Designprinzipien

moderner Prozessoren

Fachhochschule Darmstadt

Günther Fröhlich

1 Einleitung

Seit einigen Jahren wird bei der Einführung neuer Computer als Grund für die enorme Leistungssteigerung (hauptsächlich im Bereich der Arbeitsplatzrechner) auf die neue **RISC** Architektur verwiesen. ***R**educed **I**nstruction **S**et **C**omputer* sind ihren Vorläufern, den ***C**omplex **I**nstruction **S**et **C**omputern*, kurz **CISC**, in vielen Bereichen weit überlegen.

Das ist auf den ersten Blick doch verblüffend: Computer mit kleinem Befehlssatz arbeiten schneller als solche mit großem Maschinenbefehlssatz? Warum hat man dann nicht schon immer RISC gebaut, wo weniger Befehle in einer Maschine doch sicherlich einfacher zu implementieren sind als viele?

Zur Klärung dieser Fragen wird in Kapitel "**Vom CISC zum RISC"** nach der Klärung der in den 70er Jahren angeführten Argumente für große Befehlssätze und die daraus resultierenden Designprinzipen auf die mittlerweile geänderten technologischen Randbedingungen eingegangen und auf Performance Analysen, deren Ergebnisse die Reduzierung der Befehlssätze befürworten. Da die Optimierung der RISC-Prozessoren auf eine möglichst hohe Performance zielt (das war bei Mikroprozessoren bis Mitte der 80er Jahre nicht üblich!) wird die Problematik der Performance-Berechnung eingehend erörtert (ideale Performance, reale Performance, Performance Einflußfaktoren, Entwicklung der Performance-Formel)

Das RISC Postulat, in jedem Taktzyklus einen Befehl vollständig auszuführen, muß jeden Informatikstudenten erstaunen, der Rechnergrundlagen und Microprozessortechnik aufmerksam verfolgt hat. Braucht doch selbst der noch recht einfache, wenig komplexe Intel 8085 vier bis 18 Taktzyklen, um einen Befehl auszuführen Aufgrund welcher Prinzipien dieses Postulat heute tatsächlich realisiert werden kann, wird in den Kapiteln "**Die RISC-Philosophie"** und **"RISC Architektur Merkmale"** erläutert**.**

Im folgenden Kapitel "**Pipelining"** wird das allgemeine Prinzip der Fließbandverarbeitung zur Performance-Steigerung vorgestellt, wie es sowohl bei CISC als auch bei RISC Prozessoren eingesetzt wird. Es soll dabei deutlich werden, daß die RISC Prinzipien ein einfacheres und effektiveres Pipelining erlauben, da auf Grund der RISC-Architektur Daten-, Steuer- und strukturelle Konflikte leichter vermieden werden können. Aktuelle Schlagworte, wie **Superpipeline-Architektur**, **Superskalare-Architektur, Scoreboarding**, **O**ut **of Order Execution** und **Register Renaming** werden in diesem Zusammenhang mit Inhalt gefüllt. Dabei wird sich auch herausstellen, daß sie zwar aktuell, aber nicht unbedingt neue Erfindungen sind.

Damit schnelle Prozessoren nicht nur sehr schnell warten (bis der nächste Befehl zur Ausführung vorliegt), sind diese auf sehr schnelle assoziative **CACHE-Speicher** angewiesen. Das Grundlegende zur modernen "**Speicherarchitektur"** finden Sie im gleichnamigen Kapitel.

Das Postulat, in jedem Taktzyklus einen Befehl auszuführen, ist bei höheren Programmiersprachen nicht ohne Optimierungsstrategien der Compiler zu realisieren. Diese werden in den Kapiteln "**Befehlspipeline und optimierende Compiler**" sowie "**RISC-Compiler"** angesprochen.

Wie soll man es eigentlich beurteilen, daß eine neue Prozessor-Generation unter dem Namen Reduced Instruction Set Computer sich immer breiter in allen Bereichen durchzusetzen beginnt und gleichzeitig erhebliche Verunsicherung über den Begriff RISC existiert? So spricht man bei INTEL von RISC nicht als Architektur, sondern als Designmethode. Müller-Schloer meint RISC mit Regular Instruction Set Computer am besten zu benennen, Märtin spricht bei RISC am liebsten von Reduced Instruction Set Cycle und Giloi hält RISC für „die

wahrscheinlich irreführendste Bezeichnung, die je in der Computertechnik eingeführt wurde“[1] Die Vorlesung bemüht sich, etwas Licht in dieses begriffliche Dunkel zu bringen,.

Im weiteren Verlauf der Veranstaltung werden aktuelle **RISC-Prozessoren** (DEC Alpha 21064, Power-PC ....) und vielleicht ausgewählte aktuelle **RISC-Workstation Produkte** vorgestellt. Zur klaren Systematisierung von Architekturmerkmalen wird auf die INTEL 80X86 CISC Reihe in einem Exkurs eingegangen. Denn auch diese Prozessoren haben schon sehr früh Pipelines (8086), eine integrierte Memory Management Unit (80286) später interne Cache-Speicher(80486) und superskalare Architektur (Pentium) eingesetzt.

Ein wichtiges Lernziel dieser Veranstaltung ist es zu verstehen, inwiefern die traditionelle CISC Architektur, die bis heute ebenfalls weiterentwickelt wird (z.B. INTEL Pentium I - IV) dieselben Methoden der Performance-Verbesserung einsetzt wie die konkurrierenden RISC, diese Methoden bei RISC aber effizienter eingesetzt werden können.

Die dieser Veranstaltung im wesentlichen zugrunde liegende Literatur ist:

- **C. Müller-Schloer, E. Schmitter (Hrsg) RISC-Workstation-Architekturen Springer Verlag, 1991**
- **John L. Hennessy, David A. Patterson: Rechnerarchitektur - Analyse, Entwurf, Implementierung, Bewertung, Vieweg 1994.**

Der Originaltitel des Werkes von Hennesy und Patterson ist *Computer Architecture. A quantitative Approach.* Dieser Titel weist etwas deutlicher darauf hin, daß Hennessy und Patterson grundlegende quantitative Befehlssatzanalysen durchgeführt haben, deren Ergebnisse die Entwicklung zu RISC unterstützten. Der Name RISC wurde von Patterson eingeführt.

Alle Bilder und Tabellen sind diesen Werken entnommen, soweit keine andere Angabe erfolgt.

Etwas aktueller sind die Werke:

- **Christian Märtin, Rechnerarchitektur - Struktur, Organisation, Implementierungstechnik, Hanser Studienbücher der Informatik, München 1994**
- **Wolfgang K. Giloi, Rechnerarchitektur, 2. Auflage, Springer-Verlag, Berlin Heidelberg 1993**

[1] Giloi, Rechnerarchitektur, Springer-Verlag, Berlin Heidelberg 1993 S. 89

2 Historischer Abriß

Bei der Beschäftigung mit RISC fällt auf, daß eine vergleichsweise einfache Architektur sich historisch *nach* einer komplexen durchgesetzt hat. Die Gründe dafür werden in den nächsten Kapiteln eingehend erläutert. Dieser Abschnitt widmet sich einem kurzen historischen Abriß der Computerentwicklung und zeigt, daß RISC keine Erfindung der 80er Jahre ist, aber in dieser Zeit neu beurteilt wurde und deshalb an Bedeutung gewann.

2.1 IBM System/360 - Beginn der CISC Ära

Im Jahre 1964 kam die IBM /360 auf den Markt. Die Bezeichnung /360 steht für die 360 Grad eines Kreises und versinnbildlicht, daß diese Rechnerfamilie konzipiert wurde, um alle Anwendungsbreiche, vom kommerziellen bis zum technisch-wissenschaftlichen Bereich, abzudecken.

Die Idee war, eine *Familie* von Maschinen mit der gleichen Architektur zu entwickeln, die dieselbe Software betreiben können. Das war zu dieser Zeit einmalig und radikal. IBM war auch damals die führende Firma in der Computerbranche und hatte fünf unterschiedliche Architekturen vor der 360. Die Entwickler verstanden unter Rechnerarchitektur „die Struktur eines Computers, die ein Maschinensprache-Programmierer verstehen muß, um ein korrektes (zeitunabhängiges) Programm für diese Maschine zu schreiben.“[1] Das bedeutet, daß Softwarekompatibilität auch in einer Assemblersprache erreicht werden soll, für die es unterschiedliche Implementierungen geben kann.

„Die einzelnen /360 Modelle unterschieden sich in Struktur und Implementierungstechnik. Damit konnte ein Produktspektrum von preiswerten, langsameren bis hin zu leistungsstarken, aber teuren Hochleistungscomputern angeboten werden. Alle Rechner innerhalb einer Familie verfügten über einen identischen Befehlssatz, nutzten die gleichen Peripheriegeräte und konnten durch Emulation sogar die Kompatibilität zu früheren IBM-Modellen gewährleisten. Unterschiede in Struktur und Implementierung wurden durch Mikroprogrammierung überbrückt.“[2]

Die 360 war eine Universalregistermaschine, verwendete Byteadressierung auf Basis von 8-Bit-Bytes, hatte Register-Speicher und einige Speicher-Speicher Befehle. (Siehe auch Exkurs: Register-Register, Register-Speicher und Speicher-Speicher Architekturen)

IBM begründete mit der /360 die Entwicklung kompatibler Computergenerationen mit breitem Leistungsspektrum und universellem Befehlssatz und damit die Architekturphilosophie des *C*omplex *I*nstruction *S*et *C*omputers (CISC).

2.2 CDC 6600 - Vorläufer von RISC

Ebenfalls im Jahre 1964 lieferte Control Data den ersten Supercomputer aus, die CDC 6600. Chef der Entwicklungsabteilung war damals Seymour Cray, der spätere Gründer der Firma CRAY Research, die durch die Entwicklung von Hochleistungsrechnern bekannt ist. T.J. Watson von IBM sagte damals: „*At that point the System /360 was the most advanced set of designs we had, and nothing in the whole product plan was even re-*

---

1 Hennessy, Patterson, a.a.O. S 128

2 Märtin, Rechnerarchitektur, a.a.O. S. 104

motely comparable to the 6600"[1] Man kann die 6600 als Vorbild für alle nachfolgenden Supercomputer-, RISC- und Superskalar-Architekturen bezeichnen. Sie war die erste universelle Lade/Speicher Maschine (siehe auch hierzu den Exkurs: Register-Register, Register-Speicher und Speicher-Speicher Architekturen), hatte eine Hardwaresteuerung und einen kompakten Befehlssatz. Die Entwickler erkannten schon damals die Notwendigkeit, die Architektur zu vereinfachen um ein effektives Pipelining zu erreichen. „Diese Wechselbeziehung zwischen Einfachheit der Architektur und der Implementierung wurde während der 70er Jahre durch Mikroprozessor- und Compilerentwickler stark vernachlässigt und in den 80er Jahren wieder aufgegriffen."[2]

Ende der 60er entdeckte man die sogenannte Softwarekrise, die darin bestand, daß die Softwarekosten schneller anstiegen als die Hardwarekosten. Als Grund dafür gab man an, daß aufgrund unvollkommener Compiler und begrenzter Speicherkapazität der Maschinen zu dieser Zeit die meisten Systemprogramme immer noch in Assemblersprache geschrieben wurden. Viele Forscher schlugen damals vor, die Softwarekrise durch Schaffung leistungsfähigerer softwareorientierter Architekturen zu überwinden.

2.3 VAX 11/780 - Mit CISC aus der Softwarekrise

Die Entwickler der VAX versuchten der Softwarekrise Rechnung zu tragen, in dem sie sehr viel Wert darauf legten, die Übersetzung von Hochsprachen zu vereinfachen. Ziel war es, eine Architektur zu entwerfen, die die Zuordnung von Hochsprachenanweisungen zu einfachen VAX Befehlen erlaubt. Man versuchte die Codelänge zu optimieren, da übersetzte Programme häufig für den verfügbaren Speicherplatz zu groß waren. Als die VAX 1978 eingeführt wurde, war sie der erste Rechner mit echter Speicher-Speicher Architektur (auch das wird erläutert im Exkurs: Register-Register, Register-Speicher und Speicher-Speicher Architekturen)

2.4 High Level Language Computer Architecture (HLLCA)

Während die VAX entwickelt wurde, gab es einen radikaleren Ansatz zur Bewältigung der Softwarekrise. Die Idee bestand darin, die Hardware auf das Niveau von Programmiersprachen zu bringen. Also den Befehlssatz so zu erweitern, daß die Maschine ohne Compiler Anweisungen einer Hochsprache interpretieren und ausführen kann. Damit könnte man auch die sogenannte „semantische Lücke" (siehe dazu den Exkurs: Spracharchitekturen - Stackorientierte Rechner) zwischen Programmiersprache und Hardware schließen um so eine höhere Fehlerfreiheit beim Lauf der Programme zu erzielen .

Das Projekt SYMBOL war der größte und berühmteste der HLLCA Versuche. Das Ziel war eine HLL-, Timesharing Maschine, die die Programmierzeit dramatisch verringern sollte. Die SYMBOL-Maschine interpretierte in ihrer eigenen neuen Programmiersprache geschriebene Programme direkt, d.h. der Compiler und das Betriebssystem wurden direkt in Hardware realisiert. Die Nachteile wurden aber schnell offensichtlich: Die Programmierer hatten keine Wahl der Programmiersprache, nachfolgende Fortschritte bei Betriebssystemen und Programmiersprachen konnten nicht einbezogen werden und die Maschine war schwer zu entwerfen und auszutesten. Es gab erhebliche Leistungsprobleme. Während exotische Fälle relativ schnell liefen, gab es bei einfachen und allgemeinen Fällen oft Leistungsprobleme, weil viele Speicherzugriffe notwendig waren, um einfache Anweisungen eines Programms zu interpretieren.

---

[1] Märtin, Rechnerarchitektur, a.a.O. S. 115

[2] Hennessy, Patterson, a.a.O. S 128

HLLCA hatte niemals einen nennenswerten kommerziellen Einfluß. Das Codelängenproblem, das sich durch das Schreiben von Compilern und Betriebssystemen in höheren Programmiersprachen ergab, wurde durch die Erhöhung der Speichergröße und den Einsatz virtueller Speicher beseitigt.

2.5 RISC - Rückkehr zu einfacher Architektur

In den frühen 80er Jahren wandte man sich wieder von HLLCA ab. Patterson und Ditzel analysierten 1980 die durch HLL hervorgerufenen Probleme und kamen zu dem Schluß, daß die Lösung in einfacheren Architekturen liegt. In einem anderen Artikel diskutierten die Autoren im gleichen Jahr die Idee des *R*educed *I*nstruction *S*et *C*ompters

Bereits in den sechziger Jahren hatte Control Data mit der CDC 6600 bewiesen, daß durch Reduktion der Anzahl von Maschinenbefehlen, Vereinfachung der Befehlsformate, Beschränkung der Adressierungsmodi und festverdrahtete Steuerung erhebliche Performance-Gewinne zu erzielen waren. Später erschütterten die Compilerbau-Praxis und Studien der CISC-Hersteller IBM und DEC die These, wonach komplexere Befehlssätze den Bau besserer Compiler unterstützen sollten. Es zeigte sich, daß sich die Laufzeit des Codes, den der Compiler erzeugte, verbesserte, wenn man ihm nur einen elementaren Bruchteil der verfügbaren Befehle des Befehlssatzes zur Codegenerierung überließ. Hochsprachencompiler, die in der Praxis entwickelt wurden, nutzten aus Gründen der Portierbarkeit und zur Vermeidung unnötiger Compiler-Komplexität ohnehin nur die Befehle, die einfach anzuwenden waren, und sich auf den Maschinen der unterschiedlichen Hersteller strukturell möglichst wenig voneinander unterschieden.

Für die RISC Entstehung waren drei Forschungsprojekte wegbereitend:

- Bei IBM arbeitete ein team bereits 1975 an der Entwicklung eines Rechners mit RISC-Eigenschaften. 1979 wurde der erste Prototyp des *IBM 801* in ECL-Technik mittlerer Integrationsdichte fertiggestellt. Er erzielte bereits eine Leistung von 13 MIPS. Das Projekt war auf die Compiler-Unterstützung höherer Programmiersprachen ausgerichtet und lieferte einen optimierenden PL/8 Compiler. Die Ursprünge der späteren RISC-Architekturen von IBM gehen zum Teil auf dieses Projekt zurück.
- In Berkeley hatten Patterson und Sequin im Jahre 1980 mit der Entwicklung des *Berkeley RISC* begonnen, einem kompakten Prozessor mit nur 39 Instruktionen, dessen Prototyp 1982 von einer hauptsächlich aus Studenten gebildeten Projektgruppe in NMOS VLSI-Technik fertiggestellt wurde. Die Ursprünge der SPARC-Architektur von SUN gehen teilweise auf die Berkeley-Architektur zurück.
- Das dritte Projekt wurde in Stanford von Hennessy 1981 begonnen. Der Prototyp *Stanford MIPS* wurde ebenfalls in NMOS VLSI-Technik realisiert. Die Architektur wurde später von der neugegründeten Firma MIPS für kommerzielle CPUs weiterentwickelt.[1]

Heute sind gerade die ehemals führenden CISC Hersteller wie IBM, DEC, Intel und Motorola mit eigenen RISC-Entwicklungen sehr erfolgreich. DEC z.B. realisierte mit der Alpha 21x64 den leistungsfähigsten Mikroprozessor der Welt. IBM, Apple und Motorola entwickelten den bereits sehr erfolgreichen POWER-PC. Intel baute den RISC-Prozessor 80860 und setzt seit dem 80486 zunehmend mehr RISC Elemente in ihren CISC Prozessoren ein. Einen ähnlichen Weg geht die Firma AMD, die schon 1987 den RISC AMD 29000 fertigstellte, bei der Herstellung Pentium-kompatibler Prozessoren (AMD K5, K6 und Athlon). Sie erscheinen nach außen als typische CISC, sind intern jedoch als RISC-Prozessoren strukturiert.

---

[1] Märtin, Rechnerarchitektur, a.a.O. S. 160

3 Complex Instruction Set Computer

3.1 Argumente für große Befehlssätze

Im Folgenden werden die Argumente aufgeführt, die für Rechnerarchitekten der 60er und 70er Jahre galten und die CISC Philosophie legitimierten. Diese Argumente sind nicht selbstverständlich einzusehen, sondern erfordern eine eingehende Betrachtung um sie zu verstehen und auf allgemeine und zeitbedingte Richtigkeit prüfen zu können.

1. Komplexe Befehle vereinfachen den Compiler. Begründung: die optimale Ausnutzung vieler Register ist schwierig. Im Falle von stack- oder speicherorientierten Architekturen wird mehr Komplexität unterhalb der Maschinenprogrammebene (im Mikrocode) verborgen.
2. Komplexe Befehle schließen die semantische Lücke zwischen Hochsprache und Maschinensprache. Fernziel war eine Maschine, die direkt Hochsprachen verarbeiten konnte.
3. Komplexe Befehle verringern den Speicherbedarf.
4. Je kürzer der Code ist, desto schneller läuft das Programm.
5. Je mehr Funktionalität unterhalb der Maschinensprache angesiedelt wird, desto zuverlässiger wird ein Computer, denn: Hardware ist sicher, Software enthält Fehler.

Zur Beurteilung des ersten Arguments ist zu klären, was *komplexe Befehle* eigentlich ausmacht. Sie können nur in einer sogenannten Speicher-Speicher Architektur realisiert werden. Der folgenden Exkurs klärt den Unterschied zwischen komplexen und einfachen Befehlen und zeigt, inwiefern damit bereits eine Architekturentscheidung getroffen ist.

3.2 Exkurs: Register-Register, Register-Speicher und Speicher-Speicher Architekturen

Zur Begriffsbestimmung dieser drei Ausführungsmodelle (Architekturen) dient das Beispiel: A:=B+C.

Register-Register Architekturen sind Ihnen wahrscheinlich vertraut, auch wenn Sie den Begriff nicht kannten. Das Prinzip besteht darin, daß zur Ausführung einer Operation "Ändern des Inhalts einer Speicherzelle" indem zum Beispiel der Inhalt einer anderen Speicherzelle zur ersten addiert werden soll, immer folgende Schritte auf Maschinenebene ausgeführt werden:

1. Lade Speicherzelle B in Register R1
2. Lade Speicherzelle C in Register R2
3. Addiere R1 + R2
4. Speichere Ergebnis in Speicherzelle A

Man nennt dies auch LOAD-STORE Architektur, wenn alle Hauptspeicherzugriffe mittels der Befehle LOAD und STORE erfolgen und alle anderen Operationen nur mit Registeroperanden ausgeführt werden

Bei einer Speicher-Speicher Architektur sieht das obige Beispiel auf Maschinenebene folgendermaßen aus:

1. Addiere Speicherzelle B mit Speicherzelle C und lege das Ergebnis in Speicherzelle A ab.

Da die gleiche Aufgabe bei einer Speicher-Speicher-Architektur in einem statt vier Befehlen formuliert werden kann, spricht man von einem *komplexen* Befehl, der die Ausführung mehrerer *einfacher* Befehle beinhaltet. Die Speicher-Register-Architektur ist eine Mischform der beiden anderen.

8	4	16
Load	rB	B
Load	rC	C
Add	rA	rB rC
Store	rA	A

Register-Register

8	16
Load	B
Add	rC
Store	A

Speicher-Register

8	16	16	16
Add	B	C	A

Speicher-Speicher

Wie Sie leicht sehen können, benötigt die Speicher-Speicher Architektur für diese Instruktion nur 56 bit, während die Register-Register Architektur 104 bit benötigt. Deshalb verringern komplexe Befehle den Speicherbedarf für Programme.

3.2.1.1 Beispiele:

Der 6502 von Motorola ist ein Vertreter der Register Speicher Architektur. Die Befehlsfolge der obigen Aufgabe lautet hier:

```
VA = $1F0 Datendeklaration
VB = $1F1
VC = $1F2
LDA VB          Vorbereitung der Addition
ADD VC          Addition
STA VA          Ergebnis schreiben
```

Die IBM /360 kennt Speicher-Speicher Befehle, so daß mit einem Befehl eine Zeichenkette im Speicher auf eine andere Adresse kopiert werden kann. Beim Befehl

```
MVC (R2),Len,(R6)
```

ist die Adresse des Quellstrings in Register R6 abgelegt und die Adresse des Zielstrings in Register R2. Len enthält die Anzahl Bytes, die kopiert werden

3.2.1.2 Übung:Speicher-Speicher-Architekturen und Registeranzahl

Diskutieren Sie, ob Speicher-Speicher Architekturen weniger Register brauchen als LOAD/STORE Architekturen

3.2.2 Wieviele Register hat ein Prozessor - wie werden sie optimal genutzt ?

Im Zusammenhang mit dem 1. Argument ergibt sich die Frage, wieviel sind eigentlich *viele* Register bevor man klären kann, was das mit dem Compiler zu hat

Es hängt davon ab, welche *Befehlssatz-Architektur* auf einem Computer implementiert ist. Es gibt unterschiedliche Befehlssatz-Architekturen, die sich hauptsächlich darin unterscheiden, wie die Operanden auf der CPU gespeichert werden. Die Hauptvarianten sind ein *Stack*, ein *Akkumulator* oder ein *Registersatz*. Die Operanden in Stack-Architekturen sind implizit an der Spitze des Stack enthalten; in einer Akkumulator-Architektur ist ein Operand implizit der Akkumulator. Universalregister-Architekturen haben nur explizite Operanden - entweder Register oder Speicherplätze.

Bereitgestellter temporärer Speicher	Rechner/Prozessor-Beispiele	Befehlsbeispiele	Explizite Operanden pro ALU-Befehl	Ziel für Ergebnisse	Art des Zugriffs zu expliziten Operanden
Stack	B5500, HP 3000/70	PUSH B, PUSH C, ADD, POP A (Beispiel von oben : A = B+C)	0	Stack	Push und Pop zum oder vom Stack
Akkumulator	PDP-8, Intel 8085, Motorola 6502	ADC 00FF ADC (00FF),Y	1	Akkumulator	Laden/Speichern Akkumulator
Registersatz	IBM 360, DEC VAX und alle heute aktuellen Prozessoren	ADDL3 R1,737(R2),#456 (Addiere den Inhalt der Adresse 737+dem Displacement von R2 die Zahl 456 und lege sie in Reg. 1 ab.	2 oder 3	Register oder Speicher	Laden/Speichern von Registern oder dem Speicher.

Die vergleichsweise einfache Akkumulator-Architektur war bis in die 70ger Jahre aktuell. Der Intel 8086 sollte Sourcecode-kompatibel zum 8085 sein. Daraus ergab sich für die gesamte Intel 80x86 Reihe eine Mischform zwischen Akkumulator- und Registersatz-Architektur..

Der Motorola 6502 (8 Bit) aus den 70er Jahren hat 2 Indexregister und einen Akku. Der 8080 von INTEL hat drei 16 Bit Arbeitsregister plus Akku. Eine VAX 11/780 hat zwölf 32 Bit Universalregister. Der RISC Motorola 88000 und MIPS R3000 verwenden 32 Universalregister (32 Bit).

Der Trend ist offensichtlich: moderne Prozessoren haben heute mindestens 32 Register, die der Compiler verwendenden soll. Welche Probleme dabei für die Compiler auftreten, sehen wir später. Hier nur soviel: Programme verwenden Variablen zur Speicherung von Daten. Diese Variablen werden häufig mehr als einmal im Programm verwendet. Deshalb wäre es am besten, wenn jede Variable in einem Register gehalten werden kann, bis sie wieder benötigt wird, weil die CPU dann nicht einen langsamen Speicherzugriff durchführen bräuchte. Aber alle Variablen lassen sich nicht in der begrenzten Zahl von Registern halten. Deshalb muß der Compiler entscheiden, wie er die Register belegt. Bei komplexen Befehlssätzen kann man diese Entscheidung in der Hardware „verstecken“, weil bei den obengenannten komplexen Speicher-Speicher Befehlen ein Mikroprogramm die Auflösung des komplexen Befehls in eine Folge von einfachen Mikrobefehlen vornimmt. Der Compiler hat es also entscheidend einfacher!

3.3 Exkurs: Spracharchitekturen - Stackorientierte Rechner

Höhere Programmiersprachen kennen nicht nur skalare Größen als Operanden von Befehlen, sondern auch strukturierte Datenobjekte (z.B. STRING, ARRAY, RECORD). Sie haben Blockstrukturen zur Festlegung des Gültigkeitsbereiches von Variablennamen. Es gibt Kontrollkonstrukte für Wiederholungen und Iterationen. "Von all diesem weiß aber die Hardware eines von Neumann-Rechners nichts; d.h. es besteht eine weite semantische Lücke zwischen der höheren Programmiersprache und der Assemblersprache bzw. der Maschinensprache."[1]

Und was kümmert den Programmierer diese semantische Lücke, wo er doch in Hochsprachen programmieren kann? "Die semantische Lücke des von Neumann-Computers trägt wesentlich zum Problem mangelhafter Software-Zuverlässigkeit bei. Eine ganze Reihe gravierender Programmierfehler könnten von der Maschine aufgedeckt werden, wenn die semantische Lücke nicht bestehen würde. Meyers nennt dafür zwei Beispiele, nämlich (1) das Auftreten von nicht initialisierten Variablen und (2) das Referieren von nicht existierenden Elementen eines Feldes."[2]

Um diese semantische Lücke zu schließen, versuchte man spezielle Rechnerarchitekturen zu entwic??keln, sogenannte **Spracharchitekturen** und s**prachorientierte** Rechnerarchitekturen., je nach Vollständigkeitsgrad der Schließung der semantischen Lücke. Da die **Keller-Maschinen** die am häufigsten zu findende Form einer sprachorientierten Rechnerarchitektur ist (besser gesagt *war)*, wird noch ein bißchen auf diese eingegangen:

3.3.1.1 Keller Maschinen

Wahrscheinlich wissen Sie, daß alle modernen Computer einen **Stack** (Keller) benutzen. Er ist ein Speicher (in der Regel ein Teil des Hauptspeichers) der als LiFo (Last in, First out) mit entsprechender Hardwareunterstützung organisiert ist. Die Hardwareunterstützung besteht im allgemeinen in einem eigens hierfür reservierten Register, dem sog. Stackpointer. Dieser Stackpointer enthält die Adresse auf das aktuelle Element des Stacks.

---

[1] Giloi, Rechnerarchitektur, Heidelberger Taschenbücher, Sammlung Informatik,Springer Verlag, Berlin Heidelberg New York 1981, S. 239 f.

[2] Giloi, ebda S. 240

	Stack	Stack-Adresse
Stackpointer ---------->	6. Eintrag	FFFA
	5. Eintrag	FFFB
	4. Eintrag	FFFC
	3. Eintrag	FFFD
	2. Eintrag	FFFE
	1. Eintrag	FFFF

Mit der Operation **PUSH** wird ein Datenelement an der aktuellen Adresse des Stackpointers abgelegt. Der Inhalt des Stackpointers wird dekrementiert. Im obigen Beispiel war der Initialwert des Stackpointers FFFF, nach dem 6. Eintrag enthält er FFF9.

Mit der Operation **POP** wird ein Datenelement von der aktuellen Adresse des Stackpointers gelesen. Vorher wird der Inhalt des Stackpointers wieder inkrementiert. Im obigen Beispiel ist der Inhalt des Stackpointers nach sechs POP Operation wieder FFFF.

Verwendung findet der Stack in konventionellen Computer-Systemen vor allem beim Aufruf von Prozeduren: Die Parameter können über den Stack an die Prozedur übergeben werden und die Rücksprungadresse für den RETURN Befehl werden auf den Stack gelegt.

Im Gegensatz zu den üblichen Registermaschinen, die den Stack nur für bestimmte Zwecke einsetzen, kennt eine Kellermaschine auf der Ebene der Maschinensprache keine Register (das heißt übrigens nicht, daß die Maschine auf Hardwareebene keine Register besäße!). Aber für den Programmierer sind die Register transparent. Er kennt nur die Stackoperationen PUSH und POP und zusätzlich die arithmetischen Befehle wie ADD, MPY, DIV usw. Jede arithmetische Operation bezieht sich auf die letzten (oberen) beiden Stackelemente.

Reine Kellermaschinen haben sich nicht durchgesetzt. "Kellermaschinen sind im Vergleich zu Registermaschinen weniger flexibel. Der Keller-Mechanismus ist umständlich bei der Bearbeitung von Feldern und generell ungeeignet zur Handhabung großer Datenmengen. Der Vorteil einer Keller-Maschine, außer bei den Transportanweisungen PUSH und POP und den Sprunganweisungen mit adressenlosen Instruktionen auszukommen, ist unerheblich im Vergleich zu den Registermaschinen, bei denen auch nur wenige Bits für eine Registeradresse aufzuwenden sind, die sich üblicherweise leicht im Befehlsformat unterbringen lassen."[1]

3.3.1.2 Übung: Parameterübergabe

Welchen Vorteil bietet eine Parameterübergabe und Ablage der Prozedur-Rücksprungadresse mittels Stack gegenüber einer Verwendung von eigens hierfür reservierten Registern?

Welchen Vorteil bietet umgekehrt eine Parameterübergabe und Ablage der Prozedur-Rücksprungadresse mit eigens hierfür reservierten Registern ?

---

[1] Giloi, ebda. S.249

3.3.1.3 Übung: semantische Lücke und Compiler

Prüfen Sie, inwiefern die Compiler der Programmiersprachen Pascal, Modula, Fortran oder andere nach Ihrer Wahl die Mängel der semantischen Lücke kompensieren.

3.4 Design-Prinzipien der 70er Jahre

Aus den Argumenten für große Befehlssätze wurden folgende Design-Entscheidungen für die typischen Maschinen der 70er Jahre getroffen.

- Befehlssätze sind groß und mikroprogrammiert.
- Befehle sind komplex, denn Mikrobefehlsabarbeitung ist schneller als normaler Maschinencode.
- Die Befehlslänge ist variabel.
- Register sind altmodisch. Auf sie kann gegebenenfalls zugunsten von Stack- oder Speicher-Speicher-Architekturen ganz verzichtet werden.
- Caches, wenn überhaupt vorhanden, sind sehr viel kleiner als die Mikroprogrammspeicher.

	IBM 370/168	VAX 11/780	Dorado	iAPX 432
Jahr	1973	1978	1978	1982
Befehlszahl	208	303	270	222
CM-Größe (Kbit)	420	480	136	64
Befehlslänge (bit)	16-24	16-456	8-24	6-321
Ausführungsmodell	Reg-Reg Sp-Sp Reg-Sp	Reg-Reg Sp-Sp Reg-Sp	Stack	Stack Sp-Sp
Cache-Größe (Bit)	64	64	64	0

Typische Architekturen der 70er Jahre

3.5 Irrtümer

Rechnerentwerfer hielten sich lange an die Kurzregel: *Codelänge = Geschwindigkeit*, d.h., die Architektur, für die bei einem gegebenen Programm der kleinste Programmcode erzeugt wird, ist die schnellste (siehe Complex Instruction Set Computer

Argumente für große Befehlssätze). Die statische Codelänge ist wichtig, wenn der Speicherplatz knapp ist, aber es ist nicht dasselbe wie Leistung. Wie wir noch sehen werden, können große Programme, die aus Befehlen, die einfach zu holen, zu decodieren und auszuführen sind, schneller laufen als Maschinen mit extrem kompakten Befehlen, die aber schwer zu decodieren sind. *Codelänge = Geschwindigkeit* ist besonders populär bei Compiler-Entwicklern, weil es schwierig zu entscheiden ist, ob eine Befehlsfolge schneller ist als eine andere, aber man

sieht sofort, welche kürzer ist. Einen eindrucksvollen Beweis für diesen Irrtum gibt es schon seit 1973 von B.A. Wichmann, der die relative Ausführungszeit, die Befehlszahl und die Codelänge von Programmen in Algol 60 für 2 Architekturen (CDC6600 von Control-Data und Burroughs B5500) untersuchte. Obwohl die CDC-6600 Programme mehr als doppelt so groß sind, laufen Algol 60 Programme sechsmal schneller als auf der B5500, die sogar eigens für Algol 60 entwickelt wurde![1][1]

Auf einen ähnlichen Sachverhalt stieß man auch 1977, als man versuchte, die Leistung einer Architektur unabhängig von ihrer Implementierung [2] zu bestimmen. Drei quantitative Maße wurde zur genauen Untersuchung von Architekturen eingeführt:

- **S** Zahl der Bytes für den Programmcode (Übersetzungszeit)
- **M** Zahl der zwischen Speicher und CPU während der Programmausführung zu übertragenden Bytes für Code und Daten (Laufzeit)
- **R** Zahl zwischen den Registern der CPU übertragenen Bytes.

Setzt man dieses Verfahren zur Leistungsbestimmung der Rechner VAX 3100 und DEC 3100 mit realen Programmen (C-Compiler, TeX und Spice) ein, stellt man fest, daß die DEC 3100 bis zu 500% schneller ist als die VAX, obwohl ihr S-Wert bis zu 70% und ihr M-Wert bis zu 15% schlechter ist. „Der Versuch, eine Architektur unabhängig von der Implementierung einzuschätzen, war zwar heldenhaft, aber offensichtlich nicht erfolgreich.“[3]

	S Codelänge in Byte		M Megabyte Code + übertragene Daten		CPU Zeit in Sekunden	
	VAX 3100	DEC 3100	VAX 3100	DEC 3100	VAX 3100	DEC 3100
Gnu-C-Compiler	409600	688128	18	21	291	90
TeX	158720	217088	67	78	449	95
Spice	223232	372736	99	106	352	94

[1] Vgl Hennessy, Patterson Rechnerarchitektur 1994 S. 71

[2] Unter Implementierung einer Architektur versteht man die Art und Weise, wie ein Befehlssatz realisiert wird. Bei der Implementierung geht es deshalb um den internen CPU-Entwurf, das Speicher-System, das Bus-System usw. Der Einsatz von Pipelining ist also ein Aspekt der Implenentierung.

[3] Hennesy,Patterson, a.a.O. S. 79

4 Vom CISC zum RISC

4.1 Änderung der Randbedingungen

Anfang der 80er Jahre hatte sich die Argumentationsgrundlage infolge technologischer Entwicklungen, aber auch durch neue Erkenntnisse und Erfahrungen verändert.

