Bewegungskompensation
- Quantisierung von Frequenzwerten
- Lauflängenkodierung
- Huffman-Kodierung

Unter Bewegungskompensation versteht man ein Verfahren, bei dem Teile aus zeitlich früheren und / oder späteren Bildern zur Dekompression des aktuellen Bildes herangezogen werden. Da diese Teile von anderen Stellen der Referenzbilder stammen können, kann so eine Bewegung im Bild ausgeglichen werden.

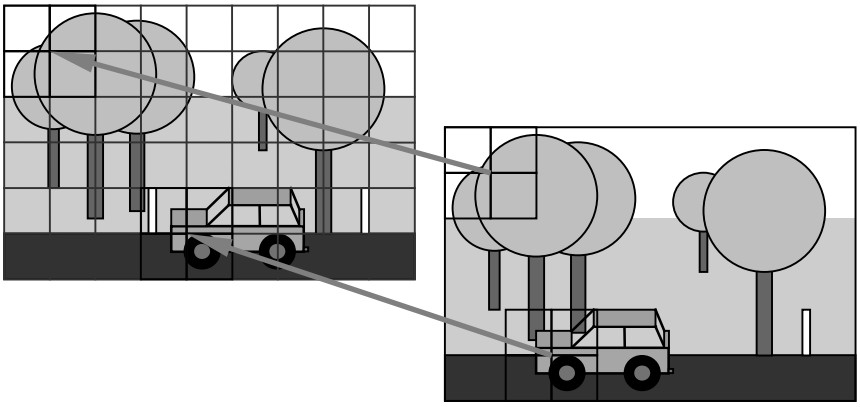


Abbildung 5-1: Beispiel für eine Bewegungskompensation

Ein Bild wird während der Kompression in Quadrate der Größe 16x16 Bildpunkte (Makroblock) aufgeteilt, die getrennt voneinander bearbeitet werden. Jeder dieser Makroblöcke besteht wieder aus sechs quadratischen Blöcken der Größe 8x8 (vier für die Helligkeits- und zwei für die Farbdaten). Die Bewegungskompensation findet auf Makroblockebene statt, alle weiteren Schritte der Kompression auf Blockebene. Es werden drei verschiedene Arten von Makroblöcken unterschieden:

- „intra“, alle Blöcke sind vorhanden, keine Bewegungskompensation
- „pattern“, einige Blöcke sind vorhanden, Bewegungskompensation
- „not-coded“, kein Block ist vorhanden, Bewegungskompensation

Blöcke die nicht vorhanden sind, werden als vollständig mit 0 belegt betrachtet.

5.2.2 Zergliederung des Datenstromes

Die einzelnen Teilströme (Audio Video etc.), die einen MPEG2 Strom ausmachen, sind in verschränkter Form (multiplexed) im eigentlichen Datenstrom gespeichert. Dies geschieht, um die Daten, die zur gleichzeitigen Präsentation vorgesehen sind, in physikalisch benachbarten Bereichen des

[12] Unter Annahme einer NTSC-Quelle im CCIR-601 mit 4:2:2 Unterabtastung („Subsampling“) ergibt sich eine Bitrate von 720*480*16*29.97 = 165.722.122 Bits/sec. Typische MPEG 2 Videos liegen im

Datenträgers abzuspeichern. Bevor die Teilströme getrennt entpackt und präsentiert werden können, müssen sie erst wieder aus dem geschachtelten Strom extrahiert werden. Die einzelnen Teilströme sind hierbei in Pakete gleicher Länge (VP - Videopaket, AP - Audiopaket und IP - Informationspaket) zerteilt, die durch eine jeweils unterschiedliche Markierung (PKH : „Packet-Header“) am Anfang jedes Paketes erkennbar sind. Diese Pakete werden dann zu einem Gesamtstrom gemischt. Hierbei werden die verschiedenen Datenraten der Teilströme entsprechend berücksichtigt. Vereinfachend kommt hier im Falle DVD (im Gegensatz zum allgemeinen MPEG-2-Standard) hinzu, dass alle Pakete eine feste Länge in der Größe eines Sektors der DVD besitzen.

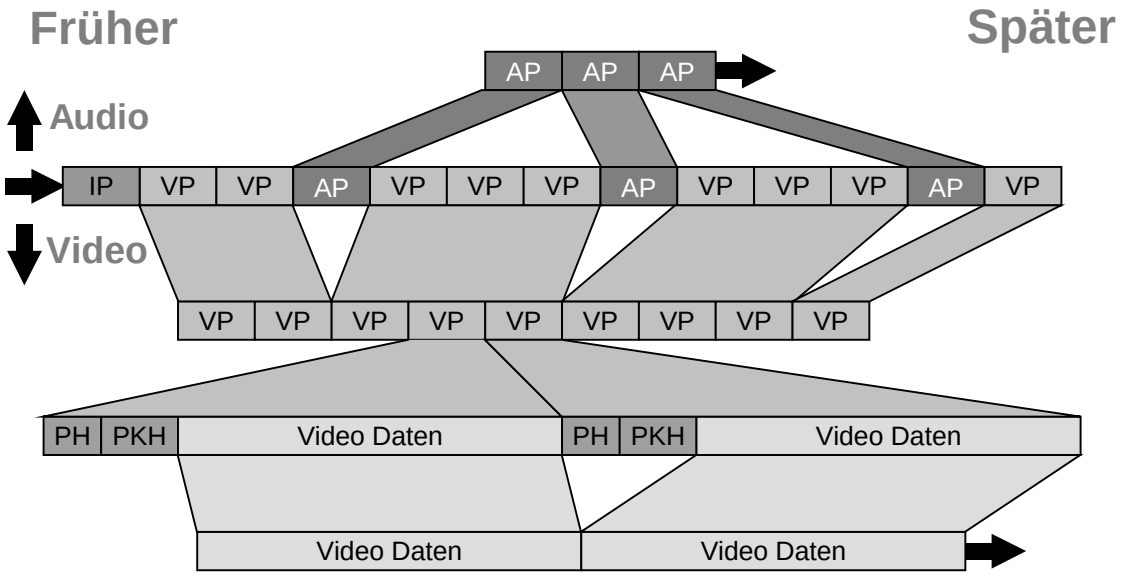


Abbildung 5-2: Aufgliederung eines MPEG Programstreams

Nachdem die Pakete der Teilströme aufgetrennt sind, können die eigentlichen Teilströme aus diesen rekonstruiert werden. Um eine synchrone Präsentation der Teilströme zu gewährleisten, befinden sich in den Verwaltungsstrukturen der Datenpakete Zeitstempel. Diese müssen während des Trennvorganges aus den Paketen extrahiert werden, um nach der Dekompression der Teilströme zur Synchronisation verwendet zu werden.

5.2.3 Parser

Die Aufgabe des Parsers im Dekodierungsvorgang ist es, die gemäß der syntaktischen Definition des Kompressionsstandards gespeicherten Bild- und Verwaltungsdaten aus dem Datenstrom herauszuziehen, um diese in ein für die weitere Bearbeitung geeignetes Format zu wandeln.

5.2.3.1 Datenstrom-Parser

Ein MPEG-2-Strom wird in einzelne hierarchische Segmente unterteilt gespeichert:

Bereich von 2 bis 10 Mbits/sec.

```
MPEGStream            ::= {MPEGSequence}
MPEGSequence          ::= SequenceHeader {Extensions} {GroupOfPicture}
GroupOfPicture        ::= GOPHeader [GOPExtension] {MPEGPicture}
MPEGPicture           ::= PictureHeader {PictureExtension} {Slice}
Slice                 ::= SliceHeader {Macroblock}
Macroblock            ::= MacroblockHeader {Block}
...
```

Bis auf Slice-Ebene hinunter sind alle zu bearbeitenden Datenstrukturen von fester Länge und nicht Huffman kodiert. Alle diese Strukturen starten zudem auf Byte-Grenzen, so dass der Parser in dieser Ebene sehr einfach gehalten werden kann.

5.2.3.2 Bild-Parser

Ein Videostrom wird als Folge einzelner Bilder gespeichert. Es werden drei verschiedene Arten von Bildern unterschieden:

- I-Bilder („Intra coded“) Bilder, die unabhängig von den anderen Bildern rekonstruiert werden können.
- P-Bilder („Predictive coded“) Bilder, die nur in Abhängigkeit von einem vorhergehenden Bild dekodiert werden können.
- B-Bilder „(Bidirectional predictive coded“) Bilder, die nur in Abhängigkeit von einem vorhergehenden und einem nachfolgenden Bild rekonstruiert werden können.

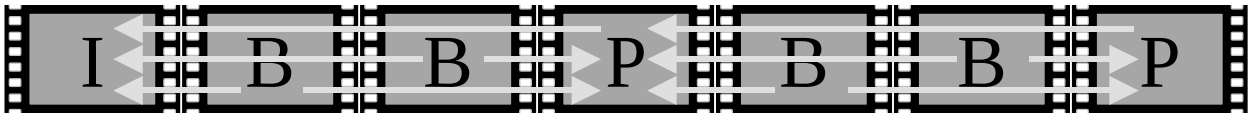


Abbildung 5-3: Beispiel für die Bildabhängigkeiten in einem MPEG-2-Strom

Die Verwendung von B-Bildern hat zur Folge, dass die Bilder nicht in ihrer Präsentationsreihenfolge gespeichert oder gesendet werden können, da zur Rekonstruktion dieser Bilder andere bereits rekonstruiert sein müssen, die von der Präsentationsfolge her zeitlich später liegen. Eine typische Bildfolge in einem Strom wäre etwa diese: $I_0P_3B_1B_2P_6B_4B_5P_9B_7B_8P_{12}B_{10}B_{11}P_{15}B_{13}B_{14}$.

5.2.3.3 Slice-Parser

Ein MPEG-Bild wird in mehrere unabhängige horizontale Streifen („Slice“) zergliedert. Dies geschieht, um bei einem geringen Verlust von Daten nur einen Teilverlust eines Bildes zu erleiden. Jedes Slice kann ohne Informationen aus den vorhergehenden Slices dekodiert werden. Typischerweise besteht ein Slice aus einer kompletten Makroblock-Zeile, kann aber auch nur einen Teil davon ausmachen. Es ist in MPEG-2 im Gegensatz zu MPEG-1 nicht gestattet, dass sich eine Slice über mehrere Makroblock-Zeilen erstreckt.

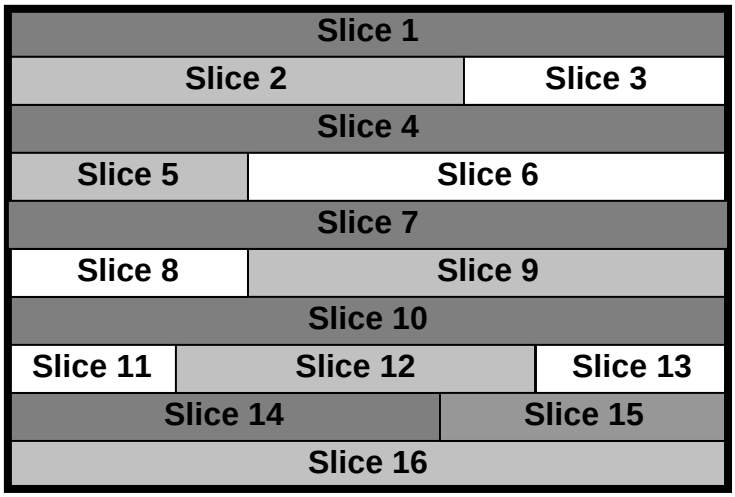


Abbildung 5-4: Beispiel einer Slice-Struktur in einem MPEG-2 Bild

Slices beginnen im Datenstrom immer auf einer ganzen Byte-Adresse mit einem eindeutigen Startcode, so dass sich im Fehlerfall relativ leicht und effizient der Start des nächsten Slice finden lässt.

5.2.3.4 Makroblock-Parser

5.2.3.4.1 Huffman- und Lauflängenkodierung

Da sich innerhalb der Blöcke häufig Koeffizienten des Wertes 0 befinden, lohnt es sich eine Lauflängenkodierung für diese Koeffizienten einzuführen. Jeder Koeffizient wird als zwei-Tupel von Nullfolgenlauflänge und abschließendem von Null verschiedenem Wert gespeichert. Für die häufigsten dieser Tupel ist ein eigener Huffmancode definiert, alle anderen Tupel werden über einen Ausnahmecode und den Werten des Tupels gespeichert.

5.2.3.4.2 Inverses ZigZag-Anordnen

Um längere Nullfolgen zu erreichen und somit den Gewinn durch die Lauflängenkodierung zu erhöhen, werden die einzelnen Koeffizienten nicht in der natürlichen Tabellenordnung, sondern nach einem Zig-Zag Schema abgespeichert. Dies ist deshalb von Vorteil, da die Koeffizienten höherer Frequenz (siehe 5.2.4.2) üblicherweise stärker quantisiert werden als die Koeffizienten niedriger Frequenz und diese Anordnung dafür sorgt, dass die Koeffizienten nach aufsteigender Frequenz sortiert behandelt werden.

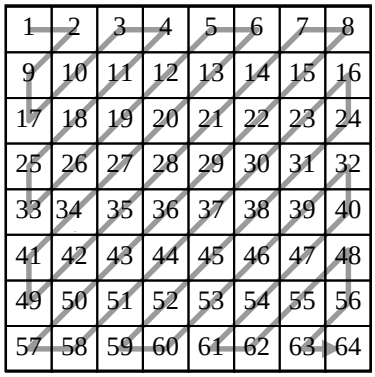


Abbildung 5-5: Ablaufordnung innerhalb eines IDCT Blockes

5.2.4 Dekodierung

5.2.4.1 Dequantisierung

Die Koeffizienten der DCT (F''(u,v)) werden in quantisierter Form (QF(u, v)) gespeichert. Dies bedeutet, sie werden bei der Kompression durch eine Konstante dividiert. Dies ist der eigentliche verlustbehaftete Schritt der MPEG Kompression, da hier Genauigkeit der Koeffizienten verloren wird. Bei der Dekompression des Datenstromes müssen die Koeffizienten wieder mit dieser Konstanten multipliziert werden. Die Konstante ist hierbei von drei Faktoren abhängig, der Position im Block (u, v) , einem Quantisierungsfaktor (quantizer_scale) für den gesamten Block und dem Typ des Blocks (intra-block, w). Der DC (nicht frequenzbehaftete) Koeffizient wird in intra-kodierten Blöcken speziell behandelt. Er wird nicht quantisiert, sondern in einem Delta-Verfahren (also als Abweichung vom Wert des vorherigen DC Koeffizienten) gespeichert.

$$F''(u,v) = \begin{cases} int\,ra\_dc\_mult \cdot QF(u,v) & u,v = 0 \wedge int\,ra - block \\ \dfrac{((2 \cdot QF(u,v) + k) \cdot W(u,v,w) \cdot quantizer\_scale)}{32} & u \neq 0 \vee v \neq 0 \vee non - int\,ra - block \end{cases}$$

$$k = \begin{cases} 0 & int\,ra - blocks \\ sign(QF(u,v) & non - int\,ra - blocks \end{cases}$$

Der Summand „k“ dient der Rundung. Er findet nur in nicht intra-kodierten Blöcken Anwendung. Die Quantisierungsmatrizen W(u,v,w) sind jeweils für eine Sequenz mehrerer Bilder festgelegt. Sie schwanken typischerweise nicht während eines Stromes. Der Quantisierungsfaktor und der Typ des Blockes können für jeden Makroblock gewechselt werden.

5.2.4.2 Inverse DCT

Die MPEG Kompression basiert wie auch die JPEG Kompression darauf, dass das menschliche Auge für höhere Frequenzen weniger stark empfindlich ist als für niedrigere Frequenzen[13]. Um dies

[13] Auf demselben Prinzip beruhen auch wavelett-basierte Verfahren, nur dass bei diesen das Bild als ganzes frequenzkonvertiert wird, wogegen bei JPEG und MPEG immer kleine 8x8 Bildelemente große

ausnutzen zu können, muss das Bild im Frequenzraum vorliegen. Hierzu wird eine diskrete Cosinus-Transformation (DCT) verwendet. Sie ist das Gegenstück der diskreten Fourier-Transformation [8] im Raum der reellen Zahlen. Für Bildbearbeitung wird die zweidimensionale Variante der DCT benötigt. Die Transformation aus dem Zeit- in den Frequenzraum ist wie folgt definiert:

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

$$C(u), C(v) = \begin{cases} \frac{1}{\sqrt{2}} & u,v = 0 \\ 1 & sonst \end{cases}$$

Für die Dekompression wird die inverse DCT benötigt:

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

```
for (x=0; x<N; x++)
   {
   for (y=0; y<N; y++)
      {
      sum = 0;

      for (u=0; u<N; u++)
         for (v=0; v<N; v++)
            sum += C[u] * C[v] * F[u][v] * cos((2*x+1)*u*pi/(2*N)) * cos((2*y+1)*v*pi/(2*N))

      f[x][y] = sum * 2 / N;
      }
   }
```

Diese Form mit einem Aufwand von $O(N^4)$ wird im allgemeinen nicht verwendet, statt dessen wird folgende Variante mit $O(N^3)$ verwendet. Sie basiert darauf, dass eine zweidimensionale DCT durch zwei aufeinander folgende eindimensionale DCTs ersetzt werden kann.

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

$$g_G(k) := \sum_{i=0}^{N-1} C(i) \cos\frac{(2k+1)i\pi}{2N} G(i)$$

$$F_u(v) := F(u,v)$$

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N-1} C(u) \cos\frac{(2x+1)u\pi}{2N} g_{F_u}(y)$$

$$f(x,y) = \frac{2}{N} g_{g_{F_u}(y)}(x)$$

Blöcke betrachtet werden. Der Vorteil der Wavelett Kompression liegt darin, dass die bei MPEG und JPEG durch diese Aufteilung entstehenden Blockartefakte (Unstetigkeiten an der Grenze zweier Blöcke) im Bild nicht entstehen.

```
for (x=0; x<N; x++)
   {
   for (y=0; y<N; y++)
      {
      sum = 0;

      for (i=0; i<N; i++)
         sum += C[i] * F[x][i] * cos((2*y+1)*i*pi/(2*N));

      t[x][y] = sum;
      }
   }

for (x=0; x<N; x++)
   {
   for (y=0; y<N; y++)
      {
      sum = 0;

      for (i=0; i<N; i++)
         sum += C[i] * t[i][y] * cos((2*x+1)*i*pi/(2*N));

      t[x][y] = sum * 2 / N;
      }
   }
```

Unter Verzicht auf Parallelität kann dieses Verfahren nun noch weiter verbessert werden[14]. Dazu betrachtet man die verwendeten Koeffizienten t(k,i) (es wird hier nur noch der Fall N=8 betrachtet, da dies der einzige für MPEG interessante Fall darstellt.)

$$t(k,i) = C(i)\cos\frac{((2k+1)i\pi}{16}$$

k\i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	$\frac{1}{\sqrt{2}}$	$\cos\frac{\pi}{16}$	$\cos\frac{2\pi}{16}$	$\cos\frac{3\pi}{16}$	$\frac{1}{\sqrt{2}}$	$\cos\frac{5\pi}{16}$	$\cos\frac{6\pi}{16}$	$\cos\frac{7\pi}{16}$
1	$\frac{1}{\sqrt{2}}$	$\cos\frac{3\pi}{16}$	$\cos\frac{6\pi}{16}$	$-\cos\frac{7\pi}{16}$	$-\frac{1}{\sqrt{2}}$	$-\cos\frac{\pi}{16}$	$-\cos\frac{2\pi}{16}$	$\cos\frac{5\pi}{16}$
2	$\frac{1}{\sqrt{2}}$	$\cos\frac{5\pi}{16}$	$-\cos\frac{6\pi}{16}$	$-\cos\frac{\pi}{16}$	$-\frac{1}{\sqrt{2}}$	$-\cos\frac{7\pi}{16}$	$\cos\frac{2\pi}{16}$	$\cos\frac{3\pi}{16}$
3	$\frac{1}{\sqrt{2}}$	$\cos\frac{7\pi}{16}$	$-\cos\frac{2\pi}{16}$	$-\cos\frac{5\pi}{16}$	$\frac{1}{\sqrt{2}}$	$\cos\frac{3\pi}{16}$	$-\cos\frac{6\pi}{16}$	$\cos\frac{\pi}{16}$
4	$\frac{1}{\sqrt{2}}$	$-\cos\frac{7\pi}{16}$	$-\cos\frac{2\pi}{16}$	$\cos\frac{5\pi}{16}$	$\frac{1}{\sqrt{2}}$	$-\cos\frac{3\pi}{16}$	$-\cos\frac{6\pi}{16}$	$-\cos\frac{\pi}{16}$
5	$\frac{1}{\sqrt{2}}$	$-\cos\frac{5\pi}{16}$	$-\cos\frac{6\pi}{16}$	$\cos\frac{\pi}{16}$	$-\frac{1}{\sqrt{2}}$	$\cos\frac{7\pi}{16}$	$\cos\frac{2\pi}{16}$	$-\cos\frac{3\pi}{16}$
6	$\frac{1}{\sqrt{2}}$	$-\cos\frac{3\pi}{16}$	$\cos\frac{6\pi}{16}$	$\cos\frac{7\pi}{16}$	$-\frac{1}{\sqrt{2}}$	$\cos\frac{\pi}{16}$	$-\cos\frac{2\pi}{16}$	$-\cos\frac{5\pi}{16}$
7	$\frac{1}{\sqrt{2}}$	$-\cos\frac{\pi}{16}$	$\cos\frac{2\pi}{16}$	$-\cos\frac{3\pi}{16}$	$\frac{1}{\sqrt{2}}$	$-\cos\frac{5\pi}{16}$	$\cos\frac{6\pi}{16}$	$-\cos\frac{7\pi}{16}$

Man erkennt, dass lediglich sieben verschieden Konstanten verwendet werden, die dazu auch noch auf einzelne Koeffizienten beschränkt sind. Es lässt sich somit ein Algorithmus mit deutlich geringerem Aufwand schaffen. Eine weitere Vereinfachung ergibt sich noch durch:

$$\overline{g}_G(k) := \sqrt{2}\, g_G(k)$$

$$f(x,y) = \frac{1}{N}\,\overline{g}_{\overline{g}_{F_u(y)}}(x)$$

Somit ergibt sich folgendes Programmfragment:

[14] Das Prinzip ist hier das selbe, das auch für die FFT verwendet wird.

```
static const double C0 = sqrt(2.);
static const double C1 = cos(1./16.) * C0;
static const double C2 = cos(2./16.) * C0;
static const double C3 = cos(3./16.) * C0;
static const double C4 = cos(4./16.) * C0;
static const double C5 = cos(5./16.) * C0;
static const double C6 = cos(6./16.) * C0;
static const double C7 = cos(7./16.) * C0;

for (y=0; y < 8; y++)
   {
   y0 = f[y][0] + f[y][4];
   y1 = f[y][0] - f[y][4];
   y2 = f[y][2] * C2 + f[y][6] * C6;
   y3 = f[y][2] * C6 - f[y][6] * C2;
   a0 = y0 + y2;
   a1 = y1 + y3;
   a2 = y1 - y3;
   a3 = y0 - y2;
   y4 = f[y][1] * C1 + f[y][3] * C3 + f[y][5] * C5 + f[y][7] * C7;
   y6 = f[y][1] * C3 - f[y][3] * C7 - f[y][5] * C1 - f[y][7] * C5;
   y5 = f[y][1] * C7 - f[y][3] * C5 + f[y][5] * C3 - f[y][7] * C1;
   y7 = f[y][1] * C5 - f[y][3] * C1 + f[y][5] * C7 + f[y][7] * C3;
   t[0][y] = a0 + y4;
   t[1][y] = a1 + y6;
   t[2][y] = a2 + y7;
   t[3][y] = a3 + y5;
   t[4][y] = a3 - y5;
   t[5][y] = a2 - y7;
   t[6][y] = a1 - y6;
   t[7][y] = a0 - y4;
   }

for (y=0; y < 8; y++)
   {
   y0 = t[y][0] + t[y][4];
   y1 = t[y][0] - t[y][4];
   y2 = t[y][2] * C2 + t[y][6] * C6;
   y3 = t[y][2] * C6 - t[y][6] * C2;
   a0 = y0 + y2;
   a1 = y1 + y3;
   a2 = y1 - y3;
   a3 = y0 - y2;
   y4 = t[y][1] * C1 + t[y][3] * C3 + t[y][5] * C5 + t[y][7] * C7;
   y6 = t[y][1] * C3 - t[y][3] * C7 - t[y][5] * C1 - t[y][7] * C5;
   y5 = t[y][1] * C7 - t[y][3] * C5 + t[y][5] * C3 - t[y][7] * C1;
   y7 = t[y][1] * C5 - t[y][3] * C1 + t[y][5] * C7 + t[y][7] * C3;
   F[0][y] = (a0 + y4) / 8;
   F[1][y] = (a1 + y6) / 8;
   F[2][y] = (a2 + y7) / 8;
   F[3][y] = (a3 + y5) / 8;
   F[4][y] = (a3 - y5) / 8;
   F[5][y] = (a2 - y7) / 8;
   F[6][y] = (a1 - y6) / 8;
   F[7][y] = (a0 - y4) / 8;
   }
```

Es sind in der Literatur noch weitere zweidimensionale IDCT Varianten mit fester Blockgröße bekannt, die mit deutlich weniger Rechenoperationen auskommen oder Multiplikationen durch Additionen ersetzen [7][80][17]. Da diese aber eine höhere Genauigkeit als die gegebenen 16 Bit verlangen und durch ihre ungleichmäßige Struktur einer Implementierung durch SIMD Anweisungen entgegenstehen, wurde auf deren Beschreibung und Verwendung in dieser Arbeit verzichtet.

5.2.4.3 Bewegungskompensation

Bei der Bewegungskompensation werden Teile aus zeitlich früheren und/oder späteren Bildern zur Rekonstruktion des aktuellen Bildes mit verwendet. Diese Bewegungskompensation findet auf Mak-

roblock- oder Halbmakroblockebene[15] statt. Die Koordinaten des Ursprungsrechteckes können hierbei auf halbe Pixel genau angegeben werden. In einem solchen Fall werden die Werte der umgebenden Pixel gemittelt.

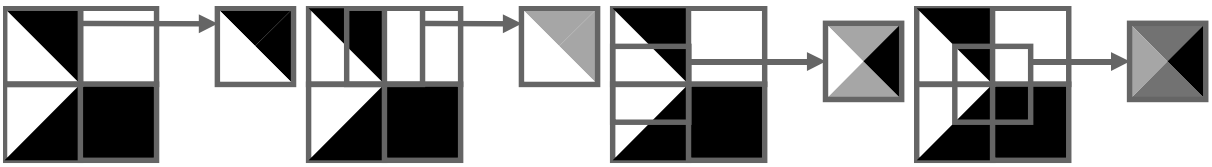


Abbildung 5-6: Verschiedene Formen der Halbpixelkompensation

$$p'(x,y)=\begin{cases} p(x/2,y/2) & x\&1=0, y\&1=0 \\ \dfrac{p(x/2,y/2)+p(x/2+1,y/2)}{2} & x\&1=1, y\&1=0 \\ \dfrac{p(x/2,y/2)+p(x/2,y/2+1)}{2} & x\&1=0, y\&1=1 \\ \dfrac{p(x/2,y/2)+p(x/2+1,y/2)+p(x/2,y/2+1)+p(x/2+1,y/2+1)}{4} & x\&1=1, y\&1=1 \end{cases}$$

Da bei B-Bildern die Bewegungskompensation aus zwei verschiedenen Bildern stammt, werden im schlimmsten Fall bis zu acht Pixel der Ausgangsbilder benötigt, um einen Pixel des Zielbildes zu berechnen.

Die Bewegungsvorhersage kann auf Vollbildern oder Halbbildern beruhen:

Abbildung 5-7: Unidirektionale Vollbildbewegungskompensation

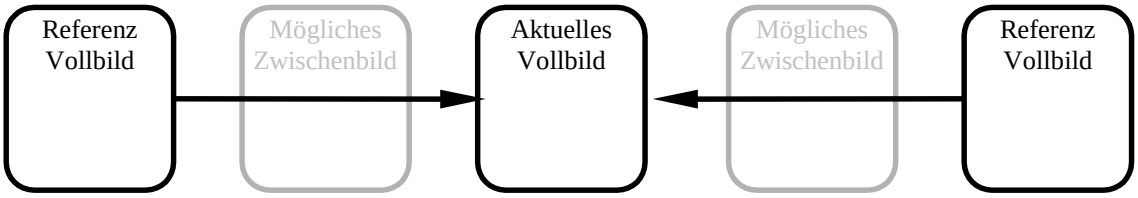


Abbildung 5-8: Bidirektionale Vollbildbewegungskompensation

[15] Ein Halbmakroblock kann entweder durch die Aufteilung eines Makroblocks in eine obere und untere Hälfte oder in die geraden und ungeraden Zeilen entstehen.

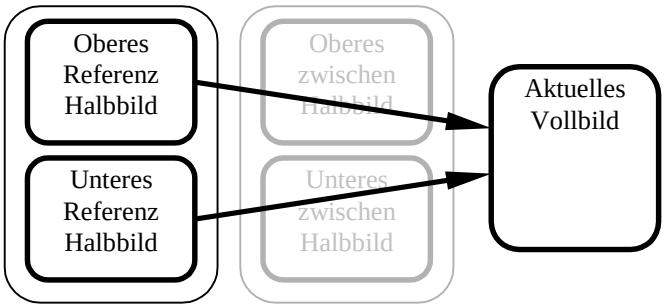


Abbildung 5-9: Unidirektionale Halbbildbewegungskompensation

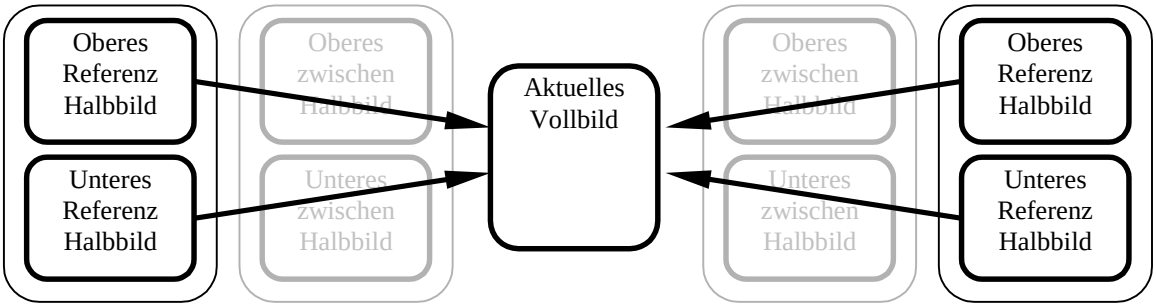


Abbildung 5-10: Bidirektionale Vollbildbewegungskompensation

Bei der Verwendung von Halbbildern wird jeder Makroblock in zwei Blöcke halber Höhe unterteilt. Hierbei besteht jeder dieser halbhohen Blöcken entweder aus den geraden oder ungeraden Zeilen des originalen Makroblockes. Dies ermöglicht es, halbbildorientierte Bildsequenzen, wie sie in normalen Fernseh- und Videofilmen verwendet werden, effektiv zu kodieren. Da sich in diesen Sequenzen die Halbbilder in zeitlicher Verschiebung befinden, wäre es unsinnig, die Bewegungskompensation auf Vollbildebene durchzuführen.

5.2.5 Darstellung und Nachbearbeiten

Die Darstellung der Bildsequenzen teilt sich in mehrere Aufgaben:

1. Umordnen der Bilder, da diese nicht in der Darstellungsreihenfolge im Strom gespeichert sind.
2. Synchronisation auf die Bildwiederholfrequenz. Da die Dekodiergeschwindigkeit nicht immer genau der Darstellungsgeschwindigkeit entspricht, muss dies durch einen Puffer erreicht werden.
3. Farbraumkonvertierung. Da MPEG-Bilder im YUV- und nicht im RGB-Farbraum gespeichert sind, muss eine Konvertierung stattfinden. Dies ist aber meist in den Graphikkarten als Funktion integriert, so dass eine eigene Konvertierung durch den Prozessor entfallen kann.
4. Häufig sind MPEG-2-Sequenzen im Zeilensprungverfahren kodiert („interlaced“), sollen aber auf einem progressiven Anzeigegerät, zum Beispiel einem Computer-Monitor, angezeigt werden. Dazu müssen die Bildhälfte der geraden und die der ungeraden Zeilen in zwei komplette Vollbilder rückkonvertiert werden. Dies kann recht einfach durch eine Interpolation der fehlenden Zeilen oder auch sehr aufwendig durch temporale und lokale Bewegungssuche geschehen.

5. Nicht immer verfügen Anzeigegeräte über alle gewünschten Einstellmöglichkeiten, wie zum Beispiel Helligkeit, Farbsättigung oder Kontrast.

5.3 Parallelität im MPEG-2-Dekodierungsprozess

5.3.1 Pipelineparallelität

Der MPEG-2-Dekodierungsprozess lässt sich sehr einfach als Fließband modellieren.

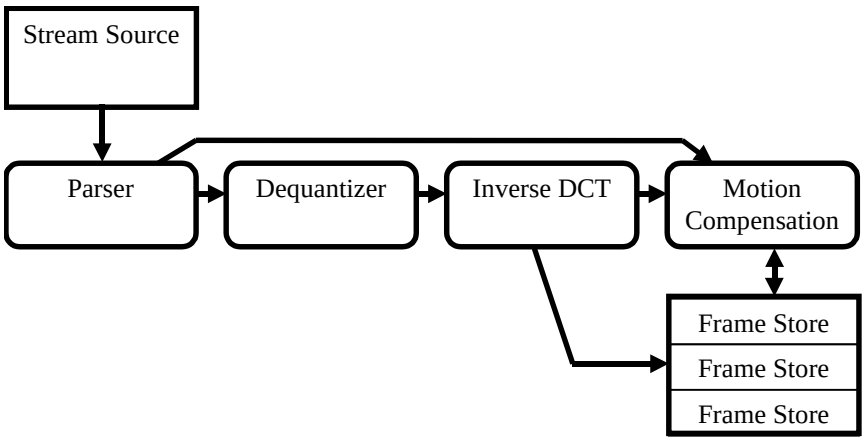


Abbildung 5-11: Fließbandmodell einer MPEG-Videodekodierung

Die kleinste Einheit, die hierbei durch die Pipeline gesendet wird, ist ein Makroblock. Ein Makroblock besteht aus sechs Blöcken zu je acht mal acht Bildelementen (Pixel oder PEL). Die ersten vier Blöcke enthalten Luminanz-Daten, die beiden folgenden Chrominanz-Daten. Jede der Fließbandstufen kann unabhängig von den anderen arbeiten, da keine Information in vorhergehende Stufen zurückgegeben werden muss. Es handelt sich also um vier echte Konsumenten-/ Produzenten-Beziehungen, so dass eine gute Entkopplung gegeben ist.

Je nach Art eines Makroblocks werden einzelne Pipelinestufen übersprungen, so dass nicht immer eine gleichmäßige Auslastung der Pipeline gegeben ist.

Blocktyp	**Häufigkeit**	**Dequantisierung**	**Inverse DCT**	**Bewegungskompensation**
intra	10%	ja	ja (6)	Nein
pattern	80%	teilw.	teilw. (≈4)	Ja
not coded	10%	nein	nein (0)	Ja

5.3.2 Vervielfältigung der Pipelinestufen

Aufgrund der Verschiedenartigkeit der Pipelinestufen ergeben sich verschiedene Möglichkeiten, diese Stufen mehrfach parallel in der Pipeline zu verwenden. Die Vervielfältigung einer Pipelinestufe erhöht die Parallelität der Ausführung, wenn diese Stufe mehr Rechenzeit benötigt als die anderen Stufen.

5.3.2.1 Parser

Die Arbeit des Parsers ist in großen Bereichen in sich sequentiell. Die syntaktische Struktur eines MPEG-2-Datenstromes lässt sich in verschiedene Ebenen unterteilen.

- Sequenz, eine in sich geschlossene Abfolge von Bildern
- Group of Pictures, eine Gruppe von Bildern mit einem I-Frame
- Frame, ein komplettes Bild (also eventuell auch zwei Halbbilder)
- Slice, eine Makroblockzeile
- Makroblock, eine Gruppe aus vier Luminanz- und zwei Chrominanz-Blöcken
- Block

Auf Block- und Makroblockebene ist eine Parallelisierung nur schlecht möglich, da diese Teile komplett auf Bitebene gepackt sind, und die Länge dieser Abschnitte aufgrund der Huffmankodierung ihrer Elemente stark variiert. Es ist unmöglich, den Anfang des nächsten Blocks oder Makroblocks zu finden, ohne den vorherigen Block bzw. Makroblock komplett bearbeitet zu haben. Dazu kommt noch, dass jeder Makroblock in seinen Parametern von den Parametern des vorherigen Blocks abhängt. Dies gilt sowohl für Position, als auch Quantisierungsfaktor und Bewegungsvektoren.

Eine Parallelisierung auf Slice-Ebene scheint leichter möglich. Jedes Slice ist für sich in seinen Parametern von den vorherigen Slices unabhängig. Auch das Auffinden eines Slice-Starts ist einfach möglich. Jedes Slice beginnt auf einem Byte und ist durch einen eindeutigen Startcode gekennzeichnet. Problematisch ist lediglich, dass der hereinkommende Bitstrom nun zweifach gelesen wird, einmal um die Slice-Anfänge zu bestimmen und einmal für die Dekodierung. Ein weiteres Problem liegt darin, dass von mehreren verschiedenen Positionen im Quellstrom gleichzeitig gelesen wird.

Soll auf Bildebene parallelisiert werden, so entstehen nicht nur die Probleme, die auf Slice-Ebene entstehen, es werden auch mehrere Bildschirmspeicher benötigt, um die zusätzlichen Bilder gleichzeitig bearbeiten zu können. Da ein einzelnes dekodiertes NTSC Bild bereits über 500 Kilobytes benötigt, erscheint dies unnötig verschwenderisch. Dies wird bei der Betrachtung einer Parallelisierung auf GOP- oder Sequenz-Ebene noch deutlicher, da hier ganze Sequenzen dekodierter Bilder gespeichert werden müssten.

Die einzig sinnvolle Parallelisierung des Parsers scheint also auf Slice-Ebene gegeben zu sein. Da es allerdings im MPEG-1-Standard keine Vorschriften über die Häufigkeit von Slices in einem Frame gibt, kann nichts über den möglichen Grad der Parallelisierung gesagt werden. Da teilweise auch komplette Bilder als Slices codiert werden, kann es sein, dass kein einziger zusätzlicher Parserprozess möglich ist. Im MPEG-2-Standard können Slices nicht über Zeilengrenzen hinausreichen, so dass eine minimale Anzahl unabhängiger Slices pro Bild sicher gegeben ist.

5.3.2.2 Dequantisierung

Die Dequantisierung lässt sich auf Blockebene beliebig parallelisieren, da einzelne Blöcke gegenseitig keine Abhängigkeiten bezüglich der Dequantisierung besitzen. Je nach Bedarf könnten beliebig

viele Instanzen dieser Pipelinestufe geschaffen werden. Der Speicherbedarf pro Stufe hält sich in Grenzen, es werden pro Block jeweils 64 Elemente benötigt.

5.3.2.3 Inverse DCT

Die inverse DCT ist wie die Dequantisierung beliebig parallelisierbar. Da diese beiden Stufen auch auf denselben Daten arbeiten und auch immer nacheinander ausgeführt werden, erscheint es sinnvoll, zugunsten mehrerer paralleler Prozesse auf die Trennung in zwei getrennte Stufen zu verzichten. Dies vermindert den Synchronisationsaufwand. Auch könnten diese beiden Stufen in der Bearbeitung geschachtelt werden, so dass sich eine bessere Cache und Registernutzung ergeben könnte.

5.3.2.4 Bewegungskompensation

Die Bewegungskompensation lässt sich auf Makroblockebene beliebig skalieren, da jeder Makroblock bezüglich dieser Operation eine Einheit darstellt. Ein Problem in der Parallelisierung der Bewegungskompensation dürfte sich in der stark erhöhten Cache-Belastung negativ bemerkbar machen. Da diese Stufen die höchsten Anforderungen an den Bildspeicher stellen, dürfte dies eine Beschränkung der Parallelisierbarkeit darstellen.

5.3.3 Datenparallelität

Da Multimediaprozessoren die im Vergleich zu normalen Prozessoren höhere Leistungsfähigkeit stark aus der Nutzung der Datenparallelität ziehen, ist diese Ebene der Parallelität für unsere Betrachtung von besonderer Signifikanz.

5.3.3.1 Parser

Aufgrund der sequentiellen Natur des komprimierten Bild-Datenstromes lässt sich in dieser Stufe kaum Datenparallelität finden. Vorteilhaft würden sich lediglich die breiten Register (64 bzw. 128 Bit) der üblichen Multimediaprozessoren auswirken. Dies würde es erlauben, längere Sequenzen[16] des Bit - basierten Datenstromes zu analysieren, ohne dazwischen liegende Speicherzugriffe durchführen zu müssen.

5.3.3.2 Dequantisierung

Die Dequantisierung stellt eine in sich hoch datenparallele Operation dar. Mit Ausnahme des DC-Koeffizienten in intra-kodierten Makroblöcken werden alle Koeffizienten gleich und unabhängig voneinander bearbeitet. Da pro Koeffizient 16 Bit benötigt werden, erlaubt dies je nach Prozessor 2, 4 oder sogar 8fache Datenparallelität (abhängig von der Registerbreite).

[16] Die meisten Kontrollstrukturen eines MPEG-2-Datenstromes haben in 64-Bit Platz, so dass der Parser mit einer derartigen Registerbreite diese mit einem einzigen Speicherzugriff für den Datenstrom komplett bearbeiten könnte.

```
Loop:          Ldl           (L0), L1       ; zwei Koeffizienten holen
               add           L0, 1, L0
               ldl           (L8), L3       ; zwei Einträge aus Quant. Tabelle
               add           L8, 1, L8

               cmp.h         L1, r0, L2     ; Vorzeichen ermitteln
               add.h         L1, L1, L1
               add.h         L1, L2, L1
               mul.h         L1, L3, L1
               mul.h         L1, L9, L1
               asr.h         L1, 5, L1

               stl           L1, (L0, -1)   ; zwei Koeffizienten speichern

               sub           L9, 1, L9
               bne           L9, Loop(PC)
```

Mit dieser Anweisungsfolge[17] können pro Schleifendurchlauf (je nach Registerbreite) mehrere Koeffizienten zugleich bearbeitet werden.

Da die Koeffizientenmatrix im Allgemeinen aber sehr dünn besetzt ist, stellt sich die Frage, ob ein derartiges Verfahren sinnvoller ist als eine Dequantisierung während der Lauflängenkodierung, da dort nur jeder von Null verschiedene Koeffizient dequantisiert würde. Hierbei ist eine Abwägung zwischen der Last im kritischen Pfad und der Last im parallelen Programmteil zu treffen. Wird die Dequantisierung innerhalb der Lauflängendekodierung durchgeführt, wird die Gesamtlast reduziert, aber der kritische Pfad verlängert.

5.3.3.3 Inverse DCT

Je nach verwendetem Algorithmus bietet die inverse DCT eine verschieden starke Datenparallelität. Wird der direkte Ansatz gewählt, so ergibt sich eine einfache Parallelisierbarkeit. Für alle 64 Koeffizienten wird dieselbe Schleife durchlaufen. Die Berechnung der Koeffizienten ist hierbei nicht von den Berechnungen der anderen Koeffizienten abhängig, so dass sich dies beliebig parallelisieren lässt. Der Nachteil dieses Verfahrens ist die hohe Anzahl an Operationen, die benötigt wird. Obwohl es einige Implementierungen einer derartigen inversen DCT in Hardware gibt, scheint der Bedarf an Gattern nicht gerechtfertigt.

Auch der „Teile und Herrsche“-Ansatz verfügt über ein hohes Maß an Parallelität. Da die einzelnen Schleifendurchläufe unabhängig voneinander sind, können diese einfach parallel ausgeführt werden. Problematisch ist hierbei allerdings die Anordnung der Daten. Da das Feld im ersten Durchlauf horizontal und im zweiten vertikal durchlaufen wird, ergibt sich für einen der beiden Durchläufe das Problem, dass die Daten falsch im Speicher abgelegt sind. Um nun eine aufwändige Umordnung der Koeffizienten zu vermeiden, kann auch ein anderer Ansatz zur Parallelisierung gewählt werden. Hierbei werden die Operationen innerhalb einer Schleife zusammengefasst, um die erwünschte Datenparallelität zu erreichen.

Zur Verdeutlichung des verwendeten Algorithmus, bzw. der möglichen Parallelität, wird hier ein Schleifendurchlauf in der Form von Vektor- und Matrizenoperationen ausgeführt.

[17] Die Aufteilung der Anweisungen ist in diesem Beispiel aus Gründen der Anschaulichkeit nicht optimal.

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_4 \end{pmatrix}, \begin{pmatrix} y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} C_2 & C_6 \\ C_6 & -C_2 \end{pmatrix} \cdot \begin{pmatrix} f_2 \\ f_6 \end{pmatrix},$$
$$\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} + \begin{pmatrix} y_2 \\ y_3 \end{pmatrix}, \begin{pmatrix} a_3 \\ a_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} - \begin{pmatrix} y_3 \\ y_2 \end{pmatrix}$$

Falls der Multimediaprozessor über eine Multiplikationsanweisung mit höherer Datenparallelität verfügt, kann dieser komplette erste Teil auch mit einer einzelnen parallelen Multiplikation durchgeführt werden.

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 1 & C_2 & 1 & C_6 \\ 1 & C_6 & -1 & -C_2 \\ 1 & -C_6 & -1 & C_2 \\ 1 & -C_2 & 1 & -C \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_2 \\ f_4 \\ f_6 \end{pmatrix}$$

Diese breite Multiplikation eignet sich auch für die Koeffizienten mit ungeradem Index[18]. Ist diese Operation nicht vorhanden, so muss diese entsprechend in andere vorhandene kleinere Operationen zerlegt werden.

$$\begin{pmatrix} y_4 \\ y_6 \\ y_7 \\ y_5 \end{pmatrix} = \begin{pmatrix} C_1 & C_3 & C_5 & C_7 \\ C_3 & -C_2 & -C_1 & -C_5 \\ C_5 & -C_1 & C_7 & C_3 \\ C_7 & -C_5 & -C_3 & C_1 \end{pmatrix} \cdot \begin{pmatrix} f_1 \\ f_3 \\ f_5 \\ f_7 \end{pmatrix}$$

Zum Abschluss müssen nun die Teilergebnisse noch zum Endergebnis zusammengefügt werden.

$$\begin{pmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} y_4 \\ y_6 \\ y_7 \\ y_5 \end{pmatrix}, \begin{pmatrix} F_7 \\ F_6 \\ F_5 \\ F_4 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} - \begin{pmatrix} y_4 \\ y_6 \\ y_7 \\ y_5 \end{pmatrix}$$

Hierbei ergibt sich das Problem, dass die Werte vier bis sieben in umgekehrter Reihenfolge im Zielregister liegen. Da diese Reihenfolge für die spätere Bearbeitung nicht erwünscht ist, müssen die Werte im Register getauscht werden. Sind lediglich zwei Werte pro Register vorhanden, beschränkt sich dies auf zwei Rotationsanweisungen, sind mehr Argumente vorhanden, so kann dies entweder über eine Spezialanweisung oder aber über mehrere Teiloperationen[19] geschehen.

Da die Eingangskoeffizienten dieser Operation nicht in aufsteigender Folge benötigt werden, empfiehlt es sich, diesen Teil als zweiten durchzuführen und im ersten die Zwischenergebnisse bereits in

[18] Eine Multiplikation von 4x4 Matrizen ist bei vielen Transformationen im dreidimensionalen Raum üblich. Einige Multimediaprozessoren, die auch für die dreidimensionale Darstellung optimiert sind, verfügen über spezielle Multiplikationsbefehle für diese Matrizen.

[19] Bei einem Intel MMX-Befehlssatz benötigt diese einfache Operation acht Anweisungen, von denen auf einem Pentium aufgrund der Datenabhängigkeiten und der limitierten Ressourcen (hier der Schiebeeinheit) nur vier zu zwei Paaren zusammengefügt werden können, so dass sich eine Ausführungszeit von sechs Takten ergibt. Der ISSE-Befehlssatz bietet eine Permutationsanweisung, so dass diese Operation in nur einem Takt durchgeführt wird.

der benötigten Folge abzulegen. Dies ist insofern kein zusätzlicher Aufwand für den ersten Teil, da in diesem die Koeffizienten in vertikaler Ausrichtung berechnet werden und somit in beliebiger Folge am Ende jedes Schleifendurchlaufes zur Verfügung stehen. Eine andere Möglichkeit besteht darin, die Koeffizienten in der inversen Zig-Zag-Stufe bereits um 90° gedreht abzuspeichern. Dies ermöglicht es, die Koeffizienten beim vertikalen Durchlauf entweder in horizontaler Richtung zu laden oder zu speichern.

5.3.3.4 Bewegungskompensation

Die Bewegungskompensation ist ebenfalls hochgradig datenparallel. Es werden komplette 16x16 Bildelement Blöcke (bzw. 16x8 bei Halbbild-Bewegungskompensation) bearbeitet, in denen die einzelnen Elemente nicht in die Bearbeitung der anderen Elemente eingehen. Weiterhin vorteilhaft ist die geringe Datenbreite von nur 8 Bit pro Bildelement, so dass ein typischer Multimediaprozessor mit maximaler SIMD-Steigerung arbeiten kann[20]. Durch die Datenparallelisierung der Bearbeitung wird allerdings die Beschränkung durch den Hauptspeicher noch deutlicher.

5.4 Messung der potenziellen Parallelität im MPEG 2 Dekodierungsprozess

5.4.1 Vorstellung des Messverfahrens

Zur Messung wurde ein Software MPEG-2-Decoder für INTEL-Prozessoren eingesetzt. Dieser wurde auf zwei verschiedenen INTEL-Prozessoren, einem Pentium MMX™ und einem Pentium II getestet.

Gemessen wurde mit einem Performancetestprogramm „Vtune“ von INTEL, das in der Lage ist, die in den INTEL Prozessoren vorhandenen Leistungszähler zu nutzen, um Aussagen über einzelne Lasten des Prozessors während der Ausführung zu machen. Da dies nur auf einem Pentium II möglich ist, beschränkt sich die Messung des Pentium-Systems auf Ausführungszeiten. Dies stellt aber für die Analyse kein Problem dar, da sich aufgrund der statischen Ausführungsreihenfolge bei diesem Prozessor die Ausführungszeiten den einzelnen Befehlen eindeutig zuordnen lassen.

