Reduzierung des Befehlsdatenstromes für RISC-Prozessoren eine Übersicht über das Problemfeld

Günter Kemnitz Gert Markwardt Sergej Sawitzki Patrick Schulz

Kurzfassung

Ein unvermeidbarer Engpaß für die Verarbeitungsgeschindigkeit eines Von-Neumann-Rechners ist die Übertragungsgeschwindigkeit der Daten und Befehle zwischen Hauptspeicher und Prozessor. Das Ziel, den Befehlsdatenstrom eines RISC-Prozessors zu reduzieren, führt zu interssanten Ansätzen für die Optimierung der Prozessorarchitektur insgesamt. In der Studie werden diese Zusammenhänge systematisiert und qualitativ bewertet.

1 Einführung

Die Studie faßt die Ergebnisse des DFG-Forschungsprojektes "RISC-Prozessoren mit reduziertem Befehlsdatenstrom" für die erste Bearbeitungsperiode zusammen. Neben den Ergebnissen einiger selektiver Untersuchungen enthält der Bericht eine systematische Darstellung des aktuellen Entwicklungsstandes der RISC-Prozessoren, eine Einordnung unserer Forschung in die weitere Prozessorentwicklung und die geplanten weiteren Untersuchungen.

Der Begriff RISC-Prozessor (RISC - reduced instruction set computer) gehört seit langem zum Vokabular der Informatik. Es gibt jedoch keine genaue Definition, nach welchen Merkmalen ein Prozessor zur Klasse der RISC-Prozessoren zählt. Die Abgrenzung von seinem Gegenstück, dem CISC-Prozessor (complex instruction set computer) nach dem Befehlsvorrat ist praktisch veraltet, da der Befehlsumfang der RISC-Prozessoren immer größer wird und sich mengenmäßig nicht mehr wesentlich von dem eines CISC-Prozessors unterscheidet. Was ist nun ein RISC-Prozessor? Diese Frage ist nur evolutionär zu beantworten. Der folgende Abschnitt beschäftigt sich mit dem ursprünglichen RISC-Konzept, insbesondere seinem Einfluß auf den Befehlsdatenstrom.

Im Flusse der Entwicklung mehrerer Generationen von RISC-Prozessoren hat sich die Architektur, die sich hinter diesem Namen verbirgt, weiterentwickelt. Einige der ursprünglichen Merkmale wie der relativ kleine Befehlssatz sind nicht mehr typisch. Andere, neuere Merkmale wie einen schnellen Cache auf dem Chip weisen fast alle Neuentwicklungen auf. Der 3. Abschnitt systematisiert diese Entwicklungstrends unter dem Blickwinkel von Hardware-Aufwand und Rechenleistung. Verfahren zur Reduzierung des Befehlsdatenstroms zielen letztendlich auf eine Verbesserung dieses Verhältnisses hin.

Ein erster Ansatzpunkt, vor allem für Spezialprozessoren, ist die Zusammenfassung von Folgen elementarer Befehle zu Spezialbefehlen (Abschnitt 4). Eine selektive Studie im Rahmen des Forschungsprojektes [1] hat sich in diesem Zusammenhang mit einem Prozessorkonzept für die Bildverarbeitung beschäftigt. Es wird am Beispiel gezeigt, daß es für Spezialanwendungen im Gegensatz zu Universalprozessoren noch große Leistungsreserven gibt.

Für unsere Untersuchungen auf dem Gebiet der Befehls- und Prozessorarchitektur ist vor allem der Übergang vom starren 32-Bit-Befehlsformat zu einem Befehlsformat mit maximaler Informationsdichte interessant: Entfernen redundanter Bitstellen, Entfernung redundanter Operanden, implizite Adressierung von Operanden und die Berücksichtigung der Nutzungshäufigkeit bei der Kodierung (Abschn. 5). Es wird eine Befehlsarchitektur mit Bestandteilen variabler Länge vorgeschlagen.

Abschnitt 6 zeigt, daß die Abarbeitung von Befehlen mit Bestandteilen variabler Länge bei der heutigen Integrationsdichte kein Problem mehr darstellen dürfte.

Wichtige Merkmale neuerer und zukünftiger Prozessoren sind Pipeline-Verarbeitung und Parallelverarbeitung auf Anweisungsniveau. Zur aggressiven Ausbeutung der sogenannten feinkörnigen Parallelität werden Techniken angewandt, die den Befehlsdatenstrom erhöhen, z.B. In-Line-Funktionen, spekulative Ausführung, Trace-Scheduling [Fisher81] (Performance-Optimierung bei VLIW-Prozessoren). Dieser Code-Overhead muß gleichfalls in die Code-Optimierung einbezogen werden (Abschn. 7).

Der Aufwand, feinkörnige Parallelität in größerem Umfang auszubeuten, ist beachtlich. Der erzielte Leistungsgewinn ist im Verhältnis dazu gering. Der Abschn. 7 fragt deshalb weiter, was die Ausbeutung grobkörniger Parallelität auf Thread-Niveau im Vergleich dazu kosten würde.

Eine wichtige Forderung an eine neue Befehlsarchitektur ist, die Abwärtskompatibilität zwischen mehreren Generationen von Prozessoren zu garantieren. Abschn. 8 diskutiert die Möglichkeit, auf der Basis von Befehlen mit Elementen variabler Länge ein Prozessormodell zu entwickeln, daß Binärkompatibilität zwischen Prozessoren von einigen Millionen bis Milliarden Transistoren bei einer sinnvollen Nutzung der Hardware gestattet. Natürlich können wir zu diesem komplexen Problemfeld nur einige Anregungen liefern.

Der letzte Abschnitt beschreibt den Stand der Vorarbeiten für die geplanten Experimente.

2 Das ursprüngliche RISC-Konzept

In einem RISC-Prozessor sind mehrere Konzepte zur Opitimierung eines Universalprozessors verwirklicht [2]. Sie sollen hier getrennt unter dem Blickwinkel bewertet werden, daß auf sie eventuell zu Gunsten eines geringeren Befehlsdatenstroms verzichtet wird.

Load/Store-Architektur

Der Begriff "reduzierter Befehlssatz" steht im wesentlichen für die Trennung zwischen Lade-/Speicherbefehlen und Verarbeitungsbefehlen. Speicherzugriffe erfolgen bei einem Von-Neumann-Rechner immer einzeln und nacheinander. Die eigentliche Verarbeitung muß auf den Abschluß der Speicherzugriffe warten. Die Addition von zwei Werten im Hauptspeicher besteht z.B. aus vier nacheinander auszuführenden Aktionen: Laden von Operand 1 und 2 in den Prozessor, Addition und Speichern des Ergebnisses. Während ein CISC-Prozessor Befehle für die Kombination aus Speicherzugriff und Operation besitzt, wird für den RISC-Prozessor jede der 4 Aktionen durch einen extra Befehl gesteuert. Wichtig ist dabei, daß der CISC-Prozessor zwar mit weniger Befehlen, aber nicht mit weniger Verarbeitungsschritten auskommt.

Die Teilung komplexer Befehle in Verarbeitungsschritte hat entscheidende Vorteile. Für die Programmierung steht jede Kombination zwischen Verarbeitung und Speichertransfer zur Verfügung. RISC-Code ist einfacher zu erzeugen und zu optimieren, bzw. im Ergebnis entstehen i.allg. schnellere Programme. Der Zusammenhang wird anschaulich, wenn man in Analogie an die Beziehung zwischen Laut- und Silbenschriften denkt. Genau wie aus RISC-Befehlen komplexere Operationen zusammengesetzt werden, dienen Laute zur Konstruktion von Silben. Man stelle sich vor, wir müßten in Zukunft wie im Chinesischen in Silbenzeichen schreiben. Neben dem Lernaufwand für die vielen Silbenzeichen könnten wir viele Nuancen und Feinheiten nicht ausdrücken. Mit diesem Problem sind die herkömmlichen CISC-Befehlssätze behaftet.

Mit der Weiterentwicklung der RISC-Technologie ist das Prinzip, den Befehlssatz auf wenige elementare Befehle zu beschränken, nicht konsequent beibehalten wurden (vgl. Abschn. 3 und 4). Die Codeoptimierung nach dem Prinzip der maximalen Informationsdichte, die eine variable Befehlsgröße impliziert, würde es erlauben, auf höherem Niveau zu einer einfachen Befehlsarchitektur zurückzukehren (Abschn. 5 und 8).

Universelle Register

Wesentliche Ansatzpunkte der ursprünglichen RISC-Architektur sind einfacher Aufbau, möglichst Integration auf einem Chip (zum Entstehungszeitpunkt entsprach das etwa 50.000 Gatteräquivalenten) und hohe Taktfrequenz. Das spiegelt sich auch in der Registerarchitektur wider. Es gibt im Prinzip nur ein Register mit Spezialfunktionen, den Befehlszähler. Die anderen Register können beliebig als Datenregister, als Konstanten, als Zeiger oder als Flags genutzt werden. Damit gibt es z.B. keine speziellen Befehle für die Stackbehandlung und für Unterprogrammaufrufe. Aktionen wie das Ablegen oder Laden von Daten aus dem Keller werden aus gewöhnlichen Lade-, Speicher- und Verarbeitungsoperationen zusammengesetzt. Der Preis ist ein größerer Befehlsdatenstrom. Aber auch das Prinzip universeller Register ist in der weiteren Entwicklung nicht konsequent beibehalten wurden.

Programmierkonventionen weisen in der Regel den einzelnen Registern bestimmte Funktionen zu (z.B. Parameterübergabe bei Funktionsaufrufen, Stack- oder Frame-Pointer). Diese Konventionen sind die Voraussetzung für standardisierte Programmierschnittstellen und führen zu großen Unterschieden in der Häufigkeit, mit der die einzelnen Register genutzt werden. In Abschnitt 5 wird gezeigt, daß es zur Spezialisierung der Register auch noch eine Alternative gibt, um die unterschiedliche Nutzungshäufigkeit zur Codereduzierung zu nutzen.

Pipelining

Ihre hohe Rechenleistung verdanken die RISC-Prozessoren der Pipeline-Verarbeitung. Die einzelnen Aktionen (Speicherzugriffe, arithmetische oder logische Operationen, Sprünge, ...) werden nicht nacheinander, sondern überlagernd ausgeführt. Alle Operationen sind in Verarbeitungsphasen geteilt. Die erste Phase einer Operation erfolgt zeitgleich mit der zweiten Phase der vorhergehenden, mit der dritten Phase der vorletzen Operation und so weiter. Im Idealfall beginnt in jedem Takt eine neue Operation, und es wird eine fertiggestellt, d.h. es werden mehrere Operationen zeitgleich bearbeitet (Bild 1).

Anschaulich läßt sich eine Pipeline-Verarbeitung nur für Befehlsmengen mit vergleichbarem Verarbeitungsfluß realisieren. Hier tritt ein ganz wesentlicher Vorteil des RISC-Konzeptes zu Tage. Die Trennung komplexer Befehle in Datentransfer und Verarbeitung vereinfacht die Pipeline-Konstruktion bzw. ermöglicht sie. Neue Prozessoren mit komplexen Befehlssätzen müssen, um in ihrer Rechenleistung mit RISC-Prozessoren vergleichbar zu bleiben, auch das Pipeline-Prinzip unterstützen. (Bei Intels P6 wird einem RISC-artigen Prozessorkern ein Werk vorgelagert, das komplexe Befehle in RISC-artige Mikrobefehle zerlegt [4].)

Die Pipeline-Technik ist für ein gutes Verhältnis zwischen Schaltungsaufwand und Rechenleistung so wichtig, daß sie durch Konzepte zur Befehlsdatenreduzierung nicht in Frage gestellt werden darf.

Einheitliche Größe der Befehlsworte

Befehlsworte klassischer und auch neuerer RISC-Prozessoren sind einheitlich 32 Bit groß, und sie besitzen einen sehr regelmäßigen Aufbau (vgl. Abschn. 5). Der Vorteil ist ein einfacher, schneller Befehlsdecoder. Davon ist heute nur noch "schnell" interessant. Einige 10.000 Gatteräquivalente zusätzlich für den Befehlsdecoder wären zu vernachlässigen.

Der starre 32-Bit-Befehlsaufbau hat auch Nachteile. Der Wertebereich für Konstanten ist begrenzt (in Dreiadreßbefehlen auf 16 Bit, für Verzweigungen auf kurze Sprungdistanzen). Das erfordert Sonderbehandlungen für die Arbeit mit größeren Konstanten (z.B. den wahlfreien Zugriff auf Datenbereiche größer 64 kByte) und für bedingte Sprünge zu weiter entfernten Codesegmenten. Die Sonderbehandlungen reichen bis zur Implementierung von Spezialbefehlen z.B. für das Zusammensetzen von 32-Bit-Konstanten aus 16-Bit-Konstanten und erschweren die Programmierung.

Der zweite Nachteil des starren Befehlsaufbaus ist eine relativ schlechte Codedichte. Dieses Problem ist in Abschn. 5 Ausgangspunkt für die Befehlsdatenstromreduzierung durch Befehlsformate mit Bestandteilen variabler Länge.

3 Das RISC-Konzept hat sich weiter entwickelt

Die ursprüngliche RISC-Architektur war für Prozessoren mit etwa 50.000 Gatteräquivalenten konzipiert und besitzt ein ausgezeichnetes Verhältnis zwischen Schaltungsaufwand und nutzbarer Rechenleistung. Für die heutigen

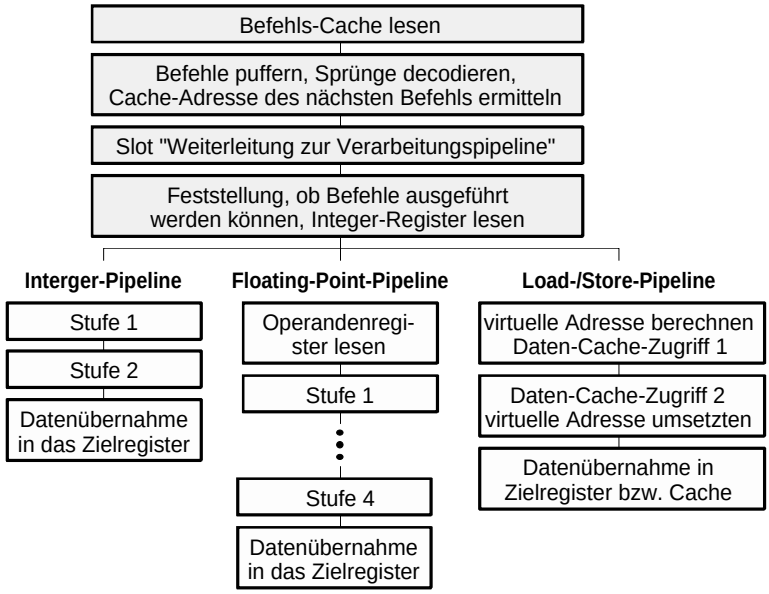

Bild 1: Pipeline-Stufen des Alpha 21164 (vereinfacht) [3]

Prozessor-Chips stehen zehn- bis hundertmal so viele und um mehrere Faktoren schnellere Hardwarestrukturen zur Verfügung. Diese zusätzlichen Ressourcen galt es in Rechenleistung umzusetzen.