- Die Verbreitung des Halbleiterspeichers auch als Hauptspeicher verringerte die Differenz bei den Zugriffszeiten zwischen Mikrocode-ROM (Control Memory CM) und Hauptspeicher. Maschinencode-Zugriffe waren nicht mehr 10 mal langsamer als Mikroprogrammzugriffe!
- Cache Speicher verringerten die effektive Hauptspeicherzugriffszeit und reduzierten damit die Lücke zwischen CM- und Hauptspeicherzugriff noch weiter.
- Bei einer genauen, quantitativen Betrachtung der Programmausführungszeiten wurde klar, daß nicht nur die Anzahl der Befehle pro Programm, sondern auch die Anzahl der Taktzyklen pro Befehl in die Zeit eingehen. Diese Zahl ist bei mikroprogrammierten Maschinen viel größer als bei solchen mit hardwired control.
- Mit Verbilligung der Speicherbausteine wurde das Optimierungsziel der Speicherplatzverringerung zweitrangig.
- Die erhoffte höhere Zuverlässigkeit der "Hardware" war nicht realisierbar, denn das unter der Maschinenbefehlsebene liegende Mikroprogramm (die sogenannte Firmware) ist nichts anderes als in Silizium gegossene Software. 400.000 Bit Firmware sind nie Fehlerfrei, aber schwer zu ändern
- Die komplexen Befehle, die eigentlich die Compiler vereinfachen sollten, wurden kaum genützt. Compiler verwendeten nur Subsets der großen Befehlssätze.

4.2 Die RISC Ursprünge: Neue Design-Prinzipien

Als Folge der erkannten Unzulänglichkeiten ergab sich vor allem der Wunsch nach einer Verkürzung des Befehlszyklus. Dies setzt, wie noch gezeigt werden wird, u.a. einen kleinen, einfachen und regelmäßigen Befehlssatz voraus. Aus dieser Eigenschaft wurde der Name Reduced Instruction Set Computer geboren. Die neuen, zur Einfachheit tendierenden Designprinzipien lauteten:

- Regel: Mehr als 90% der ausgeführten Befehle sind einfach (LOAD, STORE, ALU-Operationen, Verzweigungen). Die Unterstützung der restlichen 10% durch komplexe Maschinenbefehle erhöht wahrscheinlich die Zykluszeit, verlangsamt also die 90%. 10% der Befehle schneller zu machen mit der Folge der Verlangsamung von 90% ist nicht vertretbar, da dies die Gesamtleistung senkt. Folge: Vor Einführung eines neuen komplexen Befehls ist nachzuweisen, daß er im Mittel mehr Zyklen einspart als er an Verlusten verursacht.
- Festverdrahtete Steuerungen erlauben niedrigere Zyklenzahlen pro Befehl als mikroprogrammierte. Voraussetzung dafür sind kleine Befehlssätze und vor allem wenige und regelmäßige Befehlsformate.
- Die effektivste Maßnahme zur Reduktion der Zyklenzahl ist das Pipelining, also die zeitlich überlappte Abarbeitung des Befehlszyklus. Einfache Befehle eignen sich besser für Pipelines.

- Compiler sollten aus komplexen Befehlen (Hochsprache) einfache (Maschinensprache) erzeugen. RISC-Compiler tun genau das. CISC-Compiler stehen häufig vor dem Problem, einen komplexen Befehl in einen anderen ähnlich komplexen, aber nicht genau passenden umzuwandeln.
- RISC-Prozessoren benötigen in Folge der einfacheren Maschinenbefehle häufigere Zugriffe zum Befehlsspeicher. Um daraus keinen Nachteil entstehen zu lassen, sind effiziente Speicherhierarchien unter Einsatz von Cache-Speichern notwendig.
- Der Datenverkehr zwischen Hauptspeicher und Prozessor wird, neben dem Einsatz von Caches, reduziert durch die Lokalhaltung von Daten in großen On-Chip-Registersätzen.

4.2.1.1 Fragen : Befehlsformate , Pipelines, Caches

Folgende Fragen sollen in erster Näherung beantwortet werden. Im Verlauf der Veranstaltung werden sie vertieft.

1. Was sind regelmäßige und unregelmäßige Befehlsformate ?

2. Was sind Befehlspipelines ?

3. Was sind Speicherhierarchien ?

4. Was sind Caches ?

5. Was sind On-Chip Registersätze ?

4.3 Befehlsformat und Befehlssatz

Die Bezeichnung RISC läßt vermuten, daß Prozessoren mit RISC-Architektur allgemein über einen reduzierten Befehlssatz, also deutlich weniger Befehle, verfügen als CISC Prozessoren. Das trifft jedoch nicht grundsätzlich zu. Zahlreiche RISC-CPUs bieten mehr als 100 Befehle an. (z.B. IBM RS/6000 mit 184 Befehlen) Statt allein die Zahl der Befehlstypen zu reduzieren, tendiert man heute eher dazu, RISC im Sinne einer Reduktion der pro typischer Aufgabe benötigten Befehlszyklen (Reduced Instruktion Set Cycle). Ein wesentlicher Beitrag wird auf der Befehlssatzebene geleistet: einheitliche, leicht zu entschlüsselnde Befehlsformate, Vermeidung von Befehlstypen mit zu hohem Steuerungs- oder Berechnungsaufwand und Bevorzugung von Registerarithmetik.

Im Gegensatz zu Befehlsformaten typischer CISC-Architekturen, deren Länge in Abhängigkeit von der Funktionalität variiert, die vom Befehl realisiert werden muß, wird für RISC Befehlsformate meistens eine einheitliche Länge von 32 Bit gewählt. Das kann zu einer gewissen Redundanz führen, die sich gegenüber CISC-Architekturen durch eine etwas erhöhte statische Codelänge bemerkbar macht. Die Vorteile eines leicht dekodierbaren und möglichst einheitlichen Befehlsformates überwiegen jedoch:

- Bei gleicher Länge aller Befehle wird mit jedem Speicherzugriff ein vollständiger Befehl an die CPU übertragen. Eine Überprüfung auf Vollständigkeit und gegebenenfalls weitere Speicherzugriffe entfallen daher.
- Die einheitliche Länge vereinfacht die Zuordnung von Bedeutungsträgern an feste Bitfelder innerhalb des Befehlswortes. Wenn beispielsweise die von arithmetischen Registeropratinen angesprochenen Registeradressen immer an der gleichen Stelle innerhalb des Befehlswortes auftreten, können die Registeropranden bereits in die ALU geholt werden, während gleichzeitig noch der Opcode decodiert wird. In manchen RISC-Architekturen entfällt dadurch eine Pipelinestufe.[1]

4.3.1 Beispiel für CISC: Das IBM /360 Befehlsformat

RR-Format Register-Register	Opcode	R1	R2				
RX- Format Register Indexiert	Opcode	R1	X2	B2	D2		R1 ←R1 op M[X2+B2+D2]
RS-Format Register-Speicher	Opcode	R1	R3	B2	D2		R1 ←M[B2+D2] op R3
SI-Format Speicher-Immediate	Opcode	immediate		B1	D1		M[B1+D1] ←immediate
SS-Format Speicher-Speicher	Opcode	Länge		B1	D1	B2	D2 M[B1+D1] ← M[B1+D1] op M[B2+D2]

Befehls-Bytes	1	2	3	4	5	6

[1] Vergl. Märtin, Christian, Rechnerarchitektur, Struktur, Organisation, Implementierungstechnik, Hanser Verlag München 1994 S. 160ff

4.3.2 Beispiel für RISC: Das DEC-Alpha Befehlsformat

31 26	25 21	20 16	15		3 0	
Opcode	Number					PAL-Code-Format
Opcode	RA	Branch-Displacement				Branch Format
Opcode	RA	RB	Displacement			Memory Format
Opcode	RA	RB	Function		RC	Operate Format

4.3.2.1 Memory Format

LOAD und STORE-Anweisungen sind Speicherbefehle. Sie verschieben Longwords und Quadwords zwischen dem Register RA und dem Speicher, wobei RB zusammen mit einem 16 Bit Displacement als Speicheradresse dient.. Insgesamt umfaßt das Memory Format einen 6 Bit Opcode, die beiden 5 Bit Adreßfelder RA und RB sowie das Feld Displacement. Addiert zu Inhalt des Registers RB, bildet das Displacement eine virtuelle Adresse.

4.3.2.2 Branch-Format

Das Branch-Format wird für bedingte Verzweigungen und programmzählerabhängige Sprünge in Unterprogramme benutzt. Es besteht aus dem Opcode, dem Registeradreßfeld RA und einem 21 Bit Displacement

4.3.2.3 Operate Format

Operate Anweisungen lesen zwei Register, führen Operationen aus und Speichern das Ergebnis in ein Register RC.

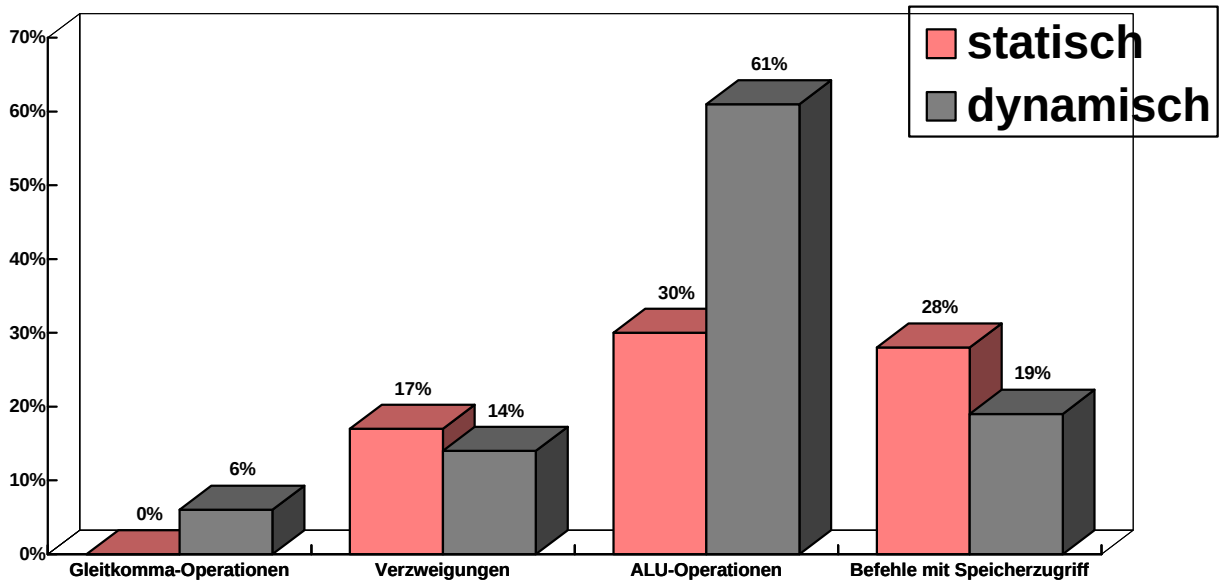
4.3.2.4 PAL-Code Format

PAL-Code Anweisungen veranlassen die Ausführung einer Systemroutine aus einer Bibliothek privilegierter oder komplexer Operationen.

4.4 Befehlssatzmessungen

4.4.1 Statische und dynamische Häufigkeitsbestimmung

Bei der Bestimmung, welche Befehle bei einer Architektur am häufigsten verwendet werden, unterscheidet man zwischen statischer und dynamischer Häufigkeit. Die statische Untersuchung zählt einfach am Maschinen-Code eines Programmes (oder besser, bei vielen Programmen) ab, wie oft die Befehle vorkommen. Eine dynamische Untersuchung hingegen mißt, wie häufig die einzelnen Befehle zur Laufzeit eines Programmes ausgeführt werden.



Daten einer IBM /360 FORTRAN- Benchmark-Messung[1]

Die 20 am häufigsten ausgeführten Befehle wurden in vier Klassen eingeteilt. Sie zeigen, wie unterschiedlich das statische und dynamische Erscheinungsbild sein kann. Im Falle der dynamischen Messungen machen diese 20 Befehle nahezu 100% der Befehlsausführung aus, jedoch nur 75% der statischen Werte.

4.5 Zeitverteilung und Häufigkeitsverteilung

Daten der Befehlssatznutzung sind wichtige Eingangsdaten für den Architekten, geben aber keine Auskunft darüber, wie lange die Befehle zur Ausführung benötigen. Ein Irrtum bestand früher in der Annahme, daß die Häufigkeitsverteilung und die Zeitverteilung dicht beieinander lägen. Ein einfaches Beispiel, in dem diese Verteilungen sehr unterschiedlich sind, ist das COBOlGO-Programm auf der 360. Die folgende Tabelle zeigt die wichtigsten Befehle nach Häufigkeit und Zeit. Die zwei häufigsten Befehle stehen für 33% der Befehlsausführungen, doch nur für 4% der Ausführungszeit.

Befehlsfolge nach Häufigkeit	Häufigkeit in %	Befehlsfolge nach Zeitverteilung	Zeitanteil in %
L, LR	19	ZAP	16
BC, BCR	14	AP	16
AP	11	MP	13
ZAP	9	MVC	9
MVC	7	CVD	5

Zeitverteilungen sind besonders wichtig bei Architekturen wie der VAX, bei der die Zahl der Zyklen für einen Befehl zwischen einem und mehreren hundert variieren kann

Der Unterschied zwischen Häufigkeitsverteilung und Zeitverteilung existiert nicht bei einfachen LOAD/STORE Architekturen.

[1] Hennessy, Patterson, a.a.O. S.140

4.6 Befehlssatzmessung - ein Beispiel

Der Befehlssatz der VAX besteht aus 303 zum Teil sehr komplexen Befehlen. Die Frage ist, in welcher Häufigkeitsverteilung die Befehle vorkommen. Denn wenn man eine Maschine optimieren will, sollte die Architektur für die am häufigsten vorkommenden Befehle optimal sein. Die VAX wird hier als Beispiel für die CISC Maschinen verwendet, weil es zu ihr sehr eingehende Untersuchungen gibt.

Bei Benchmark-Untersuchungen mit den Programmen SPICE, COBOLX, TeX und GCC stellte sich heraus, das 27 Befehle durchschnittlich 88% der ausgeführten Befehle ausmacht. Der Rest der Verteilung ist lang. Es gibt viele Befehle mit einer Häufigkeit von 0,5 bis zu 1,0% . Im SPICE machen die oberen 15 Befehle 90% der ausgeführten Befehle aus, und die oberen 26 tragen zu 95% bei. Es gibt jedoch 149 unterschiedliche Befehle, die wenigstens einmal ausgeführt werden.[1]

DEC setzte diese Ergebnisse beim Entwurf der MicroVAX 32 um. Es wurden nur noch die am häufigsten benutzten Befehle implementiert. Die restlichen Instruktionen wurden über Softwareroutinen realisiert. Die CPU der MicroVAX paßte dadurch auf einen Chip und erzielte mit nur 64 KBit Microcode und 101 K Transistoren praktisch die gleiche Leistung wie die VLSI VAX, die für den vollen Befehlssatz (303 Instruktionen) 480KBit Microcode und 1250 K Transistoren benötigte. IBM erreichte damals mit einem PL/8 Compiler, der nur die Register-Register Teilmenge des System /370-Befehlssatzes unterstützte, eine um 50 % gesteigerte Durchsatzleistung.[2]

	voller Befehlssatz (VLSI VAX)	**Subset-Befehlssatz (Micro-VAX 32)**
implementierte Befehle	100%	80%
Größe des Steuerspeichers (Bit)	480 K	64 K
% der Leistung einer VAX-11/780	100	90
Anzahl der Chips eines Prozessors	9	2

20 % der VAX-Befehle sind verantwortlich für 60% des Mikrocodes, werden aber nur zu 0,2 % der Zeit abgearbeitet.[3]

---

[1] Hennessy, Patterson, a.a.O. S.171

[2] Märtin, Christian, a.a.O. S. 159

[3] Hennessy, Patterson, a.a.O. S.243

5 Die Risc-Philosophie

5.1 Entwurfsphilosophie

In der kurzen Geschichte der Rechnerarchitektur hat sich eine Anzahl von Rechnerfamilien herausgebildet, die jeweils strikten Kompatibilitäts-Anforderungen genügen (z.B. IBM Mainfraimes 360 und 370, die Mikroprozessoren Intel 8080, 8085, 8086, 80x86 und Motorola 680x0). Sie sind dadurch gekennzeichnet, daß Maschinencode, welcher für ein bestimmtes Modell erzeugt wurde, auch auf dem Nachfolgemodell ablaufen kann. Dies erfordert, daß ein Nachfolger hinsichtlich Befehlssatz und Registerstruktur eine Obermenge des Vorgängers darstellt. Für die Verträglichkeit zwischen weiter entfernten Generationen werden dabei Abstriche in Kauf genommen, so z.B. der Verzicht auf Maschinencodekompatibilität, wodurch u.U. Re-Compilation oder Re-Assemblierung nötig werden. Aufwärtskompatibilität schützt die Investitionen in existierender Software.

Die RISC-Philosophie stellt bei weitem keine Grundlage für eine ähnlich rigorose Familienbildung dar, auch wenn sich innerhalb der Klasse der RISC-Rechner wieder aufwärtskompatible Familien bilden. Das alle RISC-Prozessoren verbindende Band ist weitaus lockerer und an keiner Stelle letztgültig festgelegt. Ein RISC-Prozessor sollte nach einer Definition von C. Müller-Schloer zwei Bedingungen erfüllen:

1. "RISC-Prozessoren sind einfach in dem Sinne, daß nur solche Komponenten (Befehle, Register, Busse etc.) verwendet werden, die nachweisbar die Leistung im Vergleich zur nötigen Komplexitätserhöhung hinreichend vergrößern. Daraus folgt, daß alle Resourcen zur Unterstützung häufig gebrauchter Funktionen eingesetzt werden, notfalls auf Kosten der selteneren. Daraus folgt auch die Notwendigkeit einer u.U. anwendungsunabhängigen Kosten-Nutzen Analyse. Wie neuere RISC-Prozessorern deutlich machen, heißt Einfachheit aber nicht unbedingt, daß die Komplexität niedrig ist; das war nur für die ersten Vertreter der RISC-Prozessoren gültig.
2. RISC-Prozessoren weisen eine Vielzahl von Architekturmerkmalen aus einer Liste typischer RISC-Eigenschaften auf. Zu diesen RISC-Merkmalen gehören:
   - Ein-Zyklus Operationen: Ausführung nahezu aller Maschinenbefehle in einem Taktzyklus.
   - LOAD-STORE-Architektur: Hauptspeicherzugriffe erfolgen nur mittels der Befehle LOAD und STORE. Alle anderen Operationen werden allein auf Registeroperanden ausgeführt.
   - Verzicht auf die Mikrobefehlsebene. Die Operationssteuerung des Leitwerks wird durch festverdrahtete Hardware realisiert.
   - wenige Befehle
   - weniger Befehlstypen und Adressierungsarten. Konzentration auf wirklich erforderliche und einfach zu realisierende Befehlstypen und Adressierungsmodi.
   - möglichts einheitliches Befehlsformat mit gleichbleibender inhaltlicher Bedeutung der Bitfelder. Möglichst geringer Decodieraufwand für alle Befehle.
   - Aufwandsverlagerung von der Hardware in den Compiler, von der Laufzeit in die Compile-Zeit."[1] Delegation von Steuerungsaufgaben an optimierende Compiler.

[1] C. Müller-Schloer, a.a.O, S. 31f

Keines dieser Merkmale ist zwingend. Das Zustandekommen dieser Liste von Architekturmaßnamen wird in den nächsten Kapiteln hergeleitet.

5.2 Adressierungsarten

Die folgende Tabelle zeigt die unterschiedlichen Adressierungsarten, mit denen die Adresse eines Objektes spezifiziert wird, auf das zugegriffen werden soll.. In Universalregister-Maschinen kann eine Adressierungsart eine Konstante, ein Register oder einen Speicherplatz im Hauptspeicher spezifizieren.. Wenn ein Speicherplatz genutzt wird, heißt die durch die Adressierungsart spezifizierte aktuelle Speicheradresse *effektive* Adresse. Für die weiteren Betrachtungen ist der Hinweis wichtig, daß es einfache und sehr aufwendige Adressierungsarten gibt[1].

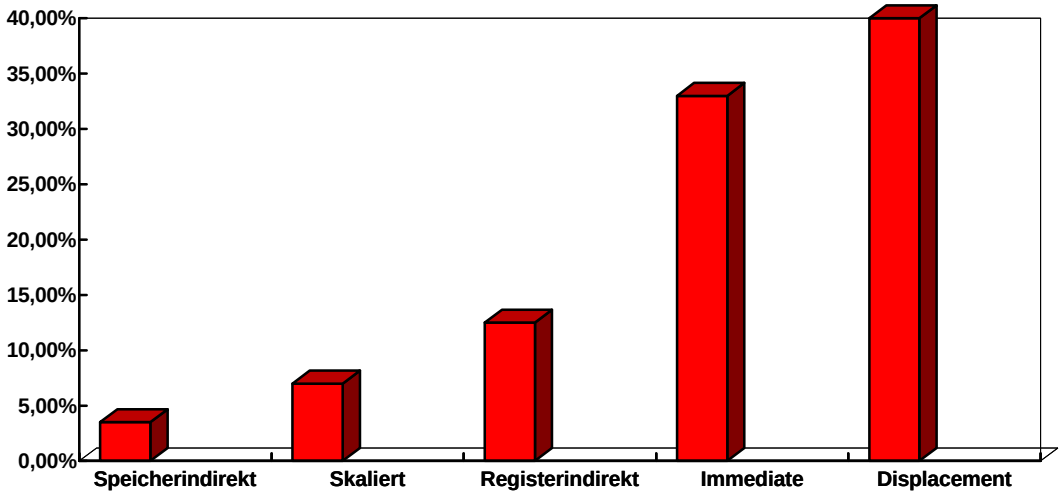
Adressierungsart	**Beispiel**	**Wirkung**	**Anwendung**
Register	ADD R4,R3	R4←R4+R3	Wert ist im Register
Immediate oder Literal	Add R4,#3	R4←R4+3	Operand ist eine Konstante (In einigen Maschinen sind Immediate und Literal zwei verschiedene Adressierungsarten
Displacement oder Based	Add R4,100(R1)	R4←R4+M[100+R1]	Zugriff zu lokalen Variablen
Register indirekt	Add R4, (R1)	R4←R4+M[R1]	Register dient als Pointer (Zeiger)
Indexiert	Add R3,(R1+R2)	R3←R3+M[R1+R2]	Manchmal nützlich für Feldadressierung - R1 = Basis des Feldes, R2=Index des Feldes
Direkt oder absolut	Add R1 (1001)	R1←R1+M[1001]	Zugriff zu statischen Daten, notwendige Adreßkonstante kann groß sein.
Speicherindirekt	Add R1, @(R3)	R1←R1+M[M[R3]]	Speicherplatz dient als Pointer
Autoinkrement	Add R1, (R2)+	R1←R1+M[R2] R2←R2 + *d*	Für Zugriff zu Feldern in Schleifen nützlich. R2 zeigt auf den Anfang eines Feldes; jeder Zugriff erhöht R2 um die Länge *d* eines Elementes
Autodekrement	Add R1, -(R2)	R2←R2 -d R1←R1+M[R2]	Dieselbe Anwendung wie Autoincrement. Beide können auch zur Implementierung der Stackoperationen PUSH und POP verwendet werden.
Skaliert oder indexiert	Add R1,100(R2)[R3]	R1←R1+M[100+R2+R3*d]	Indexierung von Feldern mit Datentypen der Länge *d*

5.2.1.1 Übung:Adressierungsarten 1

Diskutieren Sie, inwiefern die unterschiedlichen Adressierungsarten in die Laufzeitt der Befehle eingehen.

[1] Hennessy, Patterson S. 98

Untersuchungen zur Anwendung von Adressierungsarten kamen zu folgendem Ergebnis:



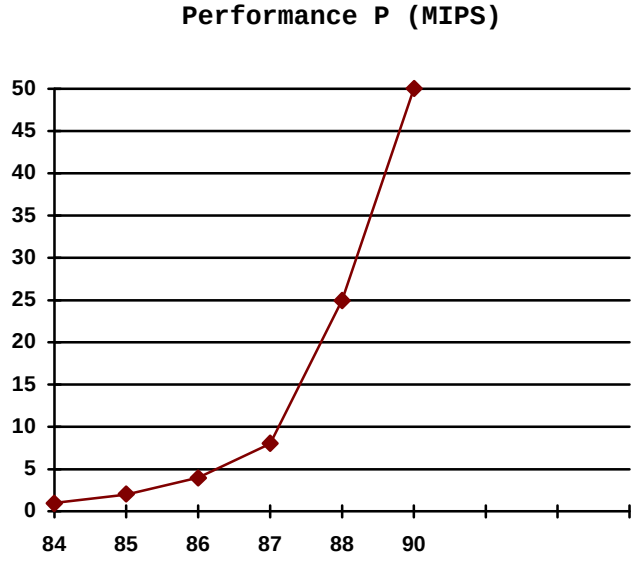
Diese Daten wurden von Hennessy und Patterson auf einer VAX gemessen. Sie wurden getrennt für die Programme Tex, Spice und Gnu-C-Compiler generiert. Für diese Darstellung wurden die Ergebnisse arithmetisch gemittelt.

5.2.1.2 Übung:Adressierungsarten 2

Beurteilen Sie das obige Diagramm.Welche Schlußfolgerung drängt sich den Rechnerarchitekten auf?

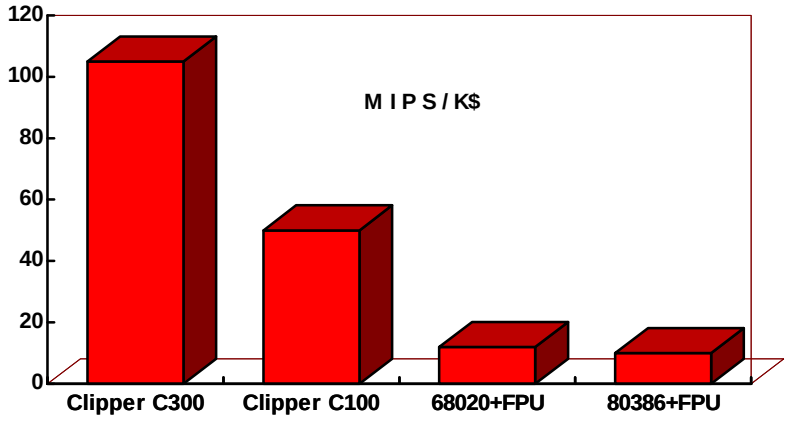
5.3 Performance als Optimierungsziel

Bis Mitte der 80er Jahre wurden Mikroprozessoren kaum nach ihrer Leistung beurteilt. Der vorwiegende Einsatz lag im Bereich der Steuerungen, wo es aber mehr auf Interruptverhalten ankam. Mit dem Einsatz von Mikroprozessoren in Universalrechnern (PC und Workstation) wurde die Performance, gemessen als Anzahl pro Zeiteinheit ausgeführter Befehle, wichtiger. RISC-Architekturen verwenden die in **MIPS** (million instruction per second) gemessene Leistung als Optimierungskriterium. Bill Joy, Entwicklungschef und Mitbegründer von SUN Microsystems, sagte eine jährliche Leistungsverdopplung voraus (bekannt als Joy´s Law)

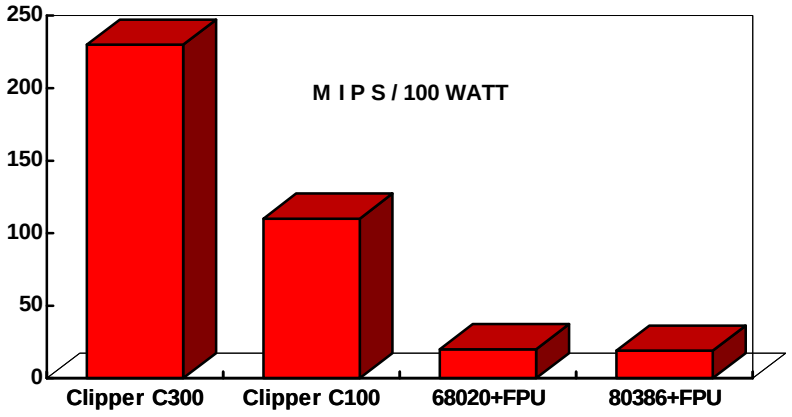


$P = 2^{Y-1984}$

Wichtiger als die absolute Meßgröße MIPS sind relative Leistungsaussagen geworden: im Sinne einer Aufwands-/Nutzenbetrachtung wird der erzielte Nutzen (MIPS) in Relation gesetzt z. B. zur Verlustleistung, zur Boardfläche oder zum Preis. Eine Zielvorgabe der RISC-Architekten ist, ein Preis-Leistungsverhältnis von 1$/MIPS zu erreichen. Meines Erachtens ist dies spätestens mit dem Einsatz des Alpha-Chips 21064 von DEC seit 1992 erreicht.



RISC: Clipper CISC:68020, 80386
Preis Leistungsverhältnis RISC und CISC.



RISC: Clipper CISC:68020, 80386
Verlustleistungsverhältnis RISC und CISC

6 Performance-Berechnung

6.1 Ideale Performance

Die CPU wird mit einem regelmäßig im Abstand t_C auftretenden Clock-Signal versorgt. Alle Aktionen der CPU können im Vielfachen dieser Taktzeit t_C gemessen werden. (Vergleichen Sie hierzu das Skript Rechnerarchitektur, Kapitel: Der SAP und die Ablaufsteuerung.)

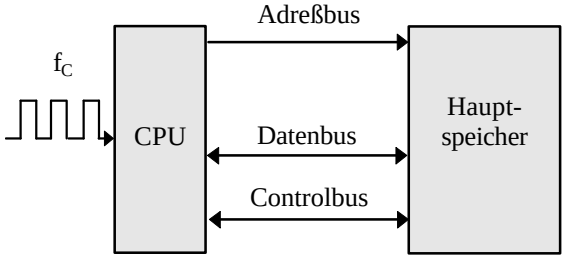
Der Kehrwert von t_C ist die Taktfrequenz $f_C = 1/t_C$

6.1.1.1 Übung: Taktfrequenz und Taktzeit.

Bestimmen Sie die Taktzeiten für die Taktfrequenzen in Nanosekunden:

a) 10 MHz

b) 50 MHz



Einfaches Computersystem

Der Befehlszyklus dieses Systems läßt sich wie folgt beschreiben:

1. A <-- Befehlsadresse	CPU legt Befehlsadresse auf Adreßbus A
2. D <-- Befehl	HS legt adressiertes Datum (=Befehl) auf Datenbus D
3. Decodieren	Die CPU decodiert den Befehl
4. A <-- Operandenadresse	CPU legt Operandenadresse auf Bus A
5. D <-- Operand	HS legt Operand auf D
6. Befehlsausführung	CPU führt Verarbeitung aus
7. A <-- Ergebnisadresse	CPU legt Ergebnisadresse auf Bus A
8. D <-- Ergebnis	CPU legt Ergebnis auf D
weiter bei 1.	

Gab es beim SAP nur die Unterscheidung zwischen Befehl holen und Befehl ausführen, kann man für moderne Rechner den Befehlszyklus gemäß obiger Tabelle wie folgt feiner aufteilen:

Schritt	1 und 2	3	4 und 5	6	7 und 8
Kürzel	BH	BD	OH	BA	ES
Inhalt	Befehl holen	Befehl dekodieren	Operand holen	Befehl ausführen	Ergebnis speichern
	<-- t_C -->	<-- t_C -->	<-- t_C -->	<-- t_C -->	<-- t_C -->

Befehlszyklus

Die Befehlsabarbeitungszeit (T_L)ergibt aus der Anzahl der dafür notwendigen Taktzyklen **CPI** (Cycles per instruction) und der Taktzeit t_C.

$T_L = CPI * t_C$.

Die Performance P gibt an, wieviel Befehle in einer Sekunde abgearbeitet werden. Deshalb ist die Maßeinheit für die Performance üblicherweise MIPS

$P = f_C / CPI$

Bei einer mit 20 MHz getakteten CPU mit einem mittleren CPI Wert von 5 ergibt sich also

$P = 20*10^6 / 5$

P = 4 MIPS

Später werden wir sehen, wie mit Hilfe von Fließbandverabeitung (Pipelining) CPI auf den Wert 1 reduziert werden kann.

6.2 Reale Performance

Abweichungen von der idealen Performance ergeben sich durch

- Verzögerung beim Speicherzugriff
- Unterschiede in der Befehlssatz-Mächtigkeit

6.2.1 Speicherzugriff

Der Zugriff auf einen aus dynamischen RAMs aufgebauten Hauptspeicher dauert etwa 80 ns bis 300 ns, also länger als die Taktzeit t_C der CPU. Das bedeutet, daß die CPU zusätzliche Leerzyklen (Wait States) einschieben

muß, die als Verlustzeit zu verbuchen sind. Diese Verzögerung wird als MEMD (memory delay) bezeichnet und muß in der Performance Formel berücksichtigt werden:

P = fc/(CPI+MEMD)

Um diese Verzögerungszeit zu minimieren, werden in modernen Prozessoren schnelle Pufferspeicher (Caches) eingesetzt, deren Zugriffszeit auf die Taktzeit der CPU zugeschnitten ist.

6.2.2 Befehlssatz Mächtigkeit

Die verschiedenen Prozessoren unterscheiden sich in Anzahl und Mächtigkeit ihrer Befehle. Benötigt ein Prozessor zu Abarbeitung eines Programms mehr Maschinenbefehle als ein anderer, dann bezeichnet man den Befehlssatz des ersteren als weniger mächtig. Allgemein gilt: Die Befehlssätze von CISC Maschinen sind mächtiger als die der RISC mit ihrer LOAD-STORE Architektur. Um die Vergleichbarkeit unterschiedlicher Befehlssätze zu erreichen, normiert man die Befehlssätze auf eine allgemein bekannte und akzeptierte Architektur, die VAX 11/780:

n_{VAX} = Anzahl Befehle VAX / Anzahl Befehle Vergleichsarchitektur

Für einen Befehlssatz, der einfacher ist als der der VAX, ist n_{VAX} kleiner 1. Typische Werte für RISC-Prozessoren liegen bei 0,5 bis 0,8.

Die Performance Gleichung lautet dementsprechend jetzt:

$P = (fc/(CPI+MEMD)) * N_{VAX}$

6.2.3 Anwendung der Perfomance Formel

Vergleich Motorola 68020 (CISC) und MIPS R3000 (RISC) bei gleicher Taktfrequenz 25 MHz:

Prozessor	CPI+MEMD	N_{VAX}	$P = (fc/(CPI+MEMD)) * N_{VAX}$
68020	6,3	1	3,968 MIPS
R3000	1,2	0,8	16,666 MIPS

6.3 Performance Einflußfaktoren

Die Performance Formel gibt für die Rechnerentwickler an, welche Parameter mit dem Ziel der Performance-Steigerung beeinflußt werden müssen.

- Mit den Mitteln der Halbleitertechnik lassen sich die Schaltzeiten verkürzen und damit die Taktfrequenz erhöhen.
- Anzahl und Durchsatz der Busse, Einsatz und Aufbau von Caches verringern MEMD.
- Optimaler Befehlssatz führt zu einem n_{VAX} in der Nähe von 1.

6.4 Beispiele

Um ein Gefühl für die Leistungssteigerung der gängigen Prozessoren zu vermitteln, hier folgende Tabelle:

Prozessor	MIPS	Takt in MHz	Typ
8086	1,0	10,0	CISC
8088	0,7	8,0	CISC
80186	1,3	12,0	CISC
80188	0,9	10,0	CISC
80286	2,0	12,0	CISC
80386	3,6 10,0	16,0 40,0	CISC
68020	2,49	16,6	CISC
80486	54	66,0	CISC
MIPS R3000	24	24,0	RISC
IBM/6000 MOD 320	29,5	20,0	RISC
IBM/6000 MOD 930	37,1	25,0	RISC
Pentium	112 ...	90,0	CISC
Alpha 21064	400 ...	275,0	RISC

6.5 Performance-Messung

Aussagen über die reale Performance eines Prozessors mit Hilfe von MIPS zu machen, ist eine problematische Angelegenheit, da

- die MIPS Rate vom Befehlssatz abhängt (Befehlssatzmächtigkeit), wodurch Vergleiche zwischen Rechnern mit unterschiedlichem Befehlssatz erschwert werden,
- die MIPS-Raten zwischen verschiedenen Programmen auf dem gleichen Rechner variieren,
- die MIPS-Raten sich umgekehrt proportional zur Leistung verhalten können.

Ein Beispiel für den letzten Fall ist die MIPS-Rate einer Maschine mit einer optionalen Gleitkomma-Hardware (FPU). Da Gleitkommaoperationen mehr Taktzyklen benötigen als Festkommaoperationen, beanspruchen Gleitkommaprogramme mit FPU zwar weniger Zeit, haben aber eine geringere MIPS Rate. Die Gleitkomma-*software* benutzt einfache Befehle, was zu einer hohen MIPS-Rate führt, aber die große Befehlsanzahl ergibt insgesamt eine längere Ausführungszeit.