Es wurden folgende Routinen getrennt erfasst:

1. Parser
2. Berechnung der Bewegungsvektoren
3. Inverse DCT

[20] Durch das Fehlen einer Mittelwertberechnungs-Instruktion im Intel MMX Befehlssatz kann diese Operation nicht auf Byteebene durchgeführt werden. Es müssen beide Operanden auf 16 Bit-Worte erweitert und danach wieder zu acht Bit Werten zusammengefasst werden. Im AMD K6-3Dnow und INTEL ISSE Befehlssatz ist eine Mittelwertinstruktion vorgesehen, so dass sich die Anzahl der verwendeten Instruktionen deutlich senken lässt. Ob sich dies allerdings aufgrund der starken Speicher-

4. Bewegungskompensation
5. Darstellung
6. AC3 Parser
7. Inverse FFT
8. Filterung und Ausgabe

Die inverse Quantisierung wurde in den Parserprozess eingebettet, da es sich während der Erstellung des Decoders gezeigt hat, dass dies aufgrund der geringen Füllung der Blöcke von deutlichem Nutzen ist. Zusätzlich wurde noch die durchschnittliche Anzahl an SIMD Operationen pro Befehl bestimmt. Da diese Instruktionen nur in den sehr regelmäßigen Programmteilen der IDCT, der Bewegungskompensation und der Bilddarstellung benutzt werden, konnte dies durch einfaches Auszählen geschehen.

Als Ausgangsmaterial wurden verschiedene MPEG-2-Clips, mit durchschnittlichen Bitraten zwischen vier und zehn Megabit pro Sekunde verwendet.

5.4.2 Durchführung der Messung und Ergebnisse

Die Messungen der Ausführungszeit wurden auf einen Pentium II normiert, so dass sich im normierten Fall eine Gesamtausführungszeit von 116,2% eines Pentium Prozessors gegenüber einem Pentium II-Prozessor mit gleicher Taktfrequenz ergibt.

	SIMD Operationen pro Befehl	**Pentium MMX™**		**Pentium II MMX™**	
Gesamtausführungszeit	-		116,2%		100,0%
MPEG Parser	-	23,7%	27,5%	26,2%	26,2%
Bewegungsvektoren	-	2,1%	2,4%	2,2%	2,2%
Inverse DCT	5,7	18,0%	20,9%	19,1%	19,1%
Bewegungskompensation	5,0	30,7%	35,7%	32,6%	32,6%
Darstellung	7,9	13,7%	15,9%	12,0%	12,0%
AC3 Parser	-	4,2%	4,9%	2,4%	2,4%
Inverse FFT	-	3,0%	3,5%	2,8%	2,8%
Filterung und Ausgabe	-	2,6%	3,0%	1,6%	1,6%

5.4.3 Analyse des Ergebnisses

Als wichtigstes Ergebnis dieser Messung zeigt sich, dass ein Großteil der Ausführungszeit in parallelisierbaren Programmteilen verbracht wird. Der hohe Grad an SIMD Operationen in den parallelisierbaren Teilen zeigt, dass bereits sehr viel Parallelität durch diese Instruktionen gebunden wird.

---

abhängigkeit der Bewegungskompensation in der Ausführungszeit erkennbar macht, ist stark von der Varianz der Bewegungsvektoren abhängig.

Als mögliche Leistungssteigerung durch die Verwendung von mehreren Kontrollfäden ergibt sich somit ein Faktor von drei, da der nicht parallelisierbare Anteil etwa 30% der gesamten Videodekodierung entspricht.

6 Struktur des simulierten Prozessors

6.1 Einleitung

Der simulierte Prozessor entspricht in seiner grundlegenden Struktur einem aktuellen superskalaren Prozessor. Er kann durch verschiedene Parameter derart konfiguriert werden, dass er an zahlreiche Prozessorarchitekturen bestehender Prozessoren angeglichen wird. Durch diese Konfigurierbarkeit ist es möglich den Prozessor unter den Bedingungen, die diese verschiedenen Architekturfamilien auszeichnen, zu testen.

6.2 Grundstruktur des Prozessors

6.2.1 Einfädiger Basisprozessor

Die einfädige Variante des simulierten Prozessors basiert auf einem PowerPC 604 Design:

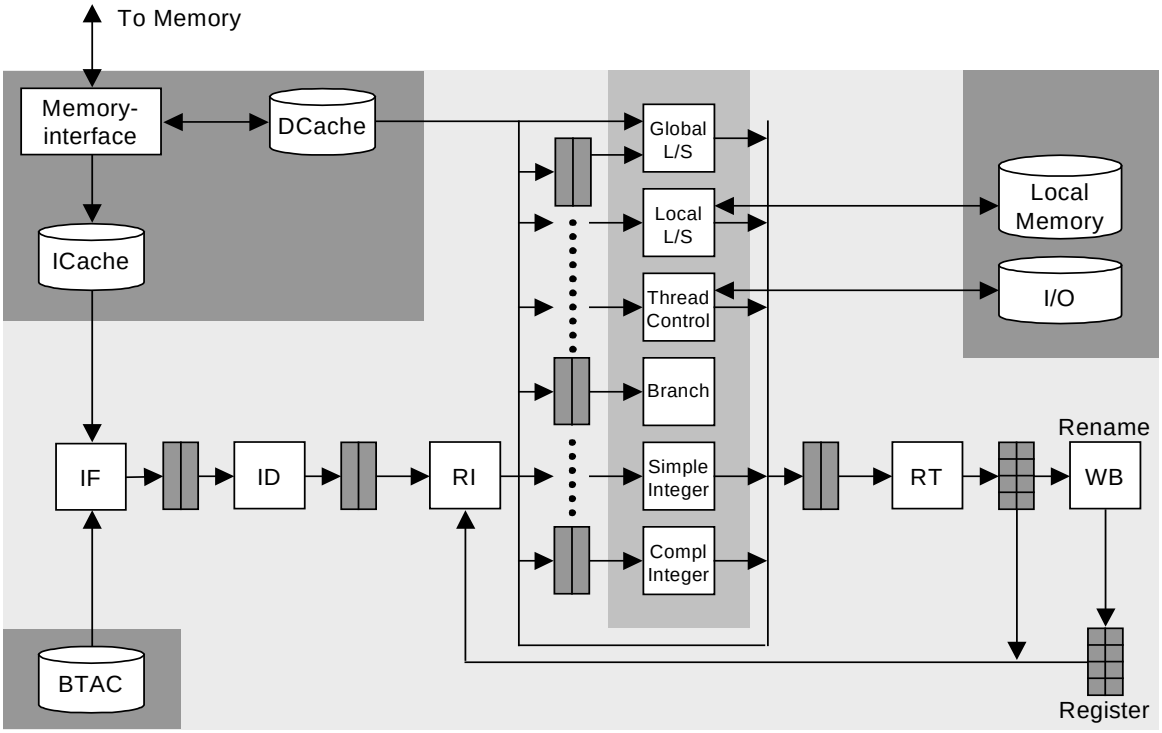


Abbildung 6-1: Aufbau des einfädigen Basisprozessors

Der Basisprozessor besitzt eine Befehlsladeeinheit (IF – „Instruction Fetch“) und eine Befehlsdekodiereinheit (ID – „Instruction Decode“). Diese laden und dekodieren die Instruktionen in Programmreihenfolge und schreiben sie in den Befehlsbereitstellungspuffer. Aus diesem Bereitstellungspuffer wählt die Registerumbenennungs- und Befehlszuordnungsstufe (RI – „Rename and Issue“) in jedem Takt ein oder mehrere Befehle aus, belegt für die Zielregister dieser Befehle freie Umbenennungsregister („Renameregister“) zur Vermeidung unechter Datenabhängigkeiten und verteilt sie auf die Be-

fehlsumordnungspuffer der Ausführungseinheiten. Diese Befehle werden auch gleichzeitig in die Befehlsrückordnungspuffer eingefügt. Es stehen fünf verschiedene Ausführungseinheiten zur Verfügung, von denen einige auch mehrfach vorhanden sein können:

- Lade- und Speichereinheit für den externen Speicher (Global L/S)
- Lade- und Speichereinheit für den internen Speicher (Local L/S)
- Steuer-, Synchronisations- und Ein-/Ausgabeeinheit (Thread Control)
- Sprungeinheit (Branch)
- Einfache Integer-/Multimediaeinheit (Simple Integer)
- Komplexe Integer-/Multimediaeinheit (Complex Integer)

Die einfache Integereinheit und die lokale Lade-/Speichereinheit können auch mehrfach vorhanden sein. Befehle, die in den Ausführungseinheiten komplett ausgeführt worden sind, werden von der Befehlsrückordnungsstufe in der originalen Reihenfolge bestätigt und aus dem Befehlsrückordnungspuffer genommen. Die Rückordnungseinheit (RT – „Retire“) kann in jedem Takt, falls vorhanden, ein oder mehrere Befehle bestätigen. In der Rückschreibstufe (WB – „Writeback“) werden die Ergebnisse aus den Umbenennungsregistern in die Architekturregister übernommen.

Der Prozessor besitzt darüber hinaus getrennte Caches für Daten und Befehle, einen Sprungzieladress-Cache, einen Speicher- und einen Ein-/Ausgabebus sowie internen Speicher [87].

6.2.2 Mehrfädiger Prozessor

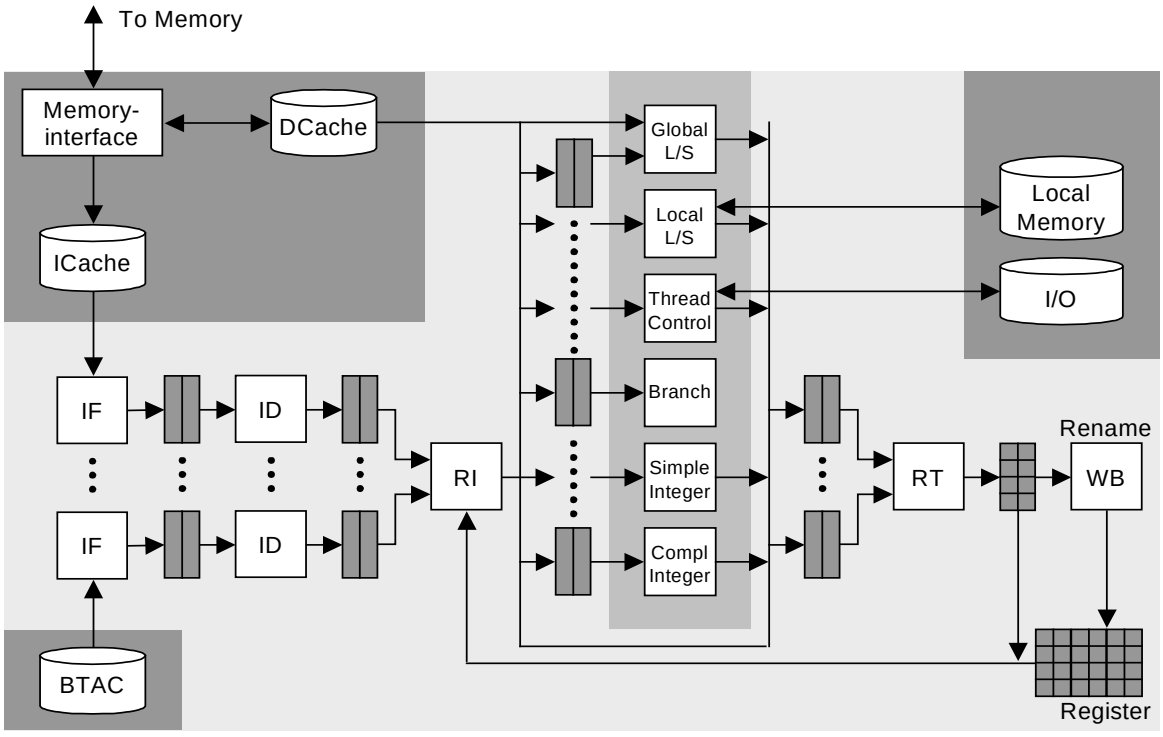


Abbildung 6-2: Aufbau des mehrfädigen Prozessors

Der mehrfädige Prozessor ist um drei Einheitengruppen erweitert:

- Jeder Ausführungskontext besitzt einen eigenen, unabhängigen Satz der Architekturregister.
- Es stehen jeweils mehr als eine Befehlslade- und Befehlsdekodiereinheit zur Verfügung.
- Jeder Ausführungskontext besitzt eigene Befehlsdekodier-, Befehlsbereitstellungs- und Befehlsrückordnungspuffer.

Die Befehlszuordnungsstufe kann in jedem Takt aus den Befehlsbereitstellungspuffern der verschiedenen Ausführungskontexte Befehle auswählen. Auch die Befehlsrückordnungsstufe kann aus den Befehlsrückordnungspuffern mehrerer Ausführungskontexte wählen.

6.2.3 Ausführungs-Kontrollpipeline

Die Ausführungs-Kontrollpipeline kontrolliert die Ausführung der Instruktionen. Nacheinander durchläuft jeder Befehl folgende Stufen, die jeweils mindestens einen Takt zur Ausführung benötigen:

- Befehlsladestufe („Fetch“)
- Befehlsdekodierstufe („Decode“)
- Registerumbenennungs- und Befehlszuordnungsstufe („Rename/Issue“)
- Befehlsausführungsstufe („Execute“)
- Befehlsrückordnungsstufe („Retire“)
- Rückschreibestufe („Writeback“)

Die Einheiten der Ausführungs-Kontrollpipeline, die in der Programmordnung arbeiten, sind superskalar ausgelegt. Sie können mehrere Instruktionen pro Takt abarbeiten. Die anderen Ausführungseinheiten bearbeiten nur maximal einen Befehl pro Takt. Die Befehlszuordnungsstufe kann je nach gewählter Prozessorarchitektur zwischen diesen beiden Extremen schwanken. Um eine bessere Auslastung des Prozessors zu erreichen, können die Instruktionsladestufe und die Instruktionsdekodierstufe auch mehrfach vorhanden sein.

6.2.4 Ausführungseinheiten

In den Ausführungseinheiten werden die jeweiligen Operationen der Instruktionen durchgeführt. Diese Ausführung findet nicht bei allen Instruktionsarten in Programmordnung statt. Je nach Art der Instruktionen werden diese in verschiedenen Ausführungseinheiten ausgeführt. Folgende Einheiten sind vorgesehen:

- Einfache Integereinheiten
- Komplexe Integereinheiten
- Lade-/Speichereinheit
- Lokale Lade-/Speichereinheit
- Sprungausführungseinheit

- Kommunikations- und Kontrolleinheit
- Fließkommaeinheit

Die Verweildauer der Instruktionen in den verschiedenen Ausführungseinheiten ist, je nach Art der Einheit, verschieden. Sie kann von einem Takt in den einfachen Integereinheiten bis zu beliebig vielen Takten in der Kommunikations- und Kontrolleinheit[21] schwanken.

6.2.5 Datenregister

Die Datenregister des Prozessors unterteilen sich in zwei Gruppen: die Hauptregisterbänke, in denen jedem logischen Register jedes Ausführungskontextes ein physikalisches Register fest zugeordnet ist und den Umbenennungspufferregistern, deren Zuordnung während der Programmausführung variabel ist. Diese Umbenennungspufferregister werden genutzt, um sowohl falsche Datenabhängigkeiten zu überdecken, als auch um echte Abhängigkeiten aufzudecken.

Einige der Register sind nicht für allgemeine Daten verfügbar, sondern mit festen Werten verbunden. Das Register mit dem Index Null, enthält immer den Wert Null. Ein Schreibzugriff auf dieses Register hat keine Auswirkung. Register Nummer Zwei enthält immer die Adresse der benutzenden Instruktion. Dieses Register kann dazu verwendet werden, um Adressen relativ zum Instruktionszeiger anzugeben. Ein Schreibzugriff auf dieses Register ist nicht gestattet. Eine Änderung muss durch eine Sprunganweisung herbeigeführt werden.

6.2.6 Daten- und Befehls-Caches

Der Prozessor verwendet getrennte Daten- und Instruktions-Caches. Es ist nur eine Ebene von Caches vorgesehen. Dies ist eine innerhalb der für diesen Prozessor gewählten Anwendung realistische Annahme. Der Grund für diese beschnittenen Speicherhierarchie ist in den typischen Speicheranforderungen von Multimediaanwendungen zu sehen. Es werden typischerweise entweder sehr kleine Bereiche (wie z.B. 64 Worte) oder aber sehr große Bereiche (> 500KByte) benutzt.

Beide Caches sind als nicht blockierend entworfen. Dies ist bei Datencaches ein übliches Verfahren, um die Parallelität der Ausführung zu erhöhen. Bei einem nicht mehrfädigen Prozessor ist ein nicht blockierender Instruktionscache unnötig, da ja nur eine Instruktionssequenz verfolgt wird. Bei einem mehrfädigen Prozessor ist diese Fähigkeit hingegen essentiell, da hierdurch Latenzen, die durch nicht im Cache vorhandene Instruktionen eines Ausführungskontextes entstehen, durch das Laden der Instruktionen eines anderen Kontextes überbrückt werden können.

Da mehrere Befehlsladeeinheiten vorgesehen sind, muss der Instruktions-Cache auch mehrere Anforderungen pro Takt bearbeiten können. Dies ist im Gegensatz zu einem Daten-Cache ein geringes

[21] Diese lange Aufenthaltszeit entsteht durch Synchronisationsinstruktionen, die auf eine Ereignis warten, das erst zu einem nicht vorhersagbaren Zeitpunkt erfüllt wird. Bei einer entsprechenden Implementierung eines Mehrkontextbetriebssystems, sollten diese Zeiten durch einen Software-Kontextwechsel überbrückt werden, um die Ressourcen des Prozessors nicht unnötig mit einem wartenden Ausführungskontext zu belasten.

Problem, da im Instruktions-Cache nur gelesen und nicht geschrieben wird. Somit können auch keine Zugriffskonflikte entstehen.

6.2.7 Speicherschnittstelle

Die Aufgabe der Speicherschnittstelle ist es, komplette Cache-Zeilen zwischen den Caches des Prozessors und dem externen Speicher zu transferieren. Die Übertragung findet, wie bei aktuellen Speichersystemen üblich, in Gruppen zu mehreren Prozessordatenworten in Speicherzugriffsfolgen („burst“) statt.

Da die Speicherschnittstelle von Instruktions- und Daten-Cache gleichermaßen benötigt wird, wird diese zwischen diesen beiden Einheiten in der Reihenfolge des Eintreffens der Anforderung bedient.

Neben dem externen Speicher für Daten und Instruktionen verfügt der Prozessor auch noch über einen internen Speicher mit fester Zugriffslatenz. Dieser wird nicht über den Daten-Cache angesprochen, sondern direkt durch eine eigene Lade-/Speichereinheit. Der Vorteil der logischen und physikalischen Trennung dieser beiden Speicher ist in der Parallelität der Zugriffe zu sehen. Es müssen weder Konflikte durch Datenabhängigkeit beachtet werden, noch können sich Anforderungen aufgrund gemeinsam benutzter Ressourcen behindern. Diese Aufteilung des Speichers in unabhängige Gruppen ist eine bei DSPs häufig zu findende Architekturentscheidung.

6.3 Aufbau der Ausführungs-Kontrollpipeline

6.3.1 Überblick

Die Ausführungs- und Kontroll-Pipeline lässt sich in zwei verschiedene Bereiche einteilen einem, der die Befehle in der ursprünglichen Programmordnung ausführt („In-order-Execution“) und einem Teil, in dem die Ausführungsreihenfolge der Befehle aufgrund ihrer Datenabhängigkeiten bestimmt wird („Out-of-order-Execution“).

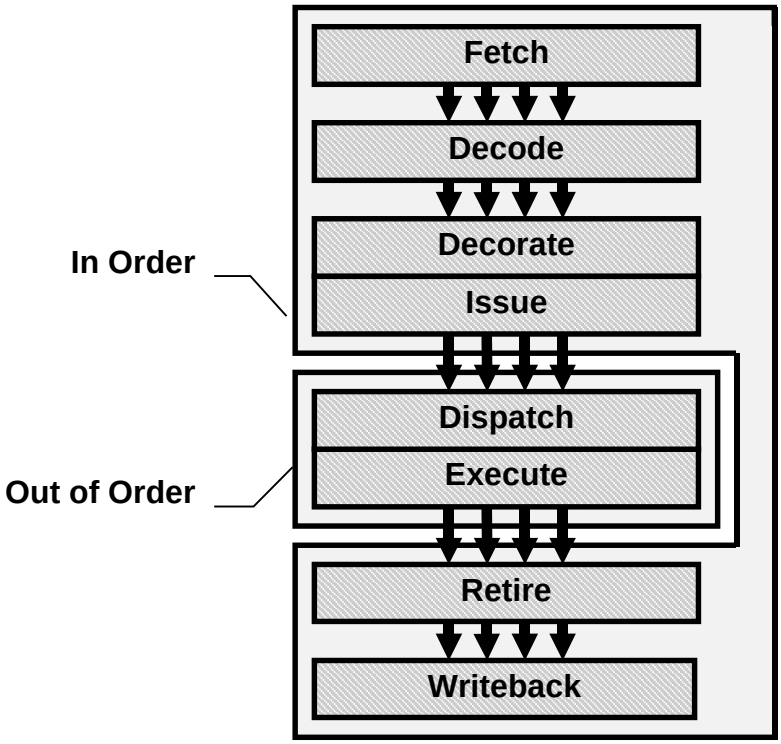


Abbildung 6-3: Aufbau der Ausführungs-Kontrollpipeline

6.3.2 Befehlsladestufe („Fetch“)

In der Befehlsladestufe werden die Befehle aus dem Befehls-Cache entnommen und, in Pakete zerlegt, an die nächste Stufe weitergereicht. Die Arbeitsweise dieser Stufe zerfällt in zwei generelle Teile dem Zugriff auf den Befehls-Cache und dem Weiterschalten des Befehlszeigers („Program-Counter“).

Kann die Anforderung der Befehle durch den Befehls-Cache erfüllt werden, werden diese aus dem Cache übernommen und an die Befehlsdekodierstufe weitergereicht, die diese dann im nächsten Takt bearbeiten soll. Kann die Anforderung hingegen nicht erfüllt werden, so wird eine Cache-Füllanforderung an den Systembus weitergereicht und der betroffene Ausführungskontext bis zu deren Erfüllung oder der Ungültigmachung des Befehlszeigers durch eine Sprungvorhersagekorrektur von weiteren Befehlsladeoperationen zurückgestellt.

Die Berechnung der nächsten Adresse des Befehlszeigers erfolgt aufgrund folgender Parameter (mit fallender Priorität):

- Sprungvorhersagekorrektur oder verspätete Sprungausführung durch die Sprungausführungseinheit
- Sprungvorhersage der Befehlsdekodiereinheit
- Sprungvorhersage des Sprungadressziel-Caches (BTAC) [30]
- Erhöhung um die Anzahl der übernommenen Befehle

Liegen am Ende eines Taktes Sprungzieladressen von Dekodier- oder Sprungausführungseinheit vor, die der Adresse der Befehle, die in diesem Takt geladen wurden, widersprechen, so sind diese Befehle ungültig, und der Puffer der Befehlsladestufe wird gelöscht. Dies hat natürlich zur Folge, dass

im nächsten Takt keine Befehle für die Befehlsdekodiereinheit bereitstehen und diese einen Takt leer läuft. Um dies zu vermeiden, wird mit Hilfe des Sprungzieladress-Caches versucht, anhand der Adresse der geladenen Instruktionen die mögliche Zieladresse eines Sprunges in diesen Instruktionen vorherzusagen. Dies geschieht, in dem unter der Adresse von als genommen vorhergesagten Sprungbefehlen, die Sprungzieladresse abgelegt wird. Wird nun einer dieser Sprünge ein zweites Mal geladen, kann bereits in der Instruktionsladestufe eine Sprungvorhersage getroffen werden.

Ein Instruktionsblock, der in einem Takt bearbeitet werden soll, darf eine Cache-Zeilengrenze nicht überschreiten[22]. Dies hat zur Folge, dass häufig am Ende einer Cache-Zeile nicht die Maximalzahl an Instruktionen erreicht wird. Je näher die Länge des maximalen Instruktionsblockes an der Größe einer Cache-Zeile liegt, um so höher wird die Wahrscheinlichkeit, dass der Block nicht komplett ausgenutzt werden kann [84]. Verstärkt wird dieser Effekt noch durch Sprünge, die in der Dekodierstufe zu einem vorzeitigen Abbruch der Bearbeitung eines Blockes führen. Um dennoch ohne die Verlängerung eines Instruktionsblockes die Anzahl der durchschnittlich geholten Instruktionen pro Takt zu erhöhen, kann auch mehr als eine Instruktionsladeeinheit verwendet werden. Gleiches gilt auch für die Instruktionsdekodierstufe. Dies hat zwar keinen positiven Effekt bei nur einem Ausführungskontext, verbessert aber deutlich den Instruktionsdurchsatz bei mehreren.

6.3.3 Befehlsdekodierpuffer

In diesen Puffern werden die Befehlspakete der Befehlsladeeinheit abgelegt, bis sie von der Befehlsdekodierstufe bearbeitet werden. Jeder Ausführungskontext besitzt einen eigenen Befehlsdekodierpuffer. Da nicht immer so viele Befehlsdekodiereinheiten wie Ausführungskontexte vorhanden sind, kann es sein, dass ein Befehlspaket länger als ein Takt auf eine Dekodiereinheit wartet. Ein anderer Grund dafür, dass die Befehle eines Befehlspaketes länger als einen Takt in diesem Puffer verweilen, ist dann gegeben, wenn der Befehlsbereitstellungspuffer des zugehörigen Kontrollfadens nicht genug freie Einträge besitzt, um alle Befehle aus dem Dekodierpuffer zu übernehmen.

6.3.4 Befehlsdekodierstufe („Decode“)

Die Aufgabe der Befehlsdekodierstufe besteht in der Umwandlung der Maschinen-Befehlscodes in interne Befehlscodes, sowie der Sprungvorhersage.

Aufgrund der sehr regulären Struktur des Befehlssatzes dürfte sich eine Dekodierung ohne Probleme in einem Befehlstakt vollenden lassen, so dass die Verwendung einer nur einstufigen Dekodiereinheit als angemessen betrachtet wird.

Für die Sprungvorhersage wird bei einer statische Spekulation nach dem klassischen Modell des genommenen Rücksprunges vorgegangen. Das heißt, jeder bedingte Sprung mit einem negativen Versatz zum Programmzähler wird als genommen betrachtet, **alle** anderen bedingten Sprünge werden dagegen als nicht genommen betrachtet. Ein unbedingter Sprung zu einer absoluten oder zum

[22] Dies ist eine bei aktuellen Prozessoren übliche Einschränkung und entspricht somit einer auch real vorhandenen Limitierung. Umgangen wird dieses Problem häufig, indem der Programmcode auf Cache-Zeilengrenzen ausgerichtet wird. Hierbei wirkt sich allerdings nachteilig aus, dass die Lücken mit Leerinstruktionen gefüllt werden müssen.

Programmzähler relativen Adresse wird ebenfalls als genommen betrachtet. Unbedingte Sprünge, deren Adresse zum Dekodierzeitpunkt nicht bekannt sind, führen zum Blockieren der Instruktionslade- und -dekodierstufen für alle weiteren Befehle dieses Kontrollfadens bis zur Ausführung dieses Sprungbefehles durch die Sprungausführungseinheit.

Für eine dynamische Sprungvorhersage stehen zwei verschiedene Verfahren zur Verfügung: eine einfache Zwei-Bit-Sprungvorhersage oder eine zweistufige Vorhersage nach McFarlings „gshare“ Verfahren [48][89][88].

6.3.5 Registerumbenennungs- und Befehlszuordnungsstufe („Rename/Issue“)

In dieser, hier auch Dekorationsstufe genannten Stufe, werden die Instruktionen mit den Umbenennungspufferregistern verbunden. Dies wird wie auch in den meisten modernen Prozessoren in einem vorzubereitenden Takt ausgeführt, bevor die Instruktionen den Ausführungseinheiten zugeführt werden.

Die Umbenennungspufferregister dienen dazu, die echten Datenabhängigkeiten des Programms aufzudecken und die unechten zu überdecken. Diese Stufe arbeitet mit den Instruktionen aller Ausführungskontexte und dekoriert mehrere Befehle pro Takt. Die so mit Umbenennungspufferregistern versehenen Instruktionen werden dann im selben Takt in die Befehlsumordnungspuffer, und somit der Ausführungseinheitenzuordnungsstufe, übergeben. Gleichzeitig werden die Befehle auch in einen Befehlsrückordnungspuffer übertragen, in dem sie gehalten werden, bis sie und alle Befehle vor ihnen komplett abgearbeitet sind.

6.3.6 Befehlsumordnungspuffer („Reservation stations“)

In dieser Stufe warten die Befehle auf die Verfügbarkeit ihrer Quelloperanden und einer Ausführungseinheit. Sind alle Bedingungen für die Ausführung eines Befehls gegeben, wird er je nach Verfügbarkeit den einzelnen Ausführungseinheiten zugeordnet und diesen zur Ausführung übergeben. Diese Stufe arbeitet nicht in der Reihenfolge des Programms, sondern basierend auf den echten Datenabhängigkeiten. Sie ist in der Lage Instruktionen aus einem Fenster („Dispatch-queue“) auszuwählen und zuzuordnen. Je nach Prozessortyp können auch mehrere dieser Fenster existieren und einzelnen Anweisungstypen oder sogar Ausführungseinheiten fest zugeordnet sein. In diesem Fall spricht man auch von „Reservation-station“.

Nicht alle Anweisungen lassen sich außerhalb der Programmreihenfolge ausführen, da nicht alle Datenabhängigkeiten einfach zu erkennen sind. Dies betrifft besonders Systemkontrollanweisungen und Anweisungen, die auf einen der beiden Speicher zugreifen. Folgende Regeln werden für diese Anweisungen verwendet:

- Anweisungen verschiedener Ausführungskontexte können sich beliebig überholen.
- Anweisungen, die auf den Speicher zugreifen, können Anweisungen, die den Speicher beschreiben, nicht überholen.

- Systemkontrollanweisungen eines Ausführungskontextes können sich nicht überholen.

Entsprechend sind drei Arten von Befehlsumordnungspuffern vorgesehen:

- **Allgemeine Zuordnungseinheit:** Jede Instruktion kann jede andere Instruktion frei überholen
- **Speicher-Zuordnungseinheit:** Eine Instruktion kann eine andere Instruktion desselben Ausführungskontextes, die kein Zielregister besitzt, nicht überholen.
- **Kontroll-Zuordnungseinheit:** Keine Instruktion kann eine andere Instruktion desselben Ausführungskontextes überholen.

Die Befehlsfenster sind als Lauscher an den Resultatsbussen beteiligt, so dass Ergebnisse der Ausführungseinheiten bereits im nächsten Takt als Quelloperanden zur Verfügung stehen.

6.3.7 Befehlsausführungsstufe („Execute“)

Diese Stufe ist nicht eigentlich in der Kontroll-Pipeline vorhanden, sondern ist komplett in den Ausführungseinheiten verborgen. Sie ist dennoch hier aufgeführt, da die Instruktionen während ihrer Ausführung in der Befehlsrückordnungswarteschlange verbleiben.

6.3.8 Befehlsrückordnungspuffer

In diesem Puffer verweilen die Instruktionen, nachdem sie von der Befehlszuordnungseinheit an die Befehlsumordnungspuffer übergeben werden, bis sie von der Befehlsrückordnungseinheit bestätigt und für beendet erklärt werden.

6.3.9 Befehlsrückordnungsstufe („Retire“)

In dieser Stufe werden die Instruktionen wieder in Programmreihenfolge bearbeitet. Ist ein Befehl und alle seine Vorgänger komplett erfolgreich ausgeführt worden, so wird er in dieser Stufe bestätigt und aus dem Rückordnungspuffer entfernt. Sein Ergebnis wird als korrekt klassifiziert, so dass es in der nächsten Stufe von den Umbenennungspufferregistern zurück in die Architekturregister geschrieben werden kann. Des Weiteren werden alle Befehle, die den Speicher beschreiben, an die Speicherschreibstufe übergeben. Dies kann erst in dieser Stufe geschehen, da aufgrund des gewählten Konsistenzmodells keine spekulativen Schreibzugriffe auf den Speicher erlaubt sind.

6.3.10 Ergebnisschreibstufe („Writeback“)

In dieser Stufe werden die Ergebnisse der Befehle sowohl in die Hauptregisterbänke als auch in den Speicher (bzw. Caches) geschrieben. Da die Ausführung dieser Stufe keine direkte Wirkung auf die Programmausführung hat, kann sie nur bedingt als Pipeline-Stufe gesehen werden. Da die Ressourcen der betroffenen Befehle (zum Beispiel Umbenennungsregister) jedoch erst nach vollendetem Schreiben des Ergebnisses frei werden, hat sie eine Auswirkung auf die Ausführungszeit.

6.4 Aufbau der Ausführungseinheiten

6.4.1 Einfache Integereinheiten

Die einfachen Integereinheiten führen alle logischen sowie die additionsbasierten Operationen aus. Sie verfügen über die Möglichkeit der Teilwortarithmetik und saturierten Arithmetik. Alle Operationen der einfachen Integereinheiten werden in einem Takt ausgeführt. Die Ergebnisse aus den einfachen Integereinheiten können bereits im nächsten Takt weiterverarbeitet werden.

6.4.2 Komplexe Integereinheiten

In den komplexen Integereinheiten werden die verschiedenen Formen der Multiplikation des Multimediaprozessors ausgeführt. Es handelt sich hier um eine Ausführungseinheit mit einer mehrstufigen Pipeline. Aus Gründen der Vereinfachung wird für jede Art der Multiplikation die gleiche Anzahl Takte angenommen. Diese Annahme findet sich in den meisten Multimediaprozessoren insofern bestätigt, als dass die Multimediamultiplikationen meist auch über gleich lange Ausführungszeiten verfügen[23].

Die komplexen Integereinheiten können in jedem Takt eine Instruktion beginnen und eine andere Instruktion vollenden, sie sind also vollständig als Pipeline implementiert. Die Latenz der komplexen Integereinheit lässt sich für die Simulation konfigurieren.

6.4.3 Lade-/Speichereinheit für externen Speicher

Die Lade-/Speichereinheit für den externen Speicher, auch als globale Lade-/Speichereinheit bezeichnet, benötigt zur Ausführung eines Befehls mindestens vier Takte, kann aber im Idealfall in jedem Takt einen neuen Befehl annehmen. Die Befehle werden derart ausgeführt, als ob sie der Reihenfolge ausgeführt würden, in der sie in die Lade-/Speichereinheit gelangen. Das heißt, dass ein Ladebefehl auch vor einem Speicherbefehl mit der gleichen Operandenadresse ausgeführt werden kann, wenn dieser auf eine Cache-Anforderung wartet. Der Ladebefehl wird dann direkt aus dem Operanden des Speicherbefehls versorgt.

Ein Befehl, der nicht erfolgreich aus dem Cache bedient werden kann, führt zu einer Cache-Zeilenanforderung an den Systembus. Bis dieser erfolgreich durchgeführt ist, wartet der Befehl in der Cache-Fehlzugriffswarteschlange. Nach erfolgreichem Einladen der Cache-Zeile wird der wartende Befehl erneut ausgeführt. Dies kann dazu führen, dass bei häufigen Cache-Fehlzugriffen die Lade-/Speichereinheit deutlich weniger als einen Befehl pro Takt ausführen kann.

Ein Ladebefehl, dessen Ziel im Cache vorhanden ist, wird mit einer Latenz von vier Takten geladen. Ist das Ziel nicht im Cache entsteht eine vom Speichersystem und den Füllstand der Cache-Fehlzugriffswarteschlange abhängige Latenz.

[23] Allerdings unterscheidet sich häufig die normale Multiplikation von den Multimediamultiplikationen. Da die Anzahl der nicht Multimediamultiplikationen in der verwendeten Last gering ist, soll auf diesen Unterschied im Simulator nicht eingegangen werden.

6.4.4 Lade-/Speichereinheit für internen Speicher

Die Lade-/Speichereinheit für den internen Speicher, auch lokale Lade-/Speichereinheit genannt, arbeitet mit einer konstanten Ausführungszeit. Da es im Gegensatz zur globalen Lade-/Speichereinheit keine Cache-Fehlzugriffe geben kann, kann sie deutlich einfacher gebaut sein. Im Simulator wird eine dreistufige Pipeline verwendet. In jedem Takt kann eine Lade- oder Speicheranweisung begonnen, und/oder beendet werden. Da es zu keinen Verzögerungen durch den Cachezugriff kommen kann, ist auch ein Überholen von Anweisungen innerhalb der Einheit nicht vorgesehen. Der Aufbau der lokalen Lade und Speichereinheit entspricht im Simulator einer dreistufigen Pipeline.

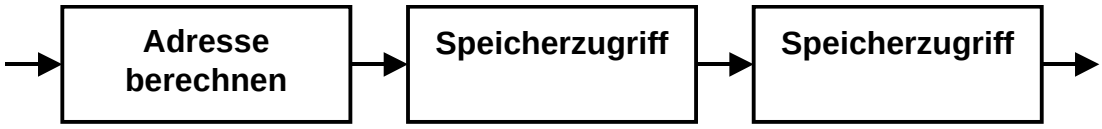


Abbildung 6-4: Pipeline der Lade-/Speichereinheit für internen Speicher

Die Aufteilung des Speicherzugriffes auf zwei Takte vereinfacht den Aufbau des lokalen Speichers, da er sich somit einfach in einer Matrixstruktur aufbauen lässt.

Aufgrund der einfachen Struktur der lokalen Lade-/Speichereinheit ist eine Vervielfältigung dieser Einheit leicht vorstellbar. Eine mögliche Aufteilung würde die Adressberechnung verdoppeln, und danach eine Verteilung der Befehle auf die zwei Speicherbänke vorsehen.

Abbildung 6-5: Pipeline der Lade-/Speichereinheit für internen Speicher, bei der Verwendung von zwei Einheiten

Bei einer Adresskollision würde diese Aufteilung zu einer Sequenzialisierung führen. Des weiteren ist darauf zu achten, dass bei einer Kollision keine Verletzung der Datenabhängigkeit entsteht. Dies kann dadurch erreicht werden, dass die beiden lokalen Lade-/Speichereinheiten entweder nur von verschiedenen Kontrollfäden versorgt oder aber die Anweisungen geordnet den beiden Ausführungseinheiten übergeben werden. Eine entsprechende Struktur findet sich im Pentium-Prozessor, der zwei Lade-/Speicherbefehle pro Takt ausführen kann, falls diese auf unterschiedliche Speicherbänke zugreifen.

6.4.5 Sprungausführungseinheit

Die Sprungausführungseinheit arbeitet zusammen mit der Befehlsdekodiereinheit die bedingten und unbedingten Sprünge ab. Unbedingte Sprünge mit absolutem oder Instruktionsadressen relativem Ziel werden bereits in der Dekodierstufe abgearbeitet und landen nicht in der Sprungausführungseinheit.

Die Sprungausführungseinheit kann pro Takt einen Sprung ausführen. Fünf verschiedene Arten von Sprüngen gelangen in die Sprungausführungseinheit:

- Nicht spekulative Sprünge mit berechnetem Ziel
- Als genommen spekulierte Sprünge, die genommen werden
- Als genommen spekulierte Sprünge, die nicht genommen werden
- Als nicht genommen spekulierte Sprünge, die genommen werden
- Als nicht genommen spekulierte Sprünge, die nicht genommen werden

Bei Sprüngen mit berechnetem Ziel ermittelt die Sprungausführungseinheit die Zieladresse und startet die Befehlsdekodier- und Befehlsladeeinheit mit diese Adresse wieder neu.

Bei korrekt spekulierten Sprüngen wird die Spekulationsmarke von den betroffenen Anweisungen und Registern genommen. Bei nicht korrekt spekulierten Sprüngen werden die Anweisungen und Register, die mit der dazugehörigen Spekulationsmarke versehen sind, aus dem Prozessor entfernt und die Dekodier und Befehlsladestufe mit der korrekten Adresse wieder gestartet.

Die Sprungausführungseinheit liefert bei spekulativen Sprüngen das Ergebnis der Spekulation an eine eventuell vorhandene dynamische Sprungvorhersage, sowie das Ziel der spekulativen Sprünge an den Sprungadresszielcache. Bei falsch spekulierten Sprüngen entstehen bis zu fünf Leerlauftakte.

6.4.6 Steuer-, Synchronisations- und Ein-/Ausgabeeinheit

Diese Einheit, im weiteren auch als Kontrollfadensteuereinheit bezeichnet, erfüllt eine Reihe von Aufgaben.

- Steuerung interner Prozesse, wie zum Beispiel das Ungültigmachen oder Herausschreiben der Caches, das Setzen von Ausführungskontextprioritäten, das Erzeugen und Vernichten von Ausführungskontexten.
- Synchronisation zwischen Ausführungskontexten durch primitive Warteoperationen auf einzelne oder mehrere Bits eines Kontrollwortes, sowie das Blockieren und Freigeben von Kontextwechseln.
- Ein- und Ausgabe von Daten über den externen Bus, zum Beispiel zu einer Bilddarstellungseinheit.

Jeder Ausführungskontext kann sich in einem von sechs verschiedenen Zuständen befinden (siehe auch 6.6.4), die durch die Kontrollfadensteuereinheit gewechselt werden können.

Einzelne Anweisungen können aufgrund ihrer Natur als Warteanweisungen beliebig lange in dieser Ausführungseinheit verbleiben.

6.4.7 Fließkommaeinheit

Auf eine Fließkommaeinheit wurde für den Simulator verzichtet. Da die verwendete Last komplett mit Festkommaarithmetik arbeitet, stellt dies keine Verfälschung des Ergebnisses dar. Eine Implementierung einer Fließkommaeinheit wäre entsprechend der komplexen Integereinheiten durchzuführen. Es könnte hierbei entweder eine Aufteilung in mehrere verschiedene Einheiten oder aber eine gemeinsame Einheit implementiert werden. Zur Vereinfachung des Prozessors wäre es günstig, den Registersatz der Integereinheiten gemeinsam mit der Fließkommaeinheit zu nutzen. Dies ermöglicht zwar lediglich einfach genaue Fließkommazahlen, doch stellt dies den Normalfall im Multimediabereich dar.

Eine sinnvollere Variante wäre die Erweiterung des Datenwortes des Prozessors auf 64 Bit. Dies würde doppelt genaue Fließkommanzahlen und höher parallele SIMD-Befehle ermöglichen. Dieses Konzept des einheitlichen Datenregisters wird zum Beispiel im SUN MAJC Prozessor verwendet [25].

6.5 Simulationsparameter

Der Prozessorsimulator ist hochgradig konfigurierbar. Es lassen sich sowohl die externen Parameter wie zum Beispiel die Busbreite oder Speicherzugriffszeit manipulieren, als auch die internen Parameter wie Zuordnungsbandbreite oder Anzahl der Ausführungseinheiten.

6.5.1 Parameter der Ausführungskontrollpipeline

Die Befehlslade- und die Befehlsdekodiereinheiten lassen sich jeweils in ihrer Anzahl, und der Anzahl der Instruktionen die sie pro Takt bearbeiten können konfigurieren. Die Befehlszuordnungs- und die Befehlsrückordnungseinheit lassen sich jeweils in der Anzahl der Befehle die pro Takt insgesamt maximal bearbeitet werden können, als auch in der Anzahl der Befehle die pro Takt von einem einzelnen Kontrollfaden maximal bearbeitet werden können konfigurieren. Die Befehlsumordnungspuffer lassen sich in ihrer Zuordnung zu den Ausführungseinheiten, ihrer Tiefe, Sequenzialisierungsanforderung als auch in der Anzahl der Befehle die sie pro Takt annehmen und abgeben können konfigurieren.

Weiterhin können für alle diese Einheiten aus mehreren Befehlsauswahlstrategien gewählt werden.

6.5.2 Parameter der Register

Sowohl die Architektur- als auch die Umbenennungsregister lassen sich in ihrer Anzahl spezifizieren. Des weiteren kann die Anzahl der Ergebnisbusse und der Rückschreibebusse von den Umbenennungsregistern zu den Architekturregistern konfiguriert werden.

6.5.3 Parameter der Sprungvorhersage

Die Sprungvorhersage kann statisch oder dynamisch erfolgen. Des weiteren kann die Größe des Sprungzieladress-Caches und des Sprungvergangenheits-Caches konfiguriert werden. Weiterhin

kann auch die Anzahl der Sprünge, die gleichzeitig spekulativ ausgeführt werden, kann beschränkt werden.

6.5.4 Parameter der Ausführungseinheiten

Die Integer- und die lokale Lade-/Speichereinheit kann mehrfach vorhanden sein. Die Ausführungseinheiten lassen sich mit verschiedenen Befehlsumordnungspuffern und Ergebnisbussen verknüpft werden. Die komplexe Integereinheit lässt sich in ihrer Pipelinelänge variieren.

6.5.5 Parameter des Speichersystems

Der externe Systembus lässt sich in Busbreite, Zugriffszeit und Zugriffsfolge konfigurieren. Die Daten- und der Instruktions-Caches können jeweils in Größe und Assoziativität eingestellt werden. Für den Daten-Cache stehen als weitere Parameter noch spekulatives Vorabladen und verschiedene Zeilenersetzungsstrategien zur Verfügung.

6.6 Programmiermodell

6.6.1 Registerstruktur

Jeder Ausführungskontext besitzt einen von den anderen Kontrollfäden unabhängigen Registersatz. Es können pro Ausführungskontext bis zu 32 Register adressiert werden. Hierbei sind die Register 3 bis 31 zur allgemeinen Verwendung freigegeben. Register 0 ist mit dem Wert 0 belegt, Register 2 enthält die aktuelle Programmadresse, und Register 1 ist reserviert. Register 3 ist für einen Stapelzeiger vorgesehen.

Jedes Register ist 32 Bit breit. Die meisten ALU Befehle können auf Wort, Halbwort (16 Bit) oder Bytegrößen verwendet werden.

6.6.2 Speicherstruktur

Der Prozessor verfügt über zwei Adressräume, einen internen und einen externen. Der Zugriff auf die zwei Adressräume wird durch verschiedene Befehle unterschieden. Der interne Adressraum ist in seiner Größe beschränkt[24], da er innerhalb des Prozessorchips angesiedelt ist. Der externe Adressraum ist logisch lediglich durch die Registerbreite beschränkt, also 32 Bit.

Adressen im internen wie im externen Speicher sind immer Wortadressen, bezeichnen also 32 Bit Einheiten. Einzelne Bytes können nicht unabhängig adressiert werden[25].

Instruktionen liegen immer im externen Speicher, wogegen Daten sowohl im externen als auch im internen Adressraum vorhanden sein können.

[24] Eine zu erwartende Größe währen etwa vier oder acht Kilobyte.

6.6.3 Cachestruktur

Der Prozessor verwendet getrennte Instruktions- und Daten-Caches für den externen Speicher. Der interne Speicher verfügt über keinen Cache, hier wird eine konstante Zugriffszeit angenommen. Änderungen des externen Speichers durch Speicheranweisungen haben keine Auswirkung auf den Instruktions-Cache, ein Überschreiben von Befehlen während der Programmausführung („patchen“) hat eine undefinierte Wirkung auf den weiteren Verlauf der Programmausführung, abhängig von der aktuellen Belegung des Befehls-Caches.

Der Daten-Cache ist als Rückschreibe-Cache („Write-back-cache“) mit Schreiballokation („Write-allocation“) ausgelegt. Schreibzugriffe, die auf nicht im Cache enthaltenen Speicher zugreifen, haben das Holen einer gesamten Cache-Zeile zur Folge. Eine Cache-Zeile wird nur dann in den Speicher zurückgeschrieben wenn eine freie Zeile benötigt wird.

6.6.4 Ausführungskontextstruktur

Je nach Konfiguration des Prozessors können verschieden viele Ausführungskontexte im Prozessor gleichzeitig bearbeitet werden. Jeder Ausführungskontext besitzt einen eigenen Registersatz und Teile der Kontroll-Pipeline. Zusätzlich werden noch für jeden Ausführungskontext Verwaltungsinformation in der Kontrollfadensteuerungseinheit gehalten. Ein Ausführungskontext kann in verschiedenen Zuständen befinden:

Waiting	Der Ausführungskontext wartet auf die Erfüllung eines Signals
Running	Der Ausführungskontext wird bearbeitet
Stopping	Der Ausführungskontext wird gerade beendet, d.h. alle Instruktionen, die sich noch in der Ausführungs-Pipeline befinden werden ausgeführt, es werden aber keine neuen Instruktionen mehr eingefügt.
Idle	Der Ausführungskontext wird nicht bearbeitet
Suspending	Der Ausführungskontext wird gerade in seiner Ausführung unterbrochen, d.h. alle Instruktionen, die sich noch in der Ausführungs-Pipeline befinden werden ausgeführt, es werden aber keine neuen Instruktionen mehr eingefügt.
Suspended	Der Ausführungskontext ist in seiner Ausführung unterbrochen.

Der Übergang zwischen den einzelnen Zuständen wird durch Befehle an die Kontrollfadensteuereinheit ausgelöst.

[25] Inwieweit dies eine starke Beeinträchtigung darstellt muss noch untersucht werden. Viele aktuelle Prozessoren unterstützen keine unausgerichteten Speicherzugriffe, oder belegen diese mit mehreren Strafzyklen.

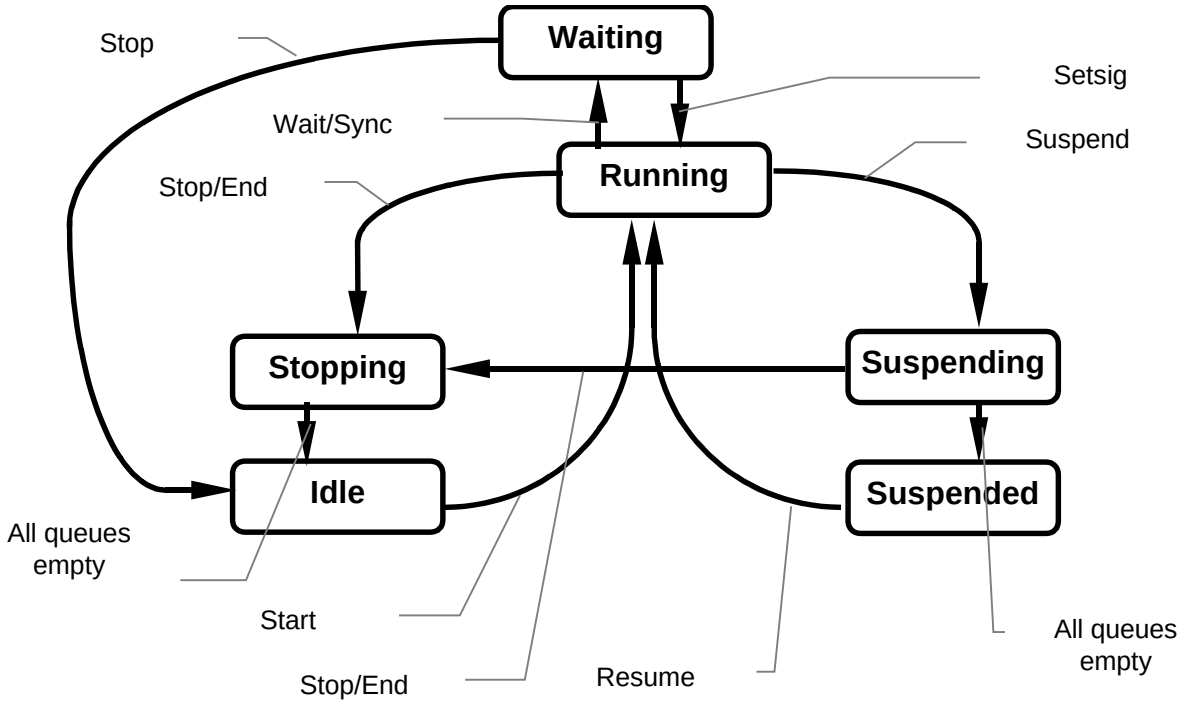


Abbildung 6-6: Zustandsübergangsdiagram der Ausführungskontexte

Eine Synchronisierung der einzelnen Ausführungskontexte in der Lade- und Speichereinheit findet nicht statt, es wird keine feste Schreib- oder Lesereihenfolge beachtet. Falls Ausführungskontexte gemeinsame Speicherbereiche nutzen, ist deshalb eine Synchronisation über explizite Anweisungen nötig.

