Leistungsoptimierte Einzelprozessorarchitekturen digitaler Universalrechner

Wolfgang Matthes

Inhalt

1. Einführung

Die Entwicklung der Rechentechnik ist im wesentlichen darauf gerichtet, das absolute Leistungsvermögen zu erhöhen, das Preis- Leistungs- Verhältnis zu verbessern und neuartige Gebrauchseigenschaften bereitzustellen.
Vorliegende Arbeit soll auf dem Gebiet der Rechnerarchitektur dazu einen Beitrag leisten. Gegenstand ist der einzelne Universalrechner. Dessen Architekturprinzipien und Schaltungsstrukturen sollen so gestaltet werden, daß im Rahmen bestimmter Aufwandsvorstellungen (vom Mikrocontroller bis zum Supercomputer) ein jeweils optimales Leistungsvermögen verwirklicht werden kann.[1] Im folgenden geht es darum, für einschlägige Forschungsarbeiten eine Arbeitsrichtung zu begründen, erste Vorstellungen zu umreißen und Anregungen für das wissenschaftlich- technische Handeln zu vermitteln.

Abschnitt 2 gibt - aus technikwissenschaftlicher Sicht - einen Überblick über jene Grundlagen der Informationsverarbeitung, die für dieses Ziel wesentlich sind, und erläutert die eigene Arbeitsrichtung: Um die Möglichkeiten moderner Technologien in überlegene Gebrauchswerte umsetzen zu können, werden Algorithmen, Datenstrukturen, Sprachkonstrukte und Schaltungsstrukturen aus einer gleichsam ganzheitlichen Sicht betrachtet, wobei das Ziel darin besteht, technische Mittel, also Hardware- Strukturen, so leistungsfähig und zweckmäßig wie möglich gestalten zu können.

Dem liegt ein elementares Verarbeitungsmodell zugrunde, das in Abschnitt 3 erklärt wird.

In Abschnitt 4 werden die Grundlagen der Bewertung von Schaltungsstrukturen, Algorithmen und Aufwendungen dargestellt.

Abschnitt 5 gibt einen Überblick über wesentliche Tiefenstrukturen des Verarbeitungsmodells anhand einer Erfahrungsbasis, die durch eingeführte Maschinenarchitekturen und Programmiersprachen gegeben ist. Diese Darstellung liefert grundlegende, universell nutzbare Abstraktionen für Datenstrukturen, Operationen und Ablaufprinzipien , und sie vermittelt Anregungen für das Vorgehen bei weiteren systematischen Untersuchungen.

In Abschnitt 6 werden bestimmte Abstraktionen für die technische Umsetzung ausgewählt und näher beschrieben.

In Abschnitt 7 wird die Ausgestaltung der Wirkprinzipien und Schaltungsstrukturen von Universalrechnern auf Grundlage der Abstraktionen diskutiert, die in den Abschnitten 5 und 6 erläutert wurden.

Abschließend werden in Abschnitt 8 Empfehlungen für künftige Arbeiten gegeben.

---

1 Konkrete technisch- ökonomische Überlegungen, die diese Zielstellung näher begründen, enthält /245/.

2. Uberblick

2.1. Rechnerarchitektur als Technikwissenschaft

Eine hinreichend entwickelte Technikwissenschaft bildet die theoretische Grundlage des jeweiligen Gebietes der Technik; sie liefert Richtlinien, Methodenlehren, Berechnungsverfahren und Grundsatzlösungen für das Analysieren und Vervollkommnen gegebener und für das Schaffen neuer technischer Gebilde bzw. Verfahren. Als Beispiele seien die Aerodynamik, die Angewandte Mechanik und die Theoretische Elektrotechnik genannt. Sie stellen Mittel bereit, mit denen Gebilde der Technik, wie Flugzeuge, Brücken, Motore, Hochfrequenzschaltungen klassifiziert, bewertet und rechnerisch behandelt werden können. (Beispielsweise konnte bereits vor 192Ø der Verbrennungsmotor in allen entscheidenden Parametern berechnet werden.)

Das Gebiet der Rechnerarchitektur ist von einem vergleichbaren Entwicklungsstand noch weit entfernt.
Der Mangel an wissenschaftlicher Grundlegung wird offensichtlich besonders stark von jenen Fachleuten empfunden, die unmittelbar mit der Schaffung hochleistungsfähiger Maschinen befaßt sind.
So schreibt Patt in der Einführung zu einer Übersichtsdarstellung, die industriell gefertigte Hochleistungsrechner betrifft (/272/), daß Rechnerarchitektur keine Wissenschaft sei.
Lincoln (/22Ø/) verweist darauf, daß gerade auf dem Gebiet der Supercomputer viele wesentliche Entscheidungen gefühlsmäßig ("gut feel") getroffen werden und daß es in der Regel besser ist, einen einzelnen hochqualifizierten Bearbeiter mit der Architekturdefinition zu betrauen als ein Kollektiv.
Colwell schreibt in /146/: "Computer systems work not only lacks a formal foundation, it has not even progressed to the point where a taxonomy or other means of codifying existing knowledge can be constructed. Proofs cannot be expected in computer systems work, for they presuppose some set of axioms that has yet to be created."
Die meisten Standardwerke beschreiben gegebene Architekturen, vergleichen diese unter verschiedenen Gesichtspunkten und geben allgemeine Hinweise. So gibt das Buch von Myers (/86/), das als Beispiel für einschlägige Publikationen (wie etwa /5Ø/, /56/, /61/) angesehen werden kann, folgende Empfehlungen für die Ausarbeitung von Rechnerarchitekturen:

- konzeptionelle Einheitlichkeit
- Orthogonalität
- Adäquatheit zu den Nutzer- Anforderungen
- Optimierung der technischen Mittel gemäß der Nutzungshäufigkeit
- Vorkehrungen für Erweiterungen
- Unabhängigkeit von Implementierung und Technologie.

Weitere Bemühungen um Systematisierung und Begriffsbildung betreffen formale Beschreibungen (wie ISP zur Beschreibung von Befehlslisten; /4/) sowie taxonomische bzw. morphologische Schemata, z. B. die Taxonomie der Rechnerarchitektur nach

Giloi (/177/), die Klassifizierung der Rechnerstrukturen nach Flynn (/167/) oder das Computer- Spektrum nach Hockney (/196/).
Daneben wurden in den letzten Jahren umfangreiche meßtechnische Untersuchungen betrieben (Beispiele: /12Ø/, /147/, /153/, /159/-/161/, /2Ø7/, /283/. Sie haben wichtige Erkenntnisse gebracht: zur Nutzungshäufigkeit von Befehlen, zu Trefferraten bei verschiedenen Cache- Organisationen und zum innewohnenden Parallelismus in Programmen. Von dieser Erfahrungsbasis aus wurden neue, an den Meßergebnissen orientierte Architekturen definiert. So wurde die Vorgehensweise bei der Schaffung einer solchen Architektur ausdrücklich als "measurement oriented approach" bezeichnet.[1]

Gegenwärtig ist die Wissenschaft von der Rechnerarchitektur also vorwiegend eine Erfahrungswissenschaft: "Computer science is an empirical discipline...Each new machine that is built is an experiment. Actually constructing the machine poses a question to nature: and we listen for the answer by observing the machine in operation and analyzing it by all analytical and measurement means available."[2]
In diesem Wechselspiel zwischen Schaffen und Bewerten neuer technischer Lösungen ist das systematische versuchsweise Entwickeln mit dem Ausarbeiten von Versuchsanordnungen in den Naturwissenschaften vergleichbar. Um den Grad an Exaktheit zu erhöhen, werden in der Literatur Vorstellungen diskutiert, die im wesentlichen auf folgende Ansätze zurückführbar sind:

1. Axiomatisierung. Das ist die klassische Methode zur Begründung der Mathematik. Sie geht in ihrer modernen Form auf Hilbert zurück: "Wenn es sich darum handelt, die Grundlagen einer Wissenschaft zu untersuchen, so hat man ein System von Axiomen aufzustellen, welche eine genaue und vollständige Beschreibung derjenigen Beziehungen enthalten, die zwischen den elementaren Begriffen jener Wissenschaft stattfinden. Die aufgestellten Axiome sind zugleich die Definition jener elementaren Begriffe, und jede Aussage innerhalb des Bereiches der Wissenschaft, deren Grundlage wir prüfen, gilt uns nur dann als richtig, fall sie sich mittels einer endlichen Anzahl logischer Schlüsse aus den aufgestellten Axiomen ableiten läßt" (/191/). Demgemäß werden die Anforderungen an die Architektur axiomatisch formuliert, und es ist zu beweisen, daß die Axiome selbst widerspruchsfrei sind und daß die Festlegungen der Architektur die Axiome erfüllen.[3]

2. Algebraische Modellierung. Als Beispiel und Anregung sei der algebraische Ansatz zur Leistungsbewertung nach /261/ genannt. Damit wird das Leistungsverhalten von (realen) Super-

1 Es handelt sich um die Precision Architecture von Hewlett-Packard (/12Ø/, /226/, /229/).
2 Aus /87/; zit. nach /146/.
3 Die Anregung wurde /224/ entnommen. Weiterhin sei z. B. auf /97/, /194/ und /195/ verwiesen. Diese Arbeiten gelten Fragen der Programmierung; die Nutzbarkeit für das Gebiet der Architekturprinzipien bedarf weiterer Untersuchungen.

rechnern mit einer Leistungsalgebra P, einer Anwendungsalgebra H und einer Abbildung F: H -> P untersucht. P und H sind geordnete Halbgruppen, womit arithmetische Operationen, Transporte und Verzögerungen (die z. B. durch das Warten auf Speicherzugriffe bedingt sind) modelliert werden können.

