Heft 5
Der Einzelprozessor

1. Einführung

1.1. Die Struktur des klassischen Einzelprozessors

Wie sehen Prozessoren aus?
Die heutigen Prozessoren sind nahezu ausnahmslos Mikroprozessoren: alle Funktionseinheiten sind auf einem einzigen Schaltkreis untergebracht. Die äußere Form: entweder ein typisches Schaltkreisgehäuse oder eine Steckkassette.

1.1.1. Prozessoren im Blockschaltbild

Blockschaltbilder bilden oft (neben mehr oder weniger ausführlichen Erläuterungen) die einzige Quelle, um uns darüber zu orientieren, wie ein Prozessor aufgebaut ist. Einschlägige Beispiele haben wir in Heft 4 zusammengestellt. Weitere Blockschaltbilder folgen in Kapitel 4 es vorliegenden Heftes.

Hinweis:
Um sich klarzumachen, worum es hier geht, sollten Sie sich in Heft 4 zunächst die Abbildungen 9.1, 9.2, 9.9 und 9.16...9.19 ansehen. Wichtig:

1. die Blockschaltbilder, die von den Herstellern veröffentlicht werden, dienen nur zur überschlägigen Orientierung - versuchen Sie nicht, allzu viele Einzelheiten herauslesen zu wollen,
2. je komplizierter ein Prozessor, desto weniger detailliert sein Blockschaltbild.

Das genaue Blockschaltbild - die RTL-Darstellung
RTL = Register Transfer Level (Register-Transfer-Ebene). In einem solchen Blockschaltbild sind alle Hardware-Register und alle kombinatorischen Schaltungen zwischen den Registern als einzelne Funktionsblöcke dargestellt. Die Abbildungen 2.1 und 2.2 (dieses Heft) sowie 9.1 und 9.2 (Heft 4) mögen als Beispiel dienen. Wollen wir uns in die Einzelheiten einer Prozessor-Hardware einarbeiten (beispielsweise um wirklich “scharfe” Testprogramme zu schreiben), so brauchen wir die entsprechenden RTL-Darstellungen*). Allerdings werden diese von den Herstellern nur selten veröffentlicht (manche Mikrocontroller werden tatsächlich so dokumentiert - vgl. Abbildung 9.17 in Heft 4 -, Hochleistungsprozessoren hingegen gar nicht).

*): jede synchron (getaktet) arbeitende Digitalschaltung läßt sich auf das Schema Register - Kombinatorik - Register zurückführen. Vgl. auch Abbildung 1.2. Die detaillierte RTL-Darstellung ist deshalb *das* Dokumentationsmittel für den Fachmann - ähnlich der exakten Gesamtzeichnung im Maschinenbau.

Was ist in der Praxis maßgebend?

- **#** für den Programmierer: die Architekturbeschreibung (Architecture Reference Manual, Programmers Reference Manual o. ä.). Hierin finden wir die Einzelheiten der Befehlsliste, der Adressierung usw., aber kaum etwas (gelegentlich: gar nichts) zum Aufbau der Hardware.
- **#** für den Hardware-Entwickler und für den (womöglich meßtechnisch vorgehenden) Servicetechniker: das Datenblatt bzw. die Hardwarebeschreibung (Data Sheet, Hardware Reference Manual o. ä.). Hierin finden wir die Einzelheiten des Prozessor-Bus, der elektrischen und thermischen Betriebsbedingungen usw., aber nur selten etwas zum inneren Aufbau der Funktionseinheiten und nur wenig (gelegentlich: gar nichts) zur eigentlichen Architektur (Befehlsliste, Adressierung usw.).

Architekturbeschreibungen
Architekturbeschreibungen "echter" (am Markt erhältlicher) Prozessoren sind recht umfangreich und ohne Fachkenntnisse nur schwer zu verstehen. Wir haben deshalb eigens zu Lehrzwecken eine fiktiven Prozessor P/F entwickelt. Dessen Kurzbeschreibung sollten Sie zunächst einmal durchsehen, um sich eine Vorstellung davon zu bilden, welche elementaren Befehle ein moderner Prozessor ausführen kann.

1.1.2. Der klassische Einzelprozessor

Im klassischen Sinne besteht ein Einzelprozessor (Abbildung 1.1) aus der Steuereinheit (Steuerwerk, Control Unit), der zentralen Verarbeitungseinheit (Operationswerk, Central Processing Unit CPU) und der Speicheranschaltung (Memory Port, Storage Adapter, Storage Control Unit SCU). Die Steuereinheit enthält den Befehlszähler, das Befehlsregister sowie alle Schaltmittel zur Ablaufsteuerung. Die CPU (sprich: Siepie-juh) enthält die Verknüpfungsschaltungen, die notwendig sind, um die in den Operationsbefehlen angegebenen Operationen auszuführen, die Adressierungsschaltungen für Datenzugriffe sowie die notwendigen Register. Beide Funktionseinheiten liefern Adressen, um Lese- und Schreibzugriffe auszuführen. Die Steuereinheit holt Befehle und Befehlsadressen aus dem Speicher (z. B. bei Rückkehr aus einem Unterprogramm oder beim Einleiten einer Unterbrechungsbehandlung) und schreibt Befehlsadressen zurück (Adressenrettung). Die CPU holt Operanden bzw. Operandenadressen und schreibt Ergebnisse zurück. Der Universalregistersatz wird von Steuereinheit und CPU gemeinsam genutzt. Die Speicheranschaltung verbindet Steuereinheit und CPU mit dem Arbeitsspeicher. Moderne Prozessoren sind um zusätzliche Funktionsblöcke erweitert, beispielsweise um eine Gleitkomma-Verarbeitungseinheit (Floating Point Processing Unit FPU; sprich: Effpie-juh), eine Befehlsvorbereitungseinheit, eine Segmentierungseinheit, eine Seitenverwaltungseinheit, Caches usw.

Abbildung 1.1 Blockschaltbild eines Einzelprozessors

1.2. Grundsätzliche Leistungsgrenzen

Der klassische Einzelprozessor führt zu einer Zeit nur eine Operation aus. Die Hardware-Grundlage dafür bilden die Verknüpfungsschaltungen in der CPU (die Operationswerke). Der Einzelprozessor hat ein einziges Operationswerk, dessen Verknüpfungsschaltungen jeweils so angesteuert werden, daß jeder Operationsbefehl die in seinem Operationscode angegebene Wirkung erbringt (Grundsatz: *ein* Befehl - *eine* Operation). Die Struktur des Operationswerkes bestimmt somit die Leistung, wenn wir annehmen, daß alle "Nebenfunktionen" (Adreßrechnungen, Verzweigungen usw.) nicht zur Verarbeitungszeit beitragen, daß also Operationen lückenlos aufeinander folgen. Abbildung 1.2 zeigt, wie die grundsätzliche Struktur eines Operationswerkes aussieht.

Abbildung 1.2 Grundsätzliche Struktur eines Operationswerkes

An Register, die aus binären Speicherelementen (Flipflops) aufgebaut sind, sind Verknüpfungsschaltungen angeschlossen, die die jeweiligen Operationen ausführen. Die Verknüpfungsschaltungen sind aus Gattern aufgebaut, das heißt aus Schaltmitteln, die aussagenlogische Verknüpfungen verwirklichen (wie UND, ODER, NICHT usw.). Ihnen sind wiederum Register nachgeschaltet, um die Verarbeitungsresultate aufzunehmen. Wir nehmen den günstigsten Fall an: die Daten werden in die erste Register-Ebene eingeschrieben, dann verknüpft, und die Resultate werden in die zweite Register-Ebene übernommen. Solche Abläufe (Maschinenzyklen) folgen lückenlos aufeinander.

Wie lange dauert ein Maschinenzyklus?
Nehmen wir an, daß die Daten gerade in die erste Register-Ebene eingeschrieben worden sind (Beginn des Maschinenzyklus). Es ist klar, daß wir die Resultate erst dann in die zweite Register-Ebene übernehmen können (Ende des Maschinenzyklus), wenn die betreffenden Signale (volkstümlich: “die Bits”) die Verknüpfungsschaltungen durchlaufen haben. Als Praktiker können wir es uns auch denken, daß man nicht die knappste Zeit ansetzen sollte, sondern den ungünstigsten Fall betrachtet und zudem einen “Sicherheitszuschlag” hinzurechnet.

Dies läßt sich durch eine einfache Formel ausdrücken:

$$t_c = (s \cdot t_p) + t_{techn}$$

Die Bedeutung der Symbole:

t_c:	Zykluszeit,
t_p:	Verzögerungszeit des einzelnen Gatters,
s:	Schaltungstiefe (s = 1, 2, ..., n); der Wert drückt aus, wieviele Gatter im ungünstigsten Fall nacheinander durchlaufen werden müssen (dies wird in Abbildung 1.3 veranschaulicht; dort ist s = 3),
t_{techn}:	Zusammenfassung aller weiteren Zeitanteile, die letztlich durch die Schaltkreistechnologie bestimmt werden.

Abbildung 1.3 Zur Veranschaulichung des Begriffs der Schaltungstiefe

Rechenbeispiel:
t_p = 5 ns, s = 12, t_{techn} = 15 ns (das sind Richtwerte, die sich bei Nutzung herkömmlicher Schaltkreise ergeben - also dann, wenn wir einen Prozessor aus einzelnen Gattern, Flipflops usw. aufbauen würden).

$$t_c = (12 \cdot 5 \text{ ns}) + 15 \text{ ns} = \underline{\underline{75 \text{ ns}}}$$

Diesen Prozessor könnten wir also mit einem Taktzyklus von 75 ns bzw. mit einer Taktfrequenz von rund 13 MHz betreiben.

Wodurch läßt sich die Verarbeitungsleistung erhöhen?
Indem wir den Prozessor mit einer geringeren Zykluszeit t_c laufen lassen. Dabei muß aber die obige Gleichung eingehalten werden. Welche Ansätze gibt es?

1. Verringerung der Zeiten t_p, t_{techn}
Dies ist Sache der Halbleiter- bzw. Schaltkreistechnologie (hat also mit der Rechnerarchitektur an sich nichts zu tun).

2. Verringerung der Schaltungstiefe s
Die Schaltungstiefe hängt im wesentlichen von der Kompliziertheit der kombinatorischen Informationswandlungen ab. Das bedeutet: wenn wir nur Befehle mit einfachen Wirkungen vorsehen, können wir die Hardware schneller laufen lassen (die sog. RISC-Philosophie; Abschnitt 1.6.). Die Vereinfachung findet aber ihre Grenzen: (1) bestimmte Operationen (z. B. die Addition zweier Binärzahlen) sind unbedingt notwendig. Diese Informationswandlungen müssen aber (2) aus elementaren aussagenlogischen Verknüpfungen aufgebaut werden - und da kann man zwar durchaus tricksen, aber nicht zaubern (gewisse Mindest-Schaltungstiefen (Richtwert: s = 10...20) sind also nicht weiter zu unterbieten).

Die Technologie macht's
Das war bisher immer so - und wird sich auch demnächst kaum ändern: die Verarbeitungsleistung ist vor allem durch technologische Verbesserungen gesteigert worden und weniger durch revolutionäre Lösungen seitens der Architektur. Diese Verbesserungen lassen sich durch 2 Entwicklungsrichtungen kennzeichnen:

1. Erhöhung des Integrationsgrades - hierdurch wurde es nach und nach möglich, die - an sich seit langem bekannten - Maßnahmen der Durchsatzverbesserung und Leistungssteigerung (Caches und TLBs, große Verarbeitungsbreite, Hardware-Multiplizierer, Gleitkommaverarbeitung, Vektorverarbeitung usw.) auf einem einzigen Schaltkreis vorzusehen,
2. Steigerung der internen Taktfrequenz. Auch das ist ein Ergebnis der Schaltungsintegration. (Vgl. unser obiges Rechenbeispiel: mit üblichen Einzelschaltkreisen kommt man auf Taktfrequenzen von knapp über 10 MHz. Mit den schnellsten Einzelschaltkreisen wären ca. 50 MHz erreichbar.)

Etwas unhöflich formuliert: die Technologie hat bisher immer die Unzulänglichkeiten der Architektur gleichsam überspielt - wenn man einen Prozessor mit 800 MHz laufen lassen kann, fällt es womöglich gar nicht auf, daß es sich bei der einen oder anderen Architekturfestlegung um eine doch recht ungeschickte Lösung oder um einen faulen Kompromiß handelt. Auch Architekturen, die auf Uralt-Grundlagen beruhen (vor allem: IA-32) können so geradezu phantastische Verarbeitungsleistungen erreichen*).

*): hieran hat nicht nur die Taktfrequenz, sondern auch die Schaltungsintegration ihren Anteil. Kennzeichnend für IA-32 ist die Abwärtskompatibilität bis herunter zum 8086. Um diese zu gewährleisten, ist eine Vielzahl von Schaltungslösungen erforderlich - und es ist allein die Halbleitertechnologie, die es ermöglicht, Prozessoren, die um solche Schaltungen erweitert sind, kostengünstig zu fertigen.

Was kann ein klassischer Einzelprozessor leisten?
Moderne Prozessoren werden so ausgelegt, daß das Operationswerk (gemäß Abbildung 1.2) in einer Periode des (internen) Prozessortakts durchlaufen wird; t_c entspricht also der Periodendauer des Prozessortakts bzw. dem Kehrwert der Taktfrequenz.

Die maximal mögliche Leistung P_{max} (in "Befehlen je Zeiteinheit") ergibt sich folgendermaßen: $P_{max} = \frac{1\ \text{Befehl}}{t_c}$ bzw. $P_{max} = f_c \cdot 1\ \text{Befehl}$

Beispiel: $f_c = 250$ MHz; $P_{max} = 250 \cdot 10^6/\text{s} \cdot 1\ \text{Befehl} = 250 \cdot 10^6$ Befehle/s = 250 MIPS.

Wir merken uns:
Der klassische Einzelprozessor hat ein maximales Leistungsvermögen, das darin besteht, in jedem internen Taktzyklus einen Befehl auszuführen. Dies ist aber nur unter idealen Betriebsbedingungen zu erreichen.

Hinweis:
In der Fachpresse, in Prospekten usw. wird der eben beschriebene Zusammenhang häufig auf bildhafte Weise dargestellt: man spricht von MIPS/MHz, und stellt es gebührend heraus, falls ein Prozessor 1 MIPS/MHz schafft (1 Million Befehle je s bei 1 MHz (internem) Prozessortakt entsprechen genau einem Befehl je Taktzyklus).

1.3. v. Neumann-Architektur und Harvard-Architektur

Die Begriffe bezeichnen allgemeine Architekturkonzepte, die sich darin unterscheiden, wieviele Speicheradreßräume bzw. Speicherzugriffswege grundsätzlich vorgesehen sind (Abbildung 1.4).

Gemeinsamkeiten:
Beide Architekturen haben einige wesentliche Prinzipen gemeinsam:

es gibt einen einzigen Befehlsstrom, wobei Befehl für Befehl nacheinander ausgeführt wird,

- die Befehlsadressierung erfolgt vorzugsweise durch Weiterzählen der Befehlsadresse (Ausnahmen davon sind Verzweigungen, Unterprogrammrufe, Unterbrechungen usw.).

Architekturen, die diese Merkmale nicht aufweisen, sind - beim aktuellen Stand der Technik - eher in den Bereich der akademischen Forschung einzuordnen (man spricht hier von rekursiven Architekturen, funktionalen Architekturen, Datenflußarchitekturen usw.). Wir wollen uns hiermit nicht weiter beschäftigen (Ausnahme: Teillösungen, die in anwendungspraktisch wichtige Prozessorfamilien eingeflossen sind; vgl. Abschnitt 3.4.4.).

Abbildung 1.4 v. Neumann- und Harvard-Architektur

Zu den Namen:

- v. Neumann-Architektur: nach dem Mathematiker John v. Neumann,
- Harvard-Architektur: nach der Harvard-Universität (Cambridge, Massachusetts (USA)).

v. Neumann-Architektur
Es gibt nur einen einzigen Speicheradreßraum und einen einzigen Zugriffsweg. Mit anderen Worten: es gibt aus der Sicht des Programmierers nur einen einzigen, von Adresse 0 an fortlaufend adressierbaren Speicher, der alle Programme, Datenbereiche, deskriptiven Angaben usw. aufnimmt, die für die laufende Arbeit benötigt werden.

Harvard-Architektur
Es gibt zwei Speicheradreßräume - einen für Daten und einen für Befehle. Gemäß der technischen Auslegung unterscheiden wir:

1. echte Harvard-Maschinen
Es gibt eine gesonderte Speicheranordnung je Adreßraum (also einen Programmspeicher und einen Datenspeicher) und unabhängige Zugriffswege zu beiden Speicheranordnungen (Abbildung 1.4b; als Architekturbeispiele vgl. die Abbildungen 9.17 und 9.22 in Heft 4).

2. Maschinen mit Harvard-Architektur
Diese haben zwar die beiden getrennten Adreßräume, aber nur einen einzigen gemeinsamen Speicherzugriffsweg (Speicherbus). Als Architekturbeispiel vgl. Abbildung 9.16 inHeft 4.

Der große Vorteil der v. Neumann-Architektur: der einheitliche lineare Adreßraum
Wir können mit der Speicherkapazität anstellen, was wir wollen. Vor allem können wir die gesamte installierte Speicherkapazität voll ausnutzen - ob wir vorwiegend Programme oder vorwiegend Daten speichern, ist gleichgültig. Auch lassen sich Programme ohne weiteres als Daten behandeln, also mit den üblichen Maschinenbefehlen transportieren, ändern usw.

Die Vorteile der Harvard-Architektur

1. Höhere Leistung
Eine v. Neumann-Maschine holt Befehle und Daten nacheinander aus dem selben Speicher. Dies beeinträchtigt naturgemäß das Leistungsvermögen*). Eine echte Harvard-Maschine kann hingegen gleichzeitig (parallel) auf Daten und Befehle zugreifen.

*): man spricht bildhaft vom "v. Neumann-Flaschenhals" (v. Neumann Bottleneck).

2. Aufwandsoptimierung
Beide Speicher einer echten Harvard-Maschine kann man unabhängig voneinander hinsichtlich der Zugriffsbreite, der Technologie*) usw. optimieren. Ist beispielsweise das Byte die elementare Datenstruktur, so müssen bei einer v. Neumann-Maschine auch die Befehlsformate in Bytestrukturen gezwängt werden (auch Befehle sind Aneinanderreihungen von Bytes). Eine echte Harvard-Maschine kann man hingegen so auslegen, daß der Befehlsspeicher eine jeweils genau passende Zugriffsbreite erhält - auch wenn es ein "krummer" Wert ist (u. a. sind Prozessoren mit Befehlen von 12, 14, 24 Bits usw. gebaut worden - Abbildung 9.17 in Heft 4 zeigt einen Prozessor mit 12-Bit-Befehlen).

*): z. B. liegt es nahe, den Programmspeicher eines Mikrocontrollers als ROM auszulegen und den Datenspeicher als RAM.

3. Höheres Adressierungsvermögen
Da es zwei Adreßräume gibt, wird das Adressierungsvermögen der Hardware praktisch verdoppelt. Beispiel: ein Prozessor sei für 16-Bit-Adressen ausgelegt (Byteadressierung). In einer v. Neumann-Architektur beträgt das Adressierungsvermögen insgesamt 2^{16} = 64 kBytes. Eine Harvard-Maschine könnte hingegen mit einem Befehls- und einem Datenspeicher von jeweils 64 kBytes bestückt werden (insgesamt 128 kBytes)*). Die schlechte Nachricht: wenn wir z. B. 20 kBytes für die Programme, aber 100 kBytes für die Daten brauchen, so paßt es nicht (gelegentlich kann man tricksen - das ist aber etwas zu spitzfindig für den Anfang).

*): deshalb hat man viele Mikrocontroller, obwohl sie nur einen einzigen Speicherbus haben, als Harvard-Maschinen ausgelegt (vgl. Abbildung 9.16 in Heft 4).

Welche dieser Architekturen bestimmt den Stand der Technik?

Alle modernen Hochleistungsprozessoren*) sind ausnahmslos v. Neumann-Maschinen. Das heißt, sie verhalten sich aus Sicht des Programmierers und des Speichersubsystems als solche. Intern gibt man sich aber Mühe, die Vorteile beider Architekturen zu vereinigen (Abschnitt 2.4.8.).

*): und auch die meisten Prozessoren und Mikrocontroller der mittleren Leistungsklassen.

Demgegenüber sind die meisten Signalprozessoren und auch viele der kleinen Mikrocontroller als Harvard-Maschinen ausgelegt (Tabelle 1.1).

Vorteil	Erläuterung	Beispiele
höheres Adressierungs-vermögen	weil es 2 unabhängige Adreßräume gibt	Intel 8051
technologische Trennung zwischen Daten- und Programmspeicher	Datenspeicher: SRAM, Programmspeicher: ROM (auch EPROM oder Flash)	Microchip PIC, Atmel AVR
interne Optimierung	Befehle werden so breit ausgelegt wie es jeweils zweckmäßig ist (also nicht in Bytestrukturen gepreßt)	Microchip PIC16x
Leistungssteigerung	durch parallelen Zugrifff auf Befehle und Daten	die meisten Signalprozessoren

Tabelle 1.1 Weshalb Mikrocontroller und Signalprozessoren oft als Harvard-Maschinen ausgelegt werden

1.4. Steuerflußprinzip und Datenflußprinzip

Steuerflußprinzip
Dieser Begriff bezeichnet einen wichtigen Gesichtspunkt des Programmiermodells der üblichen Rechnerarchitekturen: die Reihenfolge der Befehle*) bestimmt den Verarbeitungsablauf. Erst muß der Befehl gelesen werden, dann ist bekannt, welche Operanden auf welche Weise miteinander zu verknüpfen sind. Die Befehlsreihenfolge aber wird zwangsläufig vorgegeben (Befehlszählung, Verzweigung). In den derzeit üblichen Architekturen bestimmt der Befehl zudem, welche Bedeutung die gespeicherten Datenstrukturen eigentlich haben (vgl. Abschnitt 1.1.4. in Heft 4).

*): Fachbegriff: Steuerfluß (Control Flow; sprich: Kontrohl Floh).

Die Vorteile:

vergleichsweise Einfachheit (ein Befehlszähler, der zwecks Verzweigung überladen werden kann, ist eine geradezu banale Schaltungsstruktur),

vollkommene Universalität - man kann wirklich “alles” programmieren - vorausgesetzt, man hat genug Speicherplatz (und Zeit und Geduld).

Diese Vorteile sind in der Praxis so überwältigend, daß sich alle anderen Ansätze (und es hat deren eine beachtliche Anzahl gegeben) bisher nicht haben durchsetzen können.

Der wesentliche Nachteil:
Die beschränkte Verarbeitungsleistung. Erklärlich, da Befehl für Befehl nacheinander ausgeführt wird - bereitstehende Daten werden erst dann verarbeitet, wenn die entsprechenden Befehle im Rahmen des Steuerflusses gleichsam angeliefert werden. Stichwort: v.Neumann-Flaschenhals*).

*): den gibt es im Prinzip auch bei der Harvard-Maschine (1 Befehl zu einer Zeit) - obwohl er da etwas weiter ist (gesonderte Zugriffswege auf Befehle und Daten).