Die Synchronisation der Ausführungskontexte wird durch 32 gemeinsame Signalbits erreicht. Diese können in atomaren Befehlen gesetzt, gelöscht oder abgefragt werden. Weiter ist es möglich, auf einzelne oder mehrere dieser Signalbits zu warten. Mit einem speziellen atomaren Warte- und Lösche-Befehl ist es möglich, kritische Sektionen durch einzelne dieser Signalbits zu schützen.[26]

Jeder Befehl der in der Kontrollfadensteuereinheit ausgeführt wird verfügt über zwei Steuerungsbits „m“ und „w“. Wird das „m“-Bit eines Befehls gesetzt, so wird eine Zuordnung des Befehls verhindert, bis alle Operationen dieses Kontrollfadens, die den Speicher benutzen, abgeschlossen sind. Durch Setzen des „w“-Bits wird verhindert, dass weitere Befehle nach diesem Befehl dekodiert werden, solange dieser Befehl nicht komplett ausgeführt ist. Durch das erste Bit kann somit eine Synchronisierung mit dem Speicher erreicht werden. Das zweite Bit kann für Warteanweisungen benutzt werden. Üblicherweise hat deshalb ein Befehl, der ein Signal setzt, das „m“-Bit gesetzt, ein Befehl der auf ein Signal wartet das „w“-Bit.

Es existieren zwei verschiedene Prioritätsstufen. Kontrollfäden mit einer höheren Priorität können je nach gewählter Strategie in verschiedenen Prozessoreinheiten bevorzugt behandelt werden. Hierzu

[26] Eine alternative Implementierung der Synchronisation mehrerer Kontrollfäden könnte durch speicherbasierte Semaphoren implementiert werden[10]. Dieses verlangt jedoch stetiges Pollen der wartenden Kontrollfäden, oder aber eine Prozessorstruktur in der die Adressen aller Semaphoren, auf die ein Kontrollfaden wartet, gehalten wird. Die hier gewählte Variante erlaubt sehr einfache Synchronisationsmechanismen für die häufigsten Anwendungen. Komplexere Synchronisationselemente werden durch einen einfachen Kernel bereitgestellt.

existiert ein Prioritätskontrollwort, das durch eine Anweisung gelesen und geschrieben werden kann. Kontrollfäden, deren Statusbit in diesem Prioritätskontrollwort gesetzt ist, werden mit hoher Priorität behandelt.

6.6.5 Befehlsstruktur

Jeder Prozessorbefehl umfasst genau ein Wort. Direktargumente können je nach Befehl zwischen 8 und 21 Bit umfassen. Die Befehlskodierung ist sehr regulär. Es werden nur sechs verschiedene Formate verwendet, die alle durch eine Kodierung in den höchstwertigen vier Bits jedes Befehlswortes kenntlich sind.

```
0....[RD ][RS1][Extension      ]
100..[RD ][RS1][RS2][Extension ]
101..[RS ][RB ][Extension      ]
110..[RD ][RB ][Extension      ]
1110.[RS ][Extension           ]
1111.[RS ][RB ][Extension      ]
```

- RD Zielregister
- RS1, RS2 Quellregister
- RB Basisregister für Adressierungen
- Extension Direktwerte, die an die Ausführungseinheit weiter gereicht werden

6.6.5.1 Adressierungsmodi

Es stehen drei Adressierungsmodi zur Verfügung, die jedoch nicht für alle Befehle genutzt werden können, so steht zum Beispiel die Adressierungsart „Indirekt mit Versatz“ nur den Lade- und Speicherbefehlen zur Verfügung.

Direkt, durch Angabe eines Wertes	4
Register direkt	r5
Register indirekt mit Versatz	[4 , r10]

PC relative Adressierung kann durch die Verwendung von r2, absolute durch r0 als Basisregister erreicht werden.

6.6.5.2 Befehlssyntax

Je nach der Anzahl der möglichen Argumente eines Befehls wird eine der folgenden Befehlsformate verwendet.

```
command.extension arg1, arg2, arg3
```

```
command.extension arg1, arg2

command.extension arg1

command.extension
```

Das Ziel der Operation (falls vorhanden) steht immer rechts, Quelloperanden immer links.

6.6.5.3 Befehlserweiterungen

.W	Wortoperation (Bits 0..31)
.H	Halbwortoperation (Bits 0..15 und Bits 16..31)
.B	Byteoperation (Bits 0..7, 8..15, 16..23 und 24..31)
.UH, .U	Oberes Halbwort (Bits 16..31)
.LH, .L	Unteres Halbwort (Bits 0..15)
.UB	Oberstes Byte (Bits 24..31)
.UMB	Zweitoberstes Byte (Bits 16..23)
.LMB	Zweitunterstes Byte (Bits 8..15)
.LB	Unterstes Byte (Bits 0..7)

6.6.6 Vergleich mit dem DLX Befehlssatz

Der verwendete Befehlssatz ist dem DLX Befehlssatz [63] ähnlich. Er besitzt das gleiche Befehlsformat und die selben Adressierungsarten. Folgende grundlegende Unterschiede sind vorhanden:

- Die Bedingungsauswertung ist im Sprungbefehl kodiert, und nicht wie bei DLX durch einen „Setze bei Bedingung“-Befehl.
- Es werden keine Befehle mit Speicherzugriffen in Teilworteinheiten unterstützt
- Auf Fließkomma- und Divisionsbefehle wurde Verzichtet.
- Unterprogrammsprünge werden nicht durch einen „Spring und Verbinde“ Befehl ausgeführt. Die Rücksprungadresse muss explizit in ein Register geladen werden.
- Zusätzliche Multimediabefehle wurden hinzugefügt.

6.6.7 Übersicht über alle Befehle

Mnemo	Command	DLX	Execution unit
BRA	Branch unconditional	J, JR	Branch
BEQ	Branch if equal	BEQZ	Branch
BNE	Branch if not equal	BNEZ	Branch

BLT	Branch if less than	1	Branch
BLE	Branch if less or equal	1	Branch
BGT	Branch if greater than	1	Branch
BGE	Branch if greater or equal	1	Branch
ADD	Add unsigned	ADDU	ALU
ADDS	Add unsigned saturated	ADD	ALU
ADDSS	Add signed saturated		ALU
SUB	Subtract unsigned	SUBU	ALU
RSUB	Reverse subtract unsigned	SUB	ALU
SUBS	Subtract unsigned saturated	2	ALU
SUBSS	Subtract signed saturated	2	ALU
MAX	Maximum unsigned	2	ALU
MAXS	Maximum signed	2	ALU
MIN	Minimum unsigned	2	ALU
MINS	Minimum signed	2	ALU
AVG	Average unsigned	2	ALU
AVGS	Average signed	2	ALU
CMP	Compare unsigned	1	ALU
CMPS	Compare signed	1	ALU
OR	Binary logical or	OR	ALU
AND	Binary logical and	AND	ALU
XOR	Binary logical exclusive or	XOR	ALU
ANDN	Binary logical and not	2	ALU
PACKU	Pack upper	2	ALU
PACKL	Pack lower	2	ALU
EXTU	Extract upper	2	ALU
EXTL	Extract lower	2	ALU
LSL	Logical shift left	SLL	ALU
LSR	Logical shift right	SRL	ALU
ASL	Arithmetical shift left	SLL	ALU
ASR	Arithmetical shift right	SRA	ALU
ROL	Rotate left		ALU
ROR	Rotate right		ALU
BFF	Bit find first		ALU
BFL	Bit find last		ALU
LDL	Load local	LW	Local memory
STL	Store local	SW	Local memory
LDM	Load memory	LW	Memory
STM	Store memory	SW	Memory
MUL	Multiply unsigned	MULTU	Multiplier
MULS	Multiply signed	MULT	Multiplier
MULC	Multiply complex	2	Multiplier
MULCS	Multiply complex signed	2	Multiplier

MULX	Multiply extended	2	Multiplier
MULXS	Multiply extended signed	2	Multiplier
MULI	Multiply internal	2	Multiplier
MULIS	Multiply internal signed	2	Multiplier
MULXI	Multiply extended internal	2	Multiplier
MULXIS	Multiply extended internal signed	2	Multiplier
MULMI	Multiply multiple internal	2	Multiplier
MULMIS	Multiply multiple internal signed	2	Multiplier
LDMC	Load internal constant	2	Multiplier
LDMCS	Load internal constant signed	2	Multiplier
WAIT	Wait for signal	3	Control
SYNC	Synchronize to signal	3	Control
SETSIG	Set signals	3	Control
CLRSIG	Clear signals	3	Control
STOP	Stop thread	3	Control
START	Start thread	3	Control
SUSPEND	Suspend threads	3	Control
RESUME	Resume threads	3	Control
END	End current thread	3	Control
GETID	Get current thread id	3	Control
NOP	No operation		Control
SETPRI	Set scheduling priority	3	Control
SETSYS	Set system control word	3	Control
FLUSHD	Flush data cache		Control
INVALD	Invalidate data cache		Control
INVALI	Invalidate instruction cache		Control
KILL	Kill Thread	3	Control
TRACE	Dump Debug	3	Control
IN	Input data		Control
OUT	Output data		Control
ILLEGAL	Illegal Operation		Control

Für eine detaillierte Beschreibung aller Befehle und des Befehlsformats siehe 10.1.

1 Kombination aus Set und Branch Befehl

2 Multimediabefehl

3 Befehl zur Kontrollfadensteuerung

7 Aufbau und Struktur des Lastprogramms

7.1 Überblick

Zur Evaluierung des Prozessorkonzeptes wird ein parallelisierter MPEG-2-Videodekoder verwendet. Dieser entstand aus einem kommerziell vertriebenen Dekoder für Microsoft Windows und wurde komplett in Assembler implementiert. Es wurde besonders darauf geachtet, alle Codeoptimierungen, die einem Prozessor mit einem Ausführungsfaden nutzen, zu implementieren. Hervorzuheben sind Software-Pipelining, Einbettung von Prozeduren, Scheduling von Anweisungsklassen und Ausführungspfaden.

Da es sich allerdings um einen parallelisierten Dekoder handelt, sind Einbußen durch Synchronisationsanweisungen für den einfädigen Fall zu berücksichtigen. Hierauf wird in einem späteren Teil dieses Kapitels noch eingegangen.

7.2 Programmaufbau

7.2.1 Anwendungselemente

Der Dekoder ist entsprechend dem Aufbau des MPEG-2-Videodekodierungsalgorithmus in einer Pipeline aufgebaut. Die einzelnen Elemente der Pipeline sind in entsprechende Module unterteilt, und werden im Weiteren hier besprochen. Neben dieser Pipeline wird auch noch ein minimaler Betriebssystemkern („Nanokernel“) benötigt, der Synchronisation und Speicherverwaltung durchführt. Der Aufbau der Anwendung stellt sich wie folgt dar:

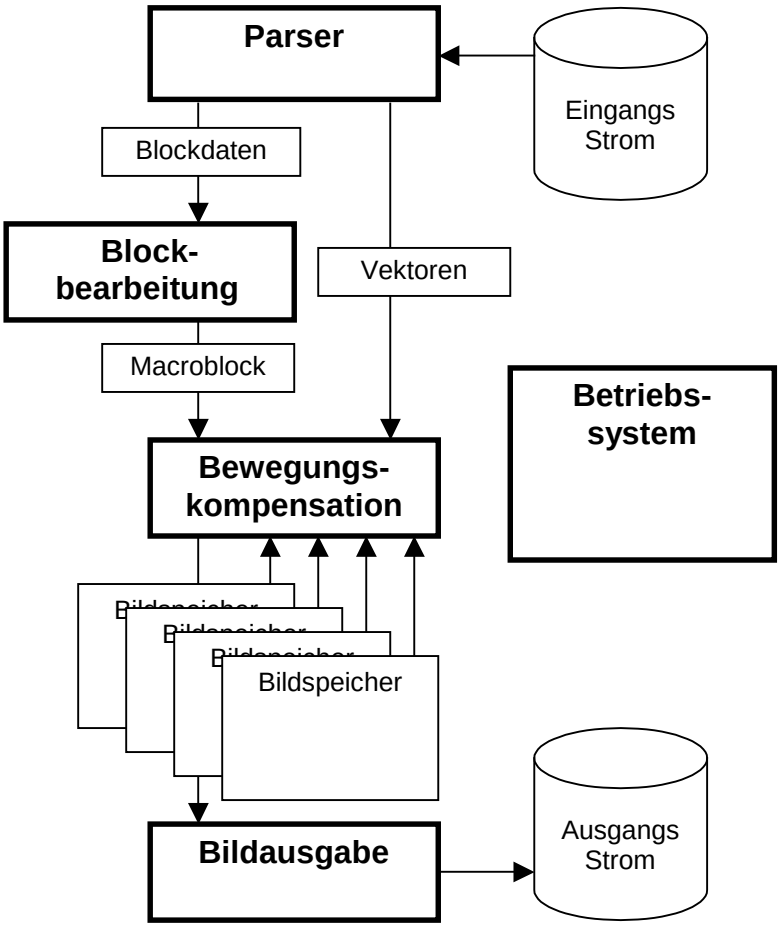


Abbildung 7-1: Aufbau der MPEG Videodekodierung

Zwischen den einzelnen Pipeline-Stufen befinden sich Datenpuffer, die diese Stufen voneinander trennen, und somit eine gleichzeitige und unabhängige Ausführung der Stufen gewährleisten. Diese Puffer sind typischerweise mehrfach vorhanden, so dass eine stärkere Entkopplung und Vervielfältigung der einzelnen Stufen möglich wird.

7.2.1.1 Betriebssystemkern

Der Betriebssystemkern stellt die minimalen Elemente zur Verfügung, die benötigt werden, um einen mehrfädigen Prozessor zu betreiben.

- **Speicherverwaltung** : Der Betriebssystemkern stellt eine Speicherverwaltung zur Verfügung, die es erlaubt, den lokalen und globalen Speicher dynamisch zu verwalten.
- **Kontrollfadenverwaltung** : Kontrollfäden können angelegt, gestartet und beendet werden. Des weiteren wird noch ein prioritätsgesteuertes kooperatives Multitasking implementiert, das es erlaubt, die auf dem Prozessor vorhandenen Kontrollfäden zu virtualisieren. Der Betriebssystemkern simuliert beliebig viele virtuelle Kontrollfäden, die auf die jeweils zur Verfügung stehenden Ausführungskontexte des Prozessors abgebildet werden. Dies erlaubt es, die verwendete Last unabhängig von der Struktur des Prozessors zu parallelisieren.

- **Synchronisation** : Der Betriebssystemkern stellt, basierend auf den Synchronisationsprimitiven des Prozessors, komplexere Synchronisationselemente bereit, die, unter Ausnutzung des kooperativen Multitaskings, eine bessere Ausnutzung des Prozessors ermöglichen.

Der Betriebssystemkern besteht aus einer Sammlung einzelner Routinen und Makros, die in das ausgeführte Programm eingebunden werden. Auf einen Speicherschutz und weitere Konzepte moderner Betriebssysteme, wie Ein- Ausgabevirtualisierung etc., wurde aus Gründen der Vereinfachung verzichtet.

7.2.1.2 Eingabe der Daten

Die Daten werden über einen simulierten Eingabeport eingelesen und in einem zirkulären Pufferspeicher gehalten. Da nur elementare Videoströme verwendet werden, kann auf einen Demultiplexer verzichtet werden. Der Eingabepuffer ist entsprechend der MPEG-Konvention groß genug, so dass während der Dekodierung nicht auf einen Unterlauf geachtet werden muss.

7.2.1.3 Parser

Der Parser ist entsprechend dem MPEG-2-Videodatenformat hierarchisch aufgebaut. Da es sich bei der MPEG-2-Syntax um eine LL1-Sprache[27] handelt, finden ein Parser des rekursiven Abstiegs Verwendung. Der Parser besteht aus folgenden Stufen:

- Sequence (SequenceHeader, SequenceExtension)
- GroupOfPicture (GroupOfPictureHeader, UserData)
- Picture (PictureHeader, PictureExtension)
- Slice (SliceHeader)
- Macroblock (MacroblockType, MotionVectors)
- Block
- Run Length Pair

Jede Stufe ruft entsprechend mehrfach die jeweils darunter liegende Ebene auf, um deren syntaktisches Element zu bearbeiten. Aufgrund des verschieden häufigen Auftretens eines syntaktischen Elementes, ergeben sich folgende Aufrufhäufigkeiten in einer typischen MPEG-2-Videosequenz.

[27] Da die Abstiegstiefe durch die nicht rekursive Struktur der MPEG-2 Syntax beschränkt ist, handelt es sich eigentlich um eine reguläre Sprache, so dass auf einen Kellerautomaten verzichtet werden könnte. Aufgrund der einfacheren und klareren Struktur wurde allerdings ein LL1-Parser verwendet.

Syntaxelement	Durchschnittliche Auftritts-häufigkeit des untergeordne-ten Syntaxelements	Durchschnittliche Auftritts-häufigkeit pro Sekunde
Sequence	1 GroupOfPicture	2
GroupOfPicture	15 Pictures	2
Picture	30 Slices	30
Slice	40 Macroblocks	900
Macroblock	5 Blocks	36.000
Block	10 Run Length Pairs	180.000
Run Length Pair		1.800.000

Entsprechend des starken Anteils des „Run Length Pair“-Parsers an der Ausführung ist dieser Teil des Parsers nicht als eigene Routine, sondern als Schleife innerhalb des Blockparsers implementiert. Die Lauflängendekodierung ist, obwohl sie eigentlich Bestandteil der Blockbearbeitung ist, ebenfalls innerhalb dieser Schleife implementiert.

Innerhalb des Parsers finden kaum Multimediaanweisungen Verwendung, da weder komplexe Rechnungen noch reguläre Datenbearbeitungen stattfinden.

7.2.1.4 Blockbearbeitung

Die Makroblockbearbeitung führt folgende Schritte der Dekodierung durch:

- Inverse Quantisierung
- Inverse Diskrete Cosinus Transformation (IDCT)

Diese Aufgaben sind regulär und müssen für alle Blöcke eines Bildes gleich ausgeführt werden[28]. Diese Routinen profitieren sehr stark von den Multimediaanweisungen des Prozessors, da sie aufgrund ihrer regulären Struktur der SIMD-Architektur nahe liegen.

7.2.1.5 Bewegungskompensation

Die Bewegungskompensation besteht aus vier Schritten:

- Berechnung der Kompensationsvektoren basierend auf dem vorherigen Vektor und den Änderungsdaten, die im Strom gespeichert sind
- Bilden des Kompensationsblocks aus den Quellbildpuffern
- Addieren der Korrekturterme
- Schreiben des Ergebnisses in den Zielbildpuffer

[28] Ein leichter Unterschied besteht in der inversen Quantisierung des DC Koeffizienten bei Intra- und NonIntrablöcken. Dieser kann jedoch leicht durch einen zusätzlichen Funktionseinsprung implementiert werden, so dass die eigentliche Schleife für beide Blocktypen identisch ist.

Die letzten drei Schritte profitieren aufgrund ihrer regelmäßigen Struktur sehr stark von den SIMD-Befehlen des Prozessors. Die Berechnung der Kompensationsvektoren ist eine sehr ungleichförmige Bearbeitung und zieht deshalb keinen Nutzen aus den Multimediaerweiterungen des Prozessors.

7.2.1.6 Bildausgabe

Die Aufgabe der Bildausgabe besteht darin, die Bilddaten entsprechend ihrer zeitlichen Reihenfolge im Film in den Bilddarstellungspuffer zu kopieren[29]. Diese Aufgabe ist sehr regulär. Da sie aber nur aus Kopieranweisungen besteht, kann sie keinen Nutzen aus Multimediaanweisungen ziehen. Die Bildausgabe ist von den restlichen Schritten der Bearbeitung sehr stark abgetrennt, da sie nur auf Bildebene mit der Dekodierung synchronisiert werden muss.

7.2.2 Speicheraufteilung

Für die Speicheraufteilung ist besonders die Verteilung der Datenstrukturen zwischen dem globalen und lokalen Speicher von Interesse. Als Ziele der Verteilung sollte eine maximale Ausnutzung des lokalen Speichers und einer Gleichverteilung der Auslastung der entsprechenden Lade- und Speichereinheiten gelten. Folgende Kriterien für die einzelnen Datenstrukturen gehen in die Auswahlsentscheidung ein:

- Größe der Struktur
- Häufigkeit des Zugriffs
- Zugriffstruktur (geordnet zu ungeordnet, nur lesen zu lesen und schreiben)
- Gleichzeitige Verwendung mit anderer Struktur

Folgende Datenstrukturen mit ihren entsprechenden Kriterien werden verwendet.

[29] In einer Echtweltanwendung käme je nach verwendeter Bilddarstellungshardware noch eine Konvertierung des 4:2:0 Formates der Bildpuffer in das Format des Bilddarstellungspuffers hinzu. Da die meisten Graphikkarten in PCs aber Bilddarstellungspuffer im 4:2:0 Format unterstützen und eingebettete Anwendungen meist mit einem 4:2:0 Puffer arbeiten, wurde auf diesen Schritt verzichtet.

Datenstruktur	Größe in Bytes	Häufigkeit in Bytes/sec	Zugriffsstruktur	Gleichzeitig mit	Position
Bildspeicher	518.400	62.208.000	Geordnet, lesen und schreiben	Blockpuffer	Global
Blockpuffer	768	248.832.000	Geordnet, lesen und schreiben	Bildspeicher, Huffmantabellen, Quantisierungstabellen	Lokal
Macroblockdaten	128	10.368.000	Ungeordnet, lesen und schreiben	Blockpuffer, Huffmantabelle	Lokal
Huffmantabelle	29.440	16.200.000	Ungeordnet, nur lesen	Blockpuffer, Macroblockdaten, Scantabellen	Global
Scantabellen	512	7.200.000	Ungeordnet, nur lesen	Blockpuffer, Huffmantabelle	Global
Quantisierungstabellen	512	15.552.000	Geordnet, lesen und schreiben	Blockpuffer	Lokal

Dies ergibt eine Gesamtlast von 85 MBytes/sec für den globalen, und 274 MBytes/sec für den lokalen Speicher. Man erkennt eine deutlich stärkere Auslastung des lokalen Speichers. Da dieser aber vier Bytes pro Takt bedienen kann, nivelliert sich dies auf 68 Millionen Zugriffe pro Sekunde[30].

7.2.3 Parallelisierung

Um aus der parallelen Struktur des Prozessors maximalen Nutzen zu ziehen, wurde der Dekoder entsprechend seiner Struktur parallelisiert. Dazu wurden die durch den Betriebssystemkern bereitgestellten virtuellen Kontrollfäden verwendet.

7.2.3.1 Kontrollfadenmodell

Das virtuelle Modell der Kontrollfäden stellt eine Erweiterung des im Prozessor vorhandenen Modells dar. Es werden noch zwei zusätzliche Kontrollfadenzustände hinzugenommen. Es existieren somit folgende statische Zustände:

Passive Der Kontrollfaden ist nicht zur Ausführung bereit.

Running Der Kontrollfaden wird ausgeführt.

Preempted Der Kontrollfaden ist aus dem Prozessor verdrängt.

Waiting Der Kontrollfaden wartet auf ein Ereignis.

Blocked Der Kontrollfaden ist durch eine Kernsperre blockiert.

Ein virtueller Kontrollfaden ist nur in den Zuständen „Running“ und „Blocked“ im Prozessor als physikalischer Kontrollfaden repräsentiert. Da es sich um ein kooperatives Multitaskingsystem handelt, kann das Verdrängen eines Kontrollfadens aus dem Prozessor nur durch freiwillige Abgabe oder

[30] Aufgrund zusätzlicher Verwaltungsdaten im lokalen Speicher ergibt sich eine Auslastung von 311 MBytes/sec für den lokalen Speicher.

durch Warten auf ein Synchronisierungselement geschehen. Dies erlaubt einen sehr leichtgewichtigen Kontextwechsel.

Es werden zwei Kontrollfadenprioritäten unterstützt. Ein verdrängter virtueller Kontrollfaden höherer Priorität wird, bei Freiwerden eines physikalischen Kontrollfadens, bevorzugt behandelt. Wird ein Prozessor mit prioritätsgesteuertem Scheduling untersucht, so werden die Prioritäten des virtuellen Kontrollfadens auch für den physikalischen Kontrollfaden übernommen.

7.2.3.2 Aufteilung in Kontrollfäden

Die Gesamtlast der Dekodierung wird in drei Bereiche aufgeteilt:

1. Parser
2. Blockbearbeitung
3. Die Bildausgabe

Da sich der Parser aufgrund seiner sequenziellen Struktur nur schlecht parallelisieren lässt, wurde ihm ein einzelner Kontrollfaden zugeordnet. Die Blockbearbeitung wurde in mehrere Kontrollfäden aufgeteilt. Ein weiterer Kontrollfaden wird für die Bildausgabe verwendet, da diese nur sehr schwach mit der Dekodierung selbst synchronisiert werden muss. Für die Blockbearbeitung werden acht Kontrollfäden bereitgestellt. Somit überlappen sich während der Dekodierung mehrere Makroblöcke gegenseitig, es wird aber immer nur ein Makroblock im Parser bearbeitet.

Der Parser-Kontrollfaden läuft mit erhöhter Priorität, alle anderen Kontrollfäden mit niedriger. Dies ist damit zu begründen, dass der Parser-Kontrollfaden aufgrund seiner sequenziellen Last nicht parallelisiert werden kann und so die Gesamtausführung dominiert.

7.3 Programmverhalten

Im folgenden Abschnitt werden Eigenheiten des verwendeten Programms erläutert. Dies ist nützlich, um die Ergebnisse der Simulation besser deuten zu können.

7.3.1 Lastverteilung

Wird die Auslastung des Prozessors auf verschiedene Routinen untersucht, ergibt sich folgendes Bild:

Anwendungselemente	Routinen	Anteil an ausgeführten Instruktionen	
Kontrollfadenverwaltung	Kontextwechsel	0,65%	1,59 %
	Initialisierung	0,37%	
	Warteschlangen	0,57%	
Parser	Header	1,51%	14,14%
	Bewegungsvektoren	2,03%	
	Koeffizienten	10,60%	
Blockbearbeitung	Inverse Quantisierung	24,08%	53,09%
	Inverse DCT	19,68%	
	Sonstiges	9,33%	
Bewegungskompensation	Vektorberechnung	1,20%	19,96%[31]
	Unidirektionale Kompensation	15,39%	
	Bidirektionale Kompensation	3,37%	
Bildkomposition	Speichern in Bildspeicher	7,45%	7,45%
Bilddarstellung	Ausgabe aus Bildspeicher	3,77%	3,77%

Man erkennt, dass sich der Großteil der Last im parallelisierbaren Teil der Anwendung findet. Nur etwa 14% der Anweisungen liegen im sequenziellen Parser-Programmteil. Dies lässt auf eine durch Parallelisierung maximal erreichbare Leistungssteigerung um einen Faktor von etwa sieben schließen.

[31] Der große Unterschied im Verhältnis zu den Werten aus 5.4.2 ergibt sich daraus, dass hier lediglich die Anzahl der ausgeführten Instruktionen, nicht aber der benötigten Zeit aufgetragen ist. Da die Bewegungskompensation stark unter Speicherlatenzen leidet, ergeben sich zusätzliche Zeiten, die hier nicht erfasst sind.

7.3.2 Instruktionsmischung

Im Schnitt ergibt sich für eine längere Dekodierungssequenz folgende Instruktionsmischung:

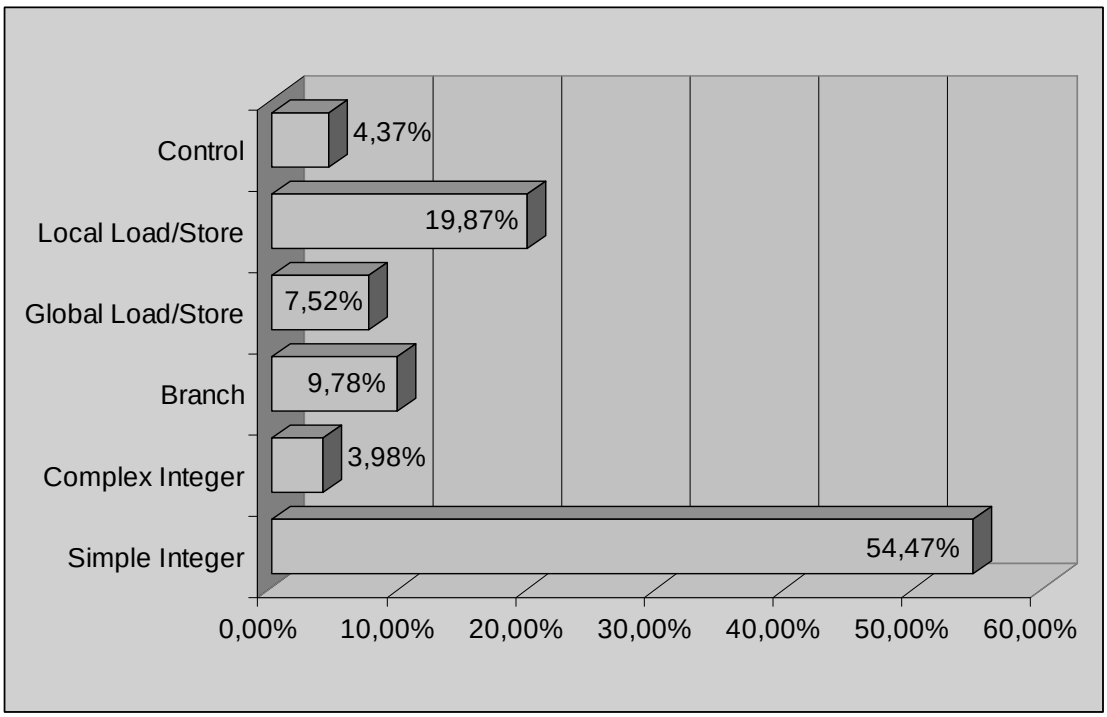


Abbildung 7-2: Durchschnittliche Instruktionsmischung

Die benötigten Instruktionen sind allerdings nicht gleichmäßig über den gesamten Dekodierungsablauf verteilt. Betrachtet man eine Dekodierungssequenz mehrerer Bilder ergibt sich folgende Verteilung der Instruktionstypen:

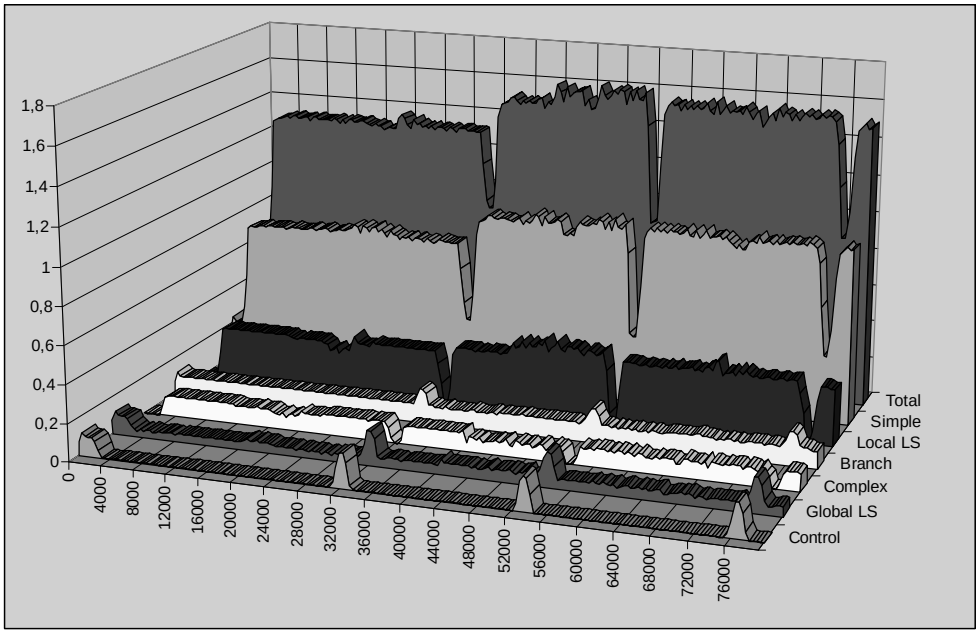


Abbildung 7-3: Lokale Instruktionsmischung wärend der Dekodierung

Man sieht hier die Instruktionsmischung über die Dekodierung dreier Frames, zuerst eines I, dann eines P und schließlich eines B-Frames (die untere Achse ist in 1000 Takten gerechnet). Die Gipfel im Bereich der Kontroll- und der globalen Lade-/Speichereinheit zeigen jeweils den Moment einer

Bildausgabe an. Man erkennt weiter, dass während der Dekodierung des B- und P-Frames ein höherer Bedarf an globalen Lade/Speicheranweisungen besteht als während des I-Frames.

Betrachtet man die Mischung noch im Detail, ergeben sich noch stärkere Ungleichheiten der Verteilung:

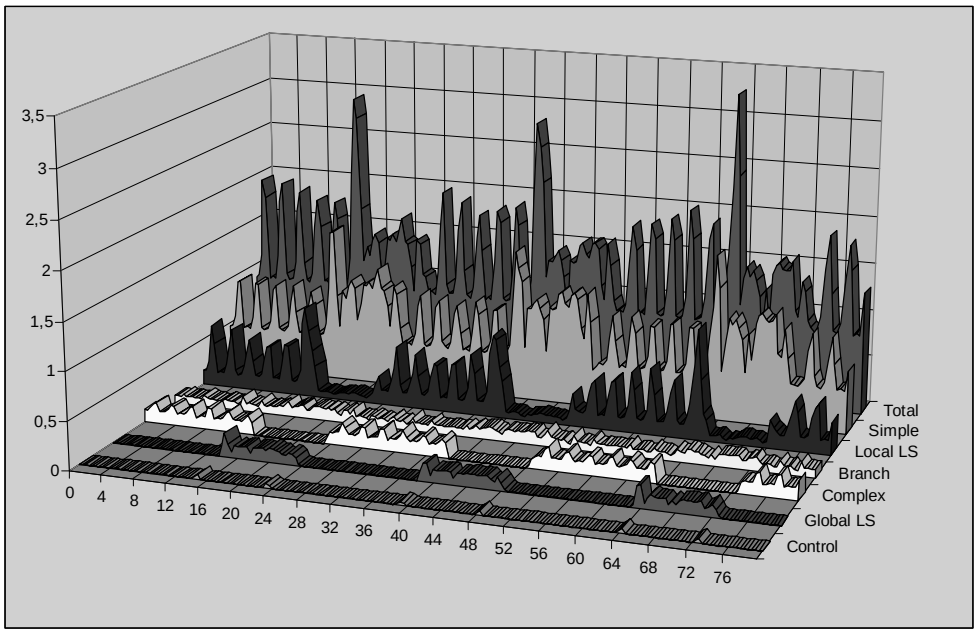


Abbildung 7-4: Instruktionsmischung wärend eines Intra Macroblocks

Hier wird die Dekodierung mehrerer Macroblöcke eines I-Frames wiedergegeben. Anhand der Gipfel in der komplexen Integereinheit sind die jeweils sechs Blöcke jedes Makroblocks zu erkennen, wobei das Tal jeweils durch die inverse Quantisierung und die Spitzen durch die IDCT geformt werden. An den leichten Hügeln im Kontrollbereich erkennt man die Übergabe der Ausführung zwischen Parser und Blockdekodierer. Während des Parsens zeigen sich die Zugriffe auf die Huffmantabellen durch eine leichte Erhöhung der globalen Lade-/Speicheranweisungshäufigkeit und der Brancheinheiten. Der starke Anstieg im Bereich der lokalen und globalen Lade/Speichereinheit, sowie der einfachen Integereinheiten ist durch das Speichern der fertigen Blöcke in den Bildspeicher verursacht.

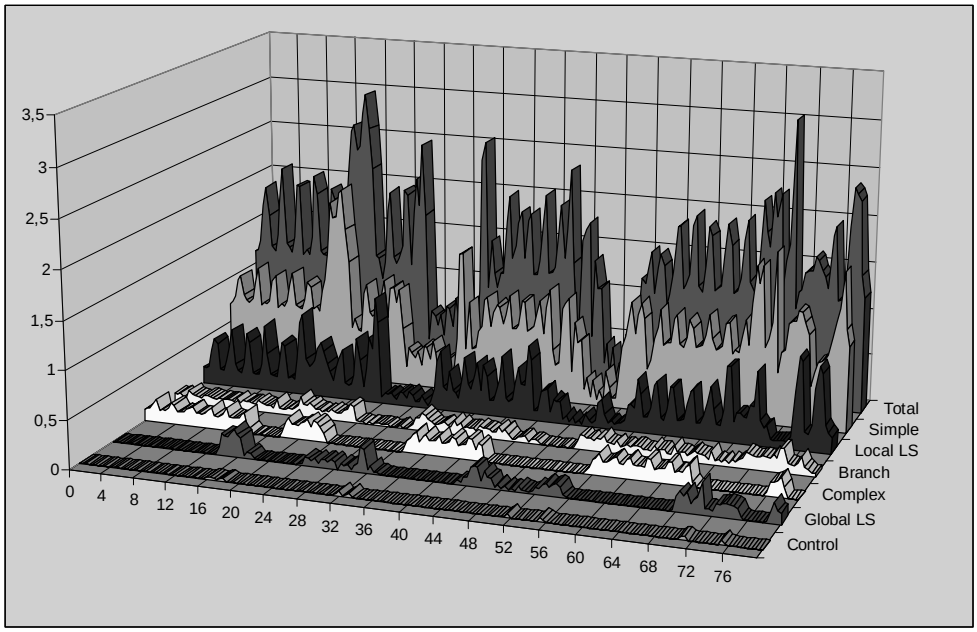


Abbildung 7-5: Instruktionsmischung während eines Makroblocks mit Bewegungskompensation

In einem Makroblock eines B-Frames ist die Verteilung nicht mehr so regelmäßig, da sich für jeden Makroblock (bzw. Block) mehr Optionen der Codierung ergeben. Dies wird daran deutlich, dass nicht immer sechs Blöcke in jedem Block vorhanden sind (sichtbar an den Spitzen in der komplexen Integereinheit). Man sieht hier auch die zusätzlichen Instruktionen der Lade-/Speichereinheit, die für die Bewegungskompensation benötigt werden.

Im mehrfädigen Fall ist diese deutliche Trennung der Anweisungsarten während der Ausführung der einzelnen Dekodierungsschritte nicht mehr zu erkennen.

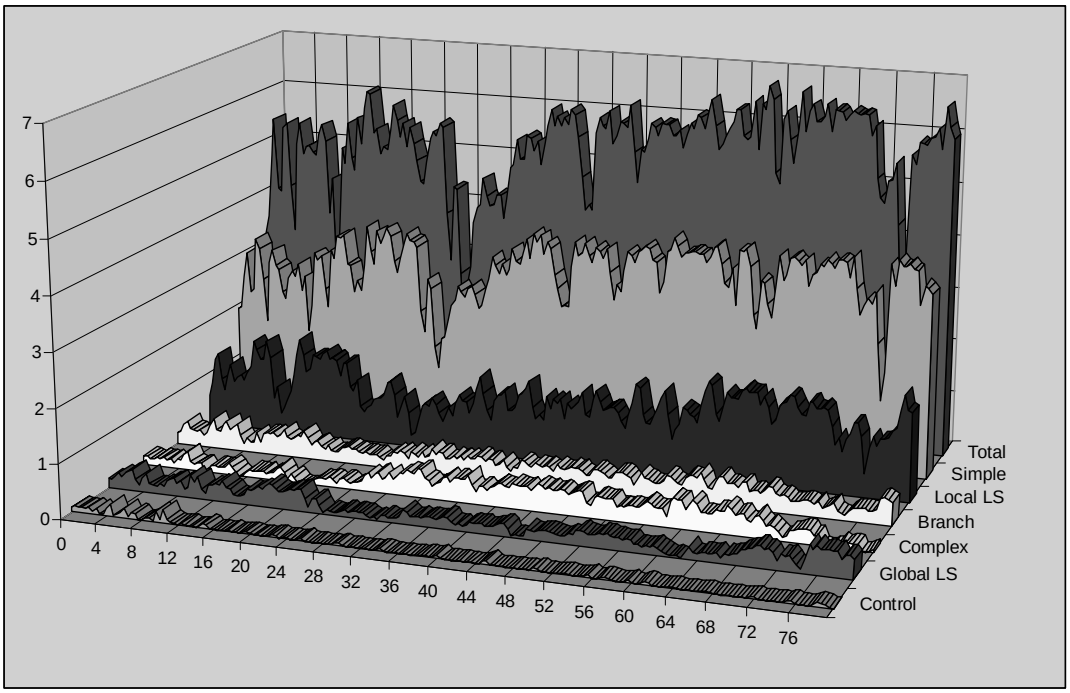


Abbildung 7-6: Instruktionsmischung im mehrfädigen Fall

Durch die Verwendung mehrerer Kontrollfäden haben sich die einzelnen Teile der Dekodierung ineinander verschliffen. Lediglich die leichten Erhöhungen in der Verwendung der Kontrollinstruktionen zeigt noch den Übergang zum nächsten Makroblock im Parser-Kontrollfaden. Die Kontrollinstruktionen am Beginn des Betrachtungszeitraumes stammen übrigens von der Ausgabe des vorherigen Bildes.

7.3.3 Speicherzugriffe

Im folgenden soll die Speichernutzung des Dekoders erläutert werden. Relevant ist hier lediglich der globale Speicher, da das Verhalten des lokalen Speichers von externen Parametern wie der Speicherzugriffsgeschwindigkeit oder der Cache-Größe und -Struktur unabhängig ist.

Dazu wurden während eines Dekodierlaufes alle Speicherzugriffe protokolliert. Das nun folgende Diagram zeigt über der Nummer des Zugriffes in Programmreihenfolge (horizontale Achse) die Speicheradresse in KByte (vertikale Achse) aufgetragen. Die Stärke der Schwärzung eines Bereichs beschreibt die Häufigkeit eines Zugriffes auf diesen.

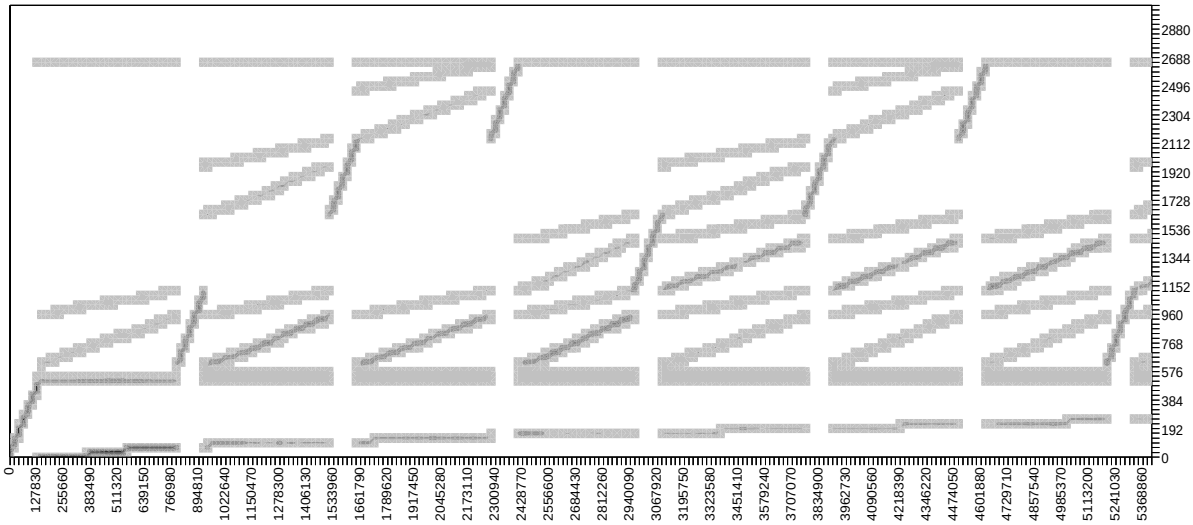


Abbildung 7-7: Speicherzugriffsfolge während der Dekodierung

Man erkennt an diesem Diagramm den Speicheraufbau, sowie die groben Zugriffsmuster des Dekoders. Der Speicheraufbau ist wie folgt:

0-511 KByte	Eingabepuffer
512-631 KByte	Huffmandekodierungstabellen
632-2655 KByte	Bildspeicher
2656-2663 KByte	Scan-Tabelle

Entsprechend können nun die Zugriffsmuster analysiert werden. Bis etwa zum 15.000. Zugriff wird der Eingabepuffer gefüllt. Danach wird bis zum etwa 80.000ten Zugriff ein I-Frame dekodiert. Man erkennt die Verwendung des ersten Bildspeichers (getrennt in Y und UV Anteil), sowie die Zugriffe auf

einen Teil der Huffmandekodierungstabellen. In den nächsten 10.000 Zugriffen wird der Bildspeicher dann in den Darstellungsspeicher übertragen. Man erkennt im weiteren den stetigen Wechsel zwischen Dekodieren und Kopieren und kann daran die einzelnen Bilder leicht unterscheiden. Das zweite Bild, das dekodiert wird, ist ein B-Frame. Man erkennt den Zugriff auf den dritten (als Ziel) und den ersten Bildpuffer (als Referenz). Das vierte Bild, das dekodiert wird, ist ein P-Frame, dies wird bei seiner Verwendung als Referenz während der Dekodierung des fünften Bildes deutlich.

Dieses Diagram zeigt deutlich die beiden typischen Zugriffsmuster, die bei der Dekodierung auftreten: Den gleichmäßig ansteigenden Zugriff auf die Bildspeicher[32], und den ungeordneten, flächendeckenden Zugriff auf die Huffmandekodierungstabellen.

Im nächsten Diagramm wird die Anzahl auftretender Cache-Fehlzugriffe (für einen einfach assoziativen Cache mit einer Zeilengröße von 32 Bytes) über der Speicheradresse aufgetragen dargestellt. Dies wird noch durch die Anzahl der Zugriffe auf die Speicherstelle insgesamt ergänzt.

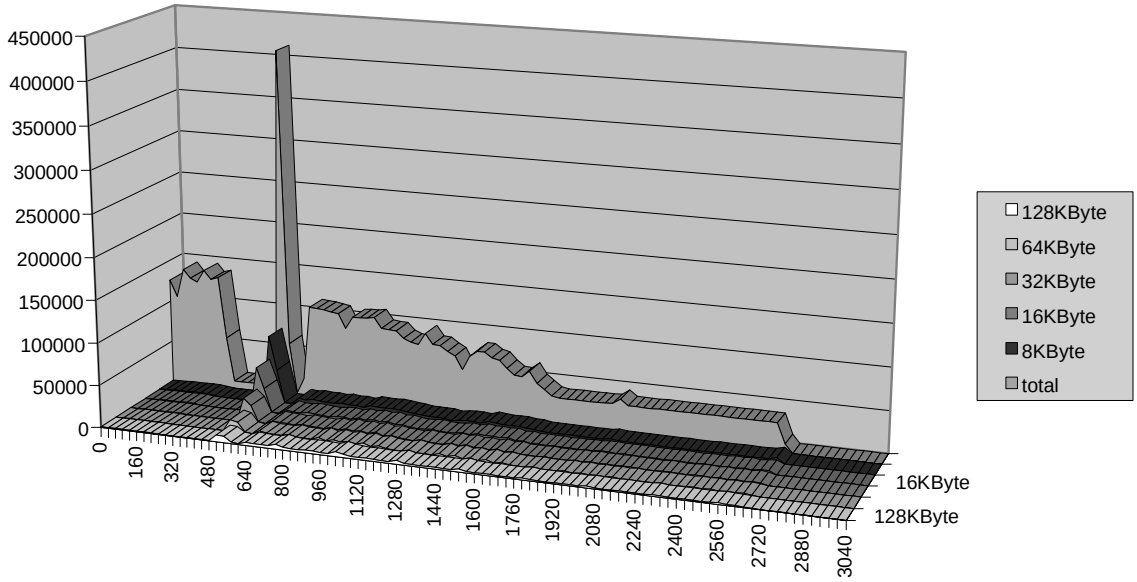


Abbildung 7-8: Cache-Fehlzugriffshäufigkeit in Abhängigkeit von Adresse und Cache-Größe

Es ist zu erkennen, dass im Bereich der Bildspeicher durch einen vergrößerten Cache nur ein geringer Gewinn erzielt werden kann. Im Bereich der Huffmandekodierungstabellen ergibt sich allerdings durch die Vergrößerung des Caches eine deutliche Reduzierung der nötigen Zugriffe auf den Hauptspeicher.

Die Reduzierung der Zugriffe auf den Hautspeicher durch die Verwendung eines Caches hat bei Bildspeichern und Huffmandekodierungstabellen verschiedene Ursachen. Bei den Huffmandekodierungstabellen ist vor allem die temporale Lokalität (ein Element wird bald nach seiner Benutzung wie-

[32] Da die vertikale Distanz eines Makroblockes zu seinem Referenzblock beschränkt ist (durch den beschränkten Suchbereich des Encoders und die Auflösung von Konturen bei schnellen Bewegungen – „Motion Blur“), bewegt sich auch der Zugriff auf das Referenzbild relativ gleichmäßig aufwärts.

der benutzt), bei den Bildspeichern die räumliche Lokalität (mit einem Element werden meist auch seine Nachbarn benötigt) ausgeprägt. Dies wird durch folgendes Diagramm deutlich, in dem verschiedene Größen der Cachezeilen bei einer festen Cachegröße von 32KByte untersucht werden. Kleine Zeilengrößen bevorzugen temporale Lokalität, wogegen große Cachezeilen räumliche Lokalität besser nutzen.

Über der Speicheradresse wurden hier die Anzahl der Zugriffstakte auf den Hauptspeicher bei einem Zugriffsmuster von 7-2-2-2[33] und einem 64Bit Datenbus aufgetragen.