Nimmt man als Bewertungskriterium die Verarbeitung von Instruktionen pro Sekunde, erhält man bei einem RISC-Prozessor eine schnellere Verarbeitungsgeschwindigkeit als bei einem CISC-Prozessor. Jedoch benötigt ein CISC-Prozessor oft weniger Befehle als eine RISC-Prozessor. Beispielsweise benötigt ein RISC-Prozessor für die Addition zweier Zahlen im Speicher vier Befehle, während ein CISC-Prozessor dies mit einem Befehl erledigt. Das heißt aber nicht, das die CISC-Maschine schneller ist! MIPS (Million Instruction per Seconds) gibt lediglich Aufschluß über die *Verarbeitungsgeschwindigkeit*, nicht aber über die *Leistung*.

Man muß sich darüber im klaren sein, daß alle heute verbreiteten Programme zur Performance-Messung immer nur einen Teilaspekt der Prozessor-Performance berücksichtigen. Zur genaueren Beurteilung eines Prozessors unterscheidet man deshalb zwischen

6.5.1 Integer Performance

Gemessen wird die Geschwindigkeit der Integer-Arithmetik, von Speicherzugriffen (Load und Store), von Sprüngen und von Unterprogrammaufrufen. Ein typischer Integer-Benchmarks ist der Dhrystone-Benchmark.

6.5.1.1 Übung: Dhrystone-Benchmark

 Stellen Sie fest, wieviele Dhrystones Ihr Computer hat:

6.5.2 Gleitkomma Performance

Bei naturwissenschaftlichen Anwendungen ist die Gleitkomma-(Floating Point) Performance von großer Bedeutung. Standard-Benchmarks dafür sind der Linpack oder der Whetstone.

6.5.2.1 Übung: Whetstone-Benchmark

 Stellen Sie fest, wieviele Whetstones Ihr Computer hat:

6.5.3 Performance großer Programme

Übliche Benchmark-Programme haben den Nachteil, daß sie ziemlich klein sind; dadurch kann das gesamte Programm in einem schnellen Pufferspeicher der CPU (Cache) untergebracht sein, was zu unrealistisch schnellen Speicherzugriffen (Load und Store) führt. Als eine Art Benchmark für große Programme wird häufig das Simulationsprogramm SPICE verwendet.

6.5.4 Task-Wechsel

Bei Multi-User und Multi-Tasking Betriebssystemen geht in die reale Prozessor-Performance auch die Zeit ein, die für die Task-Wechsel benötigt wird. Auf einen Benchmark, der diesen Aspekt berücksichtigt, konnte man sich bisher noch nicht einigen.

6.5.5 Interrupt Response Time

Für Controller-Anwendungen ist die Reaktionszeit auf Interrupts von entscheidender Bedeutung; die Reaktionszeit setzt sich zusammen aus der Interrupt Response Time und zusätzlich aus der Zeit, die für das Wiederaufsetzen des Prozessors benötigt wird. Auch hierfür existiert kein Standard-Benchmark.

6.5.6 I/O Benchmarks

Diese Benchmarks messen die Geschwindigkeit der Peripherie, vor allem des File-Systems. Sie sind ein Indiz für die Leistungsfähigkeit des Gesamtsystems. Ein Beispiel ist der IOB. Mittlerweile gibt es auch Benchmarks für Datenbanken.

6.5.7 SPECmark

Um die Performance-Werte aussagekräftiger zu machen, wurde von dem Komitee SPEC eine verbesserte Methode entwickelt. Bei den SPEC Messungen müssen die Systeme (es geht also nicht mehr nur um die Prozessoren) eine festgelegte Reihe von Benchmarks absolvieren. Deren Ausführungszeit wird in Bruchteilen der Ausführungszeit einer VAX 11/780 (SPEC Reference Time) ausgedrückt. Der geometrische Mittelwert dieser Quotienten wird SPECmark genannt und soll einen objektivierten MIPS-Wert ausdrücken. Die SPEC-Angaben geben gut das Leistungsprofil der untersuchten Rechner wieder, berücksichtigen aber auch nicht alle Performance Aspekte.[1]

6.6 RISC-Architektur-Merkmale

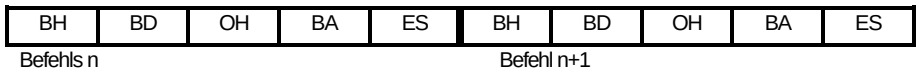
Ausgehend von der Performance-Formel lassen sich die RISC-Ziele wie folgt zusammenfassen:

- Senkung von CPI
- möglichst geringere Verschlechterung von n_{VAX}
- MEMD gegen 0

6.6.1 Ein Zyklus Operationen

Will man CPI auf Werte nahe 1 absenken, so bedeutet dies, daß pro Takt ein Befehl beendet sein muß. Wie soll das gehen ? Wie wir bereits wissen, benötigt ein Befehl allgemein einen Zyklus aus Befehl holen, Befehl dekodieren, Operand holen, Befehl ausführen und Ergebnis speichern. Dabei ist nicht garantiert, daß jeder dieser Phasen in einem Taktzyklus erledigt werden kann. Also benötigt ein Befehl mindestens fünf Taktzyklen und damit ist CPI immer noch gleich 5. Um CPI unter fünf zu senken, verwendet man ein Verfahren, bei dem die Befehlszyklen mehrerer Befehle zeitlich überlappt (Fließbandverarbeitung, Pipeline) abgearbeitet werden. Wie sieht das aus ?

Sequentielle Befehlsausführung :



Pipeline-Verarbeitung:

11 Vgl. C. Müller-Schloer, a.a.O.. S. 188 ff

Befehl n	BH	BD	OH	BA	ES					
Befehl n+1		BH	BD	OH	BA	ES				
Befehl n+2			BH	BD	OH	BA	ES			
Befehl n+3				BH	BD	OH	BA	ES		
Befehl n+4					BH	BD	OH	BA	ES	
Befehl n+5						BH	BD	OH	BA	ES

Wie Sie den beiden Bildern entnehmen können, sind bei Einsatz des Pipelineverfahrens sechs Befehle abgearbeitet worden, während bei sequentieller Verarbeitung erst zwei Befehle abgearbeitet wurden.

Man kann sich dieses Verfahren als ein Fließband vorstellen. Während Befehl n+2 sich in der Phase Befehl holen befindet, wird der Befehl n+1 dekodiert und der Befehl n holt gerade seine Operanden. Der Effekt ist, daß in jeder Phase ein Befehl abgearbeitet und somit eine fünffache Arbeitsgeschwindigkeit erreicht wird.

Man spricht im obigen Beispiel von einer **fünfstufigen Pipeline**, da der gesamte Befehlszyklus in fünf getrennte Phasen unterteilt ist. Unterteilt man den Befehlszyklus nur in die Phasen 1. Befehl holen und 2. Befehl ausführen, so kann man nur eine zweistufige Pipeline einrichten und im optimalen Fall eine doppelte Verarbeitungsgeschwindigkeit erreichen.

Dies gilt aber nur, wenn in einem Programm keine Sprungbefehle enthalten sind. Nehmen wir an, der Befehl n sei ein Sprungbefehl: der normale Befehlsfluß wird unterbrochen und der Befehl n+1 kann erst geholt werden, wenn der Befehl n vollständig abgearbeitet worden ist. Trotz dieser Schwierigkeiten wurde das Prinzip der Pipelines in modernen Rechnern perfektioniert. Es werden ganze Pipelines parallel aufgebaut, also die Befehle für beide Ergebnisse einer Bedingung (Branch Not Equal ..) vorsorglich geholt, und die Pipeline, die nicht verwendet wird, wird wieder verworfen, wenn das Ergebnis der Bedingung vorliegt.

Zurück zur fünfstufigen Pipeline. Sie ist mit einer klassischen von Neumann Maschine (Minimalsystem!) nicht realisierbar. Warum ?

In den Phasen BH, OH und ES wird die selbe Hardwareeinheit, nämlich der Daten-/Befehls-/Adressbus benötigt. Wie im obigen Pipeline-Diagramm an den schraffierten Flächen erkennbar, kollidiert der Befehl n+4 bei ´Befehl holen´ mit Befehl n+2 bei ´Operand holen´ und mit Befehl n bei Érgebnis speichern´. Daran läßt sich schon ablesen: je mehr Stufen eine Pipeline enthält, desto mehr Aufwand muß bei der Ausstattung des Prozessors mit Hardware-Resourcen getrieben werden um Pipeline-Engpässe zu vermeiden.

Aus der für CPI = 1 nötigen möglichst einfachen Pipeline ergibt sich somit die Forderung nach einem regulären Befehlssatz mit wenigen unterschiedlichen Befehlsformaten und Adressierungsmodi. Dies schließt i.d.R. komplexe Befehle aus. Der "kleine Befehlssatz", welcher dem RISC den Namen gegeben hat, ist also nur eine Folge der Forderung nach einer einfachen Pipeline.

Kleine Befehlssätze haben willkommene Nebeneffekte: sie können realisiert werden durch festverdrahtete Steuerungen. Auf Mikroprogrammierung, einen der Auslöser für CISC-Entwicklung, kann verzichtet werden. Dieser Wegfall einer Interpretationsebene bewirkt eine schnellere Dekodierung der Befehle.

6.6.1.1 Erarbeitung: Befehlsformate und Adressierungsarten

1. Welche Adressierungsmodi kennen Sie ? Benutzen Sie hierfür Ihre Unterlagen aus Maschinenorientierter Programmierung oder Ihre reichhaltigen Erfahrungen. Weshalb sind unterschiedliche Adressierungsmodi ungünstig für eine Pipeline-Organisation ?
2. Was sind unterschiedliche Befehlsformate ?
3. Weshalb sind unterschiedliche Befehlsformate ungünstig für eine Pipeline-Organisation ?

6.6.2 LOAD-STORE Architektur

Versucht man, alle Befehle in ein einheitliches Belegungsschema zu pressen, dann machen alle diejenigen Befehle Schwierigkeiten, welche Speicherzugriffe enthalten, da zum einen Speicherzugriffe relativ lange dauern und u.U. in ihrer Dauer nicht vorhergesagt werden können, z.B. im Fall von Cache-Misses (d.h. ein Befehl ist im Befehlscache nicht enthalten und muß aus dem Hauptspeicher geladen werden; aber dazu später mehr). Bei CISC-Prozessoren ist es üblich, daß alle Befehle mit allen Adressierungsarten kombiniert werden können (orthogonaler Befehlssatz). Deshalb kann das Holen eines Operanden im einfachsten Fall einen Registerzugriff, im kompliziertesten Fall einen mehrfach indizierten Hauptspeicherzugriff bedeuten. Dies läßt sich nur unter Schwierigkeiten ohne Pipeline-Konflikte abarbeiten.

RISC-Befehlssätze trennen deshalb den Speicherzugriff von den Verarbeitungsbefehlen. Letztere holen ihre Operanden aus den Registern und speichern das Ergebnis in ein Register ab (Reg-Reg-Befehl). Für den Speicherzugriff gibt es nur die beiden Befehle LOAD zum Laden ins Register und STORE zum Abspeichern.

Da Speicherzugriffe immer Zeit kosten, werden sie so weit wie möglich vermieden. Man erreicht dies durch eine Lokalhaltung und Mehrfachverwendung von Daten.

6.6.3 Lokalhaltung von Daten

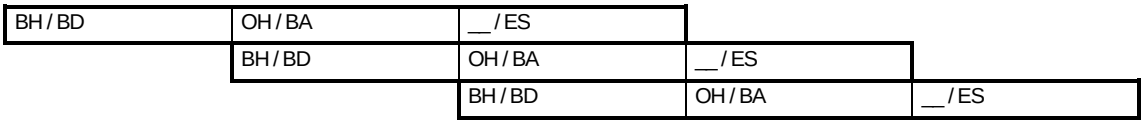
Lokalhaltung von Daten bedeutet großen Registerbedarf. Je größer die Registerzahl, desto geringer ist der dynamische Anteil von Speicherzugriffsbefehlen, wie der folgende Vergleich zeigt:

Prozessor	Anzahl Register	%Speicherzugriffe
Stanford MIPS	16 Register	35
IBM 801	32 Register	30
Berkeley RISC	138 Register	15

6.6.4 Befehls-Pipelining und optimierende Compiler

Im Zusammenhang mit der CPI-Reduktion auf 1 wurde bereits die Notwendigkeit und das Prinzip des Befehls-Pipelinings besprochen. Infolge von Abhängigkeiten zwischen aufeinanderfolgenden Befehlen kann es notwendig werden, Wartezyklen oder Leerbefehle einzuschieben, die jedoch CPI-erhöhend wirken. Für RISC-

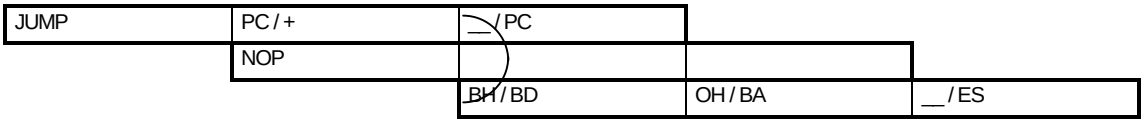
Prozessoren kommt damit dem Compiler eine wichtige Rolle zur Vermeidung von Pipeline-Konflikten zu. Die 3-stufige Befehlspipeline des Berkeley-RISC II Prozessors ohne Auftreten von Konflikten zeigt folgendes Bild:



Es können folgende Konfliktsituationen entstehen:

6.6.4.1 Steuerfluß-Konflikt

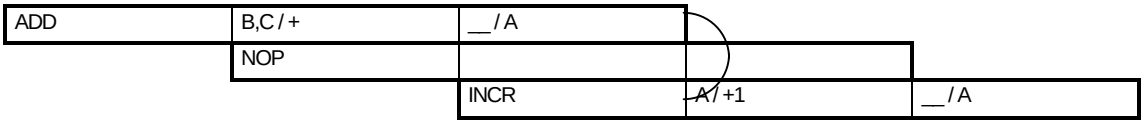
Ein Sprungbefehl besteht im Lesen des Befehlszählers (Program-Counter PC), seiner Modifikation und dem Zurückschreiben in das PC-Register.



Da der folgende Befehl während der BH-Phase den PC-Inhalt zur Adressierung benützt, müssen zwei Leerbefehle (NOPs) eingeschoben werden. Kann der neue PC-Stand direkt am ALU-Ausgang abgeholt werden (Forwarding, s. Kapitel "Pipelining"), ist nur ein NOP nötig.

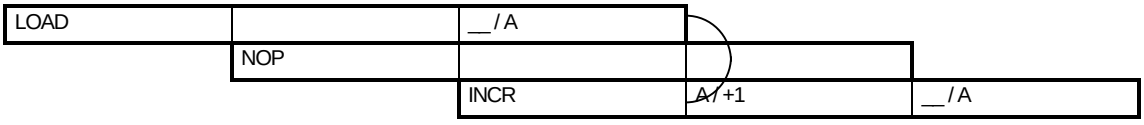
6.6.4.2 Datenfluß-Konflikt

Wird das Ergebnis eines Befehls (A im folgenden Bild) im Folgebefehl benützt, ist ebenfalls ein NOP einzuschieben.



6.6.4.3 Lade-Konflikt.

Benötigt ein Befehl das Ergebnis eines LOAD, muß ebenfalls ein NOP eingeschoben werden.



6.6.4.4 Auflösung von Pipelinekonflikten ohne Performance-Verluste

Derartige Konflikte, die die Performance einer Pipeline verringern, können durch zwei Maßnahmen vermieden werden:

1. Der Folgebefehl, welcher ein Ergebnis des Vorgängers benötigt, wartet nicht, bis es im Register abgespeichert ist (ES), um es anschließend wieder auszulesen, sondern holt es sich direkt vom ALU-Ausgang über zusätzliche Datenwege (Forwarding)
2. Eine häufig angewandte Technik zur Vermeidung von Auslastungslücken ist es, den Compiler damit zu beauftragen, die Lücken (**branch-delay slots**) der Pipeline, die durch die Verzögerung zwischen Holen und Durchführen eines Sprungs entstehen, mit sinnvollen Befehlen zu füllen. Je weniger Verzögerungszyklen ein Sprung verursacht, desto einfacher ist es für den Compiler, geeignete Befehle für diesen Zweck zu finden. Befehle, die logisch gesehen im Source-Program vor dem nächsten Sprungbefehl stehen, und von diesem datenunabhängig sind, können vom Compiler in einen Delay-Slot des Sprungbefehls eingefügt werden. Im folgenden Beispiel ist der Befehl ADD 1,A unabhängig vom Sprung JUMP 105 auszuführen. Da der dem Sprung folgende Befehl (hier NOP) auf alle Fälle ausgeführt wird, kann auf diesen Platz der Befehl ADD 1,A gesetzt werden. Der Sprung wird erst beim zweiten auf ihn folgenden Befehl wirksam (delayd branch)

Adresse	normaler Sprung	verzögerter Sprung	optimierter Sprung
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JUMP 105
102	JUMP 105	JUMP 106	ADD 1,A
103	ADD A,B	NOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

6.6.5 Speicherarchitektur

Ein RISC-Prozessor muß in jedem Taktzyklus mit einem Befehl versorgt werden. Zusätzlich erfolgt durch jeden 3. bis 5. Befehl ein Zugriff auf den externen Datenspeicher. Verzögerungen bei diesen Zugriffen würden sich stark auf die Auslastung der Pipeline auswirken. Es ist daher eine notwendige Eigenschaft von RISC-Prozessoren, daß ihre Speicherarchitektur genau auf die CPU - und damit die Pipeline-Architektur abgestimmt ist. Folgende Speicherarchitektur-Merkmale sind für RISC-Prozessoren typisch:

- Harvard-Architektur: Befehle und Daten werden durch getrennte und voneinander unabhängige Busse zwischen CPU und Hauptspeicher transportiert. Durch diese Ressourcenverdoppelung werden Konflikte vermieden.
- Cache: Ein schneller Pufferspeicher enthält häufig gebrauchte Befehle und Daten. Bei einer Harvard-Architektur gibt es separate Befehls- und Daten-Caches. Typische Cachegrößen liegen bei 64KByte bis 1 MByte.
- Primär- und Sekundär-Cache: Dem externen Sekundär-Cache wird ein noch schnellerer On-Chip-Primär-Cache vorgeschaltet, der allerdings bei heutiger Technologie sehr klein sein muß.
- Virtuelle Speicherverwaltung: RISC-Prozessoren, wie auch alle modernen CISC-Prozessoren, enthalten Speicherverwaltungseinheiten (Memory Management Unit MMU) welche die Übersetzung von virtuellen in physikalische Adressen vornehmen.


Welche Konflikte werden durch die Harvard-Architektur vermieden ?

6.7 Nur die Leistung zählt: PowerPC gegen Pentium und Pentium Pro

aus: c´t 11/1996 von Carsten Meyer, Christian Persson, Peter Siering, Andreas Stiller

Über Geschmack läßt sich bekanntermaßen trefflich streiten - der eine schwärmt für Bill Gates, der andere schwört auf bunte Äpfel. Doch unter der schönen Bedienoberfläche zählen harte Fakten: Welche Prozessorfamilie verdient den Vorzug, wenn es mehr auf die schiere Rechengewalt ankommt als auf Betriebssystem und Outfit? Wir ließen die aktuellen Champions im PC-Markt, Pentium- und PowerPC-Boliden mit 200 und mehr MHz Taktfrequenz, über ausgesuchte Benchmark-Hürden um die Wette laufen.

Ein Prozessorvergleich über Plattformgrenzen hinweg ist ein vermessenes Unterfangen. Wir erheben denn auch keineswegs den Anspruch, im überschaubaren Rahmen eines Tests und mit irdischen Mitteln eindeutige `Sieger´ küren zu wollen - unter schnöder Mißachtung von Vorzügen des einen oder anderen Betriebssystems, ohne Rücksicht auf Preise, Verbreitung und Softwareangebot. Trotzdem ist die Gegenüberstellung gelegentlich notwendig, um den Nebel von Werbesprüchen und Hausmacher-Benchmarks à la iComp etwas aufzulichten. Wir fragen also: Wer bietet mehr Performance, PowerPC oder Pentium(Pro)?

Nicht für den Alltag

Sollten Sie mit Ihrem Windows-PC oder Macintosh nur Texte verarbeiten, im Web surfen oder Daten pflegen wollen (wie 90 Prozent aller Anwender), ist der in diesem Beitrag angestellte Vergleich für Sie ohnehin nur von hypothetischer Bedeutung - etwas Schnelleres als ein 486er oder ein Schmalspur-PowerPC würde sich bei Ihnen nur langweilen. Die hier gegenübergestellten Brummer sind eigentlich nichts für Otto Normalverbraucher, und selbst anspruchsvolle User dürften mit der nächstkleineren Klasse noch gut bedient sein. Gleichwohl rekrutiert sich unsere Testriege nicht aus hochgezüchteten Labormustern, sondern aus real existierendem Konsumgut - etwas teurer zwar in der Anschaffung, aber durchaus käuflich.

Freizeitfördernd

Andererseits bringen Applikationen der Art `Animationsprogramm mit Radiosity-Renderer´ oder `Bildbearbeitung für A4-Formate´ selbst aktuelle Hochleistungs-PCs ins Schwitzen, und wer mit so etwas sein Geld verdienen muß, ist für jeden Prozentpunkt an Mehrleistung dankbar - weil die Luft während der heißen Produktionsphase dank eines schnelleren Rechners nun vielleicht eine halbe Stunde kürzer brennt. Völlig praxisfern ist unser Vergleich mithin nicht. Außerdem: Interessiert uns Golf-fahrende Motorwelt-Leser etwa nicht, was Schumis Werks-Ferrari so unter der Haube hat und warum dauernd Hill gewinnt? - Na also.

Spiele lassen wir an dieser Stelle bewußt außen vor, obwohl uns natürlich klar ist, daß die meisten High-End-PCs gerade wegen ihres Unterhaltungswertes eine Nummer größer als nötig gewählt werden und der dem Manne eigene Spieltrieb die Hardware-Entwicklung maßgeblich vorantreibt (deshalb findet man die leistungsfähigste Computer-Hardware auch eher in der Daddelhalle als beim Grafik-Designer).

Der Fairneß halber haben wir die neun Kandidaten in zwei Gewichtsklassen gruppiert (siehe Tabelle Seite 272): In die untere sortierten wir die Maschinen mit Pentium und PowerPC 603e, in die obere die mit PentiumPro und

PowerPC 604e. Alle Geräte stammen aus neuester Produktion und sind im Handel, wenn auch zum Teil noch nicht hierzulande.

Zwei-Klassen-Gesellschaft

Im Mittelgewichts-Wettkampf trat ein Compaq Deskpro 5200 gegen einen Macintosh Performa 6400/200 und einen Motorola PowerStack an, im Schwergewicht lief ein Compaq Deskpro 6000 gegen die PowerPC-604e-Mannschaft aus dem neuen Mac-kompatiblen Motorola StarMax 4200, dem Power Macintosh 9500/200, dem fast baugleichen Umax Pulsar 2000 und Taktfrequenz-Rekordhalter PowerTower 225 von Power Computing. Ihm zur Seite stand ein Elaborat aus Andreas Stillers redaktionseigener Tuning-Werkstatt: ein brandneues AMI-Board mit feinster Speicherausstattung. Diesem Superschwergewicht stellten wir einen etwas üppiger ausgestatteten 9500er Mac gegenüber.

Mit der Auswahl der Benchmarks taten wir uns diesmal bewußt schwer: Im Vordergrund sollte ja weniger der Gesamteindruck der Rechnermodelle stehen, sondern vielmehr die Leistung der Prozessoren in ihrer unmittelbaren Hardware-Umgebung, sprich L2-Cache und Hauptspeicheranbindung. Das Ergebnis ist eine wohldosierte Mischung aus synthetischen (wenngleich nicht ganz praxisfremden) Rechenaufgaben und rechenintensiven Anwendungs-Benchmarks.

Die synthetischen Benchmarks sind zum Teil Eigenproduktionen, aber auch Adaptionen `amtlicher´ Meßprogramme, wie etwa der HINT-Suite, die auf einer der nächsten Seiten ausführlich vorgestellt wird. Die Anwendungs-Benchmarks haben uns einiges Kopfzerbrechen bereitet, zumal in der Vergangenheit immer wieder Zweifel an der Gleichwertigkeit von Portierungen für verschiedene Plattformen anzumelden waren. Wir haben deshalb Gespräche mit Herstellern beziehungsweise Entwicklern geführt und detaillierte Auskünfte eingeholt. Nur diejenigen Programmfunktionen, die gemäß plausibler Darstellung vergleichbar gut implementiert sind, wurden in das Testprogramm aufgenommen.

Reimfrei

Leider ließ sich die eine oder andere Ungerechtigkeit nicht vermeiden, die eine oder andere Ungereimtheit nicht vollständig aufklären. Bei den Superschwergewichtlern mußten wir mit der angegebenen sehr üppigen RAM-Bestückung arbeiten, weil die Rechner mit anderen Speicherbausteinen bestimmte Features der Boards (Speicher-Interleave) nicht ausgenutzt hätten. Unsere synthetischen Benchmarks haben wir zwar mit aktuellen Compilern von Microsoft und Metrowerks übersetzt, es stellte sich aber heraus, daß Microsofts Visual C++ für PowerPC nicht immer bestens optimiert. Die Ergebnisse für den PowerPC-Rechner unter Windows NT sind also mit gewissem Vorbehalt zu betrachten - insbesondere, weil NT für PowerPC selbst mit diesem Compiler übersetzt worden sein dürfte. Ferner lagen die Applikationen für die Anwendungs-Benchmarks nicht für alle Plattformen vor, so daß Sie teils mit den synthetischen Werten vorlieb nehmen müssen.

Nestbeschmutzer

Ebenfalls nicht ganz zu vernachlässigen ist die `Randbedingung´ Betriebssystem; so beeinflußt beispielsweise die Verwendung von Windows 95 anstelle von NT die Performance bei einigen Benchmarks äußerst negativ, weil es anscheinend laufend den Cache kontaminiert. Wir haben deshalb auf den entsprechenden Plattformen immer beide Werte angegeben. Das Problem stellt sich auf dem Mac in weitaus geringerem Maße, da ein böswilliger Benchmark-Programmierer dessen kooperatives OS fast völlig zurückdrängen kann und die `obenliegende´ Applikation ohnehin bevorzugt Rechenzeit zugewiesen bekommt - was ansonsten eher von Nachteil ist.

Einer der Rechner, der PowerTower 225, hatte zuvor bei einem Besuch in einer anderen Redaktion sein Netzteil eingebüßt, das bei Inbetriebnahme noch auf US-Verhältnisse (110 Volt) eingestellt war. Unter dem unbeabsichtigten Feuerwerk litten offenbar noch andere lebenswichtige Organe: Wir hatten im Labor immer wieder mit Speicherfehlern und Ausfällen zu kämpfen. Einer der Anwendungs-Benchmarks (Photoshop) lief nur einmal,

nach dem Einbau unserer Testfestplatte aber nicht mehr. Die abgedruckten Ergebnisse sind daher nicht ganz so gut, wie sie hätten sein können.

Einen anderen High-Ender mit 604e, ein Modell der brandneuen RS-6000-Serie von IBM, müssen wir Ihnen leider ganz vorenthalten. Der Rechner traf zwar kurz vor Redaktionsschluß noch auf dem c't-Prüfstand ein, aber es gelang trotz nächtelanger Telefonkonferenzen mit IBM-Experten nicht, Windows NT 4.0 darauf zu installieren. Wir konnten nicht einmal unsere synthetischen Benchmarks unter AIX zum Laufen bringen, denn es fehlte der Lizenzcode für den mitgelieferten Compiler. Wir hoffen, die Maschine demnächst noch einmal ins Labor holen und die Ergebnisse in einer späteren Ausgabe nachliefern zu können.

Graue Theorie

Vor dem Startschuß zum Wettkampf lohnt ein Blick auf die Besonderheiten der verschiedenen Prozessor-Architekturen. Mit diesem Hintergrundwissen lassen sich die ermittelten Ergebnisse besser interpretieren, und scheinbare Widersprüche lösen sich womöglich auf:

Intel fertigt den klassischen Pentium inzwischen nicht mehr in einem teuren Keramikgehäuse, sondern bondet ihn auf einen Kunststoffträger.

Der PowerPC 603ev des Performa ist direkt auf die Platine gelötet. Im Gegensatz zu den Pentiums kommt er mit einem kleinen passiven Kühlkörper aus.

Pentium und PowerPC sind unter völlig unterschiedlichem Vorzeichen `aufgewachsen´. Während Pentium und -Pro einem mittlerweile 17 Jahre alten Geschlecht entstammen und direkte, kompatible Nachfahren des klassischen CISC-Bausteins (CISC = Complex Instruction Set Computing) 8086 von Anno 1979 sind, ist die PowerPC-Familie als Spin-Off von IBMs POWER-Architektur mit fünf Jahren noch relativ jung. Grundprinzip ihrer RISC-Bauweise (RISC = Reduced Instruction Set Computing) ist die Fokussierung auf die 10 % der CISC-Befehle, die in der Regel 90 % eines typischen Programms ausmachen - und die auch ein Compiler gern benutzt.

Reduced oder Complex

Doch trotz RISC-Idee steht die PowerPC-Architektur an Befehlsreichtum den Intel-Prozessoren kaum nach. Manch aufwendige und exotische Befehle der x86-ISA (ISA = Instruction Set Architecture) fehlen zwar (etwa ASCII-Adjust-, Tabellen- und Stringbefehle), ebenso die recht komplexen Indizierungen (Scaled-Index-Based-Befehle), dafür bietet sie viele andere, zum Teil ebenso komplexe Möglichkeiten, etwa indizierte Ladebefehle, die zusätzlich zu ihrer eigentlichen Tätigkeit das Indexregister verändern.

Ein entscheidener Unterschied zur Intel-CISC-Architektur ist die konstante Befehlslänge von 32Bit, was den Befehls-Dekoder erheblich vereinfacht. Andererseits kennt der PowerPC keine 32-Bit-`Immediates´, man kann also keinem Register direkt einen 32-Bit-Wert oder eine 32bittige Adresse zuweisen, wie es bei Intel-CPUs in 32-Bit-Umgebungen gang und gäbe ist. Um einen 32-Bit-Wert zu laden, sind beim PowerPC zwei Befehle nötig.

Adreß-Odyssee

Länger noch als bei Intel-Prozessoren dauert die Odyssee, welche die Adressen vom logischen Zustand zu den physikalischen Pins mitmachen. PowerPC beherrscht nämlich vier Mechanismen der Adreßbildung:

• direkt 32bittig (entspricht x86 ohne Paging)

• über die Block Address Translation (BAT), wozu jeweils acht Spezialregister für Daten und Instruktionen Übersetzungsblöcke von 128 KByte bis 256 MByte definieren

• über Segmentregister und Paging Unit: Auch der PowerPC kennt Segmentregister, und zwar gleich 16 Stück. Diese legen Segmente von jeweils 256 MByte Größe fest. Über die Segmentregister werden 52bittige virtuelle Adressen erzeugt, die über eine aufwendige Page Address Translation mit 4-KByte-Pages und zwei TLBs (Translation Lookaside Buffer) von je 64 Einträgen (PPC604) letztendlich die physikalische Adresse festlegen. Dieser Translationsweg entspricht ganz grob der x86-Architektur mit Segment-Deskriptoren und Paging

• über Segmentregister mit Direct Store Segment Translation. Das ist eine Art I/O-Zugriff am Cache vorbei, ein Relikt aus alten POWER-Zeiten, seine Nutzung wird nicht empfohlen

Rechtwinklig

Eine herausragende Eigenschaft von RISC-Prozessoren ist die Orthogonalität ihrer Register. Jedes der 32 Integer-Register ist `general purpose´, also bezüglich aller Operationen gleichwertig. Einen Akkumulator als Engpaß, über den viele Operationen abgewickelt werden müssen, kennt die Architektur nicht. Das ist ein erheblicher Unterschied zur x86-Architektur, wo zwar inzwischen alle 32-Bit-Register als Index benutzt werden können, aber so wichtige Funktionen wie etwa die Multiplikation immer das EAX-Register beziehungsweise das Registerpaar EDX/EAX verwenden müssen. Insbesondere ist jedoch der Intel-Befehlssatz auf nur acht Register beschränkt.

Ein weiterer wichtiger Vorteil, den viele RISC-Prozessoren der Intel-Architektur voraus haben, ist ihre `Harmonie mit C´. Die dortigen Pointerstrukturen, insbesondere die häufigen Pre- und Post-Dekrements und -Inkrements (- -I oder J+ +) sind hier oft in einem Prozessorbefehl abgebildet, während der x86 zwei Befehle benötigt. Der PowerPC unterstützt auch eine Variante des Post-Dekrements, nämlich die Update-Befehle. Dort, wo über Offset und ein oder zwei Indizes effektive Adressen berechnet werden, kann man noch ein Update-Register anfügen, das im Anschluß diese effektive Adresse aufnimmt.

In der Überzahl

Die gleiche Registerübermacht von 4 : 1 wie bei Integer trifft man auch bei den Fließkommaeinheiten (FPUs) an. Bei PowerPC ist die FPU als normale Drei-Adreß-Maschine organisiert: zwei beliebige FP-Register lassen sich verknüpfen und das Ergebnis einem dritten zuordnen. Ganz anders in der Intel-Architektur: die acht FPU-Register sind hier als Stack angeordnet, Operationen lassen sich nur auf das oberste Register im Stack anwenden. Compiler haben mit dem komplizierten Stackmanagement ihre liebe Not, Optimierungen über Registervariablen etwa sind hier kaum machbar - beim PowerPC gehört das hingegen zu den Pflichtaufgaben.

Außerdem sind viele Grundrechenarten des PPC604 deutlich schneller als beim Pentium oder PPro. Das gilt insbesondere für den wichtigen Multiply-Add-Befehl, den die Intel-Prozessoren mit zwei Befehlen erschlagen müssen. Weiterhin gibt es auch konditionierte Fließkommabefehle (FSEL). Andererseits fehlen alle transzendenten Befehle wie sin, cos, tan, ln oder exp, mit denen die Intel-Welt aufwarten kann. Hier muß der Compiler geeignete Approximationsroutinen in seiner Mathe-Bibliothek bereithalten, was den erzeugten Code natürlich aufbläht.

Cache und Co.

Sowohl Pentium und PentiumPro als auch die PPC60x-Prozessoren sind superskalar, das heißt, sie haben mehrere parallel arbeitende Funktionseinheiten. Zu den Integer-Rechenwerken (ALU = Arithmetic Logic Unit) gesellen sich die Fließkomma-, Load/Store- und die Sprungvorhersage-Einheit (Branch Unit). Letztere sorgt dafür, daß möglichst selten Wartezeiten durch Programmverzweigungen entstehen. Der Prozessor merkt sich in einem Branch History Buffer, welche Sprungmuster an welchen Adressen aufgetreten sind, um beim nächsten Mal mit möglichst hoher Trefferquote den weiteren Programmweg vorherzusagen.

Ganz komplex: Seine Leistungsfähigkeit unterstreicht der PentiumPro mit dem größten IC-Gehäuse aller Zeiten. Neben dem CPU-Chip findet sich darin auch (separat) der L2-Cache.

Etwas reduced: Das modernste Gehäuse besitzt der PPC604e. Lötperlen statt Beinchen auf der Unterseite sorgen für eine einfache automatische Montage, hier im Motorola StarMax.

Der Pentium hat zwei einfach gestrickte Pipelines. Es ist Aufgabe des Compilers, Abhängigkeiten zwischen den Befehlen möglichst zu entkoppeln, damit sie parallel ausgeführt werden können. Im Pentium ist der Parallelisierungsgrad daher recht klein. Hinzu kommt, daß eine Pipe warten muß, bis die andere fertig ist. So dreht sie zum Beispiel bei einer Division gut 40 Takte lang Däumchen.

PentiumPro und die PowerPCs (insbesondere der PPC 604) sind hier erheblich leistungsfähiger. Eine Vielzahl von Mechanismen sorgt für eine effiziente Befehlsentkopplung. Ein wichtiger `Trick´ dabei ist das Register-Renaming. Physikalische Register sind hierbei nicht mehr statisch einem logischen Register zugeordnet, sondern der Prozessor kann sie nach Bedarf ummappen. Außerdem müssen Befehle oft nicht unbedingt in der Reihenfolge ausgeführt werden, in der sie im Programm aufeinanderfolgen. Die modernen Prozessoren ordnen sie um und sorgen am Ende über einen Reorder-Buffer dafür, daß die Zugriffsreihenfolge wieder stimmt. Ein weiteres Bonbon sind Zwischenspeicher von ausdekodierten Befehlen, die auf eine freiwerdende Unit warten, die sogenannten Reservation States.