Mit der *n*-fache Anzahl von Hardwarestrukturen läßt sich nicht problemlos die *n*-fache Rechenleistung erzielen. Es ist einfach, auf einem Chip zehn und mehr Rechenwerke oder komplette ursprüngliche RISC-Prozessoren zu integrieren. Aber es ist sehr schwierig mehre Rechenwerke oder Prozessoren zeitgleich zu nutzen. Der oder die Prozessor(en) kann (können) nicht schneller arbeiten als der Speicher Befehle und Daten bereitstellt. Auf diesen Engpaß zielen zahlreiche Merkmale neuerer RISC-Prozessoren. Man spricht auch von der:

Aufweitung des Flaschenhalses der Von-Neumann-Architektur

- On-chip-Cache: Der größte Teil der Chipfläche neuerer schneller Prozessoren wird vom On-Chip-Cache beansprucht. Mit der zunehmenden Geschwindigkeit der Prozessoren vergrößert sich die Kluft zwischen Prozessor- und Speichergeschwindigkeit. Die Befehlsholezeit eines 350-MHz-Alpha-Prozessors unterscheidet sich z.B. von der Zugriffszeit eines 60-ns-dRAMs um den Faktor 21, d.h. ohne Zusatzmaßnahmen würden auf jeden Arbeitstakt mindestens 20 Takte Wartezeit entfallen. Der Cache, ein schneller assoziativer Speicher, der die zuletzt genutzten Adreßbereiche doppelt, verringert den Anteil der Wartetakte je nach seiner Größe und Geschwindigkeit bis auf etwa 50%. Er ist die Voraussetzung, daß die hohe Prozessorgeschwindigkeit wenigstens zu einem Teil genutzt werden kann.
- Mehrere Cache-Ebenen: Speicher werden bei gleicher Technologie mit zunehmender Größe langsamer. Das spiegelt sich in den heutigen Prozessoren darin wider, daß der On-Chip-Cache, der Prozessorzugriffe ohne Wartezeit befriedigt, relativ klein ist, und der Prozessor einen zweiten wesentlich größeren, aber langsameren Second-Level-Cache und gegebenenfalls weitere Cache-Ebenen unterstützt.
- Übergang vom Write-Through- zum Write-After-Modified-Cache: Die erste Strategie kostet weniger Hardware, und sie sichert, daß Cache- und Hauptspeichinhalt stets übereinstimmen. Im Zuge der Vergrößerung der Kluft zwischen Hauptspeicher- und Prozessorgeschwindigkeit verzichtet man auf diese Vorteile zu Gunsten eines geringeren Datentransfers zwischen Cache und Hauptspeicher.
- Erhöhung der Datenbusbreite: Zur Kompensation der hohen Wartezeit des Hauptspeichers wird die Zugriffsbreite vervielfacht. In jedem Zugriff werden mehrere aufeinanderfolgende Befehlsworte oder Daten zwischen Hauptspeicher und Cache übertragen.
- Pipelining für Hauptspeicherzugriffe und verschränkte Adressierung: Pipelining bedeutet, daß die Übertragung in mehre Phasen geteilt und das Ergebnis jeder Phase in einem Zwischenregister gespeichert wird. Nach diesem Prinzip läßt sich die Datenübertragung über den Bus bis einschließlich der Adreßdekodierung in mehrere Zeitscheiben teilen. Die Parallelisierung der Schreib- und Leseoperationen in den eigentlichen Speichermatritzen erfolgt nach dem Prinzip der verschränkten Adressierung bzw. dem Prinzip der Vervielfachung der Werke. Aufeinanderfolgende Hauptspeicheradressen sind physisch unterschiedlichen Speichermatritzen oder -chips zugeordnet, so daß ein großer Teil aufeinanderfolgender Schreib- und Leseoperationen zeitgleich ausgeführt werden kann. Eine intelligente Steuerung überwacht die Adreßreihenfolge und stoppt bei Konflikten den Prozessor bzw. den Cache-Controller. Der Prozessor i860 unterstützt z.B. drei verschachtelte Hauptspeicherzugriffe.
- Warteschlangen für Schreibzugriffe: Zu schreibende Daten werden in einen Puffer abgelegt. Treten Verzögerungen z.B. durch einen Cache-Miss auf, kann der Prozessor bis zum Pufferüberlauf weiter arbeiten (z.B. im PA7100 [5]).
- Reduzierung des Befehlsdatenstroms: Die Reduzierung des Befehlsdatenstroms, der Gegenstand unserer Untersuchungen, dient letztendlich dem selben Zweck wie der Cache, die Erhöhung der Datenbusbreite etc, der Verringerung der Wartezeit auf den Hauptspeicher.

  In integrierten Anwendungen arbeitet der Prozesssor (embedded controller) typisch mit langsamen, preiswerten Speichern zusammen. Die Cache-Größe und die Datenbusbreite ist zu minimieren. Eine systematische Reduzierung des Befehlsdatenstroms ist für solche Anwendungen eine kostengünstige Alternative.

Ausbeutung von programminterner Parallelität

Der natürliche Ansatz, die Rechenleistung eines Prozessors durch zusätzliche Hardware zu vergrößern, ist die Vervielfachung von Prozessorteilen. Ziel ist, mehrere Anweisungen oder Anweisungsfolgen gleichzeitig auszuführen. Man unterscheidet zwischen Parallelverarbeitung auf Funktions- (bzw. Thread-) Niveau (grobkörnige Parallelität) und Parallelität auf Befehlsebene (feinkörnige Parallelität). Innerhalb von RISC-Prozessoren wird bisher nur feinkörnige Parallelität genutzt. Die Prozessoren besitzen mehrere Rechenwerke, aber nur ein Werk zur Steuerung des Programmflusses (Befehlshole- und Sprunglogik).

Feinkörnige Parallelität kann nicht nur durch die Vervielfachung der Rechenwerke ausgebeutet werden. Ein vergleichbarer Effekt entsteht auch durch die Vergrößerung der Pipeline-Tiefe (Superpipelining). Pipeline-Stufen erlauben in Verbindung mit einer entsprechend hohen Taktfrequenz eine Parallelverarbeitung auf der Ebene von

Zeitscheiben. Ein Rechenwerk mit *n* Pipeline-Stufen erzielt bei gleicher Gesamtausführungszeit fast den selben Verarbeitungsdurchsatz wie *n* Rechenwerke ohne Pipeline [JoWa89]. Beide Konzepte werden kombiniert angewendet. Zusätzliche Pipeline-Stufen kosten weniger Transistoren, aber ab einer bestimmten Taktfrequenz ist es effektiver, zusätzliche Werke zu integrieren.

Einem Programm für seine Ausführung mehr Zeitscheiben oder Rechenwerke zur Verfügung zu stellen ist nur die eine Seite der Medallie. Die andere Seite, sie zu nutzen, ist weit schwieriger. Die Ressourcen-Auslastung bestimmt sehr wesentlich die Architektur, den Codeaufbau und auch den Befehlssatz moderner RISC-Prozessoren. Für die Befehlssatzoptimierung kann dieser Entwicklungstrend nicht ignoriert werden.

Techniken zur besseren Auslastung der Verarbeitungs-Pipeline

- Verringerung der Verlustzeiten bei Sprüngen und Verzweigungen: Die Berechnung der Nachfolgeadresse eines Sprunges besteht in der ursprünglichen RISC-Architektur genau wie jede andere Berechnung oder Adreßrechnung aus mehreren Schritten (z.B. Befehl holen, interne Operandenbereitstellung, Durchführung einer arithmetischen oder logischen Operation, Ergebnis in ein Register schreiben). Das Sprungziel wird erst nach mehreren Schritten in den Befehlszähler geladen. In der Zwischenzeit werden weiter Operationen in linearer Reihenfolge gestartet, die u.U. zur alternativen Verzweigungsrichtung gehören und abgebrochen bzw. rückgängig gemacht werden müssen. Zur Verringerung der Verlustzeiten durch das Löschen der Pipeline dienen unterschiedliche Maßnahmen:
  - Extra Rechenwerk für die Ausführung von Sprüngen: Durch zusätzliche Hardware u.a. einen extra Addierer für die Berechnung relativer Sprungziele wird die Anzahl der Schritte bis zum Laden des Sprungziels in den Befehlszähler gegenüber dem normalen Pipeline-Fluß verkürzt (Bsp. i860 [6]). Bedingte Sprünge, für die die Verzweigungsrichtung nicht vorzeitig bekannt ist, werden spekulativ ausgeführt. Stellt sich nach der Berechnung der Sprungbedingung heraus, daß die angenommene Verzweigungsrichtung falsch war, sind die inzwischen spekulativ begonnen Operationen ungültig und werden aus dem Pipelinefluß entfernt.
  - Statische Abschätzung der Verzweigungsrichtung: Die spekulativ auszuführende Verzweigungsrichtung wird vom Compiler festgelegt. Der Prozessor ist hierfür um entsprechende Befehle erweitert (realisiert z.B. im Motorola 88110 [7]).

    Statistisch gesehen werden Rückwärtssprünge wesentlich häufiger ausgeführt als Vorwärtssprünge. Das resultiert aus der linearen Befehlsabarbeitung. Ohne Sprung wird die Adresse mit jedem Befehl erhöht. Die Sprungdistanz zur Bildung von Schleifen ist negativ. Diese einfache Abschätzung der Verzweigungsrichtung verlangt keine zusätzliche Information im Befehlscode und wird z.B. im PA7100-Prozessor [8] angewandt.
  - Dynamische Abschätzung der Verzweigungsrichtung: Der Prozessor lernt die zu erwartenden Verzweigungsrichtungen, indem er Protokoll über die vorhergehende Ausführung bedingter Sprünge führt. In einem zusätzlichen vollassoziativen Speicher werden die Adressen der zuletzt abgearbeiteten bedingten Sprünge gemeinsam mit ihren Sprungzielen und einem Protokoll über die letzten Verzweigungsrichtungen geführt. Die Abarbeitung eines bedingten Sprunges mit Vorgeschichte wird durch einen Adreßvergleich erkannt und die Verzweigungsrichtung aus der Vorgeschichte abgeleitet. Im Alpha-Prozessor 21164 ist z.B. eine 2 Kbyte große Tabelle für die Speicherung der zuletzt ausgeführten Sprünge vorgesehen [3].
- Verringerung der Verlustzeiten auf Grund von Datenabhängigkeiten: Eine Operation kann erst gestartet werden, wenn alle Operanden bereitstehen. In einer Verarbeitungs-Pipeline stehen die Ergebnisse zum Teil erst um mehre Takte verzögert zur Verfügung. Eine Operation darf erst um eine entsprechende Anzahl von Zeitscheiben nach den Operationen, die ihre Operanden bereitstellen, gestartet werden. Die Lücke im Befehlsdatenfluß ist gegebenfalls mit NOP- (no operation) Befehlen zu füllen. Die Anzahl der NOP-Befehle wird im Flusse der Code-Optimierung minimiert, indem die Anweisungen in ihrer Reihenfolge umsortiert werden (scheduling).

  Zur weiteren Eliminierung ungenutzter Zeitscheiben gibt es das Prinzip der Operandenumleitung (forwarding). Eine spezielle prozessorinterne Logik erkennt die Datenabhängigkeit aus den Registeradressen. Statt des normalen internen Pipeline-Flusses für Verarbeitungsbefehle (..., Lesen der Operanden aus ihren Registern, Operation, Speichern des Ergebnisses in das Ergebnisregister, ...; vgl. Bild 1) wird das Ergebnis direkt auf den Eingang des Rechenwerks umgeleitet. Abhängige Operationen dürfen dichter aufeinander folgen.

Nutzung mehrerer Rechenwerke

Die Anzahl der aufeinanderfolgenden Maschinenbefehle, die zeitgleich oder unmittelbar aufeinander in einer Pipeline ausgeführt werden können, ist nicht groß. Der Abstand zwischen zwei Verzweigungen liegt typisch bei 7 Operationen. Zwischen denen bestehen auch noch Datenabhängigkeiten. Um mehrere Rechenwerke zeitgleich nutzen zu können, werden Techniken angewandt wie:

- Aufrollen von Schleifen (loop unrolling)
- Änderung der Befehlsreihenfolge (scheduling) [9, 10, 11].

- Ersetzen von Prozedur-Aufrufen durch den Prozedur-Körper (Inline-Funktionen)
- Spekulative Ausführung von Anweisungen über Verzweigungsgrenzen hinaus.

Für diese Techniken zur Erhöhung der Parallelverarbeitung gibt es unterschiedliche Formen der Aufgabenteilung zwischen Prozessor und Compiler.

- VLIW- (very long instruction word) Konzept: Der Compiler führt alle Optimierungen einschließlich der Zuordnung von Operationen zu Zeitscheiben und Rechenwerken durch. Die Informationen hierzu werden im Befehlscode verschlüsselt. Für jedes Rechenwerk wird eine bestimmte Anzahl von Bitstellen im Befehlswort (in der Regel 32 Bit) reserviert. Probleme bereitet die schleche Auslastung der Rechenwerke, die sich im VLIW-Konzept auf einen großen Code-Overhead abbildet. Für 8 Werke und 32 Bit je Werk wäre die Befehlswortbreite z.B. insgesamt 256 Bit (daher der Name "Sehr-lange-Befehlsworte"-Architektur). Bei der typischen Auslastung der Rechenwerke in der Größenordnung von 25% entfallen allein 75% des zu übertragenden Befehlscodes auf NOP- (no operation) Anweisungen.
- Das Gegenstück zur Superskalaren Architektur ist die VLIW-Architektur. In ihr sind die Zuweisungsalgorithmen in Hardware realisiert. Der Prozessor erhält sequentiellen, nicht optimierten Code, verändert in der Laufzeit die Abarbeitungsreihenfolge, führt dabei auch Anweisungen spekulativ aus und nennt bei Bedarf Register um. Das Konzept verlangt keinen Zusatzcode zur Verschlüsselung der Parallelität, aber zusätzliche Hardware.
- Eine Mischform zwischen VLIW- und superskalaren Prozessoren besteht in folgendem. Der Prozessor bestimmt die Zuordnung zwischen Operationen und Rechenwerken, aber er ändert nicht die Befehlsreihenfolge. Für letzteres und damit für die Ausnutzung der Rechenwerke ist der Compiler verantwortlich. Er kann zeitgleich ausführbare Befehle kennzeichnen, indem er sie nacheinander anordnet. Der Prozessor erkennt das an den Operanden. NOP-Befehle werden implizit eingefügt.

Die neuesten, schnellen Prozessoren sind gegenwärtig superskalar (Alpha, Pentium, P6), ihre Vorgängermodelle basierten auf der Mischform. Die Ursache liegt weniger in der erzielbaren Parallelverarbeitung als in Überlegungen zur Portierung älterer Programme und im Widerstand der Programmierer gegen neue Programmiertechniken. Die Superskalare Architektur und ihr Vorgänger ohne Veränderung der Befehlsreihenfolge erlauben Code-Kompatibilität zwischen Prozessoren mit einer unterschiedlichen Anzahl von Werken. Die Superskalare Architektur verbirgt darüber hinaus die veränderte Code-Reihenfolge vor dem Programmierer auch im Falle von Interrupts und Traps. Sie bietet jedoch kaum noch Möglichkeiten, zusätzliche Hardware in höhere Rechenleistung umzusetzen.

Eine Vergrößerung der Anzahl der Rechenwerke allein ist für Superskalare Prozessoren sinnlos. Zur Erhöhung der nutzbaren Parallelität werden neue Zuordnungsalgorithmen gebraucht, die in Hardware umzusetzen sind. Eine kompliziertere Hardware wirkt sich ihrerseits negativ auf die erzielbare Taktfrequenz aus. Genau wie zum Zeitpunkt des Aufkommens der RISC-Idee steht wieder ein Qualitätssprung bevor. Die Zukunft gehört einfacheren Architekturen, die keinen Ballast in Form veralteter Programmiermodelle mitführen. Voraussichtlich wird die Weiterentwicklung dann auf der VLIW-Architektur aufsetzten [13].

Die schlechte Codedichte der VLIW-Architektur resultiert in erster Linie aus dem Festhalten der RISC-Prozessoren am 32-Bit-Befehlsformat. Für dieses Problem sind nach unserem jetzigen Arbeitsstand gute Lösungen in Sicht (Abschn. 5). Hier sehen wir einen wichtigen Beitrag für unsere Forschung.

Noch einmal Aufwand und Nutzen

Das Verhältnis zwischen Aufwand und Nutzen der produzierten Prozessoren wird mit zunehmender Größe schlechter. In Bild 2 sind für die wichtigsten RISC-Prozessoren der letzten Jahre Transistoranzahl und Taktfrequenz der nutzbaren Rechenleistung gegenübergestellt. Grob abgeschätzt wird mit dem zehnfachen Schaltungsaufwand und der zehnfachen Taktfrequenz die fünf- bis zehnfache Rechenleistung erzielt. Der Leistungsgewinn resultiert überwiegend aus der höheren Geschwindigkeit der Halbleiterstrukturen. Der gößere Teil der zusätzlichen Hardware dient nur dazu, die höhere Geschwindigkeit nutzen zu können (On-chip-Cache, größere Busbreite, ...). Darüber hinaus werden Hardware-Ressourcen zur Ausbeutung feinkörniger Parallelität eingesetzt.