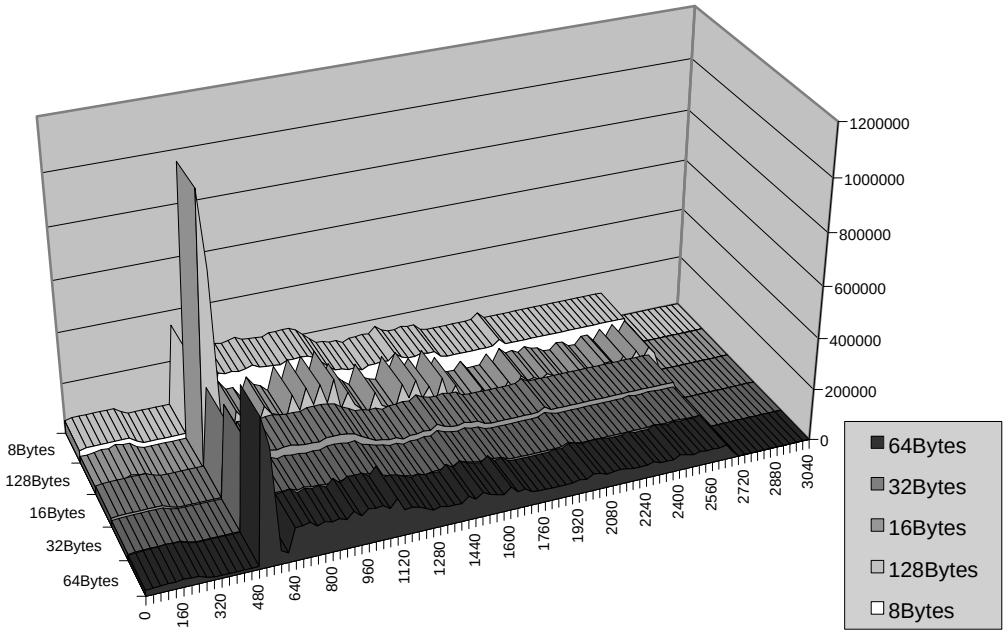


Abbildung 7-9: Cache-Fehlzugriffshäufigkeit in Abhängigkeit von Adresse und Cache-Zeilenlänge

Man erkennt, dass eine Größe von 32-Bytes pro Cache-Zeile einen guten Kompromiss zwischen den Bedürfnissen des Bildspeichers und der Huffmandekodierungstabellen darstellt.

7.3.4 Sprungvorhersage

Bei der MPEG Dekodierung treten hauptsächlich zwei Sprungmuster auf:

- Schleifen mit einer fester Anzahl von Durchläufen
- Bedingungen mit meist unvorhersehbarem, da datenabhängigem Sprungverhalten.

Beide Sprungmuster profitieren nur minimal von einer dynamischen Sprungvorhersage. Es ist deshalb zu vermuten, dass ein Prozessor mit statischer Sprungvorhersage vergleichbare Ergebnisse liefert wie ein Prozessor mit dynamischer Sprungvorhersage. Da ein mehrfädiger Prozessor in der Lage ist, Latenzen, die durch falsch vorhergesagte Sprünge auftreten, durch andere Kontrollfäden zu

[33] Es handelt sich hierbei im Bustakte und nicht um Prozessortakte, da wir uns in diesem Fall lediglich für die Auslastung des Datenbusses interessieren.

überbrücken, ist im mehrfädigen Fall nur eine geringe Steigerung durch eine dynamische Sprungvorhersage zu erwarten.

7.4 Auswirkung einer parallelen Last auf einen einfädigen Prozessor

Durch die Verwendung eines parallelen Lastprogramms sind natürlich nachteilige Auswirkungen auf die Ausführungszeit bei einem einfädigen Prozessor zu erwarten. Eine grobe Abschätzung dafür lässt sich durch die Zeit, die in den Synchronisationsroutinen verbracht wird, finden. Da sich dies im Bereich von etwa einem Prozent bewegt, kann sie für die weitere Betrachtung meist außer Acht gelassen werden.

Es wird allerdings bei den verschiedenen Prozessorkonfigurationen darauf geachtet, dass nicht einseitig für den mehrfädigen Fall optimiert wird.

8 Simulation

8.1 Überblick

Die Simulationen, die in dieser Arbeit durchgeführt wurden, lassen sich in zwei Gruppen gliedern. In der ersten Gruppe wurde versucht, die maximale Leistung des Prozessors bei verschiedenen Kontrollfaden-/Befehlszuordnungsbreitenkonfigurationen zu ermitteln. Dies geschieht unter der Annahme von teilweise unrealistischen Konfigurationsparametern.

Im zweiten Schritt wird durch eine Einschränkung der unrealistischen Parameter eine Prozessorkonfiguration ermittelt, die in den nächsten Jahren implementierbar wäre.

Dieser zweistufige Ansatz wurde gewählt, um nicht frühzeitig durch die Limitierung eines Konfigurationsparameters die Leistung des realistischen Prozessors durch einen Flaschenhals zu beschränken. Auch zeigt der maximale Prozessor den Weg der Entwicklung, so dass nützliche Schritte der Weiterentwicklung daraus abgelesen werden können.

8.2 Optimierung eines Maximalprozessors

In diesem Abschnitt wird versucht, mit Hilfe der Simulation einen Prozessor zu spezifizieren, der einen maximalen Durchsatz an Instruktionen pro Takt erreicht. Hierbei wird keine Rücksicht auf die Kosten oder Implementierbarkeit genommen, da eine obere Grenze der Architektur angenähert werde soll.

8.2.1 Grundstruktur des Maximalprozessors

Die Grundstruktur des Maximalprozessors ist durch folgende Eigenschaften gegeben:

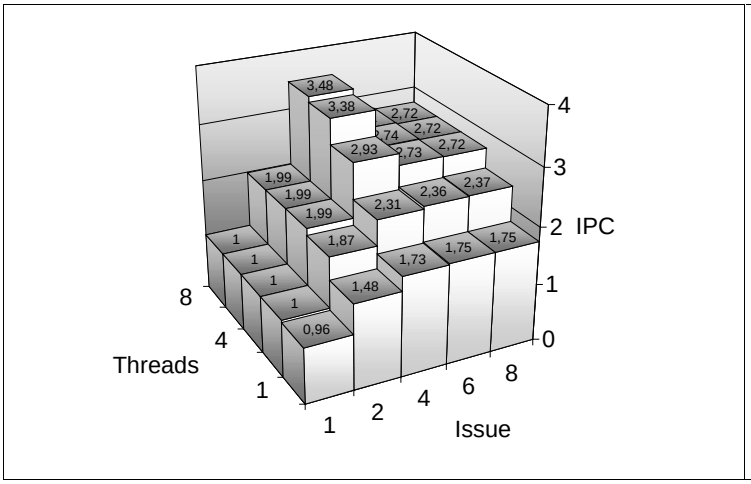
- 32 Register pro Kontrollfaden
- Vier einfache Integereinheiten
- Eine globale und eine lokale Lade- und Speichereinheit
- Je eine Kontrollfadensteuerungs-, komplexe Integer- und Sprungeinheit
- Zweistufige dynamische Sprungvorhersage (die Größe des BHR beträgt 8bit, die BHT umfasst 2048 Einträge).
- 64-Bit Datenbus

Um einen maximalen Durchsatz zu erreichen, werden einige unrealistische Annahmen getroffen. Folgende Eigenschaften zeichnen den Maximalprozessor aus:

- Daten- und Instruktions-Caches in Hauptspeichergröße

- Acht parallele Befehlslade- und Dekodiereinheiten, die jeweils acht Instruktionen pro Takt bearbeiten können
- Befehlsumordnungspuffer mit jeweils 256 Einträgen pro Ausführungseinheit
- Befehlsrückordnungspuffer mit jeweils 256 Einträgen pro Kontrollfaden
- Bis zu 64 gleichzeitig spekulierte Sprungpfade
- Ein eigener Resultatsbus pro Ausführungseinheit

Gemessen wurden Konfigurationen von jeweils 1, 2, 4, 6 und 8 Kontrollfäden mit 1, 2, 4, 6 und 8fach superskalarer Befehlszuordnung.



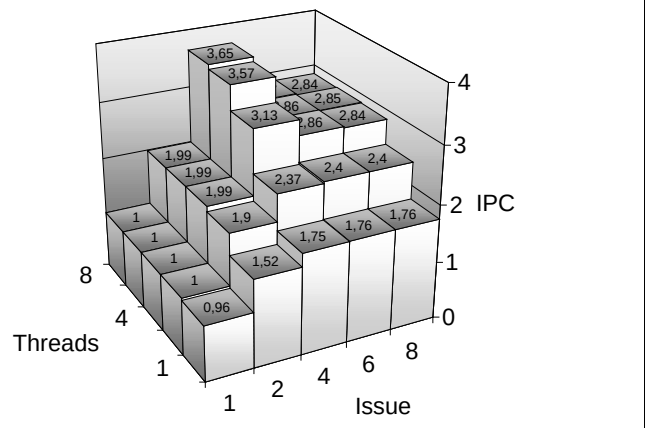
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	je 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-1: Anfänglicher Maximalprozessor

Zu erkennen ist anhand dieser ersten Messung, dass ein einfädiger Prozessor bei der verwendeten Last bei etwa 1,75 Instruktionen pro Takt (IPC) sein Maximum erreicht. Eine Steigerung dieser Rate ist auch bei höherer Befehlszuordnungsbreite nicht zu erwarten. Als weiteres Ergebnis kann man auch sehen, dass ein vierfädiger Prozessor einen zweifach superskalaren Prozessor zu 99% und ein achtfädiger Prozessor einen vierfach superskalaren Prozessor zu 87% auslastet.

Des Weiteren ist der starke Einbruch der vier- und mehrfädigen Prozessoren bei sechs- und achtfacher Befehlszuordnungsbreite zu erkennen. Dieser unerwartete Effekt soll im nächsten Abschnitt untersucht werden.

Eine erste Analyse der Lastverteilung der Integerinstruktionen zeigt, dass eine sehr ungleiche Verteilung auf die vier verwendeten Integereinheiten stattfindet. Um diese ungleiche Verteilung aufzuheben, wird den vier Integereinheiten ein gemeinsamer Befehlsumordnungspuffer zugeordnet, der jedoch je vier Befehle pro Takt akzeptieren und starten kann.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-2: Maximalprozessor mit gemeinsamen Umordnungspuffern der einfachen Integereinheiten

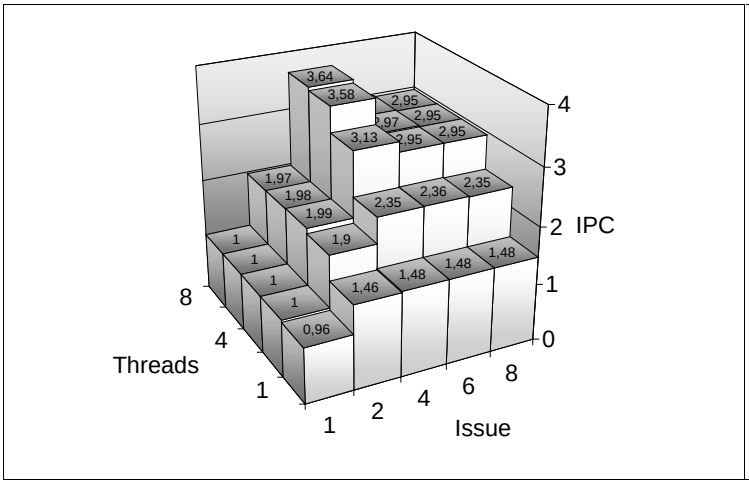
Man erkennt, dass sich bei allen Konfigurationen ein leichter Gewinn ergibt. Der Einbruch im Bereich hoher Befehlszuordnungsbreite bei den stark mehrfädigen Prozessoren bleibt jedoch bestehen. Der (8, 8) [34] Prozessor erreicht nur etwa 78% der IPC des (8, 4) Prozessors.

8.2.2 Behinderung der Ausführung durch mehrere Kontrollfäden

Wie wir im vorigen Abschnitt gesehen haben, ergibt sich für eine hohe Befehlszuordnungsbandbreite und hohe Mehrfädigkeit ein Rückgang der ausgeführten Instruktionen pro Takt. Die erste Vermutung, die hier untersucht wird, geht davon aus, dass einzelne Kontrollfäden große Teile der Ressourcen wie zum Beispiel die Befehlsumordnungspuffer belegen und somit verhindern, dass Instruktionen anderer Kontrollfäden berücksichtigt werden.

Ein erster Versuch, die Überlastung des Prozessors zu vermindern, bestand darin, die Befehlsladestufe zu verkleinern. Es wurden hierzu statt acht Befehlsladeeinheiten lediglich zwei verwendet. Der Gedanke hinter dieser Konfiguration ist, dass hierbei weniger Instruktionen in den Prozessor gelangen und somit die Ressourcen in geringerem Maße blockiert werden können.

[34] Zur Vereinfachung wird im weiteren die Thread/Dispatch Konfiguration eines Prozessors in der Form (Thread, Dispatch) angegeben.

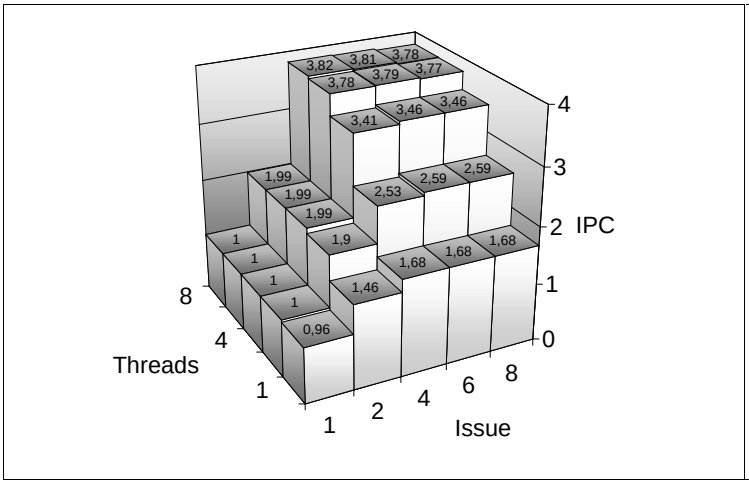


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	2 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-3: Maximalprozessor mit reduzierter Befehlsladestufe

Die Simulationsresultate zeigen keine positiven Auswirkungen[35]. Es ist zu vermuten, dass eine Einschränkung des Instruktionsladens alleine noch keine Verbesserung der Ressourcennutzung nach sich zieht. Lediglich der minimale Gewinn von 0,1 IPC in den hochparallelen Fällen lässt darauf schließen, dass tatsächlich eine Überladung mit Instruktionen besteht.

Um dies zu testen, wurde ein Prozessor simuliert, bei dem die Befehlsrückordnungspuffer auf jeweils 32 Einträge pro Ausführungskontext beschränkt wurden. Dies hat zur Folge, dass jeder Kontrollfaden nur maximal 32 Instruktionen im nicht programmgeordneten Bereich des Prozessors haben kann und somit jedem Kontrollfaden genug Ressourcen verbleiben.



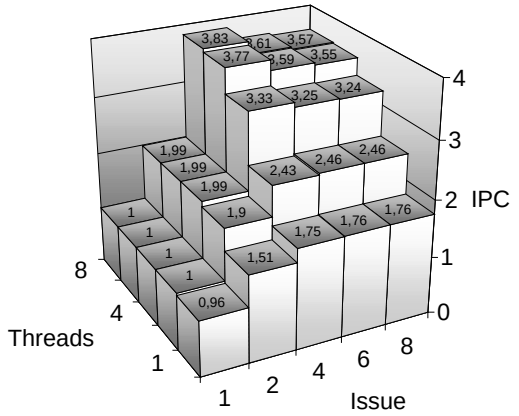
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-4: Maximalprozessor mit eingeschränkten Befehlsrückordnungspuffern

Alle mehrfädigen Konfigurationen haben von dieser Änderung profitiert, lediglich die einfädige Konfiguration muss einen leichten Rückgang von 0,07 IPC hinnehmen. Der Rückgang im (6-8, 6-8) Bereich ist deutlich schwächer ausgeprägt als im vorherigen Prozessor. Im vierfädigen Fall ist kein Rückgang mehr sichtbar.

[35] Der Rückgang des IPC im einfädigen Fall erscheint unerwartet, da ja ein einzelner Kontrollfaden keinen Nutzen aus mehreren Befehlsladeinheiten ziehen kann. Der Grund liegt darin, dass die frei zur Verfügung stehenden Befehlsladeeinheiten für ein Vorabladen sorgen, und somit bei Schleifen mit hoch spekulativen Sprüngen meist schon die Zielinstruktionen im Puffer haben. Dies vermindert die Anzahl der Straftakte bei falsch spekulierten Sprüngen um eins. Wir werden auf dieses Verhalten noch im Kapitel über Sprungvorhersage eingehen.

Eine weitere Möglichkeit zu verhindern, dass einzelne Kontrollfäden unnötig viele Prozessorressourcen belegen, besteht darin, die Anzahl der Anweisungen mit hoher Latenz zu vermindern. Dazu wurden die Befehlsumordnungspuffer der Lade-/Speicher- und der Kontrollfadensteuerungseinheit auf jeweils 32 Einträge verkleinert.

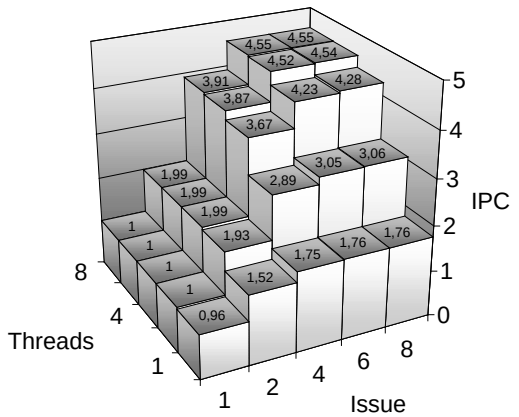


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 32
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-5: Maximalprozessor mit eingeschränkten Befehlsumordnungspuffern der Lade-/Speichereinheit

Im Ergebnis sieht man, dass auch hier eine Verbesserung eingetreten ist, die allerdings geringer ausfällt als in der vorherigen Konfiguration. Es lässt sich dennoch vermuten, dass die Befehlsumordnungspuffer der Lade-/Speichereinheiten für den ungenügenden Durchsatz der hochparallelen Konfigurationen mitverantwortlich sind.

Bei den bisherigen Prozessorkonfigurationen war es so, dass eine Speicherinstruktion, deren Adresse nicht bekannt ist, nicht überholen und nicht überholt werden kann. Dies dient dazu, die Datenabhängigkeiten über den Speicher korrekt zu bearbeiten. Da jedoch zwischen den Kontrollfäden keine Aussage über die Reihenfolge der Speicherzugriffe getroffen ist und Kontrollfäden nur durch Kontrollinstruktionen synchronisiert werden (siehe 6.6.4), ergibt sich keine Änderung im Modell, wenn sich Speicherinstruktionen verschiedener Kontrollfäden gegenseitig überholen. Dies wurde in der nächsten Prozessorkonfiguration getestet.

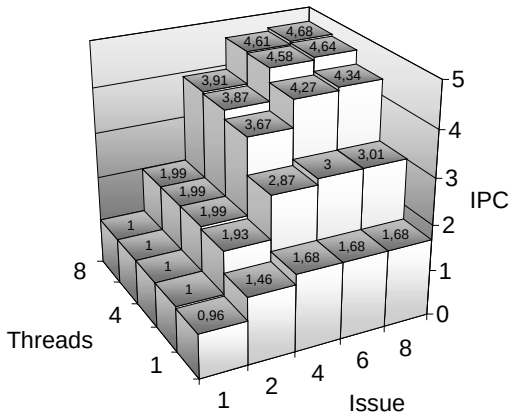


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-6: Maximalprozessor mit abgeschwächter Sequenzialisierung der Befehlsumordnungspuffer der Lade-/Speichereinheit

Anhand der hier erzielten signifikanten Leistungssteigerung lässt sich sagen, dass die unnötig starke Sequenzialisierungsbedingung für Speicherzugriffe den Leistungseinbruch bei den hochparallelen Konfigurationen mitverursacht hat.

Unklar ist noch, ob dies der einzige Faktor war. Dazu wird nun eine Konfiguration getestet, die einen eingeschränkten Befehlsrückordnungspuffer mit der abgeschwächt sequenzialisierenden Lade-/Speichereinheit kombiniert.

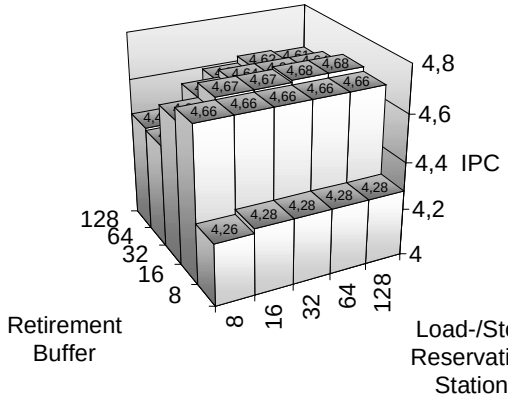


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-7: Maximalprozessor mit abgeschwächter Sequenzialisierung der Befehlsumordnungspuffer der Lade-/Speichereinheit und eingeschränkten Befehlsrückordnungspuffern

Man erkennt, dass für die hochparallelen Konfigurationen noch Leistungsgewinne erzielt werden. Dies lässt darauf schließen, dass auch noch andere Faktoren zu einer gegenseitigen negativen Beeinflussung der Kontrollfäden führen.

Im nächsten Experiment werden verschiedene Kombinationen der Längen der Befehlsumordnungsstufen der Lade-/Speichereinheiten und der Befehlsrückordnungspuffer miteinander getestet.



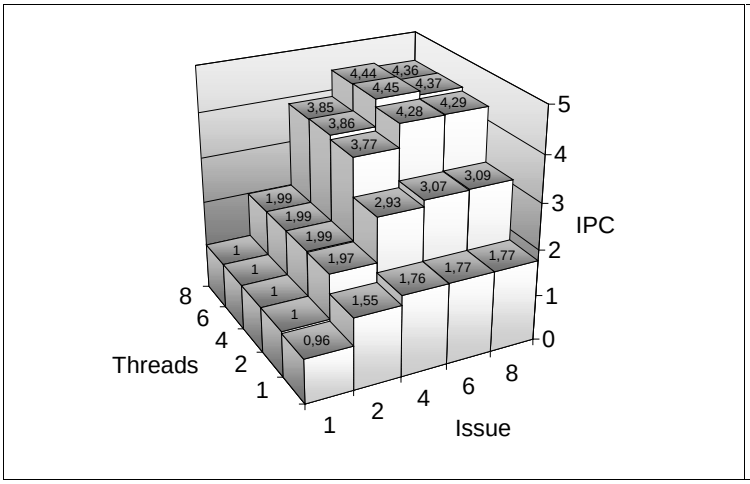
Ausführungskontexte	8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 8-128
Befehlsrückordnungspuffer	je 8-128
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-8: Maximalprozessor mit verschieden großen Befehlsumordnungspuffern der Lade-/Speichereinheiten und Befehlsrückordnungspuffern

Es zeigt sich, dass bei einer Länge der Befehlsrückordnungspuffer jedes Kontrollfadens von nur acht Instruktionen der Prozessor nur ungenügend Instruktionen zur Auswahl hat, um seinen maximalen Durchsatz zu erreichen. Dieser wird bei einer Länge der Befehlsrückordnungspuffer von etwa 16 bis 32 Instruktionen pro Kontrollfaden erreicht. In diesen beiden Konfigurationen spielt die Länge der

Befehlsumordnungspuffer der Lade-/Speichereinheit nur eine geringe Rolle. Dies ändert sich mit zunehmender Länge der Befehlsrückordnungspuffer. Es zeigt sich, dass eine mittlere Länge (also etwa 32-64 Einträge) eine bessere Prozessorauslastung ergibt, als dies bei einer maximalen Länge der Fall ist. Der Grund dafür liegt wie bei den verkürzten Befehlsrückordnungspuffern in der Einschränkung der Instruktionen, die ein einzelner Kontrollfaden im dynamischen Teil des Prozessors haben kann.

Da sehr viel Zeit damit verloren geht, nach einem falsch vorhergesagten Sprung die Pipeline wieder zu füllen, wird im nächsten Experiment ein Prozessor simuliert, dessen Befehlsumordnungspuffer pro Takt doppelt so viele Instruktionen aufnehmen wie abgeben können. Dies hat den Effekt, dass sie nach einem Leeren und erneutem Aufstarten der Pipeline, wie es bei einem falsch spekulierten Sprung der Fall ist, schneller wieder mit Instruktionen befüllt sind.



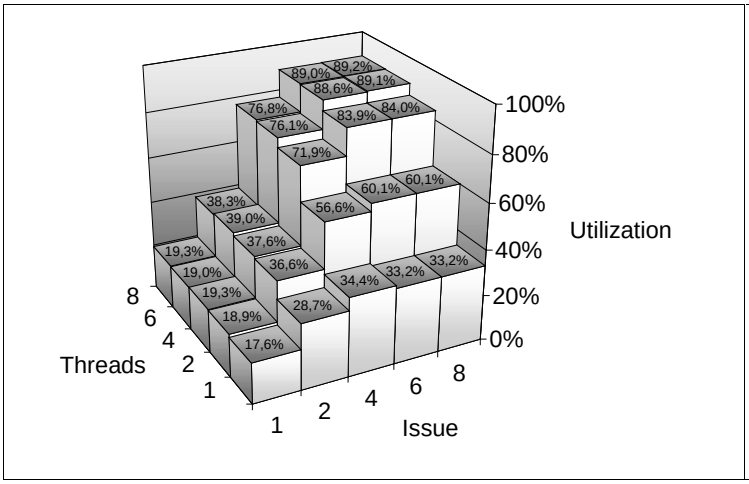
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer (doppelte Aufnahme)	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-9: Maximalprozessor mit verdoppelter Aufnahmefähigkeit der Befehlsumordnungspuffern

Es zeigt sich eine Leistungssteigerung in den Fällen mit einem und zwei Kontrollfäden. In den Konfigurationen mit mehr Kontrollfäden zeigt sich ein leichter Rückgang. In diesen Fällen wird der Instruktionsmangel durch Fehlspekulation durch die zusätzlichen Kontrollfäden kompensiert. Negativ wirkt sich aus, dass einzelne Befehlsumordnungspuffer nun wieder von einzelnen Kontrollfäden überbeansprucht werden können und somit für andere Kontrollfäden blockiert sind.

8.2.3 Leistungssteigerung durch zusätzliche Ausführungseinheiten

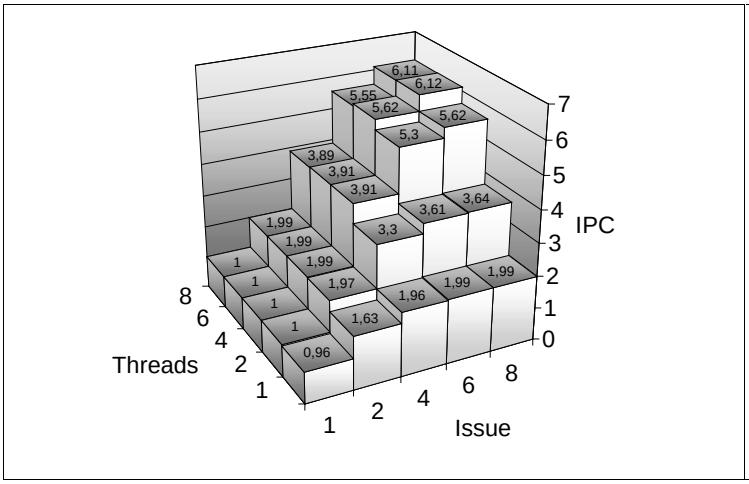
Die Frage wirft sich auf, wie nah die erreichte Leistung an der möglichen liegt. Im nächsten Diagramm wird die Auslastung der lokalen Lade-/Speichereinheit dargestellt.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-10: Auslastung der lokalen Lade-/Speichereinheit

Man erkennt aufgrund der hohen Auslastung der lokalen Lade-/Speichereinheit, dass diese den aktuellen Flaschenhals in den mehrfädigen Konfigurationen darstellt. Da es sich hier um ein Bandbreitenproblem handelt, kann es nur durch Erhöhung der Anzahl der Befehle dieser Einheit, die pro Takt bearbeitet werden können, beseitigt werden. Deshalb wurde im nächsten Experiment eine zweite lokale Lade-/Speichereinheit hinzugefügt (siehe 6.4.4).



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-11: Maximalprozessor mit zwei lokalen Lade-/Speichereinheiten

Diese Erweiterung bringt eine deutliche Leistungssteigerung der mehrfädigen Konfigurationen. Auch der einfädige Fall profitiert von dieser Erweiterung.

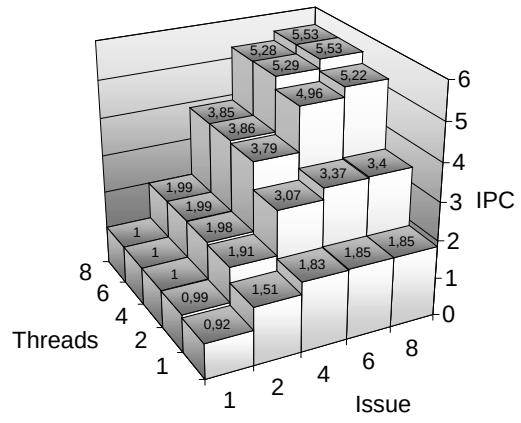
Mit diesem Prozessor schließe ich die Simulation auf eine maximale Leistung hin ab. Es hat sich gezeigt, dass ein mehrfädiger Prozessor bei entsprechender Maximalkonfiguration eine mehr als dreifach so hohe Leistung erzielt, wie ein gleich ausgestatteter einfädiger Prozessor.

Es hat sich weiter gezeigt, dass sich einzelne Kontrollfäden gegenseitig stark stören können, falls sie in den Ausführungseinheiten und Umordnungspuffern nicht unabhängig genug behandelt werden. Als potenzielle Lösung dieses Problems kann auch eine Beschränkung der Anzahl der Instruktionen, die ein Kontrollfaden pro Takt in den Umordnungspuffern haben kann, gesehen werden. Allerdings bringt dies nicht die maximale Leistung des Prozessors zum tragen und beschränkt zudem die Fähigkeit auch mit nur einem Kontrollfaden die volle Ausführungsgeschwindigkeit zu erreichen.

8.3 Verkleinerung des Maximalprozessors

8.3.1 Sprungvorhersage

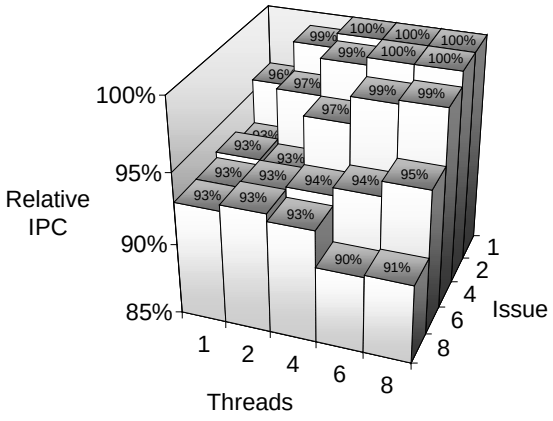
Im folgenden Experiment wird auf eine dynamische Sprungvorhersage verzichtet. Es wird eine einfache statische Vorhersage verwendet, die Sprünge zurück (typisch für Schleifen) als genommen und Sprünge nach vorne als nicht genommen spekuliert.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	statisch

Abbildung 8-12: Maximalprozessor mit statischer Sprungvorhersage

Man erkennt deutliche Leistungseinbrüche, die um so stärker sind, je höher die Zuordnungsbandbreite ist. Um dies deutlich zu machen, ist im nächsten Bild der IPC Wert des Prozessors mit statischer Sprungvorhersage in Relation zum Prozessor mit dynamischer Sprungvorhersage gezeigt.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	statisch/dynamisch

Abbildung 8-13: Maximalprozessor mit statischer Sprungvorhersage in Relation zu dynamischer Sprungvorhersage

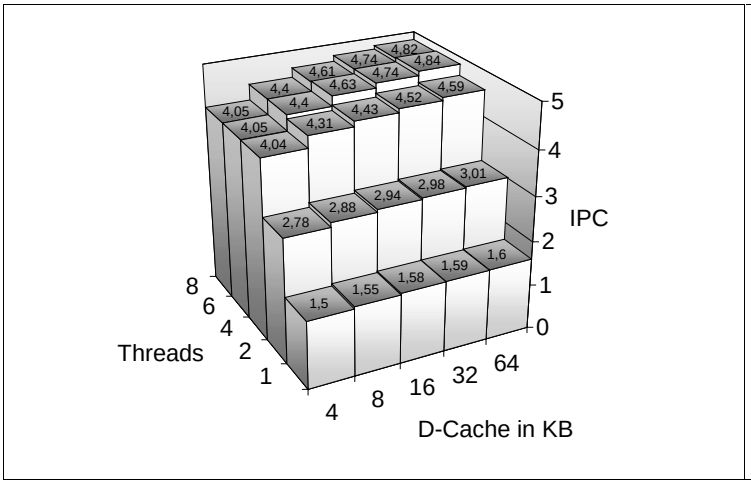
Man sieht, dass die Prozessoren mit mehreren Kontrollfäden bei geringer Zuordnungsbandbreite nur geringe Einbußen durch die statische Sprungvorhersage erleiden, bei hoher Zuordnungsbandbreite aber stärker belastet sind als Prozessoren mit wenigen Kontrollfäden. Dies ist auf die durch die verstärkte Falschspekulation hervorgerufene Steigerung der gegenseitigen Behinderung der Kontrollfäden durch falsch vorhergesagte Anweisungen zurückzuführen.

Im Gegensatz zu früheren Annahmen lässt sich also auch bei einem mehrfädigen Prozessor eine deutliche Leistungssteigerung durch eine verbesserte Sprungvorhersage erreichen. Ob dies auch für einen realistischen Prozessor gilt, dessen limitierende Elemente an anderen Stellen liegen, wird im dazugehörigen Abschnitt untersucht.

8.3.2 Cache-Größe

8.3.2.1 Verminderung der Daten-Cache-Größe

In den folgenden Experimenten werden verschieden große Daten-Caches getestet. Es wird dazu der Maximalprozessor mit dynamischer Sprungvorhersage (wie in „Abbildung 8-11: Maximalprozessor mit zwei lokalen Lade-/Speichereinheiten“) verwendet. Der Instruktions-Cache bleibt vorerst unberücksichtigt.

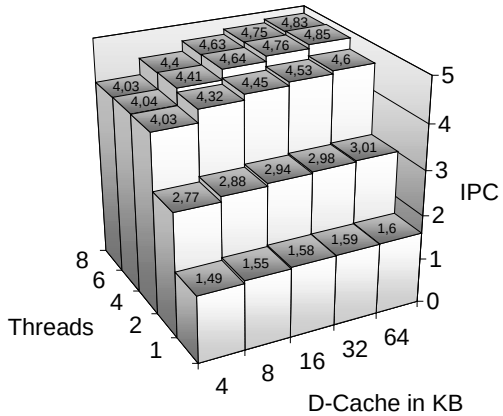


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-14: Maximalprozessor mit reduziertem Daten-Cache

Der Leistungseinbruch ist deutlich zu erkennen. Auch zeigt sich keine Leistungssteigerung beim Übergang von einem vier- zu einem achtfädigen Prozessor mehr. Um zu überprüfen, ob Latenzen den Hauptausschlag für diesen Leistungseinbruch geben, wird im nächsten Durchlauf spekulatives Vorabladen von Daten verwendet (ähnlich zu [12]). Hierzu wird jede Cache-Zeile durch ein 2-Bit Vorhersageregister erweitert, das die vier Zustände „wahrscheinlich vorwärts“, „wahrscheinlich rückwärts“, „sehr wahrscheinlich vorwärts“ und „sehr wahrscheinlich rückwärts“ repräsentieren kann. Diese Vorhersage wird aus dem Zugriffsmuster der bisherigen Zugriffe auf diese Cache-Zeile durchgeführt. Wird auf das erste oder letzte Datenwort der Cache-Zeile zugegriffen, wird je nach Vorhersage die nächste bzw. vorherige Cache-Zeile geladen.

Ein zeilenbasiertes Zugriffsmodell wie in [34] oder [42] wird nicht verwendet, da die Höhe eines typischen Zugriffsblocks in MPEG-2 nicht über sechzehn Zeilen hinausreicht und somit der zusätzliche Aufwand für die Initialisierung des Blockzugriffes nicht lohnt.

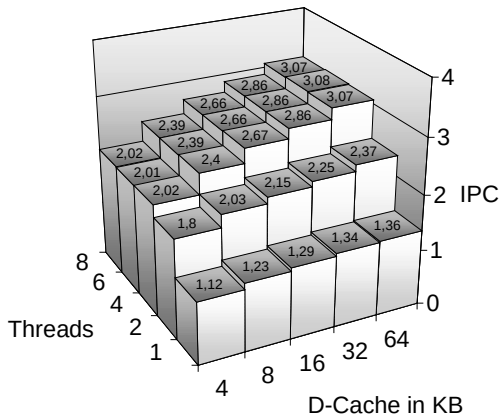


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Prefetch
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-15: Maximalprozessor mit reduziertem Daten-Cache und spekulativem Vorabladen

Auch hier zeigen sich ähnlich schlechte Ergebnisse. Eine Verbesserung der Trefferrate hat also nur einen geringen Gewinn zur Folge.

In der nächsten Simulation wird die Belastung des Speicherbusses noch weiter verstärkt, indem die Sequenz der Speicherzugriffsfolge von ihren bisherigen 6:2:2:2 Takten auf mehr realistische 32:4:4:4 Takte gesetzt[36] wird.

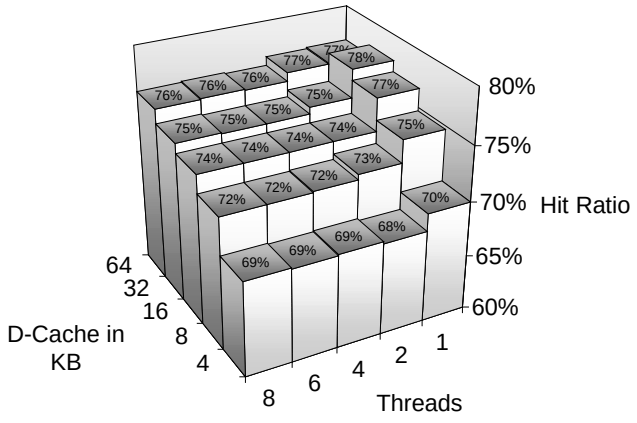


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-16: Maximalprozessor mit reduziertem Daten-Cache und verlängerten Speicherzugriffszeiten

In diesem Fall ist der Rückgang der Prozessorleistung noch deutlicher. Werfen wir nun einen Blick auf die Trefferraten im Datencache.

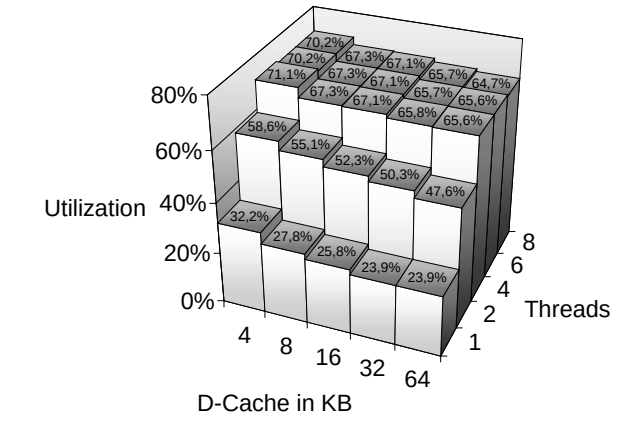
[36] Es handelt sich bei den angegebenen Takten nicht um Bustakte, sondern um Prozessortakte. Da diese bei modernen Prozessoren meist ein Vielfaches des Bustaktes sind, ergeben sich teilweise noch deutlich höhere Werte.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-17: Trefferrate im reduzierten Daten-Cache

Die relativ niedrigen Trefferraten lassen darauf schließen, dass der Leistungseinbruch primär durch die häufigeren Speicherzugriffe verursacht wird[37]. Betrachte man dazu auch noch die Auslastung des Speicherbusses im nächsten Diagramm:

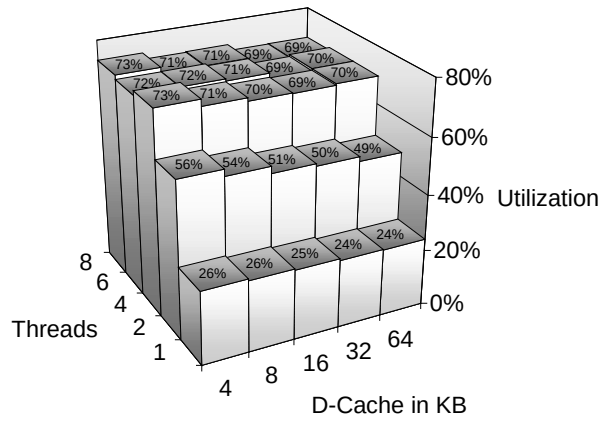


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-18: Auslastung des Hauptspeichers

Der Speicherbus ist nur zu etwa zwei dritteln ausgelastet. Dies deutet darauf hin, dass noch eine weitere Leistungsbeschränkung auftritt, die durch die Zugriffslatenz, nicht aber durch die Gesamtzugriffslast verursacht wird. Es lässt sich auch hier wieder vermuten, dass einzelne Kontrollfäden Einträge in den Befehlsumordnungspuffern durch Instruktionen füllen, die durch Speicherzugriffe indirekt blockiert werden. Betrachten wir hierzu noch die Auslastung der Befehlsumordnungspuffer der Integereinheit.

[37] Interessant ist hier auch die Beobachtung, dass lediglich beim Übergang vom einfädigen zum zweifädigen ein deutlicher Einbruch in der Trefferrate zu verzeichnen ist, nicht aber bei weiteren Steigerungen. Auch ist dieser Trefferrückgang im Fall mit einem 64-KB Daten-Cache nicht mehr zu beobachten. Dies könnte in der Natur des Lastprogramms liegen, da jenes nur zwei verschiedene Programmteile zur Verfügung stellt. Somit beeinflussen sich ein zweiter und dritter Kontrollfaden kaum negativ, da sie auf dem gleichen Speicher arbeiten.

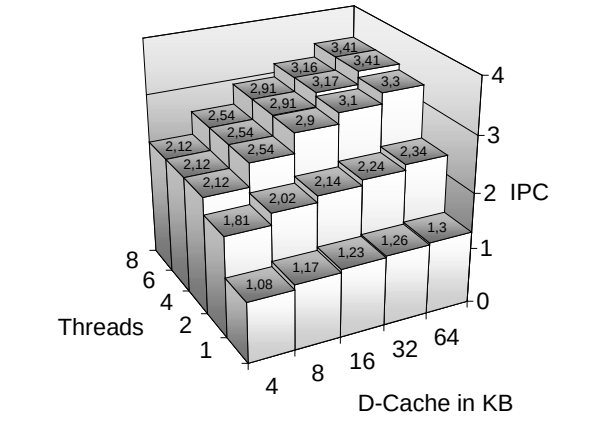


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-19: Auslastung der Befehlsumordnungspuffer der Integereinheiten

Es zeigt sich, dass die Befehlsumordnungspuffer der Integereinheiten im vierfädigen und mehrfädigen Fall, im Durchschnitt zu mehr als 70% gefüllt ist. Dies deutet darauf hin, dass unsere Vermutung richtig ist.

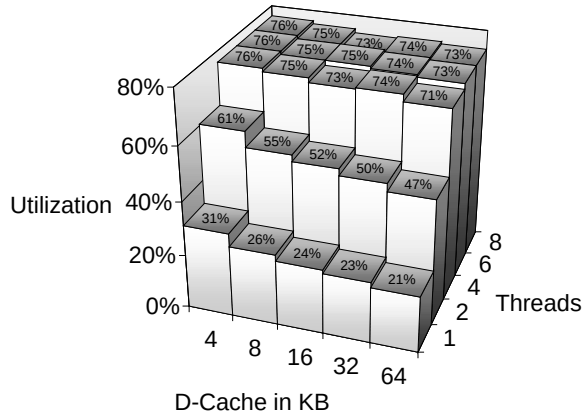
Um dies zu verifizieren, wird wieder (wie in 8.2.2) ein Prozessor mit stark reduzierten Befehlsrückordnungspuffern simuliert. Dies vermindert die Anzahl der Instruktionen, die ein einzelner Kontrollfaden gleichzeitig innerhalb der Umordnungspuffer der Integereinheiten haben kann.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-20: Maximalprozessor mit reduziertem Daten-Cache, verlängerten Speicherzugriffszeiten und reduzierten Befehlsrückordnungspuffern

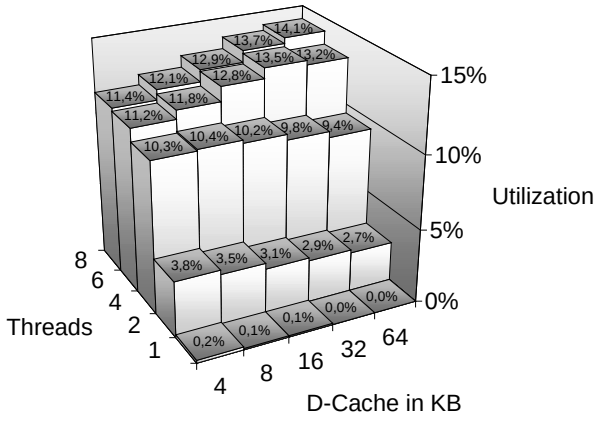
Man erkennt eine signifikante Leistungssteigerung von 0,34 IPC für den mehrfädigen Fall durch diese Reduzierung. Im einfädigen und zweifädigen Fall wird jedoch ein leichter Rückgang beobachtet, da die langen Speicherlatenzen durch die verkürzten Befehlsrückordnungspuffer nun nicht mehr durch die Ausführung ohne Berücksichtigung der Programmreihenfolge überbrückt werden können.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-21: Auslastung des Speicherbusses

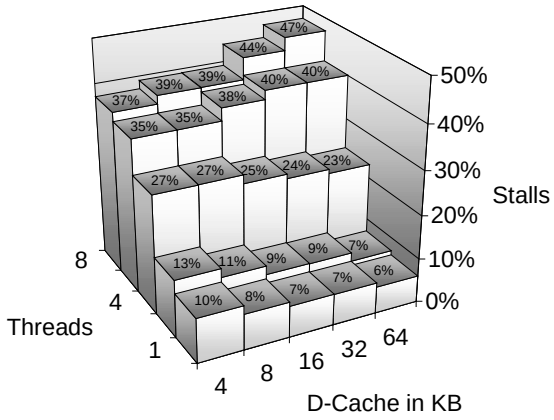
Wie erwartet ist auch die zusätzliche Auslastung des Speicherbusses um 10% gestiegen. Es ergibt sich also eine Steigerung proportional zum IPC Gewinn.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-22: Auslastung der Befehlsumordnungspuffer der Integereinheiten

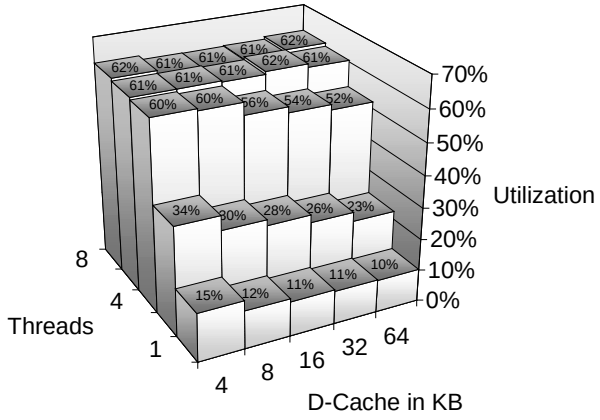
Es ist ebenfalls zu erkennen, dass eine Überfüllung der Befehlsumordnungspuffer der Integerausführungseinheiten nicht mehr stattfindet. Als weiterer potenzieller Kandidat für eine Überfüllung wäre die Cache-Fehlzugriffswarteschlange in der Lade-/Speichereinheit zu betrachten. Ein Hinweis hierzu findet sich in folgendem Diagramm, in dem dargestellt ist, in welchem Anteil aller Takte keine Anweisung aus dem Befehlsumordnungspuffer der Lade-/Speichereinheit an diese weitergereicht werden kann, weil sie über keinen Platz mehr in der Cache-Fehlzugriffswarteschlange verfügt.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-23: Überlastung der Cache-Fehlzugriffswarteschlange

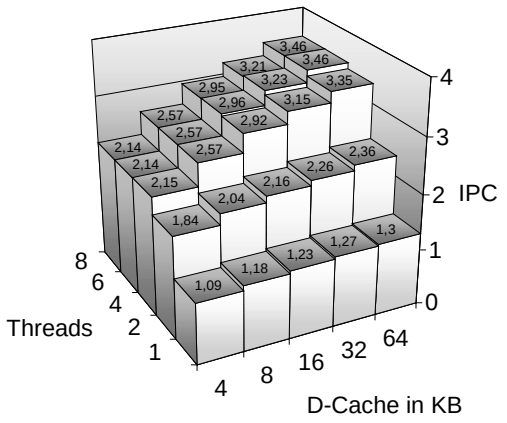
Ein Zeichen dafür ist auch der durchschnittliche Füllstand der Cache-Fehlzugriffswarteschlange:



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-24: Ausnutzung Cache-Fehlzugriffswarteschlange

Um dies weiter zu verifizieren, wird nun ein Prozessor mit einer vergrößerten Cache-Fehlzugriffswarteschlange verwendet. Bisher wurde eine Warteschlange mit acht Einträgen verwendet. Die folgende Konfiguration besitzt 256 Einträge, um jegliche Beschränkungen durch die Länge dieser Warteschlange auszuschließen.

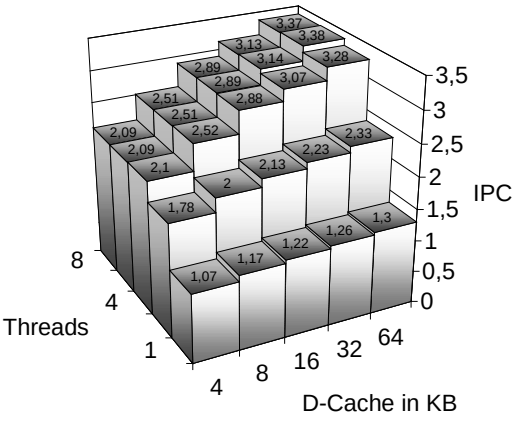


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch
Cache-Fehlzugriffswarteschlange	256

Abbildung 8-25: Maximalprozessor mit reduziertem Daten-Cache und verlängerter Cache-Fehlzugriffswarteschlange

Man erkennt nur eine sehr geringe Leistungssteigerung von bis zu 0,05 IPC. Es lässt sich also folgern, dass die Überlastung der Cache-Fehlzugriffswarteschlange nur zu geringem Teil an der unvollkommenen Auslastung des Hauptspeichers als limitierende Ressource beteiligt ist. Sie ist vermutlich weniger eine Ursache der ungenutzten Zuordnungsmöglichkeiten, als eine Folge der eigentlichen Beschränkung durch die Speicherbandbreite.

Macht nun bei dieser vergrößerten Speicherlatenz das spekulative Vorabladen mehr Sinn als bei der originalen kleinen Latenz ?