Der PPC603 hat recht wenig an parallelen Units zu bieten: er weist nur eine Integer-Unit auf. Demgegenüber kommt der PentiumPro mit zwei und der PPC604 gleich mit drei Integereinheiten daher, wobei die dritte Einheit nur für langandauernde Befehle (MUL, DIV) vorgesehen ist. Reservation States haben nur PPro und PPC604 zu bieten, der PPro speichert bis zu 20 `µOP´ genannte Befehle zentral zwischen, der PPC604 hat dezentral an jeder Funktionseinheit kleinere Speicher.

Alle hier betrachteten Prozessoren besitzen getrennte Code- und Daten-Caches. Pentium-P54C und PPRo haben je 8 KByte, Pentium-55C (demnächst) und PPC603e je 16 KByte, während der PPC604e mit zweimal 32 KByte Cache der L1-Cache-König ist. Ungeschlagen bei der L2-Performance ist hingegen der PentiumPro, da hier der im Gehäuse integrierte L2-Cache mit der vollen Prozessorgeschwindigkeit betrieben wird, während alle anderen nur mit dem erheblich niedigeren exteren Systemtakt auf den externen L2-Cache zugreifen können.

Der Pentium weist eine besonders gute L1-Cache-Anbindung auf. Mit 256 Bit Breite gelangen die Instruktionen in den Decoder. Misalignments (Zugriffe auf ungerade Adressen und Daten) kann der Pentium nahezu problemlos wegstecken - die anderen Prozessoren haben hiermit mehr Probleme.

Alle vier Prozessoren transferieren ihre Daten zumeist in einem Burst von vier aufeinanderfolgenden Zugriffen vom und zum L2-Cache oder Hauptspeicher. Beim PPC603 kann ein externer Zugriff 32bittig oder 64bittig sein, bei den andern ist er immer 64bittig. Wie schnell die Bursts im Endeffekt sind, hängt von der Umgebung, also dem Chipsatz und der Art des Cache-Speicherinterface (synchron/asynchon) ab.

Vermessen

Der erste Teil unserer Benchmark-Schar besteht aus mehr oder weniger komplizierten synthetischen Aufgaben. Eine der interessantesten Übungen ist der HINT-Bench, 1994 von der amerikanischen Regierung beim Ames Laboratory in Auftrag gegeben mit der Zielsetzung, die Rechenleistung von Supercomputern verschiedener Provenienz vergleichen zu können. Das Ames-Lab entwickelte einen beliebig skalierbaren Benchmark, mit dem man auch die Leistung eines Taschenrechners, ja, sogar die eines College-Studenten in Zahlen fassen könnte.

HINT steht für Hierarchische INTegration. Dem Bench zugrunde liegt die Idee, daß man nicht willkürlich irgendwelche Ausführungszeiten von mehr oder weniger beliebigen Befehlen oder Befehlsfolgen abstoppt, sondern eine skalierbare Aufgabenstellung erfindet, deren Berechnung am Anfang recht einfach ist und die dann immer komplexer wird. Das Ergebnis von HINT ist mithin auch keine Ausführungzeit, sondern eine Kurve, die die Qualitätsverbesserung pro Zeiteinheit darstellt (QUIPS = Qualtity Improvement Per Second).

Die Aufgabenstellung ist in der Tat recht einfach. Viele Leser entsinnen sich gewiß noch, wie sie dereinst die Integration lernten, nämlich mit den Ober- und Untersummen. Man zählte auf dem Millimeterpapier die Punkte aus, die oberhalb oder unterhalb der Schnittpunkte der betrachteten Intervalle lagen. So arbeitet auch HINT. Die ausgesuchte, recht harmlose Funktion ist (1-x)/(1+x) im Intervall [0, 1]. Sie ist dort streng monoton fallend, was die Auswertung sehr vereinfacht. Das korrekte Integralergebnis wäre übrigens 2ln(2)-1, ist also transzendent.

Statt Millimeterpapier dient bei HINT ein Gitter als Rastermaß, das durch die Darstellungsgenauigkeit der Zahlen vorgegeben ist. Bei vorzeichenbehafteten Integern mit 15 Bits beträgt das Gitter 7 × 8 Bit. Der Algorithmus ist so gewählt, daß er für den Funktionswert f in der Intervallmitte die Zahl der Gitterquadrate bestimmt, die sich sicher innerhalb der Kurve (also links unterhalb von f) beziehungsweise sicher außerhalb (rechts oberhalb von f) befinden. Die verbleibenden `unsicheren´ Quadrate im Verhältnis zur Gesamtzahl sind ein Maß für die Genauigkeit (Quality) des Zwischenergebnisses. Dann halbiert man die Intervalle und wiederholt dieses Spiel für die restlichen Quadrate, bis letztendlich nur noch die wegen begrenzter Zahlendarstellung nicht weiter zuordenbaren Quadrate übrig bleiben. Bei Erreichen der Genauigkeitsgrenze stoppt der Algorithmus.

Der vorgeschriebene Algorithmus führt zu einer typischen Befehlsfolge, in der einfache Lade- und Speicherbefehle gegenüber komplexeren Operationen deutlich in der Überzahl sind. Ähnlich verhält es sich in den meisten Anwendungen; die HINT-Entwickler John L. Gustafson und Quinn O. Snell gehen deshalb davon aus, mit ihrem Test sehr viel besser als mit anderen synthetischen Benchmarks die durchschnittliche Anwendungsperformance von Rechnern voraussagen zu können. So verteilen sich die Befehle:

Index-Operationen	Daten-Operationen
39 Adds/Subtracts	69 Fetches/Stores
16 Fetches/Stores	24 Adds/Subtracts
6 Shifts 10 Multiplies	
3 Cond. Branches	2 Cond. Branches
2 Multiplies	2 Divides

HINT legt alle Zwischenergebnisse in Feldern ab, so daß mit zunehmender Laufzeit immer mehr Speicher alloziert wird. An der HINT-Kurve läßt sich dann deutlich der Einfluß des Cache, des Hauptspeichers und gegebenenfalls des Pagings ablesen. Bei höherer Genauigkeit begrenzt der verfügba- re Speicher die Laufzeit. Wir haben für diesen Test eine Speicherobergrenze von 25 MByte vorgegeben; der Speicher wird vor Beginn der Messung bereits einmal alloziert und benutzt, so daß die späteren Ergebnisse bei der gewählten RAM-Bestückung von 32 MByte auch unter Windows NT nicht mehr durch das Auslagern auf die Festplatte beeinträchtigt wurden.

Der HINT-Benchmark zählt innerhalb eines Intervalles die Punkte aus, die oberhalb oder unterhalb der Kurve (1-x)/(1+x) liegen.

Um dem Wunsch nach einer einzigen Benchmark-Zahl nach- zukommen, haben die HINT-Entwickler den Net-QUIPS-Wert definiert als das Integral unter der QUIPS-Kurve bei logarithmischer Zeitachse. Ein Student kommt demnach auf etwa 0,1 Net-QUIPS, Supercomputer wie Intels Paragon mit 1840 Prozessoren schaffen 633 Millionen Net-QUIPS.

Kurvenreich

Die ermittelten QUIPS-Kurven haben einen hohen Informationswert. Unter anderem läßt sich an ihrem Verlauf der Einfluß der Caches und der Speicher-Performance verfolgen. Wir haben HINT sowohl mit Integer-Variablen (32 Bit) als auch mit Double-Genauigkeit (64 Bit Fließkomma) implementiert. (Auch die gängigen

Windows-Compiler arbeiten mit 64 Bit Genauigkeit,obwohl die Intel-kompatiblen Prozessoren intern mit dem 80-Bit-Standardformat rechnen.)

So legten die Power Macs beim Double-HINT aufgrund ihrer fixen Fließkommaeinheiten zunächst gewaltig los und lagen zum Teil mehr als 20 Prozent über der PentiumPro-Leistung unter Windows NT, gerieten dann aber mit zunehmender Datenmenge aufgrund des langsameren externen Caches etwas ins Hintertreffen. Beeindruckend war die Leistung desPentiumPro-Chips auf dem AMI-Board bei sehr großen Datenmengen (wiederum unter Windows NT), was auf ein hervorragendes Hauptspeicherinterface schließen läßt. Enttäuscht hat dagegen die Leistung aller Pentiums unter Windows 95, das offenbar massiv in die Cache-Kohärenz eingreift und vor allem bei den `dicken´ Pentiums die Geschwindigkeit im Extremfall auf weniger als die Hälfte (relativ zu NT) drosselt.

Beim Integer-HINT (unter Windows NT) lagen die PentiumPro-Brüder dagegen leicht vorn, dicht gefolgt von der PowerPC-Familie. Der `kleine´ Pentium, der noch gut 25 Prozent schlechtere Ergebnisse lieferte als der PowerPC603 im Performa, landete abgeschlagen auf dem letzten Platz.

p mal Daumen

Die Berechnung von p auf irrwitzig viele Stellen ist als Benchmark schon seit archimedischen Zeiten beliebt. Der diesem Benchmark-Programm zugrundeliegende Algorithmus basiert aber nicht auf einem der klassischen Verfahren, sondern wurde erst letztes Jahr von Bailey, Borwein und Plouffe entdeckt.

Er konvergiert ausgesprochen schnell, so daß sich 10 000 Stellen in wenigen Sekunden berechnen lassen. Beschränkt man sich auf weniger als 100 000 Stellen, läuft der Bench vollständig im L2-Cache ab.

Der HINT-Benchmark liefert seine Ergebnisse in MQUIPS (Millionen `quality improvements´ pro Sekunde). Je nach belegter Speichermenge ändert sich die Geschwindigkeit, und die Grenzen von primärem und sekundärem Cache zeigen sich in der Kurve ebenso wie die Zugriffsgeschwindigkeit des Haupspeichers.

Als wesentliche Operation geht in den Pi-Bench die doppeltgenaue Division ein. Unser Programm verzichtet jedoch weitgehend auf direkte Divisionen - da sonst ein einziger Befehl die Gesamtperformance bestimmen würde, sondern führt sie `zu Fuß´ mit Standardbefehlen wie Schieben und Subtrahieren, Vergleichen und Verzweigen aus. Würde man etwa auf einem Intel-Prozessor für die 64/32-Bit-Division statt dessen den vorgesehenen Prozessorbefehl benutzen, könnte man gut die sechsfache Performance erreichen. Doch für Benchmarkzwecke ist langsamere portable Lösung sinnvoller.

Für diesen Test haben wir uns auf eine Genauigkeit von 8192 Stellen beschränkt. Angegeben ist der Gesamtwert für das Ermitteln aller Stellen. c't wird in einer der nächsten Ausgaben den Algorithmus samt C-Programm von Sebastian Wedeniwski genauer vorstellen. Wie Sie der Grafik auf Seite 278 entnehmen können, liegen die PowerPC-bestückten Rechner hier sehr gut im Rennen: Selbst der Apple Performa ist um mehr als ein Drittel schneller als der beste Pentium.

Schmetterlinge im Cache

Ein weiterer Klassiker bei den wissenschaftlichen Benchmarks ist die schnelle Fourier-Transformation (FFT). Hierzu dient in aller Regel der sogenannte Butterfly-Algorithmus, wie er in [1] beschriebenen wurde. Das vorliegende Benchmarkprogramm für eine komplexe FFT von 1024 Werten nach John Greene ist hochoptimiert und arbeitet mit einer größeren Folge aufeinanderfolgender Multiply/Add-Operationen in doppelter Genauigkeit. Prozessoren mit vielen Floating-Point-Registern profitieren besonders davon - das kommt natürlich dem PowerPC mit seinen 32 FP-Registern entgegen.

Syscomp mißt die Zahl ausgeführter Operationen pro Sekunde, einmal für Integer-Daten (MOPS: Millionen Operationen pro Sekunde) und einmal für Fließkomma-Daten (MFLOPS: Millionen Fließkommaoperationen pro Sekunde).

Hinzu kommt, daß die uns bekannten Intel-C-Compiler zwar für Registervariable ausgefuchste Optimierungen beherrschen - aber nur für Integer. Für Floatingpoint-Register wird üblicherweise mühsam hin- und hergeladen - wegen ihrer stackartigen Verwaltung ist es für Compiler ausgesprochen schwierig, hier zu optimieren. Tatsächlich erledigten die PowerPCs die gestellte Aufgabe fünfmal (PPC603e) oder gar zehnmal (PPC604e) schneller (!) als die Intel-Prozessoren - der Power Tower 225 von Power Computing berechnet die 1024-Punkte-FFT in atemberaubenden 220 µs, während der schnellste PentiumPro dafür 2,78 ms benötigt.

Obwohl der FFT-Benchmark extrem `lokal´ ist, da er praktisch vollständig im prozessor internen Cache läuft, sah das Ergebnis für den PowerStack von Motorola (unter NT 4.0) mit 0,95 ms um 20 % schlechter aus als für den hier gleichwertigen Performa 6400. Beim Disassemblieren zeigte sich, daß Microsofts C++-Compiler 15 Floatingpoint-Register des PPC 603 ungenutzt ließ; der Metrowerks-Compiler für MacOS benutzt dagegen alle bis auf zwei. Obendrein gestattet Microsofts C++ als einzige globale Einstellung für den Funktionsaufruf die `cdecl´-Konvention (Parameterübergabe über den Stack), bei der Intel-Version des Compilers wird dagegen auch `fastcall´ (Parameterübergabe über Register) angeboten. Dabei wäre letztere gerade bei einem RISC-Prozessor mit vielen Registern durchgehend angebracht. Wir haben den FFT-Test schließlich mit einer zufällig vorhandenen alten Beta eines Motorola-Compilers noch einmal übersetzt und damit das abgedruckte Ergebnis erzielt, immerhin um gut 10 % besser als mit dem aktuellen Microsoft-Compiler.

Niedere Mathematik...

Hinter dem c't-eigenen Syscomp steckt ein weiterer synthetischer Benchmark. Das in ANSI-C realisierte Programm versucht, soweit das in einer Hochsprache überhaupt möglicht ist, die Prozessor-Performance in verschiedenen arithmetischen Disziplinen zu ermitteln. Syscomp tut dies, wie HINT, für die Datentypen int und für double. In handgetrimmten Meßschleifen wiederholt es dazu Rechenoperationen in einem vorgegebenen Adreßraum solange, bis eine Mindestlaufzeit erreicht ist.

Beim Datentyp int setzt sich der Benchmark aus Addition, Multiplikation und Division zusammen, beim Datentyp double gesellen sich Logarithmus, Tangens und Quadratwurzel dazu. Für jede dieser Disziplinen bestimmt Syscomp die Anzahl der pro Millisekunde ausgeführten Operationen. In das Endergebnis (gewichtetes Mittel) gehen die einzelnen Disziplinen mit folgender Wertigkeit ein:

Typ	int	double
Add	20	20
Mul	10	10
Div	5	5
Tan		1
Ln		1
Sqrt		1

Trotz der ausgefeilten Meßschleifen fließen natürlich Speicherzugriffe in das Ergebnis ein - ganz und gar lassen sich Operationen und Operanden eben nicht trennen. Der Aufbau der Meßschleifen garantiert jedoch weitestgehend, daß sich die Ergebnisse auch bei Einsatz hochoptimierender Compiler nicht besonders verbessern lassen. Um die Abhängigkeit von Speicherzugriffen zu ermitteln, wiederholt Syscomp die Messungen auf einer wachsenden, immer gleich vorbesetzten Speichermenge. Aus den separat ausgewiesenen gewichteten Resultaten ergeben sich die abgebildeten Performance-Kurven.

Wie schon beim HINT-Benchmark tritt bei den Syscomp-Kurven deutlich die Speicherabhängigkeit hervor, und ebenso deutlich fällt das miserable Ergebnis unter Windows 95 auf - trotz hochaktueller 32-Bit-Compiler. Na-

türlich kann sich der PentiumPro mit zunehmendem Datenaufkommen aufgrund seines großen internen L2-Caches länger `über Wasser halten´ als der PowerPC, der hier nur bei Fließkomma-Arithmetik im internen Cache Vorteile für sich verbuchen konnte. Das wiegt allerdings stärker, weil ihm Prozessorbefehle für die transzendenten Operationen fehlen und der Compiler diese durch Bibliotheksfunktionen nachbilden muß.

...und die höhere

MuPAD ist ein kostenlos verfügbares Computeralgebrasystem der UGH Paderborn (siehe c't 7/95, S. 194). Sowohl Windows-95/NT- wie Mac-Version sind im wesentlichen in ANSI-C geschrieben und nutzen kaum hardwarespezifische Eigenschaften aus. Insbesondere verzichtet MuPAD auf jegliche Fließkomma-Unterstützung durch den Prozessor. Kompiliert wurden die MuPAD-Versionen mit maximaler Optimierungsstufe von Microsoft Visual C++ 4.1 beziehungsweise Metrowerks Codewarrior 8.

Die in MuPAD eingebaute Langzahlarithmetik PARI, die das Rechnen mit ganzen Zahlen übernimmt, enthält einen kleinen handoptimierten Assemblerkern. Daher haben wir als Benchmarks Aufgaben gewählt, die wenig auf PARI zugreifen. Weil MuPAD zu Beginn eines jeden Tests einige Routinen von der Festplatte lädt, ließen wir jede Rechnung mehrfach hintereinander ablaufen; die stark von der Festplatte bestimmte Dauer der ersten Rechnung ignorierten wir.

Arithmetik wird in MuPAD (überraschenderweise) verhältnismäßig klein geschrieben. Der überwiegende Anteil an Rechnungen sind die Manipulationen der recht komplexen internen Ausdrucksbäume. Dabei kommt es hauptsächlich zu sehr vielen Speicherzugriffen, meist mehrfach indirekt, zu vielen Vergleichsoperationen und zu häufigem Anfordern und Freigeben von Speicherblöcken. MuPAD arbeitet ferner oft mit Rekursionen, teilweise mit sehr hoher Schachtelungstiefe (1000 und mehr). Wegen der Größe des Algebrasystems sollten RAM-Caches nicht allzuviel Wirkung zeigen, es kommt also primär auf die Effizienz der Hauptspeicherzugriffe an, so ein Programmierer von MuPAD.

Die MuPAD-Benches bestanden aus der symbolischen Berechnung eines Grenzwerts, der symbolischen Auswertung eines Integrals, der Berechnung des 20sten Elements der Fibonacci-Folge (1, 1, 2, 3, 5, 8, ...) durch 220 Rekursionen und schließlich der symbolischen Berechnung einer unendlichen Summe. Bei diesen Tests lagen eindeutig die `dicken´ PentiumPro vorn, die rote Laterne durften der Compaq-Pentium unter Windows 95 und der 603ev im Performa gemeinsam tragen.

Formelknacker

Wolfram Mathematica 2.2.3 für Windows (95 und NT mit Patch) und MacOS ist eines der am weitesten verbreiteten Computeralgebrasysteme (siehe o.g. c't). Es beherrscht sowohl symbolische Rechnungen (das Auswerten algebraischer Formeln) wie auch numerische Aufgaben (das `übliche´ Rechnen mit Zahlen).

Auf dem Mathematica-Programm standen: die Berechnung von Primzahlen, eine aufwendige Fließkomma-Rechnung, die 3D-Darstellung der `Kleinschen Flasche´ und die Darstellung eines komplizierten 3D-Körpers. Die letzteren drei Aufgaben machen rege Gebrauch von den Fließkomma-Fähigkeiten des Prozessors. Das konnten die PowerPCs zu ihrem Vorteil nutzen: Halb so lange Ausführungszeiten wie auf den schnellsten Pentiums waren hier keine Seltenheit, selbst der Sparprozessor des Performa konnte noch gut mithalten. Ein Trauerspiel wiederum die Ergebnisse unter Windows 95 - die PowerPC-Riege war teilweise mehr als zehnmal (!) so schnell.

Bücherwurm

Für einen Real-World-Test in der Disziplin Datenbanken verwendeten wir die Search Engine von eMedia, deren Browser-Version c't-ROM-Anwendern als eMedia Navigator bekannt ist. Das Programm liegt uns im Quelltext vor; wir haben es für Windows mit Microsofts Visual C++ und für MacOS mit Metrowerks Code Warrior Version 10 kompiliert.

Beim Suchen im Volltext geben Festplatte oder CD-ROM das Tempo vor, deshalb eignet sich dieser Vorgang kaum als Prozessor-Benchmark. Anders sieht es bei der Erstellung eines Volltext-Index aus. Der schnelle Indizierer in der Produktionsversion der Suchmaschine liest und schreibt ausschließlich größere Datenblöcke von zumeist 32 KByte Länge. Die Festplatte wird also mit maximaler Geschwindigkeit betrieben, und die Massenspeicherzugriffe machen nur den kleineren Teil der Laufzeit aus. Natürlich haben wir trotzdem alle Tests mit derselben Platte durchgeführt.

Die Aufgabe besteht darin, eine Textdatei von 10 MByte Länge für die `Fuzzy Logic´-Volltextsuche zu indizieren. Der Indizierer bildet dabei unter anderem eine sortierte Liste aller vorkommenden Wörter und ihrer Positionen in der Datei. Er legt im Speicher eine große Datenstruktur aus Bäumen und verketteten Listen an; virtueller Speicher und das damit verbundene Swapping werden strikt gemieden. Das Ergebnis ist eine frappierend schnelle Volltextindizierung. Die 32-Bit-Indizes werden zwecks Platzersparnis noch `komprimiert´ und in überwiegend 16bittigen Strukturen abgelegt.

Daß die PowerPCs in diesem Benchmark deutlich schlechter abschneiden als die Pentium-Familie, ließ zunächst eine Diskrepanz zwischen ihrer 32-Bit-Architektur und den 8-Bit-Textdaten vermuten. Dem ist aber nicht so: der Byte-Ladebefehl kostet, wie beim PentiumPro, nur zwei Prozessortakte. Die Programmanalyse mittels Profiler erwies denn auch, daß ein Großteil der Laufzeit beim Abklappern der weit im Speicher verzweigten Bäume und verketteten Listen draufgeht. Ihren deutlichen Sieg verdanken die Pentiums demnach vornehmlich dem schnelleren Zugriff auf den Hauptspeicher.

Hinzu kommt ein Einfluß der beiden Windows-Varianten im Vergleich zu MacOS: Dessen Massenspeicher-Cache wird nur beim Lesen wirksam, der von Windows dagegen auch beim Schreiben. Da der Indizierer temporäre Dateien schreibt und kurz darauf wieder liest, werden unter Windows einige Plattenzugriffe quasi übersprungen. Dennoch blieb auch Motorolas PowerStack unter Windows NT hinter den Erwartungen zurück; möglicherweise hat der Microsoft-Compiler auch hier das Programm für den PowerPC weniger gut optimiert als für den Pentium.

Bilderbuchbench

Mit einem CMYK-Bild von 1430 × 1793 Pixeln Größe (Datenvolumen knapp 10 MByte) führten wir Teile unseres Standard-Benchmarks für Photoshop aus, deren Eignung für Cross-Plattform-Benchmarks Adobe uns ausdrücklich bestätigt hat: Drehen des Bildes um 1 Grad (bikubische Interpolation eingeschaltet), Gaußscher Weichzeichner (Stärke 50 Pixel), Scharfzeichnungsfiler `Unscharf Maskieren´ (50 %, 3 Pixel, Schwelle 18 Stufen), Farbraumwandlung von CMYK in RGB (Werkseinstellungen). Alle diese Benchmarks prüfen vor allem Integer-Leistung und Speichertransfer. Da Photoshop mit seinem eigenen virtuellen Speicher für viele Festplattenzugriffe sorgt, haben wir alle Messungen mit derselben Platte ausgeführt. Wie stark die Ergebnisse davon bestimmt werden, zeigen die Meßwerte in der Superschwergewichtsklasse: Bei dem AMI-Board mit seiner Grundausstattung von 128MByte RAM mußte Photoshop überhaupt nicht mehr swappen - ebenso wenig wie bei dem Power Mac 9500 mit 80 MByte RAM. In diesem Licht sind auch die übrigen Ergebnisse zu sehen.

Alltägliche Aufgabe für das 3D-Programm Cinema 4D: Raytracing eines Trickfilms mit animierten Objekten.

Leider lag uns keine Photoshop-Implementierung für Windows NT auf PowerPC vor, so daß die Zeile für den Motorola PowerStack entfallen mußte. Die sonstigen Ergebnisse waren recht gemischt. Mit `nur´ 32 MByte Hauptspeicher brach Adobes NT-Anpassung stark ein, während Windows 95 sich teils von einer besseren Seite zeigte. Die PowerPC-Mannschaft durfte fast immer aufs Siegertreppchen, nur im Superschwergewicht lief das AMI-Goliath-Board dem Power Mac 9500 knapp den Rang ab.

Filmreif

Cinema 4D, eine plattformübergreifend verfügbare 3D-Animationssoftware, kompilierte Hersteller Maxon extra für diesen Test auch für den PowerPC unter Windows NT. Das Programm stand damit auf allen Rechnern zur Verfügung. (Einen Test der Versionen für Windows und MacOS finden Sie auf S. 72.)

Als Benchmark diente ein Film, der zeigt, wie sechs Tropfen auf einen Marmorfußboden fallen und zerfließen. Dabei handelt es sich um halbtransparente Objekte aus je 1392 Vierecken; die Tropfen wandeln ihre Form per `Morphing´. Gerendert wurden in höchster Qualitätsstufe (Raytracing) bei 2 × 2-Antialiasing ein Einzelbild (472 × 321 Pixel) sowie ein Film (240 × 160 Pixel) aus 26 Bildern - letzterer einmal mit eingeschalteter Transparenz (und Lichtbrechung) und einmal ohne, wodurch unzählige Iterationen des Raytracing entfallen.

Erstaunlicherweise fiel Windows 95 hier nicht negativ auf, so daß der Performa das Schlußlicht bildete. PowerPC 604e und PentiumPro lagen Kopf an Kopf, während der Pentium im Deskpro 5200 um mehr als 20 Prozent bessere Ergebnisse lieferte als der Performa mit seinem 603ev.

Schachmatt

Schach-Algorithmen sind ebenfalls eine beliebte Herausforderung für Rechner aller Art - und warum sollte man einen Prozessorvergleich nicht mit einem kleinen Turnier beschließen? Da die Berechnungstiefe der möglichen Züge pro Zeiteinheit bei einem schnelleren Prozessor natürlich größer ist, sollte im statistischen Mittel auch dieser gewinnen. Wir bedienten uns der Textversion von GnuChess 4.0 Patchlevel 73, um über eine TCP/IP-Verbindung den PowerPC 603e im Motorola PowerStack unter Windows NT 4.0 gegen den klassengleichen Pentium im Deskpro 5200 antreten zu lassen.

In einem GnuChess-Turnier von 53 Partien schlägt der Motorola PowerStack knapp den Compaq Deskpro 5200.

Als Leistungsmaß dient die Anzahl der in einem Spiel analysierten Züge. Unsere GnuChess-Adaption, die den Streithähnen abwechselnd die Spieleröffnung zuwies, ermittelte hier für den PowerPC im Schnitt um 18,5 Prozent höhere Werte. Kein Wunder, daß er das Turnier auch gewann: Nach einer langen Schachnacht von 53 Partien lag das unten illustrierte Endergebnis vor.

Für MacOS mangelte es leider an einer geeigneten GnuChess-Implementierung, so daß ein dem PentiumPro angemessener Gegner fehlte. Doch warum sollte der `kleine´ 603e nach dem überraschend guten Abschneiden in seiner Gewichtsklasse nicht das Schwergewicht herausfordern? - Als kleinen Ausgleich für das Untergewicht verpaßten wir dem PowerStack eine Sonderausstattung in Form von 1 MByte Sync-Cache. So beflügelt schlug er sich wacker gegen den Deskpro 6000.

Das Ergebnis war noch eine Überraschung: Der PPC 603e erreichte im Schnitt 94,2 % der Berechnungstiefe seines starken Gegners und holte in den sechs Spielen, für welche die Zeit bis zum Redaktionsschluß gerade noch reichte, zwei Siege und zweimal Unentschieden heraus - insgesamt also ein Remis.

Fazit

Obwohl die Einzelergebnisse für die konkurrierenden Prozessorfamilien insgesamt sich einigermaßen die Waage halten, kann man wohl kaum von einer ausgeglichenen Bilanz sprechen. Immerhin liegen die Resultate zum Teil dramatisch weit auseinander.

Die PowerPCs besitzen offenbar den `besseren Kern´: Im Umgang mit kleineren Datenstrukturen haben sie die Nase vorn, wie es sich besonders deutlich bei den isolierten Aufgaben der Pi-Berechnung und der Fourier-Transformation zeigt. Hier zahlt sich das modernere Konzept aus. Dessen Möglichkeiten freilich, speziell die Vielzahl der universellen Register, dürften oft nicht ausgeschöpft werden. Da haben nicht nur manche Compilerbauer noch Hausaufgaben zu erledigen. Auch die Softwareentwickler könnten sicher oftmals durch besser angepaßte Algorithmen wesentlich mehr Leistung herausholen. Damit ist allerdings in der Breite kaum zu rech-

nen, solange die Marktanteile der PowerPC-Systeme so klein bleiben, daß sich die eigenständige Softwareentwicklung dafür nicht rentiert. Dabei haben die PowerPCs für Anwendungen im umkämpften Multimedia-Bereich aus Programmiersicht gewiß den größeren Charme - so `signalfest´ wie Intels für 1997 angekündigte MMX-Prozessoren (siehe c't 6/96, S. 206) sind sie längst.

Die PowerPC-Rechner (nicht unbedingt die Prozessoren) kranken dafür an einem etwas nachlässig implementierten Hauptspeicher-Interface, worunter natürlich das Tempo bei der Bearbeitung größerer Datenstrukturen leidet. In den Apple-Kompatiblen fanden wir nur asynchrone Caches und mit 70 ns recht lahme Standard-RAMs; allein Motorolas StarMax machte mit EDO-RAM eine Ausnahme. Und nur der PowerStack lag mit 67 MHz Bustakt auf Pentium-Niveau - war aber ebenfalls bloß mit asynchronem Cache und Standard-RAM ausgerüstet.

Mit der durch Intels aktuelle Prozessoren und Chipsätze gesetzten Marke bei der Hauptspeicherbandbreite kann das PowerPC-Lager bisher nicht konkurrieren. 67 MHz Bustakt, Pipelined Burst Cache und die Unterstützung moderner, schnellerer RAM-Technologien sind hier inzwischen selbstverständlich. Dem superschnellen internen L2-Cache des Pentium Pro hat man erst recht nichts entgegenzusetzen, selbst wenn man den lediglich als teure und hochaufwendige Holzhammer-Methode abtut. Gewiß, der PentiumPro hat aufgrund seines stolzen Preises ein schlechteres Preis/Leistungsverhältnis als der PPC 604e - aber das bedeutet am Markt gar nichts, solange die kompletten PowerPC-Rechner keinen Deut billiger verkauft werden. Nichtsdestotrotz: der PentiumPro-Thron kippelt arg, und hätte sich Apple mit dem 604-Chipsatz nur ein wenig mehr Mühe gegeben, wäre er glatt umgefallen.

6.7.1 Prozessor-Vergleich: Testteilnehmer

Modell	Prozessor-/Takt	L1/L2-Cache	Haupt-speicher	Bustakt	Betrieb-ssystem
Compaq	Pentium/200	2 × 8/256 KByte,	32 MByte,	67 MHz	Win 95, Win NT 4.0
Deskpro 5200		synchron	EDO		
Apple	PPC603ev-/200	2 × 16/256 KByte,	32 MByte	40 MHz	MacOS 7.5.3
Performa 6400		asynchron			
Motorola	PPC603ev-/200	2 × 16/256 KByte,	32 MByte	67 MHz	Win NT 4.0
Power-Stack 2000		asynchron			

Pentium/PowerPC603-Klasse

Modell	Prozessor-/Takt	L1/L2-Cache	Haupt-speicher	Bustakt	Betrieb-ssystem
Compaq	PentiumPro/200	2 × 8/256 KByte,	32 MByte,	67 MHz	Win 95, Win NT 4.0
Deskpro 6000		integriert	EDO		
Motorola	PPC604e/200	2 × 32/256 KByte	32 MByte,	50 MHz	MacOS 7.5.3
StarMax 4200		asynchron	EDO		
Apple	PPC604e/200	2 × 32/512 KByte,	32 MByte	50 MHz	MacOS 7.5.3
Power Mac 9500		asynchron			
Umax	PPC604e/200	2 × 32/512 KByte	32 MByte	50 MHz	MacOS 7.5.3
Pulsar 20001		asynchron			

Power Computing	PPC604e/225	2 × 32/1024 KByte	32 MByte	45 MHz	MacOS 7.5.3
Power Tower		asynchron			

225					
AMI-Board	PentiumPro/200	2 × 8/256 KByte,	128 MByte	67 MHz	Win 95, Win NT 4.0
Goliath		integriert			
Apple	PPC604e/200	2 × 32/512 KByte,	80 MByte	50 MHz	MacOS 7.5.3
Power Mac 9500		asynchron			

PentiumPro/PowerPC604-Klasse

6.7.2 Ausführungszeiten in Prozessortakten

	PPC603	PPC604	Pentium	PPro
einfache Integer-Befehle	1/1	1/1	1/1	1/1
Integer Multipy 32x32	5/k.A.	4/2	10/9	4/1
Integer Divide	37/37	20/19	41/41	39/39
Integer Load	2/1	2/1	1/1	3/1
Integer Store	2/2	3/1	1/1	1/1
FP Load	3/1	3/1	1/1	1/1
FP Store	3/1	3/1	2/2	1/1
FP Double Multiply Add	3/1	3/1	3/1+3/1	5/1+3/1
FP Single Divide	33/33	18/18	39/39	38/38
FP Double Divide	33/33	31/31	39/39	38/38

Latenz und Durchsatz-Zeiten (Latency/Throughput) in Taktzyklen

6.8 Prozessor-Taxonomie

In den folgenden Tabellen und Diagrammen werden die wichtigsten Kenngrößen der Performance-Formel in ihrer zeitlichen Entwicklung als Arbeitsgrundlage dargestellt.