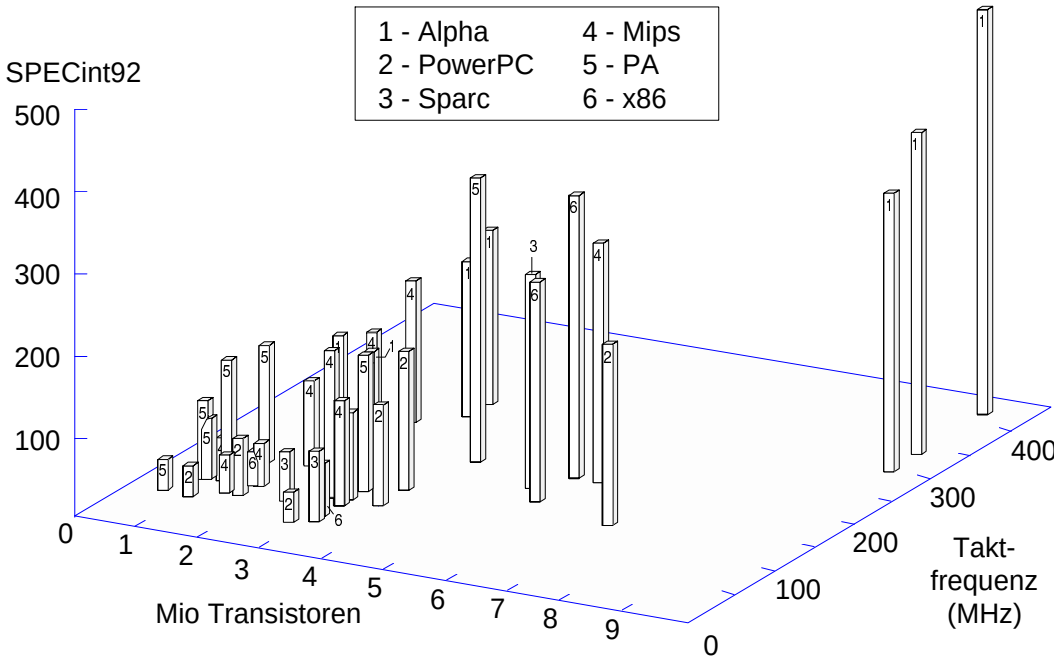

Bild 2: Entwicklung von Transistoranzahl, Taktfrequenz und Rechenleistung schneller Prozessoren

Das Potential für weitere Leistungssteigerungen ist offensichtlich weitgehend ausgeschöpft. Die Befehlsdatenstromoptimierung bietet in diesem Zusammenhang eine bisher kaum genutzte Leistungsreserve, um mit relativ wenig Zusatzaufwand noch einmal einen spürbaren Leistungsgewinn zu erzielen. Die besten Ansatzpunkte bieten integrierte Prozessoren (embeded controller) und die VLIW-Architektur. In integrierten Anwendungen muß der Prozessor in der Regel mit langsamen Speichern, kleinem Cache etc zusammenarbeiten, so daß hier die Leistungssteigerung durch einen verringerten Befehlsdatenstrom am deutlichsten zu Tage tritt. Im Zusammenhang mit der VLIW-Architektur geht es darum, Vorarbeiten für zukünftige Prozessoren zu leisten, die den Balast veralteter Programmiermodelle abgelegt haben.

4 Spezialbefehle

Spezialbefehle sind ein wichtiger Ansatz für die Reduzierung des Befehlsdatenstroms. Eine Teilaufgabe, die eine Folge von Grundbefehlen erfordert, wird zu einer Anweisung zusammengefaßt. Spezialbefehle bewirken nur eine spürbare Codereduzierung, wenn sie ausreichend häufig genutzt werden. Hier liegt das Kernproblem. Der Prozessor kann nur mit Blick auf spezielle Anwendungen (z.B. Bildverarbeitung oder Computergraphik) oder mit Blick auf bestimmte Techniken der Software-Entwicklung (Hochsprache, Compiler) optimiert werden.

Für einen Universalprozessor bieten die Werkzeuge für die Software-Entwicklung die besseren Ansatzpunkte. Sie bestimmen die Code-Eigenschaften eines breiten Spektrums von Programmen und sind sehr konservativ. Spezialbefehle, die software-technische Code-Eigenarten ausnutzen, sind von unseren beiden Zielanwendungen nur für die integrierten Prozessoren (embedded controller) interessant. Das von uns favorisierte VLIW-Modell benötigt neue Compiler und neue Algorithmen für die Code-Optimierung. Diese müßten erst entwickelt und in größerem Umfang genutzt werden, bevor an compiler-spezifische Spezialbefehle zu denken ist.

Es gibt zwei Arten von Spezialbefehlen. Mit der RISC-Philosophie konforme Spezialbefehle werden in einer Zeitscheibe der Verarbeitungs-Pipeline abgearbeitet und bewirken eine zur Verringerung der Befehlsanzahl proportionale Verkürzung der Verarbeitungszeit. Das Gegenstück sind CISC-artige Spezialbefehle, die mehrere Zeitscheiben benötigen. Sie erhöhen die Geschwindigkeit nur indirekt über den geringeren Befehlsdatenstrom und die damit verbundenen geringeren Wartezeiten auf den Hauptspeicher. Für integrierte Prozessoren sind CISC-artige Spezialbefehle sicher interessant. Bei Prozessoren mit ausreichend großem Cache ist der Leistungsgewinn eher vernachlässigbar.

Verarbeitung in einer Pipeline-Zeitscheibe

Typische aus der Softwareentwicklung resultierende und für Spezialbefehle geeignete Befehlsfolgen sind Gleitkommaaddition, -subtraktion, -multiplikation: In neueren Prozessoren in einer Zeitscheibe abgearbeitet, ersetzen sie eine längere Folge von Festkommaoperationen. Gleitkommaoperationen werden von zahlreichen

Programmiersprachen und Compilern unterstützt. Das bewirkt indirekt, daß sie viel öfter als unbedingt notwendig eingesetzt werden. Weitere Kombinationen sind in [14] beschrieben. Im Rahmen unserer Untersuchungen soll diese Aufzählung vervollständigt und der Rechenzeitgewinn, der mit den einzelnen Spezialbefehlen erzielt wird, experimentell abgeschätzt werden.

Die besten Ansatzpunkte für Spezialbefehle aus Sicht der abzuarbeitenden Algorithmen bieten die Textverarbeitung, die Computergraphik und die Bildverarbeitung. Typische Vertreter sind:

- Vektoroperationen für byte-organisisierte Größen: Die kleinsten zu manipulierenden Datenstrukturen in der Textverarbeitung und in Rasterbildern sind i.allg. Aufzählungstypen der Größe Byte (Zeichen, Grau- oder Farbwert). Hinter Vektoroperationen mit Byte-Größen verbirgt sich die Idee, die volle Verarbeitungsbreite der Rechenwerke von 4 oder 8 Byte für Operationen mit kleinen Operanden zu nutzen, indem die Operanden als Byte-Vektoren aufgefaßt werden. Bei der Addition und Subtraktion heißt das z.B. nur, daß die byte-übergreifenden Überträge unterdrückt werden [5, 15].
- Kombinierter Multiplikations-Additionsbefehl: Das ist eine Grundoperationsfolge z.B. für lineare Faltungen und Koordinatentransformationen, deren Ausführungszeit die Geschwindigkeit zahlreicher Algorithmen der Bildverarbeitung, der Computergraphik und der digitalen Signalverarbeitung wesentlich beeinflußt.

Weniger häufig angewandte rechenzeitintensive Algorithmen, die durch Spezialbefehle unterstützt werden könnten, sind in Simulationen aller Art zu finden. Auch auf dem Bereich der algorithmenspezifischen Spezialbefehle sind noch quantitative Untersuchungen geplant.

Spezialbefehle mit sequentieller Abarbeitung

Eine Grundidee des RISC-Konzepts ist, solche Befehle zu vermeiden und sequentielle Abläufe durch eine Folge von Befehlen nachzubilden. In neueren Prozessoren aus der CISC-Linie (z.B. dem P6 von Intel) werden solche Komplexbefehle von einer Art Mikroprogramsteuerwerk vor der eigentlichen Verarbeitungs-Pipeline in RISC-artige Befehle zerlegt. Das ist von der Wirkung her vergleichbar damit, daß bestimmte Prozeduren permanent im First-Level-Cache eines RISC-Prozessors gehalten werden. Diese Prozeduren sind genau wie das Mikroprogramm stets ohne Verzögerung verfügbar, aber sie blockieren Speicherplatz, der nicht von anderen Prozeduren genutzt werden kann.

Ein Leistungsgewinn entsteht nur, wenn die als Spezialbefehl implementierten Funktionen sehr oft benötigt werden (z.B. der Transfer von Daten zwischen Register und Stack oder die Operationsfolge für Unterprogrammaufrufe) oder wenn zusätzliche, nicht anders nutzbare Prozessor-Ressourcen verwendet werden (typ. für Gleitkomma-Division, Quadratwurzel und andere Standardoperationen).

Programmierbare Verarbeitungswerke

Das Problem der RISC-artigen Spezialbefehle ist, daß der Geschwindigkeitsgewinn nur für wenige Aufgaben erzielt wird. Eine zu hohe Spezialisierung des Befehlssatzes schränkt die Anwendungsbreite des Prozessors ein. Ökonomisch gesehen ist er dann kein preiswertes Massenprodukt mehr, was seinerseits ein schlechteres Preis-Leistungsverhältnis zur Folge hat.

Ein interessantes Konzept, das sowohl eine hohe Spezialisierung des Befehlssatzes als auch eine große Anwendungsbreite erlaubt, bieten programmierbare Verarbeitungseinheiten. Sie können für unterschiedliche Aufgaben unterschiedliche Spezialbefehle nachbilden. Im Rahmen des Forschungsprojektes wurde eine Teilstudie zu einem Bildverarbeitungsprozessor mit programmierbarem Befehlssatz durchgeführt [1]. Das Ergebnis ist bemerkenswert. Es konnte gezeigt werden, daß bei vergleichbarer Herstellungstechnologie und Prozessorgröße ein Rechenleistungsgewinn um den Faktor 10 bis 100 möglich ist.

Für die Forschung eröffnen programmierbare Verarbeitungswerke neue Problemfelder. Speziell auf unseren Untersuchungsgegenstand bezogen ist zusätzlich zum Befehlsdatenstrom auch der Datenstrom für Konfigurationswechsel des Rechenwerkes zu minimieren. Ein Geschwindigkeitsgewinn des gesamten Prozessors verlangt abschätzungsweise, daß der Konfigurationsdatenstrom deutlich geringer ist als die Einsparung am Befehlsdatenstrom. Aus der Studie [Kem95] geht bereits hervor, daß diese einfache Aussage großen Einfluß auf die Architektur konfigurierbarer Rechenwerke besitzt (u.a. daß die gegenwärtigen freiprogrammierbaren Schaltkreise zu viel Information für die Konfiguration benötigten). Das Problem ist insgesamt vielschichtig und sprengt den Rahmen des gegenwärtigen Forschungsprojektes.

Die Entwicklung zu immer komplexeren Prozessoren geht weiter. Es ist abzusehen, daß die erzielbaren Leistungsgewinne aus der Vergrößerung der On-chip-Caches und weiteren Verarbeitungswerken bald ausgeschöpft sind. Programmierbare Verarbeitungswerke könnten einer der nächsten Meilsteine der Prozessorentwicklung sein und für uns ein neuer Forschungsgegenstand werden.

5 Optimierung der Befehlskodierung

Den vorangegangenen Abschnitt "Spezialbefehle" könnte man auch bildlich gesehen überschreiben mit "Optimierung der Granularität der Programmbausteine". Es galt, die Vorteile kleiner, elementarer Befehle (größere Flexibilität und mehr Spielraum für die Code-Optimierung) gegen die Codeeinsparung durch Zusammenfassung von Befehlsfolgen zu Spezialbefehlen abzuwägen (vgl. Abschn. 2. In diesem Abschnitt geht es nun darum, diese Programmbausteine optimal zu codieren.

Im Befehlsdatenstrom eines RISC-Prozessors ist ein Teil der übertragenen Bits redundant. Es gibt für jede Operation einen Befehl mit drei Registeradressen und für viele dieser Befehle das Äquivalent mit zwei Registeradressen und einem Direktwert (Bild 3). Die Differenz zwischen der Länge eines Direktwertes und der Länge einer Registeradresse (typ. 16 - 5 = 11 Bit) läßt sich nicht effektiv nutzen. Die einzelnen Prozessoren codieren in diesen Bereich unterschiedliche, seltener genutzte oder gar keine Informationen. In der Studie für MIPS R2000/R3000 Prozessoren [16] beträgt der Anteil redundanter Bitstellen im Programmfluß etwa 7 bis 9% (2 bis 3 Bit je Befehlswort).

Hinzu kommt eine Art versteckte Redundanz. Kleine Direktwerte und Sprungdistanzen könnten in wesentlich weniger Bitstellen codiert werden, als für sie reserviert sind. Große Direktwerte sind durch kurze Programmsequenzen nachzubilden. In vielen Anweisungen sind komplette Registeradressen überflüssig.

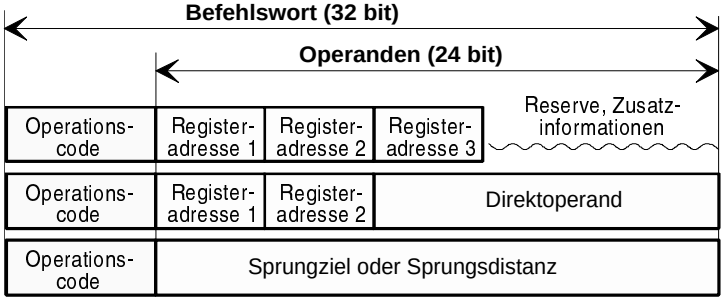

Bild 3:Typische Befehlsformate eines RISC-Prozessors

Zwei Operationen je Befehlswort

In ungenutzte Bitstellen, nicht benötigte Registeradressen und nicht benötigte Teile von Direktwerten können die Informationen für einen zweiten Befehls verschlüsselt werden. Ein Beispiel ist die Verschiebungskonstante im MIPS-Befehlssatz [17]. Sie bietet die Möglichkeit, herkömmliche 3-Registeradreß-Befehle mit einem Verschiebebefehl des Ergebnisses zu kombinieren. In [14] wird die Kombination von Registeradreßbefehlen mit Inkrement-Operationen vorgeschlagen.

Die Zusammenfassung von zwei Operationen in einem 32-Bit-Befehlswort ist ein kompliziertes Puzzle. Es müssen zum einen Operationen sein, die häufig in Kombination auftreten. Zum anderen lassen sich in 32 Bit kaum zweimal Operationscode und zweimal Operanden codiern. Der Ansatz läuft auf einen Sonderfall für Spezialbefehle hinaus. Eine Folge von zwei Operationen wird zu einer Anweisung zusammengefaßt.

Spezialbefehle, die als Lückenfüller für redundante Bitstellen dienen, sind für die Programmierung ungünstig. Der Programmierer bzw. der Compiler erzeugt erst einen Code aus Elementarbefehlen und sucht anschließend, ob der Code zufällig zusammenfaßbare Befehlskombinationen enthält. Die Nutzungshäufigkeit ist entsprechend schlecht. In unseren Untersuchungen wollen wir uns deshalb nicht weiter mit solchen Lückenfüllern beschäftigen, sondern ohne Rücksicht auf Kompatibilität zu existierenden Befehlsarchitekturen ein neues Konzept entwickeln.