Datenflußprinzip
Im Gegensatz zum Steuerflußprinzip bestimmt die Verfügbarkeit der Daten den Verarbeitungsablauf. Die reine Datenflußmaschine hat keine Befehle im üblichen Sinne. Eine Einzweck-Datenflußmaschine kann man sich sehr leicht vorstellen: sie beruht auf einer Zusammenschaltung von Operationswerken, die dem gewünschten Datenfluß (Data Flow) im Verarbeitungsablauf entspricht. Abbildung 1.5 zeigt dazu einige Beispiele. (Zu den Operationswerken - die gemäß dem Datenfluß zusammengeschaltet sind - kommen noch die Speicher sowie Adressierungs- und Steuerschaltungen.)

Abbildung 1.5 Einzweck-Datenflußschemata (Beispiele)

In einer universellen Datenflußmaschine sind die Datenwerte um Angaben erweitert, die die Verknüpfungen und Abhängigkeiten beschreiben. Eine Verknüpfung erfolgt immer dann, wenn alle dafür notwendigen Eingangs-Datenwerte bereitstehen. Ist das Resultat gebildet, so löst dies jene Verknüpfungen aus, die von diesem Resultat-Wert abhängen.

Die reine Datenflußmaschine ist eine akademische Modellvorstellung. Wenn man jede Operation sofort auslösen kann, sobald ihre Eingangsdaten bereitstehen, so bestimmt nicht die Befehlsreihenfolge die Verarbeitungsleistung, und der einem Algorithmus innewohnende Parallelismus (Abschnitt 3.4.) kann in vollem Umfang wirksam werden. Solche Vorstellungen konnten aber bisher nur ansatzweise in Versuchsmaschinen verwirklicht werden. In der Praxis hat es sich gezeigt, daß auch die klassischen Architekturprinzipien noch etliche Möglichkeiten zur Leistungssteigerung bieten. Im besonderen kann man auf dieser bewährten Grundlage die Schaltkreistechnologie bis an die Grenzen ihrer Möglichkeiten ausnutzen. Deshalb ist mit *universellen* Datenflußmaschinen in nächster Zukunft nicht zu rechnen.

Zum technischen Hintergrund
Das Steuerflußprinzip kommt mit adressierbaren Speichern (RAM und ROM) aus - und die kann man heutzutage mit geradezu riesigen Speicherkapazitäten kostengünstig fertigen. Eine universelle Datenflußmaschine erfordert hingegen assoziative Speicher, aus denen sich die Daten sofort abrufen lassen, wenn die Bedingungen zu deren Verarbeitung gegeben sind. Derartige Speicher sind aber wesentlich aufwendiger als z. B. RAM-Anordnungen.

Zur tatsächlichen Anwendung von Datenflußprinzipien
Es gibt zwei wichtige Anwendungsgebiete:

1. die Steuerung von Befehlsabläufen im Innern von Hochleistungsprozessoren (Abschnitt 3.4.4.),
2. spezielle Hochleistungshardware (Graphik, Signalverarbeitung, Datenkompression, Verschlüsselung usw.).

Wir merken uns:

sowohl die v. Neumann- als auch die Harvard-Maschinen entsprechen dem Steuerflußprinzip,

Datenflußprinzipien werden typischerweise im Innern von Prozessoren*) oder in Spezialhardware angewandt.

*): d. h. im Rahmen des v. Neumann-Programmiermodells bei völliger Kompatibilität (der Anwender merkt davon nichts).

1.5. Skalar- und Vektorverarbeitung

Skalarverarbeitung
Eine skalare Größe bzw. ein Skalar ist in der Mathematik eine einfache Zahlenangabe, wie -5 oder 3,56791. Skalarverarbeitung bedeutet, alle anwenderseitig gewünschten Informationswandlungen mit Elementaroperationen über solche Angaben auszuführen (in der Informatik zählen auch Bytes, Worte, Zeichenketten usw. zu den Skalaren).

Die klassische Vektorverarbeitung
Ein Vektor ist in der Mathematik eine Angabe, die aus mehreren Skalaren *gleichen Typs* besteht (das sind die *Elemente* des Vektors). Vektoroperationen wirken auf alle Vektor-Elemente gleichermaßen. In Tabelle 1.2 sind einige elementare Vektoroperationen angeführt.

Operation	Formelschreibweise
elementweise Multiplikation zweier Vektoren	$Y(i) := X(i) \cdot U(i)$
elementweise Addition zweier Vektoren	$Y(i) := X(i) + U(i)$
Skalarprodukt zweier Vektoren	$Y = \sum X(i) \cdot U(i)$
Summe der Elemente eines Vektors	$Y = \sum X(i)$
Summe der Quadrate der Elemente eines Vektors	$Y = \sum X(i) \cdot X(i)$

$\sum$ bedeutet "Summe über alle i"; alle Komponenten werden aufaddiert

Tabelle 1.2 Elementare Vektoroperationen (Auswahl)

Solche Operationen sind so wichtig, daß man seit vielen Jahren Hochleistungsrechner dafür ausgelegt hat. Bei Supercomputern gehören Vektoren zu den elementaren Datentypen, und wichtige Vektoroperationen sind als Maschinenbefehle implementiert. Auch hat man Zusatzeinrichtungen (Spezialprozessoren) entwickelt, um übliche Universalrechner in ihrem Leistungsvermögen entsprechend zu erweitern.

Zur Bezeichnungsweise
Vektoroperationen werden oft (so auch in Tabelle 1.2) in folgender Form beschrieben: Z(i) := X(i) **op** Y(i). Das bedeutet: die Elemente an gleichen Positionen in beiden Vektoren X, Y werden mittels der Operation **op** verknüpft. *Beispiel:* Z(i) := X(i) + Y(i): das erste Element von X wird zum ersten Element von Y addiert, die Summe wird das erste Element von Z usw. Variable ohne Indexangabe (i) sind Skalare.

Vektorverarbeitung in modernen Hochleistungsprozessoren
Der Vektorbegriff wird in der Informatik gelegentlich auf reguläre Strukturen an sich beliebiger Elemente gleichen Typs übertragen (so spricht man u. a. von Binärvektoren, die aus einzelnen Bits bestehen). Auch klingt "Vektor" irgendwie wissenschaftlich - der Begriff wird deshalb von Marketing-Leuten gerne verwendet. Wir sollten deshalb genau darauf achten, was jeweils gemeint ist (Tabelle 1.3, Abbildung 1.6):

1. die klassische Vektorverarbeitung (mit langen Vektoren)
Es handelt sich um *numerische* Vektoren, deren Elemente Gleitkommazahlen sind. Ein solcher Vektor hat eine größere Zahl an Elementen (Richtwerte: maximal 64...2048). Der einzelne Befehl betrifft typischerweise komplette Vektoroperanden, die in Vektorregistern oder im Speicher untergebracht sind. Verarbeitungsschema (siehe oben): Z(i) := X(i) **op** Y(i). Anwendung: das numerische Rechnen in größtem Umfang (Supercomputing). Verarbeitungshardware: siehe Abbildung 3.4.

2. die Vektorverarbeitung in modernen Hochleistungsprozessoren (mit kurzen Vektoren)
Der einzelne Vektor hat nur wenige Elemente. Typisch sind u. a. Vektoren von 64 oder 128 Bits Länge, die aus 8, 16 oder 32 Bits langen Elementen bestehen (Beispiele: MMX (Intel), 3DNow (AMD), SSE (Intel), AltiVec (Motorola)). Der einzelne Befehl betrifft typischerweise Inhalte einzelner Vektorregister. Verarbeitungsschema: <Vektorregister C> := <Vektorregister A> **op** <Vektorregister B>. Anwendung: was heutzutage so Mode ist (Telefonieren über's Internet, Verarbeitung von Audio- und Videodaten, Spiele in 3D und Farbe mit Gebrüll in Stereo, Internet-Routing usw.). Verarbeitungshardware: siehe Abbildung 3.5.

Hinweise:

1. In diesem Sinne ist "Vektorverarbeitung" nur ein anderer Ausdruck für "SIMD-Verarbeitung" (Abschnitt 3.3.).
2. Die Befehlslisten der modernen Hochleistungsprozessoren sind auf die "neumodischen" Anwendungen hin ausgelegt, nicht auf das numerische Rechnen. Numerische Vektoren (Abbildung 1.6a) und Operationen ähnlich Tabelle 1.2 müßten also softwareseitig implementiert werden - wobei die Ausnutzung von SSE, AltiVec usw. sicherlich einen beachtlichen Zuwachs an Verarbeitungsgeschwindigkeit erbringt, aber wohl kaum in einer Größenordnung, die den Fachmann vor Begeisterung vom Stuhl reißt.

	klassische Vektorverarbeitung	Vektorverarbeitung in modernen Hochleistungsprozessoren
Vektorelemente	Gleitkommazahlen	natürliche und ganze Binärzahlen, Gleitkommazahlen
Länge des einzelnen Vektorelements	32 oder 64 Bits	8, 16, 32, 64 Bits
Vektorelemente im Vektor	typischerweise 64...2048	typischerweise 2, 4, 8, 16
ein Vektor entspricht	256 Bytes...16 kBytes	8 oder 16 Bytes

Tabelle 1.3 Zwei Auslegungen der Vektorverarbeitung

Abbildung 1.6 Zwei Auslegungen der Vektorverarbeitung

Erklärung:
S - Vorzeichen (Sign); E - Exponent; Si - Signifikand (vgl. Heft 1);
1 - Belegungsbeispiel: 4 32-Bit-Gleitkommazahlen; 2 - Belegungsbeispiel: 8 16-Bit-Binärzahlen.

a) Vektor in einem klassischen Vektorrechner,

b) Vektor- bzw. SIMD-Register in einem modernen Hochleistungsprozessor (Beispiele: AltiVec (Motorola), SSE (Intel).

1.6. CISC und RISC

Die Begriffe bezeichnen allgemeine Architekturkonzepte, die sich in der Gestaltung der Befehlsformate unterscheiden. Zunächst eine Kurzerklärung:

CISC = Complex Instruction Set Computer = Rechner mit komplizierter und umfangreicher Befehlsliste (viele Befehle mit teils recht “komfortablen” Wirkungen),

RISC = Reduced Instruction Set Computer = Rechner mit vereinfachter Befehlsliste (vergleichsweise wenige Befehle mit vergleichsweise elementaren Wirkungen).

Herkömmliche Architekturen (CISC)
Die Entwicklung der Rechnerarchitektur war in den 70er Jahren zu einem gewissen Abschluß gekommen. Als Beispiele seien die Systeme IBM /360 und /370, CDC 6600 und 7600 sowie DEC PDP 11 und VAX genannt. Folgende Sachverhalte haben die seinerzeitigen Lösungen maßgeblich beeinflußt:

Speichermittel mit wahlfreiem Zugriff waren kostbare Ressourcen. Große Speicherkapazitäten konnten praktisch nur mit Ferrit-Kernspeichern aufgebaut werden (1 MByte war außergewöhnlich viel; die verbreiteten sog. mittleren EDV-Anlagen hatten zwischen 64 und 256 kBytes - weniger als viele der "veralteten" XT-Personalcomputer). Speicher und Verarbeitungseinrichtungen beruhten auf verschiedenen Prinzipien und Technologien. Somit war es unvermeidlich, beide Teilsysteme getrennt aufzubauen und über entsprechende Schnittstellen miteinander zu verbinden. Solche Schnittstellen wirken naturgemäß als Flaschenhals.

der Integrationsgrad der Schaltmittel war gering, die Geschwindigkeit aber bereits recht hoch (so hatte die Anlage CDC 6600 - mit über hunderttausend einzelnen Transistoren aufgebaut - eine Zykluszeit von 100 ns).

es bestand ein deutlicher Unterschied zwischen den Zykluszeiten der Verarbeitung und des Speichers (Richtwert: 1:4...1:10).

schnelle Registerspeicher waren teuer. Somit konnten nur vergleichsweise wenige Universalregister (8...16) vorgesehen werden.

hingegen waren große Festwertspeicher für Mikroprogramme (Abschnitt 2.4.) vergleichsweise kostengünstig (Transformator- oder Kondensator-Festwertspeicher). Typische Kennwerte: mehrere tausend Worte mit teils über 100 Bits; Zykluszeiten zwischen 100 und 500 ns.

es wurde überwiegend mit Maschinenbefehlen programmiert (Assembler-Programmierung). Das war Anlaß, bei der Architekturgestaltung nach möglichst leistungsfähigen und flexibel nutzbaren Maschinenbefehlen zu suchen: zum einen mußte die langsame Verbindung zwischen Speicher und Prozessor gut ausgenutzt werden, zum anderen erlaubte es das Prinzip der Mikroprogrammsteuerung, komplizierte Befehlsabläufe zu implementieren. Auch waren die Befehle so festzulegen, daß sie vom Programmierer bequem und wirkungsvoll genutzt werden konnten. Wichtige Kriterien waren Vollständigkeit, Symmetrie und Orthogonalität*) sowie die weitgehende Unabhängigkeit der Architekturdefinition von technischen Gegebenheiten, um programmkompatible Familien von Rechenanlagen in sorgfältig abgestuften Preis- und Leistungsklassen anbieten zu können.

*): *Erklärung:* Die Begriffe finden wir gelegentlich noch in der neueren Literatur, obwohl die Eleganz der Maschinenbefehlsliste heutzutage kaum noch jemanden zu einem Systemwechsel verlockt. Vollständigkeit bedeutet, daß alle aus der Erfahrung heraus sinnfälligen Elementaroperationen vorgesehen sind. Symmetrie bedeutet, daß alle vorgesehenen Operationen auf alle dafür passenden Datentypen anwendbar sind. Beispielsweise darf es nicht sein, daß es 16-Bit- und 32-Bit-Worte gibt, man aber nur letztere miteinander multiplizieren kann. Orthogonalität bedeutet, daß verschiedene Teile der Operationsprinzipien unabhängig voneinander sind. Beispielsweise darf es nicht sein, daß die Operanden eines Multiplikationsbefehls anders adressiert werden als jene eines Additionsbefehls.

Die meisten der heutzutage üblichen Architekturprinzipien wurden seinerzeit geschaffen: Mikroprogrammsteuerung, universelle Bussysteme und standardisierte Interfaces, Universalregister, Adressierungsverfahren, virtuelle Speicher, das 8-Bit-Byte als grundlegende Datenstruktur usw.

Die Mikroprozessoren, die in den 70er Jahren aufkamen, waren keine architekturseitigen, sondern technologische Neuerungen. Es war möglich geworden, einen ganzen Prozessor - in der Anfangszeit naturgemäß einen nicht allzu komplexen - auf einem einzigen Schaltkreis unterzubringen. Erst gegen Ende der 80er Jahre war die Schaltkreistechnologie so weit, daß die Vielzahl der genannten Architekturprinzipien in Mikroprozessoren angeboten werden konnte.

Architekturen mit vereinfachten Befehlslisten (RISC)

Maschinen mit vergleichsweise elementaren Befehlslisten gibt es seit der Frühzeit der Rechentechnik. In den 70er Jahren begannen systematische Untersuchungen, wobei folgende Grundsätze im Zusammenhang betrachtet wurden:

1. das System ist ausschließlich auf Programmierung in höheren Programmiersprachen ausgerichtet ("in Assembler programmiert niemand mehr"),
2. es wird ein elementarer Befehlssatz vorgesehen, der vollständig "fest verdrahtet" werden kann (hard wired). Das heißt, es gibt keine nachgeordnete Mikroprogrammsteuerung; die einzelnen Angaben in den Befehlsformaten üben unmittelbar Steuerwirkungen in der Hardware aus (direkte Steuerung).

3. um die genannten Prinzipien praktisch nutzen zu können, werden entsprechend hochentwickelte Compiler geschaffen.

Solche Bemühungen wurden maßgeblich durch folgende Sachverhalte angeregt:

- es wurden zunehmend höhere Programmiersprachen angewendet,
- aus Untersuchungen zur Nutzungshäufigkeit von Befehlen in einer Vielzahl compilierter Programme ging hervor, daß elementare Befehle weitaus am häufigsten benutzt werden*),
- mit dem Einsatz der Halbleiterspeicher konnten die Zykluszeiten von Speicher und Verarbeitung einander angeglichen werden. Auch gibt es keine technologisch erzwungene Trennung mehr. Im besonderen wurde es möglich, große Universalregistersätze vorzusehen und Caches zu realisieren, die gute Trefferraten aufweisen und deren Zugriffszeit dem Taktraster der Verarbeitungslogik entspricht (Idealfall: ein Zugriff in jedem Taktzyklus)**).

*): erklärlich: denn es ist recht selten, daß irgendein in der Programmiersprache formulierter Ablauf ohne weiteres durch einen der komplexeren Befehle ausgeführt werden kann - es ist ja keineswegs so, daß die Absichten des Programmierers immer "saugend" zum Befehlsvorrat der Maschine passen -; zudem müßte der Compiler solche "saugend passenden" 1:1-Entsprechungen erst einmal aus dem Programmtext erkennen.

**): damit werden Maschinenbefehle aus dem Cache praktisch so schnell abgearbeitet wie Mikrobefehle aus einem besonderen Steuerspeicher (Abschnitt 2.4.8.).

Die entscheidende Überlegung
Die elementaren Befehle (Laden, Speichern, Addieren, Vergleichen, Verzweigen usw.) werden weitaus am häufigsten genutzt. Solche Befehle seien beispielsweise in der Verarbeitungslogik mit einer Schaltungstiefe s = 10 zu implementieren. Damit läßt sich eine entsprechend kurze Zykluszeit t_c erreichen. Wird durch Hinzufügen komplexerer Befehle die Schaltungstiefe nur um 1 erhöht, so werden *alle* Abläufe um 10% langsamer. Nutzungshäufigkeit und Leistungsfähigkeit der hinzugefügten Befehle müßten folglich so groß sein, daß der Leistungsverlust von 10% mehr als aufgewogen wird.

Die Ziele der “klassischen” (sprich: akademischen) RISC-Entwicklung:

1. der Prozessor soll in jedem Taktzyklus einen Befehl ausführen können (1 MIPS/MHz),
2. die Befehlswirkungen sollen so einfach sein, daß man einen Prozessorschaltkreis in gegebener Technologie mit möglichst vielen MHz betreiben kann (einfache Befehlswirkungen - einfache Schaltungsstrukturen - geringe Schaltungstiefe),
3. alles, was komplizierter ist, erledigt die Software (Compiler + Laufzeitsystem).

In der Praxis sieht es allerdings etwas anders aus...

Was sind komplexe, was sind einfache Befehle?
Die Unterscheidung betrifft vor allem Operationsbefehle. Gemäß dem grundsätzlichen Ablauf eines Operationsbefehls (Operanden lesen - Operanden miteinander verknüpfen - Ergebnis speichern (Heft 1, Abschnitt 2.4.3.)) sind folgende Gesichtspunkte zu berücksichtigen:

1. welche Datenstrukturen soll der Befehl verarbeiten?
2. welche Operation soll er ausführen?
3. wie sollen die Operanden herangeschafft und die Ergebnisse abtransportiert werden?

1. Datenstrukturen
Der klassische Einzelprozessor kann nur einfache Datenstrukturen verarbeiten, z. B. Bytes und Maschinenworte, die als ganze oder natürliche Binärzahlen interpretiert werden (vgl. Kapitel 1 in Heft 1). Hinzu kommen die Gleitkommazahlen und ggf. aus elementaren Strukturen zusammengesetzte Vektoren. Dies ist CISC- und RISC-Architekturen gemeinsam. Darüber hinaus werden von manchen CISC-Architekturen noch Zeichenketten und binär codierte Dezimalzahlen unterstützt. Dies hat man bei RISC-Maschinen in die Software verlagert.

2. Operationen
Ganz elementare Operationen sind solche, die sich grundsätzlich in einem Maschinenzyklus erledigen lassen, z. B. bitweise Verknüpfungen (UND, ODER usw), Verschiebeoperationen sowie die binäre Addition und Subtraktion (vgl. die P/F-Befehlsliste). In CISC-Maschinen hat man typischerweise alle 4 Grundrechenarten über Binärzahlen vorgesehen. Hinzu kommen etliche Sonderfunktionen, die aber auch noch recht elementar sind (z. B. das Auffinden der höchstwertigen Eins in einer Bitkette, Bitfeldoperationen usw.)[*)].

Operationen, die komplizierter sind als Addition oder Subtraktion, laufen typischerweise in mehreren Maschinenzyklen ab. Das gilt für die Multiplikation in vielen Fällen, für die Division praktisch immer (auf Grund des Verhältnisses von Nutzungshäufigkeit und Aufwand lohnt sich keine besondere Hochleistungs-Hardware zum Dividieren). Diese Abläufe werden in vielen Prozessoren durch ein internes Mikroprogramm (Abschnitt 2.4.) gesteuert. In RISC-Prozessoren, die gemäß der "reinen akademischen Lehre" ausgeführt sind, müssen Multiplikation und Division hingegen ausprogrammiert werden. Dazu sind in manchen Architekturen besondere Befehle vorgesehen, in denen die elementaren Verschiebe- und Additions- bzw. Subtraktionsoperationen des schulmäßigen Multiplizierens bzw. Dividierens kombiniert sind (Multiplikationsschritt, Divisionsschritt). Die Erfahrung hat dazu geführt, daß man von dieser “reinen Lehre” abgekommen ist - der Operationsvorrat moderner RISC-Maschinen ist keineswegs geringer als der herkömmlicher CISC-Architekturen.

*): Heft 4 enthält Überblicksdarstellungen typischer Befehlslisten (CISC: Abschnitt 9.1.6. (ergänzend: Tabelle 9.6); RISC: Abschnitt 9.2.5.).

3. Operandenzugriffe
Hierin liegt der tatsächlich bedeutsame Unterschied zwischen CISC und RISC:

CISC : komfortable Operationsbefehle mit Speicher- und Registeroperanden
In den Operationsbefehlen können sowohl Speicher- als auch Registeroperanden angesprochen werden. Zum Adressieren von Speicheroperanden steht typischerweise ein reichhaltiger Vorrat an Adressierungsweisen zur Wahl (siehe hierzu in den Heften 1 und 4 die Abschnitte 2.6.3. und 9.1. sowie Tabelle 9.7). Die Vielzahl der Zugriffsmöglichkeiten und Adressierungsweisen macht die Befehlsabläufe wirklich "komplex" - und zwar bereits von der Befehlsdecodierung an (da die Befehle eine variable Länge haben und deshalb nur mit erheblichem Aufwand in einem einzigen Maschinenzyklus geholt bzw. decodiert werden können).