Prozessor	**Jahr**	**f_C (MHz)**	**(CPI + MEMD) / n_{VAX}**	**Typ**
8080	75	2	8	CISC
80286	83	6	6	CISC
68010	83	12,5	12	CISC
80286	85	12	6	CISC
68020	85	12	7	CISC
microVax II	85	12	10	CISC
Pyramid 9000	86	10	1,5	RISC
68020	86	25	6,3	CISC
Clipper C 100	86	33	6,7	RISC
80386	86	16	4	CISC
Ridge 3200	86	12	2,5	RISC
80386	87	24	4,5	RISC
T800	87	20	1,8	RISC
Ridge	87	16	1,1	RISC
ARM	87	12	2,0	RISC
MIPS R2000	87	16,7	1,4	RISC
SPARc Fuj.	87	16,7	1,4	RISC

T800	88	30	1,8	RISC
Edge 2000	88	21	1,3	RISC
NSC 32532	88	15	2,4	CISC
68030	88	25	3,7	CISC
MIPS R3000	88	25	1,25	RISC
Am29000	88	25	1,5	RISC
80960	88	20	2,7	CISC(?)
Edge	89	26	1,3	RISC
SPARC Cypr.	89	33	1,5	RISC
COLIBRI	89	12,5	1,16	RISC
Clipper C 300	89	50	3,6	RISC
88000	89	20	1,25	RISC
NSC 32532	89	33	2,4	CISC
IBM/6000	90	20	0,7	RISC
IBM/6000	90	25	0,7	RISC
80486	90	25	2,5	CISC
i860	90	33	0,9	RISC
IBM RS/6000	91	30	0,7	RISC
Pentium	93	>66	?	CISC
Alpha 21064	92	150	?	RISC

Prozessoren und ihre Kenngrößen der Performance-Formel

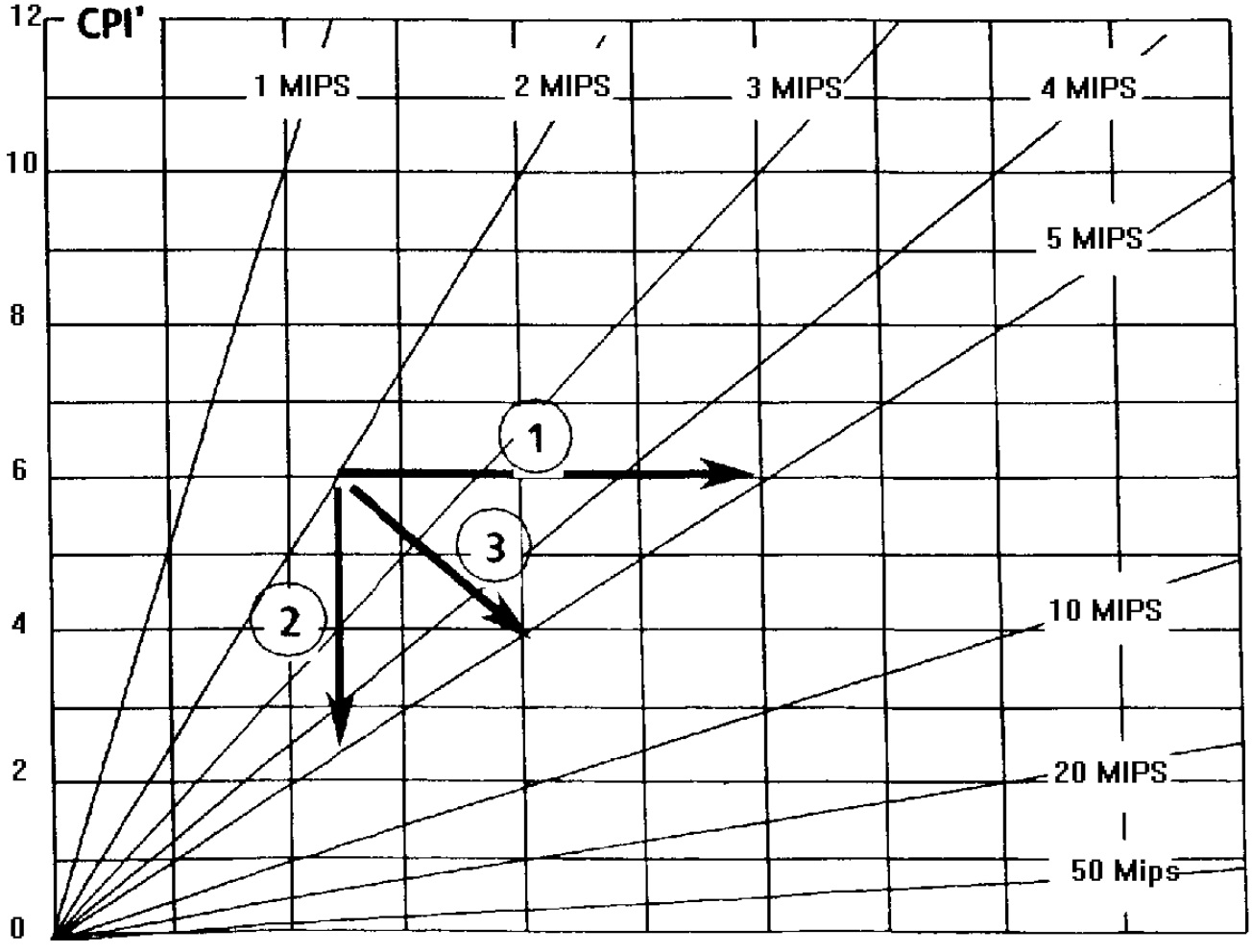
6.8.1.1 Erarbeitung: Steigerung der Performance

Erläutern Sie anhand des folgenden Schaubildes, welche prinzipiellen Möglichkeiten zur Steigerung der Performance (zum Beispiel um den Faktor 2.5) verfolgt werden können. Die horizontale Achse gibt f_C in MHz an.

In welchem Bereich müssen Fortschritte bei Pfeil 1 erzielt werden ?

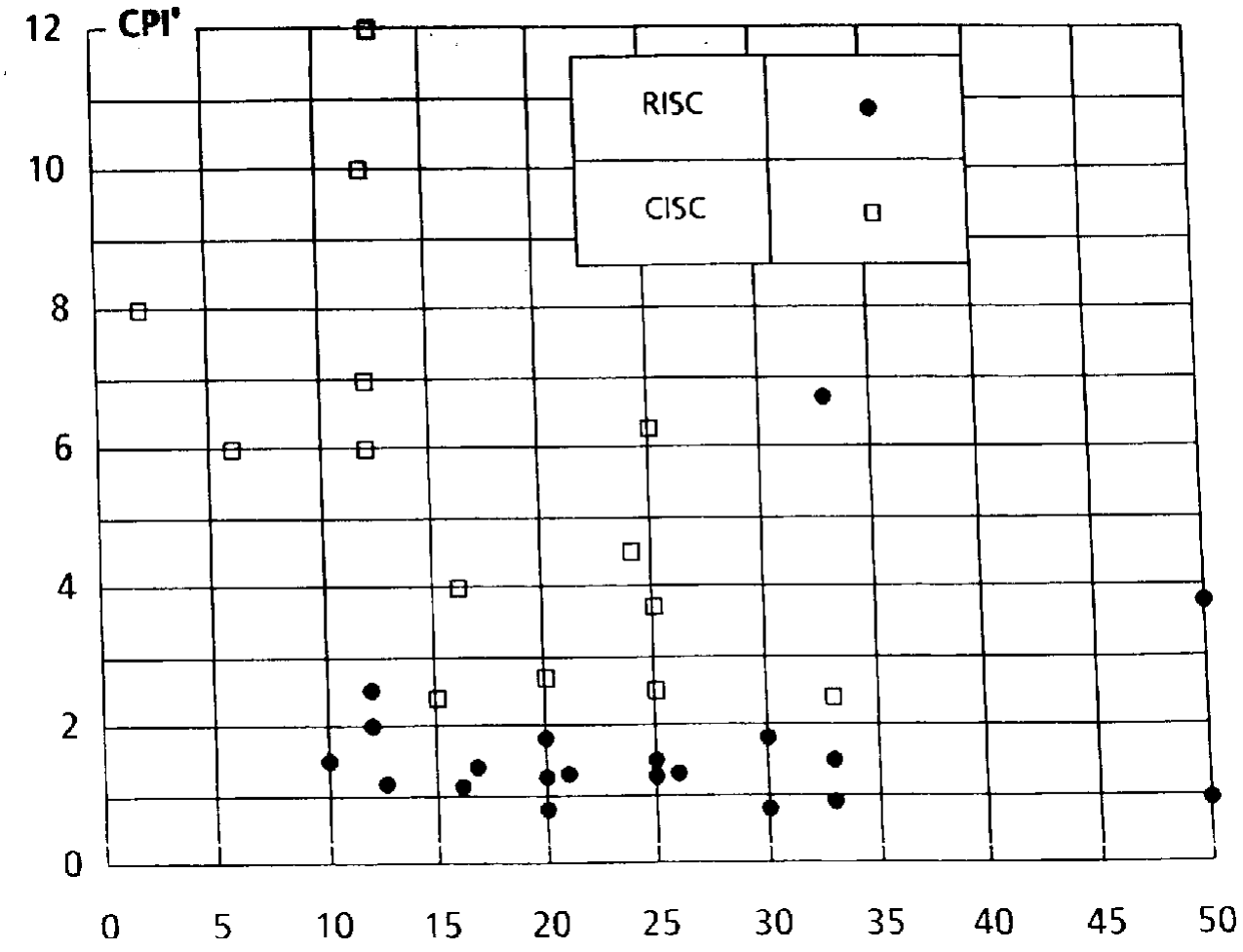
In welchem Bereich müssen Fortschritte bei Pfeil 2 erzielt werden ?

Was bedeutet Pfeil 3 ?



Verschiedene Möglichkeiten der Performance-Erhöhung um den Faktor 2,5

Trägt man die Prozessoren der obigen Tabelle in die Landkarte ein, erhält man folgende Verteilung:

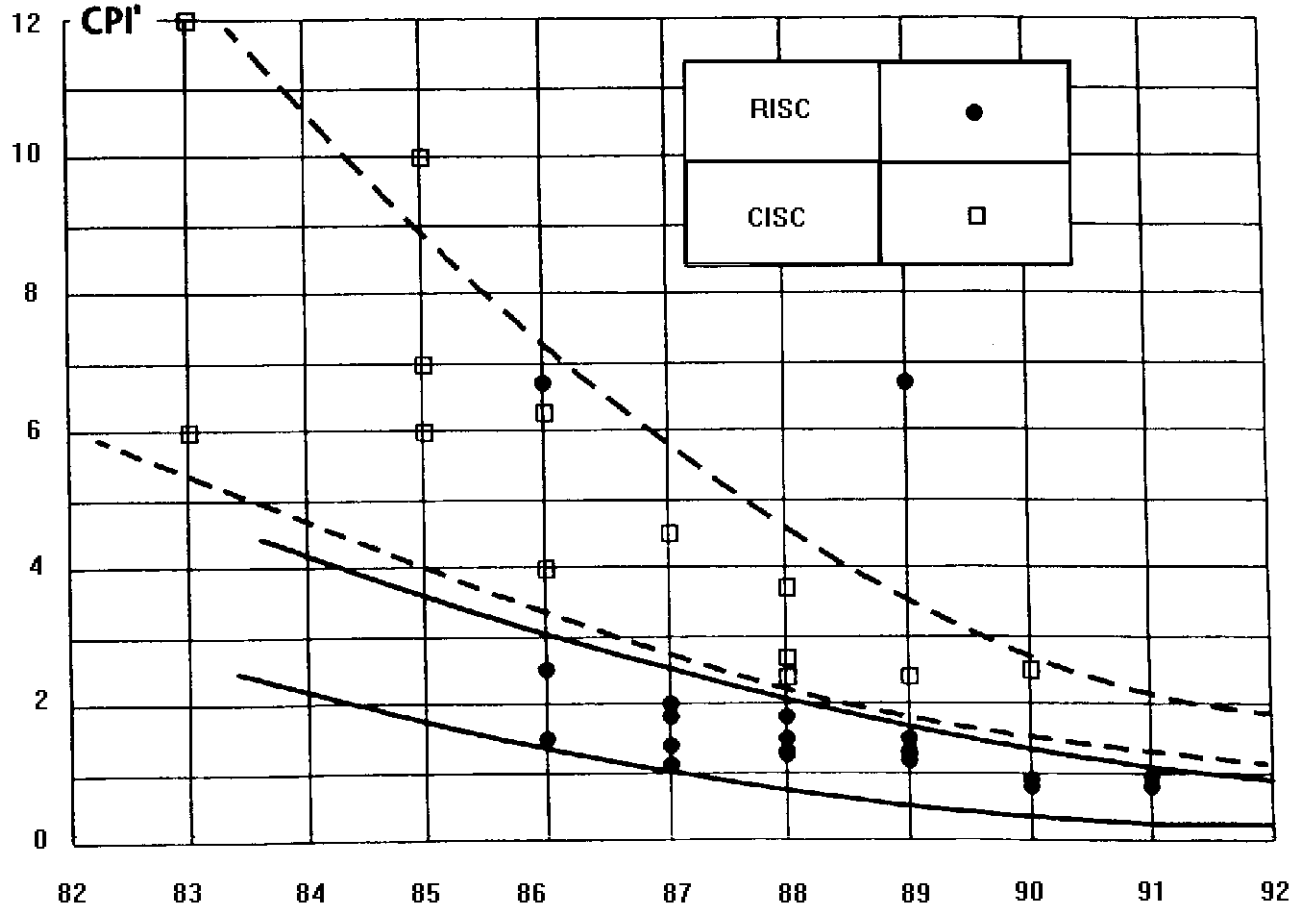


Die Zyklen-MHz-Landkarte

6.8.1.2 Erarbeitung: Entwicklung von CPI´

Tragen Sie in die Landkarte die Prozessor-Namen ein. Charakterisieren Sie anhand der Landkarte den CPI´-Einstieg und deren Entwicklung bei CISC wie bei RISC Prozessoren.

Formulieren Sie den Trend von 1982 bis 1992 auf Grundlage des folgenden Schaubildes.



Reduktion von CPI´über der Zeit

7 Speicherarchitektur

7.1 Speicherhierarchie und RISC-Prozessoren

Die Taktraten, mit der Mikroprozessoren betrieben werden, haben sich in den letzten 12 Jahren von 5 MHz auf über 66 MHz erhöht. Gleichzeitig verringerte sich dadurch die für einen Speicherzugriff zur Verfügung stehende Zeit von 240 ns auf unter 15 ns. Die Steigerung der Taktrate wurde durch **Fortschritte in der VLSI-Technologie** und durch die Übernahme neuer Architekturkonzepte, wie der RISC-Philosophie, ermöglicht. Ein RISC-Prozessor hat einen sehr hohen Befehlsdurchsatz. Beinahe in jedem Taktzyklus erfolgt ein Befehlszugriff, (da CPI in der Nähe von 1 liegt) d.h. bei einer Taktrate von 30 MHz alle 30 ns. Die Situation verschärft sich noch, weil bei jedem 3. bis 5. Befehl ein zusätzlicher Datenzugriff erfolgt. Den **Durchsatz der Speicherschnittstelle** eines Prozessors kann man durch die mittlere Transferrate (Bandbreite) beschreiben.

```
Transferrate = Breite des Busses in Bytes · Übertragungs-
frequenz
```

Bei einer Taktfrequenz von 33 MHz und 32 Bit Datenbreite ergibt sich eine notwendige Transferrate von

```
33 MHz  ·  4 Byte · 1.3  =  171,6 MByte/s.
```

Der Faktor 1.3 berücksichtigt, daß bei jedem 3. Befehls ein Datenzugriff erfolgt. Die Spitzentransferrate mit einem Datenzugriff pro Befehlszugriff liegt bei 264 MByte/s.

Die neuen Technologien kamen nicht nur den Mikroprozessoren zugute, sondern auch den SRAM (Static Random Access Memory) und DRAM (Dynamic Random Access Memory) Speicherbausteinen. Deren Zugriffszeiten konnten dennoch in der zweiten Hälfte der achtziger Jahre nicht mehr mit den Zugriffszeiten mithalten, die von den schnellsten Prozessoren gefordert wurden. Heute bieten nur noch sehr schnelle und daher teure SRAMs (Fast-SRAMs) eine ausreichende Zugriffszeit. Sind die Speicherbausteine langsam, muß der Prozessor leistungsmindernde Wartezyklen (WAIT-States) einschieben.

Das Speicherproblem wurde noch verschärft, als der Bedarf an Speicherkapazität stark zunahm. Ein Hauptspeicher mit 4 MByte ist heute die untere Grenze, wenn mit einer Benutzeroberfläche wie Windows auf dem PC effektiv gearbeitet werden soll. Typischerweise bewegt sich der Ausbau zwischen 4 und 32 MByte.

Bei einigen Rechnersystemen liegen die Hauptspeichergrößen weit über diesen Werten. Verwendete man als Speicherbausteine Fast-SRAMs, würde der Speicher zu teuer werden.

Zwecks Senkung der Speicherkosten entwickelten die Mikroprozessor-Hersteller die Speicher- oder Busschnittstellen der Prozessoren so weiter, daß auch die langsame DRAM-Bausteine eingesetzt werden können.

Zu den Neuerungen gehören:

- **Burst-Mode:** Aus dem Speicher werden nicht nur das benötigte Datum, sondern auch benachbarte Worte ausgelesen und mit der Frequenz des Busses übertragen.

- **Pipelining:** Die Adreßausgabe und das Einlesen der Befehle und Daten überlappen sich bei aufeinanderfolgenden Zugriffen.
- **Parallelbetrieb:** Befehls- und Datenbus werden getrennt geführt und erlauben damit die Verwendung zweier getrennter Speicher. (Harvard-Architektur)

Doch mit der weiteren Zunahme der Taktrate war man mehr und mehr gezwungen, eine Technik einzuführen, die aus dem Bereich der Mainframes stammt, - den **CACHE**.

Die Funktionsweise eines Caches, der einen wesentlich schnelleren Zugriff als ein Hauptspeicher erlaubt, beruht darauf, daß er die aktuell benötigten Programmteile lädt und lokal für den Prozessor bereit hält. Die Verwaltung dieser zusätzlichen Speicherebene würde jedoch den Gewinn aufheben oder sogar zu einer Verschlechterung des Speicherzugriffsverhaltens führen, wenn Programme nicht die Eigenschaft der Zugriffslokalität hätten. (Zur Erinnerung: die von Neumann Maschine, das Befehlszählerprinzip) Man unterscheidet zwei Formen von Lokalitätsverhalten:

- **Räumliche Lokalität:** Es werden kurz aufeinanderfolgend Befehle und Daten verwendet, die benachbarte Adressen haben, wie Befehlssequenzen ohne Sprung oder Daten in Arrays bzw. Strings
- **Zeitliche Lokalität:** Es werden Befehle oder Daten verwendet, auf die bereits in der jüngsten Vergangenheit zugegriffen wurde, z.B. bei Schleifendurchläufen.

Donald Knuth [1] kam bei seinen Untersuchungen 1971 unter anderem zu dem Ergebnis, daß weniger als vier Prozent eines Programmes für mehr als die Hälfte seiner Laufzeit verantwortlich sind.

7.1.1.1 Übung: Lokalitätsverhalten von Programmen:

Welche Lokalitäten sind in folgendem Programmausschnitt zu finden. Zeigen Sie die entsprechenden Bereiche an.

```
                STORE M(C)
                MOVE A,B
LOOP:           INC B
                ADD B,C
                DEC C
                BRNCH > 0 LOOP
```

Beim Cache wie auch bei anderen Speicherkomponenten gilt folgendes Prinzip:

- Hole die aktuell benötigten Befehle und Daten in den schnelleren Speicher und halte sie dort.
- Schreibe nicht mehr benötigte, aber veränderte Daten in den langsameren Speicher zurück.

Bei optimaler Abstimmung und Verwaltung der einzelnen Speicherkomponenten liegt ein Zugriffsverhalten nahe dem des jeweils schnelleren Speichers und die Kosten liegen nahe dem jeweils langsameren Speichers.

Eine ähnliche Funktion wie der Cache übernimmt daher der Hauptspeicher als nächst langsamere Speicherebene. Bereits sehr früh wußte man, daß der Speicherplatzbedarf bei Rechnersystemen ständig steigen wird. Im

[1] Knuth, D.E. „An empirical study of FORTRAN Programs“, Software Practice and Experience, Vol 1, 105-135, 1971, zitiert bei Hennesy a.a.O. S.26

Laufe der Zeit wurde deshalb der Adreßraum der Mikroprozessoren von 64 KByte auf Gigabytes und Terabytes gesteigert. Aus Kosten- und Platzgründen kann der Hauptspeicher nicht mit dem steigenden Speicherplatzbedarf mithalten. Die Softwareentwickler begannen, um die Begrenzung des Hauptspeichers zu umgehen, zunehmend auf die größeren Speicherkapazitäten der dauerhaften Speichermedien, (Floppy Disk, Festplatte) zurückzugreifen. So werden z.B. bestimmte Programmteile vom Programm Erst bei Bedarf geladen (Overlay-Technik), um mit dem vorhandenen Speicher auszukommen. Diese Lösung ist nicht transparent, d.h. sie wirkt sich auf das Programm selbst aus. Weiterhin kann der eventuell erheblich größere Adreßraum des Prozessors nicht ausgenutzt werden.

Um dieses Problem zu umgehen, wurde die virtuelle Adressierung eingeführt. Grundgedanke einer virtuellen Adressierung oder eines virtuellen Speichers ist, daß ein Programm nur mit virtuellen (logischen) Adressen arbeitet, für die kein direkter Bezug zu den Hauptspeicheradressen besteht. Das Betriebssytem sorgt zusammen mit einigen Hardwarekomponenten dafür, daß sich die benötigten Programmteile im Hauptspeicher befinden und daß auf sie zugegriffen werden kann. Für die Übersetzung virtueller Adressen in die realen Hauptspeicheradressen und das Erkennen, wann ein Nachladen eines Programmblocks nötig ist, steht eine eigene Einheit, die Memory Management Unit (MMU) zur Verfügung. Sie transformiert mit Hilfe eines eigenen Cache, des Translation Lookaside Buffers (TLB), die virtuellen Adressen in Hauptspeicheradressen.

Als Hintergrundspeicher für den Hauptspeicher muß heutzutage eine Festplatte bereitstehen.

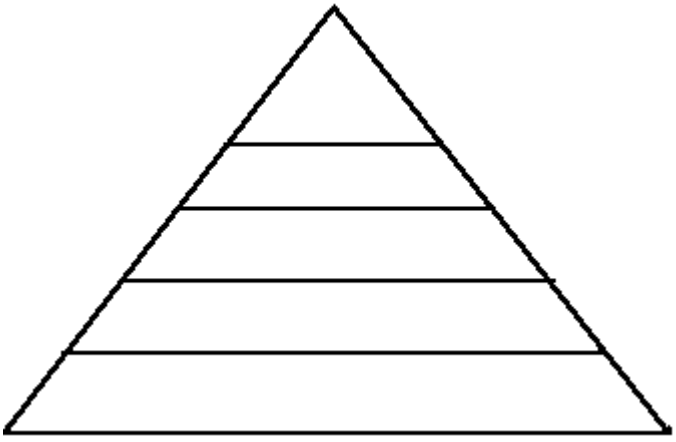
7.1.1.2 Übung: Speicherhierarchie

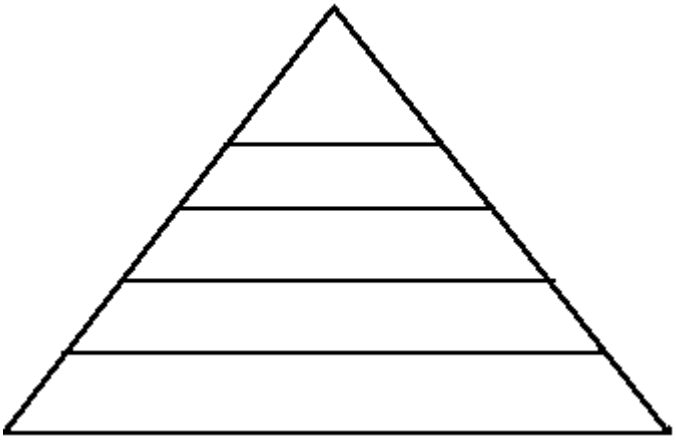

Tragen Sie in die Pyramide die Speicherbausteine ein geordnet nach

1. zunehmender Zugriffsgeschwindigkeit
2. zunehmender Größe

Die Speicherelemente sind:

- On-Board Cache
- Hauptspeicher
- On-Chip Cache
- Register
- Festplatte





7.1.2 Zusammenfassung

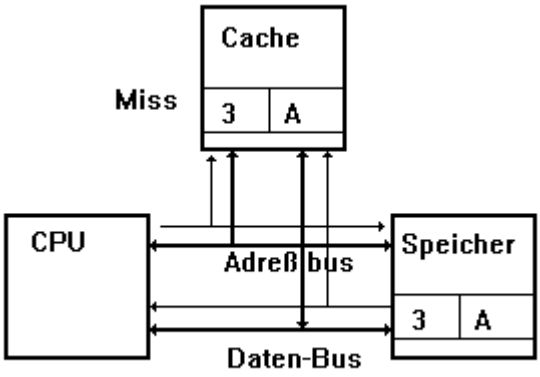
Die Lokalitätseigenschaft von Programmen erlaubt ein dynamisches Laden benötigter Programmteile. Dadurch ist es möglich, unterschiedlich große und unterschiedlich schnelle Speicherkomponenten zu verwenden. Die Einheiten, mit denen das Laden durchgeführt wird, sind an die Größe der jeweiligen speichernden Komponente angepaßt und bieten so die nötige Flexibilität. RISC-Prozessoren stellen besonders hohe Anforderungen an ihre Speicher. Um unnötige Leistungseinbußen zu vermeiden, müssen alle Komponenten der Speicherhierarchie berücksichtigt und abgestimmt werden. Da der Cache neben den Registern der CPU am nächsten ist, hat er einen entscheidenden Anteil an der Leistung, die ein Rechensystem erzielen kann. Betrachten wir deshalb Cache Speicher genauer.

7.2 Cache-Speicher

7.2.1 Funktionsweise und Aufbau

Die Aufgabe eines Cache besteht darin, den Prozessor gemäß seiner Verarbeitungsgeschwindigkeit mit Befehlen und Daten zu versorgen. Er speichert diese Befehle dynamisch, d.h. angepaßt an den augenblicklichen Bedarf.

Ein Cache wird immer dann mit neuen Informationen geladen, wenn der Prozessor auf Befehle oder Daten zugreift, die noch nicht im Cache vorhanden sind. Solch ein Ereignis nennt man Cache Miss, oder genauer, falls auf den Cache lesend zugegriffen wird, ein Read Miss. Das laden eines Cache kostet Zeit, da u.a. auf den langsameren Hauptspeicher zugegriffen werden muß. Die CPU muß während des Ladens Wartezyklen einlegen.



1 Die CPU legt Adresse 3 auf den Adreßbus

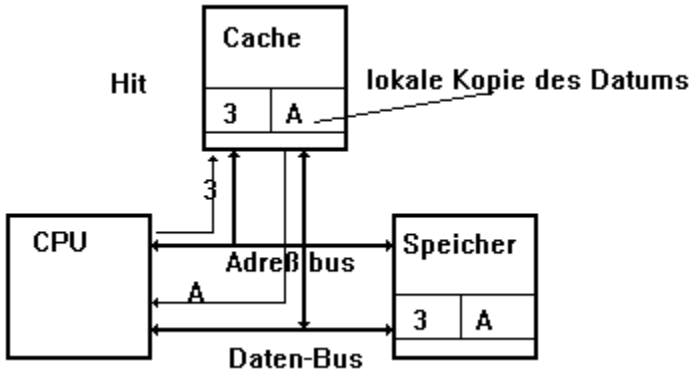
2 Die Cache Logik stellt fest, daß das Datum der Adresse 3 nicht abgespeichert ist, d.h. ein Cache Miss liegt vor.

3 Vom Hauptspeicher wird das Datum A der Adresse 3 geholt. Die CPU liest das Datum A ein

4 Die Cache Logik legt das Datum A zusammen mit dem Tag 3 in einem Cache-Eintrag ab.

Neben der eigentlichen Nutzinformation wird noch die angesprochene Adresse im Cache abgelegt. Dies dient als Markierung (Tag) und erlaubt es so, im Cache abgespeicherte Daten eindeutig wiederzuerkennen .

Greift die CPU in der weiteren Bearbeitung des Programms erneut auf Befehle oder Daten zu (zeitliche Lokalität), die bereits im Cache gespeichert sind (Hit Fall), dann können diese Daten ohne Verzögerung der CPU zur Verfügung gestellt werden.



1 Die CPU legt wieder die Adresse auf den Adreßbus

2 Die Cache Logik stellt fest, daß Adresse 3 vorhanden ist (Cache-Hit)

3 Der Cache stellt Datum A auf dem Datenbus für die CPU bereit.

Die Anzahl der Treffer oder Hits bezogen auf die Gesamtanzahl der Zugriffe ergibt die sogenannte Hit Rate. Ihr Pendant ist die Miss Rate, die in den folgenden Ausführungen verwendet werden soll:

```
Hit Rate  = Anzahl der Hits / (Anzahl der Hits + Anzahl
der Misses)

Miss Rate = 1 - Hit Rate
```

Die Miss Rate ist ein Maß dafür, wie gut ein Cache für ein bestimmtes Anwenderprogramm geeignet ist. Sehr gute Miss Rats liegen unter 5%. Die Miss Rate ist von einer Reihe Faktoren abhängig:

- Cache Größe
- Cache Organisation
- Anzahl Bytes beim Laden
- Ersetzungsstrategie

7.2.1.1 Erarbeitung

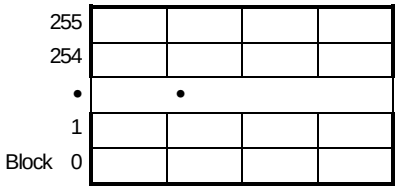

Warum ist die Miss Rate bei Befehlen geringer als bei Daten ?

Versuchen Sie zu erschließen, warum die Miss Rate von den oben genannten Faktoren 1, 3 und 4 abhängen könnte (zu Cache Organisation können Sie wahrscheinlich noch nichts sagen).

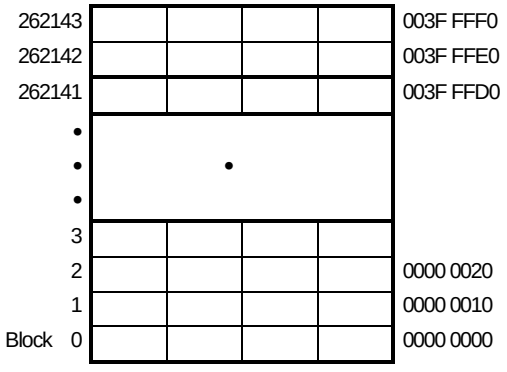
7.2.2 Der Block als Transporteinheit zwischen Hauptspeicher und Cache

Die zwei unterschiedlich großen Datenfelder von Cache und Hauptspeicher lassen sich in gleich große Einheiten unterteilen. Eine solche Einheit nennt man Block. Da Daten in diesen Block-Einheiten vom Hauptspeicher in den Cache geholt werden, kennzeichnet ein Block auch die kleinste Datenmenge, die in einem Cache vorhanden ist.

Cache 4 KByte:



Hauptspeicher 4 MByte:



7.2.3 Cache Parameter

Beim Entwurf eines Cache Systems legt man fest, wieviel Befehle oder Daten (Worte, Bytes) maximal abgespeichert werden (**cache size**), in welchen Paketen diese Daten in den Cache geholt werden (**block size**), und im Cache abgelegt werden (**line size**), wie für diese Daten Speicherplätze im Cache ausgewählt werden (**placement/replacement strategy**), wie Schreibzugriffe ablaufen (**write strategy**) und wie sichergestellt wird, daß nur gültige Daten im System verarbeitet werden (**coherency mechanism**). Es können Befehle und Daten gemeinsam oder getrennt gespeichert werden (**instruction/data cache**). Ziel eines guten Cache Design ist es, den Leistungsgewinn bei gegebenem Aufwand durch Variation einzelner Parameter zu optimieren. Dabei muß beachtet werden, daß die Ergebnisse stark programmabhängig sind. In diesem Abschnitt werden die Parameter und ihre Auswirkungen näher erläutert.

7.2.3.1 Cache Größe

Die Wirkung eines Cache nimmt mit seiner Größe zu, da immer größere Teile eines Programms im Cache Platz finden. Die Grundbausteine eines Cache - schnelle SRAMs - sind jedoch sehr teuer und setzen damit dem Ausbau Grenzen.

7.2.3.2 Übung: Cache Größe

Wie groß sind die Cache Speicher von Ihnen bekannten Prozessoren ?

7.2.3.3 Blockgröße

Größere Blöcke erlauben eine effiziente Ausnutzung der räumlichen Lokalität von Programmen und damit ein Sinken der Miss Rate.

Grenzen für die Erhöhung der Blockgröße:

- Sehr große Blöcke ´verschmutzen´ jedoch den Cache, da sie zunehmend Befehle oder Daten enthalten können, die von der CPU aktuelle nicht verwendet werden.
- Mit zunehmender Größe wächst der Zeitbedarf für einen Miss.

Beide Faktoren bestimmen diejenige Blockgröße, bei der die mittlere Speicherverzögerung minimal wird. Da Befehle eine größere Lokalität als Daten aufweisen, sollten sie auch in größeren Blöcken nachgeladen werden. Für Befehle sind Blöcke mit 16 - 32 Worte typisch, für Daten solche mit 8 - 16 Worten.

7.2.3.4 Cache Organisation

Alle Blöcke des Hauptspeichers oder besser des physikalischen Adreßraums, lassen sich den Blöcken eines Caches so zuordnen, daß

- jeweils nur ein bestimmter Cache-Block zur Verfügung steht (**einfach satzassoziativer Cache)** oder direct mapping Cache.
  - **Vorteil:** Benötigt nur geringen Hardwareaufwand (insbesondere Komparatoren), weil jeder Hauptspeicherblock in genau einem festgelegten Cache-Block abgelegt wird.
  - **Nachteil:** Besonders bei kleineren Caches ist die Gefahr groß, daß ein Programm häufig auf Adressen zugreift, die den gleichen Index haben und sich dadurch gegenseitig aus dem Cache verdrängen (*trashing*).
- jeder Block in jedem Cache-Block abgespeichert werden kann (**vollassoziativer Cache**),
  - **Vorteil:** Bietet die beste Miss-Rate, weil jeder Block des Hauptspeichers an jedem beliebigen Block des Caches abgelegt werden kann. Wegen dieser hohen Flexibilität eignet er sich besonders bei kleinen On-Chip-Caches
  - **Nachteil:** hoher Hardware-Aufwand
- jeder Block in ausgewählten Cache-Blöcken abgespeichert werden kann (**n-fach satzassoziativer Cache**)

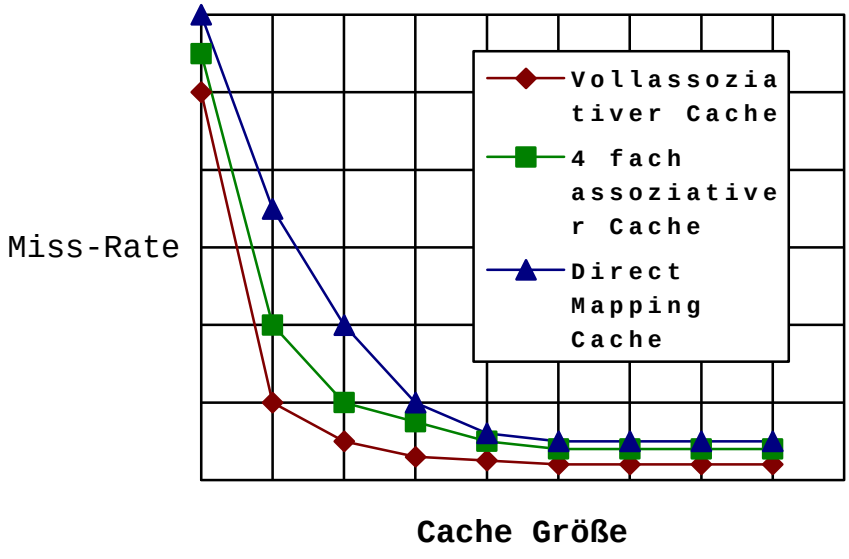
- **Vorteil:** geringere trashing Gefahr als beim einfach satzassoziativem Cache, eignet sich auf Grund der noch relativ geringen Hardware-Kosten auch für größere Cache-Speicher.

7.2.3.5 Plazierungs-/Ersetzungsalgorihmus

Während die Auswahl eines Speicherplatzes beim Cache mit direct-mapping allein über die Blockadresse gesteuert wird, ist bei den anderen ein zusätzlicher Mechanismus für die Plazierung eines neuen Datenblocks innerhalb eines Satzes notwendig. Ein **Plazierungsalgorithmus** hat die Aufgabe, freie Einträge nach und nach aufzufüllen. Während des größten Teils der Laufzeit eines Programms sind in der Regel alle Einträge belegt. Die Auswahl eines zu entfernenden Eintrages ist die Aufgabe des **Ersetzungsalgorithmus**. Seine optimale Strategie würde folgendermaßen aussehen: Ersetze denjenigen Datenblock, der in Zukunft nicht mehr angesprochen wird. Da ein solcher Blick in die Zukunft nicht möglich ist, versuchen reale Verfahren Rückschlüsse aus der Vergangenheit zu ziehen. Es gibt folgende Verfahren mit unterschiedlichen Auswirkungen auf die Miss-Rate:

- LRU (Last recently used) — der am längsten nicht mehr angesprochene Eintrag wird ersetzt.
- LFU (least frequently used) — der am seltensten angesprochene Eintrag wird ersetzt.
- FIFO (first in, first out) — der älteste Eintrag wird ersetzt.
- Random — ein per Zufall ausgewählter Eintrag wird ersetzt.

Aus Geschwindigkeitsgründen ist das gewählte Verfahren in der Hardware zu implementieren. Bei zwei- und vierfach satzassoziativen Caches findet man sehr häufig das LRU Verfahren, da es meist das beste Verfahren ist und bei diesem Assoziativitätsgrad noch eine geringe Komplexität aufweist. Bei vollassoziativen Caches wird oft das einfach zu implementierende und kaum schlechter arbeitende Random Verfahren verwendet.



Vergleich zwischen direct-mapping, 4-fach satzassoziativ und vollassoziativ

8 Pipelining

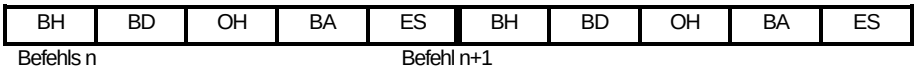
Die Befehlspipeline bildet ein wesentliches Merkmal von RISC-Prozessoren. Sie erlaubt eine parallele Bearbeitung mehrerer Befehle im Prozessor, wodurch die Performance des Prozessors deutlich vergrößert werden kann.

In den folgenden Abschnitten sollen die wichtigsten Prinzipien einer Pipeline erläutert und Architekturmerkmale von RISC-Prozessoren erklärt werden, die sich durch die Struktur der Pipeline begründen lassen. Die bei einer parallelen Verarbeitung von Befehlen auftretenden Daten- und Steuerkonflikte werden analysiert und Lösungsverfahren diskutiert.