Nutzung allgemeiner Verfahren der Datenkompression

Die Komprimierung von Daten ist eine Standardaufgabe im Zusammenhang mit der Informationsübertragung und -speicherung. Einer Studie zu herkömmlichen Verfahren der Datenkompaktierung [18] zeigt, daß sie für die Reduzierung des Befehlsdatenstroms wenig geeignet sind.

Partiell geeignet ist eine byte-weise Huffman-Kodierung der Befehlsfolge. Damit läßt sich der Codeumfang einer typischen RISC-Befehlsfolge um etwa 30% verringern. Das Verfahren wurde für das Auspacken einer komprimierten Befehlsfolge aus einem langsamen EPROM in einen wesentlich schnelleren Cache vorgeschlagen [19]. In dieser Konfiguration, typisch für integrierte Prozessoren, wird mehr Übertragungszeit gespart als für die Rückkonvertierung in herkömmlichen RISC-Code wieder verbraucht wird. Probleme bereitet das Aufsuchen von Sprungzielen im komprimierten Code, wenn der Befehls-Code, wie in [19] beschriebenen, bereits beim Laden in den Cache entkomprimiert wird. Die Anweisungen besitzen im EPROM und im Cache unterschiedliche Adressen bzw.

unterschiedliche relative Abstände zueinander, so daß nach jedem Cache-Miss die Anfangsadresse der nachzuladenden Cache-Seite über eine Umrechnungstabelle ermittelt werden muß.

Uns soll dieses Verfahren nur in modifizierter Form interessieren. Basiseinheit für die Komprimierung sollen nicht das Byte, sonderen es sollen die Bausteine des Befehlswortes sein (Operationscode, Registeradressen, ...). Das verspricht eine stärkere Codereduzierung. Zur Vermeidung von zusätzlichen Adreßrechnungen bei jedem Cache-Miss und zur besseren Ausnutzung des Caches ist die Entkomprimierung in den Befehlsdecoder zu verlagern.

Verzicht auf überflüssige Befehlsbestandteile

Der ursprüngliche RISC-Prozessor kennt nur 3-Adreßbefehle. Es gibt zahlreiche Operationen, die aus semantischer Sicht keinen, nur einen oder zwei Operanden benötigen würden. In der Einführung solcher Befehle liegt eine große Reserve für die Verringerung des Befehlsdatenstroms (Tabelle 1). Die Kehrseite dieses Ansatzes ist, daß das 32-Bit-Befehlsformat aufgegeben werden muß. Der Prozessor benötigt einen Befehlsdecoder für Befehle variabler Länge. Abschnitt 6 wird zeigen, daß ein solcher Decoder schaltungstechnisch kein Problem darstellt.

Befehlsaufbau	Beispiele
OP-Code	• NOP- (no operation) Anweisung
OP-Code | Reg.-Adr.	• Inkrementieren, Dekrementieren Verschieben des Inhalts eines Registers
OP-Code | Direktwert	• Spung
OP-Code | Reg.-Adr. 1 | Reg.-Adr. 2	• registeradressierte Load- und Store-Befehle • Move-Befehle • arithmetische und logische Operationen, in denen das Ergebnis einen Operanden überschreibt
OP-Kode | Reg.-Adr. 1 | Direktwert	• direktadressierte Load- und Store-Befehle • Laden von Konstanten • arithmetische und logische Operationen, in denen ein Registerinhalt mit einem Direktwert verknüpft und das Ergebnis in das Register zurückgeschrieben wird

Tabelle 1: Operationen mit weniger als drei Operanden

Untersuchungen zur erzielbaren Befehlsdatenstromreduzierung wurden begonnen. Grob abgeschätzt läßt sich der Befehlsdatenstrom durch die Minimierung der Anzahl der Befehlselemete etwa um 20 bis 30% verringern. Verläßliche Zahlen verlangen jedoch erst einmal, daß ein optimierter Befehlssatz entwickelt wird. Diese Aufgabe wird noch längere Zeit in Anspruch nehmen.

Darstellungsformate für Direktwerte

Nach der Minimierung der Anzahl der Befehlskomponenten müssen die einzelnen Elemente eines Befehlswortes optimal dargestellt werden. Direktwerte besitzen das größte Potential zur Code-Einsparung. In ganz grober Näherung ist die Nutzungshäufigkeit von Direktwerten proportional zur Anzahl der genutzten Bitstellen für ihre Kodierung [2]. Eine solche Verteilung verlangt ein Format mit variabler Länge.

Eine einfache Darstellung ist eine Längeninformation und einer an die Größe des Direktwertes angepaßte Anzahl von Bitstellen für die eigentliche Information. Zur Vermeidung von Sonderbehandlungen für große Direktwerte und Adreßkonstanten sollten 32-Bit- und gegebenenfalls auch größere Konstanten darstellbar sein. Für die Art und die Anzahl der Längenformate ist ein Optimum zu suchen. Die Anzahl der Bitstellen für die Längeninformation wächst mit der Anzahl der zu unterscheidenden Formate. Auf der anderen Seite werden die Bitstellen für die eigentliche Information besser ausgenutzt, je mehr Längenformate zur Verfügung stehen.

Tabelle 2 basiert auf der einfachen Darstellungsform, daß die Anzahl der Bitstellen gleich der Längeninformation multipliziert mit 16, 8, ... bzw. 1 Bit ist. Für die Auftrittshäufigkeit ist unterstellt, daß Konstanten, die mindestens 1

Bit benötigen so oft wie Konstanten, die 2, 3, ... bzw. 32 Bit benötigen, auftreten. Unter dieser Annahme ist eine byte-weise Abstufung der Längenformate sinnvoll. Die mittlere Codelänge beträgt 12 Bit.

Anzahl der Formate	Bitstellen für den Längencode	Bitstellen für die eigentliche Information	mittlere Gesamtlänge
2	5 Bit	2 oder 4 Byte	25 Bit
4	4 Bit	1, 2, 3 oder 4 Byte	12 Bit
8	3 Bit	2, 4, 6, 8, ... oder 32 Bit	12 Bit
16	2 Bit	4, 8, 12, 16, ... oder 32 Bit	12,5 Bit
32	1 Bit	1, 2, 3, 4, ... oder 32 Bit	13,25 Bit

Tabelle 2: Dartellung von Direktwerten und mittlere Codelänge

Die tatsächlich Verteilung der Direktwertlänge weicht von der in Tabelle 2 unterstellten Gleichverteilung ab. Kurze Direktwerte werden häufiger genutzt als lange Direktwerte. Für Direktwerte, die länger als 16 Bit sind, gibt es, da sie in herkömmlichen RISC-Code nur durch Befehlsfolgen nachgebildet werden können, kein verläßlichen Zahlen. Wir müssen erst einmal representative Längenverteilungen ermitteln, die nicht durch die 16-Bit-Grenze verzerrt sind, und anschließend optimale Codierungsformen suchen.

Huffman-Kodierung für Registeradressen

Der zweite Kandidat für einer Codierung mit variabler Länge sind die Registeradressen. Die Analyse von Trace-Files zeigt, daß die Nutzungshäufigkeit der einzelnen Register stark voneinander abweicht (Bild 4). Mit einer Huffman-Kodierung läßt sich der Code-Aufwand je Registeradresse von 5 Bit auf im Mittel 3,5 Bit verringern [20].

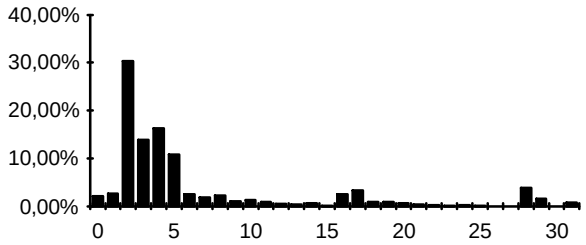

Bild 4: Registernutzung des Prozessors MIPS R4000 bei der Ausführung des Programms gzip

Die Ursache liegt im Compiler. Er reserviert jedes Register für eine spezielle Funktion. Die in Bild 4 dargestellte Verteilung ist deshalb keine Besonderheit des Programmes gzip, sondern sie ist bei allen mit dem selben Compiler übersetzten Programmen in ähnlicher Weise wiederzufinden.

Eine interessante Erweiterung, die wir bisher noch nicht untersucht haben, ist, Registeradressen in Abhängigkeit vom Verwendungszweck zu codieren. Die Idee beruht auf der Beobachtung, daß ein Teil der Register vorwiegend für Adressen und andere vorwiegend für Daten genutzt werden. Ob ein Operand ein Datum oder eine Adresse darstellt, ist wiederum mit hoher Sicherheit aus dem Befehlswort abzulesen. Mit je einer Umsetztabelle für "wahrscheinlich eine Adresse" und für "wahrscheinlich keine Adresse" läßt sich eventuell die mittlere Bitanzahl je Registeradresse noch weiter verringern.

Eine zweite Überlegung, die auch noch nicht zu Ende geführt ist, betrifft eine Vergrößerung der Registeranzahl über den zur Zeit üblichen Wert von 32 hinaus. Mit einer Huffman- oder einer vergleichbaren Codierung wird die Codelänge weniger von der Anzahl der adressierbaren Register, sondern von der Anzahl der überwiegend genutzten Register bestimmt. Zusätzliche Register erlauben mehr Daten und Konstanten im Prozessor zu halten. Es wird weniger Spill-Code für die temporäre Auslagerung von Registerinhalten benötigt. Offen ist die Frage, in welchem Umfang sich diese zusätzlichen Register nutzen lassen, bzw. welche Änderungen im Compiler dafür notwendig sind.

Implizite Registeradressen

Die meisten 8-Bit-Prozessoren verwenden den sogenannten Akkumulator als Ergebnisregister und auch als Register für einen Operanden. Die `pop`- und `push`-Befehle zum Aus- und Einlagern von Daten aus dem Stack verwenden

implizit den Stackpointer als Quell- und Zielregister. Durch die implizite Festlegung bestimmter Register als Operandenquelle oder Ziel brauchen deren Adressen nicht im Befehlswort verschlüsselt werden. Der zu übertragende Befehlsdatenstrom reduziert sich.

In RISC-Prozessoren wird auch teilweise die Technik der impliziten Registeradressen angewandt z.B. für:

- das Register, in dem die Rücksprungadresse bei `call`-Befehlen abgelegt wird
- das Flag und Statusregister als zweites Zielregister für Operationen, die Flag-Werte beeinflussen, bzw. als Quellregister für Verzweigungen.

Aus der Sicht der Befehlsdatenstromoptimierung ist es sinnvoll über die Wiedereinführung einiger Befehle mit impliziten Operanden z.B. aus der 8-Bit-Prozessortechnik nachzudenken und Simulation mit so erweiterten Befehlsarchitekturen durchzuführen.

Implizites Ergebnisregister

Typisch für RISC-Programme sind Teilbefehlsfolgen, in denen das Ergebnis einer Anweisung ausschließlich als Operand für die Folgeanweisung dient und danach nicht mehr benötigt wird (Bild 5). Einige Ergebnisse werden noch in einem kurzen darauffolgenden Intervall benutzt. Operanden mit einer langen Lebensdauer in Registern werden sehr selten berechnet. (Wir planen noch Experimente zur Abschätzung der Verteilung der Lebensdauer von Registerinhalten.)

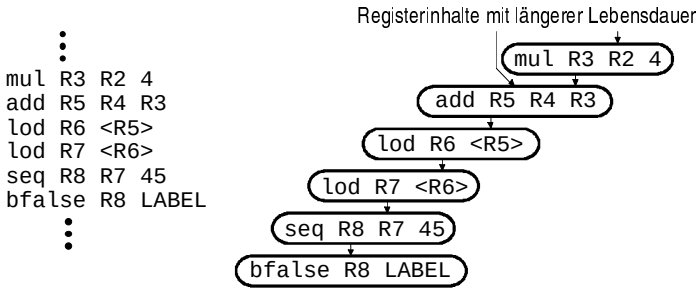

Bild 5: RISC-Befehlsfolge, in der jedes Ergebnis nur als Operand für die nachfolgende Operation dient

Für temporäre Daten mit kurzer Lebensdauer wäre ein Schieberegister mit wahlfreiem Lesezugriff der ideale Zwischenspeicher. Die Ergebnisse werden nacheinander in das Schieberegister geladen und wandern in jedem Befehlsschritt eine Position weiter. Ihre Adresse, auf der sie gelesen werden können, ist gleich dem zeitlichen Versatz zwischen Erzeugung und Nutzung abzüglich der Pipeline-Verzögerung für die Berechnung.

Ein Schieberegister kann nicht alle Aufgaben des Registerfiles übernehmen. Für Daten mit langer Lebensdauer wie globale Konstanten wird zusätzlich ein herkömmlicher Registersatz mit wahlfreiem Schreib- und Lesezugriff benötigt. Hinzu kommt eine Datenpfad zum Kopieren von Ergebnissen aus dem Schieberegister in permanente Register (Bild 6).

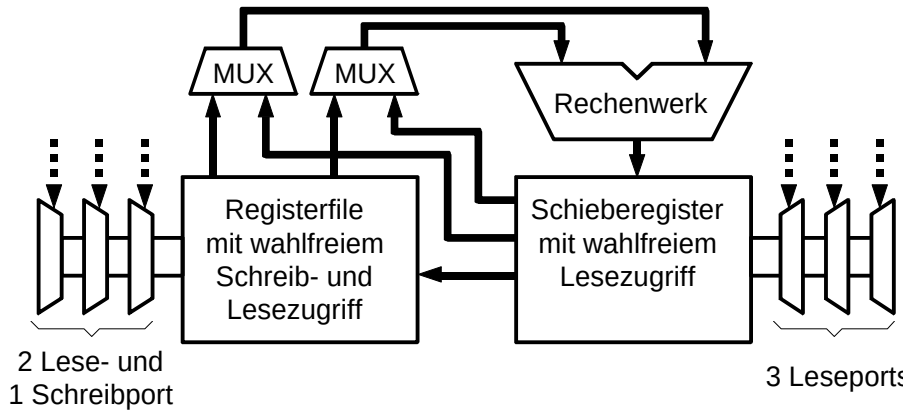

Bild 6: Teilung der Prozessorregister in einen seriell und einen wahlfrei beschreibbaren Teil

Die Teilung des Registersatzes in Schieberegister und temporäre Register besitzt neben der Code-Reduzierung weitere interessante Vorteile gegenüber der konventionellen Registerarchitektur. Die Operandenumleitung zur Vermeidung von Pipeline-Verlusten durch Datenabhängigkeiten (forwarding) und die dazu erforderliche Speziallogik wird

überflüssig (vgl. auch Abschn. 3). Im Abschnitt 8 wird weiterhin gezeigt, daß mit dieser Architektur das Umbenennen von Registern für die spekulative Außer-der-Reihe-Programmausführung überflüssig wird, und daß spekulativ berechnete Ergebnisse automatisch verfallen, wenn sie nicht gebraucht werden.

Der Operationscode

Der Operationscode ist der kleinste Bestandteil eines RISC-Befehls. Er ist folglich für die Optimierung am wenigsten interessant. Maßnahmen, wie die Einführung von 0-, 1- und 2-Adreßbefehlen und von Spezialbefehlen vergrößern die Befehlsanzahl. Ab einer bestimmten Größe des Befehlsvorrats könnte auch für den Operationscode ein Code variabler Länge interessant werden. Ein solcher Code ist auch interessant, um einen Befehlssatz abwärtskompatibel erweiterbar zu gestalten. Die Grundbefehle erhalten einen kurzen Operationscode. Später eventuell zu ergänzende Befehle werden in längere Codeworte verschlüsselt (siehe auch Abschn. 3).

In existierenden RISC-Prozessoren besitzt ausgerechnet der Operationscode, für den das aus Sicht einer kompakten Informationsdarstellen am wenigsten sinnvoll erscheint, variable Länge. Im MIPS-Befehlssatz besitzen z.B. alle Befehle ein 6-Bit-Grundoperationscode, der für 3-Registerbefehle um 6 Bit erweitert ist. Es handelt sich offenbar nicht um eine Codeoptimierung im Sinnen der Minimierung des Befehlsdatenflusses, sondern darum, die Lücken im 32-Bit-Befehlsformat zu schließen.