3. Linearoptimierung: "If it were possible to somehow enumerate all of the constraints....and then assign appropriate relative weights to them, architectural design might be reduced to a linear programming problem" (/146/).

In der Literatur herrscht Übereinstimmung darüber, daß solche Ansätze derzeit und in nächster Zukunft nicht für das Ausarbeiten konkreter Architekturen nutzbar sind. Man wird sich also weiterhin auf empirische Grundlagen stützen müssen und kann lediglich versuchen, durch Nutzung bewährter Methoden des wissenschaftlichen Arbeitens Anteil und Einfluß "gefühlsmäßiger" Entscheidungen so gering wie möglich zu halten. Das kann aber im folgenden nur anhand überschaubarer Sachverhalte demonstriert werden, da allein die bloße Beschreibung einer neuen Rechnerarchitektur den Umfang der vorliegenden Arbeit übersteigt[1], umso mehr die ausführliche Darlegung aller Grundgedanken, Einflüsse, Probeentwürfe, Variantenvergleiche und Entscheidungen. Die folgende Aufzählung gibt einen kurzen Überblick über die Methoden, die der Arbeit hauptsächlich zugrunde liegen:

1. Systematisches Vorgehen in nachvollziehbaren Schritten.

2. Rückführung von Neuem auf Bekanntes bzw. von Kompliziertem auf Einfaches. Es wird ein Modell der elementaren Informationsverarbeitung zugrunde gelegt, das auf kombinatorische Zuordnungen, also auf aussagenlogische Verknüpfungen, die durch Boolesche Gleichungen beschreibbar sind, und auf binäre Speichermittel zurückgeht. Es wird gezeigt (in Abschnitt 3.1.), wie spezielle, für die Implementierung eines einzigen Algorithmus ausgelegte Schaltmittel schrittweise in die bekannte v. Neumann- Rechnerstruktur überführt werden können. Jede Rechnerstruktur wird als Sammlung von Ressourcen aufgefaßt, woraus die Aufgabe ersichtlich ist, diese Ressourcen so zweckmäßig wie möglich zu nutzen.

3. Systematisches Aufarbeiten der Erfahrungsgrundlagen. Das betrifft zunächst bewährte Maschinenarchitekturen und Programmiersprachen.

4. Metasprachliche Betrachtungen, also Untersuchungen der Eigenschaften von Beschreibungsmitteln der fraglichen Sachverhalte. Diese auf Tarski (/300/) zurückgehende Methode wird sowohl für die Formalisierung bestimmter Aspekte des Verarbeitungsmodells (Abschnitt 3.2.) als auch für die Auslegung codierter Beschreibungen von Informationsstrukturen (strukturell- deskriptive Angaben; Abschnitte 5 und 6) genutzt.

---

1 Vgl. den Umfang von Architekturhandbüchern (/36/, /38/). Als ersten Überblick über eigene Vorstellungen s. /248/.

2.2. Grundlagen

2.2.1. Modelle der Informationsverarbeitung

Hier geht es ausschließlich um Informationsverarbeitung mit technischen Mitteln. Eine entsprechende Einrichtung kann ganz allgemein als "black box" dargestellt werden, die über Ein- und Ausgänge mit ihrer Umgebung verbunden ist (Bild 1).
Technische Mittel sind stets für bestimmte Zwecke vorgesehen; den Belegungen der Ein- und Ausgänge kommen folglich bestimmte Bedeutungen zu. Die Informationsverarbeitung besteht darin, aus aktuellen Eingangswerten, die Träger gewisser Bedeutungen sind, in endlicher Zeit Ausgangswerte zu ermitteln, die andere Bedeutungen haben. Fragen der bedeutungserhaltenden Informationsübertragung und -wandlung werden hier nicht behandelt[1]; Fragen der Informationsspeicherung sind nur von Interesse, sofern es um Speicherung zu Verarbeitungszwecken geht.[2]
Jede informationsverarbeitende Einrichtung ist gekennzeichnet durch ihre Funktionsweise, ihre innere Struktur und ihre technischen Grundlagen.
Die technischen Grundlagen umfassen die Elemente, aus denen die Einrichtung aufgebaut ist, deren Wirkprinzipien sowie die entsprechenden Technologien.
Die Struktur wird durch die Anordnung und Verbindung der Elemente bestimmt.
Die Funktionsweise bezeichnet die Prozesse der Informationsverarbeitung im einzelnen. Dabei ist zwischen äußerer und innerer Funktionsweise zu unterscheiden: erstere beschreibt die Informationswandlung zwischen den Ein- und Ausgängen der "black box" ohne, letztere mit Bezug auf die innere Struktur.

Es sind verschiedene Modelle der Informationsverarbeitung denkbar. Jedes Modell steht für eine bestimmte Betrachtungsweise; es beschreibt einen grundlegenden Ansatz bzw. eine konzeptionelle Auffassung, wobei jeweils bestimmte Gesichtspunkte hervorgehoben und andere vernachlässigt werden. So können beispielsweise im Vordergrund stehen:

- die technischen Grundlagen
- die Art und Weise der Informationsdarstellung
- die strukturelle Gestaltung
- die innere Funktionsweise
- die äußere Funktionsweise
- die Anwendungsgebiete.

Dem Anliegen der Arbeit gemäß soll sich die weitere Betrachtung auf digital arbeitende, vorzugsweise programmgesteuerte Einrichtungen auf mikroelektronischer Grundlage beschränken.

---

1 Das ist beispielsweise Gegenstand der Informationstheorie, der Nachrichtentechnik, der Theorie der „Informationswandler usw.

2 Das betrifft vorzugsweise Speicher mit wahlfreiem Zugriff für binär codierte Information (Flipflops und RAM- bzw. ROM-Zellen).

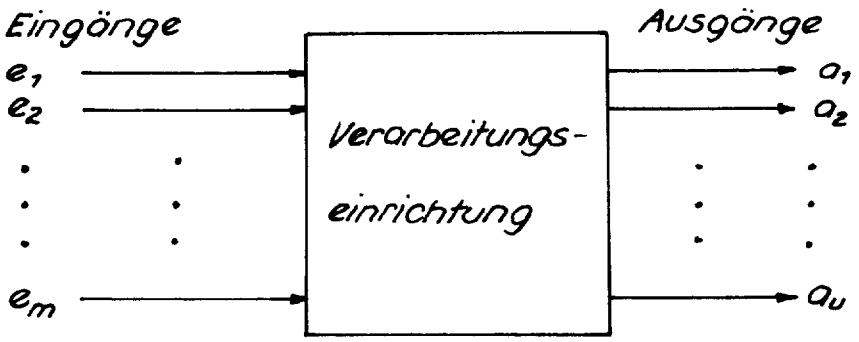


Bild 1 "Black box" Darstellung einer techn. Einrichtung zur Informationsverarbeitung

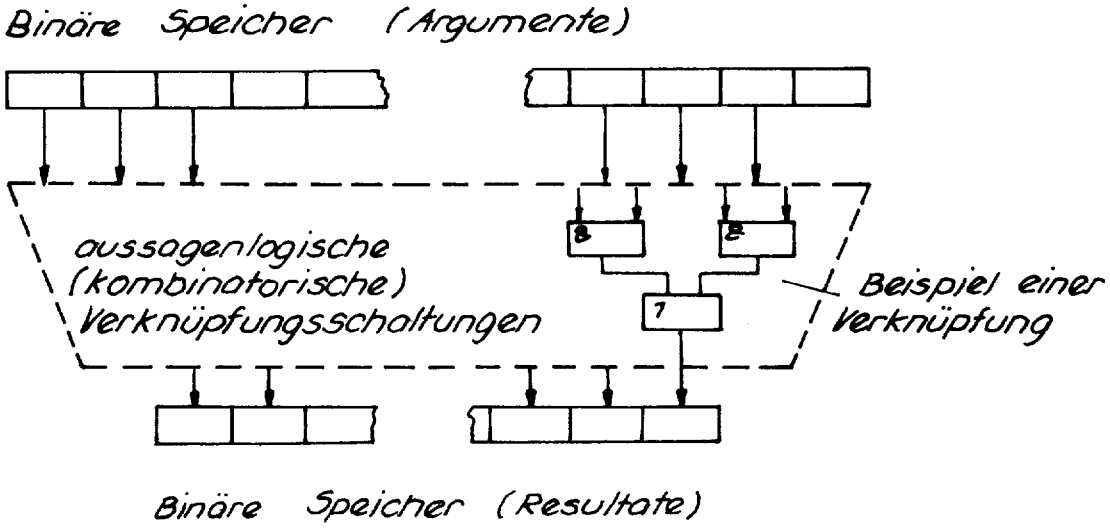
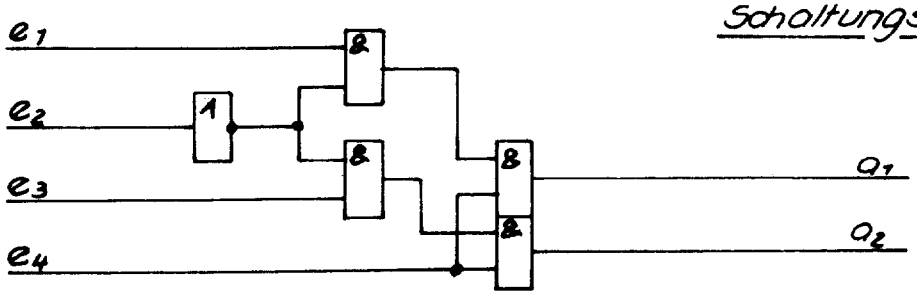


Bild 2 Allgemeines Schema der binären Informationsverarbeitung

Bild 3 Beispiel einer Verknüpfung mit Schaltungstiefe 3



Folgende Prinzipien haben sich im Ergebnis der bisherigen Entwicklung durchgesetzt:

1. Informationsdarstellung durch binäre Codierung

2. Verknüpfungen gemäß den Prinzipien der Aussagenlogik

3. algorithmische, deterministische Arbeitsweise

4. getaktete Arbeitsweise (diskrete Zeitschritte)

5. Programmsteuerung

6. weitgehende Flexibilität bis hin zur Universalität, so daß durch Wechseln des steuernden Programms eine Einrichtung beliebige Algorithmen ausführen kann (Beschränkung lediglich durch Zeitbedarf bzw. Umfang der vorhandenen Ressourcen)

7. Aufbau mit mikroelektronischen Schaltungen (Grundlagen: Schaltungen für elementare aussagenlogische Verknüpfungen (Gatter) sowie Speichermittel (Flipflops, RAM- und ROM- Zellen).