8.1 Das Pipeline Prinzip

Im Abschnitt "Ein-Zyklus-Operationen" wurde bereits die Notwendigkeit und das Prinzip des Befehls-Pipelinings besprochen. Die Einführung einer Befehls-Pipeline-Verarbeitung verfolgt das Ziel, durch eine überlappte Parallelverarbeitung die Performance eines Prozessors deutlich zu steigern. Im Gegensatz zu einer rein sequentiellen Abarbeitung der Befehle wird bei einer Pipeline der nächste Befehl zur Verarbeitung eingeschleust, sobald es die vorausgehenden zulassen.

Sequentielle Befehlsausführung :



Pipeline-Verarbeitung:

Befehl n	BH	BD	OH	BA	ES					
Befehl n+1		BH	BD	OH	BA	ES				
Befehl n+2			BH	BD	OH	BA	ES			
Befehl n+3				BH	BD	OH	BA	ES		
Befehl n+4					BH	BD	OH	BA	ES	
Befehl n+5						BH	BD	OH	BA	ES

Mit dieser Architekturmaßnahme wird demnach nicht die Bearbeitungszeit (Latenzzeit) des einzelnen Befehls verkürzt. Die Leistungssteigerung von RISC-Prozessoren resultiert aus einem erhöhten Durchsatz von abgearbeiteten Befehlen pro Zeiteinheit. Dieser Durchsatz wird durch die Zeit bestimmt, die zwischen dem Einschleusen der einzelnen Befehle verstreicht und nicht durch die Latenzzeit der einzelnen Befehle.

Eine Voraussetzung dafür, daß ein Prozessor mit einer Pipeline betrieben werden kann, ist die Zerlegbarkeit eines Befehls in verschiedene Segmente (Pipeline-Stufen), die auf der Hardware-Ebene voneinander entkoppelt sind, damit sich die Befehle nicht gegenseitig beeinflussen. Die damit notwendigen Hardware-Erweiterungen führen weg von der bekannten Von Neumann Architektur eines Minimalsystems !

Die Zeit, nach der die Bearbeitung eines Befehls um eine Stufe vorrückt, wird als interne Taktzeit definiert (nicht immer identisch mit der externen Taktfrequenz!).

Die Taktzeit einer Pipeline richtet sich nach der längsten Verarbeitungsdauer der Pipeline-Stufen.

8.1.1.1 Übung: Latenzzeit und Pipelining

Warum wird die Latenzzeit beim Pipelining eher größer als kleiner ?

8.2 Prozessorarchitektur und Befehls-Pipeline

Die Grobstruktur der Prozessorarchitektur und die Befehls-Pipeline beeinflussen sich gegenseitig sehr stark. Mit Hilfe einer formalen Beschreibung soll diese Wechselwirkung verdeutlicht werden.

8.2.1 Hardware-Belegungsschema

Zur Beschreibung einer Pipeline wird das Hardware-Belegungsschema aufgestellt, eine Matrix, die für jeden Befehl erstellt wird. Eingetragen wird, welche Einheiten der Hardware in den einzelnen Pipeline-Stufen zur Verarbeitung des Befehls herangezogen werden

Pipeline-Segment	**Pipeline-Stufe----------->**						**Befehl**
	1	2	3	4	5	6	MOVE
Speicher	•						Register
Decoder		•					--->
Register			•	•			Register
ALU							

	1	2	3	4	5	6	ADD
Speicher	•			•		•	Memory+
Decoder		•					Register
Register			•				--->
ALU					•		Memory

Hardware Belegungsschema einer multifunktionalen, nicht linearen Pipeline

An Hand dieses Schemas läßt sich eine Klassifizierung verschiedener Pipelines durchführen. Gibt es für verschiedene Befehle unterschiedliche Belegungsschemata, so spricht man von einer multifunktionalen Pipeline. Können alle Befehle durch genau ein Hardware-Belegungsschema repräsentiert werden, so wird sie unifunktional genannt.

Lassen sich alle Befehle in einem Schema vereinbaren, wobei manche Einheiten nicht von allen Befehlen benötigt werden, so ist diese Pipeline im strengen Sinne nicht unifunktional. Da sie aber im Prinzip wie eine solche behandelt werden kann, klassifiziert man sie als quasi-unifunktional.

Unter der Linearität einer Pipeline versteht man, daß eine Hardware-Einheit für einen Befehl nur in einer einzigen Pipeline-Stufe verwendet wird. Im obigen Bild ist an der Mehrfachbelegung der Speicherschnittstelle ersichtlich, daß eine nicht-lineare Pipeline vorliegt.

Besonderes Interesse gilt bei einer Pipeline der Frage, wie sich eine Abfolge von Befehlen mit möglichst großem Durchsatz realisieren läßt. Mit Hilfe des Hardware-Belegungsschemas läßt sich das graphisch darstellen, indem mehrere Befehle in ihrer Abfolge eingetragen werden (s. Bild unten). Tritt dabei ein Konflikt auf, weil zwei Befehle gleichzeitig dieselbe Verarbeitungseinheit beanspruchen, so muß der später eingeschleuste Befehl so lange verzögert werden, bis die Abarbeitung ohne Konflikte durchgeführt werden kann. Jeder eingeschobene Wartezyklus verschlechtert aber das Verhältnis CPI und damit den Durchsatz der Pipeline. Zusätzlich ist noch zu überprüfen, ob ein später begonnener Befehl Daten verändert (Registerinhalte und Speicher), auf die ein früher begonnener Befehl noch zu einem späteren Zeitpunkt zugreift.

Bei einer multifunktionalen Pipeline kann dabei jede mögliche Kombination von Befehlen auftreten und damit steigt das Potential an Konflikten an, die zu Verzögerungen führen. Andererseits ist der einfachste Fall einer Pipeline eine quasi-unifunktionale, lineare Pipeline, bei der prinzipiell keine strukturellen Konflikte auftreten können. Deshalb wurde die Architektur der RISC-Prozessoren konsequent auf eine lineare und quasi-unifunktionale Pipeline ausgerichtet, um einen weitgehend konfliktfreien Ablauf zu gewährleisten. Sie bedingt auch das strikte Festhalten an der Load-Store-Architektur, da die Erweiterung der Speicherzugriffsmöglichkeiten oder Adreßberechnungsmodi das System der linearen, quasi-unifunktionalen Pipeline durchbricht. Dieselbe Motivation führt zu einer einheitlichen Länge der Befehlsformate. Formatgrößen, die länger als ein Wort sind oder Befehle, die nicht auf Wortgrenzen ausgerichtet sind, verlangen zusätzliche Zugriffe beim Einlesen. Deshalb besitzen alle RISC-Prozessoren Befehlsformate, die genau ein Wort lang sind (in der Regel 32 Bit, Ausnahme Alpha 2064) und mit der Breite der Speicherschnittstelle übereinstimmen.

8.2.2 Beispiel-Pipeline

Für die folgenden Untersuchungen wird eine Beispiel-Pipeline mit **5 Stufen** verwendet, bei der alle Probleme, die bei RISC-Prozessoren auftauchen, erörtert werden können. Die 5 Stufen sind wie folgt definiert.:

- IF Instruction Fetch Befehl holen
- DR Decode and Read Dekodieren und Register lesen
- EX Execute Ausführen des Befehls
- MA Memory Access Speicherzugriff
- WB Write Back Register schreiben

Für die einzelnen Pipeline-Stufen werden folgende Hardware-Einheiten benötigt:

- IF Speicher
- DR Decoder und Register-File
- EX ALU
- MA Speicher
- WB Register-File

Bei der Zuordnung der einzelnen Hardware-Einheiten auf die Pipeline-Stufen gilt folgendes:

- Die Zugriffe auf die externen Speicher besitzen die längsten Ausführungszeiten und bilden damit die Taktzeit des Prozessors. Sie werden den Stufen IF und MA zugeordnet.
- Da RISC-Prozessoren meist ein einfaches Befehlsformat besitzen, kann auf die Dekodierung der Befehle mit dem Auslesen der Operanden aus dem Register-File in der Stufe DR zusammengefaßt werden.
- Die ALU, plaziert in der Stufe EX, dient nicht nur der Berechnung der arithmetischen und logischen Funktionen, sondern sie wird auch für die Berechnung der effektiven Adressen für Speicherzugriffe herangezogen
- Die Resultate werden zum Schluß (WB) in das Register-File zurückgeschrieben.

Aus dieser Architektur läßt sich das folgende Hardware-Belegungsschema herleiten. Um eine quasi-unifunktionale Pipeline zu erhalten, wird bei allen Befehlen, die nicht auf den Speicher zugreifen, die für Speicherzugriffe reservierte Pipelinestufe MA überbrückt. Für alle diese Befehle wird die Ausführungszeit länger als es die Funktionalität erfordert.

Pipeline-Segment	**Pipeline-Stufe----------->**				
	1-IF	2-DR	3-EX	4-MA	5-WB
Speicher	•			•	
Decoder		•			
Register		•			•
ALU			•		

Hardware-Belegungsschema der Beispiel-Pipeline

Für die folgenden Überlegungen werden die dynamischen Häufigkeiten einzelner Befehle benötigt, um die Auswirkungen eventueller Konflikte in der Pipeline quantitativ zu beurteilen:

Befehl	dynamische Häufigkeit in %
bedingte Sprünge	11
unbedingte Sprünge	4
Load	18
Store	8
Arithmetik	39
Rest	20

Häufigkeit von Befehlsklassen

8.3 Strukturelle Konflikte

Bei der Beispiel-Pipeline werden sowohl das Register-File als auch die Speicherschittstelle in zwei verschiedenen Pipeline-Stufen in Anspruch genommen. Deshalb stellt sich die Frage nach der **Linearität** der Pipeline.

	1-IF	2-DR	3-EX	4-MA	5-WB
Befehlsspeicher	•			•	
Decoder		•			
Register		.			.
ALU			•		
Datenspeicher	•			•	

Hardware-Belegungsschema der verbesserten Beispiel-Pipeline

Eine Verfeinerung des Hardware-Belegungsschemas zeigt eine Lösung dieses Konflikts. Gelingt es, das Register-File so schnell zu machen, daß für das Auslesen der Operanden nur die halbe Taktzeit benötigt wird, so steht für das Schreiben die andere Hälfte zur Verfügung. Wird das Belegungsschema verfeinert, indem man die einzelnen Phasen statt des gesamten Taktes betrachtet, so läßt sich die Konfliktfreiheit auch graphisch darstellen.

	1	2	3	4	5	6	7
Befehlsspeicher	1	2	3	4	5		
Decoder		1	2	3	4	5	
Register		1	2	3	1 4	2 5	3
ALU			1	2	3	4	5
Datenspeicher				1		3	4

Konfliktfreier Ablauf in einer unifunktionalen linearen Pipeline

Ähnliches gilt auch für die Speicherschnittstelle. Eine räumliche Trennung zwischen Befehls- und Datenstrom, auch **Harvard-Architektur** genannt, erlaubt den gleichzeitigen und konfliktfreien Zugriff auf Befehle und Daten. Wird dieser Konflikt an der Speicherschnittstelle nicht derart aufgelöst, so muß bei einem Load-Store Befehl i der Befehle i+3 um einen Takt durch Einschieben eines Wartezyklus verzögert werden:

LOAD n	IF	DR	EX	MA		WB					
STORE n+1		IF	DR	EX		MA		WB			
ADD n+2			IF	DR		EX		MA	WB		
ADD n+3					IF	DR		EX	MA	WB	
ADD n+4							IF	DR	EX	MA	WB

Wartezyklus durch Load und Store:

Struktureller Konflikt an der Speicherschnittstelle

Dieser Strukturkonflikt würde also die Performance der Pipeline maßgeblich verschlechtern (bis zu 26 %). Deshalb wird die Beispiel-Pipeline mit einer Harvard--Architektur ausgestattet. Das Lesen und Schreiben des Registerfiles wird in zwei nicht überlappende Phasen eines Prozessortaktes gelegt.

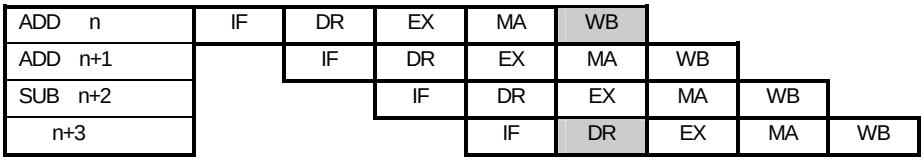
8.4 Datenkonflikte

Neben den strukturellen Konflikten innerhalb einer Pipeline tritt noch eine andere Klasse von Konflikten auf. die Verletzung der **Datenkonsistenz**. Zum Beispiel bei der Abfolge der folgenden drei Befehle in der Pipeline (Addition dreier Werte aus R1, R2 und R3 zu einem Ergebnis, das in R4 abgespeichert wird, und eine davon unabhängige Subtraktion):

```
ADD     R1 R2 R4        :R1 + R2 → R4
ADD     R4 R3 R4        :R4 + R3 → R4
```

```
SUB     R5 R0 R6        :R5 - R0 → R6
```

stellt man fest, daß der Operand R4 in dem zweiten Befehl noch nicht im Registerfile enthalten ist, da der erste Befehl n das Ergebnis erst zu einem späteren Zeitpunkt in das Registerfile einträgt, und es erst im Befehl i+3 zur Verfügung steht.



Datenkonsistenzkonflikt durch späte Verfügbarkeit des Ergebnisses

8.4.1 Softwarelösung

Für diesen Konflikt läßt sich zunächst eine Lösung finden, die dieses Problem auf die Software verlagert und keinen Eingriff in die Hardware erfordert. Dem Compiler wird die Aufgabe übertragen, die Befehle mit Datenabhängigkeiten so weit voneinander zu trennen, bis der Operand aus dem Register gelesen werden kann. Der Compiler fügt dazu zwischen diesen Befehlen leere Operationen (nop) ein, die es anschließend durch Umsortieren des Codes durch sinnvolle Befehle ersetzt (Instruction Scheduling). Gelingt dies nicht, so bleiben leere Operationen in dem Programmstück enthalten, die den Durchsatz an effektiven Befehlen herabsetzen und das Programm verlängern. Siehe dazu die Ausführungen in "Befehlspipelining und optimierende Compiler".

8.4.1.1 Übung: Code-Sortierung

Stellen Sie die Programmsequenz um und setzen Sie NOP-Befehle ein, damit kein Datenkonflikt auf unserer fünfstufigen Pipeline entsteht:

```
ADD     R1 R2 R4        :R1 + R2 → R4
ADD     R4 R3 R4        :R4 + R3 → R4
SUB     R5 R0 R6        :R5 - R0 → R6
```

8.4.2 Scoreboarding

Ein alternatives Konzept sieht eine zusätzliche Hardware vor, die Datenkonsistenzkonflikte erkennt und die entsprechenden Verzögerungen veranlaßt. Dazu wird ein **Scoreboard** eingeführt, mit dessen Hilfe für jedes Register zur Laufzeit Buch geführt wird, ob sich ein Befehl in der Pipeline befindet, der den entsprechenden Registerinhalt verändert. Wird nun ein nachfolgender Befehl in die Pipeline eingeschleust, so läßt sich am Scoreboard ablesen, ob dieser Befehl auf einen noch nicht gültigen Registerinhalt zugreifen will. Zeigt das Scoreboard einen Konflikt an, so wird dieser Befehl solange verzögert, bis die Markierung im Scoreboard gelöscht ist, die einen ungültigen Registerinhalt anzeigt.

Das Scoreboarding sorgt also für die semantisch richtige Abarbeitung einer Befehlsfolge. Häufig auftretende Datenabhängigkeiten können den Durchsatz allerdings drastisch vermindern. Ein Umsortieren des Codes, wie sie die reine Software-Lösung vornimmt, kann diese Leistungseinbußen reduzieren. Auf das Einfügen der NOP

Befehle kann dabei verzichtet werden, da die Verzögerungen dynamisch durch das Scoreboard eingefügt werden.

8.4.2.1 Übung: Vorteil des Scoreboardings

Welchen Vorteil bietet das Scoreboarding gegenüber der Softwarelösung ?

8.4.3 Forwarding

Das Ergebnis von Register-Register Operationen liegt in nach der dritten Pipeline-Stufe (EX) bereits fest, wird aber erst in der fünften (WB) erst den folgenden Befehlen zur Verfügung gestellt. Es liegt deshalb nahe, diesen Wert anstelle des ungültigen Registerinhalts der Verarbeitungseinheit für die Befehle n+1 und n+2 zuzuführen.

Für diese Lösung der Datenkonsistenzkonflikte müssen natürlich weitere Datenwege bereitgestellt werden. Eine zusätzliche Dekodierlogik, die die Registeradressen der einzelnen Befehle auf Konflikte vergleicht, muß die Auswahl des richtigen Datums steuern

ADD R1 R2 R4	IF	DR	EX	MA	WB			
ADD R4 R3 R5		IF	DR	EX	MA	WB		
SUB R5 R4 R7			IF	DR	EX	MA	WB	
n+3				IF	DR	EX	MA	WB

Mit dieser Erweiterung der Architektur - Forwarding genannt - wird die Pipeline aus der Sicht der Software unsichtbar und braucht vom Compiler nicht durch Umsortieren des Codes berücksichtigt zu werden.

8.5 Steuerkonflikte

Auf die Steuerkonflikte wurde schon im Kapitel Befehls-Pipelining und optimierende Compiler eingegangen. Sie werden durch unbedingte Sprünge, bedingte Verzweigungen und Rücksprüngen aus Unterprogrammen ausgelöst. Das Umsetzen des Befehlszählers auf die Zieladresse während der Ausführungsphase des Sprungs führt dazu, daß alle sequentiell nachfolgenden Befehle, die sich bereits in der Pipeline befinden, ungültig werden.

Neben dem bereits ausgeführten Verfahren zur Minderung von Steuerungskonflikten (Füllen der Branch-Delay-Slots mit datenunabhängigen Befehlen) , gibt es eine weitverbreitete Technik, Sprünge generell als nicht auszuführend vorauszusagen und einfach mit der Bearbeitung des sequentiell nachfolgenden Befehls fortzufahren. Trifft die Voraussage zu, dann ergibt sich keine Effizienzminderung. Andernfalls steht das bereits während der Sprungdekodierung oder kurz danach fest. Der Nachfolgebefehl wird sofort für ungültig erklärt. Durch das Neuaufsetzen der Befehlholephase ab der Zieladresse entsteht nur in diesem Falls eine Auslastunglücke. [1]

8.5.1 Branch Prediction

Wenn auch Befehle in Delay-Slots eingefügt werden sollen, die im ursprünglichen Source-Code erst **nach** dem Sprungbefehl auftreten, dann müssen leistungsfähige **Branch-Prediction**-Techniken verwendet werden, die sicherstellen, daß ein eingefügter Befehl tatsächlich zur Performance-Optimierung beiträgt. Es werden nur solche nachfolgenden Befehle in die Delay-Slots eingefügt, die keine inhaltlichen Auswirkungen auf die Logik der durchzuführenden Berechnungen haben, falls mit ihrer Verarbeitung begonnen wurde, obwohl die Voraussage falsch war. Neuere superskalare Architekturen, wie der DEC Alpha Chip, die gleichzeitig mehrere Befehle starten und ausführen können, verzichten auf Branch-delay slots und nehmen somit gewisse Auslastungslücken in Kauf. Bei einer Alpha müßten bei Verzweigungen maximal acht Befehle für das Einfügen in die Branch-Delay-Slots gefunden werden. Dies ist nicht mit vertretbarem Softwareaufwand zu erreichen. Bei der Alpha-Architektur können Sprungbefehle mit Markierungen versehen werden, die darüber informieren, ob die Sprünge voraussichtlich durchgeführt werden oder nicht. Optimierende Compiler sorgen dafür, daß sich Sprungziel-Befehle und Daten fast immer in den Caches befinden.

8.5.2 Branch-Prediction-Buffer

Zur Unterstützung korrekter Voraussagen des Sprungverhaltens können auch dynamische Techniken verwendet werden. In einem Branch-Prediction Buffer (ein kleiner Speicherbereich im Prozessor), steht für jeden Sprungbefehl, der beim letzten mal ausgeführt wurde, das Bit 1, sonst das Bit 0. Wenn der Sprungbefehl erneut geholt wird, werden die nachfolgenden Befehle je nach Bit-Wert von der Zieladresse oder aus dem sequentiellen Code geholt. Bei falscher Voraussage wird der Bitwert invertiert. Der bereits geholte Befehl wird zurückgenommen und der korrekte Befehl wird geholt. Das Schema ist erfolgreich, weil vor allem Rückwärtssprünge an den Anfang von Schleifen häufig ausgeführt werden und die Voraussage ab dem ersten Rücksprung bis zum Verlassen der Schleife korrekt bleibt.

8.5.2.1 Branch-Prediction-Stack

Ein weiterer Mechanismus zur Verringerung verzweigungsbedingter Verzögerungen, ist der Branch-Prediction-Stack. Er funktioniert nach dem gleichen Prinzip wie andere Stapelspeicher. Seine spezielle Aufgabe ist es, die

[1] Vgl. Märtin, a.a.O. S. 186 f

Zieladresse von unbedingten Verzweigungen und Sprunganweisungen vorherzusagen. Wenn eine Prozedur aufgerufen wird, was in der Regel als Sprung implementiert ist, wird der Prozessor über ein Feld in der Instruktion aufgefordert, die Rücksprungadresse im Branch Prediction-Stack abzulegen. Ist die Prozedur abgeschlossen, benutzt der Prozessor zum Rücksprung in den Befehlsfluß die auf dem Stack abgelegte Adresse.

8.5.3 Sprungvorhersage

Die Prozessoren der x86-Familie sind bis hin zum 486er eh ohne spezielle Mechanismen für Sprünge ausgestattet. Sie prefetchen immer den sequentiellen Nachfolger und nehmen somit bei Verzweigungen längere Wartezeiten in Kauf. Demgegenüber treiben Pentium, M1 und Co. viel Aufwand, um Sprünge bereits in der Prefetch-Phase zu erkennen, so daß sie auch ohne Delayed Branches keine Pipeline Stalls erleiden.

Die überwiegende Mehrheit auftretender Verzweigungen ist aber mit einer erst in der Exekutionsphase bekannten Bedingung verknüpft. Typische x86-Programme bestehen etwa zu 14 Prozent aus bedingten Sprüngen und zu 7 Prozent aus unbedingten Verzweigungen. Um hier performancevernichtende Pipeline-Stalls so selten wie irgend möglich zu machen, mußte viel Entwicklergehirnschmalz fließen.

Ein denkbarer Weg wäre, beide möglichen Sprungziele zu `prefetchen´ und spekulativ zu dekodieren. Das verdoppelt nicht nur die ersten Stufen der Pipelines, sondern erhöht stark den Busverkehr, verstärkt also Ressourcenkonflikte beim Speicherzugriff. Außerdem würden aufeinanderfolgende bedingte Sprünge das Ganze vervierfachen, verachtfachen ... Einige Mainframes (IBM 370/Mod 168, IBM 3033) haben Mehrwegeverfahren implementiert, bei den Micros kenne ich kein entsprechendes Design. Vielmehr sind verschiedene Mechanismen entwickelt worden, um mit einer hohen Wahrscheinlichkeit das richtige Sprungziel vorauszusagen.

So könnte der Instruction Set auch `Branch likely´ unterstützen: der Compiler gibt jedem Branch eine `Vorzugsrichtung´ mit auf den Weg (HP-PA-RISC, MIPS ab 4000, PowerPC). Oder aber der Prozessor entscheidet anhand des Sprungziels, nach der Prämisse, daß Rücksprünge bei Schleifen wahrscheinlicher sind als Vorwärtssprünge (Defaultmodus PowerPC 601). Ganz so leicht kann man sich die Sache allerdings nicht machen. In einer ausführlichen Analyse anhand des verbreiteten SPEC89-Benchmarks zeigten Ball und Laurus [7], daß dem bei weitem nicht so ist.

Zunächst einmal unterscheidet man zwischen solchen Sprüngen, die Schleifen kontrollieren (loop branches), und schleifenfreien (non loop branches). Ihr Anteil variiert von 4 Prozent (matrix300) bis zu 73 Prozent (gcc). Dann gibt es auch bei den Schleifen kontrollierende Vorwärtssprünge (non-backward branches), die bis zu 45 Prozent ausmachen können. Die einfache Favorisierung von Rückwärtssprüngen führt also kaum zum Ziel.

Haben die Branch-Befehle ein Bit für die Vorzugsrichtung, so sind üblicherweise profilierende Compiler gefordert. Das heißt, man muß ein Programm nicht nur übersetzen, sondern auch längere Zeit `einmessen´, damit der Compiler letztendlich die häufigeren Richtungen einsetzt. Ball/Laurus weisen allerdings nach, daß es Heuristiken gibt, mit denen der Compiler von vornherein mit einer hohen Trefferquote den häufigeren Weg weisen kann. Dennoch haben all diese `statischen´ Vorhersagen das Manko, daß sie für die ganze Programmlaufzeit festliegen, sie bleiben auf eine Trefferquote von 75 bis 90 Prozent beschränkt.

Anders arbeiten die bei modernen x86 und RISC-Prozessoren zu findenden Sprungvorhersage-Einheiten (Branch Prediction Unit BPU), die von den statischen Branch Units (PowerPC 601) zu unterscheiden sind. Die BPUs merken sich in einem Puffer (Branch Target Buffer BTB oder Branch Target Access Cache BTAC) zu den letzten bedingten Sprüngen die jeweilige Sprungrichtung. Die einfachste Form der Vorhersage (1-Bit-Predicton) ist, daß der Prozessor die letztausgeführte Sprungrichtung beibehält (Alpha 21064). Bei wiederholten Schleifen führt das aber grundsätzlich zu zwei Fehlvoraussagen. Bei zwei Bits `History´ pro abgespeichertem

Sprung in der `Branch History Table´ (BHT), dominiert hingegen die häufigere Richtung. Resultat: nur eine Fehlvorhersage pro Schleife.

Dieses üblicherweise verwendete Zwei-Bit-Schema hat ungefähr eine Vorhersagequalität von 90 Prozent - wenn der Sprung bereits im BTB eingetragen ist. Berücksichtigt man die beschränkte Größe des Buffers von meist 512 Einträgen (M1, UltraSparc, R10000) so reduziert sich die Trefferquote auf etwa 86 Prozent.

Wesentlich mehr Aufwand treibt der P6, was Wunder, kostet doch eine falsche Voraussage gleich 13 Straf-Takte (gegenüber 3 beim M1). Spezielle Algorithmen (Yeh, mit 4 History-Bits) können auch kompliziertere Sprungmuster erkennen, die Trefferquote steigt damit auf über 90 Prozent. Würde man noch die neuen bedingten Speicherbefehle des P6 hinzunehmen, käme man bei entsprechend optimierter Software vielleicht auf 95 Prozent und mehr. Bedingte Speicherzugriffe kennen die meisten modernen RISC-Architekturen, beziehungsweise fügen sie gerade hinzu (UltraSPARC), Ausnahme: PowerPC. ARM ist in dieser Beziehung besonders mächtig, seine ISA erlaubt es, jeden Befehl mit einer Bedingung zu versehen, was viele Sprünge unnötig macht.

	Branch History Table	statisch	Delay-Slots	spec. Level
486	-	nein	nein	0
Pentium	2 Bit × 256	nein	nein	0
M1	2 Bit × 512	nein	nein	4
NexGen	2 Bit × 2k	nein	nein	2
K86	1 Bit × 1k*	nein	nein	1
P6	4 Bit × 512	nein	nein	2
PPC601	-	ja	nein	2
PPC604	2 Bit × 512	ja	nein	2
PPC620	2 Bit × 2k	ja	nein	2
R3000	-	nein	ja	0
R4000/4400	-	ja	ja	0
R10000	2 Bit × 512	ja	ja	4
PA-8000	3 Bit × 256	ja	ja	?
UltraSPARC	2 Bit × 512	nein	ja	0
Alpha 21064	1 Bit × 2k	nein	nein	0
Alpha 21064A	2 Bit × 4k	nein	nein	0
Alpha 21164	2 Bit × 2k	nein	nein	0

+im Instruction-Cache

Implementierung einer Sprungvorhersage bei (noch) gängigen Prozessoren[1]

8.5.4 Pipeline-Tiefen

Zum Schluß noch ein Vergleich (noch) gängiger Prozessoren bezüglich der eingesetzten Pipeline-Stufen

	ALU..Load/Store
i486	5
Pentium	5
P6	12..17
M1	7
NexGen	7
K86	5(6)

[1] c´t 8/95 Architektur enthüllt, S. 234

MIPS R4000/4400	8
MIPS R4600	5
MIPS R10000	5..7
PowerPC 601	4..5
PowerPC 604	6
PowerPC 620	5
Alpha 21x64	7
SuperSPARC	4
UltraSPARC	5
PA-RISC 8000	7..9

9 Erweiterung des Pipeline-Konzeptes

9.1 Multiple Instruction Issue

Durch ihre effektive **Phasenpipelines** erreichen viele RISC-Architekturen CPI-Werte zwischen 1,0 und 1,5. Um mit einem Prozessor pro Taktzyklus mehr als ein Ergebnis zu erhalten (CPI < 1.0) , werden verschiedene Techniken eingesetzt. Diese Techniken kann man unter dem Begriff der Multiple Instruction Issue zusammenfassen. Darunter versteht man das Anstoßen der Ausführungsphase von mehr als einem Befehl innerhalb eines Taktzyklus. Wenn in einem Prozessor in der Ausführungsphase gleichzeitig mehrere Funktionseinheiten auf unterschiedlichen Maschinenbefehlen eines Befehlsstroms aktiv sein können, dann spricht man vom **Funktionspipelining**. Im folgenden werden zwei erfolgreiche Techniken beschrieben, mit denen der CPI-Wert unter 1 gesenkt werden kann.[1]

9.1.1 Superpipelining

Die Pipeline bedingt eine Aufteilung der Befehlsbearbeitung in verschiedene Segmente. Bei einer weiteren Verfeinerung dieser Segmente und einer Zuordnung dieser verfeinerten Stufen zu unterschiedlichen Pipelinestufen, spricht man von **Superpipelining**. Ein Beispiel für eine Superpipeline-Architektur könnte wie im folgenden Bild aussehen:

n	IF_1	IF_2	D	R	EX_1	EX_2	MA_1	MA_2	W				
n+1		IF_1	IF_2	D	R	EX_1	EX_2	MA_1	MA_2	W			
n+2			IF_1	IF_2	D	R	EX_1	EX_2	MA_1	MA_2	W		
n+3				IF_1	IF_2	D	R	EX_1	EX_2	MA_1	MA_2	W	
n+4					IF_1	IF_2	D	R	EX_1	EX_2	MA_1	MA_2	W

Ablaufdiagramm einer Superpipeline

Jede Pipelinestufe wird in zwei neue Stufen unterteilt. Auch die Cache-Zugriffe erfolgen in zwei aufeinanderfolgenden Pipelinestufen. Da in jeder Stufe nur noch die halbe Arbeit zu leisten ist, kann auch eine **Verdoppelung der Taktfrequenz** und eine **Halbierung der Zeit zwischen dem Einschleusen aufeinanderfolgender Befehle** realisiert werden. Damit wird der Durchsatz an Befehlen und die Leistungsfähigkeit verdoppelt.

9.1.2 Superskalar-Architekturen

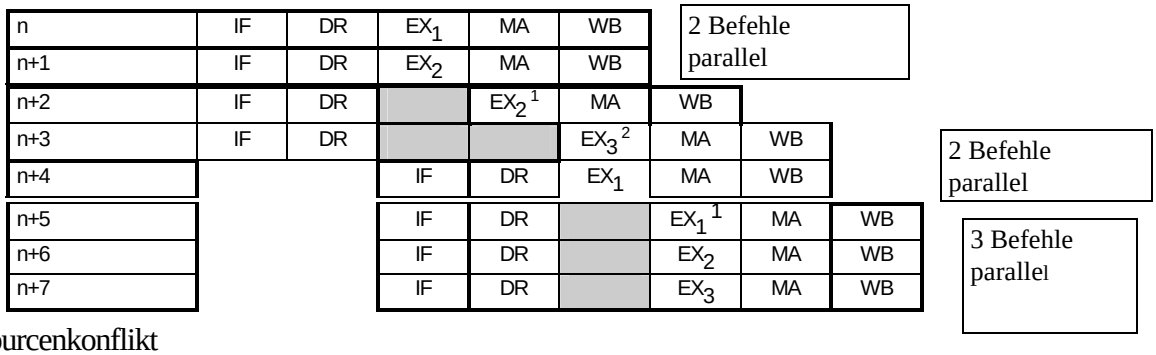
Superskalare Rechnerarchitekturen kombinieren effizientes Phasenpipeling der Befehlsabarbeitung mit Funktionspipelining in der Ausführungsstufe der Prozessor-Pipeline. Mehrere Funktionseinheiten nutzen die Parallelität des Befehlsstroms. Durch gleichzeitige Ausführung mehrerer voneinander unabhängiger Operationen pro Taktzyklus können sich bei ideal beschaffenem Befehlsstrom CPI-Werte unter 1.0 ergeben. Die Funktionseinheiten sind dabei meist auf bestimmte Befehlstypen spezialisierte und keine vollständigen universellen Rechenwerke. So hat zum Beispiel der Alpha-Prozessor nicht zwei Rechenwerke, sondern eine Funktionseinheit für Integer- und eine für Floating-Point-Arithmetik. Liegen in der Befehlspipeline zwei Integer-Befehle an, können

[1] Vgl. Märtin, a.a.O. S. 190

diese also nicht auf zwei Rechenwerke verteilt werden. Der Alpha-Chip ist also nicht superskalar. Demgegenüber ist der Pentium-Prozessor mit zwei Integer-Einheiten und einer Floating-Point-Einheit ausgestattet und gehört deshalb zu den Superskalar-Architekturen.

Das Superpipelining erfordert durch sein Konzept hohe interne Taktfrequenzen, mit denen das Design schneller als bei einfachen Pipelines an die Grenzen der Technologie stößt. Um dieses zu umgehen und die Taktfrequenz in der Größenordnung wie bei einfachen Pipelines zu halten, sieht die Superskalar-Architektur die parallele Abarbeitung mehrerer Befehle vor.

Im folgenden Bild ist eine Superskalar-Architektur skizziert: Es werden bei jedem Zugriff 4 Befehle eingelesen, die dann soweit wie möglich parallel ausgeführt werden. Die richtige Zuweisung der Befehle an die einzelnen Ausführungseinheiten übernimmt ein Scheduler. Die Berücksichtigung sämtlicher Konsistenzkonflikte wird dabei durch ein Scoreboard unterstützt. Bei Bedarf wird dann wieder ein Satz von Befehlen gelesen, um den Grad der Parallelität so groß wie möglich zu halten.



[1] Resourcenkonflikt

[2] Datenkonsistenzkonflikt

Ablaufdiagramm einer Superskalar-Architektur mit drei Ausführungseinheiten

Während Pipelines am meisten mit falsch vorhergesagten Verzweigungen kämpfen, haben superskalare Prozessoren das Problem, das serielle Programme in parallel ausführbare Befehle zu zerlegen. Sonst stehen die Fließbänder ungenutzt herum, wie etwa beim Pentium, bei dem mit normalem, nicht optimierten x86-Code nur wenige Prozent Parallelbetrieb der beiden Piepse realisiert ist. Die Hauptprobleme bei der Parallelisierung sind Abhängigkeiten zwischen den Befehlen und die Mehrfachbenutzung der spärlichen Register. RISC-Prozessoren mit 32 und mehr Universalregistern sind von solchen Abhängigkeiten deutlich seltener betroffen als der x86 mit seinen 8 Registern, von denen eines (EAX) immer noch im Sinne des klassischen Akkumulators wesentlich häufiger eingesetzt wird als der Rest. Daher ist Register Renaming für superskalare x86-Architekturen nahezu zwingend. [1] Um die Wirkungsweise dieser Technik verstehen zu können, muß man sich mit einem anderen Prinzip moderner Prozessoren vertraut machen. Moderne Mikroprozessoren besitzen häufig die Fähigkeit, Instruktionen in einer anderen Reihenfolge auszuführen, als sie im Instruktionsstrom vorkommen. Das wird deshalb gemacht, um die Ausführung nachfolgender Befehle nicht ins Stocken geraten zu lassen, wenn ein Befehl längere Zeit für seine Ausführung benötigt.