Kodierung von Spezialbefehlen

Das Befehlswort für einen Spezialbefehl besteht aus dem Operationscode und einer bestimmten Anzahl von Operanden (Registeradressen und Direktwerte). Die Zielarchitektur für unsere Untersuchungen ist ein Prozessor mit Verarbeitungs-Pipelines und mehreren Rechenwerken. Für die Abbildung eines Spezialbefehls auf diese Architektur gibt es die Varianten:

1. Zuweisung an eine Verarbeitungs-Pipeline, die diesen Befehl ausführen kann
2. Aufspaltung in Teiloperationen, und parallele Zuweisung an mehrere Verarbeitungs-Pipelines
3. Erzeugung einer Sequenz von Teiloperationen, die zeitlich nacheinander einer oder mehreren Verarbeitungs-Pipelines zugeordnet werden.

Die erste Variante verlangt Spezialrechenwerke. Der Spezialbefehl muß tief in der Hardware verankert sein.

Die zweite Variante benötigt einen speziellen Befehlsdecoder. Aus dem Operationscode des Spezialbefehls werden über einen Look-Up-Table (LUT) die Operationscodes für die einzelnen Verarbeitungs-Pipelines und ein Verteilerschlüssel für die Operanden erzeugt [21]. Für die Zuweisung von Operanden zu den Pipelines wird ein komplexer Verteiler benötigt. Er muß jeden der Operanden des Spezialbefehls an jeden Operandeneingang der Verarbeitungs-Pipelines vermitteln können, eine 1 auf *n* (broad cast) Vermittlung unterstützen und Pipeline-Eingängen, denen keine Operanden zugewiesen sind, mit Standardwerten belegen (z.B. Null). Der in Bild 7 skizzierte Befehlsdecoder wird in abgerüsteter Form auch für die Abarbeitung von 1- und 2-Adreßbefehlen gebraucht.

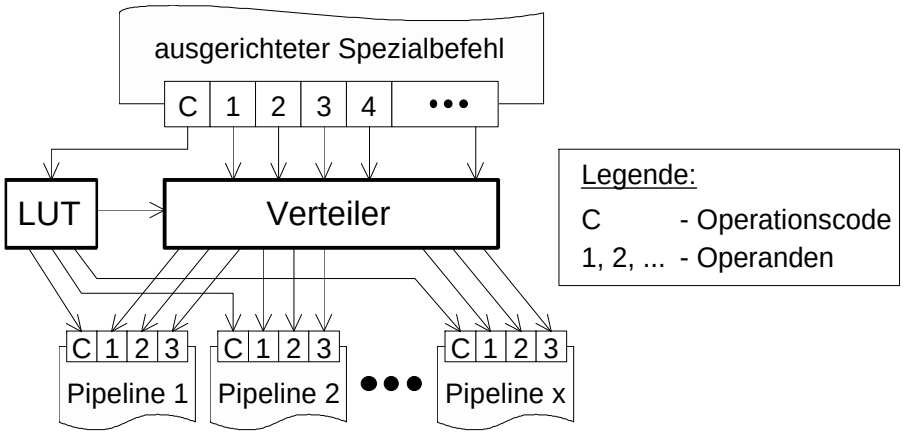

Bild 7: Prinzip der Umsetzung eines Spezialbefehls in Grundbefehle

Für Spezialbefehle, die sich auf eine Sequenz von Pipeline-Operationen abbilden, ist der Look-Up-Table in Bild 7 zu einem Mikroprogrammsteuerwerk zu erweitern (Bild 8).

Vektorbefehle, eine Sonderform der Spezialbefehle, führen eine Operation oder eine Folge von Operationen mit mehreren Daten aus (SIMD-, Single-Instruction-Multiple-Data-Verarbeitung). Im Befehlswort müssen entsprechend

viele Registeradressen codiert werden. Eine Alternative zur Befehlsdatenstromreduzierung ist der Einbau von Adreßrechnungsfunktionen in den Befehlsdecoder z.B. in Form von Zählern.

Ein Befehlsdekoder auf Mikroprogrammbasis kann im Prinzip auch einfache Schleifen ausführen. Für Schleifen muß eine Abbruchbedingung generiert werden. Dazu könnte eine zusätzliche Vergleichsschaltung dienen (Bild 8).

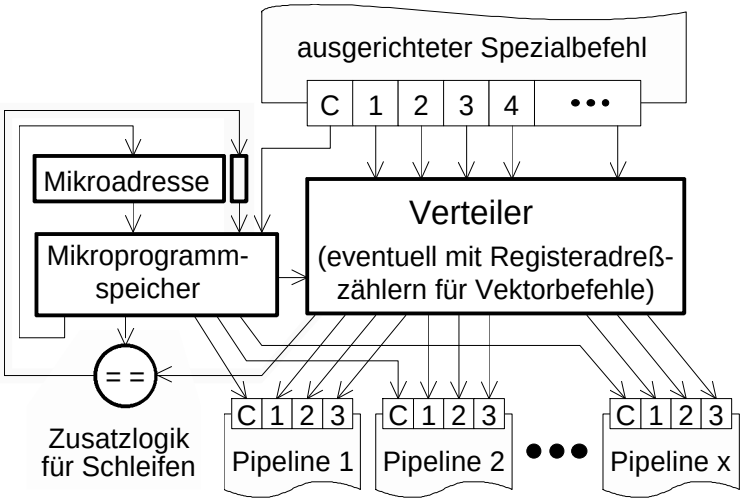

Bild 8: Erweiterung auf CISC-artige Spezialbefehle

Wir planen zu dem Thema Spezialbefehle weitere Untersuchungen. Die nächsten Schritte werden quantitative Abschätzung zur Reduzierung des Befehlsdatenstroms sein. Insbesondere ist die Gewinnabschätzung unter der Annahme interessant, daß der Speicher für den Look-Up-Tables (Bild 7) bzw. der Mikroprogrammspeicher (Bild 8) alternativ zur Vergrößerung des First-Level-Caches eingesetzt werden würde.

6 Ausrichten von Befehlen mit Elementen variabler Länge

Im vorangegangenen Abschnitt wurde eine Befehlsarchitektur entwickelt, in der alle Befehlsbestandteile (Operationscode, Registeradressen und Direktwerte) variable Länge besitzen (oder besitzen können). Die Länge jedes Bestandteils ergibt sich implizit aus seinem Code und die Anzahl der Operanden aus dem Operationscode. Eine Befehlsfolge dieser Form verlangt eine elementeweise Abarbeitung: Lesen des Operationscodes, bestimmen seiner Länge und des Beginns des ersten Operanden, lesen des ersten Operanden bestimmen seiner Länge und des Beginns des zweiten Operanden usw. (Bild 9). Die Aufbereitung eines Befehlswortes benötigt so viele Schritte wie Befehlselemente (z.B. ein 3-Adreßbefehl 4 Schritte). Die Schritte können nur nacheinander ausgeführt werden. Der entsprechende Teil des Befehlsdecoders muß mit der *n*-fachen Taktfrequenz wie der restliche Prozessor arbeiten.

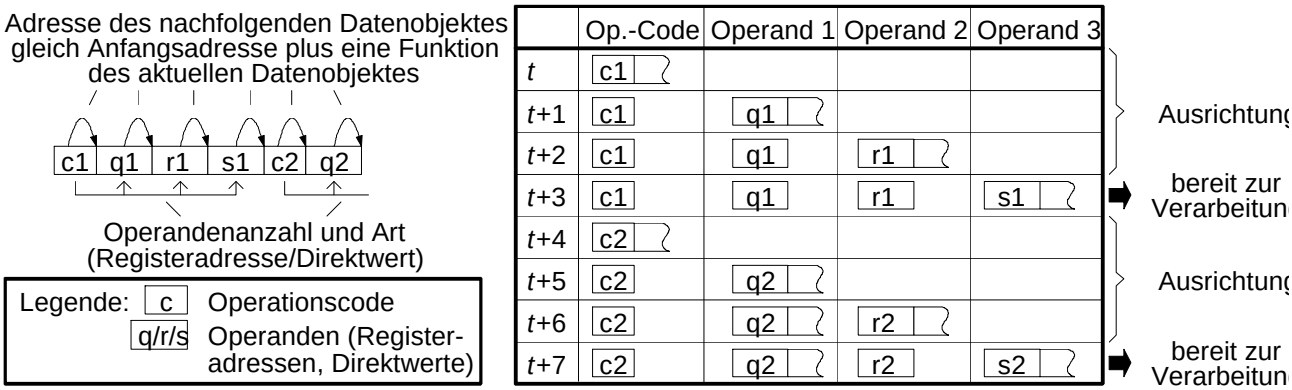

Bild 9: Aufbereiten von Befehlen mit Elementen variabler Länger

Voraussetzungen für eine Pipeline-Verarbeitung

Erwünscht ist die Extraktion der Befehlsbestandteile in wenigen Zeitstufen in einer Pipeline. Eine Pipeline-Verarbeitung ist nur möglich, wenn im ersten Befehlselement die Anfangsadresse des Nachfolgebefehls codiert ist. In

Bild 10 ist zu diesem Zweck als zusätzliches Befehlselement ein Längencode eingebaut. Die Abarbeitung beginnt mit dem Lesen des Längencodes und der gleichzeitigen Ermittlung des Beginns des Nachfolgebefehls und der Position des Operationscodes. Die weitere Aufbereitung erfolgt genau wie in Bild 9, nur daß immer um einen Schritt versetzt die Bestandteile des nächsten Befehls aufbereitet werden.

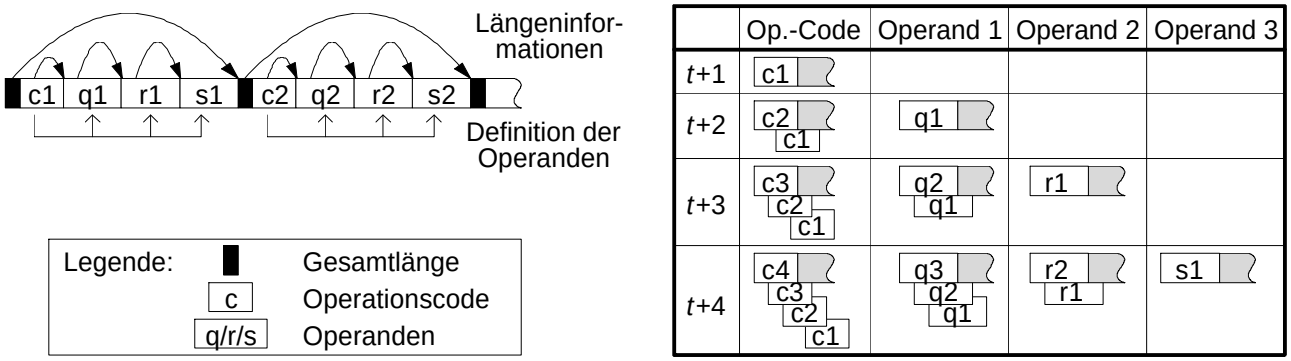

Bild 10: Einfügen einer Längeninformation zur Ermöglichung einer Pipeline-Verarbeitung

Die notwendige Pipeline-Tiefe für die Befehlsaufbereitung läßt sich durch eine geringe Modifikation des Befehlsaufbaus verringern. Der Längencode als Startpunkt für die Trennung der Befehlsbestandteile wird nicht am Anfang, sondern in der Mitte des Befehlswortes angeordnet (Bild 11). Gleichzeitig mit dem Längencode ist es nun möglich, den Operationscode zu analysieren und die Position des links vom Operationscode und die Position des rechts vom Längencode angeordneten Operanden zu bestimmen etc. Die Extraktion erfolgt vom Eintrittspunkt in Vor- und Rückwärtsrichtung, d.h. immer zeitgleich für zwei Befehlselemente. Die Anzahl der nacheinander auszuführenden Schritte halbiert sich.

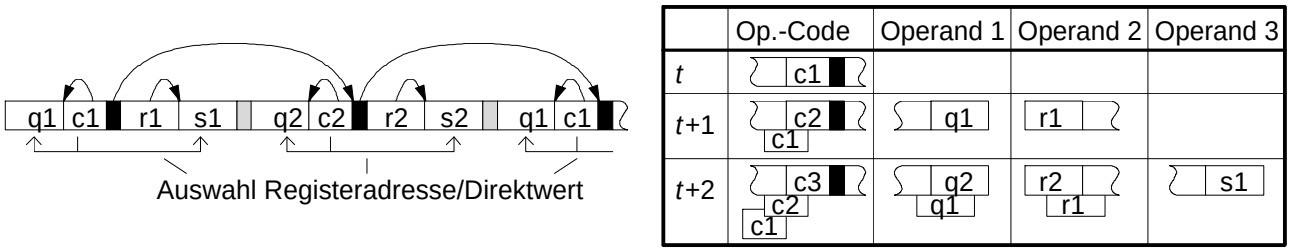

Bild 11: Halbierung der Länge der Dekodierpipeline durch Vor- und Rückwärtsverzeigerung

Der Längencode

Der Längencode ist ein Code-Overhead, der für die Pipeline-Verarbeitung zu zahlen ist. In ihm können verschlüsselt sein:

- die Länge des Befehls für die normale Verarbeitung
- eine Sprungdistanz oder ein Sprungziel für Sprünge und Unterprogrammaufrufe
- der Verweis auf ein Register, in dem das Sprungziel für die Rückkehr aus Unterprogrammen steht (Registeradresse, implizierte Annahme eines speziellen Registers).

Für die Längeninformation ist ein Code mit minimaler mittlerer Codelänge zu entwickeln, aus dem in einem Takt Codelänge und Folgeadresse extrahiert werden können. Wir planen hierzu Experimente zur Bestimmung der relativen Häufigkeit der Längeninformationstypen und die Suche geeigneter Codestrukturen. In Analogie zur Codierung von Dirktwerten wird es eventuell günstiger sein, Längenangaben auf Vielfache von 2, 4 oder 8 Bit zu runden und zu Gunsten eines kürzeren Längencodes Ausrichtungsverluste zu akzeptieren.

Schaltungsstruktur zur Trennung der Befehlsbestandteile

Bild 12 veranschaulicht den prinzipiellen Aufbau des Befehlsdekoders. Der Ausschnitt aus dem Befehlsdatenstrom, auf den die Wortadresse zeigt, wird in Eingangsregister geladen (mehrere Befehlsworte, das adressierte Wort in das mittlere Register). Ein Weiterschalten der Wortadresse führt automatisch zum Verschieben der Befehlsworte im Eingangsregister und zum Nachladen aus dem Cache. Ein Block-Shifter richtet die Eingangsdaten entsprechend der Bitadresse aus. Aus dem ausgerichteten Ausschnitt aus dem Befehlsdatenstrom werden über Tabellenfunktionen der

Startpunkt des nächsten Befehls, der Operationscode, und die relativen Verschiebungen vom Startpunkt bis zum ersten und bis zum zweiten Operanden bestimmt. Diese Daten werden im Befehlsregister bzw. in Pipeline-Registern zwischengespeichert.

In der zweiten Pipeline-Stufe werden mit Hilfe von zwei weiteren Block-Shiftern und zwei weiteren Tabellenfunktionen die ersten beiden Operanden extrahiert. Weitere Pipeline-Stufen extrahieren je paarweise weitere Befehlselemente.