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	4
• Strategie	prefetch
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

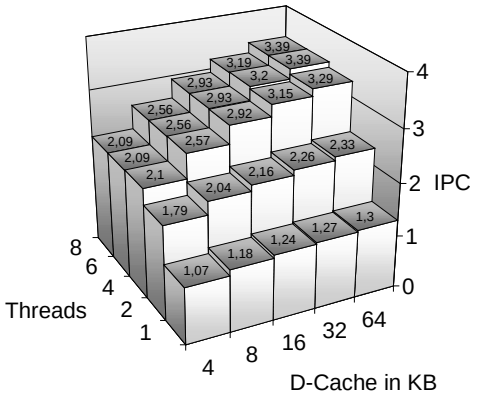
Abbildung 8-26: Maximalprozessor mit reduziertem Daten-Cache , verlängerten Speicherzugriffszeiten, reduzierten Befehlsrückordnungspuffern und spekulativem Vorabladen

Im Gegensatz zu der erwarteten Leistungssteigerung zeigt sich eine Abnahme der IPC um bis zu 0,04. Dies ist dadurch zu erklären, dass zwar die Trefferrate im Cache steigt, aber durch die zahlreichen spekulativen Bustransaktionen die Busauslastung weiter ansteigt. Da aber der Systembus bereits einen kritischen Flaschenhals darstellt, sinkt die Gesamtsystemleistung ab. Es zeigt sich auch hier wieder, dass sich zu aggressive Spekulationen im mehrfädigen Fall nicht immer auszahlen.

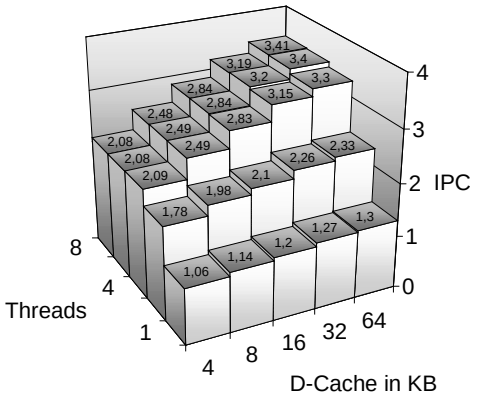
8.3.2.2 Veränderung der Cache-Assoziativität und Verdrängungsstrategie

Neben der Größe und der Füllgeschwindigkeit ist die Assoziativität eines Caches ein Leistungsbestimmender Faktor. Üblicherweise gilt, dass die Trefferrate in einem Cache mit der Assoziativität ansteigt. Dies sollte auch, oder im besonderen bei mehrfädigen Prozessoren gelten, da diese aufgrund der gleichzeitigen Ausführung mehrerer Programmteile eine deutlich geringere Lokalität aufweisen.

Betrachten wir dazu nun veränderte Assoziationstiefen des Datencaches hier für zweifach und achtfach assoziativ.



Achtfach assoziativ



Zweifach Assoziativ

Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	4 - 64 KByte
• Assoziativität	2, 8
• Strategie	Random
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-27: Veränderte Assoziativität des Daten-Caches

Wir erkennen einen leichten Rückgang des IPC trotz Erhöhung der Assoziationstiefe und nur minimale Einbußen bei Verringerung der Assoziationstiefe. Dies lässt vermuten, dass die Verdrängungsstrategie des Daten-Caches nicht optimal ist. In den bisherigen Experimenten wurde mit einer Pseudozufallsfunktion die zu verdrängende Cache-Zeile ermittelt.

Vergleichen wir nun die Trefferraten im Cache zwischen zweifach und achtfach assoziativ.

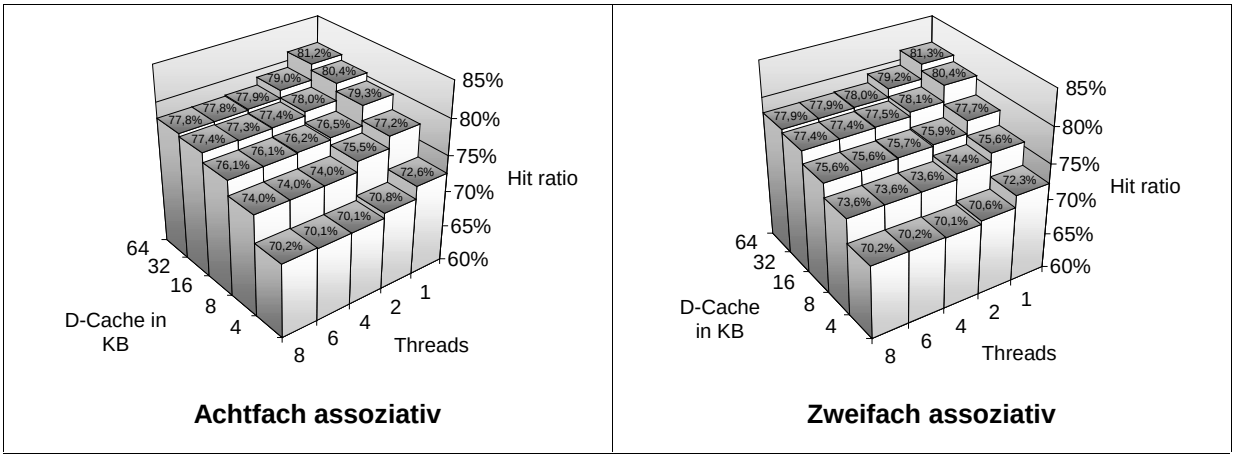


Abbildung 8-28: Trefferraten im Daten-Cache bei veränderter Assoziativität

Man erkennt für den einfädigen Prozessor eine deutlich höhere Trefferrate im achtfach assoziativen Cache. Um so größer allerdings der Cache wird, um so schlechter steht der achtfach assoziative dem zweifach assoziativen Cache gegenüber.

Betrachten wir dazu noch die Auslastung des Speicherbusses:

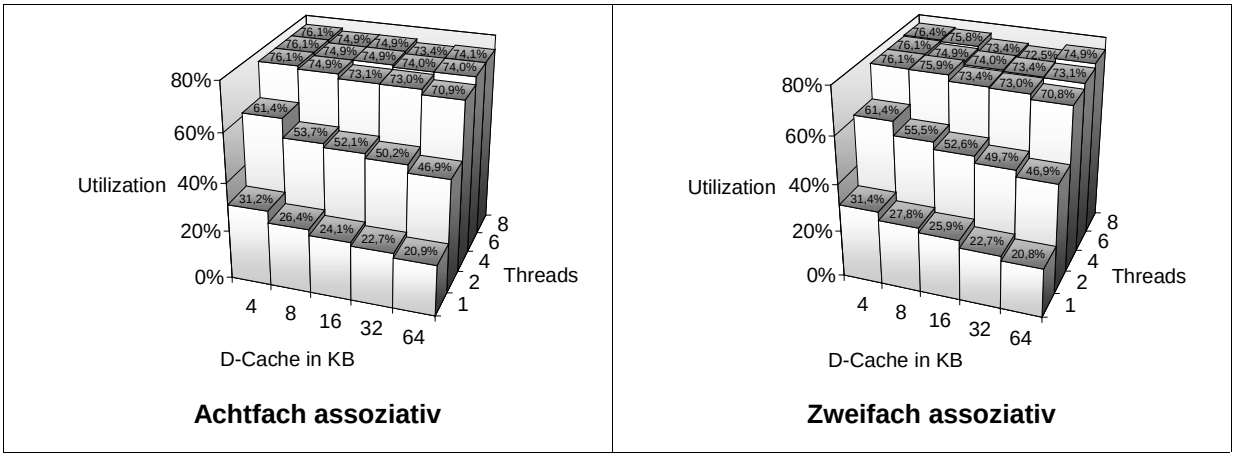
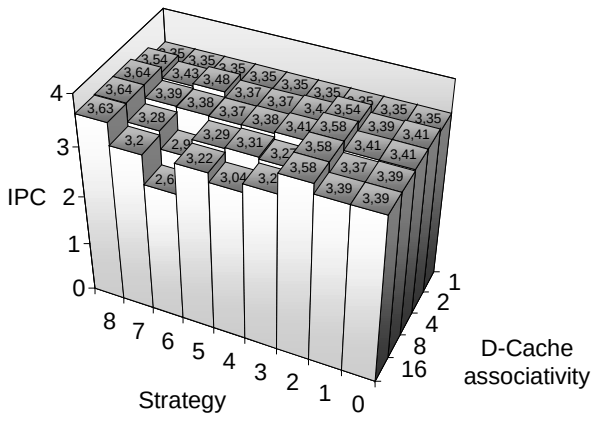


Abbildung 8-29: Speicherbusauslastung bei veränderter Assoziativität des Daten-Caches

Es zeigen sich keine deutlichen Unterschiede zwischen den Konfigurationen. Lediglich in einigen wenigen mehrfädigen Konfigurationen mit großem Cache ist ein leichter Vorteil der zweifach assoziativen Konfiguration auszumachen. Bei kleinen Caches ist dagegen die achtfach assoziative Variante überlegen.

In der nächsten Simulation wurden nun verschiedene Strategien zur Verdrängung von Cache-Zeilen aus dem Datencache getestet. Hierbei kamen folgende Strategien zum Einsatz:

0	Round Robin	Zur Auswahl der Cache-Zeile wird ein Zähler mit Modulo- Arithmetik verwendet. Da der Zähler für alle Cache-Zeilen gemeinsam benutzt wird, sollte sich diese Strategie ähnlich einer Zufallsauswahl verhalten.
1	Random	Zur Auswahl wird ein Pseudozufallszahlengenerator verwendet
2	LRU	Es wird immer die Cache-Zeile ersetzt, die am längsten nicht mehr benutzt wurde.
3	Instruction Address	Die niederwertigen Bits der Instruktionsadresse werden zur Auswahl der Cache-Zeile verwendet. Diese Strategie soll verhindern, dass einzelne Instruktionen, die mit ihren Datenzugriffen sehr große Speicherbereiche durchlaufen, zu ständigem Zeilenflattern im Cache führen [21].
4	Thread ID	Die niederwertigen Bits der Ausführungskontext-ID werden zur Auswahl verwendet. Diese Strategie soll verhindern, dass einzelne Kontrollfäden große Teile des Daten-Caches belegen.
5	Instruction Address + Thread ID	Die niederwertigen Bits der Instruktionsadresse und der Ausführungskontext ID werden zur Auswahl verwendet.
6	Priority	Es werden nur die Cache-Zeilen bevorzugt ersetzt, die von einem Kontrollfaden mit niedriger Priorität angefordert wurden (falls vorhanden). Dies soll verhindern, dass Kontrollfäden mit niedriger Priorität die Cache-Zeilen von Kontrollfäden mit höherer Priorität ersetzten und somit zu einer Art Prioritätsinversion führen.
7	Priority + Random	Es werden mit einer höheren Wahrscheinlichkeit Cache-Zeilen ersetzt, die von einem Kontrollfaden mit niedriger Priorität angefordert wurden.
8	Priority + LRU	Es werden bevorzugt Cache-Zeilen ersetzt, die von einem Kontrollfaden mit niedrigerer Priorität angefordert wurden. Bei gleicher Priorität entscheidet, welche Cache-Zeile länger nicht mehr benutzt wurde.



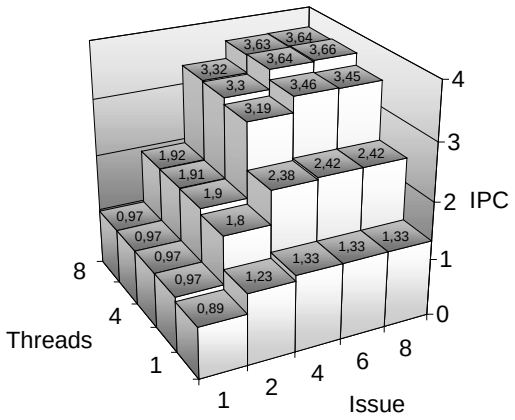
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	1 - 16
• Strategie	variabel
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-30: Maximalprozessor mit reduziertem Daten-Cache und verschiedenen Verdrängungsstrategien

Man erkennt, dass bei höheren Assoziativitätsstufen die Ersetzungsstrategie zunehmend Bedeutung erhält. Lediglich Ersetzungsstrategien, die einen LRU Algorithmus verwenden, profitieren von der hohen Assoziativität. Verfahren, die eine Partitionierung des Caches zur Folge haben, wie z.B. Ausführungskontext ID oder prioritätsbasierte Verfahren, schneiden mit zunehmender Assoziativität schlechter ab, da der Cachespeicher, der für einen einzelnen Kontrollfaden oder eine einzelne Instruk-

tion verfügbar ist, zunehmend geringer wird. Dies spricht auch deutlich gegen eine Partitionierung des Daten-Caches für einzelne Kontrollfäden und widerspricht somit den Ergebnissen aus [86][22]. Im Unterschied zu den dort präsentierten Simulationen, in denen in jedem Ausführungskontext eine andere, unabhängige Last verwendet wurde, wird in dieser Arbeit eine gemeinsame Last betrachtet, deren einzelne Kontrollfäden zwischen den Ausführungskontexten wechseln können.

Interessant ist allerdings, dass die Partitionierung nach der Priorität in Kombination mit einem LRU-Verfahren das beste Ergebnis erzielt. Man erkennt weiter, dass der Cache bereits bei vierfacher Assoziativität ein Plateau erreicht und eine weitere Steigerung nur geringe Gewinne zur Folge hat.



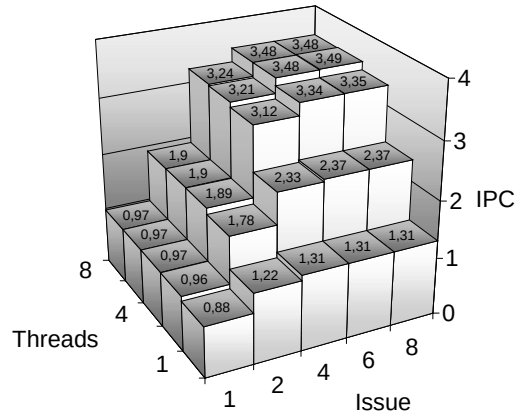
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-31: Maximalprozessor mit reduziertem Daten-Cache und optimierter Verdrängungsstrategie

Die Auswahl einer verbesserten Strategie bringt somit eine IPC Steigerung um etwa 0,18 für die achtfädigen Konfiguration, sowie eine geringe Steigerung um 0,03 für den einfädigen Prozessor.

8.3.2.3 Aufteilung des Caches nach Datenzugriffsmuster

Wie in 7.3.3 besprochen, ergeben sich verschiedene ideale Cache-Zeilenkonfigurationen für unterschiedliche Datenbereiche des Decoders. Dies wird im nächsten Experiment untersucht. Um die minimale Zeilenzahl für den Cache, der die Zugriffe auf die Bilddaten puffert, zu ermitteln, wird folgende Näherung verwendet. Die minimal nötige Zahl an Zeilen (um ein Flattern zu verhindern) ergibt sich als das Produkt aus der Anzahl der verwendeten Kontrollfäden mit der Summe aus Anzahl der zu lesenden Bildzeilen und der Anzahl der zu schreibenden Zeilen: 8 * (2 * 17 + 16) = 400. Da die ideale Zeilengröße als 64 Bytes bestimmt wurde, ergibt sich somit eine minimale Größe dieses Caches zu 25600 Bytes. Somit ergibt sich eine Aufteilung des 64-KByte Daten-Caches in zwei Partitionen zu je 32-KBytes.



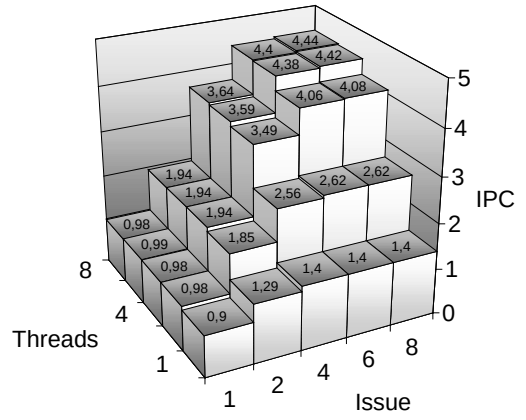
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	2 x 32 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-32: Maximalprozessor mit getrennten Caches für spatiale und temporale Lokalität

Entgegen den Erwartungen ist die Leistung des Prozessors durch die Aufteilung des Caches gesunken. Es scheint, dass die Aufteilung in zwei gleiche Abteilungen zu je 32-KByte dem Zugriffsverhalten nicht gerecht wird. Eine Analyse der Cache-Fehlzugriffe ergibt, dass diese bei allen Befehlen mit Fehlzugriffen durch die Aufteilung des Caches gleichmäßig um etwa 1% gestiegen ist. Nur zwei Befehle erreichen eine etwas geringere Fehlzugriffsrate. Da somit auch die Fehlzugriffe des Parsers angestiegen sind, lohnt es nicht, weitere Experimente mit veränderten Cache-Aufteilungen durchzuführen.

8.3.2.4 Erhöhung des Speicherdurchsatzes

Da sich in den bisherigen Messungen gezeigt hat, dass der Speicherbus bereits sehr stark ausgelastet ist, soll im nächsten Experiment untersucht werden, wie sich eine Steigerung des Durchsatzes bei Erhöhung der Latenz auswirkt. Dazu wurde die Länge einer Speicherzugriffsfolge („Burst“) auf 8*64-Bit erhöht. Hiermit erhöht sich auch gleichzeitig die Länge einer Cache-Zeile auf 64 Bytes.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4:4:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

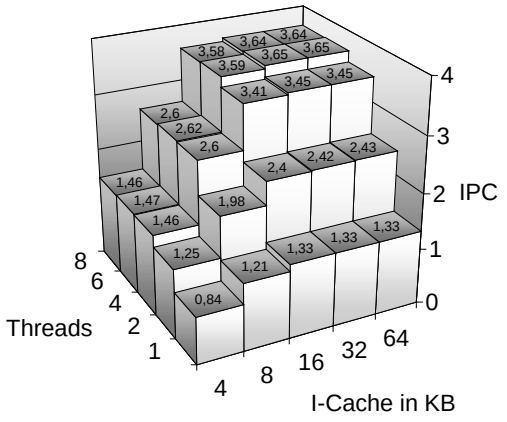
Abbildung 8-33: Maximalprozessor mit verlängerter Speicherzugriffsfolge

Man erkennt eine Steigerung um 22% im 4x4 und 8x8 Fall sowie eine geringere Steigerung in allen weiteren Fällen, da diese die zusätzliche Latenz des Zugriffes nur unvollständig ausgleichen können. Da nicht sicher ist, ob durch diese Maßnahme eine Leistungssteigerung auch bei geringerer Belastung des Speicherbusses erreicht wird, werden die weiteren Simulationen wieder mit einer vierfachen

Speicherzugriffsfolge durchgeführt. Es wird aber am finalen Prozessor noch einmal ein Test mit dieser Konfiguration des Speicherzugriffes durchgeführt

8.3.2.5 Verminderung der Instruktions-Cache-Größe

Im vorherigen Abschnitt wurde der Datencache eingeschränkt. Dies soll nun ebenfalls für den Instruktionscache geschehen. Als Ausgangsprozessor wird wieder ein achtfach superskalarer Prozessor verwendet. Der Datencache wird auf 64 KByte limitiert, die Assoziativität auf vierfach. Es wird ein Speicherzugriffsmuster von 32:4:4:4 verwendet.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	4 – 64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-34: Maximalprozessor mit reduziertem Daten-Cache und verschiedenen Instruktions-Cache Konfigurationen

Da das gesamte Code-Volumen des Dekoders nur 24 KByte beträgt, ergibt sich erwartungsgemäß kein Unterschied mehr beim Übergang von einem 32 auf einen 64 KByte Befehls-Cache.

Die interessanteste Beobachtung bei diesen Konfigurationen ist, dass die mehrfädigen Varianten stärker unter einem limitierten Befehls-Cache leiden als die einfädige. Dies lässt sich einfach daraus erschließen, dass im mehrfädigen Fall mehr Teile des Programms gleichzeitig durchlaufen werden, und somit eine geringere Lokalität vorhanden ist.

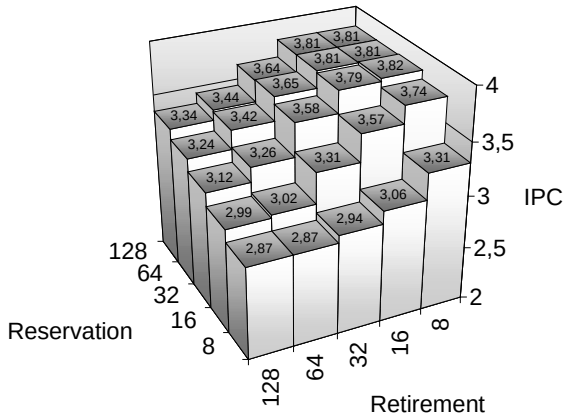
Aufgrund der geringen Code-Größe sind weitergehende Aussagen über Befehls-Cache-Konfigurationen mit diesem Lastprogramm leider nicht möglich.

8.3.3 Befehlspipeline

Nach der Cache- und Speicherkonfiguration sollen nun interne Elemente des Prozessors untersucht werden. Hierbei wird zuerst die Auswirkungen der Größen der Befehlsrückordnungs- und Befehlsumordnungspuffer untersucht.

8.3.3.1 Befehlsumordnungs- und Rückordnungspuffer

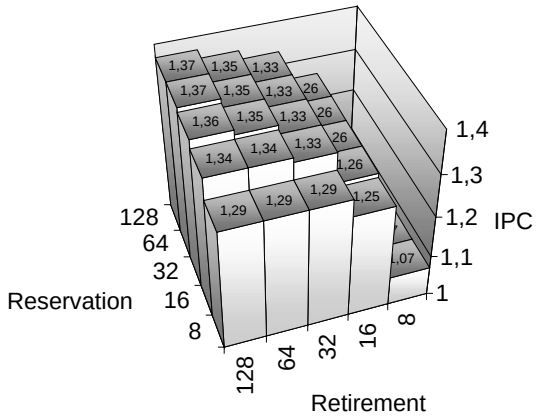
Da sich in den vorherigen Experimenten gezeigt hat, dass sich die Auswirkungen von Befehlsrückordnungspufferlänge und Befehlsumordnungspufferlänge gegenseitig beeinflussen, werden diese Größen nun gemeinsam untersucht.



Ausführungskontexte	8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 8 - 128
• Lade-/Speichereinheiten	je 8 - 128
Befehlsrückordnungspuffer	je 8 - 128
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-35: Verschiedene Konfigurationen der Befehlsrückordnungs- und Befehlsumordnungspuffern

Wie schon in einigen vorhergehenden Experimenten gesehen wurde, hat ein großer Befehlsrückordnungspuffer eine eher negative Auswirkung auf die Gesamtleistung des Prozessors. Dies ist darin begründet, dass sich mehrere Kontrollfäden bei ihrer Ausführung gegenseitig blockieren können, falls sie zu viele datenabhängige Anweisungen zugleich in die Befehlsumordnungspuffer einfügen können. Im Vergleich dazu betrachten wir noch den einfädigen Fall.



Ausführungskontexte	1
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 8 - 128
• Lade-/Speichereinheiten	je 8 - 128
Befehlsrückordnungspuffer	je 8 - 128
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

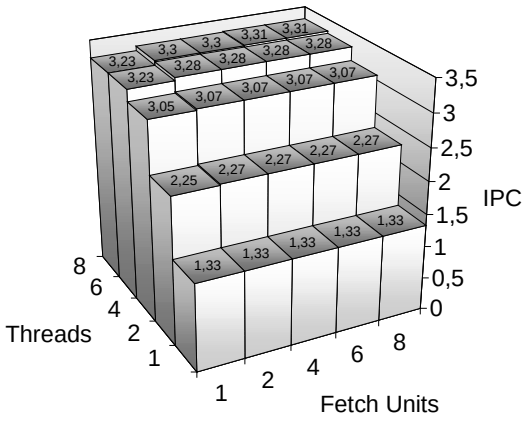
Abbildung 8-36: Verschiedene Konfigurationen der Befehlsrückordnungs- und Befehlsumordnungspuffern

Es zeigt sich hier wie erwartet ein umgekehrtes Verhältnis der Länge der Befehlsrückordnungspuffer und des IPC. Da sich im einfädigen Fall Kontrollfäden nicht blockieren können, profitiert der ausgeführte Kontrollfaden direkt von der zusätzlichen Anzahl an Instruktionen im Bereich der dynamischen Instruktionsauswahl.

Als möglicher Kompromiss zwischen einfädigem und mehrfädigem Fall bietet sich eine Kombination von 16 Einträgen in den Befehlsumordnungs- und 32 Einträgen in den Befehlsrückordnungspuffern. Auch eine Kombination aus 32 Einträgen in den Befehlsumordnungs- und 16 Einträgen in den Befehlsrückordnungspuffern wäre denkbar, da der Gewinn bei mehrfädiger Ausführung deutlich höher als der Verlust bei einfädiger Ausführung wiegt.

8.3.3.2 Befehlslade- und -Dekodiereinheit

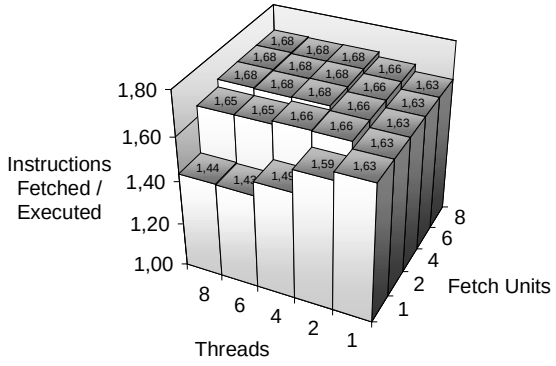
Eine Ursache für dieses Verhalten könnte in der hohen Anzahl an Instruktionen liegen, die pro Takt von der Befehlsladestufe geliefert werden. Im nächsten Experiment wird die Anzahl der Befehlsladeeinheiten in Relation zur Anzahl der Ausführungskontexte betrachtet, wobei eine realistische Größe von Befehlsrückordnungs- und Befehlsumordnungspuffern angenommen wird.



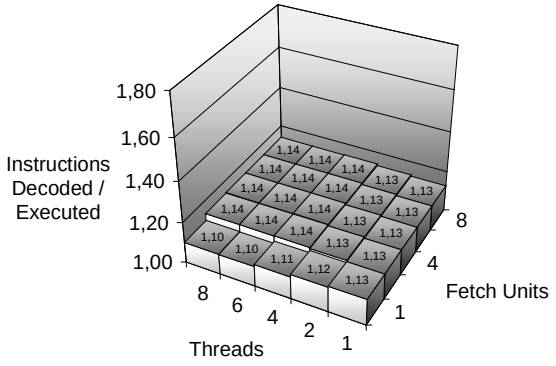
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	1 - 8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 16
• Lade-/Speichereinheiten	je 16
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-37: Verschiedene Anzahl der Befehlsladeeinheiten

Man erkennt, dass die Anzahl der Befehlsladeeinheiten bei der gewählten Konfiguration keine nennenswerte Auswirkung auf den IPC besitzt. Dies könnte darin begründet sein, dass die Gesamtzahl der geladenen Instruktionen immer noch höher ist, als die Anzahl der ausgeführten Instruktionen, und somit die Queues immer noch komplett gefüllt werden. Betrachten wir hierzu das Verhältnis von geholten zu ausgeführten Instruktionen, sowie dekodierten zu ausgeführten Instruktionen.



Geladene Instruktionen



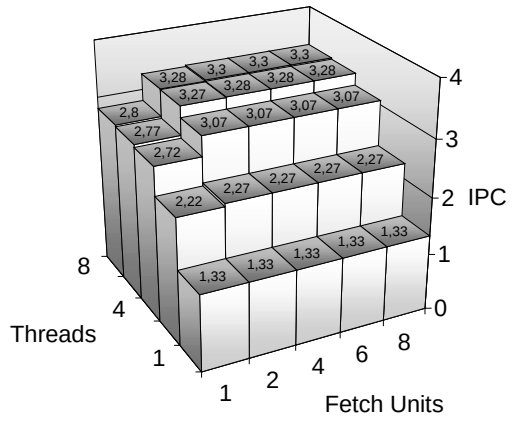
Dekodierte Instruktionen

Abbildung 8-38: Verschiedene Anzahl der Befehlsladeeinheiten

Es zeigt sich, dass zwar die Anzahl der unnötig geholten Anweisungen bei mehr Befehlsladeeinheiten stark zunimmt, diese Zunahme aber bereits vor der Dekodiereinheit verschwindet. Somit kann die Reduzierung der Befehlsladeeinheiten im durchgeführten Maße keine Auswirkungen auf die Ausführung haben.

Um die Anzahl der geholten Instruktionen weiter zu reduzieren, muss die Anzahl der Instruktionen, die durch einen einzelnen Befehlsladezugriff geholt werden, reduziert werden. Im nächsten Experi-

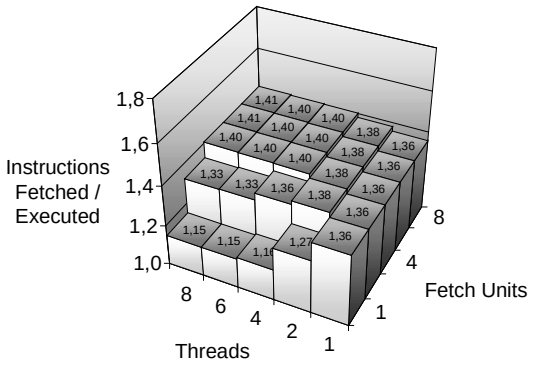
ment wird deshalb mit einer Befehlsladelänge von vier Instruktionen pro Takt und Befehlsladeeinheit getestet.



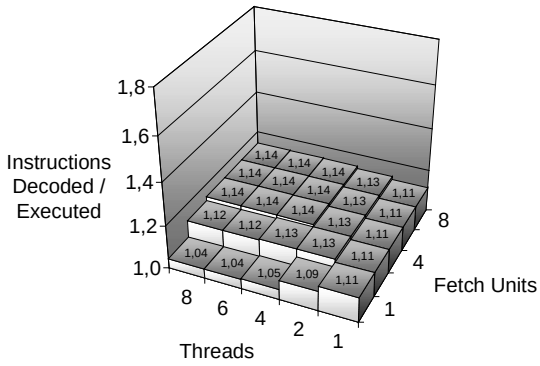
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	1 - 8 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 16
• Lade-/Speichereinheiten	je 16
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-39: Verschiedene Anzahl der Befehlsladeeinheiten, Befehlsladelänge von vier

Lediglich im Fall mit nur einer Befehlsladeeinheit zeigt sich eine Veränderung des IPC, allerdings in die negative Richtung. Dies deutet auf eine Unterfütterung der weiteren Pipeline hin.



Geladene Instruktionen

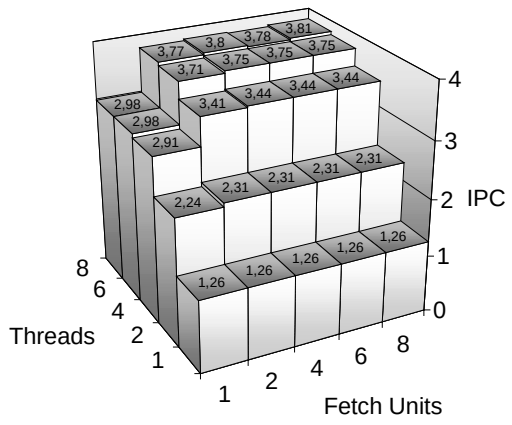


Dekodierte Instruktionen

Abbildung 8-40: Verschiedene Anzahl der Befehlsladeeinheiten

Es zeigt sich, dass die Anzahl der pro Takt geladenen Instruktionen zwar stark zurückgegangen ist, aber sich kaum eine Auswirkung auf die Anzahl der dekodierten Anweisungen zeigt. Die Anzahl der Instruktionen, die in einem Takt geholt werden können, lässt sich somit ohne starke Einschränkung der Ausführungsleistung auf vier reduzieren. Die Anzahl der Befehlsladeeinheiten muss dann aber mindestens zwei betragen.

Wird die mehrfädig freundlichere Kombination aus Befehlsrückordnungs- und Befehlsumordnungspuffern gewählt, ergibt sich folgendes Bild.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	1 - 8 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 32
• Lade-/Speichereinheiten	je 32
Befehlsrückordnungspuffer	je 16
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-41: Verschiedene Anzahl der Befehlsladeeinheiten

Auch hier zeigt sich, dass eine Verkürzung der Befehlsladelänge keine signifikante Leistungsminderung zur Folge hat (siehe Abbildung 8-35: Verschiedene Konfigurationen der Befehlsrückordnungs- und Befehlsumordnungspuffern .)

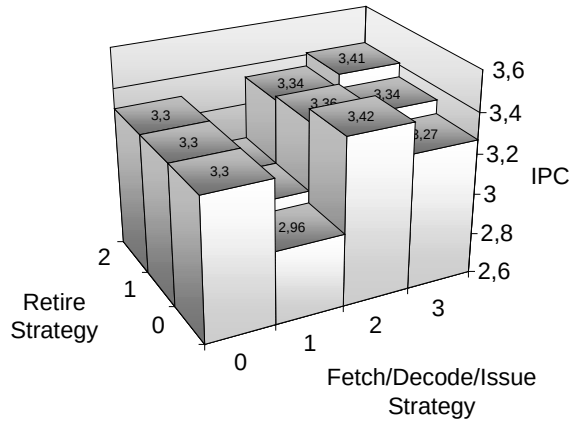
Um das Befehlsladen nur der Kontrollfäden zu limitieren, die unnötig Instruktionen holen, werden verschiedene Strategien der Befehlslade-, -dekodier- , -zuordnungs- und Rückordnungseinheit untersucht [82].

Strategien der Befehlsrückordnungseinheit

0	Round Robin	Die Kontrollfäden werden reihum gleichberechtigt berücksichtigt.
1	Priorität	Kontrollfäden mit hoher Priorität werden bevorzugt
2	Saturiert	Kontrollfäden mit einer hohen Zahl an Instruktionen im Befehlsrückordnungspuffer werden bevorzugt.

Strategien der Befehlslade-, -dekodier- und -zuordnungseinheiten

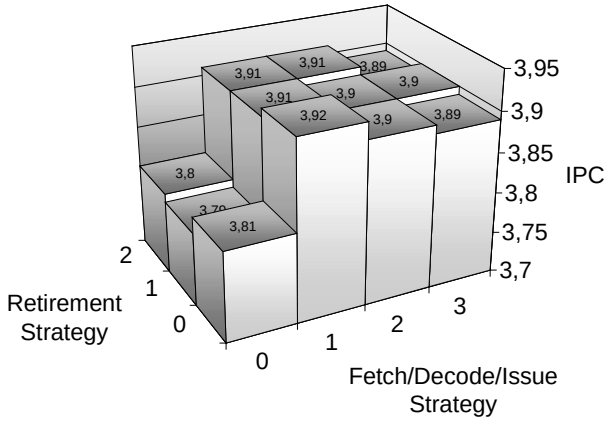
0	Round Robin	Die Kontrollfäden werden reihum gleichberechtigt berücksichtigt
1	Priorität	Kontrollfäden mit hoher Priorität werden bevorzugt
2	Spekulativ	Kontrollfäden mit nicht spekulativer Ausführung werden bevorzugt
3	Saturiert	Kontrollfäden mit einer hohen Zahl an Instruktionen im Befehlsrückordnungspuffer werden benachteiligt



Ausführungskontexte	8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 16
• Lade-/Speichereinheiten	je 16
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	variabel
Befehlslade etc. Strategie	variabel
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-42: Verschiedene Strategien der Befehlskontrollpipeline

Es zeigt sich, dass die Verwendung einer Round-Robin Strategie in der mehrfädig unfreundlichen Variante bereits sehr nah am Ideal liegt. Die prioritätsbasierte Strategie bringt sogar einen Einbruch von fast 0,4 IPC. Interessant scheinen die Kombinationen 2/0 (Befehlsladeeinheit etc / Befehlsrückordnungseinheit) und 3/1. Diese Ergebnisse scheinen unerwartet. Wir betrachten dazu noch die mehrfädig freundliche Variante.



Ausführungskontexte	8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	8 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 32
• Lade-/Speichereinheiten	je 32
Befehlsrückordnungspuffer	je 16
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	variabel
Befehlslade etc. Strategie	variabel
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-43: Verschiedene Strategien der Befehlskontrollpipeline

Hier zeigt sich im Gegensatz zur vorherigen Simulation eine Leistungssteigerung durch die Verwendung der Prioritätsstrategie. Auch die anderen Strategien zeigen ein umgekehrtes Verhältnis in der Leistungssteigerung. Die Erklärung ist relativ einfach. Der mehrfädig freundliche Fall gewinnt durch eine verbesserte Leistung des in Programmordnung arbeitenden Eingangsteil der Pipeline, wogegen die mehrfädig unfreundliche Lösung durch eine verbesserte Leistung eher zur Überlastung der Befehlsumordnungspuffer führt.

Zur Vervollständigung der Betrachtung wird noch der realistische Fall mit nur zwei Befehlslade-, -dekodiereinheiten untersucht.

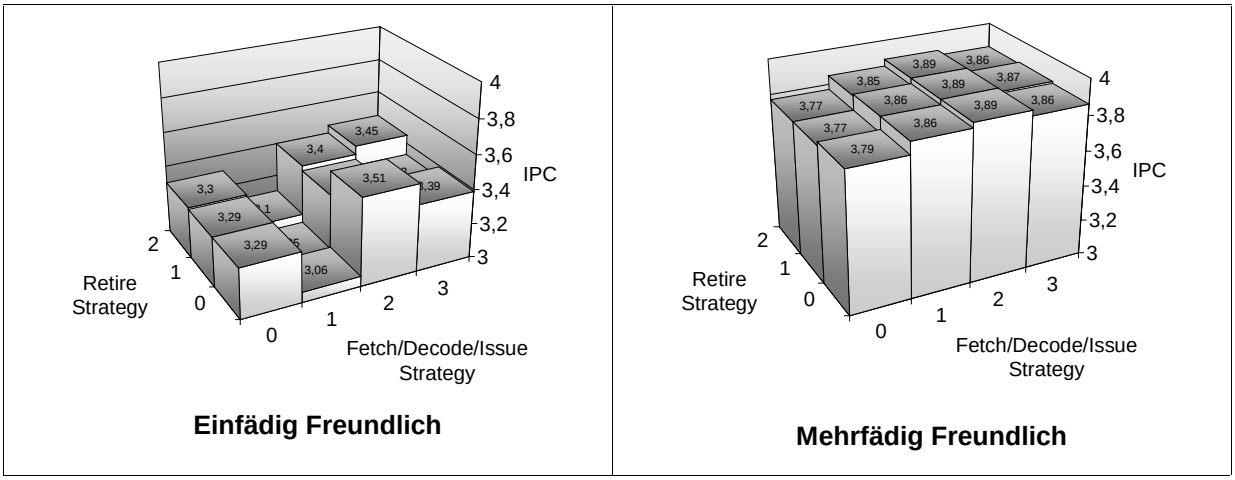
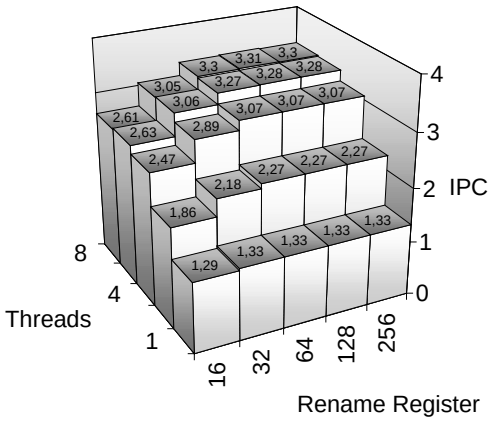


Abbildung 8-44: Verschiedene Strategien der Befehlskontrollpipeline bei nur zwei Befehlsladeeinheiten

Auch hier zeigt sich wieder der starke Einbruch der Prioritätsstrategie im mehrfädig unfreundlichen Fall. Aufgrund der beschränkten Befehlsladeressourcen, macht sich nun aber auch im mehrfädig freundlichen Fall, eine verbesserte Leistung der spekulationsgesteuerten Strategie bemerkbar. Da diese nun in beiden Fällen das beste Ergebnis liefert und auch recht einfach zu implementieren ist, wird sie als bevorzugte Lösung betrachtet.

8.3.3.3 Umbenennungspufferregister

In den bisherigen Simulationen wurde die Anzahl der Umbenennungsregister außer acht gelassen. In den folgenden Simulationen, wird die Anzahl der Umbenennungsregister limitiert, um ein ideales Kosten/Nutzenverhältnis zu erkennen.

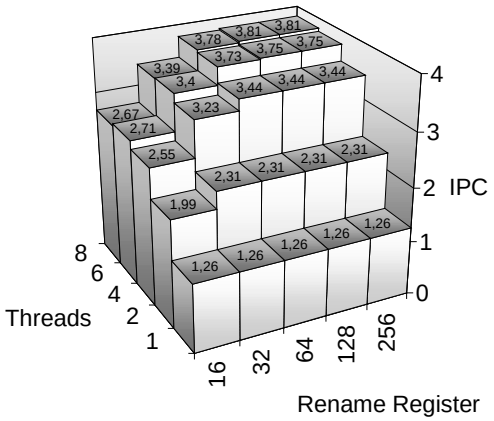


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	2 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 16
• Lade-/Speichereinheiten	je 16
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Spekulativ
Umbenennungsregister	16 - 256
Sprungvorhersage	dynamisch

Abbildung 8-45: Verschiedene Anzahl der Umbenennungspufferregister

Im einfädigen Fall ergibt sich keine Einschränkung durch eine Beschränkung der Umbenennungsregister auf 16. In den mehrfädigen Fällen zeigen sich Einbußen bei weniger als 64 Umbenennungsregistern. Darüber hinaus werden aber auch keine Leistungssteigerungen mehr erzielt.

Zur Vervollständigung wird nun noch der mehrfädig freundliche Fall simuliert.



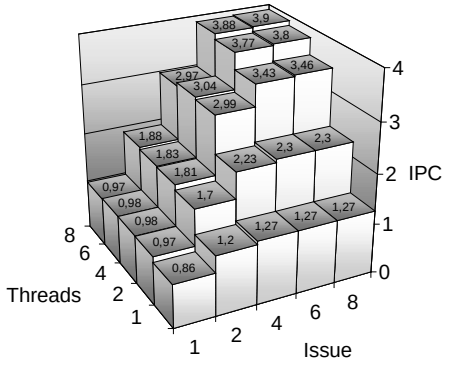
Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	8
Befehlslade-/Dekodiereinheiten	2 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 32
• Lade-/Speichereinheiten	je 32
Befehlsrückordnungspuffer	je 16
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Spekulativ
Umbenennungsregister	16 - 256
Sprungvorhersage	dynamisch

Abbildung 8-46: Verschiedene Anzahl der Umbenennungspufferregister

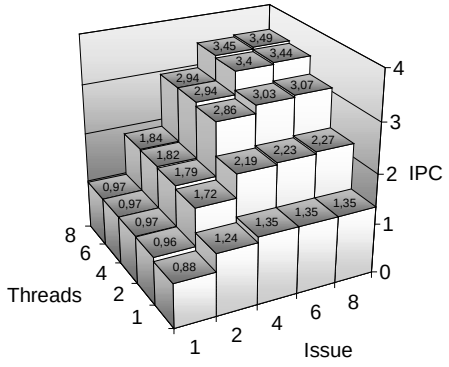
In dieser Simulation zeigen sich die selben Ergebnisse wie im vorherigen Fall (allerdings wie erwartet auf einer höheren IPC Stufe als eben).

In allen weiteren Simulationen wird deshalb die Anzahl der Umbenennungsregister auf 64 beschränkt.

Nach diesen Änderungen ergeben sich jetzt für die beiden Prozessorkonfigurationen folgende Ergebnisse für verschiedene Befehlszuordnungsbreite und Kontrollfadenzahl.



Mehrfädig freundlich



Einfädig freundlich

Abbildung 8-47: Realistische Konfigurationen der Kontroll-Pipeline

8.3.4 Ausführungseinheiten

Aufgrund der geringen Auslastung der einfachen Integerausführungseinheiten kann auf eine der vier Einheiten verzichtet werden. Somit werden nur noch drei Integereinheiten verwendet.

Da die Ausführungseinheiten nicht vollständig ausgelastet sind, können sie sich Ergebnisbusse teilen. Hierzu wurden folgende Paare gebildet. Die Lade-/Speichereinheit für den externen Speicher teilt sich mit einer Integereinheit den Bus. Auch die Einheit für komplexe Integeroperationen und die Kontrollfadensteuerungseinheit teilen sich mit jeweils einer Integereinheit einen Bus. Da die Lade-/Speichereinheit für den lokalen Speicher eine sehr kritische Komponente darstellt, behält sie ihren eigenen Bus.

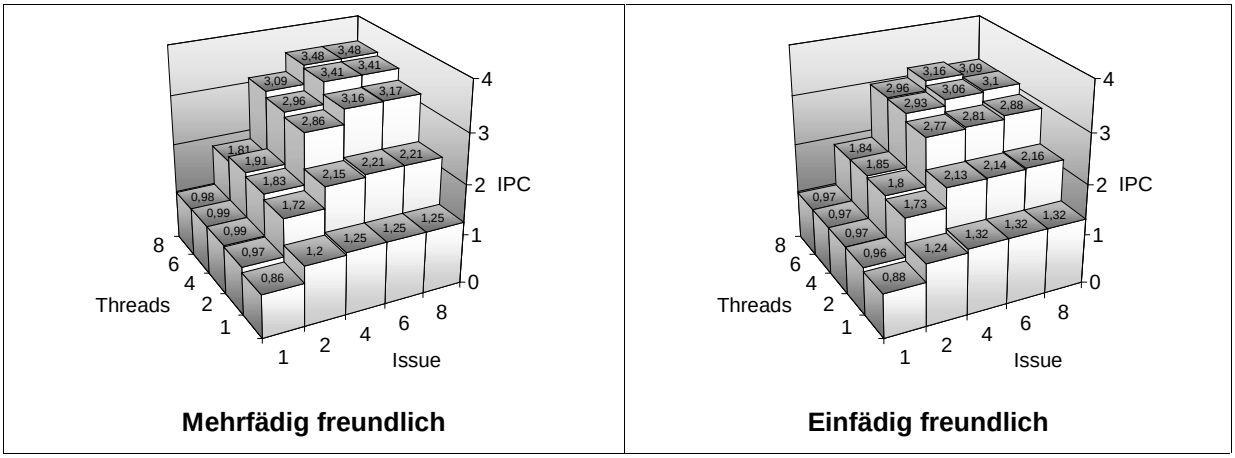


Abbildung 8-48: Realistische Konfigurationen der Ausführungseinheiten

Man sieht einen deutlichen Rückgang des IPC in den mehrfädigen Fällen und nur einen leichten Rückgang im einfädigen Fall. Dies war zu erwarten, da die Ausführungseinheiten im einfädigen Fall aufgrund des niedrigen IPC nur sehr gering ausgenutzt werden.

Ob die Verwendung einer verlängerten Speicherzugriffsfolge auch in dieser eingeschränkten Konfiguration zur Verbesserung der Leistung beiträgt, wird in der nächsten Simulation untersucht.

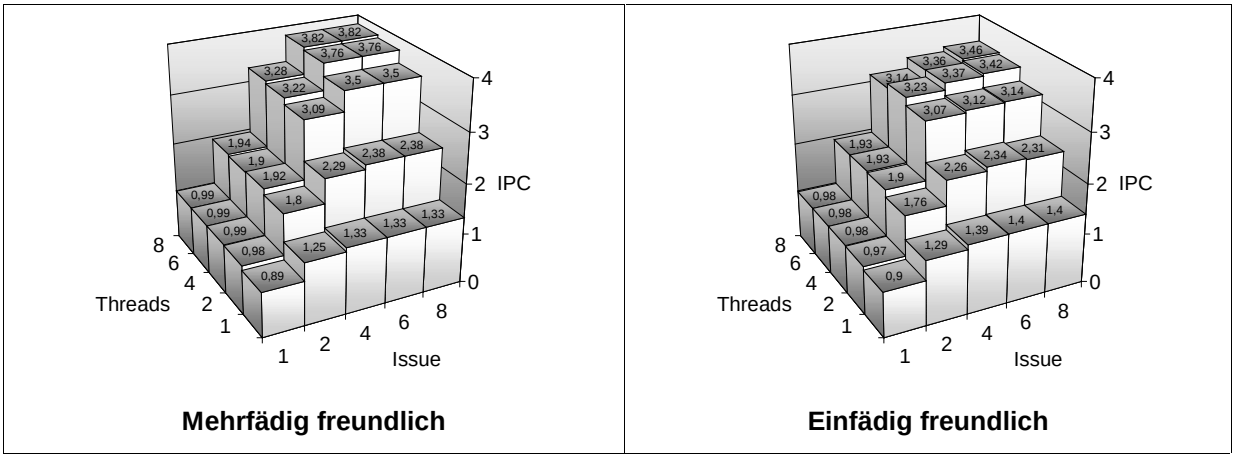


Abbildung 8-49: Realistische Konfigurationen der Ausführungseinheiten mit verlängerter Speicherzugriffsfolge

Man sieht, dass auch in diesem eingeschränkten Prozessor die Verwendung einer verlängerten Speicherzugriffsfolge zu einer deutlichen Leistungssteigerung führt. Signifikant ist auch, dass sich der einfädige Fall in diesen beiden Konfigurationen nur noch sehr gering unterscheidet. Dies deutet darauf hin, dass sich ein verlängerter Befehlsrückordnungspuffer wie erwartet besonders bei Befehlen mit sehr langer Latenz positiv auswirkt. Wird die Latenz der Lade-/Speicherbefehle gesenkt, indem der externe Bus besser genutzt wird, führt auch ein geringerer Befehlsrückordnungspuffer zur erwarteten Leistung.

Zum Abschluss wird nun auch noch die zweite Lade-/Speichereinheit für den internen Speicher entfernt.