[1] Vgl. Andreas Stiller, Architektur enthüllt, in c´t August 1995, S.234

9.2 Out of Order Execution

Das kann natürlich nicht beliebig gemacht werden, denn die Ausführung nachfolgender Befehle hängt auch von vorangegangenen Befehlen ab. Im folgenden Beispiel eines fiktiven Prozessors gibt es Abhängigkeiten der Reihenfolge der Befehle, die man in verschiedene Klassen aufteilt.

```
I1: R3 := R3 op R5
I2: R4 := R3 + 1
I3: R3 := R5 + 1
I4: R7 := R3 op R4
```

9.2.1 True Dependancy

Am einfachsten lassen sich die sogenannten „wahren Abhängigkeiten“ (true dependancies) im Instruktionsstrom erkennen. Instruktion 2 kann nicht vor Instruktion 1 ausgeführt werden, denn das Ergebnis von I1 ist für das Ergebnis von I2 von Bedeutung (siehe Abschnitt über Datenkonflikte beim Pipelining). Genauso verhält es sich mit I4 zu I3 und I2. Diese Abhängigkeit wird von der tatsächlichen Reihenfolge der Instruktionen im Programm und deren Ergebnissen vorgegeben.

9.2.2 Output dependence

Wenn der Prozessor die Fähigkeit besitzt, Instruktionen in einer anderen Reihenfolge fertigzustellen - Instruktionen also zu vertauschen - kommt es zu weiteren Konflikten. In unserem Fall könnte zum Beispiel die Ausführung I1 länger dauern als von I3. Um die Ausführung der Befehle nicht stoppen zu lassen, könnte - dank getrennter Ausführungseinheiten - I3 eher als I1 ein Ergebnis zurückliefern. Zunächst würde R3 also das Ergebnis von I3 bekommen und erst danach das von I1. In diesem Fall erhält I4 aber einen falschen Wert für seine Operation.

9.2.3 Antidependency

Wenn der Prozessor zudem über die Fähigkeit verfügt, die Befehle durch getrennte Ausführungseinheiten nicht nur in einer anderen Reihenfolge fertigzustellen, sondern diese schon in einer anderen Reihenfolge aus dem Instruktionsstrom holt, ergibt sich ein weiteres Problem. I3 kann nicht eher ausgeführt werden, bevor I2 fertig ist. Sonst wird eventuell R3 von der eher ausgeführten dritten Instruktion überschrieben, obwohl I2 das Ergebnis von I1 erwartet.

Die einfachste Lösung für diese Probleme wäre, die betreffenden Registerinhalte nicht zu vertauschen und die Ausführungseinheiten an eine strikte Reihenfolge der Abarbeitung zu binden. Das hätte zur Folge, daß die Ausführung von Befehlen doch gestoppt wird und der Prozessor in diesen fällen wie ein normaler Prozessor arbeitet. Leider sind diese Abhängigkeiten in einem Programm nicht die Ausnahme, sondern die Regel. Und wenn der Prozessor seine Fähigkeiten meistens nicht ausspielen darf, kann man den ganzen Spaß auch gleich unterlassen .

9.3 Register Renaming

Ein Ausweg biete sich über das Register Renaming an. Prinzipiell macht man dabei nichts anderes, als das Register im Prozessor beliebig oft zu kopieren. Immer dann, wenn der Wert eines Registers überschrieben wird (im Beispiel R3 in I1 und I3), wird von dem betreffenden Register eine Kopie angelegt, und diese Kopie erhält danach den neuen Wert. Praktisch umsetzbar ist das, indem etwa beim x86 Prozessor für die acht Register, die der Programmierer sieht (EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP) weitere Hintergrundregister zur Verfügung stehen. Bei Bedarf ist dann eines der Hintergrundregister für den Inhalt eines der Programmierregister verantwortlich. Der Trick liegt nun darin, daß verschiedene Hintergrundregister für das gleiche Programmierregister verwendet werden, und zwar abhängig davon, wie die Instruktionen vertauscht wurden. Dadurch kann man auch bei vertauschten Befehlen immer mit dem korrekten Inhalt eines Registers arbeiten. Mittels Register Renaming würde das Beispiel folgendermaßen aussehen:

I1: R3b := R3a op R5a
I2: R4a := R3b + 1
I3: R3c := R5a + 1
I4: R7a := R3c op R4a

Ein Register bekommt immer dann ein neues Hintergrundregister zugeordnet, wenn der Inhalt im Begriff steht, überschrieben zu werden. Da I4 jetzt das Register R3c benutzt, kann I3 zu einer beliebigen Zeit vor I2 oder I1 ausgeführt werden, denn diese benutzen das Register R3b. Auch I2 könnte jetzt später ausgeführt werden, da es das Register R3b benutzt.

Das Konzept stößt an seine Grenzen, wenn alle Hintergrundregister benutzt werden. Die wahre Abhängigkeit zwischen I1 und I2 bleibt außerdem erhalten; diese kann man nicht umgehen, aber zumindest verkürzen. Denn das Ergebnis von I1 könnte nicht nur in R3b geschrieben, sondern gleich an die Ausführungseinheit weitergeleitet werden, die mit der Berechnung von I2 beschäftigt ist.

Noch komplizierter wird es, wenn ein Interrupt auftritt. Für diese Fälle muß ein erheblicher Anteil der Chip-Hardware dafür verwendet werden, den Registern trotzdem die richtigen Inhalte zuzuweisen.[1]

9.4 Historisches

Die Anfänge des Pipelining lagen in den frühen 60er Jahren. Als erste Universal-Pipeline-Maschine wird die Stretch betrachtet, die IBM 7030. Stretch folgte nach der IBM 704 und hatte das Ziel, 100 mal schneller zu sein als diese. Das Ziel war, den Stand der Technik zu diesem Zeitpunkt bedeutend voranzutreiben (Stretch). Man erreichte einen Überlappungsfaktor von 1,6 in einer Vier-Stufen Pipeline.

1964 lieferte CDC die erste CDC 6600. Sie war in vielerlei Hinsicht einmalig. Zusätzlich zur Einführung des Scoreboards war die CDC 6600 die erste Maschine, die in einem erheblichen Umfang von mehrfachen Funktionseinheiten Gebrauch machte (superskalare Architektur). Der Zusammenhang zwischen Pipelining und Befehlssatzentwurf wurde verstanden. So wurde der Befehlssatz einfach gehalten, um das Pipelining zu unterstützen. Die CDC 6600 kannte bereits Out of Order Execution und hatte einen Befehls-Scheduler für die Fortran Compiler !

---

[1] Vgl. Elektronik, Fachzeitschrift für Industrielle Anwender und Entwickler, 25. Juli 1995, „Konkurrenz für den Pentium, neue Prozessoren unter die Lupe genommen“ S. 81

Die IBM 360/91 führte viele neue Konzepte ein, wie Register Renaming und Forwarding. Man entwickelte sogar bereits eine Methode zur Verzweigungsvoraussage.

Die RISC-Maschinen verfeinerten die Idee der Pipeline mit Compiler-Scheduling in den frühen 80er Jahren. Das Konzept der verzögerten Verzweigungen (Branch Delay Slots) - bekannt aus der Mikroprogrammierung - wurde auf die Maschinenarchitektur ausgeweitet.

In letzter Zeit gab es einige Veröffentlichungen zu Entscheidungskriterien zwischen alternativen Pipelining-Methoden. Jouppi und Wall (1989) untersuchten die Leistungsunterschiede zwischen Systemen mit Superpipelining und Superskalarität und kamen zu dem Ergebnis, daß ihre Leistung ähnlich ist, daß aber Superpipeline-Maschinen weniger Hardware erfordern, um dieselbe Leistung zu erzielen.

10 CISC mit RISC-Design

10.1 AMD K5[1]

Der Pentium-Konkurrent soll demnächst in Echt-Silizium vorliegen und dann seine Codebezeichnung K5 gegen den Namen `AMD K86´ eintauschen. Laut Johnson ist die Hardwareemulation (auf dem PiE/Quickturn-Emulatorsystem `Mars´ mit 2,5 Millionen Gates und 500 kHz Takt) erfolgreich abgeschlossen, das `Tape out´ zur Herstellung der ersten Prototypen erfolge am 20. Oktober. Auf dem Emulator laufe inzwischen Windows 3.1 und seit September auch Unix im Debug-Mode. Mit dem K86-Design will sich AMD nicht nur vom Ruch des Intel-Epigonen (und den damit verbundenen Gerichtsterminen) befreien, sondern auch die ehemalige Ziehmutter an Prozessorleistung übertrumpfen.

Dabei setzen Johnson und Mitstreiter auf ein superskalares Konzept, das die parallele Ausführung von bis zu vier Instruktionen in insgesamt sechs Funktionseinheiten (zwei Integer, zwei Load/Store, Branch-Unit und FPU) erlaubt. Allein damit wäre zunächst nicht viel gewonnen, wie die Leistungen des ebenfalls superskalar aufgebauten Pentium zeigen: wegen zu starker Abhängigkeiten der komplexen Intel-Instruktionen bei nicht optimiertem Code bleiben die tatsächlichen Leistungen von Intels Primus weit hinter seinen theoretischen Möglichkeiten zurück. Im Grunde böte sich ein reines RISC-Konzept als elegante Lösung des Problems an, doch der gigantische x86-Markt verlockt zu teils abenteuerlichen, teils pfiffigen CISC/RISC-Lösungen, so wie es NexGen vorgemacht hat.

Beim K86 markiert ein Predecoder über sogenannte `Tags´ zunächst die Längen und Art der CISC-Instruktionen, bevor sie im vierfach-assoziativen Instruction-Cache von 16 KByte Größe landen. Dadurch wird die Arbeit für den Decoderteil der nachfolgenden Pipelines wesentlich erleichtert. Er wandelt x86-Befehle variabler Länge in RISC-Operationen (ROPs) jeweils gleicher Länge um. Einfache x86-Instruktionen, etwa Register-to-Register-Operationen ergeben dabei auch ein ROP. Memory-to-Register-Operationen brauchen zwei (load, op) und Register-to-Memory drei ROPs (load, op, store). Aufwendige Instruktionen (etwa String-Bearbeitung) unterzieht der Decoder einer Sonderbehandlung: Sie laufen auf eine Mikrocode-Routine (völlig `Intel-clean´), welche eine entsprechende ROP-Sequenz erzeugt.

Decoder beziehungsweise Mikrocode liefern 4 ROPs pro Takt, mit denen sich die sechs Funktionseinheiten füttern lassen. Diese Einheiten sind völlig unabhängig voneinander und können die jeweiligen ROPs je nach verfügbaren Ressourcen `Out of Order´ ausführen und zurückschreiben, solange zwischen den Instruktionen keine echten Abhängigkeiten bestehen. Ein Reorder-Buffer samt zugehörigem Mechanismus sorgt hinterher allerdings wieder für die richtige Reihenfolge der Ergebnisse, nebenher übernimmt er die Verwaltung und Zuordnung der 16 logischen Register des Prozessors zu den 8 virtuellen x86-Registern. Aufgrund ihrer einfacheren Struktur bestehen zwischen ROPs deutlich weniger echte Abhängigkeiten als zwischen den zugrundeliegenden CISC-Instruktionen. Deshalb läßt sich ihre Ausführung wesentlich stärker parallelisieren; im optimalen Fall dekodiert der K86 vier x86-Instruktionen pro Takt und führt sie ebenso schnell aus. Typische, nichtoptimierte

[1] Entnommen einem Artikel C´T Heft 12 1994, „Alternativen gefällig?“

Software führt laut Analyse von AMD bei 16-Bit-Code zu einem durchschnittlichen Übersetzungsverhältnis von 1,9 ROPs pro x86-Instruktion, bei 32-Bit-Code verbessert es sich auf 1,3 ROPs/Instruktion.

Um die Effektivität der Pipeline bei Verzweigungen zu steigern, bedient sich der K86 eines kombinierten Mechanismus aus Sprungzielvorhersage (branch prediction) und einer vorweggenommenen Ausführung der Instruktionen am Sprungziel (speculative execution). Die Ergebnisse bleiben solange im Reorder-Buffer erhalten, bis das wirkliche Sprungziel festliegt. Hat die Branch-Prediction richtig geraten, spart man einen Takt, andernfalls wird der Cache-Block mit der richtigen Adresse geladen und die Vorhersage auf selbige gesetzt.

Der Lohn all dieser ausgefeilten Architekturtricks soll sich laut AMD bei gleicher Taktfrequenz in einer um dreißig Prozent höheren Integer-Performance gegenüber dem Pentium äußern, die Fließkommaleistung wird allerdings nur auf dem Niveau des Konkurrenten liegen. Der K86 kann direkt als Alternative in bestehende Pentium-Designs (P54C) eingesetzt werden, benötigt also keine speziell auf ihn zugeschnittenen Chipsätze und Boarddesigns. Er unterstützt auch das Pentium-Power-Management. AMD will den Prozessor zunächst in einem 3-Metal-Layer 0,5-µ-CMOS-Prozeß in Silizium ätzen, dann verträgt der 4,2-Millionen-Transistor-Chip 100 bis 120 MHz Takt. Ab 1996 soll auf einen 0,35-µ-Prozeß (150 MHz Takt) umgestellt werden. Ehe aber Normalsterbliche die ersten Exemplare des edlen Vielbeiners in den Händen halten, müssen sie sich vermutlich bis zum dritten Quartal 1995 gedulden. Der K86 soll in der gleichen Preisregion wie der P54C liegen, als `Einführungstakt´ sind 100 MHz geplant. Das ergäbe eine Performance von etwa 130 SPECint92, etwas mehr als ein bis dahin sicherlich erhältlicher Pentium-P120.

10.2 Cyrix M1

Nachdem sich Cyrix mit IBM und SGS-Thompson potente Bundesgenossen verschafft hat, bestehen für ihren Pentium-Konkurrenten M1 wohl gute Chancen, in Zukunft eine gewichtige Rolle auf PC-Mainboards zu spielen. Cyrix will alle Details zum M1 aber erst zur Comdex aufdecken, c't erhielt die Spezifikationen schon mal vorab.

Im Gegensatz zu Cyrix verwendet AMD keinen echten RISC-Prozessorkern, sondern hat sich auf eine Verbesserung und Erweiterung des Pentium-Designs konzentriert. Das Blockschaltbild des Prozessors erscheint dem Intel-Kenner denn auch recht vertraut: Zwei Integer-Pipelines, eine Fließkommaeinheit und die Branch-Prediction halten auf den ersten Blick nichts aufregend Neues bereit. Die wesentlichen Unterschiede zum Pentium liegen im Detail: So gelten für die parallele Ausführung komplexer x86-Instruktion weniger Einschränkungen als beim Pentium, zudem hat Cyrix nach eigenen Angaben für eine schnellere Abarbeitung von `Exklusiv´-Befehlen gesorgt, die sich nicht mit einfachen Instruktionen für die zweite Pipeline paaren lassen. Dabei verfügt die Decoderstufe über eine gewisse Eigenintelligenz, welche die dekodierten Instruktionen der jeweils passenden Pipeline zuweist. Diese Eigenschaft, zusammen mit der in sieben Stufen unterteilten Pipeline (superpipelined) soll laut Cyrix den Einsatz eines optimierenden Compilers in der Regel überflüssig machen.

Der RISC-Kern des K86 entspricht weitgehend dem des superskalaren 29K-Prozessors, allerdings mit FPU und vorgeschalteten Predecoder samt `ROP-Konverter´.

Auch bei der Sprungzielvorhersage hat sich einiges getan: Der M1 wartet ebenso wie der K86 mit einem Mechanismus für speculative execution und branch prediction auf. Er kann dabei bis zu vier Sprünge tief spekulieren, erst darüber hinaus muß er in eine völlig ungewisse Zukunft schauen.

Mit der Fähigkeit zur `Out-of-Order´-Ausführung von Instruktionen teilen M1 und K86 ein weiteres Architekturmerkmal. Auch der M1 verfügt über deutlich mehr Register (32) als der Pentium (8), die sich dynamisch

zuordnen lassen (Register-Renaming). Damit kann er in vielen Fällen echte Datenabhängigkeiten zwischen den Instruktionen in der Pipeline auflösen und Verzögerungen in der Execution-Stufe vermeiden.

Der M1-Cache hat eine neuartige Struktur. Er besteht aus einem gemeinsamen Instruktions- und Daten-Cache mit 16 KByte (wie etwa beim iDX4) in Verbindung mit einem vollassoziativen Look-aside-Cache von 256 Byte für Instruktionen. Bei einem gemeinsamen Cache allein könnten sich Code und Daten immer wieder gegenseitig aus dem Cache werfen (das gefürchtete Thrashing), was manch kleine Schleife erheblich verlangsamt.

Ebenso wie der K86 ist der M1 für den SPGA-Sockel des P54C designt. das Pinout unterscheidet sich jedoch geringfügig, was ein paar Jumper nötig macht. Spezielle Chipsätze braucht der M1 nicht, für die angepeilten 100 MHz (zweifachgetaktet) sind jedoch asynchrone PCI-Chipsätze sinnvoll, um die volle PCI-Performance mit 33 MHz aufrechtzuerhalten.

Zunächst will Cyrix, ebenso wie AMD, in einem 3-Metal-Layer-0,5-µ-CMOS-Prozeß fertigen (lassen). Später soll ein höher aufgelöster Prozeß mit zusätzlichen Layern zum Einsatz kommen, der höhere Clockraten erlaubt. Im Vergleich zur Intel-Konkurrenz zeigt sich der Hersteller reichlich optimistisch: Simulierte Benchmarks (Mips, ZiffDavis) ergäben bei gleichem Takt einen 30- bis 50prozentigen Leistungszuwachs im Vergleich zum Pentium.

Erste Muster will Cyrix demnächst ausliefern, für Stückzahlen wird im Augenblick das erste Quartal 95 gehandelt. So ganz mag man diesen Versprechungen aber nicht trauen: Laut Aussage des Marketing-Chefs Dan Auton läuft das Design bis jetzt nur als reine Softwareemulation (also in Taktregionen von 1 bis 10 Hz!). Boot-Erlebnisse hatte Cyrix noch nicht, da stehen bei Windows oder Unix noch allerhand Überraschungen bevor.

11 Optimierende Compiler

11.1 Einführung

Die Systemleistung eines RISC-Systems ist bestimmt durch das Zusammenspiel zwischen Betriebssystem, Hardware und. Software. Diese Zusammenspiel muß optimal ausgelegtsein.

Eine performante Systemleistung hängt von kurzem und schnellem Code für die Maschine ab.

Ein RISC-System besitzt nicht eine einzige sondern mehrere Hardware-Eigenschaften.

Daher

- kann o.g. Zusammenspiel komplex sein
- müssen die Hardware-Mechanismen und der Software-Code müssen zusammenwirken

Die Aufgabe des Compilers ist es, diese Anforderungen erfüllen:

Er muß die Ziel-Hardware kennen und ihre Abhängigkeiten beachten (Cache-Größen, Phasenpipelinig ohne/mit Scordeboarding, Funktionspipelining)

11.2 Funktionsweise eines Compilers

Der Compiler hat die Aufgabe, einen Quellcode semantikgetreu in eine Zielsprache (i.d.R. Maschinensprache) zu transformieren.

Die Compilerfunktion gleidert sich in zwei Phasen:

Analysephase

Der Compiler

- liest den Quellcode (Lexikalische Analyse)
- erkennt anhand der gegebenen Sprachsyntax Schreibfehler (Syntaktische Analy se)
- Überprüft Typverträglichkeit der Variablen (Semantische Analyse)
- analysiert Datenabhängigkeiten und Durchlaufhäufigkeiten des Quellprogramms

Synthesephase

Der Compiler

- erzeugt einen Zwischencode
- ordnet Variablen zu Speicherplätzen bzw. Registern zu
- baut Laufzeitprüfungen in den Zwischencode ein
- optimiert nach best. Verfahren vor, nach oder während der Registerzuordnung den Zwischencode anhand der in der Analysephase gewonnenen Informationen
- überführt den optimierten Zwischencode schließlich in die Zielsprache

11.3 Optimierungsmöglichkeiten eines Compilers

Das wesentliche Ziel der Codeoptimierung ist die Überführung v. Codesequenzen in „schnellere“ Codestücke unter Beachtung der Auflagen:

Semantik Quellprogramms darf nicht verändern werden

und

der Codeumfang soll reduziert, zumindest nicht erweitert werden

Grundsätzlich:

Bei den Optimierungsverfahren müssen stets Laufzeitvorteile gegen Codeverlängerung abgewogen werden, da einige der unten aufgeführter Optimierungsverfahren dem Ziel der Codereduzierung widerspricht.

Optimierungsverfahren lassen sich in globale und lokale Verfahren klassifizieren. Lokale Verfahren beziehen sich auf einen Basisblock, einer Codesequenz die zusammengehörig ist und nicht durch Sprünge unterbrochen wird. Globale Verfahren wirken über den gesamten Code hinweg.

Lokale Verfahren verfolgen die Ziele:

- schneller, übersichtlicher zu machen:
- überflüss. Befehle eliminieren
- schnellere Befehle
- günstigere Befehlsfolgen

Gefahr: ohne Kenntnis globaler Informationen kann durch lokale Optimierung ein evtl. besseres Ergebnis durch globale Optimierung zunichte gemacht werden.

Globale Information sammelt der Compiler in der Analysephase (s.o.).

11.3.1 Maschinenunabhängige Optimierungen

(COMPILER-ANALYSEPHASE)

- Constant Folding — Ausdrücke, deren Operanden aus zur Laufzeit unveränderbaren Werten bestehen, werden zur Übersetzungszeit berechnet und ersetzt
- Constant Propagation — Ausdrücke, deren Wert zur Übersetzungszeit errechnet werden kann, werden durch den berechneten Wert ersetzt. (z.B. a := 5^2+PI/2.515)
- Common Subexpression Elimination — Ausdrücke, die mehrfach berechnet werden ohne zwischendurch ihren Wert zu ändern, werden durch Laden d. Ergebnisses der ersten Berechnung ersetzt. (z.B. a:= 5^b+PI/2.515)
- Dead Store Elimination — Speicherplatz nicht mehr verwendeter Variablen wird freigegeben
- Interprocedural Constant Propagation — Die Constant Propagation wird über Prozedurgrenzen hinweg durchgeführt
- Code Inlining — Der Code einer Prozedur wird in den Rumpf der aufrufenden Prozedur expandiert
- Invariant Code Motion — Konstanter Code innerhalb einer Schleife (unabhängig vom Schleifenindex) wird vor die Schleife gezogen.
- Loop Unrolling — Schleifen werden durch Kopieren des Codes (teilweise) linearisiert.
- Loop Jamming — Zwei oder mehr Schleifen gleicher Struktur (gleicher Länge) werden zusammengefaßt.
- Strength Reduction — Arithmetische Operationen werden durch einfachere (schnellere) ersetzt. (z.B Multiplikation 2*i ersetzen durch Shift-Left od. Addition)
- Tail Recursion Elimination — Prozeduren, die end-rekursiv sind (letzte Prozedur-Anweisung ist rekursiver Aufruf), können in Iteraionen umgewandelt werden.
- Deletion of Unreachable Code — In der Analysephase d. Compilers werden Durchlaufhäufigkeiten festgestellt (s.o.). Es wäre zu ermitteln, welche Codestücke auf eine Speicherseite zusammenfaßbar sind, um überflüssiges Paging zu vermeiden. Dieses Verfahren konnte bisher nicht realisiert werden.

11.3.2 Maschinenabhängige Optimierungen

(COMPILER-SYNTHESEPHASE)

• Register Allocation Optimization	Zuordnung von Variablen zu Registern und Speicherplätzen soll hinsichtlich Zugriffszeit optimal sein. (siehe z.B. der Algorithmus von Chaitin)
• Elimination of Jumps-to-Jumps	„Sprünge auf Sprünge“ werden durch direkte Sprünge ersetzt
• LOAD/ STORE Motion	LOAD- und STORE-Befehle werden umsortiert und ggf. aus Schleifen entfernt
• Operand Permutation	Durch Umordnen von Code kann ggf. die Anzahl d. benötigten Register verringert werden. (z.B. durch geschicktes Umsortieren entfällt das Retten best. Register)
• Instruction Scheduling	Code wird so umgeordnet, daß Pipeline möglichst verzögerungsfrei (d.h. ohne das Einfügen vo NOPs) ausgelastet wird.
• Zusätzlich denkbare Optimierungsmöglichkeit: „Optimale Cache-Ausnutzung“	Der Compiler muß die Cache-Größen des Prozessors kennen. Die Kenntnis kann dazu führen, daß z.B. ´Loop-Unrolling´ vermieden wird, da sonst die Cache-Größe gesprengt werden könnte. Dieses Verfahren konnte bisher ebenfalls nicht realisiert werden.

11.4 Detailliertere Darstellung einiger hardwareabhängiger Optimierungen

11.4.1 Registerbelegung nach dem Algorithmus v. Chaitin:

Problem:	Abbilden von Variablen/ Speicherplätzen auf endlich viele Register Das Problem läßt sich auf das graphentheoretische Problem der Einfärbung eines Graphen übertragen. Die zu Verfügung stehenden Farben entsprechen den Registern
Voraussetzung:	Es muß bekannt sein, welche Variablen in demselben Codestück benutzt werden („lebendig“ sind) sowie welche Variablen vor und nach dem Codestück gebraucht wurden bzw. werden. (Compiler-Analysephase !)
Registerkonfliktgraph:	Danach wird der Registerkonfliktgraph konstruiert: Zwischen zwei Knoten (Variablen) existiert dann eine Kante, wenn sie gleichzeitig lebendig sind.
Algorithmus:	1.Suche einen Knoten, der höchstens R-1 Kanten (R - Anz. Register) hat. Lösche diesen Knoten aus dem Graphen 2. Haben die restlichen Knoten mind. R Kanten, wähle man einen davon, um ihn in den Speicher auszulagern und lösche ihn aus dem Graphen. Die

Beispielprogramm:

```
p    := m + n;
r    := n + q;
s    := p + r;
t    := p * r;
u    := p + p;
```

Lebendige Variablen zu Beginn:

m, n,q

Lebendige Variablen am Ende:

q, t, u, s

Wahl fällt auf den Knoten, der bezüglich der Zugriffshäufigkeit am günstigsten ist.

3.Beginne wieder bei (1), falls noch Knoten existieren, sonst (4)

4.Färbe die Knoten, die nicht in den Speicher ausgelagert werden, in der umgekehrten Löschreihenfolge, so daß kein Knoten die Farbe eines Nachbarn im Ausgangsgraphen hat.

Registerkonfliktgraph:

q
u
s
p
t
r
m
n

	m	n	q	r	p	t	u	s
1								
2								
3								
4								
5								

11.4.2 Instruction Scheduling

Ziel: Durch Umsortieren des Codes erreichen, daß die Pipeline ohne Verzögerung arbeiten kann, d.h. Pipeline-Konflikte (s. Skript) softwaremäßig umgangen werden.

Voraussetzung: Erkennen von Datenabhängigkeiten im Code (Compiler-Analysephase) sowie Datenabhängigkeiten im nachinein durch die Registerbelegung verursacht (Compiler-Synthesephase)

Maßnahmen (Software): Um zu verhindern, daß die dem aktuellen Befehl Folgenden in die Pipeline eingespeist werden, werden an diesen Stellen im Code NOPs eingefügt. Da diese OPs auf jeden Fall ausgeführt werden, können sie wiederum durch datenunabhängige Befehle ersetzt werden (s. Bsp.). (Compiler-Synthesephase)

Maßnahmen (Hardware): Der RISC-Prozessor verfügt über SCOREBOARDING, welches die o.g. Einfügung von NOPs, also die Verzögerung von in der Pipeline befindlichen Befehlen, durchführt.

Will ein Befehl auf ein Register zugreifen, welches aber noch nicht im Registerfile verfügbar ist, da im vorhergehenden Befehl nocht in Benutzung, so schafft FORWARDING Abhilfe.

Hierbei wird der neu berechnete Registerinhalt über eine Schaltung von Latches einer vorgelagerten Pipelinephase zur Verfügung gestellt, sodaß die Aktualisierung des Registerfiles nicht abgewartet werden muß.

Beispiele:

1) →

```
SUB   R10, R1, R13
ADD    R1, R2, R3
SUB    R3, R4, R5
ADD   R10, R11, R12
```

2)

```
100   LOAD   X, A
101   ADD    1, A
102   JUMP   105
103   ADD              B, Z
104   SUB    C, B
105   STORE  A, Z
106  ...SUB   R10, R1, R13
ADD   R1, R2, R3
NOP
NOP
SUB   R3, R4, R5
ADD   R10, R11,R12
```

```
100   LOAD   X, A
101   ADD    1, A
102   JUMP   106
103   NOP
104   ADD    B, Z
105   SUB    C, B
106   STORE  A, Z
107   ...
ADD   R1, R2, R3
SUB   R10, R1, R13
ADD   R10, R11, R12
SUB   R3, R4, R5

100   LOAD    X, A
101   JUMP    105
102   ADD     1, A
103   ADD     B, Z
104   SUB     C, B
105   STORE   A, Z
106   ...
```

11.4.3 Aspekte des Funktionspipelining

Bei superskalaren Prozessoren ist neben Phasenpipelining auch Funktionspipelinig realisiert.

Auf dem Prozessor sind parallel arbeitende Ausführungseinheiten für bestimmte Aufgaben (z.B. Integer-und Gleitkommaoperationen, Load/ Store-Aufgaben etc.) mehrfach vorhanden. Dadurch ist der Prozessor in der Lage, mehrere verschiedene Instruktionen parallel auszuführen.

Das Konzept von VLIW-Maschinen (Very Long Instruction Word) ähnelt im wesentlichen den superskalaren Prozessoren, geht jedoch bez. der Parallelität einen Schritt weiter.

Der Prozessor wird bei n Ausführungseinheiten auch mit n Befehlen versorgt.

Es existieren hier im Gegensatz zu superskalaren Prozessoren soviele Befehlsströme, wie Ausführungseinheiten vorhanden sind.

Dem Compiler (genauer: der Trace-Scheduler) obliegt das Erkennen von Parallelität im Code und führt die Optimierungsmaßnahme des Umsortierens von Code (Instruction Scheduling) zusätzlich nach diesem Kriterium durch.

Das Element des Compilers, das die Umsortierung vornimmt, nennt man den ´Trace-Scheduler´. Er bezieht seine Informationen aus der Analysephase des Compilers (Flußgraph, s.o.).

11.5 Zukunftsausblick

Eine zukunftsorientierte Weiterentwicklung von RISC-Systemen wird sich im wesentlichen auf dem Architektur- bzw. Technologiesektor abspielen.

Architektur:

Erhöhung der Breite der Maschinenworte (20-30 Worte) bei VLIW-Systemen, bei techn.-wiss. Anwednungen soll durch den Compiler eine Parallelisierung bis zu 20fach (s.o) möglich werden.

Außerhalb von VLIW-Systemen soll die Maschinenwortbreite 2-4 Worte erreichen. Die Hardware übernimmt hierbei die Aufgabe der Parallelisierung. (CPI = 0.5 bei Parallelisierung durch Hardware ohne weitere Unterstützung).

Erhöhung der Integrationsdichte auf der CPU bezüglich:

- Cache
- Controller
- MMU (Memory Managment Unit)
- FPU (Floating Point Unit)

Konsequenzen für den Compiler:

Der Compiler muß:

- Informationen über Architektur berücksichtigen (Cachegröße, Hauptspeicher etc.)
- höhere Taktraten berücksichtigen

Weiterhin muß die Compilertechnik die Potentiale bei der vertikalen Optimierung (Pipelineaspekte, Instruction Scheduling, s.o.) und der horizontalen Optimierung (Parallelisierungsgrad nach Anzahl zur Verfügung stehender AE) erreichen.

12 Vergleich von RISC und CISC Compiler

Im folgenden wird Material angeboten, anhand dessen exemplarisch einige Compiler-Optimierungen bei einem ausgereiften VAX-FORTRAN CISC-Compiler studiert werden können. Bei diesem Compiler kann der Optimizer optional eingeschaltet werden. Das Beispielprogramm wurde zusätzlich mit einem VAX FORTRAN RISC-Compiler auf einer DEC Alpha übersetzt. Bei jedem Beispiel sehen Sie zuerst den FORTRAN Source-Code und dann den zugehörigen Assembler-Code.

Untersuchen Sie, wie sich der erzeugte optimierte vom nichtoptimierten Code unterscheidet. Untersuchen Sie weiterhin, wie sich der erzeugte CODE eines CISC-Compilers von dem des RISC Compilers unterscheidet. Es sollte dabei nicht notwendig sein, die DEC Assembler genauer zu kennen.