RISC: Operationen nur mit Registeroperanden (Load-Store-Architektur)
Hier setzt die in der Praxis tatsächlich wirksame Vereinfachung an. RISC-Maschinen haben einen vergleichsweise umfangreichen Universalregistersatz. Alle Operationen betreffen nur Registerinhalte. Das Heranschaffen aus dem Speicher (Laden (Load)) und das Abspeichern (Store) muß mit besonderen Speicherzugriffsbefehlen ausprogrammiert werden. Dies ermöglicht Vereinfachungen bzw. leistungssteigernde Maßnahmen:

1. man kommt mit kurzen, fest formatierten Befehlen aus (vgl. Abschnitt 9.2. n Heft 4), die sich jeweils in einem Maschinenzyklus heranschaffen und decodieren lassen,
2. die Universalregister können als Schnellspeicher ausgenutzt werden (Operanden, die man immer wieder braucht, werden nur einmal geladen - dann wird nur noch auf das jeweilige Register zugegriffen),
3. die Hardware kann einfacher ausgelegt werden (vgl. die Erläuterungen zu Abbildung 2.2),
4. Zugriffs- und Verknüpfungsbefehle können überlappt ausgeführt werden (z. B. stößt ein Ladebefehl den Speicherzugriff nur an; im folgenden Zyklus kann bereits der nächste Befehl ausgeführt werden - sofern er den zu holenden Operanden nicht benötigt).

Dem stehen allerdings ein Nachteil und ein Problem gegenüber:

Der Nachteil
Um eine bestimmte Wirkung zu erbringen, sind mehr Befehls-Bytes heranzuschaffen als in einer CISC-Architektur (Richtwert: 1 CISC-Befehl entspricht 2...4 RISC-Befehlen). Damit wird ausgerechnet der Zugriffsweg zum Speicher (der v. Neumann-Flaschenhals!) besonders belastet. Abhilfe: ein wirksamer Befehls-Cache (mit anderen Worten: das Cache-Subsystem ist von entscheidender Bedeutung für das Leistungsvermögen einer RISC-Maschine - vgl. Abschnitt 2.4.8.).

Das Problem
Dafür, daß die zuvor in den Punkten 2 und 3 genannten Vorzüge tatsächlich wirksam werden, ist der Compiler zuständig - das Leistungsvermögen einer RISC-Maschine hängt also davon ab, daß entsprechend wirksame Compiler verfügbar sind. Die extreme Aus-

legung: der Compiler muß sich um nahezu alles kümmern. Die Konsequenz: bei Übergang auf ein neues Prozessormodell (mit abweichender Auslegung der Hardware) ist eine Neucompilierung der Software notwendig... (deshalb geht man von diesen Extremlösungen wieder ab - siehe auch die Abschnitte 3.2.3. und 3.4.1.).

Wie entstehen Befehlslisten?
Um es kurz zu machen: zumeist aus Erfahrung und Tradition (Zwang zur Abwärtskompatibilität) - allzu viel Wissenschaft steckt nicht dahinter. Damit soll keineswegs gesagt sein, daß jeder Laie ohne weiteres eine brauchbare Befehlsliste ausarbeiten könnte. Der Entscheidungsspielraum eines Entwicklers ist aber doch beachtlich. Viele Entwurfsentscheidungen werden intuitiv getroffen. Andererseits gibt es Vorgaben der Technologie, ein umfangreiches Erfahrungswissen, Forderungen der Software-Entwickler usw. Aus derartigen "Sachzwängen" heraus ergibt es sich, daß die Befehlslisten (und die Architekturen insgesamt) der modernen Hochleistungsprozessoren bei allen Unterschieden im Detail im Grundsatz sehr ähnlich sind*) - und über diese oder jene Ungeschicktheit hilft die Schaltkreistechnologie mit ihren Hunderten von MHz hinweg....(zumeist spielt es gar keine Rolle, wenn eine ungünstige Architekturfestlegung mit einem Programm-"Schlenker" von beispielsweise 5...10 zusätzlichen Befehlen umgangen werden muß (Fachbegriff: Workaround)).

*): es versteht sich geradezu von selbst, daß die Marketing-Leute in ihren Pressemitteilungen und Prospekten dies anders darstellen...*Wir merken uns:* alle Befehlslisten sind "powerful" und "advanced" - manche mehr, manche weniger - und alle sind ganz wichtig, um endlich jeden Kühlschrank mit jeder Klosettspülung und jedem Mobiltelefon (Händü) in Farbe, 3D, Raumklang und Geruch über's Internet weltweit vernetzen zu können.

Wir merken uns weiterhin: Die Unterschiede zwischen CISC und RISC sowie Feinheiten der Architektur im allgemeinen und der Befehlsliste im besonderen sind beim heutigen Stand der Technik praktisch bedeutungslos (abgesehen von extremen Anforderungen - geringste Kosten, höchstes Leistungsvermögen usw.). Viel wichtiger: Verfügbarkeit von Software, Kompatibilität, ggf. anfallende Umstellungskosten, Unterstützung durch eine Vielzahl von Anbietern, Zukunftssicherheit - und nicht zuletzt die "rohe" Verarbeitungsleistung (MIPS).

Hinweise:

1. Die MIPS vergleichen und nicht etwa allein die Taktfrequenzen - es kommt nicht allein darauf an, ob der Prozessor mit 400 oder mit 800 MHz läuft, sondern vor allem darauf, was er in jedem einzelnen Taktzyklus tatsächlich leistet.
2. Es gab immer wieder Versuche, Architekturen zu schaffen, bei denen auf Kosten der reinen Geschwindigkeit der Befehlsausführung (Low Level Performance) andere Gebrauchseigenschaften vorrangig entwickelt wurden (Unterstützung bestimmter Programmiersprachen, hochentwickelte Schutzfunktionen usw.). Diese haben sich aber nicht durchsetzen können. Auch künftig werden solche Eigenschaften kein Ersatz für unzulängliche Verarbeitungsgeschwindigkeit sein.

2. Befehlsablaufsteuerung im Einzelprozessor

Den Begriff des Maschinenzyklus kennen wir bereits; wir wissen, daß der Computer ein sequentieller Automat ist, also eine Maschine, die in einzelnen Zeitschritten arbeitet. Für den Programmierer ist allein wichtig, wie das Programmiermodell die Aufeinanderfolge der Befehle spezifiziert. Wenn wir uns jedoch näher mit technischen und mit Leistungsfragen befassen wollen, so müssen wir uns dafür interessieren, wie die Befehlsabläufe in der Hardware gesteuert werden.

2.1. Was sind Steuerwirkungen?

Die zu steuernden Einrichtungen der Hardware sind über Steuersignalleitungen an die Steuerschaltungen angeschlossen. In umgekehrter Richtung führen Bedingungssignalleitungen zu den Steuerschaltungen zurück. Steuersignale bewirken grundsätzlich irgendeine Form der *Auswahl* oder der *Erlaubnis*. In typischen Prozessorschaltungen ist zwischen verschiedenen Datenwegen, Operationen usw. auszuwählen, und es ist zu bestimmen, ob im jeweiligen Taktzyklus anstehende Signalbelegungen (z. B. Verarbeitungsresultate) in Register bzw. Flipflops übernommen werden sollen oder nicht. Manche Folgen von Steuerwirkungen hängen von Bedingungen ab, manche nicht. Demgemäß sind die Steuerwirkungen des nachfolgenden Zyklus aus jenen des aktuellen Zyklus zu bestimmen, erforderlichenfalls unter Berücksichtigung von Bedingungsangaben.

Die Wirkungen von Steuersignalen wollen wir anhand eines einfachen Prozessors veranschaulichen. Abbildung 2.1 zeigt das Blockschaltbild der Hardware, die zu steuern ist: ein Universalregistersatz, ein Adreßrechenwerk, eine universelle Arithmetik-Logik-Einheit (Arithmetic Logic Unit, ALU) sowie einige Hardware-Register. Einige weitere Einzelheiten im Überblick:

- der Universalregistersatz ist eine RAM-Anordnung, also ein kleiner adressierbarer Schnellspeicher,
- der Befehlszähler ist Bestandteil des Universalregistersatzes,
- das Adreßrechenwerk dient zu allen Arten der Adreßrechnung unter Einschluß der Befehlszählung,
- die Arithmetik-Logik-Einheit (ALU) ist der "Kern" des Operationswerkes. Sie kann jeweils eine ausgewählte Elementaroperation (Addition, bitweise logische Verknüpfung, Verschiebung usw.) ausführen,
- in der ALU sind Funktionen vorgesehen, um Eingangsbelegungen ohne Veränderung zu den Ausgängen durchschalten zu können (Transportoperation, Gate Thru),
- für den Informationstransport sind zwei Bussysteme vorgesehen (A-Bus für die Argumente bzw. Operanden, R-Bus für die Resultate). Jeweils eine eingangsseitig mit einem Bus verbundene Einrichtung (Register oder Koppelstufe) kann Information auf den Bus aufschalten. Alle ausgangsseitig an einen Bus angeschlossenen Einrichtungen können die Busbelegung gleichzeitig empfangen. Beide Bussysteme werden von der Steuereinheit (vgl. Abbildung 1.1) gesteuert.

Im folgenden wollen wir drei Steuerungsprinzipien (direkte, sequentielle und Mikroprogrammsteuerung) sowohl allgemein als auch am Beispiel von Abbildung 2.1 behandeln.

Abbildung 2.1 Zu steuernde Hardware eines (hypothetischen) Einzelprozessors (Register-Transfer-Ebene)

2.2. Direkte Steuerung

Die im Befehl codierten Angaben üben die Steuerwirkungen unmittelbar aus. Im einfachsten Fall dauert jeder Befehl nur einen Maschinenzyklus. Zu Beginn des Zyklus wird das Befehlsregister geladen. Danach wird sofort der Befehlszählerinhalt erhöht (um den Folgebefehl zu adressieren). Jeder Befehl hat nur eine Wirkung, die ausschließlich durch kombinatorische Verknüpfungen, also nicht in mehreren Zeitschritten, erbracht wird. Am Ende des Zyklus werden die Verknüpfungsresultate gespeichert. Aus technischer Sicht kann es notwendig sein, den Maschinenzyklus in mehrere Taktphasen aufzulösen, beispielsweise für Speicherzugriffe oder Verzweigungen, und erforderlichenfalls die einzelnen Phasen durch Wartezustände (Wait States) zu verlängern.

Ist die Hardware nach Abbildung 2.1 direkt zu steuern? - Das hängt davon ab, welche Maschinenbefehle implementiert werden sollen. Die Hardware ist durch Mehrfachnutzung von Schaltmitteln und Signalwegen gekennzeichnet. Somit erfordern übliche Maschinenbefehle zumeist mehrere Taktzyklen.

Bei direkter Steuerung müssen Befehlsgestaltung und Hardware "saugend" zueinander passen. Die Hardware muß so ausgelegt sein, daß sie alle im jeweiligen Befehl spezifizierten Wirkungen in einem Maschinenzyklus erbringen kann. Die Schaltung in Abbildung 2.1 müßte so geändert werden, daß sowohl das Adreßrechenwerk als auch die ALU unabhängigen Zugriff auf den Universalregistersatz haben. Das ist technisch möglich und auch in Hochleistungsprozessoren realisiert. Beispielsweise kann man Adreß- und Datenregister getrennt aufbauen oder im Falle eines wirklichen *Universal*registersatzes unabhängige Zugriffswege für zwei Lesezugriffe und einen Schreibzugriff vorsehen.

Die Alternative: RISC
Wenn man alle Speicherzugriffsbefehle auf bloße Transporte beschränkt (Laden und Speichern von Registern - Load-Store-Architektur), kann man die ALU ohne Leistungsverlust zur Adreßrechnung ausnutzen. An die Stelle des Adreßrechenwerks tritt ein einfacher Hardware-Befehlszähler. Die Operationsbefehle betreffen dann ausschließlich Registeroperanden bzw. Direktwerte aus dem Befehlsregister. Damit haben wir einen typischen RISC-Prozessor geschaffen. Abbildung 2.2 veranschaulicht Blockschaltbild und einige Befehlsformate.

Abbildung 2.2 Blockschaltbild und Befehlsformate eines (hypothetischen) RISC-Prozessors

2.3. Sequentielle Steuerung (Folgesteuerung)

Die Befehle können vergleichsweise komplexe Wirkungen erbringen. Die Angaben im Befehl (z. B. Operationscodes) wirken dabei auf Steuerschaltungen, die ihrerseits bestimmen, welche Informationswandlungen, Transporte usw. in jedem einzelnen Maschinentakt stattfinden. Die Steuerschaltungen selbst bestehen wiederum aus Flipflops und kombinatorischen Schaltungen. Die Steuerungsabläufe werden gleichsam durch die "Verdrahtung" der Schaltmittel bestimmt.

Sowohl bei direkter als auch bei sequentieller Steuerung spricht man deshalb - im Gegensatz zur Mikroprogrammsteuerung - auch von "hart verdrahteter" Steuerung (Hard Wired Control).

Als Beispiel wollen wir einen Additionsbefehl betrachten, der einen Registeroperanden zu einem Speicheroperanden addiert und das Resultat wieder im angegebenen Register ablegt (Registerinhalt := Registerinhalt + Inhalt einer Speicheradresse). Die Speicheradresse wird durch Addition eines weiteren Registerinhaltes (Basisadresse) zu einem Direktwert im Befehl (Offset) ermittelt (vgl. Abschnitt 2.6.3. in Heft 1). Abbildung 2.3 veranschaulicht sowohl das Befehlsformat als auch die einzelnen Steuerschritte für die Hardware von Abbildung 2.1.

Abbildung 2.3 Sequentielle Steuerung eines Additionsbefehls (Hardware gemäß Abbildung 2.1)

Die sequentielle Steuerung wurde von Anfang an in Rechnern aller Leistungsklassen eingesetzt, auch in den ersten Mikroprozessoren. Sie ist gut geeignet, Befehle einer mittleren Komplexität wirtschaftlich zu implementieren. Die in Abbildung 2.3 skizzierte Befehlswirkung*) mag als Beispiel dafür dienen, was unter mittlerer Komplexität zu verstehen ist (komplexer als RISC, aber einfacher als die hochentwickelten CISC-Architekturen, wie beispielsweise IA-32 oder VAX).

*): “mittlere Komplexität” bedeutet also hier die Beschränkung auf das Schema Registerin-halt: = Registerinhalt **op** Inhalt einer Speicheradresse und auf die Speicheradressierung nach dem Prinzip Basis + Displacement.

2.4. Mikroprogrammsteuerung

Mikroprogrammsteuerung bedeutet, die Befehlswirkungen zu erbringen, indem die Angaben des Befehls von einem weiteren Programm, dem sogenannten Mikroprogramm, interpretiert werden. Das Mikroprogramm beruht auf Mikrobefehlen, die in Formatgestaltung und Wirkungsweise genau an die zu steuernde Hardware angepaßt sind. Die Ausführung der Mikrobefehle wird direkt gesteuert. Grundlage einer Mikroprogrammsteuerung ist der Mikroprogramm- oder Steuerspeicher (Control Storage). Er ist üblicherweise als Festwert-

speicher (ROM) ausgebildet. Die jeweils aktiven Mikrobefehle werden aus dem Steuerspeicher in ein Mikrobefehlsregister geladen. Dieses Register wirkt direkt auf die zu steuernde Hardware.

2.4.1. Adressierung der Mikrobefehle

Hierfür gibt es vielfältige, bisweilen recht trickreiche Prinzipien (deren wesentliches Ziel darin besteht, Zeitverluste bei Verzweigungen zu vermeiden). Im einfachsten Fall werden Mikrobefehle genau so adressiert wie "gewöhnliche" Maschinenbefehle: es gibt einen Mikrobefehls-Adressenzähler als Entsprechung zum Befehlszähler, und manche Mikrobefehle können Verzweigungs-Anweisungen enthalten, die bewirken, daß der Adressenzähler mit Angaben aus dem Mikrobefehlsregister überladen wird.

Zur Entwicklungsgeschichte
Die Mikroprogrammsteuerung wurde entwickelt, um sequentielle Steuerschaltungen, die meist kompliziert und unübersichtlich sind, auf reguläre Weise gestalten zu können. Sie bietet die Möglichkeit, auch sehr komplexe Befehlslisten mit einfacher Hardware zu implementieren. Von den 60er Jahren an wurde das Prinzip genutzt, *Rechnerfamilien* anzubieten, das heißt Baureihen kompatibler Maschinen in einem sehr weiten Preis- und Leistungsbereich (verbreitete Rechnerfamilien waren z. B. die Systeme /360 und /370 von IBM sowie die VAX-Reihe von DEC).

2.4.2. Mikrobefehlsformate

Horizontale Mikrobefehle
Ein Mikrobefehlsformat heißt horizontal, wenn alle Steuerwirkungen, die in einem Mikrobefehlszyklus überhaupt ausgeführt werden können, in einem einzigen Mikrobefehl untergebracht sind. Solche Mikrobefehle können hundert und mehr Bits lang sein. Abbildung 2.4 veranschaulicht ein Mikroprogrammsteuerwerk und ein horizontales Mikrobefehlsformat für die Hardware von Abbildung 2.1.

Abbildung 2.4 Beispiel einer Mikroprogrammsteuerung

Vertikale Mikrobefehle
Der einzelne Mikrobefehl ist kürzer und erbringt jeweils nur einige der insgesamt erforderlichen Steuerwirkungen. Eine typische Unterteilung ist die in Mikrobefehle zur Datenweg- bzw. Operationssteuerung und solche zur Bedingungsabfrage und Verzweigung.

2.4.3. Mikroprogrammsteuerung für höchste Leistung

Für höchste Verarbeitungsleistung braucht man eine "horizontale" Mikroprogrammsteuerung sowie eine Verarbeitungshardware mit entsprechend parallel angeordneten Verknüpfungsschaltungen und Datenwegen. Im Extremfall entspricht ein Maschinenbefehl einem einzigen Mikrobefehl. Das bedeutet, daß der Operationscode unmittelbar einen Mikrobefehl im Mikroprogrammspeicher adressiert und daß alle Befehlswirkungen in einem einzigen Taktzyklus erbracht werden. Dies ist beispielsweise in den 486- und Penti-

um-Prozessoren für viele Befehle vorgesehen. So kann man - wenngleich mit erheblichem Aufwand - auf Grundlage einer traditionellen CISC-Architektur durchaus dem idealen Leistungsvermögen des klassischen Einzelprozessors nahekommen: in jedem Taktzyklus einen Befehl auszuführen (mit anderen Worten: eine solche CISC-Maschine kann - was die MIPS/MHz angeht*) - durchaus ebenso leistungsfähig ausgelegt werden wie eine RISC-Maschine).

*): hierbei wird allerdings vernachlässigt, was die einzelnen Befehle tatsächlich leisten.

2.4.4. Direkte, sequentielle und Mikroprogrammsteuerung - was ist schneller?

Vergleichende Betrachtungen müssen sich auf vergleichbare Größenordnungen der Aufwendungen beziehen.

1. kostengünstige Prozessoren
Viele Befehle müssen in mehreren Taktzyklen ausgeführt werden. Dies erfordert entsprechenden Aufwand in den Steuerschaltungen. Architekturen mit auch nur halbwegs komplizierten Wirkprinzipien (hierzu gehört u. a. bereits x86) kann man praktisch nur auf Grundlage der Mikroprogrammsteuerung kostengünstig implementieren.

2. Hochleistungsprozessoren
Das Ideal: möglichst alle Wirkungen des einzelnen Befehls in einem Taktzyklus zu erbringen. Das Problem ist hier weniger der Aufwand, sondern die interne Kompliziertheit. Man bevorzugt die Mikroprogrammsteuerung vor allem deshalb, weil sie auf reguläre Strukturen führt und somit die Kompliziertheit beherrschbar macht.

Etwas Theorie:

1. die Steuerung des Prozessors ist ein sequentieller Automat, der in jedem Taktzyklus die jeweiligen Steuersignale zu erregen hat,
2. in einer direkten oder sequentiellen Steuerung werden die Steuersignale mit kombinatorischen Netzwerken gebildet, die aus Gattern aufgebaut sind,
3. in einer Mikroprogrammsteuerung werden die Steuersignalbelegungen aus dem Steuerspeicher abgerufen,
4. jeder sequentielle Automat nimmt in jedem Taktzyklus einen bestimmten Zustand ein. Das Problem: welcher Zustand soll im jeweils nachfolgenden Taktzyklus eingenommen werden? Nicht selten ist dabei einer von x möglichen Folgezuständen auszuwählen. Ein typisches Programm (oder ein “vertikales” Mikroprogramm) kann einen von 2 möglichen Folgezuständen auswählen (bedingte Verzweigung), hochleistungsfähige Mikroprogrammsteuerwerke typischerweise einen von 4, 8 oder 16 Folgezuständen. Stehen nun mehr Folgezustände zur Wahl als man mit einem Mikrobefehl auswählen kann, so müssen Zwischenzustände eingefügt werden (Geschwindigkeitsverlust). Kombinatorische Gatternetzwerke kann man hingegen - zumindest dem Prinzip nach - so auslegen, daß jeder beliebige Folgezustand innerhalb eines einzigen Taktzyklus bestimmt werden kann.

5. die Zugriffszeit einer Speicheranordnung ist typischerweise größer als die Verzögerungszeit eines Gatternetzwerks,
6. nimmt man eine bestimmte Schaltkreistechnologie als gegeben an, so könnte man eine direkt oder sequentiell gesteuerte Maschine mit einem schnelleren Takt betreiben als eine mikroprogrammgesteuerte,
7. eine Speicheranordnung ist technologisch viel eleganter - und meistens auch kostengünstiger herzustellen - als ein Sammelsurium von Gatternetzwerken (sie hat eine reguläre Struktur), zudem ist ein Speicherinhalt leicht zu ändern (EPROM, Flash-ROM, RAM).

Zur Erklärung ein Beispiel: wir wollen im Befehlsablauf einen Operanden aus dem Speicher lesen (aktueller Zustand). Welcher Zustand als nächster einzunehmen ist, hängt nun davon ab, was beim Lesen passiert - und es kann allerhand passieren (Tabelle 2.1, Abbildung 2.5).

mögliche Situationen beim Lesen eines Operanden (Auswahl)	typische Reaktionen
es ist alles o. k.	Befehlsablauf fortsetzen
die Zugriffsadresse ist keine integrale Adresse	je nach Architektur: entweder Fehlerbedingung auslösen oder Zugriff emulieren (2 Zugriffe ausführen und den Operanden aus 2 Abschnitten zusammensetzen)
ein TLB Miss	einen neuen TLB-Eintrag aufbauen (Hardware/Mikroprogramm)
eine Seitenausnahme	Ausnahmebedingung auslösen
die Daten müssen erst geholt werden	Wartezustand
ein Speicherfehler	Fehlerbedingung auslösen

Tabelle 2.1 Mögliche Folgezustände beim Operandenlesen

Hier haben wir die Wahl zwischen 6 Folgezuständen. Kann die Mikroprogrammsteuerung nur zwischen 2 Richtungen wählen (wie übliche bedingte Verzweigungsbefehle), so müßte etwa so programmiert werden:

wenn o. k., dann fortsetzen,
wenn keine integrale Adresse, dann...
wenn TLB Miss, dann...
usw.

Jede dieser Entscheidungen kostet aber einen Mikrobefehl und damit einen Taktzyklus. Demgegenüber könnte eine hart verdrahtete Steuerung bereits während des ursprünglichen Taktzyklus den jeweiligen Folgezustand direkt auswählen (indem alle Bedingungen kombinatorisch (also parallel) miteinander verknüpft werden).