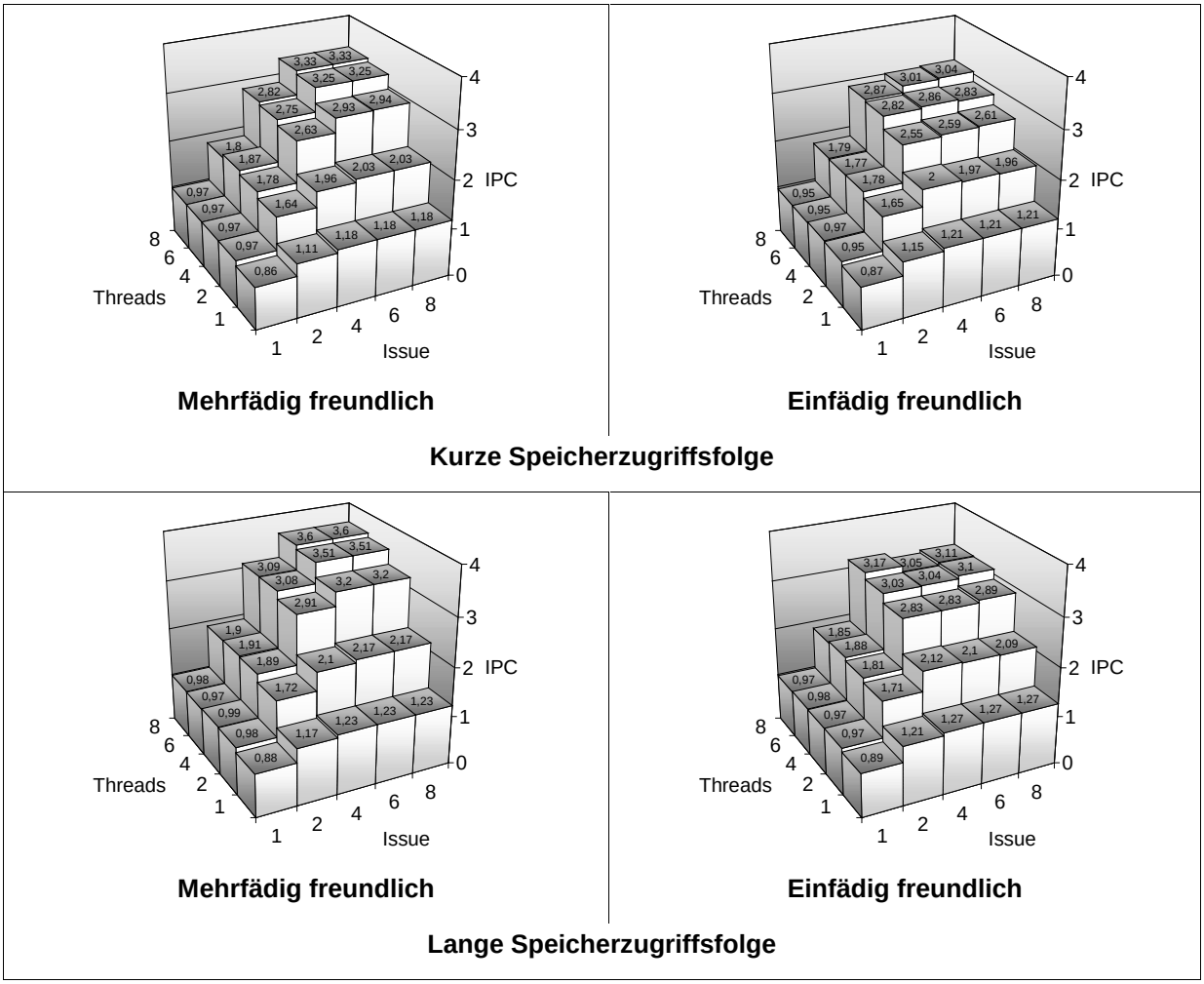


Abbildung 8-50: Realistische Konfigurationen der Ausführungseinheiten mit nur einer lokalen Lade-/Speichereinheit

Wie erwartet liefert die Variante mit verlängerter Speicherzugriffsfolge im mehrfädig freundlichen Fall eine bessere Leistung als die Variante mit kurzer Speicherzugriffsfolge. Fast entgegengesetzt verhält sich die einfädig freundliche Variante. Hier zeigt sich in den Fällen hoher Befehlszuordnungsbreite ein Leistungseinbruch bei der Variante mit verlängerter Speicherzugriffsfolge. Dies ist darauf zurückzuführen, dass mit einer Verlängerung der Speicherzugriffsfolge nicht nur der Durchsatz sondern auch die Latenz erhöht wird. Da diese Prozessorvariante bereits aufgrund der gegenseitigen Blockade der Kontrollfäden unter der Langen Speicherlatenz leidet, wird dieser Faktor durch die erhöhte Latenz natürlich noch verstärkt.

8.3.5 Sprungvorhersage bei realistischen Ressourcen

Als Basis für die folgenden Simulationen wird eine lange Speicherzugriffsfolge verwendet.

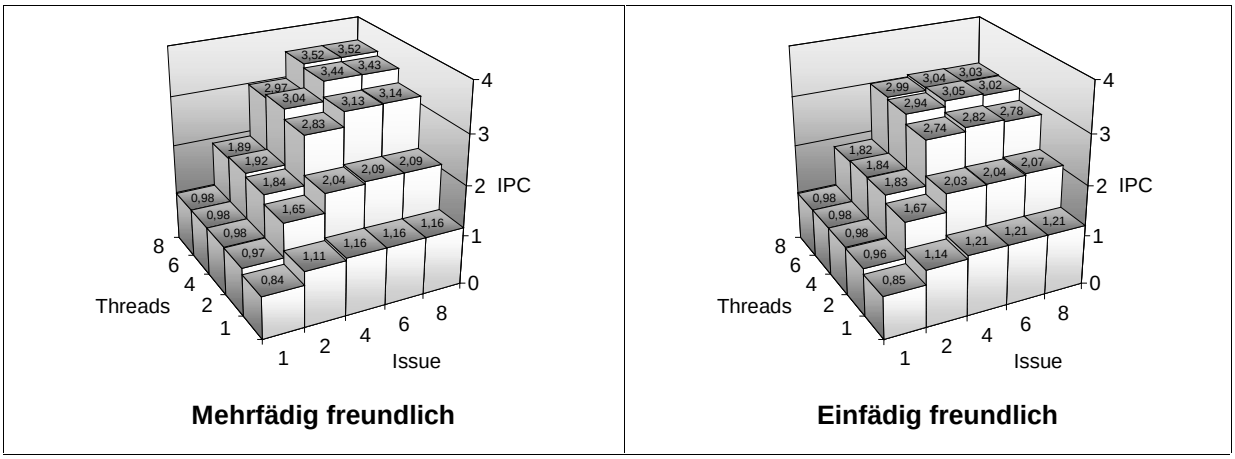


Abbildung 8-51: Statische Sprungspekulation bei realistischer Konfiguration

Die Ergebnisse zeigen wie erwartet eine klare Abnahme des IPC für alle Konfigurationen. Die IPC Differenz zum Prozessor mit dynamischer Sprungvorhersage beträgt etwa 0,06 – 0,11 IPC.

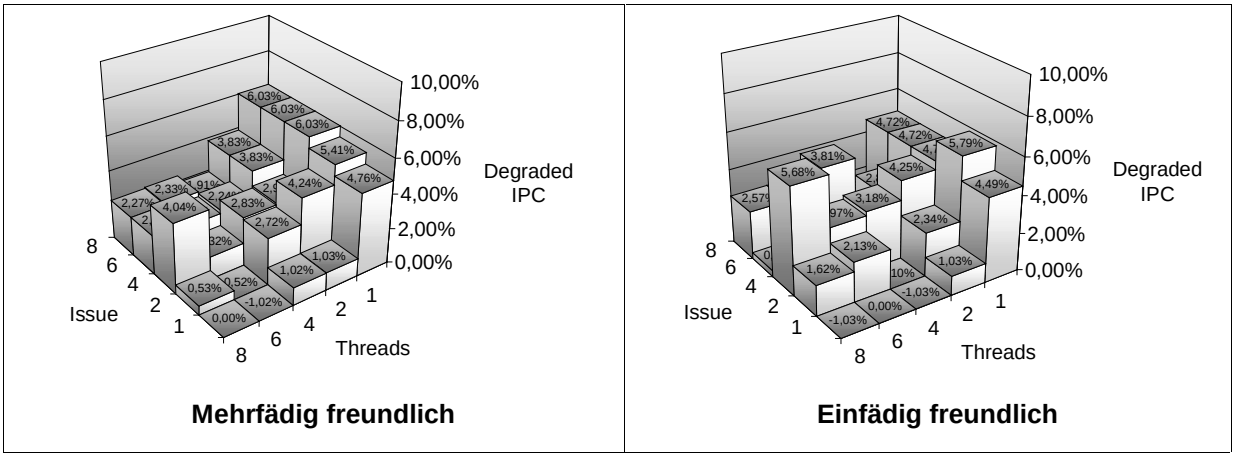


Abbildung 8-52: Relative IPC Abnahme durch statische Sprungspekulation

Bei der relativen Leistungsdifferenz wird deutlich, dass die mehrfädigen Prozessorvarianten weniger stark unter der schlechteren Sprungvorhersage leiden. Allerdings ist die Abnahme bei den meisten Konfigurationen so stark, dass der Verzicht auf eine dynamische Sprungvorhersage nicht empfohlen wird.

Wird auf eine Sprungvorhersage vollständig verzichtet, wobei alle Befehle eines Kontrollfadens nach einem Sprungbefehl in der Dekodiereinheit blockiert werden, bis der Sprung komplett bearbeitet wurde, ergibt sich folgendes Bild:

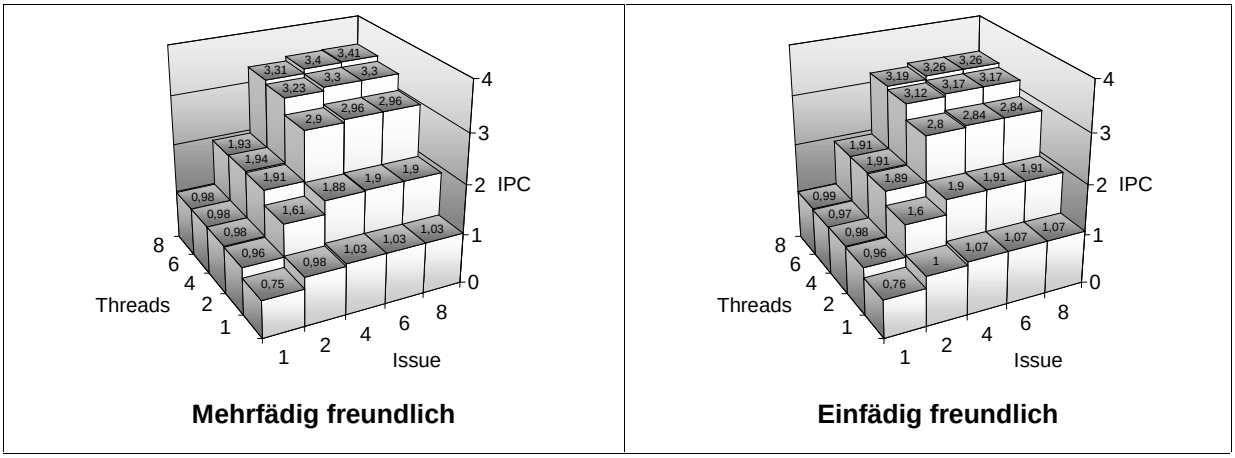


Abbildung 8-53: Realistische Konfiguration ohne Sprungvorhersage

Besonders in den einfädigen Konfigurationen zeigt sich ein deutlicher Leistungsrückgang.

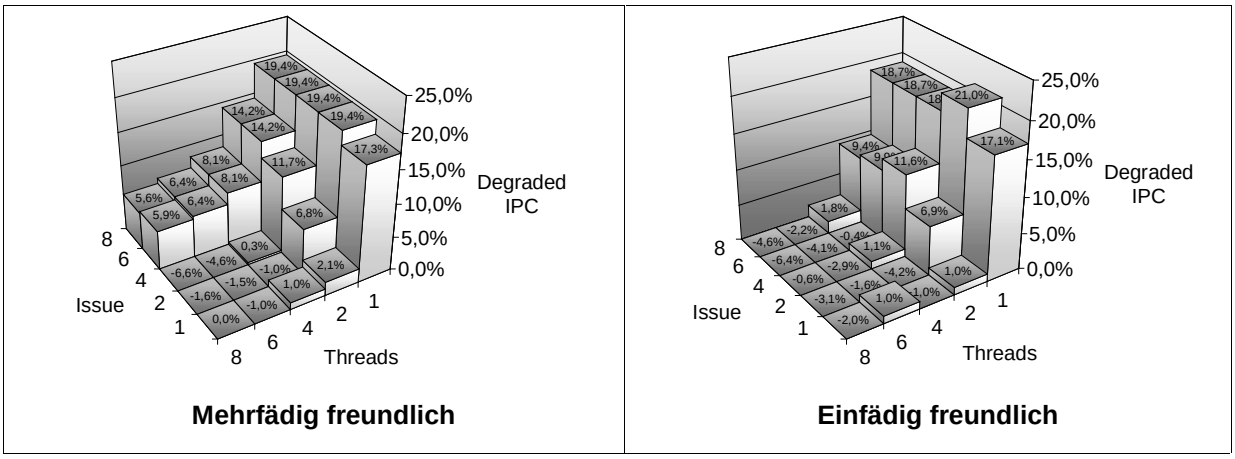


Abbildung 8-54: Relative IPC Abnahme durch Verzicht auf Sprungspekulation

Durch den Vergleich der relativen IPC Abnahme wird deutlich, dass die einfädigen Konfigurationen etwa 20% Leistung einbüßen. Die mehrfädigen Konfigurationen gewinnen allerdings teilweise sogar bis zu 6% durch den Verzicht auf eine Sprungspekulation. Dies ist ein klares Indiz dafür, dass sich bei mehrfädigen Prozessoren spekulative Ausführung störend auf die Prozessorleistung auswirken kann, da die Ressourcen, die durch falsch spekulierte Befehle belegt werden, nicht für die anderen Kontrollfäden zur Verfügung stehen. Dies ist besonders vom Standpunkt des Stromverbrauchs dieser Prozessoren interessant. Da falsch spekulierte Instruktionen unnötig Strom verbrauchen, könnte die Mehrfädigkeit durch eine höhere Leistung bei einem Verzicht auf Sprungspekulation zu einem besseren Verhältnis aus ausgeführten Instruktionen zu verbrauchtem Strom führen.

8.3.6 Verzicht auf lokalen Speicher

Als Basis für die folgenden Simulationen wird wieder eine zweistufige dynamische Sprungvorhersage und eine lange Speicherzugriffsfolge verwendet.

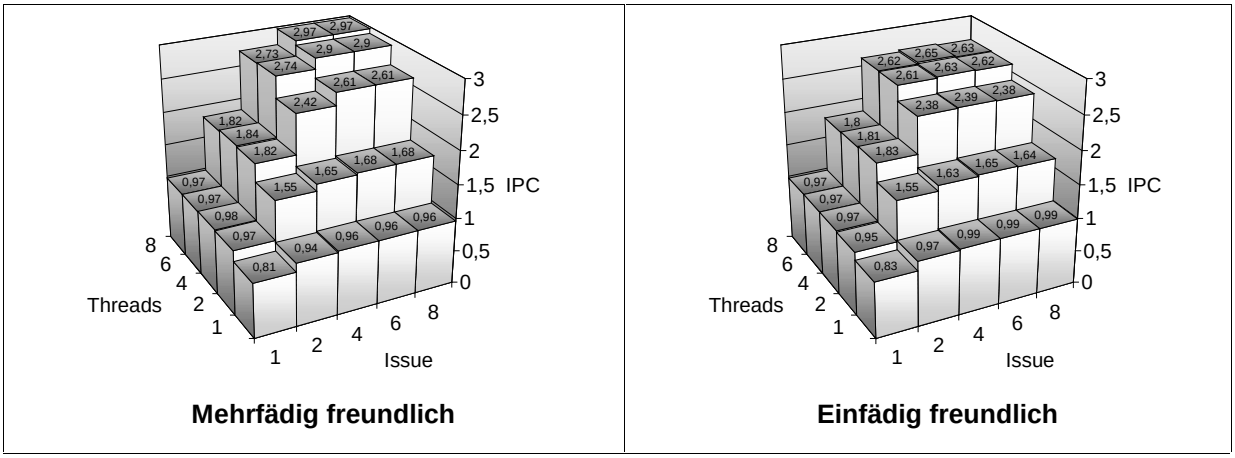


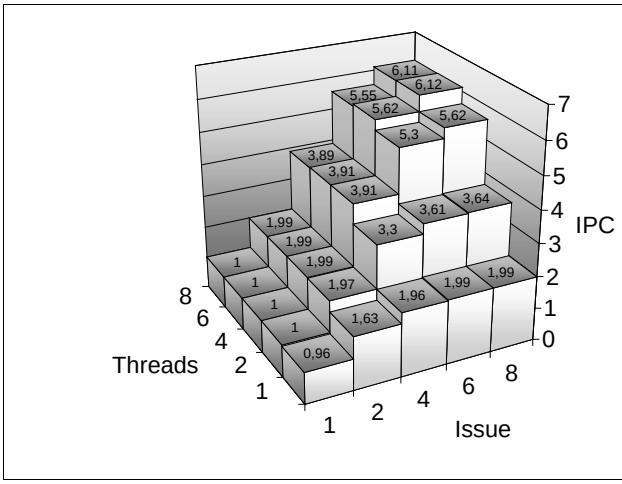
Abbildung 8-55: Realistische Konfiguration ohne lokalen Speicher

Die Ergebnisse zeigen auch hier wie erwartet eine klare Abnahme des IPC für alle Konfigurationen. Der Verlust beträgt bis zu 0,6 Befehle in jedem Takt. Allerdings bleibt der deutliche Gewinn durch die Mehrfädigkeit auch bei diesen Varianten des Prozessors vorhanden. Obwohl nun deutlich weniger Ausführungseinheiten zur Verfügung stehen, kann der mehrfädige Prozessor immer noch bis zu dreimal mehr Instruktionen pro Takt ausführen als der einfädige. Die Mehrfädigkeit lohnt sich also nicht nur bei einer deutlichen Überausstattung mit Ausführungseinheiten, sondern auch bei Minimalkonfigurationen.

8.4 Zusammenfassung

Mehrere Prozessorkonfigurationen wurden getestet, von denen die folgenden drei besondere Beachtung verdienen:

- Maximalprozessor mit idealen Caches
- Maximalprozessor mit realistischen Caches und Speicherbus
- Prozessor mit realistischen Ressourcen

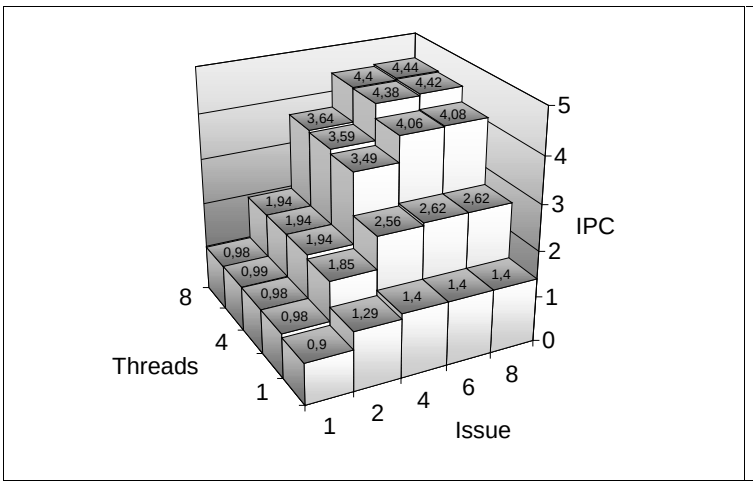


Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 256
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	6:2:2:2
Daten-Cache	
• Größe	Hauptspeichergröße
• Assoziativität	4
• Strategie	keine
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

Abbildung 8-56: Maximalprozessor mit idealen Caches

Diese Konfiguration liefert die höchste IPC Leistung, die innerhalb des Gestaltungsraumes des Simulators möglich ist. Folgende wichtige Konfigurationsentscheidungen wurden auf der Simulation basierend getroffen:

- Die einfachen Integereinheiten teilen sich eine gemeinsamen Befehlsumordnungspuffer. Dies verhindert, dass einzelne Integerausführungseinheiten durch eine vorzeitige Befehlszuordnung in der Befehlszuordnungseinheit überlastet werden.
- Befehle verschiedener Ausführungskontexte können sich in den Befehlsumordnungspuffern der Lade-/Speichereinheiten sowie der Kontrollfadensteuereinheit überholen. Dies verhindert, dass einzelne Kontrollfäden durch abhängige Befehlsfolgen mit langer Latenz zu einer Blockade dieser Einheiten führen.
- Eine zweite Lade-/Speichereinheit für den lokalen Speicher erhöhte den IPC im (8, 8) Fall um 1,5. Dies ist natürlich auch auf die Befehlsmischung in der Simulationslast zurückzuführen, zeigt aber, dass der Maximalprozessor in der Lage ist, eine einzelne Lade-/Speichereinheit für den lokalen Speicher vollständig zu nutzen.
- Eine dynamische Sprungvorhersage führt auch bei einem mehrfädigen Prozessor zu einer Leistungssteigerung. Zwar ist ein mehrfädiger Prozessor in der Lage, die durch eine Sprungfehlvorhersage entstehenden Latenzen zu überbrücken, doch senkt dies den Gesamtdurchsatz des Prozessors, da viele Befehle unnötigerweise ausgeführt werden.
- Acht Kontrollfäden zeigen gegenüber vier Kontrollfäden nur noch einen geringen Leistungsgewinn.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	8 x 8
Befehlsumordnungspuffer	
• Integereinheiten	einmal 256
• Lade-/Speichereinheiten	je 256
Befehlsrückordnungspuffer	je 32
Integereinheiten	4
Lokale Lade-/Speichereinheiten	2
Speicherzugriffsfolge	32:4:4:4:4:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	Hauptspeichergröße
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Round Robin
Umbenennungsregister	2048
Sprungvorhersage	dynamisch

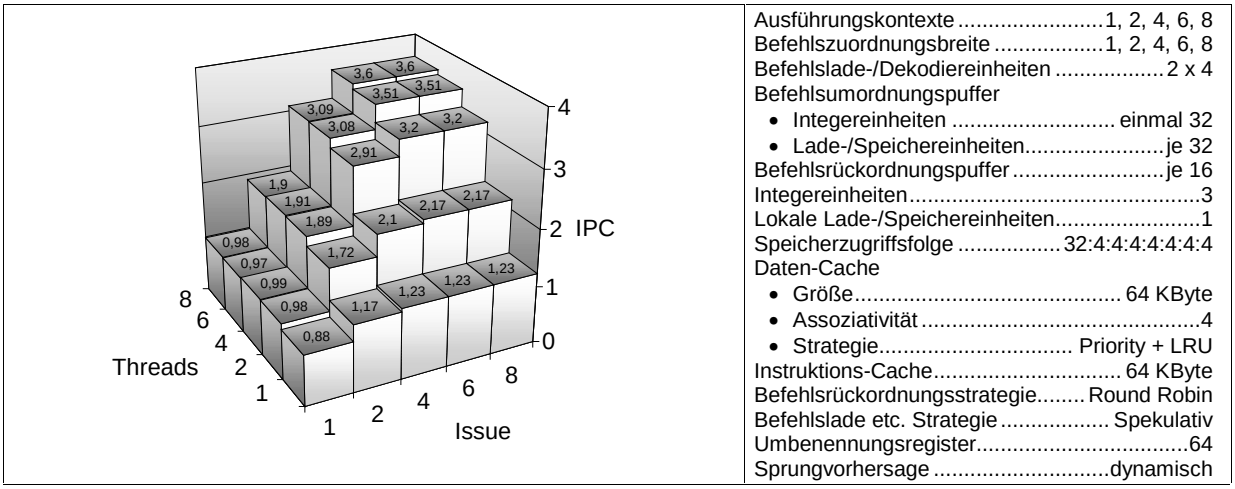
Abbildung 8-57: Maximalprozessor mit realistischen Caches

Diese Konfiguration entsteht aus dem Maximalprozessor, wenn realistische Cache-Größen und Speicherzugriffszeiten benutzt werden. Folgende Parameter wurden hierbei durch die Simulation als signifikant ermittelt:

- Die Befehlsrückordnungspuffer sollten für jeden Ausführungskontext deutlich kürzer sein, als die gemeinsam genutzten Befehlsumordnungspuffer. Dies verhindert, dass einzelne Kontrollfäden mit einer sehr hohen Cache-Fehlzugriffsrate zu einer Blockade der Ausführung durch Ressourcenbelegung führen.

- Die Cache-Zeilenersetzungsstrategie hat einen großen Einfluss auf die Leistungsfähigkeit der Caches. Dieser Einfluss nimmt mit zunehmender Assoziativität der Caches weiter zu, da die einzelnen von der Ersetzungsstrategie unabhängigen und direkt adressierten Segmente kleiner werden.

- Der Speicherdurchsatz ist teilweise wichtiger als die Latenz. Durch eine Verlängerung der Speicherzugriffsfolge konnte der IPC des Prozessors erhöht werden. Dies lässt darauf schließen, dass der Prozessor durch den erhöhten Speicherdurchsatz mehr gewinnt, als er durch die gleichzeitige Vergrößerung der Latenz verliert. Der mehrfädige Fall gewinnt aufgrund seiner höheren Latenztoleranz erwartungsgemäß stärker durch die Verlängerung der Speicherzugriffsfolge.



Ausführungskontexte	1, 2, 4, 6, 8
Befehlszuordnungsbreite	1, 2, 4, 6, 8
Befehlslade-/Dekodiereinheiten	2 x 4
Befehlsumordnungspuffer	
• Integereinheiten	einmal 32
• Lade-/Speichereinheiten	je 32
Befehlsrückordnungspuffer	je 16
Integereinheiten	3
Lokale Lade-/Speichereinheiten	1
Speicherzugriffsfolge	32:4:4:4:4:4:4:4
Daten-Cache	
• Größe	64 KByte
• Assoziativität	4
• Strategie	Priority + LRU
Instruktions-Cache	64 KByte
Befehlsrückordnungsstrategie	Round Robin
Befehlslade etc. Strategie	Spekulativ
Umbenennungsregister	64
Sprungvorhersage	dynamisch

Abbildung 8-58: Realistischer Prozessor

Durch eine Reduzierung der internen Prozessorelemente auf technisch machbare Werte entsteht eine realistische Prozessorkonfiguration. Folgende Simulationsergebnisse stehen hierbei im Vordergrund.

- Die Befehlsrückordnungspuffer jedes Ausführungskontextes sollten deutlich kleiner sein, als die gemeinsam genutzten Befehlsumordnungspuffer. Dies wirkt sich allerdings nachteilig auf die Leistung im einfädigen Fall. Hier wäre eine dynamische, von der Anzahl der aktiven Ausführungskontexte abhängige Limitierung der Befehlsrückordnungspuffer nützlich.

- Die Verwendung einer Strategie, die spekulative Befehle in der Befehlslade-, -dekodier- und -Zuordnungsstufe benachteiligt, erhöht den Befehlsdurchsatz des Prozessors. Zustandsbasierte Strategien zur Auswahl der Instruktionen innerhalb der Befehlsrückordnungsstufe bringen keinen nennenswerten Gewinn.

- Bei einer stark mehrfädigen Konfiguration kann auf eine spekulative Ausführung von Befehlen verzichtet werden. Dies kann in einzelnen Fällen sogar zu einer besseren Leistung führen. Allerdings kann nicht an der Qualität der Spekulation gespart werden, da sich bei schlechterer Spekulation die Anzahl der unnötig ausgeführten Instruktionen deutlich erhöht, und somit die Leistung des mehrfädigen Prozessors sinkt.

- Weniger als 64 Umordnungsregister führen zu einem deutlichen Leistungseinbruch.

- Die Verwendung von drei (anstatt vier) Integerausführungseinheiten und nur einer lokalen Lade-/Speichereinheit führen zu einem Rückgang des IPC um 0,5 in der (8, 8) und 0,12 in der (1, 8) Konfiguration. Obwohl die Ausführungseinheiten nicht vollständig ausgelastet sind, limitieren sie die Leistung des Prozessors. Dies ist auf die Ungleichverteilung der Befehlsarten während der Programmausführung zurückzuführen. Der mehrfädige Prozessor führt zwar zu einer deutlich regelmäßigeren Verteilung der Befehlsarten, kann dieses Problem aber nicht völlig maskieren.

Betrachten man den Geschwindigkeitsgewinn durch die Mehrfädigkeit der einzelnen Konfigurationen, so ergibt sich folgendes Bild:

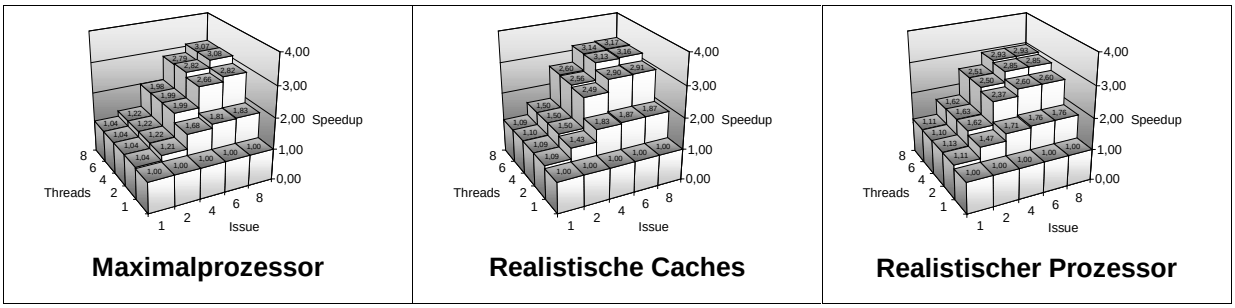


Abbildung 8-59: Realistischer Prozessor

In den Konfigurationen mit achtfacher Zuordnungsbreite ergibt sich ein relativ gleichmäßiger Geschwindigkeitsgewinn von etwa drei bei acht Kontrollfäden, 2,6 bei vier Kontrollfäden und 1,8 bei zwei Kontrollfäden. Bei zwei und vierfacher Zuordnungsbreite können die mehrfädigen Konfigurationen bei realistischen Caches stärkere Geschwindigkeitsgewinne vorweisen, als die Maximalvariante. Dies liegt darin begründet, dass die einfädigen Konfigurationen stärker unter den Latenzen des Speichersystems leiden als die mehrfädigen.

9 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein mehrfädiger Prozessor mit Multimediaerweiterungen spezifiziert und ein taktgenauer Simulator für diesen Prozessor entwickelt. Als Simulationslast wurde ein mehrfädiger MPEG-2-Dekoder in Assembler implementiert, der die Multimediafähigkeiten des spezifizierten Prozessors nutzt. Zahlreiche Simulationen verschiedener Prozessorkonfigurationen wurden durchgeführt und ausgewertet.

Die Simulationen zeigen, dass ein Prozessor mit nur einem Kontrollfaden nur geringe Gewinne durch eine höhere Zuordnungsbandbreite erzielen kann. Ein mehrfädiger Prozessor dagegen kann die zusätzliche Bandbreite nutzbringend in ausgeführte Instruktionen umsetzen.

Für das getestete Lastprogramm erreicht ein einfädiger Maximalprozessor bei achtfacher Befehlszuordnungsbreite eine Leistung von 1,99 IPC, wogegen ein achtfädiger Prozessor 6,11 IPC erreicht. Diese Leistungssteigerung um einen Faktor von etwa drei, wird auch bei der Verwendung von realistischen Cache und Speichersystems sowie einer realistischen Ressourcenverwendung erreicht. Somit zeigt sich, dass ein mehrfädiger Prozessor ideal geeignet ist, die für Multimedia typischen Algorithmen zu beschleunigen.

Es ist für die Leistungsgewinne in einem mehrfädigen Prozessors wichtig, dass gegenseitige negative Beeinflussungen der einzelnen Kontrollfäden vermieden werden. Besonders wichtig ist dies für den Kontrollfaden, der den kritischen Pfad ausführt. Als förderlich haben sich hierfür folgende Verfahren erwiesen:

- Benachteiligung spekulativer Instruktionen in der Befehlslade-, Dekodier- und Zuordnungsstufe.
- Bevorzugung der Kontrollfäden mit hoher Priorität bei der Cache-Zeilenersetzung
- Beschränkung der maximalen Anzahl an Befehlen, die ein einzelner Kontrollfaden in den gemeinsam benutzten Ausführungselementen haben kann.
- Keine Synchronisierung der Befehle verschiedener Kontrollfäden bei Lade-/Speicher und Ein-/Ausgabebefehlen

Ein mehrfädiger Prozessor ist in der Lage seinen Speicherbus deutlich besser auszulasten als ein einfädiger. Wichtig ist hierbei, dass die Cache-Architektur eine echte LRU Strategie implementiert, da die Zugriffsmuster eines mehrfädigen Prozessors deutlich ungeordneter sind.

Weiterhin hat sich gezeigt, dass eine zu aggressive Spekulation bei mehrfädigen Prozessoren zu einer geringeren Leistung führen kann. Dies ist darin begründet, dass die Ressourcen, die bei einem einfädigen Prozessor ungenutzt für spekulative Ausführung bereitstehen, bei einem mehrfädigen Prozessor durch andere Kontrollfäden sinnvoller genutzt werden können.

Auf der Seite der Simulationslast hat sich gezeigt, dass ein MPEG-2-Dekoder genug Parallelität auf Kontrollfadenebene besitzt, um acht gleichzeitig ausführende Kontrollfäden zu bedienen. Dies lässt darauf schließen, dass Multimediaanwendungen, die meist aus deutlich mehr unabhängigen und

gleichzeitig ausgeführten Programmteilen bestehen, genug Kontrollfäden für einen stark mehrfädigen Prozessor bereit stellen.

Die Ergebnisse dieser Arbeit könnten durch weitere Untersuchungen im Bereich der Bildkompression und -dekompression sowie drei dimensionalen Bilderzeugung oder der Audio-Verarbeitung ergänzt und gestützt werden. Aufgrund der ähnlichen Strukturen vieler dieser Anwendungen ist zu vermuten, dass sie einen ähnlichen Gewinn aus der Mehrfädigkeit ziehen können.

Mehrfädige Prozessoren sowie mehrere Prozessoren auf einem Chip beginnen langsam Realität zu werden. Für einen Erfolg dieser Prozessoren im breiten Markt ist es allerdings notwendig, dass Anwendungen für die gegebene Parallelität gefunden werden. Diese Arbeit hat gezeigt, dass Multimediaanwendungen ideal geeignet sind, die Leistung mehrerer Prozessoren bzw. eines mehrfädigen Prozessors zu nutzen.

10 Anhang

10.1 Beschreibung der Befehle, sortiert nach Ausführungseinheiten

10.1.1 Sprungbefehle

Die Sprungausführungseinheit führt zusammen mit der Befehlsdekodiereinheit die unbedingten und bedingten Sprünge aus. Das Sprungziel ist hierbei immer in der Form Indirekt mit Versatz (Offset) gegeben. Relative Sprünge können durch r2 im Argument, absolute Sprünge durch r0 im Argument ausgeführt werden.

10.1.1.1 BRA – Branch uncoditional

Kurzbeschreibung: Unbedingter Sprung

Adressierungsarten:

```
[d, Rb]
```

Erweiterungen:

Kodierung:

```
11100[Rb ]ddddddddddddddddddddddd
```

Detailinformation:

Der Versatz in der Sprungdistanz relativ zum Basisregister kann maximal +/- 2^{21} betragen. Der Sprung wird immer ausgeführt. Ist das Basisregister r0, so wird durch die Dekodiereinheit ein absoluter Sprung ausgeführt. Bei der Verwendung von Basisregister r2 wird der Sprung relativ zum Programmzähler ebenfalls bereits in der Befehlsdekodiereinheit genommen. Alle anderen Sprünge werden erst in der Sprungausführungseinheit ausgeführt, und führen somit zu mehreren Leertakten der Befehlslade- und -dekodiereinheit.

Beispiel:

Benutzung als Unterprogrammaufruf

```
subroutine:                                 ; Rücksprungadresse in r4
               sub          r3, 1, r3       ; Rücksprungadresse auf Stapel legen
               stl          r4, [r3]
               ...
               ldl          [r3], r4        ; Rücksprungadresse vom Stapel holen
               add          r3, 1, r3
               bra          [r4]            ; Rücksprung

main:
               add          r2, 2, r4
               bra          [subroutine, r2]
```

10.1.1.2 BEQ – Branch if equal

Kurzbeschreibung: Bedingter Sprung

Adressierungsarten:

```
Rn, [d, Rb]
```

Erweiterungen:

```
.W, .UH, .LH, .LB, .LMB, .UMB, .UB
```

Kodierung:

```
10101[Rs ][Rb ]ssscccddddddddddd
```

sss : Operandengröße

000 Low Byte

001 Low Mid Byte

010 High Mid Byte

011 High Byte

100 Low Half

101 High Half

110 Word

ccc : Bedingung

000 Rs == 0

001 Rs != 0

010 Rs < 0

011 Rs >= 0

100 Rs > 0

101 Rs >= 0

Detailinformation:

Der Versatz in der Sprungdistanz relativ zum Basisregister kann maximal +/- 2^{11} betragen. Der Sprung wird ausgeführt, falls die Bedingung erfüllt ist. Es kann auch nur ein Teil des Quelloperanden

zur Entscheidung herangezogen werden. In diesem Fall wird der betroffene Teil durch die Befehlserweiterung angegeben.

Sprünge relativ zum Programmzeiger (r2) werden durch die Sprungvorhersage bearbeitet, absolute Sprünge werden immer als nicht genommen angenommen.

Beispiel:

```
              add           r0, 100, r4
loop:
              sub           r4, 1, r4
              bge           r4, [loop, R2]
```

10.1.1.3 BNE – Branch if not equal

siehe BEQ

10.1.1.4 BLT – Branch if less than

siehe BEQ

10.1.1.5 BLE – Branch if less or equal

siehe BEQ

10.1.1.6 BGT – Branch if greater than

siehe BEQ

10.1.1.7 BGE – Branch if greater or equal

siehe BEQ

10.1.2 ALU Befehle

10.1.2.1 ADD – Add unsigned

Kurzbeschreibung: Addition mit Überlaufsarithmetik

Adressierungsarten:

1. `Rs1, Rs2, Rd`
2., 3. `Rs, Imm, Rd`

Erweiterungen:

1. `.W, .H, .B`
2. `.L, .U`
3. `.L, .LM, .UM, .U, .LH, .UH, .B`

Kodierung:

1. `10000[Rd ][Rs1][Rs2]xxxxxsseeeee`

```
2. 00ccc[Rd ][Rs1]uiiiiiiiiiiiiiii
3. 00011[Rd ][Rs1]iiiiiiiippsseeeee
```

ccc : Kommando

000	ADD
001	SUB
010	RSUB
011	EXT (siehe 3.)
100	OR
101	AND
110	ANDN
111	XOR

eeeee : Erweitertes Kommando

00000	ADD
00001	ADDS
00010	ADDSS
00100	SUB
00101	SUBS
00110	SUBSS
01000	MAX
01001	MAXS
01010	MIN
01011	MINS
01100	AVG
01101	AVGS
10000	AND
10001	OR
10010	XOR
10011	NAND
11000	LSL
11001	LSR
11010	ASL
11011	ASR
11100	ROL
11101	ROR

ss : Operandengröße

00 Byte

01 Halbwort

10 Wort

pp : Position des acht Bit Direktoperanden

00 Low Byte

01 Low Mid Byte

10 High Mid Byte

11 High Byte

u : Position des 16 Bit Direktoperanden

0 Low Word

1 High Word

Detailinformation:

Die beiden Quellargumente werden mit Überlauf addiert, und das Ergebnis dem Zielregister zugewiesen. In den Formen 1 und 3 können auch jeweils Teilworte getrennt bearbeitet werden. Das bedeutet, es werden die jeweils zusammengehörigen Bytes oder Halbworte getrennt addiert, und die Ergebnisse wieder zu einem Wort zusammengefasst. Überläufe werden in jedem Byte oder Halbwort getrennt behandelt. Die Erweiterungen .UH, .LH und .H stehen für Halbwortarithmetik, die Erweiterung .B steht für Bytearithmetik.

Beispiel:

```
r4 = $01020304
r5 = $102030ff

               add.b          r4, r5, r6

r6 = $11223303
```

10.1.2.2 ADDS – Add unsigned saturated

Kurzbeschreibung: Vorzeichenlose Addition mit Saturierungsarithmetik

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie ADD, nur entsteht kein Überlauf. Würde ein Überlauf in die nächste Stelle erfolgen, so wird automatisch auf den größten möglichen Wert (also $ff, $ffff, $ffffffff) abgerundet.

Beispiel:

```
r4 = $01120304
r5 = $107030ff

               adds.b          r4, r5, r6

r6 = $118233ff
```

10.1.2.3 ADDSS – Add signed saturated

Kurzbeschreibung: Vorzeichenbehaftete Addition in Saturierungsarithmetik

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie ADDS, nur werden hierbei die Operanden (bzw. Teiloperanden) als vorzeichenbehaftete Zahlen betrachtet. Das hat zur Folge, das die Saturierung zu den entsprechenden Zweierkomplementwerten hin erfolgt.

Beispiel:

```
r4 = $01120304
r5 = $107030ff

               addss.b         r4, r5, r6

r6 = $117f3303
```

10.1.2.4 SUB – Subtract unsigned

Kurzbeschreibung: Subtraktion mit Unterlauf

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Der zweite Quelloperand wird von dem ersten Quelloperanden abgezogen, das Ergebnis dem Zielregister zugewiesen. Wie auch bei ADD können Teiloperanden verwendet werden.

Beispiel:

```
r4 = $0172e304
r5 = $103070ff

               sub.b           r4, r5, r6

r6 = $f1427305
```

10.1.2.5 RSUB – Reverse subtract unsigned

Kurzbeschreibung: Subtraktion mit Unterlauf

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie SUB, nur wird hier der erste vom zweiten Operanden abgezogen. Dieser Befehl wird benötigt, falls der erste Operand einer Subtraktion durch einen Direktwert gegeben wird.

Beispiel:

```
r4 = $0172e304
r5 = $103070ff

               rsub.b          r4, r5, r6

r6 = $0fbe8dfb
```

10.1.2.6 SUBS – Subtract unsigned saturated

Kurzbeschreibung: Vorzeichenlose Subtraktion mit Saturierungsarithmetik

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie SUB, nur wird bei einem Unterlauf das Ergebnis durch den Wert 0 ersetzt.

Beispiel:

```
r4 = $0172e304
r5 = $103070ff

               sub.b           r4, r5, r6

r6 = $00427300
```

10.1.2.7 SUBSS – Subtract signed saturated

Kurzbeschreibung: Vorzeichenbehaftete Subtraktion in Saturierungsarithmetik

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie SUBS, nur werden hierbei die Zahlen als vorzeichenbehaftet betrachtet, es wird also auf die entsprechenden Zweierkomplement-Extreme begrenzt.

Beispiel:

```
r4 = $0172e304
r5 = $103070ff

               sub.b           r4, r5, r6

r6 = $f1428005
```

10.1.2.8 MAX – Maximum unsigned

Kurzbeschreibung: Vorzeichenlose Bildung des Maximums

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Das Maximum der beiden Quelloperanden wird in den Zieloperand geladen. Findet die Operation auf Subwortebene statt, so wird der Test auf Maximum für jedes Subwort getrennt durchgeführt.

Beispiel:

```
r4 = $114499ee
r5 = $223377bb

               max.b          r4, r5, r6

r6 = $224477ee
```

10.1.2.9 MAXS – Maximum signed

Kurzbeschreibung: Bildung des Maximums vorzeichenbehafteter Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie MAX, nur werden die Argumente als vorzeichenbehaftet betrachtet

Beispiel:

```
r4 = $114499ee
r5 = $223377bb

               maxs.b         r4, r5, r6

r6 = $224477ee
```

10.1.2.10 MIN – Minumum unsigned

Kurzbeschreibung: Bildung des Minimums vorzeichenloser Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

siehe MAX

10.1.2.11 MINS – Minimum signed

Kurzbeschreibung: Bildung des Minimums vorzeichenbehafteter Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

siehe MAXS

10.1.2.12 AVG – Average unsigned

Kurzbeschreibung: Mittelwertbildung zweier vorzeichenloser Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Das arithmetische Mittel der beiden Quelloperanden wird dem Zielregister zugewiesen. Hierzu werden die beiden Operanden addiert, und das Ergebnis durch zwei geteilt. Es wird eine Rundung zur nächsten Zahl verwendet, wobei ein halbes LSB auf den nächsthöheren Wert gerundet wird.

Beispiel:

```
r4 = $104090e0
r5 = $203070b0

               avg.b          r4, r5, r6

r6 = $183880c8
```

10.1.2.13 AVGS – Average signed

Kurzbeschreibung: Mittelwertbildung zweier vorzeichenbehafteter Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie AVG, nur werden hierbei die Zahlen als vorzeichenbehaftet betrachtet.

Beispiel:

```
r4 = $104090e0
r5 = $203070b0

               avgs.b         r4, r5, r6

r6 = $183800c8
```

10.1.2.14 CMP – Compare unsigned

Kurzbeschreibung: Vergleich vorzeichenloser Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Die beiden Quelloperanden werden miteinander verglichen. Ist der erste Operand kleiner als der zweite, so wird -1 in das Zielregister geschrieben. Sind beide identisch 0, sonst 1. Das Ergebnis kann dann in einem bedingten Sprungefehl ausgewertet werden. Wird der Befehl auf Subwortebene ausgeführt, so wird jedes Subwort getrennt betrachtet. Das Ergebnis kann auch verwendet werden, um mit der Hilfe der AND/ANDN und OR Befehle eine Maskierungskaskade zu bilden.

Beispiel:

```
r4 = $104090b0
r5 = $203070b0

               cmp.b          r4, r5, r6

r6 = $ff010100
```

10.1.2.15 CMPS – Compare signed

Kurzbeschreibung: Vergleich vorzeichenbehafteter Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie CMP, nur werden hierbei die Zahlen als vorzeichenbehaftet betrachtet.

Beispiel:

```
r4 = $104090b0
r5 = $203070b0

               cmps.b         r4, r5, r6

r6 = $ff01ff00
```

10.1.2.16 OR – Binary logical or

Kurzbeschreibung: Bitweises „oder“ zweier Operanden

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

An den beiden Quelloperanden wird ein bitweises Oder durchgeführt, das Ergebnis dem Zielregister zugewiesen. Auf die Ausführung hat eine eventuelle Subwortarithmetik keinen Einfluss, lediglich auf die Verwendung eines Direktoperanden.

10.1.2.17 AND – Binary logical and

Kurzbeschreibung: Bitweises „und“ zweier Operanden

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Wie OR, nur mit einer bitweisen Und-Verknüpfung

10.1.2.18 XOR – Binary logical exclusive or

Kurzbeschreibung: Bitweises „exklusives oder“ zweier Operanden

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

siehe OR

10.1.2.19 ANDN – Binary logical and not

Kurzbeschreibung: Bitweise Und-Verknüpfung eines Operanden mit dem 1-Komplement eines anderen

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Aus dem ersten Operanden und dem 1-Komplent des zweiten Operanden wird die bitweise UND-Verknüpfung gebildet und dem Ergebnis zugewiesen. Diese Operation kann zusammen mit der AND Anweisung verwendet werden, um eine Auswahl aus zwei Quelloperanden mit der Hilfe einer Maske zu bilden.

Beispiel:

Auswahl der Bytes aus zwei Quellen basierend auf einer Maske:

```
r4 = $ff0000ff ; Maske
r5 = $51525354 ; Quelle bei 1
r6 = $6a6b6c6d ; Quelle bei 0

               and            r5, r4, r5
               andn           r6, r4, r6
               or             r5, r6, r5

r5 = $516b6c54
```

10.1.2.20 PACKU – Pack upper

Kurzbeschreibung: Zusammenfassen zu Teilworten

Adressierungsarten:

Rs1, Rs2, Rd

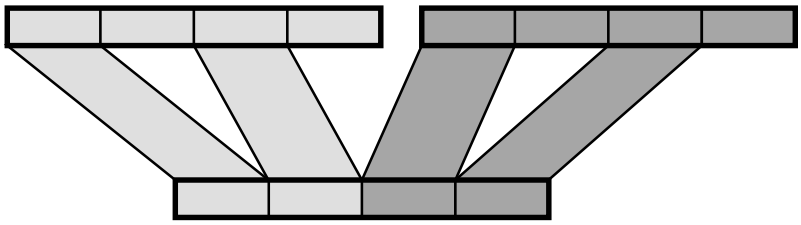
Erweiterungen:

.H, .B

Kodierung: siehe ADD

Detailinformation:

Die höherwertigen Halbworte (bzw. Bytes) der beteiligten Wortoperanden (bzw. der vier Halbwortoperanden) werden zu einem neuen Wort zusammengefasst.



Beispiel:

```
r4 = 0x01234567
r5 = 0x89abcdef

               packu.h        r4, r5, r6
               packu.b        r4, r5, r7

r6 = 0x12389ab
r7 = 0x014589cd
```

10.1.2.21 PACKL – Pack lower

Kurzbeschreibung: Zusammenfassen zu Teilworten

Adressierungsarten:

```
Rs1, Rs2, Rd
```

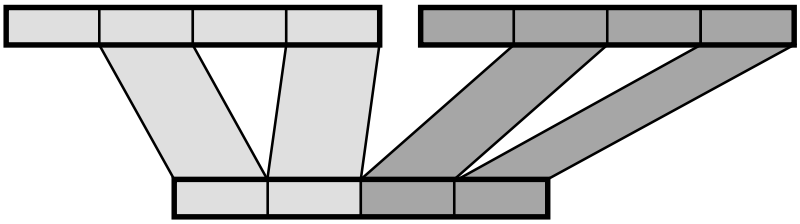
Erweiterungen:

```
.B, .H
```

Kodierung: siehe ADD

Detailinformation:

Die niederwertigen Halbworte (bzw. Bytes) der beiden beteiligten Wortoperanden (bzw. der vier Halbwortoperanden) werden zu einem neuen Wort zusammengefasst.



Beispiel:

```
r4 = 0x01234567
r5 = 0x89abcdef

               packl.h        r4, r5, r6
               packl.b        r4, r5, r7

r6 = 0x4567cdef
r7 = 0x2367abef
```

10.1.2.22 EXTU – Extract upper

Kurzbeschreibung: Erweitern von Teilworten

Adressierungsarten:

```
Rs1, Rs2, Rd
```

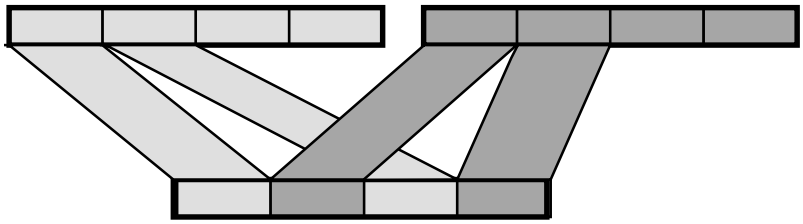
Erweiterungen:

```
.B, .H
```

Kodierung: siehe ADD

Detailinformation:

Die höherwertigen Halbworte (bzw. Bytes) der beiden beteiligten Wortoperanden (bzw. der vier Halbwortoperanden) weden zu einem neuen Wort zusammengefasst.



Beispiel:

```
r4 = 0x01234567
r5 = 0x89abcdef

               extu.b         r4, r5, r7
               extu.b         r0, r4, r8

r7 = 0x018923ab
r8 = 0x00010023
```

10.1.2.23 EXTL – Extract lower

Kurzbeschreibung: Erweitern von Teilworten

Adressierungsarten:

```
Rs1, Rs2, Rd
```

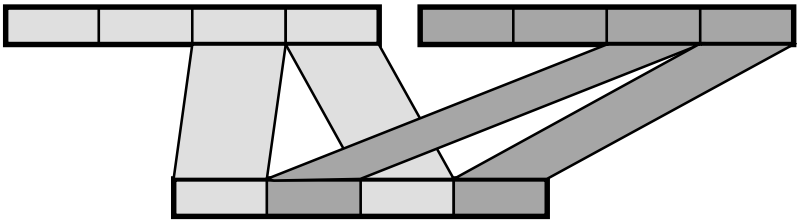
Erweiterungen:

```
.B, .H
```

Kodierung: siehe ADD

Detailinformation:

Die niederwertigen Halbworte (bzw. Bytes) der beteiligten Wortoperanden (bzw. der vier Halbwortoperanden) werden zu einem neuen Wort zusammengefasst.