12.1 CISC-Compiler-Beispiel mit Optimizer

```
                                                    15-Jul-1993 16:28:49      VAX  FORTRAN
V5.7-133                Page   1
                                                    15-Jul-1993                   16:04:15
       US12$ROOT:[FROEHLICH]COMP1.FOR;2

00001          OPTIONS /EXTEND_SOURCE
00002          PROGRAM OptimizeTest1
00003          IMPLICIT  NONE
00004
00005          INTEGER*4  I
00006          INTEGER*4  V1  /10/
00007          INTEGER*4  V2  /20/
00008          INTEGER*4  V3
00009          INTEGER*4  V4
00010          INTEGER*4  V5
00011          INTEGER*4  V6
00012          INTEGER*4  V7
00013
00014          DO I = 1,10
00015              V3 = I * 2
00016              V4 = I * V3
00017              V5 = I * 10
00018              V6 = I * V1
00019              V7 = I * V2
00020          ENDDO
00021
00022    D     print *, V1,V2,V3,V4,V5,V6,V7,I
00023
00024          END


OPTIMIZETEST1                                       15-Jul-1993 16:28:49      VAX  FORTRAN
V5.7-133                Page   2
01                                                                  15-Jul-1993 16:04:15
       US12$ROOT:[FROEHLICH]COMP1.FOR;2

                .TITLE  OPTIMIZETEST1
                .IDENT  01

    0000        .PSECT  $CODE

                                                        ; 00002
    0000  OPTIMIZETEST1::
    0000        .WORD   ^M<IV,R2,R3,R4,R5,R6>
                                                        ; 00014
    0002        MOVL    #1, R12
    0005        NOP
    0006        NOP
    0007        NOP
    0008  L$1:
                                                        ; 00015
```

```
    0008       MULL3   #2, R12, R2
                                                                ; 00016
    000C       MULL3   R2, R12, R3
                                                                ; 00017
    0010       MULL3   #10, R12, R4
                                                                ; 00018
    0014       MULL3   #10, R12, R5
                                                                ; 00019
    0018       MULL3   #20, R12, R6
                                                                ; 00020
    001C       AOBLEQ  #10, R12, L$1
                                                                ; 00022
    0020       MNEGL   #1, -(SP)
    0023       CALLS   #1, FOR$WRITE_SL
    002A       PUSHL   #10
    002C       CALLS   #1, FOR$IO_L_V
    0033       PUSHL   #20
    0035       CALLS   #1, FOR$IO_L_V
    003C       PUSHL   R2
    003E       CALLS   #1, FOR$IO_L_V
    0045       PUSHL   R3
    0047       CALLS   #1, FOR$IO_L_V
    004E       PUSHL   R4
    0050       CALLS   #1, FOR$IO_L_V
    0057       PUSHL   R5
    0059       CALLS   #1, FOR$IO_L_V
    0060       PUSHL   R6
    0062       CALLS   #1, FOR$IO_L_V
    0069       PUSHL   R12
    006B       CALLS   #1, FOR$IO_L_V
    0072       CALLS   #0, FOR$IO_END
                                                                ; 00024
    0079       MOVL    #1, R0
    007C       RET
               .END

  FOR/LIST/CHECK=ALL/EXTEND/NOANA/D_LINES/OPT/MACH COMP1
  Total Space Allocated                 125


COMPILATION STATISTICS

  Run Time:           0.08 seconds
  Elapsed Time:       0.38 seconds
  Page Faults:        609
  Dynamic Memory:     460 pages
```

12.2 CISC-Compiler-Beispiel mit Optimizer (keine Ausgabe)

```
                                                    15-Jul-1993 16:29:01      VAX  FORTRAN
V5.7-133                Page   1
                                                    15-Jul-1993                   16:04:15
       US12$ROOT:[FROEHLICH]COMP1.FOR;2

00001          OPTIONS /EXTEND_SOURCE
00002          PROGRAM OptimizeTest1
00003          IMPLICIT  NONE
00004
00005          INTEGER*4  I
00006          INTEGER*4  V1  /10/
00007          INTEGER*4  V2  /20/
00008          INTEGER*4  V3
00009          INTEGER*4  V4
00010          INTEGER*4  V5
00011          INTEGER*4  V6
00012          INTEGER*4  V7
00013
00014          DO I = 1,10
00015              V3 = I * 2
00016              V4 = I * V3
00017              V5 = I * 10
00018              V6 = I * V1
00019              V7 = I * V2
00020          ENDDO
```

```
00021
00022 C  D     print *, V1,V2,V3,V4,V5,V6,V7,I
00023
00024          END
OPTIMIZETEST1                                                  15-Jul-1993 16:29:01      VAX  FORTRAN
V5.7-133                Page   2
01                                                                                    15-Jul-1993 16:04:15
       US12$ROOT:[FROEHLICH]COMP1.FOR;2
                .TITLE  OPTIMIZETEST1
                .IDENT  01
    0000        .PSECT  $CODE
                                                                          ; 00002
    0000  OPTIMIZETEST1::
    0000        .WORD   ^M<IV>
                                                                      ; 00014
    0002  L$1:
                                                                      ; 00024
    0002        MOVL    #1, R0
    0005        RET
                .END


 FOR/LIST/CHECK=ALL/EXTEND/NOANA/OPT/MACH COMP1
  Total Space Allocated                   6

COMPILATION STATISTICS

  Run Time:           0.07 seconds
  Elapsed Time:       0.36 seconds
  Page Faults:        564
  Dynamic Memory:     460 pages
```

12.3 CISC-Compiler-Beispiel mit Stringverarbeitung und Loopotimierung

```
                                                               15-Jul-1993 15:38:07      VAX  FORTRAN
V5.7-133                Page   1
                                                               15-Jul-1993                  15:37:37
       US12$ROOT:[FROEHLICH]COMP.FOR;1

00001          OPTIONS /EXTEND_SOURCE
00002          PROGRAM OptimizeTest
00003          IMPLICIT  NONE
00004
00005          INTEGER*4  I
00006          INTEGER*4  V1
00007          INTEGER*4  V2
00008          INTEGER*4  V3
00009          INTEGER*4  V4
00010
00011          CHARACTER   * 20    Text1   /'Dies ist Text 1'/
00012          CHARACTER   * 20    Text2
00013
00014
00015          DO I = 1,10
00016              V1 = 10
00017              V2 = 10 * V1
00018              V3 = I * V4
00019          ENDDO
00020
00021    D     print *, V1,V2,V3,I
00022
00023          Text2 = Text1
00024
00025    D     print *, Text1, Text2
00026          END

OPTIMIZETEST                                                   15-Jul-1993 15:38:07      VAX  FORTRAN
V5.7-133                Page   2
01                                                                                    15-Jul-1993 15:37:37
       US12$ROOT:[FROEHLICH]COMP.FOR;1
```

```
                .TITLE  OPTIMIZETEST
                .IDENT  01

    002C        .PSECT  $LOCAL
    002C        .LONG   ^X010E0014
    0030        .ADDR   TEXT1
    0034        .LONG   ^X010E0014
    0038        .ADDR   TEXT2
    0000  TEXT1:
    0000        .XBYTE  44,69,65,73,20,69,73,74,20,54,65,78,74,20,31
    000F        .XBYTE  20 [5]
    0000        .PSECT  $CODE
                                                                    ; 00002
    0000  OPTIMIZETEST::
    0000        .WORD   ^M<IV,R2,R3,R4,R5,R11>
    0002        MOVAL   $LOCAL+^X28, R11
    0009        MOVL    V4(R11), R0
                                                                    ; 00015
    000C        MOVL    #1, R12
    000F        NOP
    0010  L$1:
                                                                    ; 00018
    0010        MULL3   R0, R12, R2
                                                                    ; 00019
    0014        AOBLEQ  #10, R12, L$1
                                                                    ; 00021
    0018        MNEGL   #1, -(SP)
    001B        CALLS   #1, FOR$WRITE_SL
    0022        PUSHL   #10
    0024        CALLS   #1, FOR$IO_L_V
    002B        MOVZBL  #100, -(SP)
    002F        CALLS   #1, FOR$IO_L_V
    0036        PUSHL   R2
    0038        CALLS   #1, FOR$IO_L_V
    003F        PUSHL   R12
    0041        CALLS   #1, FOR$IO_L_V
    0048        CALLS   #0, FOR$IO_END
                                                                    ; 00023
    004F        MOVC3   #20, TEXT1(R11), TEXT2(R11)
                                                                    ; 00025
    0055        MNEGL   #1, -(SP)
    0058        CALLS   #1, FOR$WRITE_SL
    005F        PUSHAB  $LOCAL+^X2C(R11)
    0062        CALLS   #1, FOR$IO_T_DS
    0069        PUSHAB  $LOCAL+^X34(R11)
    006C        CALLS   #1, FOR$IO_T_DS
    0073        CALLS   #0, FOR$IO_END
                                                                    ; 00026
    007A        MOVL    #1, R0
    007D        RET
                .END

  FOR/LIST/CHECK=ALL/EXTEND/NOANA/D_LINES/OPT/MACH COMP
  Total Space Allocated                 186

COMPILATION STATISTICS

  Run Time:           0.09 seconds
  Elapsed Time:       0.55 seconds
  Page Faults:        627
  Dynamic Memory:     460 pages
```

12.4 RISC-Compiler-Beispiel ohne Optimizer

```
OPTIMIZETEST1                                          15-JUL-1993 16:55:28    DEC Fortran
T6.1-333-2622           Page   1
                                                      15-JUL-1993                16:04:15

CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

             1        OPTIONS /EXTEND_SOURCE
             2        PROGRAM OptimizeTest1
             3        IMPLICIT  NONE
             4
             5        INTEGER*4  I
             6        INTEGER*4  V1  /10/
```

```
             7        INTEGER*4  V2  /20/
             8        INTEGER*4  V3
             9        INTEGER*4  V4
            10        INTEGER*4  V5
            11        INTEGER*4  V6
            12        INTEGER*4  V7
            13
            14        DO I = 1,10
            15            V3 = I * 2
            16            V4 = I * V3
            17            V5 = I * 10
            18            V6 = I * V1
            19            V7 = I * V2
            20        ENDDO
            21
            22 D      print *, V1,V2,V3,V4,V5,V6,V7,I
            23
            24        END
```

Total Space Allocated 704

```
OPTIMIZETEST1                    Machine Code Listing            15-JUL-1993 16:55:28    DEC Fortran
T6.1-333-2622           Page   2
                         OPTIMIZETEST1$BLK                                 15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

                             .PSECT  $CODE$, OCTA, PIC, CON, REL, LCL, SHR,-
                                     EXE, NORD, NOWRT
                0000      OPTIMIZETEST1::
                              ; 000002
23DEFFA0        0000              LDA     SP, -96(SP)                     ; SP, -96(SP)
B7FE0008        0004              STQ     R31, 8(SP)                      ; R31, 8(SP)
B77E0000        0008              STQ     R27, (SP)                       ; R27, (SP)
B75E0040        000C              STQ     R26, 64(SP)                     ; R26, 64(SP)
B45E0048        0010              STQ     R2, 72(SP)                      ; R2, 72(SP)
B7BE0050        0014              STQ     FP, 80(SP)                      ; FP, 80(SP)
60000000        0018              TRAPB                                   ;
47FE041D        001C              MOV     SP, FP                          ; SP, FP
47FB0402        0020              MOV     R27, R2                         ; R27, R2
47E03400        0024              MOV     1, R0                           ; 1, R0
                              ; 000014
A4220030        0028              LDQ     R1, 48(R2)                      ; R1, 48(R2)
B0010000        002C              STL     R0, I                           ; R0, (R1)
                0030      L$1:
A6020030        0030              LDQ     R16, 48(R2)                     ; R16, 48(R2)
                              ; 000015
A2100000        0034              LDL     R16, I                          ; R16, (R16)
4E005010        0038              MULL    R16, 2, R16                     ; R16, 2, R16
A4020030        003C              LDQ     R0, 48(R2)                      ; R0, 48(R2)
B2000004        0040              STL     R16, V3                         ; R16, 4(R0)
A4220030        0044              LDQ     R1, 48(R2)                      ; R1, 48(R2)
                              ; 000016
A0210000        0048              LDL     R1, I                           ; R1, (R1)
A4020030        004C              LDQ     R0, 48(R2)                      ; R0, 48(R2)
A0000004        0050              LDL     R0, V3                          ; R0, 4(R0)
4C200001        0054              MULL    R1, R0, R1                      ; R1, R0, R1
A6020030        0058              LDQ     R16, 48(R2)                     ; R16, 48(R2)
B0300008        005C              STL     R1, V4                          ; R1, 8(R16)
A4020030        0060              LDQ     R0, 48(R2)                      ; R0, 48(R2)
                              ; 000017
A0000000        0064              LDL     R0, I                           ; R0, (R0)
4C015000        0068              MULL    R0, 10, R0                      ; R0, 10, R0
A4220030        006C              LDQ     R1, 48(R2)                      ; R1, 48(R2)
B001000C        0070              STL     R0, V5                          ; R0, 12(R1)
A6020030        0074              LDQ     R16, 48(R2)                     ; R16, 48(R2)
                              ; 000018
A2100000        0078              LDL     R16, I                          ; R16, (R16)
A4020050        007C              LDQ     R0, 80(R2)                      ; R0, 80(R2)
A0000000        0080              LDL     R0, V1                          ; R0, (R0)
4E000010        0084              MULL    R16, R0, R16                    ; R16, R0, R16
A4220030        0088              LDQ     R1, 48(R2)                      ; R1, 48(R2)
B2010010        008C              STL     R16, V6                         ; R16, 16(R1)
A4020030        0090              LDQ     R0, 48(R2)                      ; R0, 48(R2)
                              ; 000019
A0000000        0094              LDL     R0, I                           ; R0, (R0)
A4220050        0098              LDQ     R1, 80(R2)                      ; R1, 80(R2)
```

```
A0210004     009C            LDL     R1, V2                          ; R1, 4(R1)
4C010001     00A0            MULL    R0, R1, R1                      ; R0, R1, R1
A6020030     00A4            LDQ     R16, 48(R2)                     ; R16, 48(R2)
B0300014     00A8            STL     R1, V7                          ; R1, 20(R16)
A4020030     00AC            LDQ     R0, 48(R2)                      ; R0, 48(R2)
                  ; 000014
A0000000     00B0            LDL     R0, I                           ; R0, (R0)
40003000     00B4            ADDL    R0, 1, R0                       ; R0, 1, R0
A4220030     00B8            LDQ     R1, 48(R2)                      ; R1, 48(R2)
B0010000     00BC            STL     R0, I                           ; R0, (R1)
A6020030     00C0            LDQ     R16, 48(R2)                     ; R16, 48(R2)
A2100000     00C4            LDL     R16, I                          ; R16, (R16)
2210FFF5     00C8            ADDQ    R16, -11, R16                   ; R16, -11, R16
EA1FFFD8     00CC            BLT     R16, L$1                        ; R16, L$1
22620038     00D0            LDA     R19, 56(R2)                     ; R19, 56(R2)
                  ; 000022


OPTIMIZETEST1                    Machine Code Listing            15-JUL-1993 16:55:28    DEC Fortran
T6.1-333-2622           Page   3
                          OPTIMIZETEST1$BLK                              15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

A4020050     00D4            LDQ     R0, 80(R2)                      ; R0, 80(R2)
B41D0038     00D8            STQ     R0, 56(FP)                      ; R0, 56(FP)
221D0008     00DC            LDA     R16, 8(FP)                      ; R16, 8(FP)
223FFFFF     00E0            MOV     -1, R17                         ; -1, R17
225F0180     00E4            MOV     384, R18                        ; 384, R18
229D0038     00E8            LDA     R20, 56(FP)                     ; R20, 56(FP)
47E0B419     00EC            MOV     5, R25                          ; 5, R25
A7420060     00F0            LDQ     R26, 96(R2)                     ; R26, 96(R2)
A7620068     00F4            LDQ     R27, 104(R2)                    ; R27, 104(R2)
6B5A4000     00F8            JSR     R26, DFOR$WRITE_SEQ_LIS                 ; R26, R26
22220038     00FC            LDA     R17, 56(R2)                     ; R17, 56(R2)
A4020050     0100            LDQ     R0, 80(R2)                      ; R0, 80(R2)
20000004     0104            LDA     R0, V2                          ; R0, 4(R0)
B41D0038     0108            STQ     R0, 56(FP)                      ; R0, 56(FP)
221D0008     010C            LDA     R16, 8(FP)                      ; R16, 8(FP)
225D0038     0110            LDA     R18, 56(FP)                     ; R18, 56(FP)
47E07419     0114            MOV     3, R25                          ; 3, R25
A7420040     0118            LDQ     R26, 64(R2)                     ; R26, 64(R2)
A7620048     011C            LDQ     R27, 72(R2)                     ; R27, 72(R2)
6B5A4000     0120            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT    ; R26, R26
22220038     0124            LDA     R17, 56(R2)                     ; R17, 56(R2)
A4020030     0128            LDQ     R0, 48(R2)                      ; R0, 48(R2)
20000004     012C            LDA     R0, V3                          ; R0, 4(R0)
B41D0038     0130            STQ     R0, 56(FP)                      ; R0, 56(FP)
221D0008     0134            LDA     R16, 8(FP)                      ; R16, 8(FP)
225D0038     0138            LDA     R18, 56(FP)                     ; R18, 56(FP)
47E07419     013C            MOV     3, R25                          ; 3, R25
A7420040     0140            LDQ     R26, 64(R2)                     ; R26, 64(R2)
A7620048     0144            LDQ     R27, 72(R2)                     ; R27, 72(R2)
6B5A4000     0148            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT    ; R26, R26
22220038     014C            LDA     R17, 56(R2)                     ; R17, 56(R2)
A4020030     0150            LDQ     R0, 48(R2)                      ; R0, 48(R2)
20000008     0154            LDA     R0, V4                          ; R0, 8(R0)
B41D0038     0158            STQ     R0, 56(FP)                      ; R0, 56(FP)
221D0008     015C            LDA     R16, 8(FP)                      ; R16, 8(FP)
225D0038     0160            LDA     R18, 56(FP)                     ; R18, 56(FP)
47E07419     0164            MOV     3, R25                          ; 3, R25
A7420040     0168            LDQ     R26, 64(R2)                     ; R26, 64(R2)
A7620048     016C            LDQ     R27, 72(R2)                     ; R27, 72(R2)
6B5A4000     0170            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT    ; R26, R26
22220038     0174            LDA     R17, 56(R2)                     ; R17, 56(R2)
A4020030     0178            LDQ     R0, 48(R2)                      ; R0, 48(R2)
2000000C     017C            LDA     R0, V5                          ; R0, 12(R0)
B41D0038     0180            STQ     R0, 56(FP)                      ; R0, 56(FP)
221D0008     0184            LDA     R16, 8(FP)                      ; R16, 8(FP)
225D0038     0188            LDA     R18, 56(FP)                     ; R18, 56(FP)
47E07419     018C            MOV     3, R25                          ; 3, R25
A7420040     0190            LDQ     R26, 64(R2)                     ; R26, 64(R2)
A7620048     0194            LDQ     R27, 72(R2)                     ; R27, 72(R2)
6B5A4000     0198            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT    ; R26, R26
22220038     019C            LDA     R17, 56(R2)                     ; R17, 56(R2)
A4020030     01A0            LDQ     R0, 48(R2)                      ; R0, 48(R2)
20000010     01A4            LDA     R0, V6                          ; R0, 16(R0)
B41D0038     01A8            STQ     R0, 56(FP)                      ; R0, 56(FP)
```

```
221D0008     01AC             LDA     R16, 8(FP)                          ; R16, 8(FP)
225D0038     01B0             LDA     R18, 56(FP)                         ; R18, 56(FP)
47E07419     01B4             MOV     3, R25                              ; 3, R25


OPTIMIZETEST1                    Machine Code Listing            15-JUL-1993 16:55:28    DEC Fortran
T6.1-333-2622           Page   4
                          OPTIMIZETEST1$BLK                                 15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

A7420040     01B8             LDQ     R26, 64(R2)                         ; R26, 64(R2)
A7620048     01BC             LDQ     R27, 72(R2)                         ; R27, 72(R2)
6B5A4000     01C0             JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
22220038     01C4             LDA     R17, 56(R2)                         ; R17, 56(R2)
A4020030     01C8             LDQ     R0, 48(R2)                          ; R0, 48(R2)
20000014     01CC             LDA     R0, V7                              ; R0, 20(R0)
B41D0038     01D0             STQ     R0, 56(FP)                          ; R0, 56(FP)
221D0008     01D4             LDA     R16, 8(FP)                          ; R16, 8(FP)
225D0038     01D8             LDA     R18, 56(FP)                         ; R18, 56(FP)
47E07419     01DC             MOV     3, R25                              ; 3, R25
A7420040     01E0             LDQ     R26, 64(R2)                         ; R26, 64(R2)
A7620048     01E4             LDQ     R27, 72(R2)                         ; R27, 72(R2)
6B5A4000     01E8             JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
22220058     01EC             LDA     R17, 88(R2)                         ; R17, 88(R2)
A4020030     01F0             LDQ     R0, 48(R2)                          ; R0, 48(R2)
B41D0038     01F4             STQ     R0, 56(FP)                          ; R0, 56(FP)
221D0008     01F8             LDA     R16, 8(FP)                          ; R16, 8(FP)
225D0038     01FC             LDA     R18, 56(FP)                         ; R18, 56(FP)
47E07419     0200             MOV     3, R25                              ; 3, R25
A7420040     0204             LDQ     R26, 64(R2)                         ; R26, 64(R2)
A7620048     0208             LDQ     R27, 72(R2)                         ; R27, 72(R2)
6B5A4000     020C             JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E03400     0210             MOV     1, R0                               ; 1, R0
                ; 000024
60000000     0214             TRAPB                                       ;
47FD041E     0218             MOV     FP, SP                              ; FP, SP
A75D0040     021C             LDQ     R26, 64(FP)                         ; R26, 64(FP)
A45D0048     0220             LDQ     R2, 72(FP)                          ; R2, 72(FP)
A7BD0050     0224             LDQ     FP, 80(FP)                          ; FP, 80(FP)
23DE0060     0228             LDA     SP, 96(SP)                          ; SP, 96(SP)
6BFA8001     022C             RET     R26                                 ; R26

Routine Size: 560 bytes,    Routine Base: $CODE$ + 0000


                              .PSECT  $LINK$, OCTA, NOPIC, CON, REL, LCL,-
                                      NOSHR, NOEXE, RD, NOWRT

             0000             ; Heavyweight Frame invocation descriptor
                              Entry point:          OPTIMIZETEST1
                              Static Handler:        DFOR$HANDLER
                              Registers used:        R0-R2, R16-R27, FP, F0-F1, F10-F30
                              Registers saved:       R2, FP
                              Fixed Stack Size:      96
00000000                      ; Handler data for DFOR$HANDLER
00000000
00000000
00000000

00000000     0030             .ADDRESS  $BSS$
00020309     0038             .ASCII  <9><3><2><0>
             0040             .LINKAGE  DFOR$WRITE_SEQ_LIS_XMIT
00000000     0050             .ADDRESS  $DATA$
00010309     0058             .ASCII  <9><3><1><0>
             0060             .LINKAGE  DFOR$WRITE_SEQ_LIS

                              .PSECT  $DATA$, OCTA, NOPIC, CON, REL, LCL,-
                                      NOSHR, NOEXE, RD, WRT


OPTIMIZETEST1                    Machine Code Listing            15-JUL-1993 16:55:28    DEC Fortran
T6.1-333-2622           Page   5
                          OPTIMIZETEST1$BLK                                 15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

             0000      V1:
0000000A     0000             .LONG   10
```

```
            0004      V2:
00000014     0004              .LONG   20

COMMAND QUALIFIERS

  /ALIGN=(COMMONS=PACKED,RECORDS=NATURAL)
  /ASSUME=(ACCURACY_SENSITIVE,BACKSLASH,NODUMMY_ALIASES,NOUNDERSCORE)
  /CHECK=(NOASSERTIONS,NOBOUNDS,NOOVERFLOW,NOUNDERFLOW)
  /DEBUG=(NOSYMBOLS,TRACEBACK)
  /SHOW=(NODICTIONARY,NOINCLUDE,MAP,NOPREPROCESSOR)
  /STANDARD=(NOSEMANTIC,NOSOURCE_FORM,NOSYNTAX)
  /WARNINGS=(ALIGNMENT,NOARGUMENT_CHECKING,NODECLARATIONS,GENERAL,NOTRUNCATED_SOURCE,UNCALLED,
             UNINITIALIZED,UNREACHABLE,UNUSED)
  /NOAUTOMATIC    /BLAS=NOMAPPED    /CONVERT=VAXG    /NOCROSS_REFERENCE    /D_LINES    /ERROR_LIMIT=30
/NOEXTEND_SOURCE
  /F77        /FLOAT=G_FLOAT        /GRANULARITY=QUADWORD        /INTEGER_SIZE=32        /MACHINE_CODE
/MATH_LIBRARY=ACCURATE  /NAMES=UPPERCASE
  /OPTIMIZE=LEVEL=0  /REAL_SIZE=32  /NORECURSIVE
  /NOSEPARATE_COMPILATION  /TERMINAL=NOSTATISTICS  /NOTIE  /VMS
  /NOANALYSIS_DATA
  /NODIAGNOSTICS
  /LIST=CLARA::US12$ROOT:[FROEHLICH]COMP1.LIS;2
  /OBJECT=CLARA::US12$ROOT:[FROEHLICH]COMP1.OBJ;5
  /NOLIBRARY
   sys$lib=

COMPILATION STATISTICS

  CPU time:          0.45 seconds
  Elapsed time:      9.96 seconds
  Pagefaults:         260
  I/O Count:          598
```

12.5 RISC-Compiler-Beispiel mit Optimizer

```
OPTIMIZETEST1                                                  15-JUL-1993 16:50:02    DEC Fortran
T6.1-333-2622           Page   1
                                                              15-JUL-1993                  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

             1        OPTIONS /EXTEND_SOURCE
             2        PROGRAM OptimizeTest1
             3        IMPLICIT  NONE
             4
             5        INTEGER*4  I
             6        INTEGER*4  V1  /10/
             7        INTEGER*4  V2  /20/
             8        INTEGER*4  V3
             9        INTEGER*4  V4
            10        INTEGER*4  V5
            11        INTEGER*4  V6
            12        INTEGER*4  V7
            13
            14        DO I = 1,10
            15            V3 = I * 2
            16            V4 = I * V3
            17            V5 = I * 10
            18            V6 = I * V1
            19            V7 = I * V2
            20        ENDDO
            21
            22 D      print *, V1,V2,V3,V4,V5,V6,V7,I
            23
            24        END


PROGRAM SECTIONS

    Name                            Bytes   Attributes

  1 $LINK$                            112 NOPIC CON REL LCL NOSHR NOEXE   RD NOWRT OCTA
  2 $CODE$                            524   PIC CON REL LCL   SHR   EXE NORD NOWRT OCTA
  3 $DATA$                              8 NOPIC CON REL LCL NOSHR NOEXE   RD   WRT OCTA
```

Total Space Allocated 644


```
                        .PSECT  $CODE$, OCTA, PIC, CON, REL, LCL, SHR,-
                                EXE, NORD, NOWRT
            0000      OPTIMIZETEST1::
                          ; 000002
23DEFF60    0000              LDA     SP, -160(SP)                    ; SP, -160(SP)
47E29400    0004              MOV     20, R0                          ; 20, R0
                  ; 000015
B7FE0008    0008              STQ     R31, 8(SP)                      ; R31, 8(SP)
                  ; 000002
47E15401    000C              MOV     10, R1                          ; 10, R1
                  ; 000015
B77E0000    0010              STQ     R27, (SP)                       ; R27, (SP)
                  ; 000002
47E15410    0014              MOV     10, R16                         ; 10, R16
                  ; 000015
B75E0058    0018              STQ     R26, 88(SP)                     ; R26, 88(SP)
                  ; 000002
47E05411    001C              MOV     2, R17                          ; 2, R17
                  ; 000015
B45E0060    0020              STQ     R2, 96(SP)                      ; R2, 96(SP)
                  ; 000002
47E29412    0024              MOV     20, R18                         ; 20, R18
B47E0068    0028              STQ     R3, 104(SP)                     ; R3, 104(SP)
B49E0070    002C              STQ     R4, 112(SP)                     ; R4, 112(SP)
B4BE0078    0030              STQ     R5, 120(SP)                     ; R5, 120(SP)
B4DE0080    0034              STQ     R6, 128(SP)                     ; R6, 128(SP)
B4FE0088    0038              STQ     R7, 136(SP)                     ; R7, 136(SP)
B51E0090    003C              STQ     R8, 144(SP)                     ; R8, 144(SP)
B7BE0098    0040              STQ     FP, 152(SP)                     ; FP, 152(SP)
60000000    0044              TRAPB                                   ;
47FE041D    0048              MOV     SP, FP                          ; SP, FP
47FB0402    004C              MOV     R27, R2                         ; R27, R2
47E03403    0050              MOV     1, I                            ; 1, R3
                  ; 000014
5FFF041F    0054              FNOP                                    ;
            0058      L$1:
42211004    0058              ADDL    R17, 8, V3                      ; R17, 8, R4
A6620050    005C              LDQ     R19, 80(R2)                     ; R19, 80(R2)
                  ; 000022
40609012    0060              ADDL    I, 4, R18                       ; R3, 4, R18
                  ; 000016
4E440005    0064              MULL    R18, V3, V4                     ; R18, R4, R5
42215011    0068              ADDL    R17, 10, R17                    ; R17, 10, R17
                  ; 000014
42051006    006C              ADDL    R16, 40, V5                     ; R16, 40, R6
40251007    0070              ADDL    R1, 40, V6                      ; R1, 40, R7
400A1008    0074              ADDL    R0, 80, V7                      ; R0, 80, R8
2251FFEB    0078              ADDQ    R17, -21, R18                   ; R17, -21, R18
4060B003    007C              ADDL    I, 5, I                         ; R3, 5, R3
400C9000    0080              ADDL    R0, 100, R0                     ; R0, 100, R0
40265001    0084              ADDL    R1, 50, R1                      ; R1, 50, R1
42065010    0088              ADDL    R16, 50, R16                    ; R16, 50, R16
EA5FFFF2    008C              BLT     R18, L$1                        ; R18, L$1
A7420060    0090              LDQ     R26, 96(R2)                     ; R26, 96(R2)
                  ; 000022
47F30414    0094              MOV     V1, R20                         ; R19, R20
B69D0050    0098              STQ     R20, 80(FP)                     ; R20, 80(FP)
221D0008    009C              LDA     R16, 8(FP)                      ; R16, 8(FP)
A7620068    00A0              LDQ     R27, 104(R2)                    ; R27, 104(R2)
223FFFFF    00A4              MOV     -1, R17                         ; -1, R17
225F0180    00A8              MOV     384, R18                        ; 384, R18
22620030    00AC              LDA     R19, 48(R2)                     ; R19, 48(R2)
229D0050    00B0              LDA     R20, 80(FP)                     ; R20, 80(FP)
47E0B419    00B4              MOV     5, R25                          ; 5, R25
6B5A4000    00B8              JSR     R26, DFOR$WRITE_SEQ_LIS                 ; R26, R26
47E07419    00BC              MOV     3, R25                          ; 3, R25
A4020050    00C0              LDQ     R0, 80(R2)                      ; R0, 80(R2)
221D0008    00C4              LDA     R16, 8(FP)                      ; R16, 8(FP)
A7420040    00C8              LDQ     R26, 64(R2)                     ; R26, 64(R2)
22220030    00CC              LDA     R17, 48(R2)                     ; R17, 48(R2)
```

```
A7620048     00D0            LDQ     R27, 72(R2)                         ; R27, 72(R2)


OPTIMIZETEST1                   Machine Code Listing            15-JUL-1993 16:50:02    DEC Fortran
T6.1-333-2622           Page   3
                                OPTIMIZETEST1$BLK                               15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

225D0050     00D4            LDA     R18, 80(FP)                         ; R18, 80(FP)
20000004     00D8            LDA     R0, V2                              ; R0, 4(R0)
B41D0050     00DC            STQ     R0, 80(FP)                          ; R0, 80(FP)
6B5A4000     00E0            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E07419     00E4            MOV     3, R25                              ; 3, R25
A7420040     00E8            LDQ     R26, 64(R2)                         ; R26, 64(R2)
201D0048     00EC            LDA     R0, V3                              ; R0, 72(FP)
B41D0050     00F0            STQ     R0, 80(FP)                          ; R0, 80(FP)
221D0008     00F4            LDA     R16, 8(FP)                          ; R16, 8(FP)
A7620048     00F8            LDQ     R27, 72(R2)                         ; R27, 72(R2)
22220030     00FC            LDA     R17, 48(R2)                         ; R17, 48(R2)
B09D0048     0100            STL     V3, V3                              ; R4, 72(FP)
225D0050     0104            LDA     R18, 80(FP)                         ; R18, 80(FP)
6B5A4000     0108            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E07419     010C            MOV     3, R25                              ; 3, R25
A7420040     0110            LDQ     R26, 64(R2)                         ; R26, 64(R2)
209D0044     0114            LDA     R4, V4                              ; R4, 68(FP)
B49D0050     0118            STQ     R4, 80(FP)                          ; R4, 80(FP)
221D0008     011C            LDA     R16, 8(FP)                          ; R16, 8(FP)
A7620048     0120            LDQ     R27, 72(R2)                         ; R27, 72(R2)
22220030     0124            LDA     R17, 48(R2)                         ; R17, 48(R2)
B0BD0044     0128            STL     V4, V4                              ; R5, 68(FP)
225D0050     012C            LDA     R18, 80(FP)                         ; R18, 80(FP)
6B5A4000     0130            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E07419     0134            MOV     3, R25                              ; 3, R25
A7420040     0138            LDQ     R26, 64(R2)                         ; R26, 64(R2)
209D0040     013C            LDA     R4, V5                              ; R4, 64(FP)
B49D0050     0140            STQ     R4, 80(FP)                          ; R4, 80(FP)
221D0008     0144            LDA     R16, 8(FP)                          ; R16, 8(FP)
A7620048     0148            LDQ     R27, 72(R2)                         ; R27, 72(R2)
22220030     014C            LDA     R17, 48(R2)                         ; R17, 48(R2)
B0DD0040     0150            STL     V5, V5                              ; R6, 64(FP)
225D0050     0154            LDA     R18, 80(FP)                         ; R18, 80(FP)
6B5A4000     0158            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E07419     015C            MOV     3, R25                              ; 3, R25
A7420040     0160            LDQ     R26, 64(R2)                         ; R26, 64(R2)
209D003C     0164            LDA     R4, V6                              ; R4, 60(FP)
B49D0050     0168            STQ     R4, 80(FP)                          ; R4, 80(FP)
221D0008     016C            LDA     R16, 8(FP)                          ; R16, 8(FP)
A7620048     0170            LDQ     R27, 72(R2)                         ; R27, 72(R2)
22220030     0174            LDA     R17, 48(R2)                         ; R17, 48(R2)
B0FD003C     0178            STL     V6, V6                              ; R7, 60(FP)
225D0050     017C            LDA     R18, 80(FP)                         ; R18, 80(FP)
6B5A4000     0180            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E07419     0184            MOV     3, R25                              ; 3, R25
A7420040     0188            LDQ     R26, 64(R2)                         ; R26, 64(R2)
20DD0038     018C            LDA     R6, V7                              ; R6, 56(FP)
B4DD0050     0190            STQ     R6, 80(FP)                          ; R6, 80(FP)
221D0008     0194            LDA     R16, 8(FP)                          ; R16, 8(FP)
A7620048     0198            LDQ     R27, 72(R2)                         ; R27, 72(R2)
22220030     019C            LDA     R17, 48(R2)                         ; R17, 48(R2)
B11D0038     01A0            STL     V7, V7                              ; R8, 56(FP)
225D0050     01A4            LDA     R18, 80(FP)                         ; R18, 80(FP)
6B5A4000     01A8            JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT        ; R26, R26
47E07419     01AC            MOV     3, R25                              ; 3, R25
A7420040     01B0            LDQ     R26, 64(R2)                         ; R26, 64(R2)
20BD004C     01B4            LDA     R5, I                               ; R5, 76(FP)


OPTIMIZETEST1                   Machine Code Listing            15-JUL-1993 16:50:02    DEC Fortran
T6.1-333-2622           Page   4
                                OPTIMIZETEST1$BLK                               15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

B4BD0050     01B8            STQ     R5, 80(FP)                          ; R5, 80(FP)
221D0008     01BC            LDA     R16, 8(FP)                          ; R16, 8(FP)
A7620048     01C0            LDQ     R27, 72(R2)                         ; R27, 72(R2)
22220058     01C4            LDA     R17, 88(R2)                         ; R17, 88(R2)
B07D004C     01C8            STL     I, I                                ; R3, 76(FP)
```

```
225D0050     01CC             LDA     R18, 80(FP)                            ; R18, 80(FP)
6B5A4000     01D0             JSR     R26, DFOR$WRITE_SEQ_LIS_XMIT           ; R26, R26
47E03400     01D4             MOV     1, R0                                  ; 1, R0
                         ; 000024
60000000     01D8             TRAPB                                          ;
47FD041E     01DC             MOV     FP, SP                                 ; FP, SP
A75D0058     01E0             LDQ     R26, 88(FP)                            ; R26, 88(FP)
A45D0060     01E4             LDQ     R2, 96(FP)                             ; R2, 96(FP)
A47D0068     01E8             LDQ     R3, 104(FP)                            ; R3, 104(FP)
A49D0070     01EC             LDQ     R4, 112(FP)                            ; R4, 112(FP)
A4BD0078     01F0             LDQ     R5, 120(FP)                            ; R5, 120(FP)
A4DD0080     01F4             LDQ     R6, 128(FP)                            ; R6, 128(FP)
A4FD0088     01F8             LDQ     R7, 136(FP)                            ; R7, 136(FP)
A51D0090     01FC             LDQ     R8, 144(FP)                            ; R8, 144(FP)
A7BD0098     0200             LDQ     FP, 152(FP)                            ; FP, 152(FP)
23DE00A0     0204             LDA     SP, 160(SP)                            ; SP, 160(SP)
6BFA8001     0208             RET     R26                                    ; R26

Routine Size: 524 bytes,    Routine Base: $CODE$ + 0000
                              .PSECT  $LINK$, OCTA, NOPIC, CON, REL, LCL,-
                                      NOSHR, NOEXE, RD, NOWRT

             0000             ; Heavyweight Frame invocation descriptor
                              Entry point:          OPTIMIZETEST1
                              Static Handler:        DFOR$HANDLER
                              Registers used:        R0-R8, R16-R27, FP, F0-F1, F10-F30
                              Registers saved:       R2-R8, FP
                              Fixed Stack Size:      160
00000000                      ; Handler data for DFOR$HANDLER

OPTIMIZETEST1                                                  15-JUL-1993 16:50:02    DEC Fortran
T6.1-333-2622           Page   5
                                 Generated  Code                                 15-JUL-1993  16:04:15
CLARA::US12$ROOT:[FROEHLICH]COMP1.FOR;2

COMMAND QUALIFIERS

  /ALIGN=(COMMONS=PACKED,RECORDS=NATURAL)
  /ASSUME=(ACCURACY_SENSITIVE,BACKSLASH,NODUMMY_ALIASES,NOUNDERSCORE)
  /CHECK=(NOASSERTIONS,NOBOUNDS,NOOVERFLOW,NOUNDERFLOW)
  /DEBUG=(NOSYMBOLS,TRACEBACK)
  /SHOW=(NODICTIONARY,NOINCLUDE,MAP,NOPREPROCESSOR)
  /STANDARD=(NOSEMANTIC,NOSOURCE_FORM,NOSYNTAX)
  /WARNINGS=(ALIGNMENT,NOARGUMENT_CHECKING,NODECLARATIONS,GENERAL,NOTRUNCATED_SOURCE,UNCALLED,
             UNINITIALIZED,UNREACHABLE,UNUSED)
  /NOAUTOMATIC    /BLAS=NOMAPPED    /CONVERT=VAXG    /NOCROSS_REFERENCE    /D_LINES    /ERROR_LIMIT=30
/NOEXTEND_SOURCE
  /F77        /FLOAT=G_FLOAT        /GRANULARITY=QUADWORD        /INTEGER_SIZE=32        /MACHINE_CODE
/MATH_LIBRARY=ACCURATE  /NAMES=UPPERCASE
  /OPTIMIZE=LEVEL=4  /REAL_SIZE=32  /NORECURSIVE
  /NOSEPARATE_COMPILATION  /TERMINAL=NOSTATISTICS  /NOTIE  /VMS
  /NOANALYSIS_DATA
  /NODIAGNOSTICS
  /LIST=CLARA::US12$ROOT:[FROEHLICH]COMP1.LIS;1
  /OBJECT=CLARA::US12$ROOT:[FROEHLICH]COMP1.OBJ;3
  /NOLIBRARY
   sys$lib=

COMPILATION STATISTICS

  CPU time:          0.50 seconds
  Elapsed time:      8.58 seconds
  Pagefaults:         441
  I/O Count:          576
```

1