Dafür werden absolute Leistungsgrenzen diskutiert; im Anschluß soll gezeigt werden, welche Konsequenzen sich ergeben, wenn die genannten Prinzipien geändert bzw. durch andere ersetzt werden.

2.2.2. Leistungsgrenzen eines elementaren Systems

Das Leistungsvermögen betrifft sowohl den Zeitbedarf für die jeweilige Verarbeitungsaufgabe als auch die Möglichkeit, eine solche Aufgabe überhaupt ausführen zu können. Es wird bestimmt durch:

- die Kompliziertheit der Verarbeitungsaufgabe
- die Ausbreitungsgeschwindigkeit der informationstragenden Signale und die Länge der Signalwege
- die Arbeitsgeschwindigkeit der Verarbeitungseinrichtungen
- die vorgesehenen Aufwendungen.

Zunächst sollen die Leistungsgrenzen der elementaren binären Informationsverarbeitung mit aussagenlogischen Verknüpfungen untersucht werden. Dem liegt das allgemeine Schema nach Bild 2 zugrunde: gespeicherte Bits werden miteinander verknüpft; die entstehenden Resultate werden ebenfalls gespeichert.
Die Leistungsfähigkeit wird durch die konkrete Ausgestaltung dieses Schemas bestimmt. Die Zeit für einen Verknüpfungszyklus ergibt sich zu:

$$t_c = s \cdot t_p + t_A + t_{SETUP} + t_L + t_{TOL}. \qquad (2.1)$$

Bedeutung der Symbole:

t_c : Zykluszeit

t_p : Verzögerungszeit des einzelnen Gatters (Richtwerte moderner Technologien: CMOS 1 - 2 ns, ECL 350 ps, GaAs 150 ps)[1]

s : Schaltungstiefe (s = 1,2,...,n); der Wert drückt aus, wieviele Gatter im ungüstigsten Fall nacheinander durchlaufen werden müssen (Bild 3 zeigt ein Beispiel; dort ist s = 3)

t_A : Zugriffszeit zu den gespeicherten Bits (Richtwert: 4 t_p)[2]

t_{SETUP} : Vorhaltezeit für die Resultatübernahme (Richtwert: 4 t_p)[3]

t_L : Summe der Laufzeiten durch alle Verbindungsleitungen (für jedes Resultatbit zu bestimmen, der ungünstigste Wert wird zugrunde gelegt); Näherungswert: 1 ns/15 cm.

t_{TOL} : technisch bedingte Zugabe (Toleranz- Ausgleich); Richtwert: wenige t_p, stark von technologischen Gegebenheiten abhängig.

Betrachtet man Bild 2 als Illustration für das Operationswerk eines Universalrechners und nimmt man idealisierte Betriebsbedingungen an, so ergibt

$$P = \frac{1}{t_c} \qquad (2.2)$$

die Maximalleistung in "Befehlen pro Zeiteinheit". Die idealisierten Betriebsbedingungen bestehen darin, daß eine lückenlose Aneinanderreihung von Operandenverknüpfungen angenommen wird, also mit zusätzlichen Schaltmitteln parallel die steuernden Befehle gelesen, die Operanden herangeschafft und die Resultate abtransportiert werden.

Welche Möglichkeiten gibt es, die Parameter zwecks Leistungssteigerung zu beeinflussen?

t_p: Die Schaltkreistechnologie hat nach wie vor beachtliche Auswirkungen auf die Hardwarekosten: je "schneller" die Technologie, um so bedeutsamer die Zusatzaufwendungen (Kühlung, Stromversorgung usw.), um so geringer der fertigungstechnisch beherrschbare Integrationsgrad. t_p kann aus physikalischen Gründen nicht beliebig vermindert werden. Für elektronische Technologien werden bis zu 30 ps angestrebt[4], bei optischen Prinzipien hofft man, 1 ps noch "auf dem Wege normaler Ingenieurarbeit" erreichen zu können.[5]

---

1 Zu GaAs s. /305/; /278/ beschreibt ein neues Logikprinzip auf Si- Basis, das besser als ECL sein soll (190 ps).
2 Zeit zwischen Taktflanke und Verfügbarkeit der Information.*
3 Zeit zwischen Bereitstellung und Übernahme der Information.*
4 Vgl. /206/.
5 Nach /315/; t_p = 100 fs erscheint noch möglich (absolut kürzeste Impulsdauer: 10 fs).
* Richtwert nach Datenblättern üblicher Logikbaureihen.

t_A , t_{SETUP} , t_{TOL} hängen ebenfalls direkt mit der Technologie zusammen (vgl. die Richtwerte auf S. 12). Die Verminderung von t_{TOL} erfordert hohe fertigungstechnische Aufwendungen (Präzision der Leiterplattenfertigung, aufwendige Meßtechnik, ggf. Abgleichvorgänge).

t_L: Die Schaltkreistechnologie bestimmt in erster Linie die physische Größe. Wenn das Schema gem. Bild 2 auf einen Schaltkreis paßt, ist t_L < 1 ns und üblicherweise vernachlässigbar. Kritisch sind die Übergänge zwischen den Schaltkreisen. Die notwendigen Koppelstufen verlangsamen den Informationstransport, und zwar um so mehr, je leistungsfähiger die Basistechnologie ist (bei GaAs hat das Laufzeitverhältnis zwischen internen Verbindungen und solchen, die Schaltkreisgrenzen überschreiten, die Größenordnung 1:1Ø). Optische Verbindungen sind schneller (bis 3Ø cm/ns), wobei die Laufzeit nicht durch induktive, kapazitive und ohmsche Belastung verlängert wird.

s: Die Schaltungstiefe wird vom Gatter- Sortiment bestimmt. Entscheidende Parameter sind:

- Eingangszahl: übliche Werte liegen zwischen 2 und 8; bei großer Zahl an Eingängen wird die Schaltungsstruktur größer und daher langsamer.
- Funktionsvielfalt. Die Verknüpfungen werden mit Transistorstrukturen realisiert. Komplizierte Verknüpfungen bedingen ausgedehntere Anordnungen mit entsprechend längeren Verzögerungszeiten, deshalb gibt es in den meisten Technologien nur einen Typ elementarer Verknüpfungen (NAND bzw. NOR).
- Anzahl der nachschaltbaren Eingänge ("fan out").

Alle Verknüpfungen werden letztlich durch Boolesche Gleichungen beschrieben, die für jedes Resultatbit die Abhängigkeit von den jeweiligen Argumentbits angeben. Eine bestimmte Schaltungstiefe s ist dann erreichbar, wenn sich für jede Boolesche Gleichung eine Dekomposition für das jeweilige Gattersortiment[1] derart angeben läßt, daß für kein Netzwerk s überschritten wird. Beim Streben nach höchstem Leistungsvermögen hat man somit nur 2 Alternativen:

1. Beschränkung von s im Hinblick auf ein bestimmtes kleines t_c, indem nach jeweils s Gatter- Stufen Speichermittel eingefügt werden. Kompliziertere Operationen werden so durch eine Kaskaden- Anordnung von Schaltungen gem. Bild 2 verwirklicht: das ist das bekannte Pipelining- Prinzip.

2. Anordnung kombinatorischer Schaltmittel für die gewünschten komplizierten Verknüpfungen, wobei t_c nach dem notwendigen s gewählt wird; das Ziel besteht darin, eine Verarbeitungsaufgabe statt in n Schritten mit $t_c 1$ in einem Schritt mit $t_c 2$ auszuführen.

Beim Pipelining wird das einzelne Resultat nicht schneller gebildet als bei kombinatorischer Zuordnung, sondern es wird

1 Zur Theorie s. /52/; zur rechenpraktischen Nutzung /164/.

mehr Zeit benötigt, nämlich für jede eingefügte Speicherstufe ein Taktzyklus. Der Vorteil besteht vielmehr darin, daß bei n Speicherstufen n-1 Operationen parallel mit einem Zeitversatz von jeweils t_c ausgeführt werden können. Das Prinzip lohnt sich also nur, wenn eine Vielzahl gleichartiger Operationen auszuführen ist ("Vektorrechner").[1] Der 2. Ansatz läuft darauf hinaus, sehr komplexe Verknüpfungen vorzusehen. Diese sind aber nicht immer universell nutzbar ("Spezialrechner"). Für die einzelne Operation ist dieser Ansatz überlegen, wenn gilt:

$$t_c 2 < z t_c 1, \qquad (2.3)$$

wobei z die Zahl der Taktzyklen angibt, die notwendig sind, um die Operation mit den einfacheren Verknüpfungen auszuführen. Läßt sich beim Pipelining die Parallelverarbeitung ausnutzen ("Vektorisierung"), so müßte gelten [2]

$$t_c 2 < t_c 1/n, \qquad (2.4)$$

wenn die kombinatorische Zuordnung überlegen sein soll.