Beispiel:

```
r4 = 0x01234567
r5 = 0x89abcdef

               extl.b         r4, r5, r7
               extl.b         r0, r4, r8

r7 = 0x45bc67ef
r8 = 0x00450023
```

10.1.2.24 LSL – Logical shift left

Kurzbeschreibung: Logisches links Verschieben

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Der erste Operand wird um die Anzahl Bits die im zweiten Operanden angegeben sind nach links verschoben und das Ergebnis dem Zielregister zugewiesen. Die frei werdenden Bits werden mit 0 aufgefüllt.

Beispiel:

```
r4 = $ff36c351
                lsl.b           r4, 4, r4
r4 = $f0603010
```

10.1.2.25 LSR – Logical shift right

Kurzbeschreibung: Logisches rechts Verschieben

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Der erste Operand wird um die Anzahl Bits, die im zweiten Operanden angegeben sind nach rechts verschoben, und das Ergebnis dem Zielregister zugewiesen. Die freiwerdenden Bits werden mit 0 gefüllt.

Beispiel:

```
r4 = $ff36c351
                lsr.b           r4, 4, r4
r4 = $0f030c05
```

10.1.2.26 ASL – Arithmetical shift left

siehe LSL

10.1.2.27 ASR – Arithmetical shift right

Kurzbeschreibung: Arithmetisches rechts Verschieben

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Der erste Operand wird um die Anzahl Bits, die im zweiten Operanden angegeben sind nach rechts verschoben, und das Ergebnis dem Zielregister zugewiesen. Die freiwerdenden Bits werden mit dem Vorzeichenbit aufgefüllt.

Beispiel:

```
r4 = $ff36c351
                asr.b           r4, 4, r4
r4 = $ff03fc05
```

10.1.2.28 ROL – Rotate left

Kurzbeschreibung: Linksrotieren eines Operanden

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Der erste Operand wird um die Anzahl Bits, die im zweiten Operanden angegeben sind, nach links verschoben und das Ergebnis dem Zielregister zugewiesen. Die freiwerdenden Bits werden mit den auf der anderen Seite herausgeschobenen Bits wieder aufgefüllt.

Beispiel:

```
r4 = $ff36c351

               rol.h          r4, 4, r4

r4 = $f36f351c
```

10.1.2.29 ROR – Rotate right

Kurzbeschreibung: Rechtsrotieren eines Operanden

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Der erste Operand wird um die Anzahl Bits, die im zweiten Operanden angegeben sind, nach rechts verschoben und das Ergebnis dem Zielregister zugewiesen. Die freiwerdenden Bits werden mit den auf der anderen Seite heraus geschobenen Bits wieder aufgefüllt.

Beispiel:

```
r4 = $ff36c351

               ror.h          r4, 4, r4

r4 = $6ff31c35
```

10.1.2.30 BFF – Bit find first

Kurzbeschreibung: Suchen des ersten Auftretens eines Bits

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Es wird das niederwertigste Bit im ersten Operanden gesucht, das mit dem niederwertigsten Bit des zweiten Operanden übereinstimmt. Die Position dieses Bits wird in das Zielregister geschrieben. Wird keines gefunden, so wird -1 als Ergebnis in das Zielregister übernommen.

Beispiel:

```
r4 = 0xff55337f
r5 = 0x00010000

               bff.b          4, r5, r6

r6 = 0xff010207
```

10.1.2.31 BFL – Bit find last

Kurzbeschreibung: Suchen des letzten Auftretens eines Bits

Adressierungsarten, Erweiterungen, Kodierung: siehe ADD

Detailinformation:

Es wird das höchstwertigste Bit im ersten Operanden gesucht, das mit dem niederwertigsten Bit des zweiten Operanden übereinstimmt. Die Position dieses Bits wird in das Zielregister geschrieben. Wird keines gefunden, so wird -1 als Ergebnis in das Zielregister übernommen. Dieser Befehl kann verwendet werden, um die Länge der Huffman-Dekodierungstabellen zu vermindern. Da für längere Codes meistens relative lange Nullfolgen mit einer kurze Abschlussfolge aus Einsen und Nullen verwendet werden, kann somit der Code in eine Nullfolgenlänge und einen kurzen (meist acht Bit) langen Tabellenindex übersetzt werden. Dies ermöglicht es, die Tabellen für die Rückübersetzung auf den signifikanten Teil des Codewortes zu beschränken.

Beispiel:

```
r4 = 0xff55337f
r5 = 0x00010000

               bfl.b          r4, r5, r6

r6 = 0xff060707
```

10.1.3 Lade-/ Speicherbefehle für internen Speicher

10.1.3.1 LDL – Load local

Kurzbeschreibung: Laden eines Wortes aus dem lokalen Speicher

Adressierungsarten:

```
[d, Rb], Rd
```

Erweiterungen:

Kodierung:

```
11010[Rd ][Rd ]ddddddddddddddddd
```

Detailinformation:

Die Zieladresse des Befehls berechnet sich als die Summe aus dem Inhalt des Basisregisters und eines konstanten vorzeichenbehafteten Offsets. Der Offset darf maximal +/- 2^{16} betragen. Je nach Größe des internen Speichers wiederholen sich die Speicherstellen im Adressraum. Um eine absolute Adressierung zu erreichen, kann Register 0 als Basisregister verwendet werden. Ist kein interner Speicher vorhanden, so werden die Zugriffe durch den Übersetzer auf den externen Speicher umgebogen.

Beispiel:

```
data           LOCAL          3

start:
               ldl            [data, r0], r4
```

10.1.3.2 STL – Store local

Kurzbeschreibung: Schreiben eines Wortes in den internen Speicher

Adressierungsarten:

```
Rs, [d, Rb]
```

Erweiterungen:

Kodierung:

```
10110[Rs ][Rb ]dddddddddddddddd
```

Detailinformation:

Die Zieladresse des Befehls berechnet sich als die Summe aus dem Inhalt des Basisregisters und eines konstanten vorzeichenbehafteten Offsets. Der Offset darf maximal +/- 2^{16} betragen. Je nach Größe des internen Speichers wiederholen sich die Speicherstellen im Adressraum. Um eine absolute Adressierung zu erreichen, kann Register 0 als Basisregister verwendet werden. Ist kein interner Speicher vorhanden, so werden die Zugriffe durch den Übersetzer auf den externen Speicher umgebogen.

Beispiel:

```
data            LOCAL           3
start:
                stl             r4, [data, r0]
```

10.1.4 Lade-/ Speicherbefehle für externen Speicher

10.1.4.1 LDM – Load memory

Kurzbeschreibung: Laden eines Wortes aus dem externen Speicher

Adressierungsarten:

```
[d, Rb], Rd
```

Erweiterungen:

Kodierung:

```
11000[Rb ][Rd ]dddddddddddddddd
```

Detailinformation:

Das Zieladresse des Befehls ergibt sich durch die Summe aus dem Inhalt des Basisregisters und eines konstanten vorzeichenbehafteten Versatzes. Der Versatz darf maximal +/-2^{16} betragen. Speicheradressen sind immer Wortadressen. Um eine absolute Adressierung zu erreichen kann für die ersten 64 KByte des Speichers Register 0 als Basisregister verwendet werden.

Beispiel:

```
data            BSS             3
start:
                ldm             [data, r0], r4
```

10.1.4.2 STM – Store memory

Kurzbeschreibung: Schreiben eines Wortes in den externen Speicher

Adressierungsarten:

```
Rs, [d, Rb]
```

Erweiterungen:

Kodierung:

```
10100[Rs ][Rb ]ddddddddddddddddd
```

Detailinformation:

Der Versatz darf maximal +/-2^{16} betragen.

Beispiel:

```
data            BSS             3

start:
                stm             [data, r0], r4
```

10.1.5 Multiplikationsbefehle

Die Multiplikationseinheit verfügt über einen 64 Einträge großen Konstantenspeicher, dessen Elemente als zusätzliche Operanden der Multiplikation verwendet werden können. Die 32x32 Bit breite Multiplikationseinheit kann als ganzes, in vier 16x16 oder sechzehn 8x8 Bit Multiplikationseinheiten genutzt werden. Auf das Ergebnis kann, um Festkommazahlen zu implementieren, in vielen Fällen noch eine Rechtsverschiebung angewendet werden.

10.1.5.1 MUL – Multiply unsigned

Kurzbeschreibung: Vorzeichenlose Multiplikation zweier Operanden

Adressierungsarten:

```
1. Rs, Imm, Rd
2. Rs1, Rs2, Rd
```

Erweiterungen:

```
1. .W, .LW, .UW, .H, .LH, .UH, .B, .LB, .UB
2. .W, .H, .B + 0..31
```

Kodierung:

```
01ccc[Rd ][Rs ]sddddddddddddddd
```

ccc : Befehlskodierung

000 MUL.LW

001 MUL.LH

010 MUL.LB

100 MUL.UW

101 MUL.UH

110 MUL.UB

s: Vorzeichen

0 Vorzeichenlose Operation

1 Vorzeichenbehaftete Operation

```
1001[Rd ][Rs1][Rs2]SSSsIIIIITTT
```

SSS : Ergebnisverschiebung

	Byte	Halbwort	Wort
000	0	0	0
001	1	1	1
010	2	2	2
011	3	4	4
100	4	8	8
101	6	12	16
110	7	14	24
111	8	16	32

s : Vorzeichen

0 Vorzeichenlose Operation

1 Vorzeichenbehaftete Operation

IIII : Adresse im Konstantenspeicher

TTT : Typ der Multiplikation

000 MULI.H

001 MULXI.H

010 MULMI.H

011 MULI.B

100 MULMI.B

101 LDMC.L

110 LDMC.U

```
10001[Rd ][Rs1][Rs2]SSSs00XXX111
```

XXX : Erweiterte Befehlskodierung

000 MUL.W

001 MUL.H

010 MUL.B

011 MULC.H

100 MULX.H

101 MULX.B

Detailinformation:

Multiplikation zweier Operanden ohne Berücksichtigung des Vorzeichens. Wird diese Operation auf Subworte angewendet, so wird jedes Teilwort unabhängig multipliziert. Die Erweiterung L bzw. U bezeichnen das Wort/Teilwort des Ergebnisses, das verwendet werden soll. Nach erfolgter Multiplikation wird das Ergebnis um die angegebene Konstante nach rechts geschoben. Bei einem eventuell auftretenden Überlauf wird das Ergebnis abgeschnitten. Die Verwendung einer konstanten Rechtsverschiebung erlaubt die Multiplikation von Festkommazahlen ohne die Verwendung eines doppelt genauen Zwischenregisters.

Eine Multiplikation von 64 Bit genauen Ergebnis kann durch die Durchführung von zwei Multiplikationen erreicht werden, wobei das höherwertige Ergebnis um 32 Bit nach rechts geschoben verwendet wird.

Beispiel:

```
r4 = 0x10001000
r5 = 0x00004444

               mul.l          r4, r5, r6
               mul.u          r4, r5, r7

r6 = 0x44444000
r7 = 0x00000444
```

10.1.5.2 MULS – Multiply signed

Kurzbeschreibung: Multiplikation zweier vorzeichenbehafteter Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe MUL

Detailinformation:

Multiplikation zweier Zahlen unter Berücksichtigung des Vorzeichens. Wird diese Operation auf Teilworte angewendet, wird jedes Teilwort unabhängig multipliziert. Die Erweiterung L bzw. U bezeichnen das Wort/Teilwort des Ergebnisses, das verwendet werden soll.

10.1.5.3 MULC – Multiply complex

Kurzbeschreibung: Multiplikation zweier vorzeichenloser komplexer Zahlen

Adressierungsarten:

```
Rs1, Rs2, Rd
```

Erweiterungen:

```
.H + 0..31
```

Kodierung: siehe MUL

Detailinformation:

Zwei vorzeichenlose komplexe Zahlen, deren reeller und imaginärer Anteil in je einem Halbwort der Operanden gespeichert ist, werden miteinander multipliziert. Danach wird das Ergebnis um die angegebene Konstante nach rechts geschoben. Bei einem eventuell auftretenden Überlauf wird das Ergebnis durch Saturierungsarithmetik auf das Zieldatenformat beschränkt. Die Verwendung einer konstanten Rechtsverschiebung erlaubt die Multiplikation von Festkommazahlen ohne die Verwendung eines doppelt genauen Zwischenregisters.

```
Rd.UH = (Rs1.UH * Rs2.UH - Rs1.LH * Rs2.LH) >> SSS;
Rd.LH = (Rs1.UH * Rs2.LH + Rs1.LH * Rs2.UH) >> SSS;
```

Beispiel:

```
r4 = 0x00080004
r5 = 0x00030002

              mulc.h0        r4, r5, r6

r6 = 0x0012001c
```

10.1.5.4 MULCS – Multiply complex signed

Kurzbeschreibung: Vorzeichenbehaftete Multiplikation zweier komplexer Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe MULC

Detailinformation:

Zwei vorzeichenbehaftete komplexe Zahlen, deren reeller und imaginärer Anteil in je einem Halbwort der Operanden gespeichert ist, werden miteinander multipliziert.

Beispiel:

Kern der Mandelbrotmengenberechnung $M_{i+1} = M_i^{\,2} + M_0$ mit 8.8 Festkommazahlen:

```
; r4 : M0
; r5 : Anzahl der Iterationsschritte

               add            r4, r0, r6     ; Sichern von M0
               add            r0, 0, r5      ; Löschen des Zählers
mloop:
               mulcs.h8       r6, r6, r6     ; Mi * Mi
               addss.h        r4, r6, r6     ; + M0
               muls.h8        r6, r6, r7     ; |Mi+1|2
               asr            r7, 16, r8
               add.h          r7, r8, r7
               cmp.h          r7, 16, r7     ; vergleich mit r2
               add            r5, 1, r5
               ble.lh         r7, [mloop, r2]       ; weiter, wenn noch in Kreis
```

10.1.5.5 MULX – Multiply extended

Kurzbeschreibung: Erweiterte Multiplikation zweier vorzeichenloser Zahlen

Adressierungsarten:

```
Rs1, Rs2, Rd
```

Erweiterungen:

```
.H, .B + 0..31
```

Kodierung: siehe MUL

Detailinformation:

Das niederwertige Teilwort des zweiten Operanden wird jeweils mit allen Teilworten des ersten Operanden multipliziert. Danach wird das Ergebnis um die angegebene Konstante nach rechts geschoben. Bei einem eventuell auftretenden Überlauf wird das Ergebnis durch Saturierungsarithmetik auf das Zieldatenformat beschränkt. Die Verwendung einer konstanten Rechtsverschiebung erlaubt die Multiplikation von Festkommazahlen ohne die Verwendung eines doppelt genauen Zwischenregisters.

```
Rd.xH = Rs1.xH * Rs2.LH;
```

oder

```
Rd.xB = Rs1.xB * Rs2.LB;
```

Beispiel:

```
r4 = 0x01020304
r5 = 0x00000011

               mulx.b0        r4, r5, r6

r6 = 0x11223344
```

10.1.5.6 MULXS – Multiply extended signed

Kurzbeschreibung: Erweiterte Multiplikation zweier vorzeichenbehafteter Zahlen

Adressierungsarten, Erweiterungen, Kodierung: siehe MULX

Detailinformation:

Das niederwertigste Teilwort des zweiten Operanden wird jeweils mit allen Teilworten des ersten Operanden multipliziert.

10.1.5.7 MULI – Multiply internal

Kurzbeschreibung: Multiplikation zweier vorzeichenloser Zahlen mit Konstanten und Summe der Produkte

Adressierungsarten:

```
Rs1, Rs2, const, Rd
```

Erweiterungen:

```
.H, .B + 0..31
```

Kodierung: siehe MUL

Detailinformation:

Die Teilworte der beiden Operanden werden mit je einer Konstante aus dem Konstantenspeicher multipliziert und die Ergebnisse addiert. Danach wird das Ergebnis um die angegebene Anzahl Bits nach rechts geschoben. Bei einem eventuell auftretenden Überlauf wird das Ergebis durch Saturierungsarithmetik auf die Größe des Zieldatentyps beschränkt.

```
Rd.xH = Rs1.xH * Ca[c].xH + Rs2.xH * Cb[c].xH
```

oder

```
Rd.xB = Rs1.xB * Ca[c].xB + Rs2.xB * Cb[c].xB
```

Beispiel:

```
r4 = 0x00010002
r5 = 0x00030004
r6 = 0x00110022
r7 = 0x11002200

               ldmc.l         r4, r5, 0, r0
               muli.h         r6, r7, 0, r8

r8 = 0x33118844
```

10.1.5.8 MULIS – Multiply internal signed

Kurzbeschreibung: Multiplikation zweier vorzeichenbehafteter Zahlen mit Konstanten, und Summe der Produkte

Adressierungsarten, Erweiterungen, Kodierung: siehe MULI

Detailinformation:

Die Teilworte der beiden Operanden werden mit je einer Konstante aus dem Konstantenspeicher multipliziert und die Ergebnisse addiert.

10.1.5.9 MULXI – Multiply extended internal

Kurzbeschreibung: Erweiterte Multiplikation zweier Operanden mit Konstanten und Summe der Ergebnisse

Adressierungsarten, Erweiterungen, Kodierung: siehe MULI

Detailinformation:

Die Teilworte der vorzeichenlosen Operanden werden jeweils mit Konstanten multipliziert und innerhalb des Wortes summiert.

```
Rd.LH = Rs1.LH * Ca[c].LH + Rs1.UH * Ca[c].UH
Rd.UH = Rs2.LH * Cb[c].LH + Rs2.UH * Cb[c].UH
```

Beispiel:

```
r4 = 0x00010002
r5 = 0x00030004
r6 = 0x00112200
r7 = 0x01102002

               ldmc.l         r4, r5, 0, r0
               mulxi.h        r6, r7, 0, r8

r8 = 0x44116336
```

10.1.5.10 MULXIS – Multiply extended internal signed

Kurzbeschreibung: Erweiterte Multiplikation zweier Operanden mit Konstanten und Summe der Ergebnisse

Adressierungsarten, Erweiterungen, Kodierung: siehe MULXI

Detailinformation:

Die Teilworte der vorzeichenbehafteten Operanden werden jeweils mit Konstanten multipliziert und innerhalb des Wortes summiert.

10.1.5.11 MULMI – Multiply multiple internal

Kurzbeschreibung: Vorzeichenlose Multiplikation von vier Operanden mit Konstanten und Summe der Ergebnisse

Adressierungsarten, Erweiterungen, Kodierung: siehe MULI

Detailinformation:

Jeweils vier Teilworte werden mit Konstanten multipliziert, und die Ergebnisse aufsummiert. Da jedes der Zwischenergebnisse der Multiplikation in 31 Bit Platz hat, kann das Ergebnis bis zu 33 Bit einnehmen. Bei einem eventuell autretenden Überlauf wird das Ergebnis durch Saturierungsarithmetik auf 32 Bit beschränkt.

```
Rd = Rs1.LH * Ca[c].LH + Rs1.UH * Ca[c].UH +
     Rs2.LH * Cb[c].LH + Rs2.UH * Cb[c].UH;
```

oder

```
Rd.xB = Rs1.LB * Ca[c].xB + Rs1.LMB * Cb[c].xB + Rs1.UMB * Cc[c].xB + ...
```

10.1.5.12 MULMIS – Multiply multiple internal signed

Kurzbeschreibung: Vorzeichenbehaftete Multiplikation von vier Operanden mit Konstanten und Summe der Ergebnisse

Adressierungsarten, Erweiterungen, Kodierung: siehe MULMI

Detailinformation:

Jeweils vier Teilworte werden mit Konstanten multipliziert, und die Ergebnisse aufsummiert (siehe MULMI).

10.1.5.13 LDMC – Load multiplier constant

Kurzbeschreibung: Laden einer vorzeichenlosen Konstante in den Konstantenspeicher

Adressierungsarten:

```
Rs1, Rs2, const, Rx
```

Erweiterungen:

```
.L, .U
```

Kodierung: siehe MULI

Detailinformation:

Die beiden Quelloperanden werden in den Konstantenspeicher an der angegebenen Stelle abgelegt. Jeder Eintrag im Konstantenspeicher enthält vier Worte. Die Unterscheidung in welche Worte die Konstanten abgelegt werden, erfolgt durch die Erweiterung. Im Konstantenspeicher wird nach vorzeichenlosen und vorzeichenbehafteten Konstanten unterschieden. Diese Konstanten können durch Multiplikationsbefehle benutzt werden. Dies ermöglicht die Verwendung mehrerer Operatoren in einem Befehl, als durch das Befehlsformat möglich wäre.

```
ldmc.l     :     Ca[c] = Rs1, Cb[c] = Rs2
ldmc.u     :     Cc[c] = Rs1, Cd[c] = Rs2
```

10.1.5.14 LDMCS – Load multiplier constant signed

Kurzbeschreibung: Laden einer vorzeichenbehafteten Konstante in den Konstantenspeicher

Adressierungsarten, Erweiterungen, Kodierung, Detailinformation: siehe LDMC

10.1.6 Kontrollfaden- und Systemkontrollbefehle

Die Kontrollfadensteuerungseinheit verfügt über 32 Signale, die unabhängig voneinander gesetzt oder gelöscht werden können.

10.1.6.1 WAIT – Wait for signal

Kurzbeschreibung: Warten auf ein Signalbit

Adressierungsarten:

```
Rs, imm, Rd
```

Erweiterungen:

.W Verhindert, dass weitere Instruktionen ausgeführt werden, bis diese Instruktion komplett ausgeführt worden ist. Dies ist besonders für Synchronisationsanweisungen sinnvoll, mit denen auf ein Ereignis gewartet wird.

.M Verhindert, dass diese Instruktion ausgeführt wird, solange noch eine Lade oder Speicheroperation nicht komplett abgeschlossen ist. Dies ist besonders für Synchronisationsanweisungen sinnvoll, mit denen die Komplettierung eines Vorganges angezeigt werden soll.

Kodierung:

```
11001[Rd ][Rs ]WM00000ppiiiiiiii
```

W entspricht der Erweiterung W

M entspricht der Erweiterung M

pp Angabe auf welches Byte der Direktoperand iii angewendet werden soll.

Detailinformation:

Die beiden Quelloperanden werden durch ein binäres Oder miteinander verknüpft. Da Ergebniss bildet die Wartemaske. Ist ein Signalbit gesetzt, das auch in der Wartemaske gesetzt ist, wird die Operation ausgeführt, sonst wird auf dieses Ereignis gewartet. Das Ergebnis der Instruktion ist der Zustand des Signalregisters zum Ausführungszeitpunkt.

Beispiel:

Implementierung eines Ereignisses:

```
WaitForEvent:                                   ; Signalbit in r4, Rücksprung in r5
              wait.w        r4, 0, r0
              bra           [r5]

SetEvent:                                       ; Signalbit in r4, Rücksprung in r5
              setsig.m      r4, 0, r0
              bra           [r5]

ClearEvent:                                     ; Signalbit in r4, Rücksprung in r5
              clrsig.w      r4, 0, r0
              bra           [r5]
```

10.1.6.2 SYNC – Synchronize to signal

Kurzbeschreibung:

Warten auf ein Signalbit, mit anschließendem Löschen

Adressierungsarten, Erweiterungen: siehe WAIT

Kodierung:

```
11001[Rd ][Rs ]WM00001ppiiiiiiii
```

Detailinformation:

Die beiden Quelloperanden werden durch ein binäres Oder verknüpft. Das Ergebnis bildet die Wartemaske. Ist ein Signalbit gesetzt, das auch in dieser Wartemaske gesetzt ist, wird die Operation ausgeführt, sonst wird auf dieses Ereignis gewartet. Nach der Ausführung der Operation werden alle betroffenen Signalbits zurückgesetzt. Das Ergebnis der Instruktion ist der Zustand des Signalregisters nach der Wartezeit, aber bevor die Signalbits gelöscht werden. Es ist garantiert, dass kein zweiter Kontrollfaden zur gleichen Zeit mit denselben Signalbits aus dem Wartezustand erweckt werden kann.

Beispiel:

Implementierung einer „Kritischen Sektion“

```
EnterCriticalSection:                           ; Signalbit in r4, Rücksprung in r5
               sync.wm        r4, 0, r0
               bra            [r5]

LeaveCriticalSection:                           ; Signalbit in r4, Rücksprung in r5
               setsig.m       r4, 0, r0
               bra            [r5]
```

Benutzung der Sektion

```
               add            r0, SEMAPHORE_1_BIT, r4
               add            r2, 2, r5
               bra            [EnterCriticalSection]
               .
               .
               .
               add            r0, SEMAPHORE_1_BIT, r4
               add            r2, 2, r5
               bra            [EnterCriticalSection]
```

10.1.6.3 SETSIG – Set signal

Kurzbeschreibung: Setzen eines oder mehrerer Signalbits.

Adressierungsarten, Erweiterungen: siehe WAIT

Kodierung:

```
11001[Rd ][Rs ]WM00010ppiiiiiiii
```

Detailinformation:

Die beiden Quelloperanden werden durch ein binäres Oder miteinander verknüpft. Die in diesem Wort gesetzten Bits werden auch im Signalregister gesetzt. Wartet ein Thread auf eines der betroffenen Signale, so wird dieser freigegeben (dessen Warteanweisung ausgeführt). Das Ergebnis der Operation ist der Inhalt des Signalregisters bevor die neuen Bits gesetzt wurden.

Diese Anweisung kann auch verwendet werden, um den Inhalt des Signalregisters zu lesen. Als Argument kann dazu einfach ein Nulloperand angegeben werden.

Beispiel: siehe WAIT oder SETSIG

10.1.6.4 CLRSIG – Clear signal

Kurzbeschreibung: Löschen eines oder mehrerer Signalbits

Adressierungsarten, Erweiterungen: siehe WAIT

Kodierung:

```
11001[Rd ][Rs ]WM00011ppiiiiiiii
```

Detailinformation:

Die beiden Quelloperanden werden durch ein binäres Oder miteinander verknüpft. Die in diesem Wort gesetzten Bits werden im Signalregister zurückgesetzt. Das Ergebnis der Operation ist der Inhalt des Signalregisters bevor die Bits zurückgesetzt wurden.

Diese Anweisung kann auch verwendet werden, um den Inhalt des Signalregisters zu lesen. Als Argument kann dazu einfach ein Nulloperand angegeben werden.

Beispiel: siehe WAIT

10.1.6.5 STOP – Stop thread

Kurzbeschreibung: Beenden eines Kontrollfadens

Adressierungsarten:

```
Rt, Rd
```

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11001[Rd ][Rt ]WM00100xxxxxxxxxx
```

Detailinformation:

Der angegebene Kontrollfaden (in Rt) wird nach der Ausführung der zuletzt von ihm dekodierten Anweisung beendet. Das Ergebnis der Operation ist die Adresse der nächsten Anweisung danach. Dieser Befehl wird erst dann abgeschlossen, wenn der betroffene Kontrollfaden beendet ist. Der Rückgabewert kann benutzt werden, um den Kontrollfaden ohne Datenverlust von der nächsten logischen Adresse aus wieder aufzustarten.

Befand sich der Kontrollfaden zum Befehlsausführungszeitpunkt bereits im beendeten Zustand, so wird der Befehl sofort abgeschlossen. Das Ergebnis ist dann der erste Befehl nach dem letzten ausgeführten Befehl des betroffenen Kontrollfaden.

Beispiel:

Zeitweise Unterbrechung eines anderen Kontrollfadens

```
CallSuspended:                          ; Kontrollfaden in r4, Rücksprung in r5
                                        ; Aufgerufene Routine in r6
              add           r5, 0, r7   ; Rücksprung sichern
              stop.w        r4, r8      ; Anhalten des Konkurenten
              add           r2, 2, r5   ; Eigenen Rücksprung sichern
              bra           [r6]        ; Routine aufrufen
              start.m       r8, r4      ; Konkurenten wieder starten
              bra           [r7]        ; Zurück an Aufrufer
```

10.1.6.6 START – Start thread

Kurzbeschreibung: Starten eines Kontrollfadens

Adressierungsarten:

```
Rs, Rt
```

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11111[Rs ][Rt ]WM00101xxxxxxxxxx
```

Detailinformation:

Der in Rt angegebene Kontrollfaden wird an der Programmadresse in Rs aufgestartet. Befand sich der Kontrollfaden zum Befehlszeitpunkt bereits im laufenden Zustand, so hat dieser Befehl keine Auswirkung.

Beispiel: siehe STOP

10.1.6.7 SUSPEND – Suspend threads

Kurzbeschreibung: Zeitweises Unterbrechen eines oder mehrerer Kontrollfäden

Adressierungsarten:

```
Rm, Rd
```

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11001[Rd ][Rm ]WM00110xxxxxxxxxx
```

Detailinformation:

Die in der Maske (Rm) angegebenen Kontrollfäden (jedem Kontrollfaden entspricht ein Bit in der Maske) werden in ihrer Ausführung unterbrochen. Alle bereits dekodierten Anweisungen werden allerdings noch ausgeführt. Das Ergebnis der Operation ist ein Bitfeld, in dem für jeden Kontrollfaden, der vor dieser Anweisung unterbrochen war, das entsprechende Bit gesetzt ist.

Beispiel: siehe GETID

10.1.6.8 RESUME – Resume threads

Kurzbeschreibung: Fortsetzen unterbrochener Kontrollfäden

Adressierungsarten, Erweiterungen: siehe SUSPEND

Kodierung:

```
11001[Rd ][Rm ]WM00111xxxxxxxxxx
```

Detailinformation:

Die in der Maske (Rm) angegebenen Kontrollfäden werden aus dem unterbrochenen Zustand wieder in den laufenden Zustand überführt. Das Ergebnis der Operation ist ein Bitfeld, in dem für jeden Kontrollfaden, der vor dieser Anweisung unterbrochen war, das entsprechende Bit gesetzt ist.

Beispiel: siehe GETID

10.1.6.9 END – End current thread

Kurzbeschreibung: Beenden des ausführenden Kontrollfadens

Adressierungsarten:

Erweiterungen:

```
.M
```

Kodierung:

```
11111xxxxxxxxxx1M01000xxxxxxxxxx
```

Detailinformation:

Beendet eine Kontrollfaden durch eine eigene Instruktion. Dies ist die ideale Variante um einen Kontrollfaden zu beenden, da dieser sicherstellen kann, dass dies zu einem korrekten Zeitpunkt geschieht. Für diesen Befehl muss immer das W Bit gesetzt sein.

10.1.6.10 KILL – Kill thread

Kurzbeschreibung: Beenden eines Kontrollfadens

Adressierungsarten:

```
Rs
```

Erweiterungen:

```
.M, .W
```

Kodierung:

```
11111[Rs ]xxxxxWM10011xxxxxxxxxx
```

Detailinformation:

Beendet einen anderen Kontrollfaden. Diese Variante einen Kontrollfaden zu beenden ist etwas problematisch, da dem beendeten Kontrollfaden keine Möglichkeit gegeben wird, seine Ressourcen freizugeben.

10.1.6.11 GETID – Get current thread ID

Kurzbeschreibung: Nummer des eigenen Kontrollfadens holen

Adressierungsarten:

Rd

Erweiterungen:

.W, .M

Kodierung:

11001[Rd]xxxxxWM01001xxxxxxxxxx

Detailinformation:

Die Nummer des eigenen Hardware-Ausführungskontextes wird zurückgegeben.

Beispiel:

Kurzzeitiges Unterbrechen aller anderen Kontrollfäden

```
getid          r4             ; Thread ID nach r4
add            r0, 1, r5      ; 1 nach r5
lsl            r5, 4, r5      ; Maske mit Bit # Thread ID gesetzt
xor.b          r5, $ff, r5    ; Invertieren der Maske
suspend.w      r5, r6         ; Unterbrechen aller anderen Threads
andn           r5, r6, r5     ; Maske mit unterbrochenen Threads bilden
.
.
.
resume.m       r6, r0         ; unterbrochene Threads wieder fortsetzen
```

10.1.6.12 NOP – No operation

Kurzbeschreibung: Keine Operation

Adressierungsarten:

Erweiterungen:

.W, .M

Kodierung:

11111xxxxxxxxxxWM01010xxxxxxxxxx

Detailinformation:

Diese Anweisung hat keine direkte Auswirkung, sie kann allerdings durch ihre Seiteneffekte (W/M) Auswirkungen zeigen.

10.1.6.13 SETPRI – Set scheduling priority

Kurzbeschreibung: Setzen der Prioritäten der Ausführungskontexte

Adressierungsarten:

```
Rs, Rd
```

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11001[Rd ][Rs ]WM01100xxxxxxxxxx
```

Detailinformation:

Das bisherige Prioritätsstatuswort wird in das Zielregister geladen. Danach wird das Quellregister mit dem bisherigen Prioritätsstatuswort exklusiv oder verknüpft und in das Prioritätsstatuswort übertragen. Durch diese Verknüpfung ist es möglich, das Statuswort auszulesen ohne es zu ändern.

Beispiel:

Erhöhen der eigenen Ausführungspriorität:

```
RaisePriority:
              setpri        r0, r4
              getid         r5
              add           r0, 1, r6
              lsl           r6, r5, r6
              and           r4, r6, r6
              xor           r4, r6, r6
              setpri        r6, r4
```

10.1.6.14 SETSYS – Set system control word

Kurzbeschreibung: Setzen / Lesen des Systemstatuswortes

Adressierungsarten:

```
Rs, Rd
```

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11001[Rd ][Rs ]WM01101xxxxxxxxxx
```

Detailinformation:

Nicht spezifiziert

10.1.6.15 TRACE – Trace instruction flow

Kurzbeschreibung: Ausgabe eines Programmablaufsmeldung

Adressierungsarten:

```
Imm, Rs
```

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11111[Rs ]xxxxxWM10100xxiiiiiiii
```

Detailinformation:

Die angegebene konstante Meldungsnummer wird zusammen mit dem Inhalt des Quelloperandenregisters auf ein externes Programmablaufsverfolgungsmedium ausgegeben. Dies sind bei dem Simulator das Kommandofenster und eine Programmablaufsverfolgungsdatei.

10.1.6.16 FLUSHD – Flush data cache

Kurzbeschreibung: Heraus schreiben aller veränderten Cache-Zeilen

Adressierungsarten:

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11111xxxxxxxxxxWM10000xxxxxxxxxx
```

Detailinformation:

Alle im Daten-Cache enthaltenen Cache-Zeilen, die in diesem verändert wurden, werden in den externen Speicher übertragen, und die entsprechenden Statusbits gelöscht.

10.1.6.17 INVALD – Invalidate data cache

Kurzbeschreibung: Ungültigmachen des Inhalts des Datencaches

Adressierungsarten:

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11111xxxxxxxxxxWM10001xxxxxxxxxx
```

Detailinformation:

Alle im Datencache enthaltenen Daten werden als ungültig erklärt.

10.1.6.18 INVALI – Invalidate instruction cache

Kurzbeschreibung: Ungültigmachen der Einträge im Instruktions-Caches

Adressierungsarten:

Erweiterungen:

```
.W, .M
```

Kodierung:

```
11111xxxxxxxxxxWM10010xxxxxxxxxx
```

Detailinformation:

Alle im Instruktionscache enthaltenen Instruktionen werden als ungültig erklärt.

10.1.6.19 IN – Input data

Kurzbeschreibung: Einlesen eines Datenwortes von einem Eingabekanal

Adressierungsarten:

```
[d, Rb], Rd
```

Erweiterungen:

Kodierung:

```
11001[Rd ][Rb ]0011000dddddddddd
```

Detailinformation:

Die Zielkanalnummer des Befehls berechnet sich als die Summe aus dem Inhalt des Basisregisters und eines konstanten vorzeichenbehafteten Versatzes. Der Versatz muss kleiner als 2^{10} sein.

10.1.6.20 OUT – Output data

Kurzbeschreibung: Ausgeben eines Datenwortes auf einen Ausgabekanal

Adressierungsarten:

```
Rs, [d, Rb]
```

Erweiterungen:

Kodierung:

```
11111[Rs ][Rb ]0011001dddddddddd
```

Detailinformation:

Die Zielkanalnummer des Befehls berechnet sich als die Summe aus dem Inhalt des Basisregisters und eines konstanten vorzeichenbehafteten Versatzes. Der Versatz muss kleiner als 2^{10} sein.

10.2 Aufbau und Benutzung des Simulators

10.2.1 Aufbau der Konfigurationsdateien

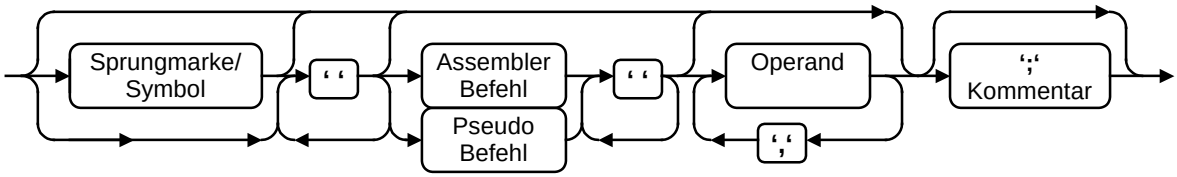
10.2.1.1 Aufbau der Quellcodedateien (xxx.asm)

Der Simulator erwartet den zu simulierenden Programmcode als Assemblerquelltext. Dieser kann je nach Anwendung von einem Compiler oder von Hand erzeugt sein. Aufgrund der Nähe des Prozessors zu einem DSP dürfte er aber typischerweise manuell erstellt sein.

10.2.1.1.1 Aufbau von Quellcodezeilen

Quellcodezeilen können entweder leer sein, oder eine Assembler- oder Pseudoanweisung enthalten. Pseudoanweisungen generieren keinen Programmcode, sondern wirken sich auf die Übersetzung auf. Jede Zeile kann durch einen Kommentar abgeschlossen werden. Ein Kommentar wird hierbei durch ein Semikolon eingeleitet.

Der prinzipielle Aufbau einer Zeile entspricht folgender Struktur:



Folgende Regeln sind zu beachten:

- definiert eine Zeile eine Sprungmarke oder ein Symbol, so muss dieses mit dem ersten Zeichen der Zeile beginnen.
- Vor einem Assembler- oder Pseudobefehl muss mindestens ein Leerzeichen oder Tabulator stehen
- Enthält eine Zeile Operanden, so müssen diese von den Befehlen durch mindestens ein Leerzeichen oder Tabulator getrennt sein
- Enthält eine Zeile mehr als einen Operanden, so werden diese durch Kommata getrennt
- Zeichen, die nach einem Semikolon folgen, werden nicht berücksichtigt

10.2.1.1.2 Symbole und Sprungmarken

Symbole und Sprungmarken folgen einer gemeinsamen Syntax. Jeder dieser Bezeichner muss durch einen Buchstaben eingeleitet werden. Danach kann einen beliebige Sequenz aus Buchstaben, Zahlen und dem Unterstrich ‘_’ folgen. Es werden beliebig lange Bezeichner zugelassen. Alle Buchstaben eines Bezeichners werden bei einem Vergleich herangezogen. Groß-/Kleinschreibung wird bei der Unterscheidung von Symbolen ebenfalls benutzt (allerdings nicht bei der Unterscheidung von Assembler- und Pseudobefehlen).

10.2.1.1.3 Operatoren und Ausdruckauswertung

Der Assembler kann während der Übersetzung arithmetische und logische Ausdrücke auswerten, und die Ergebnisse für die Übersetzung nutzen. Es werden hierbei die meisten in „C“ üblichen Operatoren verwendet. Die Schreibweise von Konstanten ist allerdings stärker an die Schreibweise in Assemblern angelehnt.

Folgende Operatoren sind dem Assembler bekannt:

Symbol	Typ	Priorität	Operation
!	unär	1	Binäres nicht, jedes Bit im Operanden wechselt seinen Wert
-	unär	1	Arithmetische Negation
*	binär	2	Produkt
/	binär	2	Quotient
+	binär	3	Summe
-	binär	3	Differenz
<<	binär	4	binäres Linksschieben
>>	binär	4	binäres Rechtsschieben
==	binär	5	Test auf Gleichheit, liefert 1 falls gleich, sonst 0
!=	binär	5	Test auf Ungleichheit
<=	binär	5	Test auf kleiner oder gleich
>=	binär	5	Test auf größer oder gleich
<	binär	5	Test auf kleiner
>	binär	5	Test auf größer
&	binär	6	Binäre Und-Verknüpfung
|	binär	7	Binäre Oder-Verknüpfung

Operatoren werden in der Reihenfolge steigender Priorität ausgeführt. Operatoren gleicher Priorität werden von links nach rechts ausgewertet.

Numerische Konstanten können zu verschiedenen Basissystemen angegeben werden

- 33254 Dezimalsystem (Basis 10)
- $32af Hexadezimalsystem (Basis 16)
- %11001 Binärsystem (Basis 2)

Dezimalkonstanten können auch als Fixkommawerte angegeben werden. Die Anzahl der binären Nachkommastellen wird jeweils durch ein ‘:’ nach der Zahl angegeben, also z.B. 10.73:8 um die Zahl 10,73 mit acht binären Nachkommastellen anzugeben.

Folgende weitere Elemente können in Ausdrücken verwendet werden:

- **‘*’** Bezeichnet die aktuelle Zieladresse der Assemblierung
- r0, r1,... Bezeichnet die Prozessorregister
- ‘(‘ und ‘)’ Werden verwendet um die Ausführungsreihenfolge im Ausdruck zu ändern

- @0, @1,... Argumente innerhalb einer Makroauswertung

10.2.1.1.4 Assembler Pseudoanweisungen

Pseudoanweisungen sind Anweisungen, die zwar ihrer Syntax nach wie normale Assembleranweisungen verwendet werden, aber nicht in Maschinencode übersetzt werden. Ihre Aufgabe ist die Steuerung der Übersetzung. Sie werden verwendet, um das Schreiben von Assemblerprogrammen zu vereinfachen. Die Aufgaben, die Pseudoanweisungen in einem Assembler besitzen, entsprechen in Hochsprachen den Präprozessoren.

10.2.1.1.4.1 Zuweisung symbolischer Konstanten

Um die Lesbarkeit von Programmen zu erhöhen, sollten Konstanten in Assemblerprogrammen durch symbolische Bezeichner dargestellt werden. Hierzu dient die EQU Anweisung. Sie weist dem Symbol auf der linken Seite den Wert des Ausdrucks auf der rechten Seite zu.

```
Symbol          EQU             Ausdruck
```

Von dieser Definition an kann dann anstelle des Ausdrucks das Symbol im Quelltext verwendet werden. Es können auch Register oder komplette Adressen als Ausdruck verwendet werden. Somit lassen sich also auch symbolische Namen für die Prozessorregister vergeben.

10.2.1.1.4.2 Laufzeitkonstanten und Speicherbelegung

Häufig ist es nötig Speicherbereiche für Konstante und globale Variablen während der Übersetzung zu belegen, und eventuell zu initialisieren. Hierfür dienen folgende Anweisungen:

```
Marke           DATA            Ausdruck1, Ausdruck2, ...
Marke           BSS             Ausdruck
Marke           LOCAL           Ausdruck
```

Die DATA Anweisung belegt und initialisiert einen Speicherbereich. Die einzelnen Datenworte werden in aufsteigender Reihenfolge mit den Ergebnissen der einzelnen Ausdrücke belegt. BSS und LOCAL belegen lediglich globalen bzw. lokalen Speicher, ohne diesen aber zu initialisieren. Es werden jeweils so viele Datenworte belegt, wie der entsprechende Ausdruck angibt.

10.2.1.1.4.3 Strukturdefinition

Um die Lesbarkeit und Programmierbarkeit weiter zu erhöhen, lassen sich auch Strukturen definieren (entspricht den „structs“ in C und C++). Es handelt sich hierbei nicht um echte Strukturen mit eigenem Sichtbarkeitsbereich, sondern lediglich, um eine andere Form der Definition von Symbolen.

```
StruktName      STRUCT
ItemName        DS              Ausdruck
ItemName        DS              Ausdruck
...
                ENDSTRUCT
```

Dem Namen der Struktur wird die Länge zugeordnet, den einzelnen Elementnamen, der Versatz zum Start der Struktur. Die STRUCT Anweisung selbst kann noch durch ein Argument erweitert wer-

den. Dieses bestimmt dann die Länge eines unbenannten führenden Elements, und kann somit zu einer Art Strukturerweiterung benutzt werden.

Hier ein Beispiel für verschiedene Strukturdefinitionen:

```
POINTER         EQU             1
INTEGER         EQU             1

NODE            STRUCT
NS_SUCC         DS              POINTER
NS_PRED         DS              POINTER
                ENDSTRUCT

LIST            STRUCT
LS_HEAD         DS              POINTER
LS_SENTINEL     DS              POINTER
LS_TAIL         DS              POINTER
                ENDSTRUCT

THREAD          STRUCT          NODE
TH_SP           DS              POINTER
TH_RP           DS              POINTER
                ENDSTRUCT

EVENT           STRUCT
EV_FLAGS        DS              INTEGER
EV_PENDING      DS              LIST
                ENDSTRUCT

EV_FLAG_SET     STRUCT
EVF_SET         DS              1
EVF_AUTORESET   DS              1
                ENDSTRUCT
```

Wie am letzten Beispiel sichtbar, können Strukturdefinitionen auch verwendet werden, um einzelne Bits in einem Bitfeld zu nummerieren.

10.2.1.1.4.4 Sichtbarkeitsbereiche

Um Abstraktion und Kapselung zu unterstützen, verfügt der Assembler über die Möglichkeit, geschachtelte Sichtbarkeitsbereiche zu verwalten. Alle Symbole und Marken, die innerhalb eines Sichtbarkeitsbereiches definiert werden, sind nur dort bekannt. Bezeichner innerhalb eines Sichtbarkeitsbereiches, können Bezeichner außerhalb des Sichtbarkeitsbereiches temporär überdecken. Ein Sichtbarkeitsbereich wird durch die Anweisungen PROC und ENDPROC geklammert.

```
Marke           PROC
                ...
LokaleMarke     ...
                ...
                ENDPROC
```

Sichtbarkeitsbereiche können beliebig geschachtelt werden.

10.2.1.1.4.5 Makros

Makros sind das wohl mächtigste Element, um das Schreiben von Assemblerroutinen zu verbessern. Ein Makro ist eine Textschablone, die, nachdem sie definiert wurde, beliebig im Programm verwendet werden kann. Die Leistungsfähigkeit von Makros wird noch durch die Möglichkeit, sie mit Argumenten zu versehen deutlich gesteigert. Ein Makro wird durch die Anweisungen MACRO und ENDMACRO bei seiner Definition geklammert. Der Aufruf erfolgt durch seinen Namen (dem Symbol vor der MACRO Anweisung). Die Argumente, die dem Makro mitgegeben werden, sind diesem durch

die Symbole @1, @2, ... verfügbar. Die Anzahl der Argumente befindet sich in @0. Dies ermöglicht es Makros mit einer variablen Anzahl an Argumenten zu definieren.

```
MacroName       MACRO
                ...
                ENDMACRO

                MacroName       Ausdruck, Ausdruck,...
```

Beispiele für die Definition und Benutzung von Makros, zur Implementierung von Unterroutinen:

```
CALL            MACRO           ; destination
                add             r2, 2, r4
                bra             @1
                ENDMACRO

RETURN          MACRO
                bra             [r4]
                ENDMACRO

OutBytes        PROC            ; r8 : data, r9 : num, r10 : port
loop            ldm             [r8], r11
                out             r11, [r10]
                add             r8, 1, r8
                sub             r9, 1, r9
                bne             r9, [loop, r2]
                RETURN
                ENDPROC

Buffer          DATA            100, 101, 102, 103
BufferSize      EQU             * - Buffer
WritePort       DATA            $ffd2

Main            PROC
                lea             [Buffer, r2], r8
                movl            BufferSize, r9
                ldm             [WritePort], r10
                CALL            [OutBytes]
                end
                ENDPROC
```

Zusammen mit Sichtbarkeitsbereichen können Makros auch zur strukturierten Programmierung eingesetzt werden, hier am Beispiel einer der Kernroutinen eines Primzahlprogramms:

```
; ------------------------------------------------------------------------

MIFNE           MACRO           ; reg
                PROC
                beq             @1, [endOfIf, PC]
                ENDMACRO

MENDIF          MACRO
endOfIf         ENDPROC
                ENDMACRO

DO              MACRO
                PROC
loopStart
                ENDMACRO

WHILELT         MACRO           ; reg
                bge             @1, [loopDone, PC]
                ENDMACRO

ENDDO           MACRO
loopEnd         bra             [loopStart, PC]
loopDone        ENDPROC
                ENDMACRO

; ------------------------------------------------------------------------

NextPrime       PROC            ; r8 : buffer, r9 : size, r10 : previous
                add             r8, r9, r9
                add             r8, r10, r10

                DO
                  add           r10, 2, r10
                  cmp           10, r9, r11
                WHILELT         r11
                  ldm           [r10], r11
                  MIFNE         r11
                    sub         r10, r8, r8
                    RETURN
                  MENDIF
                ENDDO

                movl            0, r8
                RETURN
                ENDPROC
```

10.2.1.1.4.6 Bedingte Übersetzung

Mithilfe der bedingten Übersetzung lassen sich in Abhängigkeit von den Ergebnissen von Ausdrücken die Übersetzung verschiedener Programmteile unterdrücken. Dies wird insbesondere bei der Verwendung von Makros nützlich, da deren Expansion somit durch Argumente des Aufrufs gesteuert werden kann.

```
                IF              Ausdruck
                ...
                ELSIF           Ausdruck
                ...
                ELSIF           Ausdruck
                ...
                ELSE
                ...
                ENDIF
```

Es können beliebig viele „ELSIF“ Fälle verwendet werden. Der ELSE Fall ist wie üblich optional. Durch geschickte Nutzung von Bedingter Übersetzung und Makros, lassen sich auch Rekursionen erzeugen, wie in diesem Makro zum Initialisieren eines Speicherbereichs durch die Replikation eines Wertes:

```
DUP             MACRO           ; val, num
                IF              @2
                DATA            @1
                DUP             @1, @2-1
                ENDIF
                ENDMACRO
```

Oder auch dasselbe mit Divide-and-conquer:

```
DUP             MACRO           ; val, num
                IF              @2 == 1
                DATA            @1
                ELSIF           @2 > 1
                DUP             @1, @2 / 2
                DUP             @1, @2 - @2 / 2
                ENDIF
                ENDMACRO
```

10.2.1.1.4.7 Einbeziehen anderer Dateien

Um ein Projekt in Mehrere Quelldateien zu zerlegen, ist es nötig während der Übersetzung eine andere Quelldatei einzubinden. Dies geschieht hier mit Hilfe der INCLUDE Anweisung. Jede Datei wird während der Übersetzung nur einmal eingebunden, unabhängig davon, wie oft sie durch eine INCLUDE Anweisung aufgerufen wird. Dies hat den Vorteil, dass nicht darauf geachtet werden muss, das durch das Rekursive Einbinden von Quelldateien, Programmteile mehrfach übersetzt werden.

```
                INCLUDE         DateiName
```

Die einzubindende Datei muss sich entweder im selben Verzeichnis, wie die Aufrufende Datei befinden, oder durch den in den Umgebungsvariablen spezifizierten Suchpfad.