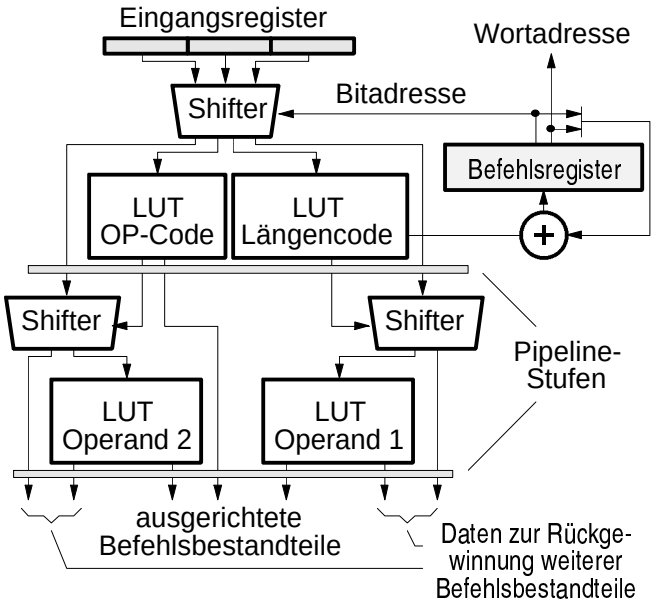

Bild 12: Prinzip der Extrahierung der Befehlsbestandteile für eine Befehlsarchitektur nach Bild 11

Wir haben bisher noch keine genaueren Untersuchungen zum Hardware-Aufwand für die Extraktion der Befehlsbestandteile durchgeführt. Nach groben Schätzungen ist er sicher nicht höher als einige 10.000 bis 100.000 Gateräquivalente. Das ist für die heutigen und erst recht für zukünftige Prozessoren ein vertretbarer Zusatzaufwand.

Konsequenzen aus der Verlängerung der Befehlshole-Pipeline

Das angedachte Konzept zur Verarbeitung von Befehlen mit Bestandteilen variabler Länge verlängert die Befehlshole-Pipeline. Die Extraktion des ersten Befehlscodes und des Startpunktes für die Nachfolgeoperation kostet eine Pipeline-Stufe. Jedes Paar von weiteren Befehlsbestandteilen verlangt eine weitere Stufe. Für einfache RISC-Befehle mit vier Befehlsbestandteilen ist das sicher zu tolerieren. Die Aufbereitung von VLIW-artigen Befehlen mit 4*n* Bestandteilen verursacht erhebliche Verzögerungen.

Verzögerungen in der Befehlsaufbereitung führen zu Leistungsverlusten bei Verzweigungen. Ein wesentlicher Lösungsgedanke zur Minimierung dieser Verluste ist bereits im Unterabschnitt "Längencode" dargelegt. Im Längencode muß die Entfernung zum oder die Adresse des als nächstes abzuarbeitenden Befehls codiert sein. Das kann der nachfolgende Befehl oder ein Sprungziel sein. Auf diese Weise wirkt sich die Verzögerung in der Befehlsaufbereitung nicht auf eine Verzögerung bei der Ausführung von unbedingten Sprüngen und von Prozedure-Aufrufen aus.

Bei Verzweigung in der unterstellten Vorzugsrichtung treten auch keine Verluste auf. Stellt sich die spekulativ ausgeführte Verzweigungsrichtung jedoch nachträglich als falsch heraus, stehen die Zeitscheiben für das Ausrichten der Befehle des falschen Programmzweiges als Verlust zu Buche. Unser Architekturansatz verlangt eine gute Vorhersage von Verzweigungsrichtungen. Einschlägige Konzepte für die Verzweigungsvorhersage (vgl. Abschn. 3) sind hinsichtlich ihrer Irrtumsrate zu untersuchen. Auf der anderen Seite sind Befehlskonzepte, die die Pipeline-Tiefe verringern, gefragt. Ansätze sind:

- Begrenzung der Anzahl der Befehlsbestandteile: Ungeachtet der tatsächlichen Anzahl der Befehlsbestandteile müssen alle Befehle eine konstante Anzahl von Pipeline-Stufen durchlaufen. Eine Begrenzung auf etwa 4 bis 6 Pipeline-Stufen (8 bis 12 Befehlsbestandteile in der Grundvariante) ist unumgänglich.
- Begrenzung der Anzahl der Befehlsbestandteile variabler Länge: Bestandteile konstanter Länge können bei geeigneter Befehlsarchitektur in einer Pipeline-Stufe gemeinsam extrahiert werden. Die Kombination von

Bestandteilen variabler mit Bestandteilen konstanter Länge würde erlauben, in einer begrenzten Anzahl von Pipeline-Stufen komplexere Befehle mit mehr Elementen aufzubereiten.

- Trennung der Längeninformation von der eigentlichen Information: Die Längeninformationen (konstanter Länge) werden unmittelbar am Startpunkt angeordnet und in der ersten Pipeline-Stufe parallel ausgewertet. In der zweiten Pipeline-Stufe stehen dann die Anfangsadressen von mehreren Befehlselementen gleichzeitig zur Verfügung. Die Extraktion kann parallel erfolgen.
- Reihenfolge der Extraktion: In einer typischen Verarbeitungs-Pipeline für Festkomma-Werte (Bild 1) werden im ersten Takt die Registeradressen für Operanden, im zweiten Takt Direktwerte und der Code zur Operationsauswahl und im dritten Takt die Adresse für das Ergebnis benötigt. Die Verarbeitung könnte damit bereits nach der Extraktion der ersten Befehlsbestandteile beginnen. Mit einem entsprechenden Befehlsaufbau dürfen sich Extrahierung und Verarbeitung um bis zu zwei Zeitebenen überlagern.
- Mehrere Startpunkte je Befehlswort: Dieses Konzept ist nur für sehr lange Befehlsworte (VLIW, very long instruction word) interessant. Das Befehlswort erhält zwei oder allgemein *n* Startpunkte für die Extrahierung. An den Startpunkten ist jeweils ein Längencode angeordnet, der auf die Startpunkte des nachfolgenden Befehls verweist. Gleichzeitig können in der ersten Pipeline-Stufe *n* und in jeder weiteren Pipeline-Stufe 2*n* Befehlselemente variabler Länge extrahiert werden.

  Zur Extrahierung mit *n* Startpunkten werden *n* Werke nach Bild 12 benötigt. Interessant an diesem Ansatz ist, daß die Hardware-Funktionen zur Befehlsaufbereitung von mehreren Startpunkten aus viele Funktionen für eine Multithread-Verarbeitung bzw. die Ausbeutung grobkörniger Parallelität zur Verfügung stellen (Bild 13).

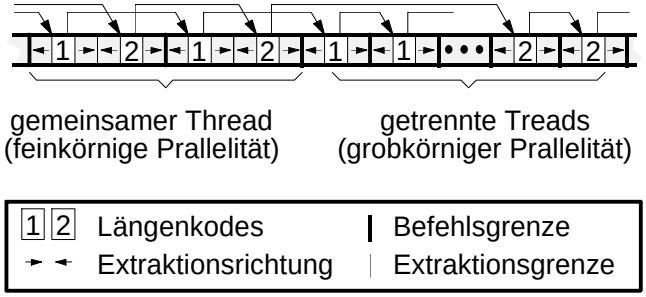

Bild 13: Mehre Startpunkte je Befehl und Multithread-Verararbeitung

Die Aufzählung zeigt insgesamt, daß zum Thema Befehlscodierung noch zahlreiche Untersuchungen erforderlich sind.

7 Ausnutzung programminterner Pararallelität

Von den beiden Formen der Parallelverarbeitung

- auf Funktions- (bzw. Thread-) Niveau (grobkörnige Parallelität)
- auf Befehlsebene (feinkörnige Parallelität)

nutzen die heutigen RISC-Prozessoren nur die feinkörnige Parallelität. Sie besitzen mehrere Rechenwerke, aber nur ein Werk zur Steuerung des Programmflusses (Befehlshole- und Sprunglogik). Zur Ausbeutung feinkörniger Parallelität genügt eine einfacherer Prozessorarchitektur. Diese Aussage gilt ohne Zweifel für eine Parallelverarbeitung von maximal 2 bis 3 Operationen. Ein höherer Grad an Parallelverarbeitung, wie er von den heutigen Prozessoren angestrebt wird, verlangt wesentlich mehr als eine Vervielfachung der Rechenwerke (vgl. Abschn. 3).

Ausnutzung feinkörniger Parallelität

Mehrere Verarbeitungs-Pipelines und in einem gewissen Fenster auch aufeinanderfolgende Zeitscheiben dieser Pipelines können nur in dem Maße genutzt werden, wie der abzuarbeitende Programmausschnitt unabhängige Anweisungen enthält, d.h. Anweisungen, die ihre Ergebnisse untereinander nicht als Operanden verwenden und zwischen denen keine Verzweigungen liegen. Die meisten Programmen sind so aufgebaut, daß die Verarbeitungs-Pipelines nur in geringem Maße ausgelastet werden können. Zur Verbesserung der Auslastung wird der abzuarbeitende Befehlscode optimiert (vgl. Abschn. 3}:

- Veränderung der Reihenfolge der Anweisungen in verzweigungsfreien Programmsegmenten

- Aufrollen von Schleifen (Vergrößerung der verzweigungsfreien Programmsegmente, innerhalb derer die Befehlsreihenfolge variiert werden kann)
- Ersetzen von Prozedur-Aufrufe durch den Prozedur-Körper (Verringerung des Organisations-Overheads im Befehlsdatenfluß und Vergrößerung der verzweigungsfreien Programmsegmente)
- spekulative Befehlsausführung über Verzweigungsgrenzen hinweg.

Das Aufrollen von Schleifen und das Ersetzen von Prozedur-Aufrufen durch den Prozedur-Körper verbessert die Auslastung der Verarbeitungs-Pipelines auf Kosten eines höheren Befehlsdatenstroms. Die Anzahl der nacheinander auszuführenden Verarbeitungsschritte verkürzt sich, aber es fallen mehr Wartezeiten an, weil mehr Befehlscode in den Cache zu laden ist. Mit der zunehmenden Kluft zwischen Prozessor- und Speichergeschwindigkeit wächst der Einfluß der Wartezeit auf die Rechenleistung. Es sind kaum noch ein Rechenleistungsgewinn zu erzielen.

Spekulative Befehlsausführung

Dieses Verfahren verbessert die Auslastung der Zeitscheiben und Rechenwerke, ohne daß sich der Befehlsdatenstrom stark erhöht. Es ist deshalb für neue, schnelle Prozessoren von zentraler Bedeutung. Die spekulative Befehlsausführung verlangt aber eine Unterstützung durch die Hardware, den Compiler und die Befehlsarchitektur.

Die vorgezogen Ausführung von Anweisungen über Verzweigungsgrenzen hinweg bedeutet, daß der Prozessor anders nicht nutzbare Zeitscheiben für Berechnungen nutzt, die mit gewisser Wahrscheinlichkeit später gebraucht werden. Zu den Berechnungen eines RISC-Prozessors gehören z.B. auch Adreßrechnungen, das Laden von Operanden, d.h. Aktionen, auf die die nachfolgenden Anweisungen warten müssen. Im positiven Fall, wenn die spekulativ berechneten Ergebnisse im nachfolgenden Programmfluß gebraucht werden, verkürzt sich die Rechenzeit. Erfolgen die Verzweigungen in anderer Richtung, so daß der ursprüngliche Platz der vorgezogenen Anweisung nicht erreicht wird, wurde das Ergebnis umsonst berechnet. Es wird verworfen. Der Anhang zeigt ein Beispiel für die Codeoptimierung durch spekulative Befehlsausführung.

Superskalare Prozessoren nutzen Hardware-Algorithmen, um aus einem sequentiellen Programm die vorzuziehernden Anweisungen auszuwählen. Im Gegensatz zu compiler-gesteuerten Optimierungen wie dem Aufrollen von Schleifen vergrößert sich der Code nicht. Die stärkere Parallelverarbeitung wird nicht durch Wartezeiten auf den Speicher erkauft.

Die Hardware-Steuerung für die "außer-der-Reihe-Ausführung" von Anweisungen der superskalaren Architektur ist aufwendig, die Gesamtarchitektur kompliziert und die prozessorinternen Vorgänge sind für den Programmierer schwer zu überblicken. Das Konzept bietet kaum noch Ansätze für Erweiterungen (vgl. Abschn. 3).

Eine softwaremäßige Codeoptimierung ist flexibler und erzielt einen höheren Grad an Parallelverarbeitung. Sie vergrößert jedoch den Codeumfang. Das Verschieben eines Befehls über Vereinigungspunkte im Programmfluß (Ansprungstellen) hinweg verlangt, daß die Anweisung auf jedem Weg ausgeführt wird. Sie erscheint mehrfach im Code. Darüber hinaus müssen mehr Informationen im Befehlscode verschlüsselt sein:

- Größere Registeradressen: In superskalaren Prozessoren werden die Registeradressen des Programmiermodells dynamisch auf eine wesentlich größere Anzahl physisch vorhandender Register umgesetzt. Der Compiler muß diese Zuordnung explizit verschlüsseln und benötigt insgesamt mehr Bitstellen für Registeradressen.
- Gültigkeit vorgezogener Anweisungen: Ein superskalarer Prozessor erhält die Befehle in ihrer ursprüglichen Reihenfolge. Diese Information nutzt er, um spekulative Rechnungen für Programmzweige, die nicht erreicht werden, zu verwerfen und um dem Programmierer die ursprüngliche Abarbeitungsreihenfolge vorzutäuschen. Der Compiler muß die Information, an welcher Programmstelle eine spekulativ ausgeführte Anweisung gültig ist, explizit codieren.
- Fortsetzung nach Unterbrechungen (Interrupts, Traps): Ein superskalarer Prozessor benötigt zur Fortsetzung eines Programms nach einer Unterbrechung nur den Inhalt der für den Programmierer sichtbaren Register und den für den Programmierer sichtbaren Befehlszählerstand. Ausgehend davon können, wenn auch auf Kosten einer gewissen Anlaufzeit, alle verborgenen Registerzustände neu berechnet werden. Die interne Abarbeitungsreihenfolge wird entsprechend des Startpunktes neu gewählt. Bei Codierung spekulativ auszuführender Befehle durch den Compiler kann dieses Verhalten nur mit zusätzlichen Informationen im Befehlscode nachgebildet werden. Für jeden möglichen Unterbrechungspunkt muß codiert sein, welche Ergebnisse schon sichtbar sein dürfen und entsprechend nach der Unterbrechung wiederhergestellt werden müssen. Und es wird für jeden zulässigen Unterbrechungspunkt ein spezieller Anlaufcode benötigt, in dem die im eigentlichen Code spekulativ bereitgestellten Ergebnisse neu berechnet werden.

Den quantitativen Einfluß der zusätzlichen Befehle auf den Befehlsdatenstrom haben wir bisher nicht untersucht. Der Code-Overhead durch die größere Anzahl der explizit zu adressierenden Register läßt sich nach unseren Untersuchungen in Abschn. 5 sehr gering halten.

Probleme bereiten die Informationen, die benötigt werden, um bei Unterbrechungen den Eindruck der ursprünglichen Abarbeitungsreihenfolge zu wahren. Der Prozessor kann die veränderte Befehlsreihenfolge einschließlich der spekulativ ausgeführten Befehle wahrscheinlich nicht mit einem tolerierbaren Code- und Rechenzeit-Overhead verbergen. Es sind alternative Konzepte gefragt. Gegebenenfalls muß eine Software-Schicht über den Prozessor gelegt werden, die die korrekte Unterbrechungsausführung sicherstellt und die dem Programmierer das gewohnte Prozessormodell beim "normalen" Debuggen vortäuscht.

Ausnutzung grobkörniger Parallelität

Die Ausnutzung von Parallelität auf Befehlsniveau rechtfertigt keine größere Anzahl von Rechenwerken. Es gibt nur einen plausiblen Ansatz, die Umfang der Parallelverarbeitung weiter zu erhöhen. Das ist die zeitgleiche Bearbeitung mehrerer Aufgaben. Leistungsfähige Betriebssysteme wie UNIX unterstützen seit vielen Jahren die gleichzeitige Arbeit mit mehreren Programmen. Der Prozessor arbeitet dazu, gesteuert vom Betriebssystem, jeweils zyklisch einige Millisekunden an jeder Aufgabe. Wenn es gelingt, einen Prozessor zu konstruieren, der zeitgleich mehrere Programmflüsse verfolgen kann und der den einzelnen Threads dynamisch Zeitscheiben und Rechenwerke zuordnet, wäre das Problem der Auslastung zusätzlicher Rechenwerke für mehrere Generationen neuer Prozessoren gelöst.

Welche Hardware-Erweiterungen wären erforderlich?