2.2.3. Alternative Prinzipien

Einen Anwender interessiert an sich nur die reine Verarbeitungszeit für sein Problem und nicht die interne Struktur der Verarbeitungseinrichtung. Es ist deshalb gerechtfertigt, die allgemein üblichen Prinzipien (S. 11) in Frage zu stellen:

1. keine binäre Codierung, also mehrwertige Codierung, Analogdarstellung u. a.

2. andere als aussagenlogische Verknüpfungen. Beispiele: mehrwertige Logik, Wahrscheinlichkeitslogik, Schwellwertlogik, assoziative Verknüpfungen, neuronale Prinzipien

3. Verzicht auf algorithmische Arbeitsweise, also assoziative Arbeitsweise, Selbstorganisation usw.

4. keine getaktete Arbeitsweise, stattdessen kontinuierliche Zeitabläufe

5. keine Programmsteuerung. Alternativen: entweder Zwangssteuerung für einen bestimmten Zweck oder Prinzipien des Lernens und der Selbstorganisation

6. Verzicht auf Universalität und Flexibilität, also: Einzwecksysteme

7. andere technische Grundlagen, z. B. optische oder biochemische Prinzipien.

---

1 Deshalb sind manche Hochleistungsschaltkreise (bes. solche für Gleitkommaoperationen) entsprechend umschaltbar.

2 Näherungsweiser Ausdruck; n: Anzahl der Pipeline- Stufen. Mit praktischen Werten um 4...16 ist (2.4) kaum zu erfüllen.

Manche dieser Alternativen sind in der Technik bereits verwendet worden. Sie sind aber mit spezifischen Problemen verbunden (Beispiel: Genauigkeit bei der analogen Informationsverarbeitung).
Andere Alternativen sind technisch ohne weiteres durchführbar; ihre Verwirklichung ist eine Frage von Kosten- Nutzen- Rechnungen (z. B. Einzweckmaschinen im Vergleich zu Universalmaschinen).
Einige Prinzipien haben - für sich gesehen - durchaus Vorteile. Aus der Sicht des Technikers ist aber zu prüfen, ob im nutzbaren Erzeugnis Kosten- bzw. Leistungsvorteile tatsächlich verwirklicht werden können. Beispiele:

- Mehrwertige Logik oder Schwellwertlogik: Fragen der Störsicherheit, der Schaltgeschwindigkeit und der Fertigungstoleranzen
- asynchroner Betrieb: Beherrschbarkeit in den Fertigungs- und Prüfprozessen der Schaltkreise und Systeme.

Manche Prinzipien bedürfen noch umfangreicher Forschungsarbeiten, bevor sie für eine Nutzung in Betracht gezogen werden können. Die Forschungen betreffen sowohl die Erhöhung der Verarbeitungsgeschwindigkeit im Rahmen eingeführter Modelle der Informationsverarbeitung als auch vollkommen neuartige Lösungsansätze. Tafel 1 vermittelt einen Eindruck von den vielfältigen Möglichkeiten.

Wichtige Leistungen der Informationsverarbeitung lebender Systeme konnten bisher nur sehr unvollkommen oder gar nicht technisch nachgeahmt werden. Die bekannten Schaltmittel haben aber bereits einen Geschwindigkeitsvorteil von $1:10^3 \ldots 1:10^6$ gegenüber den Nervenzellen.[1] Folglich kann bloße Geschwindigkeitssteigerung nicht der einzige Weg sein; vielmehr sind künftig neue Ansätze zur technischen Informationsverarbeitung notwendig, die über den Gedankenkreis "berechenbare Funktionen und Algorithmen" hinausgehen (man muß in einem bestimmten Zeitabschnitt t_c [2)] offenbar wesentlich mehr leisten, als Verknüpfungsresultate gemäß der Aussagenlogik zu bilden).
Für eine technische Umsetzung solcher Vorstellungen gibt es noch keine gesicherte Grundlage.[3] Es ist deshalb gerechtfertigt, Bemühungen um bessere technische (also: beherrschbare und anwendbare) Lösungen auf die Mikroelektronik und die bekannten Prinzipien der digitalen Informationsverarbeitung zu stützen. Wegen der wissenschaftlichen, technischen , anwendungspraktischen und wirtschaftlichen Bedeutung ist die weitere Beschränkung auf programmgesteuerte Rechner zweckmäßig.
Nachfolgend werden die wesentlichen Entwicklungswege dieser Technik kurz skizziert.

---

1 Bezogen auf die gängigen Modellvorstellungen zur Arbeitsweise der Neuronen.
2 Nach /28/ erkennt der Mensch ein Bild in 20-30 ms, d. h. mit höchstens wenigen hundert Verarbeitungsschritten.
3 Trotz aller Bemühungen dürften solche Prinzipien für die nächsten Produkt- Zyklen (Forschung -> Entwicklung -> Markteinführung -> Effekte beim Anwender) nicht wirksam werden.

Ziel	Gegenstand der Bemühungen		
	Struktur		Technologie
höhere Geschwindigkeit	• einfachere Strukturen für kurze Zykluszeiten (RISC) • Pipelining • mehrere Verarbeitungswerke: – Scoreboard – VLIW – Datenflußprinzipien	In jeder Hinsicht:	• GaAs • HEM- Transistoren • CCL (/278/) • Josephson- Effekt • optische Prinzipien
komplexere Funktionen	• Sondermaschinen • mehrwertige Logik • zelluläre Systeme • neuronale Systeme	Massive Parallelisierung	• optische Prinzipien (z. B. holographische) • biochemische Prinzipien • mehrwertige bzw. Analog- Prinzipien (z. B. Schwellwertlogik)

Tafel 1 Grundsätzliche Möglichkeiten zur Verbesserung der Verarbeitungsleistung

2.3. Entwicklungswege zum heutigen Stand der Rechnerarchitektur

2.3.1. Universalrechner

2.3.1.1. Herkömmliche Architekturen (CISC)

Die Entwicklung der Rechnerarchitektur war in den 70er Jahren zu einem gewissen Abschluß gekommen (Beispiele: IBM/370, CDC 6600 und 7600, DEC PDP 11 und VAX 11). Die seinerzeitigen Lösungen wurden maßgeblich von folgenden Sachverhalten bestimmt:

1. Speichermittel mit wahlfreiem Zugriff waren kostbare Ressourcen. Große Speicherkapazitäten konnten praktisch nur mit Ferritkernspeichern realisiert werden; das führte aus technischen Gründen zur Trennung zwischen zentralen Speichern und angeschlossenen Verarbeitungseinrichtungen.

2. Der Integrationsgrad der Schaltmittel war vergleichsweise gering, die Geschwindigkeit aber bereits recht hoch (CDC 6600: mit diskreten Transistoren 100 ns Zykluszeit bereits 1965).

3. Es bestand eine Diskrepanz zwischen der Zykluszeit der Verarbeitung und der des Speichers (Richtwert: 1:5...1:10).

4. Schnelle Registerspeicher waren teuer. Übliche technische Lösungen: Flipflops (360/75, EC 1040), Kondensatorspeicher (360/65), schnelle Kernspeicher (360/50, Siemens 4004), Dünnschichtspeicher (Univac).

5. Große Festwertspeicher für Mikroprogramme waren vergleichsweise kostengünstig (z. B. nach dem Transformatorprinzip arbeitend, einige tausend Worte mit über 100 bit, 200 - 400 ns Zyklus).[1]

6. Es wurde überwiegend mit Maschinenbefehlen programmiert (Assembler- Programmierung). Das führte dazu, bei der Gestaltung der Architektur nach möglichst leistungsfähigen und flexibel nutzbaren Maschinenbefehlen zu suchen (die langsame Verbindung zwischen Speicher und Prozessor mußte gut ausgenutzt werden, die Mikroprogrammsteuerung erlaubte es, komplizierte Verarbeitungsabläufe 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 anbieten zu können.

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

---

1 Beispiel: Transformatorspeicher der EC 1040 (/40/).

Die Mikroprozessoren, die in den 7Øer Jahren aufkamen, waren keine architekturseitigen, sondern technologische Neuerungen. Es ist erst in den 8Øer Jahren gelungen, die Vielfalt der eingeführen Architekturprinzipien in Mikroprozessoren anzubieten (ein typisches Beispiel ist die Schaltkreisfamilie Motorola 68 ØØØ...68 Ø4Ø).

2.3.1.2. Architekturen mit reduzierten Befehlslisten (RISC)

Maschinen mit vergleichsweise elementaren Befehlslisten gibt es seit der Frühzeit der Rechentechnik. In den 7Øer Jahren begannen systematische Untersuchungen, wobei folgende Prinzipien im Zusammenhang betrachtet wurden:

1. das System ist ausschließlich auf die Programmierung in höheren Programmiersprachen orientiert

2. es wird ein elementarer Befehlssatz vorgesehen, der vollständig "hart verdrahtet" werden kann (kein Rückgriff auf Mikroprogrammsteuerung); Ziel ist, die meisten Befehle jeweils in einem Zyklus auszuführen

3. hochentwickelte Compiler zur praktischen Nutzung der Prinzipien 1 und 2.

Diese Forschungsrichtung wurde maßgeblich durch folgende Sachverhalte angeregt:

- Es wurden zunehmend höhere Programmiersprachen verwendet.

- Untersuchungen zur Nutzungshäufigkeit von Befehlen in einer Vielzahl compilierter Programme haben ergeben, daß elementare Befehle weitaus am häufigsten benutzt werden.

- Mit dem Einsatz der Halbleiterspeichern konnten die Zykluszeiten von Speicher und Verarbeitungslogik zunehmend angeglichen werden (im besonderen wurden große Cache- Speicher realisierbar, die bei akzeptablen Trefferraten mit derselben Zykluszeit wie die Verarbeitungslogik betrieben werden konnten).