10.2.1.1.4.8 Ausgabe von Meldungen während der Übersetzung

Zur Fehlersuche kann es nützlich sein, während der Übersetzung Meldungen auszugeben. Dies ist besonders bei starker Nutzung von Makros und bedingter Übersetzung sehr hilfreich. Für diesen Zweck ist eine ECHO Anweisung vorgesehen. Sie hat keine andere Funktion, als während der Übersetzung eine Meldung auf die Konsole auszugeben, so dass der Fluss der Übersetzung verfolgt werden kann.

```
                ECHO            Meldungstext bis zum Ende der Zeile
```

10.2.1.2 Aufbau der Prozessorkonfigurationsdatei (xxx.cpu)

10.2.1.2.1 Einführung

Die Konfiguration des simulierten Prozessors wird durch eine Textdatei durchgeführt. In dieser Datei werden alle spezifischen Parameter des jeweils zu testenden Prozessors spezifiziert. Das Format der Konfigurationsdatei ist zeilenorientiert. In jeder Zeile wird eine Einheit spezifiziert, oder mehrere Einheiten des Prozessors verknüpft. Wird bei einer Simulation keine Konfigurationsdatei angegeben, so erzeugt der Simulator eine Datei mit Standardparametern. Ausgehend von dieser Datei können dann sehr einfach verschiedene Varianten getestet werden. Es ist für größere Testläufe natürlich empfehlenswert, diese Konfigurationsdateien automatisiert zu erstellen, um ganze Parameterbereiche testen zu können.

10.2.1.2.2 Beispiel einer Konfigurationsdatei

Der folgende Text zeigt eine Standardkonfigurationsdatei, wie sie der Simulator von sich aus erzeugt.

```
Threads 4
Registers 32
Bus Width 2
Global Memory Size 1048576
Burst Size 4 Rate 32 4
Local Memory Size 8192 Use 1
Instruction Cache Lines 512 Associativity 4
Data Cache Lines 512 Associativity 4 Strategy 8 Prefetch 0
Secondary Data Cache Lines 0 Associativity 0 Strategy 0 Prefetch 0
Branch Target Address Cache Size 1024 Associativity 4
Fetch Units 2 Size 4 Strategy 2
Dual Dynamic Branch Prediction Cache Size 2048 Max Level 2 History 8
Decode Units 2 Size 4 Predicts 64
Dispatch 4 Queue Size 32 Look Depth 16 Strategy 2 Threshold 8
Completion 4 Queue Size 16 Look Depth 16 Strategy 0 Threshold 12
Rename Registers 64
Register Writeback 4
Load Store Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2 Overtake 1 Predicted 0
Load Store Reservation Stations 1 Queue Size 32 Accept 2 Dispatch 2 Strategy 2 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2 Overtake 1
Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2
Reservation Stations 1 Queue Size 32 Accept 4 Dispatch 4 Strategy 2
Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2
Result Busses 4
Load Store Units 1 Pending 8 Strategy 0
Local Load Store Units 2
Thread Control Units 1 Pending 16
Branch Units 1
Integer Units 3
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 1
Link Unit 2 to Reservation Station 2
Link Unit 3 to Reservation Station 3
Link Unit 4 to Reservation Station 4
Link Unit 5 to Reservation Station 4
Link Unit 6 to Reservation Station 4
Link Unit 7 to Reservation Station 5
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 1
Link Unit 2 to Result Bus 2
Link Unit 4 to Result Bus 0
Link Unit 5 to Result Bus 2
Link Unit 6 to Result Bus 3
Link Unit 7 to Result Bus 3
```

10.2.1.2.3 Struktur und Elemente der Konfigurationsdatei

Jede Zeile der Konfigurationsdatei ist entweder eine Kommentarzeile (erkennbar durch ein einleitendes „//“) oder eine Konfigurationsanweisung. Jede Konfigurationsanweisung spezifiziert eine oder mehrere gleichartige Einheiten, oder aber die Verbindung mehrerer Einheiten untereinander. Der Aufbau der einzelnen Konfigurationsanweisungen ist fest, sie können nicht in sich verschoben oder vertauscht werden, auch ist es nicht zulässig, dass eine Konfigurationsdatei über mehrere Zeilen verläuft. Die Reihenfolge der Konfigurationsanweisungen untereinander ist unbestimmt. Sind widersprüchliche Anweisungen in einer Konfigurationsdatei enthalten, so überstimmt die spätere Anweisung die frühere. Fehlerhafte Anweisungen werden durch eine entsprechende Fehlermeldung des Simulators gemeldet.

10.2.1.2.3.1 Konfigurationsanweisungen für die allgemeine Prozessortstruktur

Mit diesen Anweisungen wird die generelle Struktur des Prozessors spezifiziert.

- **Threads %1**

  spezifiziert die Anzahl paralleler Ausführungskontexte (%1), die im Prozessor enthalten sind. Dieser Parameter bestimmt die Anzahl der Registerbänke, als auch die Breite der Kontroll-Pipeline.

- **Registers %1**

  spezifiziert die Anzahl der Register (%1) pro Ausführungskontext. Dieser Parameter sollte stets auf 32 gesetzt sein, um die Kompatibilität unter den verschiedenen simulierten Prozessoren zu erhalten.

10.2.1.2.3.2 Konfigurationsanweisungen für Speicherhierarchie

In diesem Anweisungsblock werden die verschiedenen Speicher, Datenbusse und Caches des Prozessors spezifiziert.

- **Bus Width %1**

  spezifiziert die Breite des Datenbusses in Prozessorworten (%1). Dieser Parameter hat einen direkten Einfluß auf die Größe einer Cache-Zeile, und die Länge eines Speicherzugriffes in Prozessorworten.

- **Global Memory Size %1**

  spezifiziert die Größe des externen Prozessorspeichers (%1). Dieser Parameter hat keine direkte Auswirkung auf die Simulation, er sollte aber stets so bemessen sein, dass das getestete Programm und seine Daten im Speicher Platz finden.

- **Burst Size %1 Rate %2 %3**

  spezifiziert die Anzahl der durchgeführten Datenoperationen (%1) während eines Speicherblockzugriffes (Speicherzugriffsfolge). Des weiteren wird die Dauer des ersten (%2) und aller weiteren (%3) Zugriffe während dieses Transfers festgelegt. Da in einem Speicherblockzugriff jeweils eine ganze Cache-Zeile transferiert wird, legt dieser Parameter zusammen mit der Datenbusbreite die Länge einer Cache-Zeile in Prozessorworten fest. Kann durch eine besonders breite Busstruktur eine komplette Cache-Zeile in einem Zugriff übertragen werden, hat der dritte Parameter dieser Konfigurationsanweisung keine Bedeutung.

- **Local Memory Size %1 Use %2**

  spezifiziert die Größe des lokalen Speichers (%1). Da ein Programm die Größe des lokalen Speichers anhand von auftretenden Spiegelungen erkennen kann, kann dieser Parameter je nach verwendetem Programm eine Auswirkung auf die Simulation haben. Bei Programmen, die die Größe des lokalen Speichers nicht beachten empfiehlt es sich, diesen Parameter großzügig zu bemessen. Soll kein lokaler Speicher benutzt werden, kann durch (%2) spezifiziert werden, dass alle Zugriffe auf lokalen Speicher auf globalen umgelegt werden sollen.

- **Instruction Cache Lines %1 Associativity %2**
- **Data Cache Lines %1 Associativity %2 Strategy %3 Prefetch %4**
- **Secondary Data Cache Lines %1 Associativity %2 Strategy %3 Prefetch %4**

  spezifizieren die Anzahl der Cache-Zeilen (%1) und der Assoziativität (%2) des Instruktions- bzw. der Datencaches. Die Größe der Caches in Prozessorworten ergibt sich als das

Produkt aus Datenbusbreite, Speicherblocktransferlänge, Cache-Zeilenlänge und Assoziativität. Soll also z.B. die Assoziativität eines Caches erhöht werden, ohne dessen Größe zu verändern, muss entsprechend einer der anderen Parameter verkleinert werden. Für die Daten-Caches kann außerdem auch noch die Zeilenersetzungsstrategie und ein eventuelles spekulatives Vorabladen gewählt werden.

10.2.1.2.3.3 Konfigurationsanweisungen für die Kontroll-Pipeline

In diesem Anweisungsblock werden die Größe und Anzahl der einzelnen Elemente der Kontroll-Pipeline spezifiziert.

- **`Branch target address cache Size %1 Associativity %2`**

  spezifiziert die Größe (%1) und die Assoziativität (%2) des Branch-Target-Address-Caches. Die effektive Größe diese Caches ergibt sich aus dem Produkt der beiden Parameter. Soll also bei gleichbleibender Größe die Assoziativität erhöht werden, so muss der Parameter für die Größe entsprechend vermindert werden.

- **`Static Branch Prediction`**
- **`Single Dynamic Branch Prediction Cache Size %1 Max Level %2`**
- **`Dual Dynamic Branch Prediction Cache Size %1 MaxLevel %2 History %3`**

  spezifiziert die Art der Sprungvorhersage. Bei dynamischer Vorhersage kann die Größe des Sprungvorhersage-Caches (%1), so wie der maximal Wert des saturierenden Zählers (%2) gesetzt werden. Für die erweiterte Sprungvorhersage kann auch die Länge des Sprungereignisregisters bestimmt werden (%3).

- **`Fetch Units %1 Size %2 Strategy %3`**

  spezifiziert die Breite in Befehlen (%2) und die Anzahl (%1) der Instruktionsladeeinheiten. Die Breite ist hierbei als die maximale Anzahl an Befehlen, die für einen Ausführungskontext pro Takt bereitgestellt werden kann, zu sehen. Dies ist ein Maximalparameter, da dieser Wert aufgrund der Cache-Zeilenanordnung nicht unbedingt erreicht wird. Weiterhin kann auch die Auswahlstrategie bestimmt werden (%3) nach welcher der nächste Ausführungskontext gewählt wird, für den Befehle geladen werden.

- **`Decode Units %1 Size %2 Predicts %3`**

  spezifiziert die Breite in Befehlen (%1) und die Anzahl (%2) der Instruktionsdekodiereinheiten. Die Breite ist hierbei als die maximale Anzahl an Befehlen, die für einen Ausführungskontext pro Takt dekodiert werden kann zu sehen. Die Breite der Instruktionsdekodiereinheit, sollte stets der Breite der Instruktionsladeeinheit entsprechen. Weiterhin kann die Anzahl der spekulativen Programmverzweigungen angegeben werden (%3), die gleichzeitig verfolgt werden können.

- **`Dispatch %1 Queue Size %2 Look Depth %3 Strategy %4 Threshold %5`**

  spezifiziert die Maximalanzahl der bearbeiteten Instruktionen pro Takt (%1), sowie die Länge der Bereitstellungswarteschlange (%2) und der maximalen Bearbeitungstiefe pro Ausführungskontext (%3) der Befehlszuordnungseinheit. Ist nur ein Ausführungskontext aktiv, so wird die Maximalanzahl der bearbeiteten Instruktionen durch das Minimum aus (%1) und (%3) limitiert. Weiterhin kann auch die Auswahlstrategie (%4) und der Füllstand für ausreichend gefüllt (%5) angegeben werden.

- **`Completion %1 Queue Size %2 Look Depth %3 Strategy %4 Threshold %5`**

  Spezifiziert die Maximalanzahl der bearbeiteten Instruktionen pro Takt (%1), sowie die Länge der Bereitstellungswarteschlange (%2) und der maximalen Bearbeitungstiefe pro Ausführungskontext (%3) der Befehlsrückordnungseinheit. Weiterhin kann auch die Auswahlstrategie (%4) und der Füllstand für ausreichend gefüllt (%5) angegeben werden.

10.2.1.2.3.4 Konfigurationsanweisungen für die Umbenennungsregister und Datenpfade

- **`Rename Registers %1`**

  Spezifiziert die Anzahl der Umbenennungspufferregister (%1).

- **`Register Writeback %1`**

  spezifiziert die maximale Anzahl an Umbenennungspufferregister, die pro Takt in die allgemeinen Register zurückgeschrieben werden können (%1).

- **`Result Busses %1`**

  spezifiziert die Anzahl der Resultatsbusse (%1) im Prozessor. Da sich mehrere Ausführungseinheiten einen Resultatsbus teilen können, kann dieser Wert kleiner als die Anzahl an Ausführungseinheiten sein. Die Verbindung der einzelnen Resultatsbusse mit den Ausführungseinheiten wird durch Verknüpfungskommandos erreicht.

10.2.1.2.3.5 Konfigurationsanweisungen für die Befehlsumordnungspuffer

Diese Anweisungen spezifizieren die jeweilige Anzahl, Größe und Struktur der drei verschiedenen Arten an Befehlsumordnungspuffern, die im simulierten Prozessor Verwendung finden.

- **`Reservation Stations %1 Queue Size %2 Accept %3 Dispatch %4 Strategy %5`**

  spezifiziert die Anzahl (%1) an und Länge (%2) der allgemeinen Befehlsumordnungspuffer. Des weiteren wird die Maximalanzahl an Anweisungen spezifiziert, die jeder dieser Puffer pro Takt in der Lage ist anzunehmen (%3) oder an die Ausführungseinheiten abzugeben (%4). Weiterhin kann auch die Auswahlstrategie der Puffer (%5) bestimmt werden.

- **`Load Store Reservation Stations %1 Queue Size %2 Accept %3 Dispatch %4 Strategy %5 Overtake %6 Predicted %7`**

  spezifiziert die Anzahl (%1) an und Länge (%2) der Befehlsumordnungspuffer für Lade- und Speichereinheiten. Des weiteren wird die Maximalanzahl an Anweisungen spezifiziert, die jeder dieser Puffer pro Takt in der Lage ist anzunehmen (%3) oder an die Ausführungseinheiten abzugeben (%4). Weiterhin können auch die Auswahlstrategie der Puffer (%5), die Sequenzialisierungsbedingung zwischen den Kontrollfäden (%6) und die Möglichkeit zur Ausführung spekulativer Ladeanweisungen (%7) angegeben werden.

- **`Thread Control Reservation Stations %1 Queue Size %2 Accept %3 Dispatch %4 Strategy %5 Overtake %6`**

  spezifiziert die Anzahl (%1) an und Länge (%2) der Befehlsumordnungspuffer für Kontrolleinheiten. Des weiteren wird die Maximalanzahl an Anweisungen spezifiziert, die jeder dieser Puffer pro Takt in der Lage ist anzunehmen (%3) oder an die Ausführungseinheiten abzugeben (%4). Weiterhin können auch die Auswahlstrategie der Puffer (%5) und die Sequenzialisierungsbedingung zwischen den Kontrollfäden (%6) bestimmt werden.

10.2.1.2.3.6 Konfigurationsanweisungen für die Ausführungseinheiten

- **`Load Store Units %1 Pending %2 Strategy %3`**
  spezifiziert die Anzahl der Lade- und Speichereinheiten (%1). Dieser Parameter sollte stets den Wert eins haben. Weiterhin kann die Länge der Cache-Fehlzugriffswarteschlange (%2) und der Auswahlstrategie (%3) angegeben werden.
- **`Local Load Store Units %1`**
  spezifiziert die Anzahl der Lade- und Speichereinheiten für den lokalen Speicher (%1).
- **`Thread Control Units %1 Pending %2`**
  spezifiziert die Anzahl an Kontrolleinheiten (%1) und der Länge der Warteschlange (%2) für Ausführungskontexte in diesen Einheiten. Die Anzahl sollte stets eins betragen und die Länge der Warteschlange mindestens so viele Einträge umfassen, wie Ausführungskontexte vorhanden sind.
- **`Branch Units %1`**
  spezifiziert die Anzahl an Sprungausführungseinheiten (%1)
- **`Integer Units %1`**
  spezifiziert die Anzahl einfacher Integerausführungseinheiten (%1).
- **`Complex Integer Units %1 Pipeline Size %2`**
  spezifiziert die Anzahl (%1) und Ausführungs-Pipeline-Länge (%2) der Ausführungseinheiten für komplexe Integeranweisungen.

10.2.1.2.3.7 Konfigurationsanweisungen für die Verknüpfung der Einheiten

Mit folgenden Anweisungen werden die Ausführungseinheiten mit den Befehlsumordnungspuffern und den Resultatsbussen verbunden. Die Ausführungseinheiten und Befehlsumordnungspuffer werden hierbei fortlaufend durchnummeriert. Dies geschieht bei den Befehlsumordnungspuffern in der Reihenfolge: lade- und speicherspezifische, kontrollspezifische, allgemeine. Bei den Ausführungseinheiten wird folgende Reihenfolge verwendet: global Lade- und Speichereinheit, lokale Lade- und Speichereinheit, Kontrolleinheit, Sprungeinheit, einfache Integereinheiten, komplexe Integereinheiten.

- **`Link Unit %1 to Reservation Station %2`**
- **`Link Unit %1 to Result Bus %2`**
  Verbindet die Ausführungseinheit (%1) mit dem Befehlsumordnungspuffer bzw. Ergebnisbus (%2). Jede Ausführungseinheit darf nur mit einem Befehlsumordnungspuffer und Ergebnisbus verbunden werden. Wird eine Ausführungseinheit mehrfach verbunden, so gilt die letzte der Ausgeführten Verbindungen.

10.2.1.2.4 Beispielkonfigurationen für verschiedene Prozessortypen

In diesem Abschnitt werden einige Beispielkonfigurationen für verschiedene am Markt befindliche oder ältere Prozessoren gezeigt. Die Konfigurationsdateien wurden hierbei auf das wesentliche beschränkt.

10.2.1.2.4.1 Motorola Power PC (604/ 620)

Diese Prozessorfamilie zeichnet sich durch die explizite Zuordnung der Befehlsumordnungspuffern (Reservation-stations) zu den Ausführungseinheiten aus.

```
Fetch Size 4 Units 1
Decode Size 4 Units 1
Dispatch 4 Queue Size 8 Look Depth 4
Completion 4 Queue Size 12 Look Depth 4
Rename Registers 12
Register Writeback 3
Load Store Reservation Stations 1 Queue Size 2 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 2 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Reservation Stations 2 Queue Size 2 Accept 1 Dispatch 1 Strategy 0
Result Busses 3
Load Store Units 1
Local Load Store Units 1
Thread Control Units 1 Pending 4
Branch Units 1
Integer Units 2
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 0
Link Unit 2 to Reservation Station 1
Link Unit 3 to Reservation Station 2
Link Unit 4 to Reservation Station 2
Link Unit 5 to Reservation Station 3
Link Unit 6 to Reservation Station 3
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 0
Link Unit 2 to Result Bus 0
Link Unit 4 to Result Bus 1
Link Unit 5 to Result Bus 2
Link Unit 6 to Result Bus 2
```

10.2.1.2.4.2 Pentium MMX

Dies ist ein klassischer In-order aber zweifach superskalarer Prozessor. Um dieses Verhalten zu simulieren, wird als einziger Befehlsumordnungspuffer ein Thread-Kontroll-Puffer verwendet, der zur seriellen Abarbeitung der Anweisungen zwingt.

```
Fetch Size 2 Units 1
Decode Size 2 Units 1
Dispatch 2 Queue Size 2 Look Depth 2
Completion 2 Queue Size 6 Look Depth 2
Rename Registers 4
Register Writeback 2
Load Store Reservation Stations 0 Queue Size 2 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 2 Accept 2 Dispatch 2 Strategy 0 Overtake 1 Predicted 0
Reservation Stations 0 Queue Size 2 Accept 1 Dispatch 1 Strategy 0
Result Busses 2
Load Store Units 1
Local Load Store Units 1
Thread Control Units 1 Pending 4
Branch Units 1
Integer Units 2
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 0
Link Unit 2 to Reservation Station 0
Link Unit 3 to Reservation Station 0
Link Unit 4 to Reservation Station 0
Link Unit 5 to Reservation Station 0
Link Unit 6 to Reservation Station 0
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 0
Link Unit 2 to Result Bus 0
Link Unit 4 to Result Bus 0
Link Unit 5 to Result Bus 1
Link Unit 6 to Result Bus 1
```

10.2.1.2.4.3 Pentium Pro/ Pentium II

Es handelt sich hier um einen typischen Vertreter der dynamischen superskalaren Prozessoren mit einem großen Befehlsumordnungspuffer. Die Zuordnung einer Anweisung zu einer Ausführungseinheit findet erst im letzten Takt vor der Ausführung statt.

```
Fetch Size 3 Units 1
Decode Size 3 Units 1
Dispatch 3 Queue Size 3 Look Depth 3
Completion 3 Queue Size 24 Look Depth 3
Rename Registers 24
Register Writeback 3
Load Store Reservation Stations 1 Queue Size 24 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 24 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Reservation Stations 1 Queue Size 24 Accept 2 Dispatch 2 Strategy 0
Result Busses 3
Load Store Units 1
Local Load Store Units 1
Thread Control Units 1 Pending 4
Branch Units 1
Integer Units 2
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 0
Link Unit 2 to Reservation Station 1
Link Unit 3 to Reservation Station 2
Link Unit 4 to Reservation Station 2
Link Unit 5 to Reservation Station 2
Link Unit 6 to Reservation Station 2
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 0
Link Unit 2 to Result Bus 0
Link Unit 4 to Result Bus 1
Link Unit 5 to Result Bus 2
Link Unit 6 to Result Bus 2
```

10.2.1.2.5 Konfigurationen der wichtigsten simulierten Prozessoren

10.2.1.2.5.1 Maximalprozessor

```
Threads 8
Registers 32
Bus Width 2
Global Memory Size 1048576
Burst Size 4 Rate 6 2
Local Memory Size 8192 Use 1
Instruction Cache Lines 32768 Associativity 4
Data Cache Lines 32768 Associativity 4 Strategy 0 Prefetch 0
Secondary Data Cache Lines 0 Associativity 0 Strategy 0 Prefetch 0
Branch Target Address Cache Size 1024 Associativity 4
Fetch Units 8 Size 8 Strategy 0
Dual Dynamic Branch Prediction Cache Size 2048 Max Level 2 History 8
Decode Units 8 Size 8 Predicts 64
Dispatch 8 Queue Size 32 Look Depth 16 Strategy 0 Threshold 8
Completion 8 Queue Size 256 Look Depth 16 Strategy 0 Threshold 12
Rename Registers 1024
Register Writeback 8
Load Store Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Load Store Reservation Stations 1 Queue Size 256 Accept 2 Dispatch 2 Strategy 0 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0 Overtake 1
Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0
Reservation Stations 1 Queue Size 256 Accept 4 Dispatch 4 Strategy 0
Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0
Result Busses 8
Load Store Units 1 Pending 8 Strategy 0
Local Load Store Units 2
Thread Control Units 1 Pending 16
Branch Units 1
Integer Units 4
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 1
Link Unit 2 to Reservation Station 2
Link Unit 3 to Reservation Station 3
Link Unit 4 to Reservation Station 4
Link Unit 5 to Reservation Station 4
Link Unit 6 to Reservation Station 4
Link Unit 7 to Reservation Station 4
Link Unit 8 to Reservation Station 5
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 1
Link Unit 2 to Result Bus 2
Link Unit 4 to Result Bus 3
Link Unit 5 to Result Bus 4
Link Unit 6 to Result Bus 5
Link Unit 7 to Result Bus 6
Link Unit 8 to Result Bus 7
```

10.2.1.2.5.2 Maximalprozessor mit realistischem Speichersystem

```
Threads 8
Registers 32
Bus Width 2
Global Memory Size 1048576
Burst Size 8 Rate 32 4
Local Memory Size 8192 Use 1
Instruction Cache Lines 32768 Associativity 4
Data Cache Lines 256 Associativity 4 Strategy 8 Prefetch 0
Secondary Data Cache Lines 0 Associativity 4 Strategy 8 Prefetch 1
Branch Target Address Cache Size 1024 Associativity 4
Fetch Units 8 Size 8 Strategy 0
Dual Dynamic Branch Prediction Cache Size 2048 Max Level 2 History 8
Decode Units 8 Size 8 Predicts 64
Dispatch 8 Queue Size 32 Look Depth 16 Strategy 0 Threshold 8
Completion 8 Queue Size 32 Look Depth 16 Strategy 0 Threshold 12
Rename Registers 1024
Register Writeback 8
Load Store Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0 Overtake 1 Predicted 0
Load Store Reservation Stations 1 Queue Size 256 Accept 2 Dispatch 2 Strategy 0 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0 Overtake 1
Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0
Reservation Stations 1 Queue Size 256 Accept 4 Dispatch 4 Strategy 0
Reservation Stations 1 Queue Size 256 Accept 1 Dispatch 1 Strategy 0
Result Busses 8
Load Store Units 1 Pending 8 Strategy 0
Local Load Store Units 2
Thread Control Units 1 Pending 16
Branch Units 1
Integer Units 4
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 1
Link Unit 2 to Reservation Station 2
Link Unit 3 to Reservation Station 3
Link Unit 4 to Reservation Station 4
Link Unit 5 to Reservation Station 4
Link Unit 6 to Reservation Station 4
Link Unit 7 to Reservation Station 4
Link Unit 8 to Reservation Station 5
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 1
Link Unit 2 to Result Bus 2
Link Unit 4 to Result Bus 3
Link Unit 5 to Result Bus 4
Link Unit 6 to Result Bus 5
Link Unit 7 to Result Bus 6
Link Unit 8 to Result Bus 7
```

10.2.1.2.5.3 Prozessor mit realistischer Ressourcenverteilung

```
Threads 8
Registers 32
Bus Width 2
Global Memory Size 1048576
Burst Size 8 Rate 32 4
Local Memory Size 8192 Use 1
Instruction Cache Lines 256 Associativity 4
Data Cache Lines 256 Associativity 4 Strategy 8 Prefetch 0
Secondary Data Cache Lines 0 Associativity 0 Strategy 0 Prefetch 0
Branch Target Address Cache Size 1024 Associativity 4
Fetch Units 2 Size 4 Strategy 2
Dual Dynamic Branch Prediction Cache Size 2048 Max Level 2 History 8
Decode Units 2 Size 4 Predicts 64
Dispatch 8 Queue Size 32 Look Depth 16 Strategy 2 Threshold 8
Completion 8 Queue Size 16 Look Depth 16 Strategy 0 Threshold 12
Rename Registers 64
Register Writeback 4
Load Store Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2 Overtake 1 Predicted 0
Load Store Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2 Overtake 1 Predicted 0
Thread Control Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2 Overtake 1
Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2
Reservation Stations 1 Queue Size 32 Accept 4 Dispatch 4 Strategy 2
Reservation Stations 1 Queue Size 32 Accept 1 Dispatch 1 Strategy 2
Result Busses 4
Load Store Units 1 Pending 8 Strategy 0
Local Load Store Units 1
Thread Control Units 1 Pending 16
Branch Units 1
Integer Units 3
Complex Integer Units 1 Pipeline Size 5
Link Unit 0 to Reservation Station 0
Link Unit 1 to Reservation Station 1
Link Unit 2 to Reservation Station 2
Link Unit 3 to Reservation Station 3
Link Unit 4 to Reservation Station 4
Link Unit 5 to Reservation Station 4
Link Unit 6 to Reservation Station 4
Link Unit 7 to Reservation Station 5
Link Unit 0 to Result Bus 0
Link Unit 1 to Result Bus 1
Link Unit 2 to Result Bus 2
Link Unit 4 to Result Bus 0
Link Unit 5 to Result Bus 2
Link Unit 6 to Result Bus 3
Link Unit 7 to Result Bus 3
```

10.2.1.3 Eingabedatendatei (xxx.pin)

Für die Ein-/Ausgabedatei werden künstliche Eingaben während der Programmausführung benötigt. Diese werden für jeden simulierten Eingabekanal in einer einzelnen Datei gehalten. Bei jedem Eingabebefehl wird ein Datenwort aus dieser Eingabedatei gelesen und an die Kontrolleinheit übergeben. Diese Datei wird in binärer Form gelesen.

10.2.2 Aufbau der Ergebnisdateien

Der Simulator produziert während und am Ende seiner Ausführung mehrere Ergebnisdateien. Diese können direkt oder durch ein weiteres Programm ausgewertet werden.

10.2.2.1 Ausführungsverfolgungsdatei (xxx.trc)

Durch Ausführungsverfolgungsbefehle (TRACE) kann ein Programm über seine Ausführung Informationen geben. Diese werden während der Simulation auf der Konsole angezeigt, und in der Ausführungsverfolgungsdatei gesammelt, so dass sie nach erfolgter Simulation überprüft werden können.

10.2.2.2 Ausführungsstatusdatei (xxx.sys)

In dieser Datei wird der aktuelle Zustand der Simulation gespeichert. Ein Simulationslauf kann einen der folgenden vier Zustände haben:

- Nicht gestartet
- Läuft
- Fertig
- Abgebrochen

Mit diesen Dateien lässt sich eine parallele Ausführung mehrerer Simulationen auf verschiedenen Rechnern einfach überprüfen. Neben dem Zustand der Simulation ist auch der Name des simulierenden Rechners in der Datei vorhanden.

10.2.2.3 Ausführungsprofildatei (xxx.prf)

Die Ausführungsprofildatei enthält für jeden Programmteil die Anzahl der in ihm ausgeführten Instruktionen sowie die Instruktionsmischung. Weiterhin sind alle Instruktionen, die mehr als 0,01 Prozent der Ausführungszeit benötigen zusammen mit ihrer anteiligen Ausführungszeit gelistet. Zudem werden noch alle falsch spekulierten Sprünge und Cache-Fehlzugriffe die 0,01 Prozent überschreiten aufgeführt.

Mit dieser Datei können einzelne Anweisungen gesucht werden, die besonders an der Programmausführung beteiligt sind, oder sehr hohe Fehlspekulationen besitzen.

10.2.2.4 Speicherzuordnungs- und Übersetzungsdatei (xxx.lst)

Diese Datei entsteht während der Übersetzung. Sie enthält das gesamte übersetzte Programm in Maschinen- und Quellcode. Mit ihrer Hilfe kann von einer Programmadresse auf die Quellanweisung zurückgeschlossen werden. Weiterhin kann die korrekte Funktion des Assemblers überprüft werden.

10.2.2.5 Speicherauszug des lokalen Speichers (xxx.dmp)

Am Ende der Programmausführung oder des Simulationslaufes wird der gesamte Inhalt des lokalen Speichers in diese Datei geschrieben. Mit ihrer Hilfe kann die korrekte Programmausführung überprüft werden.

10.2.2.6 Gesamtprofildatei (xxx.out)

Die Gesamtprofildatei enthält alle während der Simulation anfallenden globalen Messwerte. Dazu gehören die gesamte Anzahl an Instruktionen die geladen, dekodiert, zugeordnet, ausgeführt und rückgeordnet wurden. Weiterhin werden die Auslastungen sämtlicher Ausführungseinheiten sowie der Caches protokolliert.

Zusätzlich werden auch die Gründe für nicht genutzte Ausführungsmöglichkeiten innerhalb der einzelnen Pipelinestufen gesammelt, und mit ihrer anteiligen Beteiligung ausgegeben.

10.2.2.7 Fehlerprotokoll- und Systemzustandsdatei (xxx.core)

Sollte das simulierte Programm eine unerlaubte Anweisung oder einen fehlerhaften Speicherzugriff versuchen, so wird es abgebrochen, und eine Fehlerprotokoll- und Systemzustandsdatei erzeugt. Diese enthält neben der Fehlerursache den gesamten Systemzustand. Dazu gehören die Register aller Ausführungskontexte, die Umbenennungsregister, die Befehlspuffer aller Pipelinestufen sowie den lokalen Speicher.

10.2.2.8 Ausgabedatendatei (xxx.pout)

Die Ausgaben, die die Ein-/Ausgabedatei während der Programmausführung produziert, werden für jeden Ausgabekanal getrennt in einer Ausgabedatei gesammelt. Anhand dieser Dateien lässt sich das Ergebnis des Simulators und seines Lastprogramms überprüfen. So erzeugt der MPEG-Dekoder vollständig dekodierte Bilder im YUV Format.

10.2.3 Durchführung einer Messserie

Da meistens eine ganze Serie von Prozessoren gemeinsam simuliert und analysiert werden soll, der Simulator aber nur jeweils eine einzelne Konfiguration simulieren kann, wurden einige Hilfsprogramme geschaffen, um eine Serie von Messungen durchführen zu können.

Dazu wird zuerst eine Prozessorkonfigurationsdatei erzeugt, die alle gemeinsamen Simulationsparameter der Messserie enthält. Danach wird mit Hilfe eines Programms für jede zu simulierende Konfiguration eine einzelne Konfigurationsdatei erzeugt. Zudem wird noch eine Kontrolldatei erzeugt, die die Namen aller erzeugten Konfigurationen enthält. Diese Konfigurationsdatei wird von einem weiteren Programm benutzt, um nacheinander alle diese Konfigurationen durch den eigentlichen Simulator zu simulieren. Dieses Programm kann auf beliebig vielen Rechnern gleichzeitig gestartet sein und ermöglicht so eine Beschleunigung des Messvorganges durch Parallelisierung.

Ist die Serie komplett durchgeführt, so können die einzelnen Ausgabedateien mit einem weiteren Programm in eine einzelne Ergebnisdatei konvertiert werden.

10.3 Abkürzungsverzeichnis

AC ..... Alternating Current
AC3 ..... Audio Coding Format 3
ADC ..... Analog to Digital Converter
ALU ..... Arithmetic Logic Unit
B-Frame ..... Bidirectional Predictive Coded Frame
BHR ..... Branch History Register
BHT ..... Branch History Table
BLIT ..... Block Image Transfer
BTAC ..... Branch Target Address Cache
CCIR ..... International Radio Consultative Committee
CD ..... Compact Disc
CPI ..... Cycles Per Instruction
DAC ..... Digital to Analog Converter
DC ..... Direct Current
D-Cache ..... Data Cache
DCT ..... Discrete Cosinus Transformation
DSP ..... Digital Signal Processor
DTV ..... Digital Television
DVB ..... Digital Video Broadcast
DVD ..... Digital Versatile Disc
FFT ..... Fast Fourier Transform
GOP ..... Group Of Picture
HDTV ..... High Definition Digital Television
I-Cache ..... Instruction Cache
ID ..... Identifier
ID ..... Instruction Decode
IDCT ..... Inverse Discrete Cosinus Transformation
IF ..... Instruction Fetch
I-Frame ..... Intra Coded Frame
IPC ..... Instructions Per Cycle
JPEG ..... Joint Photographers Expert Group
L/S ..... Load/Store
LRU ..... Least Recently Used
MAC ..... Multiply Accumulate
MMX ..... Multi Media Extension
MPEG ..... Motion Pictures Expert Group
NTSC ..... National Television Broadcast System
PAL ..... Phase Alternating Line
PC ..... Personal Computer
PC ..... Program Counter
PEL ..... Picture Element
P-Frame ..... Predictive Coded Frame

PTS..... Presentation Time Stamp
RGB..... Red Green Blue
RI .....Rename and Issue
RISC .....Reduced Instruction Set Computer
RT ..... Retirement
SIMD..... Single Instruction Multiple Data
SMP.....Symetric Multi Processing
VLIW.....Very Large Instruction Word
VOD..... Video On Demand
WB.....Writeback
YIQ ..... (color components in NTSC color space)
YUV .....(color components in SECAM and PAL color spaces

10.4 Quellenangaben

[1] „Intels Architecture Software Developer's Manual", http://developer.intel.com/design/PentiumII/manuals, 1997

[2] „MMX Developers Guide", http://developer.intel.com/drx/mmx/manuals/dg/devguide.htm, 1997

[3] „MMX Technology Technical Overview", http://developer.intel.com/drx/mmx/manuals/overview, 1996

[4] „Pentium II Processors Developer's Manual", http://developer.intel.com/design/PentiumII/manuals, Oktober 1997

[5] „TriMedia, TM1000 Preliminary Data Book", Philips Electronics North America Corporation, 1997

[6] Agarwal, A., Lim, B.-H. , Kranz, D. and Kubiatowicz, J.. "April: A processor architecture for multiprocessing". In Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 104-114, May 1990.

[7] Arai, Y., Agui, T., Nakajima, M.: "*A Fast DCT-SQ Scheme for Images*", Trans. of the IEICE.E 71(11):1095(Nov.1988)

[8] Birgham, E.O., "The Fast fourier Transform", 1974 Englewood Cliffs, NJ: Prentice-Hall

[9] Bittnar, K., Grünewald, W., Ungerer, T.: „Entwurf einer vielfädigen Prozessorarchitektur zum Einsatz in Distributed-Shared-Memory-Systemen"

[10] Bradford, J. P. and Abraham, S. "Efficient Synchronization for Multithreaded Processors", in Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC), January 31-February 4, 1998. Workshop held in conjunction with the Fourth International Symposium on High-Performance Computer Architecture (HPCA).

[11] Burns, J., Gaudiot, J.-L.,: "Quantifying the SMT Layout Overhead – Does SMT Pull its Weight? " in Proceedings of the sixth International Symposium on High-Performance Computer Architecture HPLA, Jan. 8^{th} –12^{th} 2000 Toulouse Frankreich

[12] Chen, T.-F., Baer, J.-L. : „Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Transactions on Computers, Vol. 44, No. 5, May 1995, S. 609-623

[13] Chou, Y. C., Siewwiorek, D. P., Shen, J. P.: „A realistic Study on Multithreaded Superscalar Processor Design", Euro-Par 97, Passau April 1997

[14] Chromatic Research: „Media Processors: An Introduction", http://www.mpact.com/tech/, 1997

[15] Eggers, Emer, Levy, Lo, Stamm and Tullsen: „Simultaneous Multithreading: A Platform for Next-Generator Processors", in IEEE Micro, October 1997

[16] Emer, J.: „Simultaneous Multithreading: Multiplying Alpha's Performance", Microprocessor Forum, Oct 5 - 6, 1999

[17] Feig, E., Linzer, E.: "*Discrete Cosine Transform Algorithms for Image Data Compression*", Proceedings Electronic Imaging '90 East, pp.84-7, Boston, MA(Oct.29 - Nov.1,1990)

[18] Foley, P.: „The MPACT Media Processor Redefines the Multimedia PC", in IEEE Proceedings of COMPCON '96, 1996, S. 311-318

[19] Frigo, M., Leiserson, C. E., Randall, K. H.: "The Implementation of the Cilk-5 Multithreaded Language," 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), June 17-19, Montreal, Canada.

[20] G. Tyson and M. Farrens, "Techniques for Extracting Instruction Level Parallelism on MIMD Architectures" , Proceedings of the 26th Annual International Symposium on Microarchitecture, Austin, Texas (December 1-3, 1993)

[21] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, "A Modified Approach to Data Cache Management" , Proceedings of the 28th Annual International Symposium on Microarchitecture Ann Arbor, MI (Nov 29-Dec 1, 1995).

[22] García, M., González, J., Gonzáles, A.: "Data Caches for Multithreaded Processors"

[23] Goldstein, S. C., Schmitt, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R. R., Laufer, R.: „PipeRench: A Coprocessor for Streaming Multimedia Acceleration". ISCA 1999

[24] Greenley, D. et al., „UltraSPARC™ : The Next Generation Superscalar 64-bit SPARC", IEEE Compcon Spring 1995, S. 442 – 451

[25] Gwennap, L.: „MAJC Gives VLIW a New Twist", Microprocessor Report, Volume 13, Number 12, September 13, 1999

[26] Halfhill, T. R.: „Intel Network Processor targets Routers", Microprocessor Report, Volume 13, Number 12, September 13, 1999

[27] Halstead, R. H., Jr. and Fujita, T.." MASA: A multithreaded processor architecture for parallel symbolic computing". In Proceedings of the 15th Annual International Symposium on Computer Architecture, pages 443-451, June 1988.

[28] Higgins, R.: „Digital Signal Processings in VLSI", Prentice Hall 1990

[29] Huffman, D.A. 1952, Proceedings of the Institute of Radio Engineers, vol.40, pp. 1098-1101

[30] Hunt, D., "Advanced Performance Features for the 64-bit PA-8000", in Proceedings of the 40th IEEE Computer Society International Conference COMPCON 95, March 1995, San Francisco, CA, 123-128

[31] Hunt, D., „Advanced Performance Features of the 64-bit PA-8000", Compcon Spring '95, 5. – 9.3.95 San Francisco

[32] Iannucci, R. A. "Toward a dataflow/von Neumann hybrid architecture". In Proc. Int. Symp. Comput. Arch., pages 131-140, June 1988.

[33] ISO/IEC JTC1/SC29/WG11 „Generic Coding of Moving Pictures and Associated Audio Information : Video"

[34] J. Fu, J. H. Patel, and B. L. Janssens, "Stride Directed Prefetching in Scalar Processors," Int. Symp. on Micro-Architecture, Portland, OR, Dec. 1992.

[35] Kahle, J.: „Power4: A Dual-CPU Processor Chip", Microprocessor Forum, Oct 5 - 6, 1999

[36] Kalapathy, P., „Hardware-Software Interactions on MPACT", IEEE Micro, March/April 1997 S. 20 - 26

[37] Kleiman, S.; Eykholt, J.: Interrupts as Threads. Operating Systems Review, Vol.29, No.2, pp. 21-26, April 1995.

[38] Koelbel, C., Loveman, D., Schreiber, R., Steele Jr., G., and Zosel, M.: „The High Performance Fortran Handbook." MIT Press, 1994.

[39] Kohn, L., Maturana, G., Tremblay, M., Prahbu, A., Zyner, G., „The Visual Instruction Set (VIS) in Ultra SPARC™", IEEE Compcon Spring 1995, S. 462 - 469

[40] Krishnan, V., Torrellas, J. : "A Chip-Multiprocessor with Speculative Multithreading" in IEEE Transactions on Computers, Vol. 48 No. 9, Sept. 1999

[41] Kumar, A., „The HP PA-8000 RISC CPU", IEEE Micro, March/April 1997 S. 27 - 32

[42] L. Lo, J., Barroso, A., Eggers, S. J., et. Al. : "An Analysis of Database Workload Performance on Simulataneous Multithreaded Processors", ISCA 25 Proceedings, June 27th-July 1st 1998, Barcelona, Spain

[43] Laudon, J., Gupta, A., Horowitz, M.: "Interleaving: A Multihreading Technique Targeting Multiprocessors and Workstations". ASPLOS VI- 10/94 San Jose, California USA, S. 308-318

[44] Lee, Ruby B., „Realtime MPEG Video via Software Decompression on a PA-RISC Processor", IEEE Compcon Spring 1995, S. 186 – 192

[45] Lo, Eggers, Emer, Levy, Stamm and Tullsen : „Converting Thread-Level Parallelism to Instruction Level Parallelism via Simultaneous Multithreading" in ACM Transactions on Computer Systems, August 1997

[46] LSI LOGIC, „L64702 JPEG Coprocessor, Technical Manual", 1993

[47] M. Farrens and A. Pleszkun, "Strategies for Achieving Improved Processor Throughput" , Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada (May 27-30, 1991).

[48] McFarling, S. “Combining Branch Predictors.” DEC WRL Technical Note TN-36, DEC Western Research Laboratory 1993

[49] Motorola Inc, „MC88110 Second Generation RISC Microprocessor User’s Manual“

[50] Mowry, T. C. : "Tolerating Latency Through Software-Controlled Data Prefetching" Ph.D. Thesis Computer Systems Laboratory Stanford University, CA 94305 March, 1994

[51] Oehring, H., Sigmund, U., Ungerer, T. : "Performance of Simultaneous Multithreaded Multimediaenhanced Processors for MPEG-2 Video Decompression", Journal of Systems Architecture

[52] Oehring, H., Sigmund, U., Ungerer, Th.: “Simultaneous Multithreading and Multimedia”. Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC 99) in conjunction with Fifth International Symposium on High Performance Computer Architecture (HPCA-5), Orlando, 9.-12.1.1999.

[53] Oehring, H., Sigmund, U., Ungerer, Th.: „Evaluierung mehrfädiger Prozessortechniken zur Weiterentwicklung von Multimediaprozessoren“. 15. GI/ITG-Fachtagung ARCS’99 Architektur von Rechensystemen 1999, Jena 4.-7.10.1999.

[54] Oehring, H., Sigmund, U., Ungerer, Th.:” MPEG-2 Video Decompression on Simultaneous Multithreaded Multimedia Processors”. 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99), Newport Beach, Ca., 12.-16.10.1999.

[55] Palarcharla, S., Kessler, R.E.: Evaluating Stream Buffers As a Secondary Cache Replacement. 21st Int. Symp. on Computer Architecture. April 1994, 24-33

[56] PCI Special Interest Group: „PCI LOCAL BUS SPECIFICATION, Revision 2.1“, 1995

[57] Philippsen, M., Herter, C. G., and Tichy, W. F.: “The Triton project”. Technical Report No. 33/90, University of Karlsruhe, Department of Informatics, December 1990.

[58] Philippsen, M., Tichy, W.F.: ” Modula-2* and its compilation”. In First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991, pages 169{183. Springer Verlag, Lecture Notes in Computer Science 591, 1992

[59] Ranganathan, P., Adve, S., Jouppi, N. P. : “Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions”, ISCA 1999

[60] Rathnam, S., Slavenburg, G., „An Architectural Overview of the Programmable Multimedia Processor, TM-1“, Compcon, S. 319 – 326, Spring 1996

[61] S. Lam, M., P. Wilson, R.: “Limits of Control Flow on Parallelism”, in 19th Annual International Symposium on Computer Architecture, 1992 S. 46-57

[62] S. Wallace, B. Calder, and D. Tullsen, Threaded Multiple Path Execution, *25th International Symposium on Computer Architecture*, June 1998.

[63] Sailer, P. M., Kaeli, D. R.: „The DLX Instruction Set Architecture Handbook“, 1996 by Morgan Kaufmann Publishers, Inc. ISBN 1-55860-371-9

[64] Sedgewick, R.: „Algorithmen in C++“, Addison Wesley 1992

[65] Seminar „Multimedia-Prozessoren“, Theo Ungerer, Institut für Rechnerentwurf und Fehlertoleranz Fakultät für Informatik, Universität Karlsruhe, Januar 1998

[66] Sigmund, U., Ungerer, Th.: “Evaluating a Multithreaded Superscalar Microprocessor Versus a Multiprocessor Chip”. Proceedings of the "4th PASA - Workshop - Parallel Systems and Algorithms", Forschungszentrum Jülich, 10.-12. April 1996, World Scientific Publishing 1996, Seite 147 - 159.

[67] Sigmund, U., Ungerer, Th.: “Identifying Bottlenecks in a Multithreaded Superscalar Processor.” Bougé, L. et al. (Hrsg.): Proceedings of the "Europar 96 Conference", Lyon, 26.-29. August 1996, Band 1, Springer-Verlag, Lecture Notes in Computer Science 1123, Seite 797 - 800.

[68] Sigmund, U., Ungerer, Th.: “Memory Hierarchy Studies of Multimedia-Enhanced Simultaneous Multithreaded Processors for MPEG-2 Decompression”, Workshop on Multi-Threaded Execution, Architecture and Compilation, Januar 2000 in Toulouse, France

[69] Sigmund, U., Ungerer, Th.: „Ein mehrfädiger superskalarer Mikroprozessor“. GI/ITG Workshop PARS, Physikzentrum Bad Honnef, 13. - 15. Mai 1996, PARS-Mitteilungen, Band 15, Seite 75 - 84.

[70] Smith, B. J. “Architecture and applications of the HEP multiprocessor computer system”. SPIE, 298:241-248, 1981.

[71] Steffan, J. G., Colohan, C. B. and Mowry. T. C. ,"Extending Cache Coherence to Support Thread-Level Data Speculation on a Single Chip and Beyond." Technical Report CMU-CS-98-171, School of Computer Science, Carnegie Mellon University, December 1998.

[72] Stroustrup, B.: „The C++ Programming Language, second edition“, Addison Wesley 1991

[73] Texas Instruments: TMS320C80 (MVP) Master Processor User’s Guide (spru109a); www.ti.com, 1995

[74] Texas Instruments: TMS320C80 (MVP) Multitasking Executive User’s Guide (spru112a); www.ti.com, 1995

[75] Texas Instruments: TMS320C80 (MVP) Parallel Processor User’s Guide (spru110a); www.ti.com 1995

[76] Texas Instruments: TMS34010 Users Guide (SPVU001A)

[77] Texas Instruments: TMS34020 Users Guide (SPVU019)

[78] Tremblay, M.: „MAJC-5200: A VLIW Converget MPSOC“, Microprocessor Forum Oct. 5-6, 1999

[79] Tsai, J-Y., Huang, J., Amlo, C., Lilja, D. J., Yew, P-C.: “The Superthreaded Processor Architecture” in IEEE Transactions on Computers, vol. 48, No. 9, Sept. 1999

[80] Tseng, B.D., Miller, W.C.: “*On Computing the Discrete Cosine Transform”*, IEEE Trans. on Computers, C-27(10):966-8(Oct. 1978)

[81] Tullsen, Eggers and Levy: „Simultaneous Multithreading: Maximizing On-Chip Parallelism“ in ISCA95

[82] Tullsen, Eggers, Emer, Levy, Lo and Stamm: „Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreaded Processor“ in ISCA96

[83] Unger, A., Zehendner, E., Ungerer, Th.: „A Combined Compiler and Architecture Technique to Control Multithreaded Execution of Branches and Loop Iterations”, in Workshop on Interaction between Compiler and Computer Architecture, Toulouse, Jan. 9th 2000

[84] Wallace, S., Bagherzadeh, N. :”Instruction Fetching Mechanisms for Superscalar Microprocessors“, *Euro-Par '96*, August 1996.

[85] Weber, W.-D., Gupta, A.: “Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results”. In Proceedings of the 16th Annual International Symposium on Computer Architecture, pages 273-280, June 1989.

[86] Wong, W. F., Goto, E.: “A Simulation Study on the Interactions Between Multithreaded Architectures and the Cache”, in International Journal on High Speed Computing 6(2)

[87] Yamauchi, T, Hammnd, L., Olukotun, K.: „A Single Chip Multiprocessor Intgrated with DRAM“. Technical Report CSL-TR-731, Computer Systems Laboratory, Stanford University

[88] Yeh, T.-Y., Patt, Y.N. “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History”. In Proceedings of the 20th Annual International Symposium on Computer Architecture May 1993, San Diego, CA, 257-266

[89] Yeh, T.-Y., Patt, Y.N. “Alternative Implementation of Two-Level Adaptive Branch Prediction” in Proceedings of the 19th Annual Symposium on Computer Architecture, May 1992, Gold Coast, 124-134

[90] Zhou, C., Kohn, L., Rice, D., Kabir, I., Jabbi, A., Hu, X., „MPEG Video Decoding with the UltraSPARC Visual Instruction Set“, IEEE Compcon Spring 1995, S. 470 – 475

[91] ZORAN Corp, „I11 MPEG 1 SYSTEM and VIDEO DECODER, Technical Specification“, 1994, http:\\www.zoran.com

10.5 Lebenslauf

Zur Person:

Name:	Ulrich Sigmund
Geburtsdatum:	29. Dezember 1968
Geburtsort:	Freiburg im Breisgau
Staatsangehörigkeit:	Deutsch

Schulbildung:

1975 -1979	Weiherhof Grundschule in Freiburg
1979 -1988	Droste-Hülshoff-Gymnasium in Freiburg
1988	Abitur (Note 1,2)
1988 - 1989	Grundwehrdienst

Hochschulstudium:

WS 89/90 - WS 95/96	Studium der Informatik (Diplom) an der Universität Karlsruhe
1995	Studienabschluss als Diplom-Informatiker (Note 1,3)

Berufliche Tätigkeit:

1995 -	Geschäftsführer der „VIONA Development Hard- & Software Engineering GmbH“[38] Karlstr. 27, D-76133 Karlsruhe

[38] Seit dem 1.1.1998 ist der Firmenname geändert in: „VIONA Development Hard- & Software Engineering GmbH & Co. KG.”