Die Multi-Thread-Erweiterung ist genau wie die Befehlsdatenstromoptimierung ein unkonventioneller Ansatz. Bei einem genaueren Blick auf die Hardware der heutigen Prozessoren ist sie keineswegs unrealistisch. Ein Multithread-Prozessor benötigt folgende Bausteine:

- Befehls-Cache mit mehren Lese-Ports: Schnelle Caches mit wahlfreinem Zugriff für mehre Werke gibt es z.B. in Form eines kombinierten Befehls- und Daten-Cache. Verwenden die Treads den selben Code, wie es bei der Zerlegung einer Folge von Schleifendurchläufen in mehrere Threads der Fall ist, genügt die bisherige Cache-Größe. Für Threads auf der Ebene unterschiedlicher Programme entsteht kein Nachteil, wenn die Befehlsholeeinheiten mit unterschiedlichen Caches arbeiten.
- Mehrere Werke für die Verfolgung des Programmflusses (Befehlszähler und Sprunglogik): Für die Abarbeitung von Befehlen mit Bestandteilen variabler Länge, mit der wir uns innerhalb des Forschungsprojektes beschäftigen, werden ab einer bestimmten Anzahl von parallel nutzbaren Rechenwerken ohnehin mehrere solche Werke benötigt (vgl. Abschn. 6).
- Eine Einheit, die den einzelnen Threads Rechenwerke und Zeitscheiben zuweist: Die Algorithmen werden sicher nicht so kompliziert sein wie die Veränderung der Ausführungsreihenfolge in Superskalaren Prozessoren.
- Mehrere Rechenwerke einschließlich von Load-/Store-Werken: Das ist auch für die Ausbeutung feinkörniger Parallelität erforderlich.
- Daten-Cache
- Eine größere Zahl von Arbeitsregistern: Auch das ist für die Ausbeutung feinkörniger Parallelität erforderlich.
- Eine Einheit, die die Zuordnung von Thread-Registern zu physischen Registern vornimmt: Dieser Algorithmus ist einfach im Vergleich zur Umbenennung von Registern in den heutigen Superskalaren Prozessoren.

Der Sprung von den heutigen Superskalaren Prozessoren ist nicht mehr groß. Auf der anderen Seite erlaubt eine Multithread-Architektur auch Vereinfachungen:

- Weniger Aufwand für die Ausbeutung feinkörniger Parallelität: Es wäre unsinnig, Hardware in die Veränderung der internen Abarbeitungsreihenfolge zu investieren und damit die Auslastung der Rechenwerke um einige Prozent zu erhöhen, wenn die gleichen Ressourcen auch für andere Treads genutzt werden können.
- Weniger Aufwand für die Vorhersage der Verzweigungsrichtung: Wenn die Verzweigungsrichtung nicht mit einer Sicherheit von angenommen mindestens 80% vorhersagbar ist, wird nicht spekulativ verzweigt. Die Zeitscheiben, bis die tatsächliche Verzweigungsrichtung berechnet ist, werden von anderem Thread genutzt.
- Einfachere Behandlung von Unterbrechungen (Interrupts, Traps): Ein Multithread-Prozessor muß keine oder nur einen Teil seiner Ressourcen bei einer Unterbrechung freigeben. Im einfachsten Fall, wenn die aktiven Threads noch genügend Register und ein Steuerwerk freigelassen haben, erzeugt die Unterbrechung nur einen neuen Thread. Bei Ressourcen-Mangel muß zuvor eine Betriebssystemroutine, die ein priveligierter Thread mit fest zugeordneten Resourcen sein müßte, einen anderen Thread auslagern. Vor allem Pages-Segment-Faults, ein großes Problem für Prozessoren mit Pipeline-Verarbeitung und virtueller Adressierung, lassen sich mit einem zweiten Thread viel eleganter beheben als durch herkömmliche Unterbrechungskonzepte.
- Längere Befehlhole- und -dekodier-Pipeline: Eine Multithread-Architektur erlaubt es in vielen Fällen, Verzweigungen und Unterbrechungen ohne das Löschen von Pipeline-Zuständen auszuführen. Damit reduzieren sich die Verluste an Rechenleistung, die durch zusätzliche Pipeline-Stufen verursacht werden. Zusätzliche

Pipeline-Stufen für den Cache-Zugriff würden eine Vergrößerung des First-Level-Caches erlauben. Bei der Verarbeitung von Befehlen mit variablen Bestandteilen erlaubt eine längere Pipeline, Befehle mit mehr Befehlsbestandteilen variabler Länge aufzubereiten.

Ein Multithread-Prozessor muß aus dieser Sicht nicht komplizierter und größer als die heutigen Superskalaren Prozessoren sein. Vor allem eröffnet das Konzept wieder Möglichkeiten weitere Hardware in höhere Rechenleistung umzusetzten.

Tiefgreifende Untersuchungen zu einer Multithread-Erweiterungen und zu alternativen Unterbrechungskonzepten sprengen natürlich den Rahmen unseres Forschungsvorhabens. Aber wir müsse diese Optionen in unsere Untersuchungen für zukünftige Befehlsarchitekturen mit einbeziehen.

8 Einige Gedanken zur weiteren Entwicklung der RISC-Prozessoren

Die Entwicklung der Prozessoren hat einen toten Punkt erreicht. Wie in Bild 2 am Ende von Abschnit 3 gezeigt, werden die Prozessoren zwar immer größer und die Taktfrequenz nimmt zu. Aber die nutzbare Rechnenleistung erhöht sich nur noch spärlich. Die Suche nach lokalen Ansätzen für die Optimierung der Prozessoren wie die Optimierung des Befehlsdatenstroms ist ohne Zweifel wichtig. Dringender ist natürlich die Suche nach neuen Prozessorarchitekturen. Wie bereits in den vergangen Abschnitten dargestellt, sind wir im Rahmen unserer Untersuchungen auf mehrere interessante Ansätze gestoßen, die hier noch etwas weitergeführt werden sollen.

Kompatibilität

Ein wichtiges Auswahlkriterien für Prozessoren, wenn ein neues Produkt entwickelt wird, ist Abwärtskompatibilität zu etablierten, älteren Prozessoren. Die Software ist i.allg. der aufwendigere Teil der Produktentwicklung. Deshalb ist es sehr wichtig, daß existierende Programme und Programmstücke wieder verwendet werden können. Das Preis-Leistungsverhältnis des Prozessors selbst ist von zweitrangiger Bedeutung.

Die Forderung nach Binärkompatibilität hat u.a. dazu geführt, daß moderne Prozessoren mit Pipelines und mehreren Rechenwerken dafür optimiert sind, Code abzuarbeiten, der ursprünglich für einen Prozessor mit einem einfachen Rechenwerk ohne Pipeline, einem eng begrenzten Adreßraum etc entwickelt wurde. Das Paradebeispiel ist die Intel-Line vom 8086 bis zum P6. Der Superskalare 32-Bit-Prozessor P6 ist u.a. für die Abarbeitung alter 8-Bit-Programme ausgelegt.

Auch in Zukunft müssen die Prozessoren über mehrere Generationen hinweg abwärtskompatibel sein. Interessant wäre eine Untersuchung, ob nicht auch der umgekehret Weg möglich ist. Es wird ein idealer Prozessor definiert, der z.B. unbegrenzt viele Rechenwerke besitzt. Dafür wird ein erweiterbarer Befehlssatz entwickelt. Die ersten Prozessoren sind nicht größer als die heutigen Superskalaren Prozessoren. Mit der weiteren Entwicklung der integrierten Schaltungstechnik wachsen sie in dieses Modell hinein. (Gegenwärtig könnte man sagen, daß die Prozessoren aus ihrem Programmiermodell herauswachen.)

Ein Befehlssatz mit Bestandteilen variabler Länge ist für diesen Ansatz ideal. Er erlaubt die Rückkehr zu einem kleinen orthogonalen Satz von Grundbefehlen. Das war eine der ursprünglichen Ideen der RISC-Prozessoren, die nicht nur den Prozessor, sondern auch die Programmierung vereinfacht (vgl. Abschn. 2). Mit variablen Befehlsbestandteilen läßt sich diese Idee weiter ausbauen. Jede Operation darf 0, 1, 2, 3 und mehr Operanden besitzen. Direktwerte unterliegen nicht mehr der starren 2-Byte-Grenze wie bei herkömmlichen RISC-Befehlen. Dadurch entfallen einige Tricklösungen in der Befehlsarchitektur.

Auf der anderen Seite ist ein variabler Befehlssatz offen, um bei Bedarf Spezialbefehle für bestimmte Anwendungen wie die Bildverarbeitung zu ergänzen (vgl. Abschn. 7).

Der skalierbare Prozessor

Ein Prozessorkonzept, daß die Integration einer wachsenden Anzahl von Rechenwerken unter Wahrung der Binärkompatibiliät erlaubt, muß im Grundansatz grob- und feinkörnige Parallelverarbeitung unterstützen. Es benötigt ein Konzept, um Rechnewerke und Zeitscheiben nicht nur zu 20% oder 30%, sondern möglichst zu 80% bis 90% auszulasten. Dazu gehört die Vermeidung und Nutzung von Wartezeiten auf den Hauptspeicher (einschließlich der Optimierung des Befehlsdatenstroms). Die Gesamtarchitektur darf, um hohe Taktfrequenzen zu erzielen, aber auch nicht zu kompliziert sein. Bild 14 zeigt einen Ansatz für einen solchen Prozessor, der aus konventionellen Prozessorbestandteilen und aus Bestandteilen, die in den vorhergehenden Abschnitten entwickelt wurden, zusammengesetzt ist.

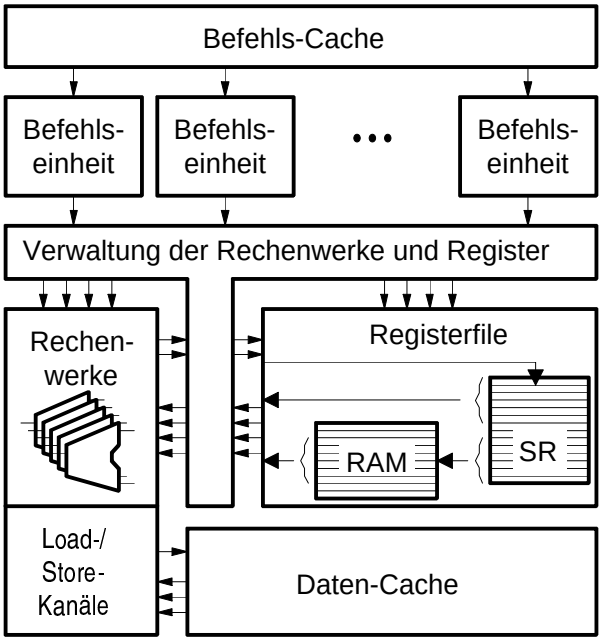

Bild 14: Das Modell des skalierbaren Prozessors

Nutzung grobkörniger Parallelität

Die Multithread-Verarbeitung ist in folgender Weise berücksichtigt. Der Prozessor enthält eine variable Anzahl von Befehlsholeeinheiten nach Bild 12, eine große Anzahl von Registern und eine Einheit, die die Umrechnung zwischen Tread-Registeradressen und physischen Registeradressen durchführt. Jeder Thread muß sich, wenn er gestartet wird, eine bestimmtes Registerfenster von angenommen 32, 64 oder 128 Registern reservieren. Während seiner Abarbeitung werden die thread-spezifischen Registeradressen um einen Registerindex ergänzt, der in einem thread-spezifischen Register steht.

In Abschnitt 7 wurde bereits angesprochen, daß man die Delay-Slots bei Verzweigungen und Wartezeiten auf den Speicher auch nutzen könnte, um andere Threads weiter zu bearbeiten. Voraussetzung ist eine Thread-Umschaltung in einem Takt. Für die programmgesteuerte Verschachtellung von zwei Threads müssen die Eingangsregister, der Befehlszähler und das Register für den Registeradreßindex doppelt vorhanden sein. Signalisiert der aktuelle Befehl des Haupt-Threads, daß er *n* Takte warten muß (z.B. auf die Berechnung einer Verzweigungsbedingung oder Sprungadresse), wird für die entsprechende Zeit der Ersatz-Thread weitergeführt. Auf die gleiche Weise können Wartezeiten genutzt werden, die durch einen Cache-Miss oder einen Page-Segment-Fault in der Befehlsholephase entstehen.

Wartezeiten auf Daten sind nicht so einfach durch andere Threads zu nutzen. Der wartende Thread blockiert die Befehlsholeeinheit. Es erscheint hier zweckmäßiger, auf die Auslastung der Ressourcen durch feinkörnige Parallelität zu orientieren. Den Threads der anderen Befehlsholeeinheiten werden mehr Rechenwerke zur Verfügung gestellt.

Nutzung feinkörniger Parallelität

Hier ist an das Konzept nach Abschnitt 5 gedacht. Unabhängige Befehle werden über einen gemeinsamen Längencode zu einem VLIW-Befehl zusammengefaßt. Der Prozessor wird von der Suche zeitgleich ausführbare Befehle, der Optimierung der Befehlsreihenfolge und dem Umbenennen von Registern während der Abarbeitung entlastet. Diese Aufgaben hat der Compiler zu lösen. Der Prozessor erhält dafür eine andere Aufgabe. Benötigen die aktiven Threads insgesamt mehr Rechenwerke als im Prozessor verfügbar, wird entweder ein Teil der Threads angehalten oder parallel ausführbare Anweisungen eines oder mehrere Threads werden nacheinander ausgeführt.

Die Anzahl der zeitgleich ausführbaren Anweisungen eines Threads unterscheidet sich von Zeitscheibe zu Zeitscheibe. Für eine hohe Auslastung der Rechenwerke ist es sinnvoll, die Rechenwerke den Treads zeitscheibenweise zuzuordnen.

Durch Ausnutzung von feinkörniger Parallelität, benötigt der Prozessor entsprechend weniger Befehlsholeeinheiten als Rechenwerke. Die Minimierung der Anzahl der Befehlsholewerke verlangt eine hohe mittle Parallelität auf Anweisungsniveau. Dazu gehört, daß die Hardware eine spekulative Ausführung von Befehlen erlaubt (vgl. Abschn. 6). Letzteres kann z.B. erreicht werden durch:

Realisierung eines Teils des Registerfiles als Schieberegister

Die meisten Zwischenergebnisse eines RISC-Programms werden nur kurze Zeit in Prozessorregistern gespeichert. Die Codedichte erhöht sich, wenn sie ersteinmal in einen Schieberegisterspeicher abgelegt werden (Bilder 6 und 14). Dieses Prinzip ist skalierbar. Stellt der Prozessor *n* Ergenisse fertig, werden *n* neue Ergenisse abgelegt und die zuvor berechneten Ergebnisse wandern *n* Positionen weiter. Ein Auslagern von Daten aus dem Schieberegister in den direktadressierbaren Registerteil erfolgt nur für Daten mit langer Lebensdauer (Basiszeiger, globale Konstanten) und wird in zusätzlichen Anweisungen codiert. Neben der Einsparung der Ergebnisadressen werden durch diese einfache Architektur komplizierte Mechanismen wie das Umbenennen von Registern und das Verwerfen spekulativ berechneter Ergebnisse gelöst. Jedes Rechenergebnis ist eindeutig einem Speicherplatz zugeordnent. Wird für spekulativ berechnete Werte das Programmsegment, in dem sie genutzt werden sollen, nicht erreicht, werden sie weder in permanente Register kopiert, noch als Operand genutzt noch gespeichert. Sie verfallen einfach.

Die hier skizzierte Registerarchitektur würde in herkömmlichen Befehlsarchitekturen Probleme bereiten. Der Adreßraum für Operanden (direkt adressierbares und Schieberegister zusammen) muß voraussichtich größer als 5 Bit (32 Register) sein. Ergebnisadressen sind kleiner und werden nur in wenigen Befehlen verwendet (kopieren vom Schieberegister oder vom Speicher in ein direkt adressierbares Register). Die in dieser Studie entwickelten Befehlsarchitektur ist hinreichend flexibel, um den Befehlssatz an diese neue Registerarchitektur anzupassen.