- Große Registerspeicher wurden kostengünstig realisierbar.[1]

Die entscheidende Überlegung besteht darin[2]: Die elementaren Befehle (LOAD, STORE, ADD, COMPARE, BRANCH usw.) werden weitaus am häufigsten benutzt. Solche Befehls seien beispielsweise mit einer Schaltungstiefe s = 1Ø zu implementieren, wodurch sich eine entsprechend kurze Zykluszeit t_c verwirklichen läßt. Wird durch Hinzufügen komplexerer Befehle die Schaltungstiefe nur um 1 erhöht, so werden alle Abläufe um 1Ø% langsamer. Die Nutzungshäufigkeit und Leistungsfähigkeit der komplexen Befeh-

---

1 Auf demselben Schaltkreis wie die Verarbeitungslogik (oft 32 Register zu 32 bit; eine Architektur hat 192 Register).

2 Die Darstellung folgt der "klassischen" Veröffentlichung von Radin (/283/). Eine weitere grundlegende Arbeit ist /273/.

le muß dann diesen Verlust überkompensieren und die zusätzlichen Kosten rechtfertigen.

2.3.1.3. Interne Parallelarbeit

Elementare Formen der internen Parallelarbeit sind gegeben durch die Überlappung verschiedener Phasen des Befehlsablaufs: Holen der nachfolgenden Befehle, Heranschaffen von Operanden, deren Verknüpfung, Abtransport der Resultate. Es wird also der innewohnende Parallelismus der Befehlsablaufsteuerung ausgenutzt.
Um den innewohnenden Parallelismus in Programmen zu nutzen, sind mehrere Operationswerke erforderlich, so daß Operationen, die nicht voneinander abhängen, gleichzeitig ausgeführt werden können.
Bei beiden Formen der Nutzung des internen Parallelismus ist es wesentlich, wie dieser erkannt und gesteuert wird.
Das ist zum einen ausschließlich mit schaltungstechnischen Vorkehrungen möglich. Befehle werden nacheinander automatisch vorbeugend gelesen, Operandenadressen berechnet usw. Sonderfälle, wie Verzweigungen, Schreiben auf Adressen, von denen bereits voreilend Operanden gelesen wurden usw. werden von besonderen Schaltmitteln erkannt und automatisch korrigiert.[1]
Auf ähnliche Weise sind mehrere Verarbeitungswerke nutzbar.[2]
Ein neuerer Ansatz besteht darin, auf Schaltmittel zum Erkennen und Beheben von Konflikten zu verzichten und diese Aktivitäten dem Compiler zu übertragen. Schaltungstechnisch sind nur sehr elementare Formen der überlappenden Arbeitsweise vorgesehen (z. B. das voreilende Befehlslesen). Der Compiler kennt das Taktschema der Zielmaschine. Er muß potentielle Konflikte erkennen und durch Umstellen bzw. Ändern der Befehlsfolge auflösen. Bei Anordnung mehrerer Operationswerke wird dem Compiler deren Ausnutzung übertragen. Dazu sind die Befehlsformate so gestaltet, daß alle Werke gleichzeitig gesteuert werden können. Solche Befehle sind extrem lang (VERY LONG INSTRUCTION WORD VLIW).[3] Der Vorteil des compiler- gestützten Ansatzes liegt zum einen darin, daß die Hardware einfach gehalten werden kann (hohe Geschwindigkeit durch geringe Schaltungstiefe), und zum anderen darin, daß man hoffen kann, durch Analyse des gesamten Programmtextes mehr Möglichkeiten zur Parallelarbeit zu erkennen als durch Auswertung des Befehlsstromes zur Laufzeit. Allerdings sind auch schwerwiegende Nachteile nicht zu übersehen:[4]

- solche extrem optimierenden Compiler werden langsam und unzuverlässig
- es ist nicht mehr möglich, programmkompatible Rechnerfamilien zu schaffen: jede Änderung etwa der Taktverhältnisse an der Befehlspipeline erfordert eine Neucompilierung, auch bei vollständig identischer Befehlsliste.

---

1 Eine konkrete Lösung ist in /40/ beschrieben.
2 Das wurde bereits bei CDC 6600 verwirklicht ("scoreboard").
3 Eine solche Maschine ist in /147/ beschrieben.
4 Darauf wird z. B. von Wirth in /330/ hingewiesen.

2.3.1.4. Architekturen mit Orientierung auf höhere Programmiersprachen

Mit zunehmender Nutzung höherer Programmiersprachen erschien es sinnvoll, Maschinenarchitekturen so auszubilden, daß sie direkt an bestimmte Sprachen angepaßt sind. Dazu hat es viele Vorschläge und auch einige ausgeführte Maschinen gegeben.[1] Solchen Lösungen ist bisher der entscheidende Durchbruch versagt geblieben. Im besonderen wurde das Leistungsvermögen bemängelt. Dazu einige Anmerkungen:

1. Das Preis- Leistungs- Verhältnis verschlechtert sich grundsätzlich, wenn Aktivitäten, die ein Compiler erledigen kann, zur Laufzeit von Schaltmitteln ausgeführt werden.

2.. Die Ausrichtung der Architektur auf ein einziges Sprachkonzept ist eine fast sicher Garantie für Erfolglosigkeit in wirtschaftlicher Hinsicht; zumindest ein beachtliches Risiko (Änderung der Sprachumgebung während der Entwicklung; Marktsituation im Fertigungszeitraum usw.).

3. Bei manchen Sprachkonzepten (z. B. Smalltalk) können bestimmte Aktivitäten nicht vom Compiler übernommen werden; sie sind grundsätzlich zur Laufzeit auszuführen. Dann ist natürlich eine schaltungstechnische Unterstützung von Vorteil.

4. Wenn der Compiler alle Aktivitäten erledigen muß (Typkontrolle, Variablenbindung usw.), kann bei hochentwickelten Programmiersprachen und großen Programmkomplexen die Compilierzeit untragbar werden; das hat z. B. einen führenden Anbieter von Ada- Systemen dazu veranlaßt, eigene Hardware (mit Vorkehrungen zur Laufzeit- Unterstützung) zu entwickeln.

5. Es geht grundsätzlich um die Frage, welche funktionelle Anforderung auf welcher Architekturebene (Hardware, Mikroprogramm, Laufzeitsystem usw.) am besten zu implementieren ist ("function to level mapping"). Um dafür eine Methodenlehre angeben zu können, stehen noch nicht genügend experimentelle Daten zur Verfügung (/146/).[2]

6. Um überzeugende Leistungsvorteile zu erzielen, ist es notwendig, unkonventionelle Schaltungsanordnungen vorzusehen, die gewisse Mindestaufwendungen erfordern.[3] In /146/ wurde im einzelnen nachgewiesen, daß Leistungsverluste des iAPX 432 gegenüber herkömmlichen Architekturen ihre Ursache in unzureichender Hardwarestruktur bzw. Ressourcenausstattung haben. (Der iAPX 432 ist 4...2Ø mal langsamer als der 8Ø86. Aber: er ist 1Ø mal schneller als ein softwareseitig implementiertes Laufzeitsystem.)

---

1 Beispiele u. a. in /79/, /86/, /88/, /89/, /186/.
2 Erklärlich, weil bisher die meisten Forschungsarbeiten auf Leistungsverbesserungen der eingeführten Architekturen, auf RISC- Konzepte und auf Parallelverarbeitung gerichtet waren.
3 D. h. die über den Aufwandsrahmen von Mikroprozessoren hinausgehen.

2.3.1.5. Universelle Emulatoren

Seit schnelle ladbare Mikroprogrammspeicher großer Kapazität verfügbar sind, gibt es Versuche, die Ebene der Mikroprogrammierung dem Anwender zugänglich zu machen, teils um auf einer Hardware unterschiedliche Befehlslisten implementieren zu können, teils um die Geschwindigkeitsvorteile von Mikroprogrammen gegenüber üblichen Maschinenprogrammen wirksamer zu nutzen.[1]
Solche Maschinen eignen sich gut für Sonderanwendungen und dazu, existierende Software, die nicht ohne weiteres umgeschrieben werden kann, durch Emulation der jeweiligen Befehlsliste weiterhin abzuarbeiten.
Die Geschwindigkeitsvorteile beim Übergang vom Maschinenprogramm zum Mikroprogramm liegen erfahrungsgemäß bei 1:2 bis etwa 1:2Ø (nach /146/ 1:8...1:15). Wodurch kommen sie zustande?

1. Das Mikroprogramm nutzt unmittelbar die Hardware; ein Mikrobefehl wird - in den weitaus meisten Fällen - in einem Taktzyklus abgearbeitet.

2. Mit leistungsfähigen Mikrobefehlsformaten lassen sich Datenwege und Verknüpfungsschaltungen direkt steuern, so daß, wenn immer möglich, alle Schaltmittel parallel ausgenutzt werden können. Des weiteren kann man Mikroprogrammsteuerungen so auslegen, daß Verzweigungen (auch in mehrere Richtungen) keine zusätzlichen Zyklen erfordern.

Die Beziehungen zu RISC- bzw. VLIW- Konzepten sind offensichtlich. RISC- Befehle entsprechen den "vertikalen" Mikrobefehlsformaten (kurze Mikrobefehle, zumeist nur eine Wirkung pro Mikrobefehl); VLIW- Befehle entsprechen den "horizontalen" Mikrobefehlsformaten (lange Mikrobefehle mit mehreren parallelen Wirkungen).[2]
Zwischenzeitlich wird es offenbar beherrscht, aus Programmtexten in üblichen Programmiersprachen gute Mikroprogramme zu compilieren: - eine sehr notwendige Voraussetzung für die praktische Nutzung, denn bisher haben Anwender nur selten von der Möglichkeit Gebrauch gemacht, eigene Befehlslisten zu definieren und zeitkritische Abläufe direkt in Mikroprogramme umzusetzen.[3]

---

1 Beispiele: Emulator- und Assist- Vorkehrungen (Burroughs 17ØØ, S/36Ø, S/37Ø, ESER); dem Nutzer zugängliche Mikroprogrammspeicher (PDP 11/7Ø); anwenderseitig mikroprogrammierbare Maschinen (Interdata, MLP 9ØØØ, WISC (/254/)).