Abbildung 2.5 Bedingungsauswertung beim Operandenlesen

Erklärung zu Abbildung 2.5:

a) die ankommenden Bedingungssignale (aus der Speicheranschaltung),
b) Bildung der o.k.-Bedingung (wird immer dann wirksam, wenn aus der Speicheranschaltung "nichts" gemeldet wird),
c) Bedingungsabfrage (durch Mikrobefehle) - erfordert mehrere Maschinenzyklen,
d) Bestimmung des Folgezustandes mittels parallel angeordneter kombinatorischer Verknüpfungen - erfordert nur einen einzigen Maschinenzyklus.

Hinweise:

1. Die in der Praxis sehr wichtigen Fragen des Vorranges (Beispiel: was tun, wenn gleichzeitig eine nichtintegrale Adresse und ein TLB Miss gemeldet werden?) wollen wir hier nicht betrachten.
2. Achten Sie auf Abbildung 2.5b. Treten keine besonderen Bedingungen auf, so sollte der Befehlsablauf sofort (im nächsten Maschinenzyklus) weitergeführt werden. Es erfordert aber ein wenig Hardware, um diesen Zustand (o.k.) zu erkennen. Hätten wir das NOR-Gatter eingespart, so müßte das Mikroprogramm alle Hiobsbotschaften aus der Speicheranschaltung (Abbildung 2.5a) zunächst abfragen, um schließlich darauf zu kommen, daß alles in Ordnung ist.

Wir merken uns:
Der Theorie nach ist die Mikroprogrammsteuerung vergleichsweise langsamer: weil (1) Folgezustände oftmals nur sequentiell (in mehreren Schritten) bestimmt werden können, und weil (2) die Zugriffszeit des Steuerspeichers typischerweise größer ist als die Verzögerungszeit eines Gatternetzwerks. Sie wird aber oft eingesetzt, weil sie kostengünstig zu fertigen ist und weil sie es ermöglicht, komplizierte Wirkprinzipien der Architektur praxisgerecht (in überschaubarer Weise, mit angemessenem Aufwand usw.) zu implementieren.

Mikrobefehle in ganz modernen Hochleistungsprozessoren
Unsere Überlegungen gelten für die gleichsam klassische Mikroprogrammsteuerung. Worauf es vor allem ankommt, sind die Verzweigungen. Die Steuerwerke der modernen Hochleistungsprozessoren (Abschnitt 4.2.) unterscheiden sich aber durch einige Besonderheiten von der herkömmlichen Mikroprogrammsteuerung:

die Mikrobefehle werden an unabhängige Verarbeitungseinrichtungen ausgegeben und dort selbständig abgearbeitet,
viele Maschinenbefehle lassen sich in einen einzigen Mikrobefehl umsetzen. Darüber hinaus kommt man oftmals mit wenigen Mikrobefehlen aus (bis zu 4 sind typisch), wobei es keine Verzweigungen gibt.
wenn sich ein architekturseitiger Ablauf nicht im Rahmen dieses Schemas erledigen läßt, wird wieder die herkömmliche Mikroprogrammsteuerung wirksam - und dann dauert es eben seine Zeit...

2.4.5. Mikroprogramme ändern

Bei Entwicklungsarbeiten aller Art können Fehler unterlaufen. Entscheidend ist deshalb oftmals der Aufwand, der erforderlich ist, um solche Fehler zu beseitigen (Änderungsaufwand):

- bei einer direkten oder sequentiellen Steuerung sind Entwurfsfehler nur durch Änderungen in der Hardware zu beseitigen (bedeutet heutzutage: Fertigung einer neuen Version des Schaltkreises!),
- bei einer Mikroprogrammsteuerung ist der Entwurfsfehler aber typischerweise ein Programmierfehler - und folglich auch durch Programmänderung abzustellen (man braucht keine neue Hardware*).

*): man kann sogar wirkliche Hardwarefehler durch Mikroprogrammierung umgehen.

In den vergangenen Jahrzehnten haben sich deshalb die Hersteller von Mainframes und anderen großen Systemen viel Mühe gegeben, einen organisierten Änderungsdienst bis hin zumAnwender aufzubauen (u. a. wurden die allerersten Disketten eigens hierfür entwickelt). Seit etlichen Jahren gehört auch die Fernänderung zum Stand der Technik.

Steuerungsprinzipien und Service
Wenn der Prozessor einen ganzen Schrank belegt, hat das Steuerungsprinzip eine beachtliche Bedeutung. Eine komplizierte sequentielle Steuerung ist dann der Schrecken des Servicetechnikers. Die Mikroprogrammsteuerung hat hier drei Vorteile: (1) vergleichsweise Einfachheit der Hardware, (2) Möglichkeit der mikrobefehlsweisen Ablaufverfolgung, (3) Möglichkeit zur Ausführung von *Testmikroprogrammen*.

Ist der gesamte Prozessor auf einem Schaltkreis untergebracht, so ist das interne Steuerungsprinzip für den Servicetechniker bedeutungslos. Die Mikroprogrammsteuerung hat hier lediglich den Vorteil, daß sie ohne nennenswerten Aufwand recht umfassende Eigentestvorkehrungen ermöglicht. (Eingebaute Eigentests erleichtern oft die Entscheidung, ob der Prozessor selbst oder ob seine Umgebung fehlerverdächtig ist.) Die Praxisfrage: in welcher Weise sind diese Vorkehrungen dem Servicetechniker zugänglich? (Lassen sich die Tests auslösen?, werden Fehlermeldungen angezeigt?, gibt es eine brauchbare Fehlersuchdokumentation? usw.)

2.4.6. Mikroprogrammierung für den Anwender?

Da man mit Mikrobefehlen jeden Taktzyklus in der Hardware genau unter Kontrolle hat, kann man eine gegebene Schaltungsstruktur viel besser ausnutzen, indem man den "Zwischenschritt" über die Maschinenbefehle vermeidet und das Anwendungsprogramm gleich als Mikroprogramm schreibt. Dieser naheliegende Gedanke hat zu verschiedenen Maschinen geführt, deren Mikrobefehlslisten dem Anwender zugänglich sind. Dies hat sich aber, von Sonderanwendungen abgesehen, am Markt nicht durchsetzen können. Wichtigster Grund: mangelnde Kompatibilität. Die Gestaltung der Mikrobefehle ist sehr maschinenspezifisch, und die zweckmäßige Ausnutzung aller Möglichkeiten und Tricks ist eine Kunst für sich; ein

gutes Mikroprogramm zu schreiben, ist viel mühsamer als das Programmieren in einer höheren Programmiersprache. Bei Übergang auf eine andere Hardware - mit wiederum spezifischer Mikrobefehlsgestaltung - muß die viele Mühe als weitgehend verloren gelten. Demgegenüber hat es sich sehr bewährt, von seiten der Hersteller wichtige Abschnitte von Systemprogrammen ins Mikroprogramm zu verlagern (davon wurde beispielsweise bei der Weiterentwicklung des Systems /370 ausgiebig Gebrauch gemacht).

Der Leistungsgewinn:
Die Verlagerung ins Mikroprogramm bringt erfahrungsgemäß eine 2-fache bis etwa 40-fache Beschleunigung (verglichen mit den "richtigen" Maschinenbefehlen (die auf der gleichen Hardware laufen)).

2.4.7. Die Mikroprogrammsteuerung der PC-Prozessoren

Alle modernen Prozessoren haben irgend eine Form der Mikroprogrammsteuerung - wenngleich nicht immer unter diesem Namen (vgl. die Abschnitte 4.2.2. und 4.2.3.).

Was es - beim Stand der Technik - nicht gibt:

- die Möglichkeit, Mikroprogramme zu laden (somit lassen sich auch keine eigenen Mikroprogramme einbringen),
- die Möglichkeit, Testmikroprogramme gezielt laufen zu lassen.

Was es in manchen Prozessoren aber gibt:

- Testmikroprogramme, die nach dem Einschalten durchlaufen werden (im Rahmen des eingebauten Selbsttests BIST),
- die Möglichkeit, Mikroprogrammänderungen vergleichsweise geringen Umfangs nachzuschieben (Beispiel: die P6-Prozessoren (Abschnitt 9.1.8. in Heft 4)).

2.4.8. Mikroprogrammsteuerung, RISC, VLIW?

Vergleichen Sie ein typisches Mikrobefehlsformat (Abbildung 2.4) mit den Formaten typischer RISC-Befehle (Abbildung 2.2; Abbildungen 9.10 und 9.18 in Heft 4). Alle Formate enthalten Angaben zur Ausübung *direkter* Steuerwirkungen. Im Format von Abbildung 2.4 ist alles angegeben, was sich mit der Hardware in einem Maschinenzyklus anstellen läßt, in jedem RISC-Format hingegen nur jeweils ein Teil. Wenn wir aber die Mikrobefehlswirkungen konsequent "vertikal" einteilen (Trennung zwischen Operations-, Transport- und Verzweigungsmikrobefehlen), so ergeben sich Mikrobefehlsformate, die kaum noch von RISC-Befehlen zu unterscheiden sind. Der einzige wesentliche Unterschied: Mikrobefehle stehen im Steuerspeicher, RISC-Befehle im Arbeitsspeicher. Dieser Unterschied betrifft in erster Linie die Geschwindigkeit, machmal auch die Zugriffsbreite. Für die Mikrobefehle gibt es unabhängige Adressen- und Datenwege (die "Mikro-Architektur" ist also eine Harvard-Architektur), der (vergleichsweise kleine) Mikroprogrammspeicher hat eine sehr kurze Zykluszeit, und seine Zugriffsbreite kann genau auf das Mikrobefehlsformat abgestimmt werden. Könnte man bei einer RISC-Maschine ähnliche Verhältnisse herstellen, so

könnte man die Leistungs-Vorteile der Mikroprogrammierung vereinigen mit den anwendungspraktischen Vorteilen einer grundsätzlich hardware-unabhängigen Architekturspezifikation (Kompatibilität). Und genau das ist möglich, wenn für die RISC-Befehle ein hinreichend leistungsfähiger Cache vorgesehen wird (der Befehls-Cache entspricht praktisch dem Mikroprogrammspeicher von Abbildung 2.4). Noch besser: eine Anordnung aus Befehls- und Datencache (Abbildung 2.6). Eine solche Anordnung entspricht offensichtlich einer Harvard-Maschine (vgl. Abbildung 1.4b).

Abbildung 2.6 Hochleistungsprozessor mit Befehls- und Datencache

Wir merken uns:

1. durch entsprechende Anordnung von Befehls- und Datencaches läßt sich auch eine v. Neumann-Architektur im Sinne einer Harvard-Maschine (also mit unabhängigen Zugriffswegen auf Daten und Befehle) implementieren.
2. aus einem Befehls-Cache geholte und somit extrem schnell ablaufende RISC-Elementarbefehle sind nahezu ebenso leistungsfähig wie Mikrobefehle. Die besonderen Vorteile im Vergleich zum Mikroprogramm:
   - RISC-Befehle sind von der Hardware unabhängig (im Idealfall: vollkommen, in der Praxis: im wesentlichen - vgl. die Abschnitte 3.2.3. und 3.4.1.),
   - RISC-Architekturen bilden - wegen ihrer vergleichsweise überschaubaren, regulären Befehlsstrukturen - gute Ziele für Compiler.

Hinweis:
Nahezu alle RISC-Programme werden in höheren Programmiersprachen geschrieben. Vgl. demgegenüber die hardwarespezifischen und oftmals trickreichen Mikrobefehlsformate (das von Abbildung 2.4 ist noch vergleichsweise einfach) - es verlangt Intelligenz im wahrsten Sinne des Wortes, so etwas geschickt auszunutzen (ein erfahrener Entwickler schafft im Jahr Mikroprogramme mit einem Umfang von wenigen hundert bis zu einigen tausend Mikrobefehlen).

Mehrere Befehle auf einmal: VLIW
Steht “genügend Hardware” zur Verfügung, kann man mehrere RISC-Befehle gleichzeitig holen und parallel ausführen. Ein solches “Bündel” von RISC-Befehlen stellt gleichsam einen extrem langen Befehl bzw. etwas Ähnliches wie einen horizontalen Mikrobefehl dar. Näheres in Abschnitt 3.4.2.

Wegen der genannten Vorteile - Hardware-Unabhängigkeit, gute Eignung zur Hochsprachen-Programmierung - werden Lösungen auf Grundlagen von RISC- und VLIW-Prinzipien mehr und mehr bevorzugt.

2.5. Architekturnachbildung (Emulation)

Emulation bedeutet hier, Befehlswirkungen mit Software nachzubilden. Das heißt, wir schreiben ein Programm, das das jeweils auszuführende Programm einer bestimmten Architektur (der Zielarchitektur) wie einen Datenbereich behandelt, aus dem es Befehl für Befehl liest und zwecks Ausführung interpretiert.

Emulation und Simulation
Die Übergänge zwischen den Begriffen sind fließend. Der typische Unterschied:

- die Simulation betrifft Einzelheiten des Befehlsablaufs der Zielarchitektur - ein Prozessor-Simulator dient dazu, Befehlswirkungen im einzelnen zu verfolgen, statistische Daten zu erheben (z. B. zur Nutzungshäufigkeit von Befehlen, zur Wahrscheinlichkeit von Verzweigungen und zu Trefferraten), die Brauchbarkeit der Architektur nachzuweisen, ohne einen entsprechenden Prozessorschaltkreis fertigen zu müssen usw. Die Verarbeitungsgeschwindigkeit spielt hier eine eher untergeordnete Rolle; wichtig ist vor allem, an die gewünschten Einzelheiten heranzukommen.
- bei der Emulation geht es hingegen um reine Verarbeitungsleistung. Anwendung: die Befehlswirkungen der Zielarchitektur nachzubilden, um die Kompatibilität zu gewährleisten (z. B. um vorhandene Software auch auf Systemen einer anderen Archiektur nutzen zu können).

Beispiel: die Macintosh-PCs
Die ersten Modelle hatten Prozessoren der 68k-Familie (Motorola). In den 90er Jahren wurde aber auf PowerPC-Prozessoren umgestellt. Um die vorhandene Software weiternutzen zu können, wurde die 68k-Architektur emuliert.

Emulation mit Mikroprogrammen
Zunächst Ansichtssache: man kann die Wirkprinzipien der jeweiligen Mikroprogrammsteuerung als “primäre” Architektur ansehen und davon sprechen, daß das Mikroprogramm eine Zielarchitektur (z. B. IA-32) emuliert. Verfolgt man den Gedanken weiter, so liegt es nahe, mikroprogrammgesteuerte Maschinen gezielt auf Vielseitigkeit hin auszulegen.

Die eierlegende Wollmilchsau?
So etwas wurde tatsächlich immer wieder probiert. Der Grundgedanke - eine Hardware-Plattform, mehrere Architekturen (stellen Sie sich eine Blechkiste vor, die wie ein PC aussieht, aber wahlweise als Wintel-PC, als S/390, als VAX, als Alpha-Workstation, als Macintosh usw. betrieben werden kann). Besonders schwierig scheint es auch nicht zu sein: ein Prozessor mit einer besonders leistungsfähigen Mikroprogrammsteuerung - der Rest ist dann eine Fleißaufgabe (je Architektur ist ein Satz Mikroprogramme zu schreiben). Das Leistungsvermögen solcher Prozessoren war aber bisher immer enttäuschend.

Was wirkt sich so bremsend aus? - Es sind weniger die elementaren Befehlswirkungen (z. B. das Addieren von Binärzahlen), sondern vielmehr die zahlreichen Feinheiten, in denen sich die einzelnen Architekturen voneinander unterscheiden. In mikroprogrammierten Prozessoren, die auf eine einzelne Architektur hin entworfen werden, setzt man hierfür gezielt Hardware ein - die eben im universellen Prozessor fehlt und durch zeitaufwendige Mikroprogrammabläufe ersetzt werden muß.

Beispiel: wir greifen auf Abbildung 2.5 zurück. Hier geht es u. a. darum, eine Seitenausnahme auszulösen, wenn die betreffende Seite nicht im Arbeitsspeicher anwesend ist. Die Tabellenstrukturen der Seitenverwaltung sind aber in jeder Architektur anders aufgebaut. In letzter Konsequenz ist die Anwesenheits-Anzeige im jeweiligen Seitentabelleneintrag

auszuwerten (sei es direkt, sei es über den TLB). In der IA-32-Architektur ist dies die Bitposition 0 (vgl. Abbildung 1.44). In der Sparc-Architektur hingegen ist die Anwesenheit der Seite in den Bits 1 und 0 codiert (vgl. Abbildung 9.15 in Heft 4). In anderen Architekturen kann diese Anzeige in ganz anderen Bitpositionen untergebracht und ganz anders codiert sein. Entwerfen wir nun beispielsweise einen mikroprogrammgesteuerten IA-32-Prozessor, so schließen wir an den betreffenden Signalweg der Bitposition 0 eine einfache Hardware an, die die in Rede stehende Bedingung auswertet. Eine als "eierlegende Wollmilchsau" ausgelegte Maschine müßte entweder eine Erkennungs-Hardware haben, die sich auf jeden nur möglichen Fall einstellen läßt (Aufwand!), oder sie müßte die betreffende Bedingung ausschließlich durch programmseitiges Abfragen erkennen - und das kostet wiederum Zeit. Dementsprechend (wir haben hier nur ein wirklich einfaches Beispiel betrachtet - in der Praxis gibt es noch ganz andere Spitzfindigkeiten) sind solche universellen Maschinen einer Einzweck-Hardware leistungsmäßig weit unterlegen.

Allerdings hat man sich auch hier etwas Neues einfallen lassen (Abschnitt 4.2.4.).

3. Leistungssteigerung im Einzelprozessor

3.1. Überblick

Welche Möglichkeiten gibt es, das Leistungsvermögen eines Einzelprozessors zu erhöhen? - Wir wollen hierbei die Schaltkreistechnologie - und damit die höchstmögliche Taktfrequenz - als gegeben hinnehmen. Offensichtlich müßte es auf zwei Wegen gelingen:

1. wir versuchen, dem Ziel soweit wie möglich nahezukommen, in jedem internen Taktzyklus einen Befehl auszuführen. Das bedeutet, wir müssen uns Schaltungsmaßnahmen einfallen lassen, um die Zeitverluste im Befehlsablauf zu vermeiden.
2. wir versuchen, in einem Taktzyklus mehr zu leisten, als nur einen einzigen der üblichen Befehle auszuführen. Auch hierfür bieten sich zwei Möglichkeiten an:
   a) wir führen nach wie vor nur einen einzigen Befehl aus, der aber weit mehr als ein gewöhnlicher Befehl leistet,
   b) wir führen mehrere gewöhnliche Befehle gleichzeitig (parallel) aus.

Wichtige Prinzipien und Fachbegriffe:

Weg 1
Kennzeichnend für den klassischen Einzelprozessor ist die geradezu langweilige Arbeitsweise: Befehl holen - Operanden holen - Operanden miteinander verknüpfen - Ergebnis wegschaffen - den nächsten Befehl holen usw. Das Holen und Wegschaffen ist mit Speicherzugriffen verbunden, die oftmals Zeit kosten (Wartezustände). Es liegt also nahe, hier anzusetzen:

- durch Verringerung der Wartezeiten bei Speicherzugriffen,
- durch zeitlich überlappte Ausführung von Abläufen, die an sich aufeinander folgen (Befehlspipelining).

Weg 2 a)
Übliche Transportbefehle bewegen eine einzige elementare Datenstruktur, übliche Verarbeitungsbefehle verknüpfen zwei Operanden miteinander, um ein Ergebnis zu bilden. Die naheliegenden Ansätze:

- die Befehle betreffen mehr als eine Datenstruktur, mehr als ein Operandenpaar usw. (Mehrfachverarbeitung). Der übliche Fachbegriff: SIMD (Single Instruction, Multiple Data; vgl. auch Tabelle 8.2 in Heft 3).
- es werden Datenstrukturen verarbeitet, die aus jeweils mehreren einzelnen Operanden bestehen (Vektorverarbeitung).

Weg 2 b)
Hierfür kann eine Erfahrungstatsache ausgenutzt werden: der sog. innewohnende (inhärente) Parallelismus in üblichen Programmen. Es gibt jeweils mehrere Möglichkeiten, diesen Parallelismus (1) überhaupt zu erkennen und (2) die Hardware so auszulegen, daß tatsächlich mehrere elementare Operationen gleichzeitig ablaufen können.

3.2. Leistungssteigerung im klassischen Einzelprozessor

3.2.1. Vermeiden von Wartezuständen

Ideale Betriebsverhältnisse beim Speicherzugriff
Jeder Speicherzugriff dauert nicht länger als ein interner Taktzyklus. Um dies näherungsweise (d. h. in einer möglichst großen Anzahl von Zugriffsfällen) zu erreichen, gibt es verschiedene Lösungen:

Caches
Zugriffe, die Treffer in einem internen Cache sind, dauern nicht länger als einen Taktzyklus. Näheres in Heft 6.

Zugriffspufferung
Der Zugriff wird entweder vorbeugend ausgeführt (Lesen) oder gleichsam nur angeschoben, wobei das Zu-Ende-Führen anderen Einrichtungen überlassen wird (Schreiben). Man kann nur puffern, was man kennt oder vergleichsweise sicher vorhersagen kann - wahlfreie Lesezugriffe auf Daten kann man nicht im voraus erraten, also auch nicht vorbeugend auslösen. Deshalb werden herkömmlicherweise nur zwei Arten von Zugriffen gepuffert:

1. Befehlslesen. Infolge der Befehlsadreßzählung ist bekannt, welche Befehle als nächste wahrscheinlich ausgeführt werden. Es lohnt sich deshalb, vorbeugend Befehlslesezugriffe auszuführen und die gelesenen Angaben zwischenzuspeichern (Warteschlangen- bzw. FIFO-Prinzip). Wird dann tatsächlich der nächste Befehl benötigt, genügt es, auf den Puffer zuzugreifen. Wird allerdings eine Verzweigung wirksam, so ist der Pufferinhalt zu verwerfen und ein neuer Befehlslesezugriff auszuführen (der dann seine Zeit dauert). Im PC-Bereich gibt es das von Anfang an - bereits der 8086 hat eine Befehlswarteschlange von 6 Bytes.

2. Schreibzugriffe. Hier sind sowohl Adresse als auch Datenbelegung bekannt. Der Prozessor kann sich also damit begnügen, beides in einem Puffer abzulegen (dauert lediglich einen Taktzyklus) und es einer nachgeordneten Steuerschaltung überlassen, den Zugriff zu Ende zu bringen (Fachbegriff: Write Posting). Im PC-Bereich wurde dies mit dem 486 eingeführt. Näheres in Heft 6.

Hinweis:
In Heft 6 gehen wir des weiteren darauf ein, was das Speichersubsystem von sich aus tun kann, um Zugriffe zu beschleunigen.