Unterbrechungskonzept

Superskalare Prozessoren sind so konzepiert, daß sie bei Unterbrechungen wie ein Prozessor mit einem Rechenwerk reagieren. Die Anweisungen werden zwar außer der Reihe ausgeführt. Das Kopieren in für den Programmierer sichtbare Register erfolgt jedoch erst, wenn die Ergebnisse aller vorhergehenden Operationen bereits sichtbar sind (in order retiring). Nur die Zustände dieser Register müssen nach einer Unterbrechung wiederhergestellt werden. Alle anderen Teilergebnisse werden neu berechnet. Der Sinn dieses Konzeptes ist außschließlich Kompatibilität zu älteren Prozessoren.

In zukünftigen Prozessorkonzepten muß der Compiler wieder die Codeoptimierung übernehmen und für den Prozessor einige Informationen zur Steuerung der Parallelverarbeitung verschlüsseln. An ein identisches Prozessormodell oder gar an Binärkompatibilität mit älteren Prozessoren ist nicht zu denken. Damit ist es auch an der Zeit, das Interrupt- und Trap-Konzept von gewachsenem Overhead zu bereinigen.

Die Alternative zu einer teilweisen Wiederherstellung von Registerinhalten und einer teilweisen Neuberechnung ist einfach. Alle Ergebnisse werden nach der Unterbrechnung wieder hergestellt.

Der Ansatz eines Multithread-Prozessors bietet darüber hinaus eine interessante Alternative für das Interruptsystem. Statt zu unterbrechen erzeugen externe Anforderungen einen neuen Thread. Dazu muß auf dem Prozessor ständig ein Betriebssystemkern (ein priveligierter Thread) verfügbar sein, der die Anforderung entgegennimmt und dem neuen Thread Ressorcen zuordnet: Eingangsregister, Befehlszähler, Registeradreßindex und ein Registerfenster. Die gesamte Verwaltung erfolgt in Software.

Für Traps bietet sich ein ähnliches Konzept an. In einer Ausnahmesituation reaktiviert der Prozessor den Betriebssystemkern und übergibt bestimmte Parameter. Dieser Thread besitzt Zugriff auf alle Register und kann gesteuert durch Software Threads starten, abbrechne, auslagern, wieder laden, bei einem Page-Segment-Fault die fehlende Seite in den Speicher laden und Diagnosefunktionen wahrnehmen. Das einzige was er nicht darf ist, selbst Traps verursachen.

9 Experimentelle Basis

Für die Durchführung experimenteller Untersuchungen an Befehlssätzen realer und hypothetischer RISC-Prozessoren wurden ein Prozessor-Simulator [22] und ein C-Compiler [23] zur Code-Erzeugung entwickelt. Mit diesen Werkzeugen ist es möglich, anhand realer Anwendungsprogramme statische und dynamische Kenngrößen, wie Programmcodelänge, Befehlshäufigkeiten, Registerauslastung usw. zu ermitteln. Diese Kenngrößen können dann zur Bewertung und Optimierung der Prozessor-Architektur herangezogen werden.

Der Prozessor-Simulator arbeitet als Interpreter auf Maschinenbefehlsebene. Sein Befehlssatz entspricht etwa dem eines typischen RISC-Prozessors und kann relativ einfach modifiziert und erweitert werden. Es werden Maschinenprogramme in einer assemblerähnlichen Notation verarbeitet, die Simulationsgeschwindigkeit beträgt auf einer DECstation 5000/150 ca.300000 Befehle pro Sekunde. Der Simulator verfügt über eine grafische Benutzeroberfläche die eine einfache Steuerung und Auswertung der Simulationsläufe ermöglicht.

Der C-Compiler basiert auf dem SUIF-Compiler-Toolkit [24] der Stanford University, dessen Zwischencode-Format zur Konstruktion experimenteller Compiler besonders gut geeignet ist. Das Toolkit stellt verschiedene Compilerpässe

zur Code-Transformation und Optimierung bereit, die als Bausteine für einen Compiler verwendet werden können. Es wurde um einen speziell für die Experimente entwickelten Code-Generator ergänzt, der als Back-End arbeitet und den Zwischencode in Maschinenbefehle übersetzt. Im Gegensatz zu üblichen Code-Generatoren kann er mit relativ wenig Aufwand an unterschiedliche Ziel-Architekturen angepaßt werden. Diese Flexibilität beruht auf einer editierbaren Übersetzungstabelle, in der die Code-Auswahl für den Ziel-Befehlssatz definiert wird.

Mit diesen Werkzeugen sollen in der weiteren Bearbeitung des Forschungsprojektes die in dieser Studie diskutierten Ansätze quantitativ untersucht und verifiziert werden.

10 Zusammenfassung

Der Begriff RISC beschreibt keine konkrete Prozessorarchitektur, sondern charakterisiert die Suche nach guten Lösungen, um mit gegebenen Hardware-Möglichkeiten für allgemeine Programme eine hohe Rechenleistung zu erzielen. Gegenwärtig ist die Entwicklung in einer Sackgasse geraten. Die Prozessoren werden größer und komplizierter, ohne daß im gleichen Maße die Rechenleistung zunimmt (vgl. Abschn. 2 und 3).

Die Reduzierung des Befehlsdatenstroms ist ein bisher wenig genutztes Mittel, um mit relativ geringem zusätzlichen Hardware-Aufwand die nutzbare Prozessorleistung zu erhöhen. Der Gewinn resultiert genau wie der Gewinn aus der Vergrößerung der Cache-Speicher und der Verbreiterung der Datenbusse aus der Verringerung der Wartezeiten auf den Hauptspeicher. Für integrierte Prozessoren (embedded controller), die typisch mit langsamen Speichern und kleinen Caches zusammenarbeiten, ist der höchste Nutzen zu erwarten.

Für die Reduzierung des Befehlsdatenstromes gibt es zwei Ansätze: Zusammenfassen häufig verwendeter Befehlsfolgen zu Spezialbefehlen und die optimale Codierung der Befehle selbst.

Spezialbefehle sind nur für oft genutzte Teilaufgaben sinnvoll. Ein Universalprozessor bietet aus Anwendungssicht kaum Ansatzpunkte. Lediglich die Compiler verursachen einige häufig wiederkehrende Codestücke. Unsere bisherigen Untersuchungen beschränken sich auf eine Systematisierung der Möglichkeiten für die Befehlsdatenstromreduzierung. Experimente sind geplant und vorbereitet.

Ein vielversprechendes Konzept für Spezialbefehle ist ihre Kombination mit speicherprogrammierbaren Verarbeitungswerken. Jedes Programm kann seine eigenen Spezialbefehle definieren. Die Befehle werden nicht von herkömmlichen Rechenwerken nachgebildet, sondern in der Hardware verankert. Am Beispiel von ausgewählten Algorithmen der Bildverarbeitung wurde in einer seperaten Studie gezeigt, daß dieses Konzept große Reserven an Rechenleistung in sich birgt. Weiterführende Untersuchungen in dieser Richtung sprengen jedoch den Rahmen des Forschungsprojektes.

Eine optimale Codierung der Befehle verlangt den Übergang vom starren 32-Bit-Befehlsformat zu Befehlen variabler Länge. Dadurch lassen sich redundante Bits und überflüssige Operanden in den Befehlen vermeiden. Zur Verringerung der mittleren Bitanzahl werden Konstanten, Registeradressen und Operationscode mit Codeworten unterschiedlicher Länge verschlüsselt. Eine weitere Reserve liegt in der impliziten Annahme, daß für bestimmte Funktionen stets das selbe Register verwendet wird. Wir haben eine ähnliche Lösung für die Registeradressen, in die die Ergebnisse abzulegen sind, vorgeschlagen. Unter Ausnutzung ihrer i.allg. kurzen Lebensdauer werden Ergebnisse in ein Schieberegister abgelegt und nur bei Bedarf in permanente Register verlagert. Die Abarbeitung von Befehlen mit Bestandteilen variabler Länge dürfte bei der heutigen Integrationsdichte kein Problem mehr darstellen.

Durch eine optimierte Befehlscodierung läßt sich der Befehlsdatenstroms auf etwa 50 bis 70% reduzieren. Für genauere experimentelle Untersuchungen wurden Werkzeuge geschaffen. Die Experimente selbst stehen noch aus.

Mit den zunehmenden Möglichkeiten der integrierten Schaltungstechnik wird nicht nur versucht, die Geschwindigkeit der Prozessoren zu erhöhen. Die zweite Entwicklungsrichtung ist Parallelverarbeitung. Superskalare Prozessoren verbergen die interne Parallelverarbeitung vor dem Programmierer. Das Konzept verursacht ein schlechtes Verhältnis zwischen Schaltungsaufwand und Rechenleistung und bietet kaum noch Möglichkeiten zur Weiterentwicklung der Prozessoren. In Zukunft kann die tatsächliche Arbeitsweise des Prozessors bestenfalls durch Software, aber nicht mehr von der Prozessor-Hardware selbst versteckt werden. Das verlangt zusätzliche Informationen im Befehlscode.

Die Prozessoren wachse weiter. Die feinkörnige Parallelität erlaubt kaum noch eine Erhöhung der Parallelverarbeitung. Der nächste Leistungssprung wird wahrscheinlich durch eine Multithread-Erweiterung erzielt. Der Multitread-Ansatz verlangt nur noch relativ geringe Erweiterungen gegenüber den heutigen Superskalaren Prozessoren, die zum Teil durch Vereinfachungen an anderen Stellen kompensiert werden.

Ein Befehlssatz mit Elementen variabler Länge könnte neben der Reduzierung des Befehlsdatenstroms eine weiter Funktion übernehmen. Er eignet sich als Basis für ein neuartiges Prozessormodell, daß für mehrere Generationen zukünftiger Prozessoren Abwärtskompatibilität auf Binärebene garantiert. Die Idee ist, einen skalierbarer Prozessor mit einer variablen Anzahl von Befehls- und Verarbeitungswerken zu definieren und für das Modell eine optimierte

Befehlsarchitektur zu entwickeln. Zukünftige Prozessoren sollen in dieses Modell hereinwachsen, indem sie von Generation zu Generation zunehmend mehr Operationen und später Threads parallel bearbeiten.

Ein Befehlsformat mit Elementen variabler Länge hat für so ein Modell ganz entscheidende Vorzüge. Es vereinfacht das Programmiermodell, und es verlangt nicht, daß alle potentiellen Erweiterungen (z.B. die Erweiterungen des Wertebereichs von Konstanten, der Registeranzahl oder der Thread-Anzahl) schon von Anfang an geplant werden.

Wir wollen die Entwicklung eines solchen Befehlssatzes für unsere Untersuchungen im Auge behalten. Die personelle Kapazität unseres Forschungsprojektes reicht natürlich nur für einige selektive Untersuchungen.

Literatur:

[1] Kemnitz, G.: Untersuchungen zu Spezialhardware für ausgewählte Algorithmen der Bildverarbeitung. Forschungsbericht TUD / FI / 95 / 11-Oktober 95.

[2] Hennessy, J.; Patterson, D.: Computer architecture - a quantitative approach. Morgan Kaufmann Publishers 1990

[3] Internal Organization of the Alpha 21164. Digital Technical Journal, vol.7, no.1, 1995

[4] Intel: A tour of the P6 microarchitecture. 1995

[5] Lee, R.; Mahon, M.; Morris, D.: Pathlength reduction features in the PA-RISC architecture. COMPCON 92, p. 129-35

[6] Atkins, M.: Performance and the i860. IEEE Micro, 10/91, p. 24-78

[7] Diefendroff, K.; Allen, M.: Organization of the Motorola 88110 Superscalar RISC microprocessor. IEEE Micro, 4/92, p. 40-63

[8] Asprey, T. et.al.: Performance features of the PA7100 microprocessor. IEEE Mikro 6/93, p. 22-35

[9] Fisher, J.: Trace Scheduling: A Technique for global microcode compaction. IEEE Transaction on Computers, Vol. C-30, No.7, July 1981

[10] Gupta, R., Soffa, M.: Region scheduling: An approach for detecting and redistributing parallelism. IEEE Transactions on Software Engineering, April 1990

[11] Nicolau, A.: Percolation scheduling: A parallel compilation technique. Computer Sciences Technical Report 85-678, Cornell University, 1985

[13] Cianciolo, S.: Aufbruch in neue Welten: Mikroprozessorforum in San Jose ct 12/95, S.16

[14] Markwardt, G.: Diplomarbeit an der TU Dresden

[15] SPARC Technology Business: Introducing UltraSPARC. September 1994

[16] Sawitski, S.:Anteil der redundanten Bitstellen im Befehlsdatenstrom am Beispiel MIPS R2000/R3000. Interne Studie am Institut für Technische Informatik der TU Dresden, 1995

[17] Kane, G.: MIPS/RISC Architecture. Prentice Hall, 1989

[18] Sawitski, S.: Möglichkeiten der Reduzierung des Befehlsdatenstroms eines RISC-Prozessors durch Anwendung einiger gängiger Methoden der Datenkompression. Interne Studie am Institut für Technische Informatik der TU Dresden, 1995

[19] Wolfe, A.; Chanin, A.: Executing compressed programs on an embedded RISC architecture. MICRO 25 proceedings, 1992, p. 81-8

[20] Markwardt, G.: Mittlere Coderwortlänge bei ungleichmäßiger Codierung von Registeradressen. Interne Studie am Institut für Technische Informatik der TU Dresden, 1995

[21] De Gloria, A.: VISA: A variable instruction set architecture. Computer Architecture News, vol. 18, no. 2, 1990, p. 76-82

[22] Markwardt, G.; PROSIM - Ein Werkzeug zur Befehlsebenen-Simulation von RISC-Prozessoren. TU Dresden Forschungsbericht FI/95/16

[23] Schulz, P.: Entwurf und Implementierung eines Objektcode-Generators für SUIF. TU Dresden Forschungsbericht FI/95/16

[24] Stanford Compiler Group: The SUIF library. Dokumentation SUIF 1.0 Release 1994

Anhang: Ein Beispiel für die Codeoptimierung durch spekulative Befehlsausführung

C-Programm

```
int i, j, x, a[];
for (i = 0; i < j; i++)
       if (a[i] > x)
              a[i] = x;
```

Assembler-Programm

```
R1 = i, R2 = j, R3 = x, Rsp = Stack-Pointer, #a = Offset von a
```

1. sequentiell

```
        blei R2, 0, L3          wenn R2 <= 0, springe zu L3
        xor  R1, R1, R1         R1 = 0
        addi R4, Rsp, #a        R4 = Stack-Pointer + Offset von a
L1:     lw   R5, 0, R4          Wort auf Adresse R4 in R5 laden
        ble  R5, R3, L2         wenn R5 < R3, springe zu L2
        sw   R3, 0, R4          Inhalt von R3 auf Adresse R4 speichern
L2:     addi R4, R4, 4          R4 = R4 + 4
        addi R1, R1, 1          R1 = R1 + 1
        blt  R1, R2, L1         wenn R1 < R2, springe zu L1
L3      ...
```

3 Zyklen für die Vorbereitung, 5 oder 6 Zyklen für die innere Schleife

2. parallel mit spekulativer Ausführung

	blei R2, 0, L3	xor R1, R1, R1	addi R4, Rsp, #a
L1:	lw R5, 0, R4		
	ble R5, R3, L2	addi R1, R1, 1	
	sw R3, 0, R4		
L2:	blt R1, R2, L1	addi R4, R4, 4	
L3:	...		

1 Zyklus für die Vorbereitung, 3 oder 4 Zyklen für die innere Schleife

3. parallel mit spekulativer Ausführung, optimiert

	blei R2, 0, L3	xor R1, R1, R1	addi R4, Rsp, #a	lw R5, #a, Rsp
L1:	ble R5, R3, L2	addi R1, R1, 1	lw R5, 4, R4	addi R4, R4, 4
	sw R3, -4, R4			
L2:	blt R1, R1, L1			
L3:	...			

1 Zyklus für die Vorbereitung, 2 oder 3 Zyklen für die innere Schleife