2 Praktische Unterschiede: Mikroprogramme werden in besonderen Speichern oder dem Nutzer unzugänglichen Speicherbereichen (z. B. bei '36Ø/25) gehalten. Bei den meisten Maschinen sind die Mikrobefehlsformate nicht mit dem Ziel einer eigenständigen regulären Architektur, sondern einer kostengünstigen Emulation der Zielarchitektur ausgelegt worden.

3 Ein Programmsystem wird z. B. in /334/ beschrieben. Ohne Unterstützung beträgt die Produktivität oft nur einige hundert Mikrobefehle je Programmierer und Jahr.

2.3.2. Vektorrechner

Die konsequente Verwirklichung des Pipeline- Prinzips ermöglicht es, eine Vielzahl gleichartiger Operationen mit geringem Zeitversatz (jeweils 1 Taktzyklus) parallel mit denselben Schaltmitteln auszuführen.
Damit lassen sich sehr kurze Taktzyklen verwirklichen (Richtwert derzeit 4- 1Ø ns).[1]
Erfahrungsgemäß lohnt es sich nicht, die Anzahl der Pipeline-Segmente über 6- 8 zu erhöhen; mehr als durchschnittlich 2 Funktionen lassen sich nicht sinnvoll nutzen (z. B. Multiplikation und Addition). Die Anlaufzeit der Pipeline ("vector start up overhead") hat maßgeblichen Einfluß auf die Leistung, und zwar um so mehr, je kürzer die Vektoren sind. So stehen dem Nutzer oft nur 5...15% der Maximalleistung zur Verfügung.[2]

2.3.3. Parallelverarbeitung

An Problemen der Parallelverarbeitung wird seit mehr als 2Ø Jahren gearbeitet. Die verschiedenen Ansätze lassen sich unterscheiden:

- nach der Ausgestaltung der einzelnen Verarbeitungseinheiten (Struktur, Umfang, Leistungsvermögen)
- nach der Größenordnung der Anzahl an zusammenwirkenden Verarbeitungseinheiten
- nach den Verbindungsprinzipien (z. B. Bussysteme, Gitterstrukturen mit Direktverbindungen zu den benachbarten Moduln, gemeinsame Speicher)
- nach den Steuerprinzipien des gesamten Systems (bekannteste Unterscheidung: SIMD und MIMD).

Dem Gegenstand der Arbeit gemäß ist von Bedeutung, wie leistungsfähig bzw. aufwendig die einzelne Verarbeitungseinrichtung gestaltet ist.
Lohnt es sich weiterhin, die Leistung des Einzelprozessors zu verbessern, oder sind Forderungen nach extremen Verarbeitungsleistungen eher durch Anordnungen aus einer sehr großen Anzahl vergleichsweise einfacher Verarbeitungseinrichtungen zu erfüllen?
Der zuletzt genannte Ansatz ist vorrangig in 2 Richtungen untersucht worden:

1. SIMD- Anordnungen aus einer sehr großen Zahl (4k...64k) von äußerste elementaren Verarbeitungseinrichtungen, die in den meisten Fällen bitseriell arbeiten

2. Anordnungen, die aus einer großen Zahl (128...4k) zuhandener ("off the shelf"-) Mikroprozessoren aufgebaut sind.

---

1 Beispiele sind die kommerziell verfügbaren Supercomputer und Vektorprozessoren verschiedener Hersteller (vgl. u. a.: /137/, /163/, /192/, /256/, /29Ø/).
2 Die Darstellung folgt /212/; vgl auch Tafel 4 (S. 5Ø).

Die Massenanwendung beider Ansätze ist bisher ausgeblieben, und es gibt Gründe dafür, daß sich diese Entwicklungsrichtungen auch in Zukunft nicht durchsetzen werden, zumindest was den Bereich der Universalrechner angeht.[1]
Die Ausführungszeit vieler elementarer Algorithmen (dazu gehört z. B. die Addition) läßt sich durch Parallelisierung in n Verarbeitungseinrichtungen günstigstenfalls in der Größenordnung O(log n) verringern.[2] Selbst wenn es gelingt, einen komplexen Algorithmus zu parallelisieren, wird die Gesamtlaufzeit durch die vergleichsweise langsame Bearbeitung der Teilalgorithmen bestimmt. Zudem ist eine Vielzahl von Synchronisations- und Kommunikationsabläufen notwendig, die die Ausführungszeit weiter verlängern. Sollte auch das alles beherrscht werden: die Hardware wird aus sehr vielen Komponenten bestehen; das bringt elementare technische Schwierigkeiten mit sich (Zuverlässigkeit, Strombedarf, Kühlung, Fehlersuche usw.). Der Vorteil zuhandener Schaltmittel[3] verschwindet also, wenn sehr viele davon einzusetzen sind; auch wird die einfache betriebswirtschaftliche Rechnung zeigen, daß - bei Fertigung in größeren Stückzahlen - die Entwicklungsaufwendungen für leistungsfähigere Hardwarekomplexe bei weitem aufgewogen werden durch die geringere Anzahl der Funktionseinheiten, die für ein gewünschtes Leistungsvermögen vorgesehen werden müssen.

Im Gegensatz dazu sind Parallelverarbeitungssysteme mit einer mittleren Anzahl an Verarbeitungseinrichtungen sowohl von der technischen als auch von der algorithmischen Seite aus wesentlich besser beherrschbar. Es hat sich gezeigt, daß die Schwierigkeiten der Programmierung bisher überschätzt und die Vielfalt der Anwendungsmöglichkeiten bisher unterschätzt wurden. Für manche Probleme hat es sich erwiesen, daß es einfacher ist, eine parallele Formulierung zu finden als eine sequentielle. Tafel 2 gibt einen Überblick über die Entwicklung solcher Parallelverarbeitungssysteme in den nächsten Jahren.[4]

---

1 Die Darstellung folgt /111/. Für einen Überblick s. u. a. /3/, /8/, /58/, /201/. Alle maßgeblichen Autoren verweisen darauf, daß Versuche mit voll ausgebauten Maschinen notwendig sind, um die Grundfragen entscheiden zu können (zu den Aufwendungen vgl. etwa /117/, /275/).

2 Dazu gibt es viele Untersuchungen. Für arithmetische Operationen vgl. etwa /91/. Anregung: für Elementaroperationen dürfte sich die Parallelisierbarkeit anhand der Booleschen Gleichungen, die die Informationswandlungen beschreiben, exakt untersuchen lassen (Ausbau des Ansatzes von /243/).

3 Die üblichen Mikroprozessoren sind nicht ausdrücklich für Parallelverarbeitungssysteme entworfen worden. Für einen solchen Einsatzfall sind andere Auslegungen erforderlich (spezifische Kompromisse bei der Nutzung der Siliziumfläche für Verarbeitungs-, Speicher-, Speicheranschluß- und Kommunikationshardware). Ein bekanntes Beispiel für einen auf Einsatz in Parallelverarbeitungssystemen optimierten Schaltkreis ist der "Transputer" T 800 (/162/).

4 Die Darstellung (einschließlich Tafel 2) folgt /111/. Beispiele für solche Systeme: Suprenum, GF 11, RP 3.

Generation	1. 1983...87	2. 1988...92	3. 1993...97
T y p i s c h e r K n o t e n			
Leistung in MIPS	1	1Ø	1ØØ
MFLOPS skalar	Ø,1	2	4Ø
MFLOPS vektoriell	1Ø	4Ø	2ØØ
Speicher (MBytes)	Ø,5	4	32
T y p i s c h e s S y s t e m			
Knoten	64	256	1Ø24
Leistung in MIPS	64	256Ø	1ØØ K
MFLOPS skalar	6,4	512	4Ø K
MFLOPS vektoriell	64Ø	1Ø K	2ØØ K
Speicher (MBytes)	32	1Ø24	32 K
Latenzzeit für Nachricht aus 1ØØ Bytes			
a) zum Nachbarn (µs)	2ØØØ	5	Ø,5
b) nicht lokal (µs)	6ØØØ	5	Ø,5

Tafel 2 Übersicht: Parallelverarbeitungssysteme für mittleren Parallelisierungsgrad ("medium grain")

2.3.4. Datenflußmaschinen

Die Prinzipien der Datenflußsteuerung sind bisher vorwiegend theoretisch bzw. in Form von Versuchsmaschinen bearbeitet worden. Neuerdings werden solche Prinzipien in Mikroprozessoren zur Signalverarbeitung angewendet.[1] Seit längerem ist bekannt, einen Befehlsstrom zur Laufzeit in Angaben zur Datenflußsteueung umzuschlüsseln, um mehrere Verarbeitungswerke parallel betreiben zu können.[2] Auch sind gewisse Prinzipien der Datenflußsteuerung in Maschinen mit langen Befehlsworten (VLIW) angewendet worden.[3]
En Grund für das Ausbleiben von Erfolgen in großem Maßstab besteht darin, daß von Neumann- Rechner sequentielle Programme recht effizient abarbeiten können und dazu wesentlich weniger Speicherbandbreite brauchen als vergleichbare Datenflußmaschinen.[4]