Spekulative Zugriffe
Es liegt nahe, auch Daten vorbeugend zu lesen und sich dabei auf die Annahme zu stützen, daß der nächste Zugriff höchstwahrscheinlich die nächste Adresse betreffen wird. Im Speichersubsystem wird dies tatsächlich getan (Fachbegriff: Read Prefetch; Heft 6). Im Prozessor hat sich dieses einfache Prinzip aber als wenig brauchbar erwiesen - der Anteil der wahlfreien Zugriffe ist doch so groß, daß ein solches Verfahren nur sehr geringe Trefferraten aufweist (demgegenüber werden außerhab des Prozessors vorwiegend Datenblöcke bewegt: Cache-Einträge, Sektoren, Pixel-Blöcke, Pakete der Datenkommunikation usw.). Der Ausweg: die Software muß mithelfen. Moderne Architekturen enthalten deshalb Befehle, die es der Software (in der Praxis: vor allem dem Compiler) ermöglichen, Daten vorbeugend zu lesen oder wenigstens in das Cache-Subsystem zu holen. IA-32-Beispiel: der Prefetch-Befehl (im Rahmen der Erweiterungen 3DNow (AMD) und SSE (Intel)). Dessen Wirkung: von der im Befehl angegebenen Adresse wird ein Cache-Eintrag geholt. Folgen dann wirkliche Zugriffe auf diesen Adreßbereich nach, so werden diese garantiert zu Cache Hits. Weiterentwicklungen (IA-64; Abschnitt 4.3.): (1) in den Speicherzugriffsbefehlen lassen sich Hinweise unterbringen, die die Placierung der Daten innerhalb des Cache-Subsystems betreffen; (2) es gibt spezielle Befehle, die der Compiler einfügen kann, um Daten, die demnächst benötigt werden, vorbeugend zu holen (spekulatives Laden).

Zwei Prinzipien der RISC-Philosophie:

1. Lücken stopfen (nacheilendes Laden (Delayed Load))
Der Ladebefehl stößt den Speicherzugriff nur an. Im nächsten Zyklus wird bereits der nächste Befehl ausgeführt. Parallel dazu läuft der Speicherzugriff ab. Dieses einfache Prinzip funktioniert, sofern die unmittelbar nachfolgenden Befehle nicht auf die zu ladenden Daten angewiesen sind. Sind sie darauf angewiesen, so liegt ein Konfliktfall vor. In “klassischen” RISC-Maschinen muß der Compiler dafür sorgen, daß dies gar nicht vorkommen kann. Hierzu versucht der Compiler, weitere Befehle zu finden, die sich zwischen den Ladebefehl und jenem Befehl einfügen lassen, der als erster auf die geladenen Daten zugreift (Vorordnen von Befehlen). Gelingt dies nicht, so muß er die Wartezeit mit wirkungslosen Befehlen (NOPs) auffüllen. Die alternative Auslegung: der Konflikt wird von der Hardware erkannt und durch Einfügen von Wartezuständen aufgelöst. Aber auch da schadet es nicht, nützliche Befehle zu finden, die sich vorordnen lassen. Beispiel: der Ausdruck $A \cdot (B + C)$ ist zu berechnen. In Tabelle 3.1 sind eine “naiv” programmierte Befehlsfolge und eine mit einem vorgeordneten Befehl gegenübergestellt.

herkömmliche Befehlsfolge	Befehlsfolge bei nacheilendem Laden
1. Laden B nach Register 1 2. Laden C nach Register 2 3. <Register 1> := <Register 1> + <Register 2> 4. Laden A nach Register 2 5. <Register 1> := <Register 1> · <Register 2>	1. Laden B nach Register 1 2. Laden C nach Register 2 3. Laden A nach Register 3 4. <Register 1> := <Register 1> + <Register 2> 5. <Register 1> := <Register 1> · <Register 3>

Tabelle 3.1 Zur Veranschaulichung des nacheilenden Ladens

Erklärung:
Wir beziehen uns auf eine einfache RISC-Architektur mit Zweioperandenbefehlen (z. B. P/F). In der herkömmlichen Befehlsfolge braucht der 3. Befehl den vom 2. Befehl zu ladenden Operanden. Das Laden erfordert aber wenigstens einen weiteren Maschinenzyklus. Dieser ist - je nach Architektur - entweder mit einem Leerbefehl (NOP) oder mit einem Wartezustand (der Hardware) zu überbrücken. Der Ausweg: wir ordnen den 4. Befehl entsprechend vor. Da es sich auch um einen Ladebefehl handelt, müssen wir allerdings ein weiteres Register verwenden. Nun stehen dem 4. Befehl die von den Befehlen 1 und 2 geholten Daten zur Verfügung. Während der 4. Befehl ausgeführt wird, treffen die vom 3. Befehl adressierten Daten im Register 3 ein, so daß der 5. Befehl mit diesem Registerinhalt arbeiten kann.

Hinweis:
Was geschieht, wenn der Prozessor mit einem ausgesprochen langsamen Speicher zusammenwirkt? - Dies zu berücksichtigen mutet man dem Compiler nun wirklich nicht zu. Man bezieht sich vielmehr auf ein starres Taktschema, das unter idealen Betriebsverhältnissen (sprich: dann, wenn alle Zugriffe Treffer in die internen Caches sind) erfüllt ist (Stichwort: Befehlspipelining; Abschnitt 3.2.3.). Verzögerungen, die vom Speichersubsystem her zusätzlich eingefügt werden (Cache Miss, Warten am Prozessorbus usw.), sind dann Sache der Hardware (Wartezustände).

2. Registerzugriffe statt Speicherzugriffe
Große Registersätze ermöglichen es, Daten, mit denen gerade gearbeitet wird, in Registern zu halten, so daß sich die Anzahl der überhaupt erforderlichen Speicherzugriffe deutlich verringert. (Stichwort: Speicherhierarchie; vgl. Abschnitt 3.1.4. in Heft 2.)

3.2.2. Pipelining

Das Wort (sprich: Peippleining) soll eine bildhafte Vorstellung davon vermitteln, was es bezeichnet: einen fortlaufenden Informationsfluß durch die Hardware, vergleichbar einem Flüssigkeitsstrom durch eine Rohrleitung. Eine solche Leitung ist ganz von Flüssigkeit ausgefüllt, die sich in ständiger gerichteter Bewegung befindet. Was am Rohrende herausfließt, wird am Anfang sofort nachgefüllt.

Nun arbeiten digitale Schaltungen in einzelnen (diskreten) Zeitschritten (Taktzyklen), so daß das Bild vom durchströmten Rohr nicht in voller Strenge gelten kann. Gemeint ist vielmehr folgendes grundsätzliche Schema (Abbildung 3.1): Register, kombinatorische Schaltungen

und wiederum Register sind in mehreren Stufen hintereinandergeschaltet. Zunächst werden die Register der ersten Stufe geladen. Im folgenden Taktzyklus können die Verknüpfungsergebnisse in die zweite Registerstufe übernommen werden. Dann ist die erste Registerstufe aber wieder frei und kann neue Angaben aufnehmen. Eine Anordnung aus n Registerstufen liefert so nach n Taktzyklen das erste End-Ergebnis. Des weiteren befinden sich in jedem Taktzyklus n-1 Informationswandlungen in Arbeit. Dieses Prinzip kann in vielfältiger Weise genutzt werden, um durch Überlappung von Abläufen die Verarbeitungsleistung zu steigern.

Abbildung 3.1 Prinzip des Pipelining

3.2.3. Befehlspipelining

Die verschiedenen Phasen des Befehlsablaufs kennen wir aus Keft 1, Kapitel 2. Sind für jede Befehlsphase gesonderte Schaltmittel vorgesehen, so kann man, wenn ein Befehl die jeweilige Schaltung durchlaufen hat, bereits den nächsten Befehl "anarbeiten". Dieses vorbeugende Holen und Anarbeiten von Befehlen wird auch als Befehlsvorausschau (Instruction Prefetch, Instruction Look Ahead) bezeichnet. Die entsprechende Funktionseinheit heißt Befehlsvorbereitungseinheit (Instruction Prefetch bzw. Lookahead Unit).

Abbildung 3.2 veranschaulicht das Prinzip am Beispiel des 486*). Dessen Befehlspipeline hat 5 Stufen, wobei jede Stufe zumeist nur einen Taktzyklus benötigt. Nach dem Holen des Befehls wird er in zwei Stufen decodiert. Bereits in der ersten Stufe werden, falls erforderlich, Speicherzugriffe (zum Operandenlesen) ausgelöst. Sind Resultate in Register zu schreiben, so wird währenddessen bereits der nächste Befehl ausgeführt. So lassen sich oftmals Lade- und Verknüpfungsbefehle lückenlos nacheinander abarbeiten.

*): die 486-Prozessoren sind die am weitesten entwickelten “echten” Einzelprozessoren der IA-32-Architektur (zu einer Zeit befindet sich nur ein Befehl in der Ausführungsphase). Sie eignen sich deshalb besonders gut als Lehrbeispiele.

Abbildung 3.2 Ablauf des Befehlspipelining (486)

Verzweigungen und Befehlspipelining
Daß Befehle überhaupt vorbeugend geholt werden können, ist an sich nur ein nützlicher "Nebeneffekt" der fortlaufenden Befehlsadressierung. Demgegenüber unterbricht jede *Verzweigung* den fortlaufenden Befehlsfluß. Bei einer unbedingten Verzweigung muß zumindest der Verzweigungsbefehl herangeschafft worden sein, um die Adresse des Folgebefehls zur Verfügung zu haben. Bei einer bedingten Verzweigung wird es noch schwieriger: woher soll eine "vorbeugend" wirkende Hardware wissen, ob überhaupt zu verzweigen ist, bevor die Bedingung ausgewertet wurde?

Verzweigungen haben einen nicht unbeträchtlichen Anteil an den auszuführenden Befehlen (10...20% und mehr). Deshalb hat man sich verschiedene Lösungen einfallen lassen, um Lücken im Befehlsstrom zu vermeiden:

1. Sprungzielvorhersage (Branch Direction Prediction)
Sobald die Befehlsvorbereitungs-Hardware eine Verzweigung erkennt, wird das vorbeugende Befehlslesen auf die jeweils wahrscheinlichere Richtung umgesteuert. Die Wahrscheinlichkeit ergibt sich jeweils aus Erfahrungswerten und ist in der Hardware entsprechend "verdrahtet". Alternative: im Befehl sind Hinweise zur wahrscheinlichen Verzweigungsrichtung codiert (Beispiel: IA-64; Abschnitt 4.3.). Diese Hinweise müssen vom Compiler entsprechend gesetzt werden.

2. Sprungzielpuffer (Branch Target Cache)
Dies ist ein besonderer Schnellspeicher, der jene Befehle aufnimmt, die zuletzt als Ziele von Verzweigungen "angesprungen" wurden. Beispielsweise sind die letzten 256 Sprungziele in diesem Cache enthalten. Es ist sehr wahrscheinlich, daß immer wieder zu diesen Befehlen verzweigt wird. Die Verzweigung kostet dann keinen zusätzlichen Zyklus, und die Vorbereitungs-Hardware hat genügend Zeit, die jeweils folgenden Befehle heranzuschaffen.

Auffüllen der Lücke mit anderen Befehlen (Nacheilende Verzweigung; Delayed Branch)
Dieses Prinzip ist für RISC-Maschinen charakteristisch: Es gibt eine "sture" Befehlspipeline, die wegen einer Verzweigung nicht angehalten wird. Demzufolge wird die Verzweigung nicht sofort (im zugehörigen "aktuellen" Zyklus der Befehlsausführung), sondern beispielsweise einen Zyklus später ausgeführt. Im aktuellen Zyklus kommt vielmehr der nächste Befehl zur Wirkung (und zwar immer, unabhängig von der dann tatsächlich gewählten Verzweigungsrichtung). Es ist Angelegenheit des Compilers, einen passenden Befehl entsprechend einzufügen. Das kann beispielsweise ein Registertransport sein, der ohnehin erforderlich ist - gibt es nichts Passendes, ist ein Befehl einzufügen, der nicht schadet (NOP).

Hinweise:

1. Die nacheilende Verzweigung ist die Entsprechung zum nacheilenden Laden.
2. IA-32 ist eine CISC-Architektur. Dementsprechend sind Sprungzielvorhersage und Sprungzielpuffer von besonderer Bedeutung. Mehr dazu in Abschnitt 4.1.

Konflikte in der Pipeline
Konflikte entsehen dann, wenn Datenabhängigkeiten auftreten. Beispiel (wir beziehen uns auf Abbildung 3.1): ein Befehl verändert in der Ausführungsphase (Stufe 3) einen Speicherinhalt, den der nachfolgende Befehl lesen möchte. Dieser Befehl befindet sich aber bereits in Stufe 2 der Pipeline - worin auch dessen Speicherzugriff zwecks Operandenlesen gestartet wird. Inhaltsänderung und Lesezugriff werden also gleichzeitig ausgelöst. Damit der Prozessor aber korrekt (gemäß der IA-32-Architekturspezifikation) arbeitet, muß der zweite Befehl den vom ersten veränderten Speicherinhalt vorfinden. Um solche Konflikte zu behandeln, gibt es 2 Philosophien:

1. CISC-Philosophie: die Konflikte werden von der Hardware erkannt und behandelt,
2. RISC-Philosophie: die Konflikte sind vom Compiler zu erkennen und durch Erzeugen geeigneter Befehlsfolgen aufzulösen. Zur Laufzeit kommen somit gar keine Konflikte vor.

Zur CISC-Philosophie
Die Hardware bewirkt, daß aus Sicht der Software - trotz der Ablaufüberlappung - die streng sequentielle Abarbeitungsreihenfolge des Programmiermodells eingehalten wird. Kommt man in irgendeiner Stufe nicht mit einem einzigen Taktzyklus aus, werden Wartezustände eingefügt. Konflikte werden automatisch erkannt und vermieden. Beispiel: ein Befehl schreibt einen Wert in ein Register, das der Folgebefehl für seine Adreßrechnung benötigt. Der Speicherzugriff (der die Adreßrechnung einschließt) wird aber schon in der zweiten Pipeline-Stufe gestartet. Folglich muß die Befehlspipeline solange angehalten werden, bis der Registerinhalt verfügbar ist.

Zur RISC-Philosophie
Das Erkennen der Konflikt- und Sonderfälle erfordert Schaltungsaufwand. In einigen RISC-Architekturen hat man deshalb auf solche Schaltmittel verzichtet. Manche Konflikte, auf die beispielsweise ein IA-32-Hochleistungsprozessor reagieren muß, gibt es - infolge der Trennung zwischen Speicherzugriffs- und Verarbeitungsbefehlen - in RISC-Maschinen gar nicht. Die verbleibenden Konflikte zu vermeiden ist dann Aufgabe des Compilers. So vergeht beispielsweise nach Ausführung eines Ladebefehls wenigstens ein weiterer Zyklus, ehe die adressierten Daten im Prozessor bereitstehen. Wenn der normalerweise folgende Befehl diese Daten benötigt, darf er eben nicht unmittelbar folgen. Vielmehr muß der Compiler einen anderen Befehl einfügen, der die Daten nicht braucht, oder einen sogenannten Leerbefehl (No Operation, NOP), der nur die Lücke füllt, aber keine Verarbeitungswirkung ausübt (nacheilendes Laden (Delayed Load)).

Dem Anwender kann es gleichgültig sein, ob die Verarbeitung durch automatisch eingefügte Wartezustände oder durch Leerbefehle aufgehalten wird. Für die softwareseitige Konfliktvermeidung spricht die schaltungstechnische Einfachheit, die - wegen der geringeren Schaltungstiefe - durchaus etwas mehr Verarbeitungsleistung ermöglicht. Zudem kann ein Compiler Gelegenheiten erkennen, Lücken sinnvoll (mit nutzbringenden Befehlen) zu stopfen; eine Schaltung kann hingegen nur Wartezustände einfügen.

Der wesentliche Nachteil: Wenn die Wirkprinzipien der Befehlspipeline Bestandteil der Architektur sind, hat man viel weniger Freiheiten bei neuen Implementierungen. (Ein neuer Prozessor mit an sich derselben Befehlsliste usw., aber mit anderer Befehlspipeline wäre inkompatibel. Im klassischen Verständnis sollte aber eine Architektur Programmiermodell und Implementierung gegeneinander abgrenzen bzw. von Einzelheiten der Implementierung abstrahieren.)

In manchen RISC-Maschinen hat man diesen (akademischen) Ansatz nicht verwirklicht, eben wegen dieses anwendungspraktischen Nachteils - und auch die ganz modernen Architekturen (z. B. IA-64) sind entsprechend ausgelegt.

Der sinnfällige Kompromiß
Tatsächlich auftretende Konflikte in der Befehls-Pipeline sind Sache der Hardware. Das Ziel, in jedem Maschinenzyklus einen Befehl auszuführen sowie die Tatsache, daß durch Befehlspipelining versucht wird, diesem nahezukommen, gehört zum Programmiermodell. Daraus ergeben sich grundsätzliche Optimierungsziele für Compiler. Beispielsweise ist es

immer sinnvoll, dafür zu sorgen, daß nach einem Ladebefehl der geladene Registerinhalt nicht sofort im nächsten Befehl verarbeitet wird. Kann der Compiler einen anderen nützlichen Befehl dazwischenschieben, so wird das bei jeder Implementierung einer Befehlspipeline von Vorteil sein, zumindest kann es nicht schaden. Es ist aber nicht nötig, Leerbefehle einzufügen. Vor allem: der Compiler muß nicht darauf achten, wie die Befehls-Pipeline des jeweiligen Prozessors tatsächlich aufgebaut ist.

Die Pipeline ohne Konflikte stets gefüllt halten: Hardware-Multitasking
Eine neuere Bezeichnung (Fa. Sun): Vertical Multithreading. Zum Prinzip vgl. Abschnitt 2.4.4. in Heft 1 (bes. Abbildung 2.9). Es liegt nahe, das Umschalten zwischen den einzelnen Tasks in das Pipeline-Schema einzuordnen (Abbildung 3.3).

Abbildung 3.3 Hardware-Multitasking in der Befehlspipeline

Erklärung:
Wir beziehen uns auf die 5-stufige Pipeline gemäß Abbildung 3.2. Anstatt die Befehle einer einzelnen Task nacheinander durch die Pipeline zu schleusen, holen wir zunächst den 1. Befehl der 1. Task und reichen ihn an die nächste Stufe weiter, holen dann den 1. Befehl der 2. Task, reichen ihn weiter usw. Offensichtlich kann in einer n-stufigen Pipeline je ein Befehl von nTasks gleichzeitig in Arbeit sein. Mit anderen Worten: anstelle einer einzigen Maschine mit (bestenfalls) 1 MIPS/MHz (1 ausgeführter Befehl je Taktzyklus) haben wir n virtuelle Maschinen mit jeweils 1/n MIPS/MHz. Die Gesamtleistung bleibt natürlich - milchmädchenmäßig gerechnet - die gleiche. Der Vorteil: es gibt keine Pipeline-Konflikte im einzelnen Befehlsstrom. Bezüglich der einzelnen Task verhält sich die Maschine so wie ein "langweiliger" Prozessor ohne Pipelining, der alle Phasen des Befehlsablaufs nacheinander ausführt (aus Abbildung 3.3 ist ersichtlich, daß der 2. Befehl der 1. Task erst dann geholt wird, nachdem die Resultate des 1. Befehls dieser Task abgespeichert worden sind).

Bereits vor Jahrzehnten hat man Supercomputer gebaut, die so ausgelegt waren. Der Grundgedanke: wenn es um höchste Leistung geht, kommt man irgendwann nicht umhin, sich um die Parallelisierung Gedanken zu machen (also um die Aufteilung der Verarbeitungsaufgabe auf mehrere unabhängige, parallel ausführbare Programmabläufe (Threads)). Also fangen wir doch gleich damit an - und bauen eine Maschine, die so viele MIPS wie möglich bringt, allerdings gleichmäßig aufgeteilt auf n Befehlsströme (Tasks, Threads). Wer dieses potentielle Leistungsvermögen nutzen will, der muß eben parallelisieren...

3.2.4. Die klassische Verarbeitungspipeline: Vektorverarbeitung

Es liegt nahe, das Pipeline-Prinzip auch auf die Verarbeitungsschaltungen (also auf das eigentliche Operationswerk) zu übertragen: in die Verknüpfungs- bzw. Rechenschaltungen werden Zwischenregister (Pipeline-Register) eingefügt, und die ganze Anordnung wird mit einem sehr schnellen Takt betrieben. Das Problem: es können sich zwar mehrere Operanden gleichzeitig in der Pipeline befinden, aber alle Operanden können nur jeweils der gleichen Verknüpfung bzw. Rechenoperation unterworfen werden. Eine solche Pipeline ist somit nur zum Verarbeiten von Vektoren brauchbar (Vektorrechenwerk, Abbildung 3.4).

Abbildung 3.4 Vektorrechenwerk (hypothetisches Beispiel)

Erklärung:
Die dargestellte Pipeline-Struktur unterstützt u. a. die (beim numerischen Rechnen sehr wichtige) Vektoroperation Y(i) := A · X(i) + Y(i) sowie das Berechnen von Skalarprodukten. Für Addition und Multiplikation ist je ein gesondertes Rechenwerk vorgesehen. Um die einzelnen Vektorelemente (vgl. Abschnitt 1.5.) nacheinander heranzuschaffen bzw. abzutransportieren, sind zusätzliche Schaltungen erforderlich.

Hinweis:
Die Verkettung von Multiplikation und Addition (Skalarprodukt, Multiply-Add) hat in vielen Algorithmen des numerischen Rechnens und der digitalen Signalverarbeitung eine leistungsentscheidende Bedeutung. Deshalb weisen die typischen Signalprozessoren - und auch die ganz modernen universellen Hochleistungsprozessoren - solche Verarbeitungswerke auf (vgl. in Heft 4 Abbildung 9.21, (Positionen 4, 5)).

Der technische Hintergrund
Additions- und Multiplikationsschaltungen kann man auch als reine kombinatorischen Netzwerke ausführen und in ein Schema gemäß Abbildung 1.2 einordnen. Der Nachteil: wir hätten eine sehr große Schaltungstiefe (im besonderen bei der Multiplikation). Der Vorteil der Pipeline-Struktur (Abbildung 3.4): durch das Einschieben der Pipeline-Register verringert sich die Schaltungstiefe der Kombinatorik zwischen den Registern, so daß man die Taktfrequenz deutlich erhöhen kann. Beispiel: wir haben gesehen (Seiten 7, 8), daß man mit herkömmlichen Schaltkreisen Taktfrequenzen von etwa 10 bis ca. 50 MHz erreichen kann. Pipeline-Maschinen auf Grundlage herkömmlicher Schaltkreise hat man hingegen bereits in den 70er Jahren mit Taktfrequenzen von 100...200 MHz ausgeführt (die legendären Supercomputer der Fa. Cray).

Wieviele Pipeline-Stufen?
Das ist im wesentlichen Optimierungs- und Erfahrungssache. Es hat sich gezeigt, daß es sich nicht lohnt, über 6 bis 8 Stufen hinauszugehen. Auch ist es kaum sinnvoll, mehr als zwei Verarbeitungswerke hintereinanderzuschalten. Je mehr Pipeline-Stufen zu durchlaufen sind, um so länger dauert es, bis die Pipeline tatsächlich "gefüllt" ist, also mit voller Leistung arbeitet. Diese Anlaufzeit (Vector Start Up Overhead) wird gemessen vom Bereitstellen der ersten Operanden-Elemente an den Eingängen bis zum Erscheinen des ersten Resultat--Elements am Ausgang. Sie hat natürlich um so mehr Einfluß auf die Leistung, je kürzer die Vektoren sind. Deshalb stehen dem Nutzer oft nur 5...15% der Maximalleistung (Peak Performance) zur Verfügung.

3.3. Mehrfachverarbeitung (SIMD)

Der einzelne Transport- oder Verarbeitungsbefehl betrifft fest formatierte Aneinanderreihungen elementarer Datenstrukturen (mit anderen Worten: kurze Vektoren - vgl. Seite 16), z. B. 8 Bytes, 4 16-Bit-Worte, 2 32-Bit-Gleitkommazahlen usw. Alle elementaren Datenstrukturen in den Vektoren werden gemeinsam, aber unabhängig voneinander der gleichen Operation unterzogen (Abbildung 3.5).

Abbildung 3.5 Mehrfachverknüpfung elementarer Datenstrukturen

3.4. Der innewohnende (inhärente) Parallelismus

Dieser Begriff (engl. Inherent Parallelism*)) bezeichnet die Tatsache, daß man, obwohl vom Programmiermodell her eine strikt serielle Befehlsreihenfolge vorgegeben ist, durchaus einiges gleichzeitig (parallel zueinander) erledigen kann, vorausgesetzt, die Hardware bietet die Gelegenheit dazu. Beispiel: Es sei zu berechnen (1) A := B + C, (2) X := Y + Z. Dafür programmiert man normalerweise zwei Additionsbefehle, denen erforderlichenfalls Ladebefehle vorangehen. Beide Operationen und auch die Ladevorgänge können offensichtlich jeweils gleichzeitig ablaufen; die Ergebnisse werden davon in keiner Weise beeinflußt. Natürlich muß die Hardware so ausgelegt sein, daß sie dies tatsächlich zuläßt (Stichwort: mehr als 1 MIPS/MHz). Die grundsätzlichen Auslegungsformen, die auch in Kombination nutzbar sind, bezeichnet man mit Superskalarität und Superpipelining.

*): auch: Instruction Level Parallelism (ILP).

Superskalarität
Eine Superskalar-Maschine kann mehrere Operationen mit skalaren Operanden gleichzeitig ausführen. Sie hat dazu eine entsprechende Anzahl parallel angeordneter Verarbeitungswerke.

Superpipelining
Eine Superpipelining-Maschine kann mehrere Operationen mit skalaren Operanden gleichzeitig, aber in kleinen Schritten zueinander zeitversetzt, in einer Pipeline-Struktur in Arbeit haben. Prinzip: Die einzelne Operation erfordert normalerweise einen Taktzyklus von n Nanosekunden. Das Operationswerk wird in p Pipeline-Stufen unterteilt. Dann sind p Operationen gleichzeitig, aber mit einem Versatz von n/p Nanosekunden, ausführbar.

Beachten Sie bitte den Unterschied: In der klassischen Vektormaschine (Abbildung 3.4) arbeitet die Pipeline an gleichartigen Verknüpfungen gleichartiger Operanden; in der SuperpipelineMaschine laufen hingegen verschiedenartige Operanden durch die Pipeline, die dort unterschiedlichen Operationen unterzogen werden. Die Verknüpfungen in einer Vektor--Pipeline werden zumeist von einem einzigen Befehl ausgelöst, der ganze Vektoren betrifft. Die einzelnen Operationen, die in einer Superpipeline-Maschine in Arbeit sind, werden von verschiedenen Befehlen gesteuert.

Hinweis:
Wir haben hier die sozusagen akademische Definition erläutert (“Superpipelining ist, wenn sich mehrere Befehle gleichzeitig in Ausführungs-Stufen der Pipeline befinden können”). Da “Super” aber immer gut klingt, verwendet man den Begriff gelegentlich auch für Maschinen, die einfach eine größere Zahl von Pipeline-Stufen haben (z. B. 10..15 anstelle der üblichen 4...6).

3.4.1. Den innewohnenden Parallelismus erkennen

Man kann versuchen, innewohnenden Parallelismus aufzufinden, indem man sich eine deutliche Vorstellung vom Verarbeitungsablauf bildet. Man "sieht einfach", welche Informationswandlungen parallel zueinander ausführbar sind und welche nicht (intuitive Erkennung). In einfachen Beispielen geht so etwas recht gut. Bei umfangreicheren Programmen wird es jedoch schnell undurchführbar. Hier bietet es sich an, den Computer auch dafür auszunutzen, also das "Durchforsten" eines gegebenen Programms seinerseits zu programmieren. Solche Programme sind üblicherweise Bestandteil eines Compilers, so daß während der Compilierung *(zur Compilierzeit*) erkannt wird, welche Aktivitäten gleichzeitig ausführbar sind und welche nicht. Alternativ dazu kann man die Hardware so auslegen, daß sie versucht, soviele Aktivitäten wie möglich gleichzeitig auszuführen. Dazu gehört weiterhin, daß die Hardware Konflikte erkennt und darauf entsprechend reagiert. Man kümmert sich also erst dann um den innewohnenden Parallelismus, wenn das Maschinenprogramm tatsächlich läuft (Erkennung *zur Laufzeit*).

Erkennung zur Compilierzeit
Zwei Vorteile sprechen für diesen Ansatz:

1. man braucht keine besonderen Schaltmittel, um den Parallelismus zu erkennen und um die Parallelarbeit zu steuern. Solche Schaltungen sind nicht einfach, belegen Siliziumfläche und können den Maschinenzyklus verlängern (zusätzliche Schaltungstiefe). Es ist demgegenüber verlockend, die verfügbare Siliziumfläche weitgehend für Verarbeitungsfunktionen und Caches auszunutzen.
2. eine Hardware kann nur die jeweils geholten Befehle übersehen, ein Compiler das ganze Programm. Deshalb ist zu erwarten, daß der Compiler mehr Gelegenheiten hat, innewohnenden Parallelismus zu erkennen und davon im Verarbeitungsablauf Gebrauch zu machen.

In manchen Architekturen ist dieses Prinzip bis zum Extrem getrieben worden: der Compiler muß sich sogar um die Ablaufüberlappung der Befehlsausführung (Befehlspipelining) kümmern. Die Befehls-Pipeline an sich ist zwar vorgesehen, es gibt aber keine hardwareseitige Verriegelung (Interlocking), keine Steuerung von Wartezuständen usw. Der Compiler kennt das Taktschema, muß die Takte der einzelnen Befehlsphasen auszählen und Lücken im Befehlsstrom mit umgeordneten Befehlen oder mit Leerbefehlen füllen. Eine Architektur leitet von diesem Prinzip ihren Namen ab (Mips = Microprocessor Without Interlocked Pipeline Stages - in den neueren Prozessormodellen hat man dieses Prinzip allerdings aufgegeben).

Erkennung zur Laufzeit
Es werden mehrere Befehle gleichzeitig geholt. Die Hardware erkennt, welche dieser Befehle parallel zueinander ausführbar sind und welche Verarbeitungswerke dafür zur Verfügung stehen. Dementsprechend werden die einzelnen Verarbeitungswerke mit Operanden versorgt. Entsprechende Schaltungsanordnungen werden üblicherweise als *Scoreboard* bezeichnet. Das Prinzip wurde erstmals Mitte der 60er Jahre in der Anlage CDC 6600 implementiert.

Effektivität
Das Erkennen des innewohnenden Parallelismus darf die Hardware nicht verlangsamen. Grundsätzlich kann eine Schaltung nur soviele Befehle übersehen, wie sie in einem Zyklus parallel holen kann (darüber hinaus sind die bereits in der Ausführung befindlichen oder im Prozessor wartenden Befehle einzubeziehen). Eine praktikable Obergrenze liegt bei etwa 8 bis 16 Befehlen. Des weiteren ist es eine Erfahrungstatsache, daß in vielen Programmen die "linearen" Abschnitte (mit aufeinanderfolgenden Operations- und Transportbefehlen), die zwischen Verzweigungsbefehlen liegen, recht kurz sind (es ist selten, daß solche Folgen - sogenannte Basisblöcke - mehr als 5 bis 7 Befehle umfassen). Damit ist der Grad des nutzbaren innewohnenden Parallelismus grundsätzlich beschränkt; eine übermäßige Anzahl von Verarbeitungswerken kann nichts mehr zur Leistungssteigerung beitragen.

Diskussion
Überläßt man alle einschlägigen Aufgaben dem Compiler, so wird zwar die Hardware einfacher, man muß aber mit zwei Nachteilen leben:

1. extrem optimierende Compiler sind langsam und vergleichsweise unzuverlässig (schließlich kann niemand die 100%ige Funktionsfähigkeit einer solch komplexen Software garantieren). Solche Maschinen kann man kaum noch "von Hand" (in Assembler) programmieren; man muß sich absolut auf den Compiler verlassen.
2. es ist praktisch kaum noch möglich, kompatible Rechnerfamilien zu schaffen. Da der Compiler alle Einzelheiten des Ablaufs in der Befehlspipeline berücksichtigt, würde ein neues Prozessormodell mit neuartig ausgelegter Befehlspipeline (1) neue Compiler und (2) die Neucompilierung der gesamten Anwendungssoftware erfordern.

Erkennungsprinzip und Architektur
Wird der innewohnende Parallelismus zur Laufzeit erkannt, so ist es - zumindest dem Grundsatz nach - nicht notwendig, die Architektur dafür auszulegen. Das heißt: man kann entsprechende Hochleistungsprozessoren auch auf Grundlage “uralter” Architekturen bauen (es ist “nur” erforderlich, für alles, was gleichzeitig ablaufen soll, entsprechende Schaltmittel vorzusehen).

Soll hingegen der Parallelismus zur Compilierzeit erkannt werden, so braucht man entsprechende architekturseitige Vorkehrungen, beispielsweise besondere Befehlsformate, in denen codiert ist, wie die Parallelverarbeitung gesteuert wird (Beispiel 1: IA-64 (Abschnitt 4.3.); Beispiel 2: die Superskalar-Signalprozessoren TMS320C6x (vgl. bes. Abbildung 9.24 in Heft 4)).

Zum Stand der Technik:

- **#** man gibt sich viel Mühe, IA-32-Prozessoren zu bauen, die den innwohnenden Parallelismus zur Laufzeit erkennen (Abschnitte 4.1., 4.2.),
- **#** moderne Architekturen legt man hingegen so aus, daß der Compiler den innewohnenden Parallelismus erkennen und in den Befehlen entsprechend kennzeichnen kann. Mit den Einzelheiten des Befehlspipelining hingegen muß sich der Compiler nicht beschäftigen.

3.4.2. Superskalarmaschinen

Es werden mehrere Operationswerke parallel angeordnet, so daß sich entsprechend viele Verknüpfungen gleichzeitig ausführen lassen. Hierfür gibt es mehrere Gestaltungsmöglichkeiten:

Art der Operationswerke
Wir können mehrere gleichartige, universell nutzbare Werke vorsehen (jedes Werk kann jede Operation ausführen) oder mehrere spezialisierte (z. B. Addierwerke, Multiplizierwerke, Werke für logische Verknüpfungen usw.), vielleicht auch einen "Mix" aus universellen und spezialisierten Werken. Beispielsweise liegt dem Pentium-Prozessor (P5) folgende Kompromißlösung zugrunde: Es gibt zwei Operationswerke. Davon ist das erste mikroprogrammgesteuert und kann alle Befehle ausführen. Das zweite ist direkt gesteuert und kann nur entsprechend einfache Befehle ausführen (Blockschaltbild: Abbildung 9.2 in Heft 4). Weitere Beispiele in Abschnitt 4.2. sowie in Heft 4.

Wieviele Operationswerke?
Grundsätzlich ist die Sinnfälligkeit solcher Mehrfachanordnungen nur durch Versuche, Messungen usw. zu ermitteln. Einschlägige Erfahrungen im Überblick:

- **#** mehr als 2 *universelle* Werke können von üblichen Programmen kaum sinnvoll gleichzeitig beschäftigt werden,
- **#** bei vorwiegend numerischen Aufgabenstellungen sind Multiplikation und Addition von Gleitkommazahlen sowie zwei und mehr Operationen mit ganzen Binärzahlen (Integer-Operationen) durchaus sinnvoll parallelisierbar. Weshalb? Die eigentlichen numerischen Rechnungen werden mit Gleitkommazahlen durchgeführt. Die Integer-Operationen dienen hingegen dazu, die Adressen der Gleitkomma-Operanden zu berechnen[*)].
- **#** geht es um “richtige” numerische Anwendungen (Lösen von Gleichungssystemen usw.), so kann man auch eine größere Zahl von Operationswerken sinnvoll beschäftigen (man hat bereits in den 80er Jahren VLIW-Maschinen gebaut, in denen bis zu 28 Operationen gleichzeitig ausgeführt werden können),
- **#** moderne, für PCs, Workstations, Server usw. typische Anwendungen können durchaus bis zu 4 Operationswerke ausnutzen (und zwar in einem gewissen “Mix” - Rechnen mit Binärzahlen, Adreßrechnung, Gleitkommaverarbeitung usw.) - aber nicht wesentlich mehr (vgl. auch die Ausstattung der modernen Hochleistungsprozessoren (Kapitel 4; Abbildungen 9.22...9.25 in Heft 4).

*): Beispiel: Numerische Daten sind häufig in Form von Matrizen (zweidimensionalen Feldern) zusammengefaßt. Um einen Operanden in Zeile n und Spalte m einer Matrix zu adressieren, ist (bei zeilenweiser Speicherung) folgende Rechnung auszuführen (OL = Operandenlänge (in Bytes), SZ = Spaltenzahl der Matrix):

$$\text{Adresse} = ((n - 1) \cdot OL \cdot SZ) + ((m - 1) \cdot OL)$$

Mit jeder derartigen Adreßrechnung könnte also wenigstens ein Multiplikations- und ein Additionswerk für Binärzahlen parallel zur Gleitkommaverarbeitung beschäftigt werden.

Steuerung der Operationswerke
Es gibt zwei Prinzipien, mehrere Verarbeitungswerke zu steuern: (1) Starten mehrerer Befehle, (2) extrem lange Befehle.

Starten mehrerer Befehle (Multiple Instruction Launch)
Der einzelne Befehl betrifft ein einziges Operationswerk. Die Hardware holt mehrere Befehle gleichzeitig und startet die Befehlsausführung in allen Operationswerken, in denen dies möglich ist.

Erkennung zur Laufzeit
Die Hardware ist allein verantwortlich, parallel ausführbare Befehle zu erkennen. Sie startet jene Operationswerke, die (1) die anhängigen Befehle ausführen können und die (2) gerade frei sind.

Das klassische Beispiel ist die Anlage CDC 6600, die 10 Verarbeitungswerke unterschiedlicher Zweckbestimmung enthält, aktuelle Beispiele sind die P6-Prozessoren von Intel sowie verschiedene kompatible Typen anderer Hersteller (Abschnitt 4.2.).

Erkennung zur Compilierzeit
Der Compiler muß in den Befehlen, die parallel zueinander ausgeführt werden können, entsprechende Steuerbits setzen bzw. diese Befehle zu größeren Einheiten zusammenfassen. Moderne Prozessoren sind zumeist als VLIW-Architekturen ausgelegt - mehrere Befehle werden als fest formatiertes “Bündel” zusammen geholt und in parallelen Operationswerken ausgeführt.

Extrem lange Befehle (Very Long Instruction Word VLIW)
In diesen Architekturen ist das Befehlsformat so gestaltet, daß ein Befehl alle parallel ausführbaren Funktionen gleichzeitig steuern kann. Solche Befehle können deshalb extrem lang werden (128 Bits und mehr). Die mderne Auslegung: mehrere kürzere Befehle (die einzeln wie RISC-Befehle aussehen), werden zu Bündeln oder Paketen zusammengefaßt. Beispiele: in Abschnitt 4.2.4. und 4.3. sowie in Heft 4 (vgl. die Abbildungen 9.22...9.26).

VLIW und SIMD
Beachten Sie bitte die Unterschiede (Tabelle 3.2).

VLIW	SIMD
P lange Befehle mit Adreß- und Steuerangaben für mehrere gleichzeitig ausführbare Operationen, P die Befehle adressieren mehrere (zumeist) elementare Datenstrukturen, P es können verschiedenartige Operationen über (zumeist) elementare Datenstrukturen gleichzeitig ausgeführt werden	P Befehle können kurz sein, da in jedem SIMD-Befehl nur eine Operation angegeben ist, P die Befehle adressieren nur wenige Datenstrukturen, in denen aber mehrere gleichartige Operanden verpackt sind (Vektorprinzip), P es wird jeweils nur eine einzige Operation ausgeführt, aber mit allen Operanden gleichzeitig

Tabelle 3.2 VLIW und SIMD - eine Gegenüberstellung

3.4.3. Superpipelining

Eine extrem schnelle Verarbeitungs-Pipeline, in der mehrere Befehle gleichzeitig in Arbeit sind, ist eine Alternative zur Anordnung mehrerer Operationswerke. Zumindest theoretisch bietet das Superpipelining das bessere Preis-Leistungs-Verhältnis: es kostet weniger, beispielsweise in ein Addierwerk die erforderlichen Pipeline-Register einzufügen, als mehrere Addierwerke vorzusehen. Dem entgegen steht die größere Kompliziertheit der Steuerung (der Fluß verschiedenartiger Befehle durch eine universelle Pipeline erfordert es, die einzelnen Pipeline-Stufen von Takt zu Takt umzuschalten, was praktisch nur mit einer Art Datenflußsteuerung zu erreichen ist). Zudem wirkt sich die Anlaufzeit der Pipeline nachteilig aus. Dem Stand der Technik entsprechen "Mischformen": eine gemäßigt ausgelegte Superskalar-Hardware (2, höchstens 4 Verarbeitungswerke), wobei die einzelnen Werke intern als Superpipeline aufgebaut sind.

3.4.4. Prinzipien der Datenflußsteuerung

Solche Prinzipien wurden bereits in den 60er Jahren verwirklicht (beispielsweise im Hochleistungsrechner IBM 360/91). Der Grundgedanke besteht darin, die Befehlspipeline von den ebenfalls in Pipeline-Struktur ausgebildeten Operationswerken zu entkoppeln und beide mit jeweils maximaler Taktfrequenz arbeiten zu lassen. Prinzip: im Rahmen der Befehlspipeline werden die Operanden aus dem Speicher geholt. Jeder Operand erhält dann eine Kennung in Form zusätzlicher Bits (Token bzw. Tag Bits) und wird so in die Operations-Pipeline geschickt. Die Kennung gibt an, in welchem Werk und mit welchem anderen Operanden er zu verknüpfen ist. Die Kennungsbits leiten die Daten gleichsam durch die Pipeline. Pipeline-Stufen, die für die betreffende Operation nicht von Bedeutung sind, werden umgangen oder ohne Wirkung durchlaufen. Gelangt eine solche Angabe zur ersten Stufe, die zur gewünschten Verknüpfung beiträgt, so wartet sie dort auf die zweite Angabe mit gleicher Kennung. Sobald diese eintrifft, wird die Verknüpfung wirksam. Datenflußsteuerung bedeutet hier, daß Daten und Steuerangaben zusammen durch die Verarbeitungshardware fließen. Das ist ein sehr wirksames Verfahren, um eine Superpipeline-Organisation zu verwirklichen: man gibt praktisch den entsprechend aufbereiteten Operationscode der auszuführenden Befehle die jeweiligen Daten als Direktwerte bei (es ist nichts mehr zu

adressieren) und kann somit recht elegant die zeitversetzt parallele Ausführung mehrerer Operationsbefehle steuern.

Eine alternative Auslegung besteht darin, die zur Parallelausführung aufbereiteten Befehle in einem assoziativen Steuerspeicher abzulegen. Dieser wird nicht von einem Befehlszähler adressiert, sondern von den Verarbeitungseinrichtungen aus über Zustandsmeldungen angesprochen. Ist ein Verarbeitungswerk imstande, eine bestimmte Operation auszuführen, und stehen die erforderlichen Daten bereit, so bewirkt die jeweilige Zustandskombination, daß ein passender Befehl aus dem Steuerspeicher abgerufen wird.

Wenn in Zusammenhang mit modernen Hochleistungsprozessoren von Datenflußsteuerung die Rede ist, meint man zumeist irgendeine Ausführungsform der hier beschriebenen Prinzipien (z. B. in verschiedenen Signalprozessoren, in P6, Athlon usw).

Operationsverkettung (Chaining)
Eine weitere elementare Nutzungsweise des Datenflußprinzips besteht darin, mehrere Operationswerke vorzusehen und sie so zusammenzuschalten, daß Resultate des einen Werkes unmittelbar als Operanden zu einem anderen Werk fließen können - also praktisch Strukturen ähnlich Abbildung 1.5 aufzubauen. Typischerweise beschränkt man sich dabei allerdings auf die anwendungspraktisch wichtigste Verkettung von Multiplikation und Addition (Multiply-Add; vgl. Abbildung 3.5). Beispiele: IA-64, Majc (Sun) und verschiedene Signalprozessoren.

3.5. Universal- und Spezialprozessoren

Bevor digitale Universalrechner kostengünstig verfügbar waren, war es geradezu selbstverständlich, für die verschiedenen Aufgaben der Informationsverarbeitung jeweils besondere Einrichtungen zu entwerfen. Dazu wurden gegebene technische Mittel vielfältigster Art (mechanische, elektrische, pneumatische usw.) trickreich eingesetzt. Wer hingegen mit dem Mikroprozessor gleichsam aufgewachsen ist, wird wohl eher an ein Programm als an eine zweckgebundene Hardware denken. Um so überraschender sind die Ergebnisse, wenn man den altbewährten Ansatz auf die moderne Elektronik-Technologie überträgt. Ist ein einzelner, genau abgegrenzter Komplex von Algorithmen und Datenstrukturen gegeben, so bereitet es oft keine grundsätzlichen Schwierigkeiten, dafür spezielle Schaltungsanordnungen zu entwerfen, die jeden Universalrechner an Leistungsfähigkeit (bezogen auf die jeweiligen Aufgaben) weit übertreffen. Lohnt es sich dann eigentlich, den *universellen* Prozessor leistungsseitig bis zum Äußersten weiterzuentwickeln? Schließlich könnte man das gewünschte Leistungsvermögen auch bereitstellen, indem man einen Universalprozessor mittleren Leistungsvermögens mit Spezialprozessoren zusammenschaltet (Abbildung 3.6).

Abbildung 3.6 Hochleistungssystem aus Universalprozessor und Spezialprozessoren (hypothetisches Beispiel)

Ist das die Zukunft? - Für Embedded Systems, wo (1) hohe Leistungen gefordert werden, (2) die Hardware-Stückkosten entscheiden, aber (3) die Nutzer nie programmieren, war diese Auslegung bisher üblich und wird auch weiterhin Bestand haben (Beispiel: KFZ-Technik).

Betrachten wir demgegenüber eher typische PC- bzw. Workstation-Anwendungen. Namentlich in den 80er Jahren hat man manche Systeme so aufgebaut, wie in Abbildung 3.6 gezeigt: ein universelles Bussystem (z. B. der VME-Bus), ein bewährter Universalprozessor (z. B. ein Motorola 68k-Typ), hinreichend RAM und Peripherie sowie - auf besonderen Steckkarten - Spezialschaltungen zur Beschleunigung leistungsbestimmender Abläufe. Beschleunigungs- Hardware gab es beispielsweise für das Entflechten von Leiterplatten, für Simulationsaufgaben, für Graphik-Darstellungen usw.