2.3.5. Datenstruktur- bzw. Spezialmaschinen

Ehe digitale Universalrechner kostengünstig verfügbar waren, war es geradezu selbstverständlich, für jede Klasse von Aufgaben der Informationsverarbeitung (im weitesten Sinne) spezifische Einrichtungen zu entwerfen. Dazu wurden zuhandene technische Mittel der vielfältigsten Art (mechanische, hydraulische, elektrische usw.) zweckgerichtet kombiniert.
Heutzutage denkt man eher an ein Programm für einen Universalrechner als an eine zweckgebundene technische Sonderlösung.
Um so überraschender sind die Ergebnisse, wenn man den altbewährten Ansatz auf die modernen technologischen Grundlagen überträgt - schließlich hat man mit Gattern, Flipflops und RAMs noch weit mehr Freizügigkeit als mit Schaltwalzen, Zahnrädern und Hydraulikzylindern -: ist ein einzelner, genau abgegrenzter Komplex von Algorithmen und Datenstrukturen gegeben, so bereitet es oft keine grundsätzlichen Schwierigkeiten, dafür spezielle Schaltungsanordnungen auszuarbeiten, die ohne weiteres technisch realisierbar sind und jeden Universalrechner im Leistungsvermögen übertreffen. Die Bezeichnung "Datenstrukturmaschine" soll genau dies zum Ausdruck bringen: die zweckgerechte Nutzung von Schaltmitteln, um bestimmte Operationen über bestimmte Datenstrukturen auszuführen.[5]
Probeentwürfe haben gezeigt, daß Beschleunigungsfaktoren von 10Ø...>2ØØØ gegenüber technologisch vergleichbaren Universalrechnern mittlerer Leistung ohne weiteres mit üblichen Schaltmitteln (TTL, CMOS) und beherrschbaren Aufwendungen (1ØØ...4ØØØ Schaltkreise) zu erreichen sind.[6]

---

1 Beispiel: NEC µPD 7281 (/3Ø6/).
2 Z. B. verwirklicht in 36Ø/91 (/76/).
3 Z. B. die "directed dataflow architecture" (/284/).
4 Die Darlegung folgt /1Ø8/; neuere Ergebnisse, die Verbesserungen bringen sollen (/271/), sind noch nicht zugänglich.
5 Die Bezeichnung "Datenstrukturmaschine" geht auf Giloi zurück (/61/, /176/).
6 Die einschlägige Erfahrungen sind in /243/ zusammengefaßt.

Die Erfahrung wird von anderen Autoren bestätigt. So wird in /176/ angemerkt, daß Datenstrukturarchitekturen den Datenflußarchitekturen an Effizienz und Kosteneffektivität deutlich überlegen sind, da die explizite Parallelität in den Algorithmen viel wirksamer technisch genutzt werden kann.

Es ist deshalb naheliegend, für anwendungspraktisch bedeutsame Algorithmenkomplexe solche Maschinen zu entwerfen und diese mit einem Universalrechner zusammenzuschalten, beispielsweise durch Anschluß an ein universelles Bussystem.

Es hat sich aber gezeigt, daß die Transporte zwischen den einzelnen Verarbeitungseinrichtungen die Leistung begrenzen, wenn die wesentlichen Verarbeitungsalgorithmen erst einmal beschleunigt sind.[1] Man stelle sich beispielsweise vor, es seien Relationen zu durchmustern (in einem speziellen Datenbasisprozessor), und die gefundenen Tupel enthielten numerische Angaben zur Verarbeitung in einem Numerikprozessor. Dann sind fortlaufend Transporte zwischen den beiden Spezialprozessoren notwendig, die zudem vom Universalrechner (d. h. vom übergeordneten Anwenderprogramm) gesteuert werden müssen.
Somit ist zu prüfen, ob eine Sammlung von Spezialmaschinen in Verbindung mit Universalrechnern mittleren Leistungsvermögens grundsätzlich besser ist als ein universeller Hochleistungsrechner, namentlich dann, wenn Operationen über umfangreiche, komplizierte und vielfältige Datenstrukturen im Verbund auszuführen sind.
Dabei darf nicht nur die Geschwindigkeit gesehen werden: 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. Die Schnittstellen zur speziellen Hardware werden von den eingeführten Programmiersprachen nur unvollkommen unterstützt (Einfügen von Maschinencode, Assembler- Unterprogramme). Bei Wechsel der Hardware- Konfiguration sind diese recht aufwendigen Arbeiten stets von neuem erforderlich.[2]

Daraus ergibt sich, daß es zweckmäßig ist, zunächst das Leistungsvermögen von Universalrechnern weiter zu verbessern, und es erweist sich als naheliegend, dafür nach Schaltungslösungen zu suchen, die bei universeller Nutzbarkeit ähnlich leistungsfähig sind wie Datenstrukturmaschinen für wichtige Anwendungsgebiete.

---

1 Das ist in /243/ mit vielen Zahlenbeispielen belegt.
2 Es geht hier grundsätzlich um die Zukunftssicherheit der Spezialhardware, um die Übertragbarkeit der Anwendungsprogrammkomplexe auf Nachfolgesysteme. Dieser Gesichtspunkt wird z. B. in /31Ø/ kritisch gewertet.

2.4. Die eigene Arbeitsrichtung

2.4.1. Vorhaben und Methodik

Der Universalrechner, dessen Prinzipien auf Babbage, Zuse und v. Neumann zurückgehen, konnte bisher nicht von anderen informationsverarbeitenden Einrichtungen verdrängt werden. Er wird noch viele Jahre seine überragende Bedeutung behalten: grundsätzlich andere Lösungen, die ihn ersetzen können, sind in absehbarer Zeit nicht zu erwarten, und bereits die derzeitigen Investitionen in Anwendungslösungen sind so hoch, daß die weitere massenhafte Nutzung einfach eine Folge wirtschaftlicher Zwänge ist.[1]
Die Architektur von Universalrechnern ist weiterhin Gegenstand umfassender Forschungsarbeiten, die sowohl den Einzelprozessor als auch Parallelverarbeitungssysteme aller Art betreffen.
Wesentliche Triebkräfte dafür bestehen in den Forderungen der Anwender nach Leistungssteigerung und Kostensenkung, in der Verfügbarkeit leistungsfähiger Technologien, in der Nutzbarkeit einer umfassenden Erfahrungsbasis und darin, daß auf Grund der wirtschaftlichen Bedeutung Mittel für Forschung und Entwicklung vergleichsweise problemlos bereitgestellt werden.[2]
Es ist nach wie vor entscheidend, die Verarbeitungsgeschwindigkeit zu erhöhen. Rechner, bei denen auf Kosten der reinen Geschwindigkeit der Befehlsausführung ("low level performance") andere Gebrauchseigenschaften (Unterstützung bestimmter Programmiersprachen, gute Compiler, Zuverlässigkeit usw.) vorrangig weiterentwickelt wurden, konnten sich, Sonderanwendungen ausgenommen, nicht durchsetzen. Auch künftig werden solche Eigenschaften kein Ersatz für unzulängliche Verarbeitungsgeschwindigkeit sein.[3]
Die eigene Arbeitsrichtung betrifft daher folgendes Ziel: leistungsentscheidende Wirkprinzipien und Strukturen aufzufinden und zu studieren, sowie Richtlinien, Aufgabenstellungen und Bewertungskriterien für die Gestaltung leistungsoptimierter Universalrechner anzugeben. Die Vorgehensweise ist eine gleichsam experimentelle[4]: durch konstruktives Handeln, durch Ausarbeiten neuer Vorschläge soll versucht werden, bedeutsame Verbesserungen zu erreichen.
Dabei ist es ein wichtiges methodisches Prinzip, den Fragen der "Kompatibilität" auf der Ebene der Maschinenbefehle nicht jenen Rang einzuräumen, den sie derzeit in der Praxis innehaben. Nur so ist herauszufinden, welche absoluten Verbesserungen noch zu erwarten sind. Sind solche Verbesserungen hinreichend bedeutsam, beispielsweise eine Verzehnfachung des Durch-

---

1 Derzeit sind z. B. weltweit 20 Millionen Personalcomputer im Einsatz, und es wird abgeschätzt, daß der Markt noch weitere 80 Millionen Stück aufnehmen kann.
2 Dongarra nennt im Vorwort zu /9/ folgende Sachverhalte, die neue Architekturentwicklungen anregen: Verfügbarkeit von leistungsfähigen Mikroprozessoren, standardisierten Bussystemen, gate- array- Technologien und von Risikokapital.
3 Die Darstellung folgt /146/; sie wird durch die Verkaufsstatistiken, Geschäftsberichte usw. bestätigt.
4 Vgl. Abschnitt 2.1. (S. 7).

satzes im Rahmen bestimmter Aufwendungen, so muß man darüber nachdenken, sie in die Praxis einzuführen; sind sie es nicht, so ist es recht wahrscheinlich, daß der Stand der Technik einem Optimum bereits sehr nahe kommt, so daß die nächsten Ziele von Forschung und Entwicklung eher in Verfeinerungen, Verbesserungen der Technologie usw. zu sehen sind. Mit diesen Überlegungen läßt sich der Gegenstand der Arbeit folgendermaßen eingrenzen:

1. Es geht um den einzelnen programmierbaren Universalrechner. Sein Anwendungsgebiet sind Algorithmen aller Art, also alle Aufgabe der Informationsverarbeitung, die sich letztlich auf berechenbare Funktionen zurückführen lassen, vorzugsweise auf Operationen über binär codierte numerische und nichtnumerische Informationsstrukturen unter Einbeziehung des logischen Schließens.

2. Es sollen Überlegungen zu technisch- ökonomisch leistungsoptimalen Universalrechnern angestellt werden; Ziel ist das höchste Leistungsvermögen in der jeweiligen Größenklasse (vom Mikrocontroller bis zum Supercomputer).

3. Hier werden - ohne Einschränkung der Allgemeinheit - jene Größenordnungen bevorzugt betrachtet (aus naheliegenden Gründen der Nutzbarkeit), die für die Breitenanwendung in Frage kommen: vom Mikrocontroller bis zum Hochleistungs- OEM- Rechner, der beispielsweise als Maschinensteuerung, als "workstation", aber auch als Verarbeitungsmodul in Supercomputern[1] nutzbar ist.