Solche Systeme erfordern vergleichsweise mäßige Aufwendungen, die auch von kleineren Firmen erbracht werden können (eine Steckkarte mit gekauften Schaltkreisen bringen Sie - als Funktionsmuster - notfalls am Küchentisch zustande, einen Prozessor der P6-Klasse auch dann nicht, wenn Sie 10 Millionen DM dafür einsetzen würden).

Die Probleme sind aber nicht zu übersehen:

- die Transporte zwischen den verschiedenen Beschleunigungsschaltungen begrenzen die Verarbeitungsleistung. Stellen Sie sich - ohne Bezug auf eine bestimmte Anwendung - nur vor, daß fortwährend große Datenmengen zwischen dem Universalprozessor, einem Gleitkommabeschleuniger, einem Simulationsbeschleuniger und einem Graphikbeschleuniger hin- und herzutransportieren sind.
- die meisten der Algorithmen, die für eine hardwareseitige Beschleunigung in Frage kommen, sind in sich nicht trivial, und sie sind häufig in umfangreiche Programmkomplexe eingebettet. Schnittstellen zu spezieller Hardware werden von den üblichen Programmiersprachen kaum unterstützt. Man muß also hier auf Maschinenebene programmieren. Bei Wechsel der Hardware sind diese zeitaufwendigen Arbeiten also stets von neuem erforderlich; mit "einfacher" Neu-Compilierung ist es nicht getan.
- stellen Sie sich vor, der Lieferant der Spezialhardware unterstützt Ihr System nicht mehr oder verschwindet schlicht und einfach vom Markt. Worauf lassen Sie dann Ihre Software (die ja auf die Beschleunigungs-Hardware angewiesen ist) laufen?
- ein solches System ist eine Spezialmaschine, die in allen andersartigen Einsatzfällen nur mittelmäßige Leistung erbringt (wenn sie überhaupt dafür zu gebrauchen ist). Wir können ein solches System also nicht immer mit anderen Anwendungen auslasten (z. B. die Gehälter berechnen, wenn gerade keine Leiterplatten zu entflechten sind).

Zudem gibt es heute eine technologische Lücke zwischen den Hochleistungsprozessoren und spezieller Hardware. Ein Hochleistungsprozessor wird gleichsam bis auf den letzten Transistor optimiert - die sehr hohen Entwicklungskosten kann man sich leisten, weil anschließend Millionenstückzahlen verkauft werden. Solche Prozessoren können mit Taktfrequenzen von 500, 700, 800 und mehr MHz betrieben werden. Spezialhardware ebenso zu optimieren lohnt sich praktisch nur dann, wenn ebenfalls Millionenstückzahlen absetzbar sind. Alles, was wirklich “speziell” ist (geringere Stückzahlen), kann man nur auf Grundlage hochentwickelter programmierbarer Schaltkreise wirtschaftlich fertigen. Solche Schaltkreise kann man aber - wegen des “Overheads” der programmierbaren Schaltungs- und Verbindungsstrukturen - nur mit deutlich geringeren Taktfrequenzen betreiben (Richt-

wert: 1/5...1/10 gegenüber einem Hochleistungsprozessor in vergleichbarer Technologie). Die Rechnung ist also ganz einfach: eine Spezialschaltung, die in einem Taktzyklus nicht mehr leistet als ein moderner Hochleistungsprozessor mit, rund gerechnet, 10...40 Befehlen*) auch erledigen kann, lohnt sich eigentlich nicht.

*): moderne Prozessoren sind Superskalarmaschinen und führen typischerweise 2...4 Befehle gleichzeitig aus. Demgemäß kann der Prozessor während eines Taktzyklus der Spezialhardware zwischen $2 \cdot 5 = 10$ und $4 \cdot 10 = 40$ Befehle ausführen (Taktfrequenzen Spezialhardware : Prozessor = 1 : 5 bis 1 : 10).

Anmerkungen zum Stand der Technik:

Systeme ähnlich Abbildung 3.6 sind kaum noch in Mode,

spezielle (= über die Befehlswirkungen herkömmlicher Prozessoren hinausgehende) Funktionen, die allgemein (= für eine Vielzahl von Anwendungen) nutzbar sind, werden in den Universalprozessor eingebaut (Beispiele: die verschiedenen SIMD-Erweiterungen, wie MMX, 3DNow, SSE, AltiVec usw.). Das entschärft das Transportproblem (die Transporte zwischen dem "universellen Kern" des Prozessors und den Erweiterungsschaltungen laufen mit der im Prozessor üblichen Zugriffsbreite und Taktfrequenz ab).

spezielle Schaltkreise: sie werden nach wie vor entwickelt und gefertigt, und zwar vor allem:

- für Aufgaben, an denen massenhafter Bedarf besteht (Beispiel: Graphikbeschleuniger und Graphikmaschinen),
- für Aufgaben, die so kompliziert sind, daß sich die leistungsbestimmenden Abläufe nicht von Universalprozessoren mit wenigen Befehlen erledigen lassen (Beispiele: Datenkompression, Verschlüsselung, Spracherkennung).

4. Wirkprinzipien moderner Hochleistungsprozessoren

4.1. Ablaufbeschleunigung

4.1.1. Sprungvorhersage (Branch Prediction)

Programmverzweigungen (bedingte Sprünge) "halten den Betrieb auf": da man zunächst nicht weiß, in welcher Richtung es weitergeht, bleibt an sich nichts anderes übrig, als die Befehlspipeline anzuhalten und nach Entscheidung über die Verzweigungsrichtung wieder aufzufüllen (mit anderen Worten: Verzweigung und Ablaufüberlappung schließen normalerweise einander aus; die Verzweigung "serialisiert" den Befehlsstrom). Dieser Effekt läßt sich mit einer gewissen Wahrscheinlichkeit*) vermeiden, wenn die Befehlsablaufsteuerung gewisse Annahmen über die wahrscheinliche Verzweigungsrichtung trifft und die Befehlspipeline entsprechend steuert.

Einige naheliegende Beispiele:

1. Verzweigung auf Überlauf (Branch on Overflow): es ist wahrscheinlich, daß nicht verzweigt wird, da eine Überlaufbedingung vergleichsweise selten auftritt.
2. Verzweigung bei “nicht Null” (Branch on Not Zero): es ist wahrscheinlich, daß verzweigt wird, da alle anderen Ergebniswerte häufiger auftreten werden als der (einzige) Wert Null.
3. bei "kurzen" bedingten Verzweigungen (Short Jumps)**) geht es sehr wahrscheinlich in Verzweigungsrichtung weiter. Solche Befehle werden oft zum Schließen von Programmschleifen verwendet, und Schleifen werden zumeist wesentlich mehr als einmal durchlaufen.
4. "lange" bedingte Verzweigungen (Long Jumps)**), namentlich solche, die Sonderbedingungen (z. B. Überlauf) abfragen, werden zumeist nicht wirksam (Weiterführung in Geradeausrichtung).

*): man spricht hier von einer “Vorhersagegenauigkeit “ oder - wie beim Cache - von einer Trefferrate. In modernen Prozessoren erreicht man Werte von über 90% (bei über 90% aller Verzweigungen wird die Verzweigungsrichtung richtig vorhergesagt).

**): die Kürze bzw. Länge betrifft die Adreßangabe im Verzweigungsbefehl.

Einfache Implementierungen arbeiten tatsächlich so: jedem Verzweigungsbefehl wird eine wahrscheinliche Verzweigungsrichtung fest zugeordnet. Damit kann die Trefferrate aber nicht allzu hoch sein, und manchmal kann man sich damit, wie die US-Amerikaner sagen, richtig in den Fuß schießen. Beispiel: Wir verwenden die Verzweigung auf Überlauf als Abbruch-Test am Ende einer Programmschleife (die Schleife wird beendet, wenn *kein* Überlauf mehr auftritt). Funktioniert unsere Hardware wie in Beispiel 1, so ist in allen Schleifendurchläufen (das können etliche tausend sein) die Vorhersage wirkungslos!

Der Ausweg: eine Art Lernmechanismus. Der Prozessor merkt sich die vorhergesagten Verzweigungen und zieht entsprechende Schlüsse daraus, falls Vorhersagen nicht eingetroffen sind. Die Angaben hierzu werden in einer Speicheranordnung gehalten, die als Branch History Table (BHT) oder ähnlich bezeichnet wird. Beispielsweise kann man sich die Adresse des Verzweigungsbefehls sowie die vorhergesagte und die tatsächliche Verzweigungsrichtung merken. Wird der Befehl wiederholt ausgeführt und ist aus der “Vorgeschichte” erkennbar, daß falsch vorhergesagt wurde, so wird die nächste Vorhersage eben genau andersherum ausfallen. (Hierdurch werden z. B. falsche Voraussagen in Schleifen-Testbefehlen nach dem ersten Schleifendurchlauf bemerkt und berichtigt.)

Vorausschau in beiden Verarbeitungsrichtungen

In manchen Hochleistungssystemen ist die Befehlsvorbereitung so leistungsfähig, daß Befehle in beiden Richtungen vorbereitet werden können. So werden - ohne raten zu müssen - Lücken in allen Verzweigungsfällen vermieden, allerdings mit beträchtlichem Aufwand. (Auch das Speichersubsystem muß hinreichend leistungsfähig sein - es muß in der Lage sein, kurzzeitig beide Befehlsströme zu liefern, ohne deswegen Wartezustände einzufügen.)

4.1.2. Sprungziel- und Rückkehrpuffer

Diese Schaltmittel (engl. Branch Target Buffer, Return Address Stack) dienen dazu, Verzweigungen und das Verlassen von Unterprogrammen (Subroutine Return) zu beschleunigen, genauer gesagt: auch bei diesen Abläufen die Befehlspipeline lückenlos gefüllt zu halten.

Sprungzielpuffer
Es gibt verschiedene Implementierungen. Manche sind eine Art Cache für die (fertig berechneten) Verzweigungsadressen, manche enthalten unmittelbar die Befehle, die mit der jeweiligen Verzweigungsadresse angesprungen werden. Hierdurch werden Zugriffe auf den L1-Cache bzw. Durchläufe durch die Befehlsdecodierung vermieden.

Rückkehrpuffer
Dies ist ein Hardware-Stack für die Rückkehr aus Unterprogrammen. Normalerweise befinden sich die Rückkehradressen im "architekturseitigen" Stack, der über den Stackpointer (SP) adressiert wird. Rückkehrbefehle müssen demzufolge auf den Speicheradreßraum zugreifen (d. h. wenigstens auf den L1-Cache - nicht selten sind aber tatsächlich Buszugriffe notwendig). Hingegen befindet sich der Rückkehrpuffer im Prozessorschaltkreis und ist praktisch ohne Zeitverlust nutzbar. (Auch hier gibt es die gleichen Lösungen wie beim Sprungzielpuffer: Pufferung der Rückkehradresse oder Pufferung des Befehls, der nach dem Rückkehrbefehl ausgeführt wird.)

4.1.3. Voreilende Befehlsausführung (Speculative Execution)

Hat man eine Verzweigungsrichtung vorausgesagt, so werden in dieser Richtung die nächsten Befehle geholt und vorbeugend schon ausgeführt. Bei richtiger Voraussage ist alles in Ordnung. Ansonsten war gleichsam die Arbeit umsonst.

Hinweis:
Wenn man schon so etwas implementiert, ist es wichtig, eine möglichst große Trefferrate der Sprungvorhersage zu erreichen (deshalb treibt man in dieser Hinsicht einen beachtlichen Aufwand) - denn jede falsche Voraussage bedeutet, daß man die Befehlspipeline sozusagen zurückkurbeln muß (die Pipeline muß von den "falschen" Befehlen geleert und wieder neu gefüllt werden).

4.1.4. Übergehen der Ausführungsreihenfolge

Dieses Prinzip (engl. Out-of-Order Execution) betrifft Prozessoren mit mehreren Verarbeitungswerken (Superskalar-Organisation). Jedes Verarbeitungswerk schließt selbständig die Befehlsausführung ab, sobald die jeweiligen Ergebnisse vorliegen, und zwar auch dann, wenn vorhergehende Befehle (in anderen Verarbeitungswerken) noch in Arbeit sind.

4.1.5. Datenweitergabe

Nach diesem Prinzip (engl. Data Forwarding) werden Schreib-Lese-Abhängigkeiten umgangen. Eine solche Abhängigkeit tritt beispielsweise dann auf, wenn ein Befehl Daten transportiert bzw. Ergebnisse bildet und ein nachfolgender Befehl diese wieder als Operanden benötigt. Solche Daten können gleichsam mitten aus der Pipeline heraus entnommen (und dem zweiten Befehl zur Verfügung gestellt) werden, noch bevor der erste Befehl vollständig ausgeführt worden ist.

4.1.6. Registerumbenennung (Register Renaming)

Befehle greifen naturgemäß auf die architekturseitig definierten Register zu. Diese Tatsache kann dazu führen, daß manche Beschleunigungsvorkehrungen wirkungslos und manche gar nicht durchführbar sind (z. B. die voreilende Befehlsausführung: was geschieht, wenn die Befehle Registerinhalte verändert haben, die Vorhersage aber falsch war?). Zudem kann es in einem Superskalar-Prozessor zu ganz elementaren Konflikten kommen, wenn mehrere Verarbeitungswerke auf einen gemeinsamen Registersatz zugreifen wollen. Der Ausweg: wir sehen eine größere Anzahl an Hardware-Registern vor und vergeben diese “nach Bedarf” an die in den verschiedenen Pipeline-Stufen und Verarbeitungswerken befindlichen Befehle. Die architekturseitig definierten Register werden dann den Hardwareregistern dynamisch zugewiesen, und zwar derart, daß sich aus Sicht des Programms der in der Architekur vorgesehene sequentielle Ablauf ergibt. So kann es sein, daß beispielsweise als architekturseitiges Register EBX (IA-32) zunächst das Hardware-Register 20 und später das Hardware-Register 13 verwendet wird.

4.1.7. Befehlserledigung (Instruction Retirement)

Diese Funktion dient dazu, Ordnung zu schaffen, so daß sich die Hardware dem Programm gegenüber trotz aller Beschleunigungsmaßnahmen wie eine architekturgemäß sequentiell arbeitende Maschine verhält. Ein Befehl ist erst dann “erledigt” (und kann, wie es im Englischen wörtlich heißt, gleichsam zur Ruhe geschickt werden), wenn alle Konflikte aufgelöst und die Ergebnisse an den architekturseitig richtigen Stellen hinterlegt sind.

Beispiel 1: Befehle werden voreilend ausgeführt. Die Ergebnisse werden in Hardware-Registern abgelegt. War die Verzweigungsrichtung richtig vorhergesagt worden, so werden diesen Hardware-Registern die in den Befehlen jeweils spezifizierten architekturseitigen Register zugewiesen (Beispiel: das Hardware-Register 20 wird zum architekturseitigen Register EBX). Danach gelten die Befehle als “erledigt”.

Beispiel 2: In einem Verarbeitungswerk wurde die Befehlsausführung abgeschlossen. Es kann aber sein, daß ein im Programm vorausgehender Befehl in einem anderen Werk noch in Arbeit ist - und daß dessen Ergebnis als Operand des nachfolgenden (aber bereits abgeschlossenen) Befehls dient. Es wurde also mit einem falschen Operanden gearbeitet. Ein solcher Befehl kann demnach noch nicht als “erledigt” gelten. Statt dessen ist die jeweilige Operandenverknüpfung erneut auszuführen.

4.2. Moderne IA-32-Prozessoren

4.2.1. Überblick

Die IA-32-Hochleistungsprozessoren versuchen, den innewohnenden Parallelismus in den Programmen zur Laufzeit zu erkennen und auszunutzen. Die Software merkt davon nichts - und darf auch nichts davon merken (Kompatibilität). Hierzu werden alle soeben beschriebenen Prinzipien angewandt. Die Prozessortypen unterscheiden sich im Aufwand und in Einzelheiten der Auslegung. Es geht um Marktanteile. Manche Anbieter zielen auf ausgesprochene Marktnischen*), manche wollen überall dabeisein und müssen sich dazu etwas einfallen lassen - schließlich kann man die Lösungen der Konkurrenz nicht einfach kopieren (Patentrecht). Im folgenden wollen wir einige typische Lösungen überblicksmäßig vorstellen.

*): betrifft vor allem besonders preisgünstige Prozessoren und solche mit geringem Stromverbrauch (für Value PCs und "mobile" Computer) - die Entwickler wählen hierzu verschiedenartige "Abmischungen" der bekannten Prinzipien (verschiedene Grade der Superskalarität, der Anzahl an Pipeline-Stufen usw.).

Begriffe wie "Speculative Execution", "Register Renaming" usw. werden ausgiebig gebraucht. Die einzelnen Prozessortypen unterscheiden sich aber sowohl in der Anzahl der entsprechenden Ressourcen als auch in deren Auslegung und in Einzelheiten der Wirkungsweise*).

*): die gern herausgestellte Taktfrequenz allein besagt also keineswegs alles über das tatsächliche Leistungsvermögen!

Weshalb manche superschnellen Prozessoren manchmal gar nicht so schnell sind
So etwas kann vorkommen. Beispielsweise hat es sich gezeigt, daß der Pentium Pro (P6) 16-Bit-Programme langsamer ausführt als ein vergleichbarer Pentium der P5-Klasse. Die Ursache: wie jede technische Entwicklung ist auch der "dickste" Prozessor ein ingenieurmäßiger Kompromiß - ab und an haben die Entwickler darüber zu entscheiden, was sie einbauen und was sie weglassen. Beim Pentium Pro hat man nun die 16-Bit-Verarbeitung als weniger wichtig angesehen und (im P5 vorhandene) Schaltungsmaßnahmen zu deren Ablaufbeschleunigung weggelassen (vom Pentium II an hat man in dieser Hinsicht nachgebessert).

Zudem hat so eine Superskalar-Superpipelining-Hardware ihre Eigenheiten: es lassen sich ja nicht alle Befehle unabhängig voneinander ausführen. Also muß man die Abhängigkeiten erkennen und darauf reagieren. Und je mehr Abhängigkeiten es sind, um so aufwendiger wird die Hardware. Hierbei gibt es einen alten Entwurfstrick: man nimmt bestimmte Abhängigkeiten als "normal" an und legt die Hardware dafür aus. Alles, was nicht "normal" ist, wird lediglich erkannt. Zur Behandlung dieser Fälle wird jeweils ein Mikroprogramm gerufen - und dann wird es natürlich langsam*). Eine solche Maschine ist also in allen Fällen, die den zugrunde liegenden Entwurfsentscheidungen entsprechen (sagen wir: für Win32-typischen Code) wirklich überlegen - und der Rest muß sehen, wo er bleibt.

*): das Prinzip können wir uns anhand von Abbildung 2.5 sinngemäß veranschaulichen: aus der Hardware kommen die verschiedenen Bedingungssignale, die auf einfache Weise zu einer Bedingung “o.k. (= Normalfall) / nicht o. k. (= Sonderfall)” verknüpft werden. Tritt nun ein solcher Sonderfall ein, so fragt das Mikroprogramm die einzelnen Bedingungen ab und verzweigt entsprechend.

Steuerungsprinzipien
Ein einziges Mikroprogrammsteuerwerk ist praktisch nicht in der Lage, eine Superskalarmaschine effektiv zu steuern, und sequentielle Steuerschaltungen sind für die Implementierung einer komplizierten Rechnerarchitektur (wie es IA-32 nun einmal ist) zu aufwendig.

Wir kennen bereits die Lösung, die Intel beim Pentium (P5) gewählt hat (vgl. Seite 53): nur eine der Verarbeitungs-Pipelines wird vom zentralen Mikroprogramm gesteuert und kann alle Befehle ausführen; die zweite hat eine direkte Steuerung und kann nur entsprechend einfache Befehle ausführen.

Modernere Prozessoren setzen die IA-32-Befehle in Folgen von Steuerbefehlen um, die unmittelbar auf die Hardware wirken. Bei Intel (P6) heißen diese Steuerbefehle “Micro-OPs”, bei AMD “RISC86" (K6) oder - man beachte die sinnige Abwechslung - “MacroOPs” (Athlon) usw. Was ist hierbei anders als bei herkömmlichen Steuerungsprinzipien?

- es können mehrere solcher Steuerbefehle gleichzeitig wirksam sein,
- welche Steuerbefehle gleichzeitig wirken, hängt vom Verarbeitungszustand ab,
- die Ausführung der Steuerbefehle kann ihrerseits mit den zuvor beschriebenen Verfahren (z. B. durch Ausführen “außerhalb der Reihe” - Out-of-Order-Execution -) beschleunigt werden - was in letzter Konsequenz auf eine Art Datenflußsteuerung in den Verarbeitungswerken hinausläuft (P6, Athlon).

4.2.2. P6 (Intel)

Die P6-Prozessoren sind Superskalarmaschinen mit - so Intel - dynamischer Mikroprogrammausführung (Dynamic Execution Microarchitecture). Aufbau und Wirkungsweise sind recht kompliziert, zudem werden sie nur sehr grob beschrieben. Es ist deshalb schlicht und einfach unmöglich, sich in Einzelheiten einzuarbeiten. Beschränken wir uns also auf einen Überblick (Abbildungen 4.1, 4.2).

Abbildung 4.1 P6-Prozessoren (1). Überblicksmäßiges Blockschaltbild (Intel). Erklärung der Bezugszeichen 1...7 im Anschluß an Abbildung 4.2

Kurzübersicht über den Fluß der Mikrobefehle:
a - Befehlsdecodierung liefert die Mikrobefehle, die die Ausführung des jeweiligen Befehls steuern; b - Mikrobefehle werden ausgeführt; c - Mikrobefehle werden zwecks Ausführung abgerufen; d - Befehlserledigung; e - Abruf/Zuweisung der architekturseitigen Registerinhalte.

Abbildung 4.2 P6-Prozessoren (2). Nähere Einzelheiten (Intel)

Erklärung:
1 - Befehls-Cache; 2 - Daten-Cache; 3 - Befehlsdecodierung; 4 - Mikrobefehlsverteilung und Ausführung; 5 - Mikrobefehls-Assoziativpuffer*); 6 - Befehlserledigung; 7 - architekturseitiger Registersatz; 8 - Mikroprogrammablaufsteuerung; 9 - Mikrobefehlsabruf; 10 - Verarbeitungseinheiten (Operationswerke; 5 Stück); 11 - Registerzuordnungseinheit; 12 - architekturseitiger Befehlszähler; 13 - Speicherzugriffspuffer; 14 - Sprungzielpuffer; 15 - Befehlsdecodierer (3 Stück).

*): die Intel-Begriffe: Instruction Pool oder Reordering Buffer (ROB).

Befehlslesen/Befehlsdecodierung
Die Funktionseinheit 3 besteht aus der eigentlichen Befehlsleseeinheit (Instruction Fetch Unit), den Befehlsdecodierern 15 und der Registerzuordnungseinheit 11. Die Befehle werden aus dem Befehls-Cache 1 gelesen und in Mikrobefehle (Intel: Micro-Ops) umgesetzt. Diese Mikrobefehle werden im Befehlspuffer 5 gespeichert. Der einzelne Mikrobefehl betrifft typischerweise eine elementare Operation mit 2 Operanden und einem Ergebnis (Schema: C := A **op** B).

Die aus dem Befehls-Cache 1 gelesenen Einträge werden zunächst in einzelne Befehle zerlegt, die den Befehlsdecodierern 15 zugeführt werden. Es gibt 3 parallel arbeitende Befehlsdecodierer: zwei (15a, b) für einfache und einen (15c) für komplexe Befehle. Viele IA-32-Befehle können in einen einzigen Mikrobefehl umgesetzt werden, andere lassen sich mit Folgen von maximal 4 Mikrobefehlen ausführen. Was noch komplizierter ist, wird nach Art der klassischen Mikroprogrammierung implementiert (hierfür ist die Mikroprogrammablaufsteuerung 8 zuständig).

Die Mikrobefehle enthalten besondere Bits, die das Übergehen der Ausführungsreihenfolge steuern (Out-of-Order Execution).

In jedem Taktzyklus können bis zu 6 Mikrobefehle ausgegeben werden (je einer von den Decodern für einfache Befehle und 4 vom Decoder für komplexe Befehle).

Registersatz
Neben dem Satz der architekturseitigen Register 7 sind 40 frei nutzbare Register vorgesehen, die sowohl binäre Angaben als auch Gleitkommazahlen aufnehmen können. Die von den Befehlsdecodierern 15 ausgegebenen Mikrobefehle laufen über die Registerzuordnungseinheit 11, die die jeweils zu verwendenden Registeradressen in die Mikrobefehle einträgt.

Mikrobefehls-Assoziativpuffer
Der Mikrobefehls-Assoziativpuffer 5 ist ein assoziativer Steuerspeicher (vgl. Abschnitt 1.4.), der 40 Mikrobefehle aufnehmen kann. Die Mikrobefehle werden von den Befehlsdecodierern 15 geliefert - und zwar zunächst gemäß der ursprünglichen Befehlsreihenfolge. Sie bleiben so lange im Assoziativpuffer 5, bis sie erledigt sind.

Mikrobefehlsverteiler und Ausführung
Die Funktionseinheit 4 besteht aus der Mikrobefehlsabrufeinheit 9 und den Verarbeitungseinheiten (Operationswerken) 10. Die Abrufeinheit 9 fragt den Assoziativpuffer 5 nach Mikrobefehlen ab, die jeweils ausgeführt werden können (ein Mikrobefehl kann dann ausgeführt werden, wenn alle seine Operanden verfügbar sind und wenn ein passendes Operationswerk frei ist). Da es sich um einen Assoziativspeicher handelt, kommen die Mikrobefehle gemäß der Verfügbarkeit von Operanden und Operationswerken zur Ausführung - ohne Rücksicht auf die ursprüngliche Befehlsreihenfolge (Prinzip der Datenflußsteuerung).

Es gibt 2 Verarbeitungseinheiten für Binärzahlen usw. (Integer Units 10c, d)*), 2 Gleitkomma-Verarbeitungseinheiten (Floating Point Units 10a, b - davon ist 10a auf SIMD-Befehle spezialisiert) und eine Speicherzugriffseinheit 10e. Dementsprechend können in jedem Taktzyklus bis zu 5 Mikrobefehle ausgeführt werden (Mittelwert: 3 Mikrobefehle je Taktzyklus).

*): eine davon ist für die Verzweigungen zuständig. Sie arbeitet hierzu mit dem Sprungzielpuffer 14 zusammen.

Befehlserledigung (Retire Unit)
Die Funktionseinheit 6 fragt den Assoziativpuffer 5 nach Mikrobefehlen ab, die bereits ausgeführt wurden und deren Ergebnisse nicht bzw. nicht mehr von anderen Mikrobefehlen abhängen. Diese Mikrobefehle werden aus dem Assoziativpuffer 5 entfernt; dabei werden die Ergebnisse in die architekturseitigen Register 7 übertragen. In jedem Taktzyklus können bis zu 3 Mikrobefehle “erledigt” werden.

Sprungzielpuffer
Der Sprungzielpuffer 14 hat 512 Einträge. Die Hardware ist in der Lage, aus dem Eintreffen oder Nicht-Eintreffen von Voraussagen gleichsam zu lernen.

4.2.3. K6 und Athlon (AMD)

Man verwendet ähnliche, aber in den Einzelheiten abweichende Lösungen (Abbildungen 4.3, 4.4).

Abbildung 4.3 K6 im Überblick (nach: AMD; vereinfacht)

Erklärung:

1) Befehlsvordecodierung. Die aus dem Speicher gelesenen Befehle gelangen in den Befehls-Cache. Auf dem Weg dorthin werden bestimmte Befehlsmerkmale (z. B. die Befehlslänge) bereits entschlüsselt. Die Entschlüsselungsergebnisse werden im Befehls-Cache mitgespeichert.

2) Befehlsdecoder. In jedem Takt können bis zu 16 Befehlsbytes abgeholt und ausgewertet werden (sie stammen entweder aus dem Befehls-Cache oder aus dem Sprungzielpuffer). Die decodierten Befehle werden in RISC86-Befehle umgeschlüsselt. Einige IA-32-Befehle können direkt ausgeführt werden, einige werden in einen einzigen RISC86-Befehl umgesetzt. Wenn mehr RISC-Befehle erforderlich sind: bis zu 4 RISC-Befehle kann der Decoder direkt erzeugen. Für alle IA-32-Befehle, die noch mehr RISC-Befehle benötigen, ist ein (in der Abbildung nicht dargestellter) ROM auf dem Schaltkreis vorgesehen, der die Befehlsfolgen enthält (dies entspricht praktisch der herkömmlichen Mikroprogrammsteuerung).
3) RISC-Befehlspuffer und Befehlsvermittlung (Buffer/Scheduler). Der Puffer kann bis zu 24 RISC86-Befehle zwischenspeichern. In jedem Taktzyklus können bis zu 6 RISC-Befehle an die Verarbeitungseinheiten ausgegeben werden.
4) Sprungzielpufferung. In diesem Block sind zusammengefaßt:
   - der eigentliche Sprungzielpuffer (16 Einträge mit je 16 Befehlsbytes),
   - die Branch History Table BHT (8k Einträge),
   - der Rückkehrpuffer (Return Address Stack).
5) Verzweigungsauflösung (Branch Resolving). Diese Funktionseinheit steuert den Ablauf der Verzweigungsbefehle und die Auswertung der Sprungvorhersage. Sie veranlaßt das Bestätigen voreilend (spekulativ) ausgeführter Befehle bzw. das Zurückstellen der jeweiligen Befehlswirkungen. Es können bis zu 7 Verzweigungen verfolgt werden.
6) Verarbeitungseinheiten. Es sind 4 Verarbeitungseinheiten, eine Ladeeinheit (zum Lesen von Daten aus dem Speicher) und eine Speichereinheit vorgesehen.

Hardware-Register und Registerumbenennung
Den RISC86-Befehlen stehen 48 universelle Register zur Verfügung: 24 interne und 24 umbenennbare zur Aufnahme der 8 architekturseitigen (IA-32-) Register.

Abbildung 4.4 Athlon im Überblick (AMD)

Erklärung:

1) Befehlsdecodierung: bis zu 3 Befehle können gleichzeitig decodiert und in Mikrobefehle - hier MacroOPs genannt - umgesetzt werden,
2) dies ist ein Assoziativspeicher (Reorder Buffer), der 72 Mikrobefehle aufnehmen kann,
3) die Mikrobefehle werden zwecks Ausführung an Verteilereinheiten (Schedulers) geliefert und von dort auf die einzelnen Verarbeitungseinheiten (Operationswerke) weiterverteilt,
4) Verarbeitungseinheiten für Binärzahlen usw.: 3 für die eigentliche Befehlsausführung, 3 zur Adreßrechnung (IEU = Instruction Execution Unit; AGU = Address Generation Unit),
5) 3 Verarbeitungseinheiten für Gleitkomma-, MMX- und 3DNow-Operationen. Jede Verarbeitungseinheit ist auf bestimmte Befehle spezialisiert.
6) Sprungzielcache mit 2k Einträgen,

7) wie beim K6 werden beim Füllen des Befehls-Caches bereits bestimmte Befehlsmerkmale entschlüsselt (Vordecodierung) und mitgespeichert,

8) das Prozessor-Interface zu den Steuerschaltkreisen,

9) gesondertes Interface zum L2-Cache. Dessen Steuerung (L2 Cache Controller) befindet sich bereits auf dem Prozessorschaltkreis.

4.2.4. Crusoe (Transmeta)

Um kostengünstige, leistungsfähige und stromsparende Prozessoren anbieten zu können, hat die Fa. Transmeta eine außergewöhnliche Lösung geschaffen (Abbildungen 4.5...4.8). Der eigentliche Prozessor ist eine vergleichsweise einfache VLIW-Maschine, deren Befehlsgestaltung keine Rücksicht auf IA-32 nimmt. Der Trick: die IA-32-Befehle werden *zur Laufzeit durch Software* in VLIW-Befehle umgesetzt.

Abbildung 4.5 Zum Vergleich: herkömmliche Superskalarprozessoren (Transmeta)

Erklärung:
Wir erkennen das Prinzip, das auch den vorstehend beschriebenen Intel- und AMD-Prozessoren zugrunde liegt:

- IA-32-Befehle werden gelesen und decodiert,
- sie werden in Mikrobefehle umgesetzt,
- die Mikrobefehle werden in einem assoziativen Steuerspeicher (Reordering Buffer) gepuffert und von einem Verteiler zu den Operationswerken geführt,
- die Operationen laufen ab, ohne die ursprüngliche Verarbeitungsreihenfolge zu berücksichtigen,
- die Befehlserledigung sorgt dafür, daß die Befehlswirkungen dem Programmierer gegenüber so erscheinen, wie es sich gemäß der ursprünglichen Verarbeitungsreihenfolge ergeben hätte.

Abbildung 4.6 Überblick über die VLIW-Hardware der Crusoe-Prozessoren (Transmeta)

Erklärung:
Ein VLIW-Befehl ist 64 oder 128 Bits lang und kann bis zu 4 RISC-ähnliche Einzelbefehle enthalten. Diese Einzelbefehle*) werden parallel ausgeführt; sie wirken jeweils direkt auf fest zugeordnete Verarbeitungseinheiten - es gibt keinen Verteiler, der die einzelnen Befehle jeweils auf freie Verarbeitungseinheiten leitet. Es gibt insgesamt 5 Verarbeitungseinheiten: eine für Gleitkommazahlen, 2 für Binärdaten (Integer ALUs - in der Abbildung ist nur eine dargestellt), eine Speicherzugriffseinheit (Load/Store Unit) und eine Verzweigungseinheit (Branch Unit).

*): Befehlsbeispiele in der Abbildung: FADD - Gleitkomma-Addition; ADD - Integer-Addition; LD - Laden; BRCC - bedingte Verzweigung.

Registersatz
Es sind 64 Universalregister (Integer Registers) vorgesehen, von denen einige als architekturseitige Register verwendet werden.

Es liegt nahe, mit den VLIW-Befehlen die IA-32-Architektur zu emulieren. Dann hätte man aber kaum etwas Besseres als eine klassische mikroprogrammgesteuerte Maschine, die recht leistungsbeschränkt wäre. Deshalb hat man sich etwas Neues einfallen lassen, nämlich eine Software-Schicht, die die IA-32-Befehle in VLIW-Befehlsfolgen wandelt. Der Fachbegriff: Code Morphing. Diese Software - die typischerweise in ROMs außerhalb des Prozessors gespeichert wird - holt die IA-32-Befehle aus dem Speicher und wandelt sie in VLIW-Befehle, die im Prozessor in einem sog. Translation Cache gespeichert werden.

Abbildung 4.7 Das Software-Schichtenmodell der Crusoe-Prozessoren (Transmeta)

Erklärung:
Es gibt zwei Programmschnittstellen: 1 - VLIW-Befehle; 2 - IA-32-Befehle.

Weshalb ist dieses Verfahren der herkömmlichen Emulation überlegen?

- die Software kann ganze Befehlsfolgen analysieren und so Gelegenheiten erkennen, diese Befehlswirkungen mit vergleichsweise wenigen VLIW-Befehlen nachzubilden[*)],
- was einmal gewandelt wurde, bleibt im Translation Cache stehen und ist sofort verfügbar, falls der gleiche Befehl bzw. die gleiche Befehlsfolge erneut vorkommt,
- man verspricht sich weitere Möglichkeiten der Geschwindigkeitsverbesserung dadurch, während der Codewandlung gleichsam lernen zu können - man könnte z. B. zunächst statistische Daten über das Laufzeitverhalten des jeweiligen Programms sammeln und diese anschließend zur Optimierung nutzen (betrifft die Sprungvorhersage, das spekulative Holen von Daten usw.). Ein so ausgestatteter PC würde dem Nutzer beim ersten Aufruf eines Programms zunächst recht “zähe” erscheinen, dann aber nach und nach schneller werden.

Ob das etwas wird? - Beobachten. Einerseits bietet die Idee viele Möglichkeiten, die weit über IA-32 hinausreichen. Andererseits hat man eine weitere Software-Schicht, die sicherlich nicht vollkommen fehlerfrei sein wird (und die Erfahrung zeigt: je ehrgeiziger das Vorhaben - hier: die Ausführung der IA-32-Befehle so weit wie möglich zu optimieren - desto mehr Fehler sind zu erwarten). Zudem bleiben die vielen Eigenheiten und Spitzfindigkeiten der IA-32-Architektur (Segmentierung, Seitenverwaltung, Programmausnahmen usw.). Da die Programmschnittstelle zur System- und Anwendungssoftware nach wie vor nur IA-32 ist, muß jede Eigenheit bis in’s Kleinste nachgebildet werden - und dann wird es entweder kompliziert (in der Hardware) oder langsam - womöglich mit krassen Leistungsunterschieden (man bekommt es zweifellos hin, daß Windows oder Linux vernünftig laufen, aber Software, die ohne Rücksicht auf die Programmiergepflogenheiten dieser Systemplattformen zusammengeschrieben wurde, könnte durchaus Probleme bekommen).

*): was für den Ansatz spricht: die weitaus meisten Programme werden von Compilern erzeugt - und dabei entstehen nun einmal typische Befehlsfolgen, die - wir sind hier nicht die Marketing-Abteilung eines Prozessorfabrikanten - oftmals nichts anderes sind als Umgehungen (Workarounds) unangepaßter Architekturlösungen (so schaffen sich Compiler-Autoren gerne eigene virtuelle Maschinen mit fiktiven Befehlslisten ("Zwischensprachen")). Wenn man solche Befehlsfolgen erkennen und deren Wirkung mit entsprechend schnell ablaufenden VLIW-Befehlen nachbilden kann, so könnte dies durchaus zu beachtlichen Beschleunigungen führen. Aber: eine Optimierung, die sich beispielsweise auf einen Microsoft-Compiler bezieht, kann für einen Borland-Compiler oder für einen Gnu-Compiler vollkommen unbrauchbar sein...

4.3. Expliziter Parallelismus: IA-64

Die herkömmlichen Architekturen (darunter auch IA-32) waren an sich gar nicht für das extreme Leistungsvermögen ausgelegt, das man heutzutage erwartet. Die Notwendigkeit, die Abwärtskompatibilität zu gewährleisten, zwingt zu den Tricks, die wir im Überblick beschrieben haben. Natürlich kostet das Aufwand. Ist es nicht mehr notwendig, eine 100%ige Kompatibilität zu wahren, liegt es nahe, nach neuen Wegen zu suchen. Anstelle der aufwendigen Steuer- und Überwachungsschaltungen (zum Erkennen von Abhängigkeiten, zum "Erledigen" von Befehlen usw.) könnte man mehr (oder bessere) Verarbeitungswerke, größere Caches usw. vorsehen. Im Extremfall gibt es in der Hardware gar keine Sondervorkehrungen, und der innewohnende Parallelismus muß vom Compiler erkannt werden. Wir haben aber bereits die grundsätzlichen Nachteile dieses Prinzips diskutiert. Der Ausweg ist - wie oftmals - ein Kompromiß: der Compiler erkennt den innewohnenden Parallelismus, und die ausführende Hardware steuert die parallele Abarbeitung der Befehle. Hierzu wird in den Befehlen selbst vermerkt, ob sie parallel ausführbar sind oder nicht. Ein "dicker" Prozessor (mit mehreren Verarbeitungswerken) kann diese Angaben nutzen, um die Befehle tatsächlich parallel auszuführen, während ein Billigprozessor dieselben Befehle nacheinander abarbeitet. Diesen - nicht ganz neuen - Ansatz haben Intel und Hewlett-Packard in der Architektur IA-64 verwirklicht (er heißt dort EPIC = Explicitely Parallel Instruction Computing). Anhand der Abbildungen 4.8...4.10 wollen wir einen ersten Einblick in die Wirkprinzipien geben.

Abbildung 4.8 IA-64: Befehlsformat zur Unterstützung des expliziten Parallelismus

Erklärung:
Befehle werden stets in "Bündeln" (Bundles) von 128 Bits Länge gespeichert und bearbeitet. Ein Bündel enthält 3 Befehle zu je 41 Bits sowie ein Kennungsfeld (Template) zu 5 Bits. Im Kennungsfeld ist codiert, welcher Befehl welche Funktionseinheiten belegt und welche Möglichkeiten der Parallelausführung es gibt. Beispiele: (1) Kennung = 0H: alle drei Befehle können parallel zueinander ausgeführt werden, (2) Kennung = 1H: die Befehle 0 und 1 können parallel zueinander ausgeführt werden, dann folgt Befehl 3; (3) Kennung = 2H: zunächst ist Befehl 0 auszuführen, dann folgen die Befehle 1 und 2 (parallel zueinander ausführbar).

Hinweise im Befehl

Hinweise (Hints) ermöglichen es dem Programmierer, die Hardware beim vorbeugenden Laden, bei der Sprungvorhersage usw. zu unterstützen (Abbildungen 4.9, 4.10). Diese Hinweise sind aber wirklich nur Hilfen - es kann sein, daß sie von der Hardware vollkommen ignoriert werden.

Abbildung 4.9 Ladebefehl mit Hinweis

Erklärung:
Dieser Befehl lädt den adressierten Speicherinhalt in ein Register (Registerauswahl im Feld Register 1). Die Adresse wird in einem zweiten Register erwartet (gemäß Feld Register 3). Das Hinweisfeld betrifft die Placierung der zu holenden Daten in den einzelnen Ebenen des Cache-Subsystems. Es kann einer von folgenden Hinweisen gegeben werden:

- **#** vorrangig im L1-Cache placieren (es ist bald mit weiteren Lesezugriffen zu rechnen),
- **#** im L1-Cache placieren,
- **#** in einer beliebigen Cache-Ebene placieren ("kommt nicht so darauf an").

Abbildung 4.10 Verzweigungsbefehl mit Hinweisen

Erklärung:
Dieser Befehl kann zu einer Adresse verzweigen, die entsteht, indem das Adreß-Offset-Feld vorzeichengerecht zum Inhalt des Befehlszählers addiert wird. Das ausgewählte Prädikatregister (Seite 75) bestimmt, ob verzweigt wird oder nicht. Es können mehrere Hinweise gegeben werden:

- **#** p-Hinweis (Prefetch): betrifft das voreilende Befehlslesen vom Sprungziel an. Wahlmöglichkeiten: (1) nur wenige Befehls-Bündel lesen*), (2) mehrere Befehls-Bündel lesen (die jeweilige Anzahl ist hardwarespezifisch).
- **#** wh-Hinweis (Whether Prediction): betrifft die Sprungvorhersage. Wahlmöglichkeiten: (1) nicht vorhersagen, (2) Sprung stets in Verzweigungsrichtung vorhersagen, (3) und (4) dynamische Sprungvorhersage nutzen (hat die Hardware keine dynamische Sprungvorhersage, dann bei (3) die Geradeausrichtung und bei (4) die Verzweigungsrichtung vorhersagen).
- **#** d-Hinweis (Predictor Deallocation): betrifft die zum Befehl gespeicherten Daten der Sprungvorhersage (die u. a. zu Lernzwecken dienen - vgl. Abschnitt 4.1.1.). Wahlmöglichkeiten: (1) Vorhersage-Daten erhalten, (2) Vorhersage-Daten löschen.

*): z. B. dann, wenn bald wieder verzweigt wird (u. a. in kurzen Schleifen).

Bedingungsgesteuerte Befehlsausführung (Predication)
Eine Programmverzweigung führt üblicherweise zu einer von 2 Verzweigungsrichtungen. Über die Verweigungsrichtung zu entscheiden kostet Zeit. Die bekannte Abhilfe: Sprungvorhersage. Diese ist aber mit einer Trefferrate < 100% behaftet und - vor allem wenn die Trefferrate hoch sein soll -, schaltungstechnisch recht aufwendig (Sprungzielpuffer, Branch

History Table usw.).

Eine Alternative: wir holen zunächst die Befehle beider Verzweigungsrichtungen gleichzeitig und entscheiden im Moment der Ausführung, welche wir verwenden und welche nicht. Hierzu macht man die Befehlsausführung von Bedingungs- bzw. Prädikatbits (Predicate Bits) abhängig (Predication; sprich: Preddikeeschn). Ist das jeweilige Bedingungsbit gesetzt (mit anderen Worten: ist das Prädikat erfüllt), so wird der Befehl ausgeführt, ansonsten wird er übergangen. Wirkung bei bedingten Verzweigungen: bei gesetztem Bedingungsbit wird verzweigt, bei gelöschtem Bedingungsbit nicht.

Einzelheiten:

- in der Architektur sind 64 sog. Prädikatregister pr0...pr63 spezifiziert (vgl. Abbildung 9.27 in Heft 4). Jedes Register umfaßt nur ein einziges Bedingungsbit. Prädikatregister 0 ist fest auf Eins gesetzt (dieses Prädikat ist also immer erfüllt).
- die Prädikatbits dienen sowohl zur bedingten Befehlsausführung wie auch als Flagbits zu bedingten Verzweigungen (IA-64 hat kein “richtiges” Flagregister wie etwa IA-32),
- die Befehle haben ein Prädikatregisterfeld (vgl. die Abbildungen 4.9 und 4.10). Über dieses Feld wird jeweils eines der 64 Prädikatregister ausgewählt. Soll ein Befehl immer (unbedingt) ausgeführt werden, so ist dessen Prädikatregisterfeld auf Null zu setzen (wodurch das immer erfüllte Prädikat ausgewählt wird).
- Stellen der Bedingungsbits: es gibt eigens Vergleichs- und Bittestbefehle. Ein solcher Befehl kann zwei beliebig auswählbare Prädikatregister beeinflussen. Hierzu stehen verschiedene Verfahren zur Wahl. Das einfachste: je nach Vergleichsergebnis wird das erste der angegebenen Prädikatregister auf Eins gesetzt und das zweite auf Null oder umgekehrt. (Solche Tricks sind notwendig, da die Befehlausführung bzw. Verzweigung nur bei gesetztem Prädikatbit stattfindet. Man möchte aber gelegentlich auch auf eine nicht erfüllte Bedingung hin verzweigen usw.)