4. Dem Vorhaben liegt das Prinzip der Ingenieurarbeit zugrunde, Bewährtes und Neues im Hinblick auf ein optimales Ergebnis zu vereinen und zuhandene technische Mittel für einen bestimmten Zweck in bestmöglicher Weise zu nutzen. Als zuhanden werden angesehen[2]:

- die Prinzipien der binären Informationsverarbeitung als technischer Umsetzung der Aussagenlogik
- die Technik der integrierten Schaltkreise bis hin zur "wafer scale integration" (WSI)
- der Stand der Technik hinsichtlich der konstruktiven Gestaltung von Digitalrechnern (mechanischer Aufbau, Kühlung, periphere Einrichtungen usw.)
- die bekannten Verfahren, Prinzipien und Algorithmen der numerischen und nichtnumerischen Informationsverarbeitung.

Das heißt, von den Grundlagen her wird die bisher bewährte Hauptrichtung der Entwicklung weiter verfolgt: keine optische Informationsverarbeitung, keine Wahrscheinlichkeitslogik, keine Schwellwertelemente, keine neuronalen Strukturen.

---

1 Das betrifft Supercomputer, die als Parallelverarbeitungssysteme aus vergleichsweise leistungsfähigen Moduln aufgebaut sind, also Strukturen ähnlich Suprenum, RP 3 u. a.
2 Mit Blick auf die Zeit der Einführung und Nutzung.

Die Arbeit gilt ausschließlich dem einzelnen Rechner, dessen struktureller Vervollkommnung. Fragen der Parallelverarbeitung und der Nutzung extrem leistungsfähiger Technologien werden damit keineswegs gegenstandslos; aus der Sicht der vorliegenden Arbeit sind sie vielmehr Gegenstand künftiger Aktivitäten: erst wird der Einzelprozessor strukturell auf höchstes Leistungsvermögen gebracht, bevor man sich damit befaßt, eine Vielzahl davon zusammenzuschalten[1] oder sehr kostenaufwendige Technologien einzusetzen. Das allgemeine Ziel soll durch folgenden methodischen Ansatz erreicht werden:

1. Für bedeutsame Anwendungsgebiete der Rechentechnik werden grundsätzliche und leistungsbestimmende Abstraktionen (Datentypen + Operationen + Ablaufprinzipien) gesucht. Diese werden exakt beschrieben.

2. Es werden Hardwarestrukturen entwickelt, um diese Abstraktionen so effektiv wie möglich implementieren zu können.

3. Auf Grundlage dieser Strukturen werden Rechnerarchitekturen definiert, die die erforderliche Universalität gewährleisten.

4. Der Auswahl der Abstraktionen und der Ausgestaltung der Schaltungslösungen werden die Tiefenstrukturen der jeweiligen Prozesse der Informationsverarbeitung zugrunde gelegt und nicht die Oberflächenstrukturen, wie sie durch konkrete Programmiersprachen, Betriebssystem- Umgebungen, Standards für die Datendarstellung usw. gegeben sind. Es wird also versucht, das Wesen der Verarbeitungsprozesse zu erfassen und die technischen Lösungen hinreichend allgemeingültig auszulegen, das heißt eine einseitige Orientierung, z. B. auf ein bestimmtes Sprachkonzept, zu vermeiden.

5. Für jede Abstraktion wird versucht, die Schaltmittel so leistungsfähig wie möglich auszulegen. Dafür wird - im Rahmen der technischen Beherrschbarkeit - Hardware ohne Rücksicht auf übliche Gepflogenheiten oder Konventionen eingesetzt; sowohl hinsichtlich der Aufwendungen als auch der strukturellen Gestaltung.[2]

6. Allen technischen Lösungsvorschlägen, vom einzelnen Schaltungskomplex bis zur Architekturdefinition, wird der Stand der Technologie zugrunde gelegt, der zu Beginn ihrer Nutzung erwartet werden kann. Bis zur Einführung einer neuen Architektur sind etwa 4 Jahre zu rechnen[3]; erfolgreiche Architekturen haben eine Nutzungsdauer von mehr als 2Ø Jahren. Derzeit werden auf Schaltkreisen von etwa 1Ø*15 mm^2 mit über $1Ø^6$

---

1 Für die umfassende Bearbeitung dieses Problems sind unbedingt Ergebnisse der laufenden (sehr aufwendigen) Versuche (Suprenum, GF 11, RP 3 u. a.) auszuwerten.

2 Die meisten der eingeführten Architekturkonzepte wurden unter der Bedingung grundsätzlich knapper Hardware- Ressourcen (Speicher, Verarbeitungswerke usw.) ausgearbeitet.

3 Vgl. einschlägige Erfahrungsberichte, z. B. in /4/, /7/, /19/, /113/, /15Ø/, /2ØØ/.

Transistoren sehr leistungsfähige Einzelprozessoren verwirklicht.[1] Die Verarbeitungsleistung wird vor allem deshalb erreicht, weil ein ingenieurmäßig sinnvoller Kompromiß bekannter Konzepte (Cache, RISC- Prinzipien, parallele Verarbeitungswerke) auf einem einzigen Schaltkreis untergebracht werden kann, wodurch sehr kurze Zykluszeiten realisierbar sind (Taktfrequenzen um 40 MHz). Will man diesen Stand der Technik übertreffen, sind neue Konzepte notwendig, die von Grund auf die Möglichkeiten der Technologie nutzen, um dem Anwender überlegene Leistungen[2] bereitzustellen (Realisierungsbasis: Anordnungen aus mehreren derart hochintegrierten Schaltkreisen[3], WSI- Technologien). Wenn man Forschungen zu neuen Architekturen nicht nur des Erkenntnisgewinns willen betreibt, sondern auf Erfolge prätendiert, ist grundsätzlich zu vermeiden, daß zeitweilige technologische Einschränkungen Beschränkungen bei den Architekturkonzepten zur Folge haben.[4]

Allein der Umfang des Vorhabens schließt eine umfassende Behandlung in der vorliegenden Arbeit aus. Hier geht es vielmehr darum, die Grundlagen zu erörtern, Bewertungskriterien aufzustellen und Methoden vorzuschlagen. Das wird beispielhaft vorgeführt, indem verbreitete und bekannte Maschinenarchitekturen und Programmiersprachen als Erfahrungsbasis für das Auffinden von Abstraktionen verwendet werden.

Rechnerarchitekturen und Schaltungslösungen auf dieser Grundlage werden noch der weiteren Vervollkommnung bedürfen; sie werden aber aufzeigen, in welcher Größenordnung die Leistungsvorteile liegen können, die von künftigen optimierten Rechnerstrukturen zu erwarten sind.

---

1 Vgl. Intel 80860 (/325/, /327/), 80486 (/328/), Motorola 68040 (/329/).

2 Als weiteres Ziel verbleibt die Verbesserung des Preis-Leistungs- Verhältnisses, wo Erfolge auch mit deutlich geringeren Anforderungen an die Technologie demonstriert werden können (z. B. bei Mikrocontrollern).

3 Das erfordert sicherlich neue Vorstellungen zur Funktionsaufteilung, die über den Stand der Technik (vgl. etwa /105/) hinausgehen.

4 Für die ersten Hardwaremodelle sind dann eher Mehraufwendungen (z. B. Realisierung mit mehreren Standardzellen-Schaltkreisen) und gewisse Leistungsverluste (durch niedrigere Taktfrequenzen; ggf. auch durch mikroprogrammtechnische Emulation) in Kauf zu nehmen. Der Nutzer wird eine neue Architektur nur dann akzeptieren, wenn die Umstellungsaufwendungen nicht ins Gewicht fallen oder durch perspektivische Aussichten (Leistungsverbesserung, Zukunftssicherheit, künftige weite Verbreitung) gerechtfertigt werden. Vergleichsweise geringfügige Verbesserungen, die Umstellungen erfordern, haben gegenüber dem Stand der Technik kaum eine Chance.

2.4.2. Einordnung in den Stand von Wissenschaft und Technik

Die vorgeschlagene Arbeitsrichtung geht von folgenden Ansätzen aus:

1. Die Optimierung der Maschinenbefehle. Der RISC- Ansatz ist in Betracht zu ziehen, soll aber durch andere Überlegungen, beispielsweise zur bestmöglichen Ausnutzung von Datenwegen, ergänzt werden. Die Übergänge zwischen RISC und Konzepten der Mikroprogrammsteuerung sind fließend und sollen näher untersucht werden (RISC- Befehle lassen sich als "vertikale" Mikrobefehle auffassen, die im allgemeinen Speicheradressenraum abgelegt sind).
Auch sind Konzepte zur schaltungstechnischen Unterstützung höherer Programmiersprachen (HLL- Architekturen) von neuem zu durchdenken (Nutzung der zwischenzeitlich gewonnenen Erfahrungen und der Möglichkeiten der modernen Schaltungstechnik).
Wesentlich ist, daß die Untersuchung nicht vordergründig auf statistische Analysen gegebener Programme gestützt wird, sondern auf eine weitgehend analytische Betrachtung der leistungsbestimmenden Sachverhalte.[1]

2. Die Nutzung des innewohnenden Parallelismus in üblichen Programmen. Die Konzepte der Anordnung und Steuerung mehrerer Operationswerke ("scoreboard"- Prinzip, VLIW, "horizontale" Mikrobefehle, Datenflußprinzipien) sind dafür in Betracht zu ziehen.

3. Der Entwurf von Spezialmaschinen. Es ist wirtschaftlich nicht durchführbar, für jeden anwendungspraktisch bedeutsamen Algorithmenkomplex eine Sondermaschine bereitzustellen, die leistungsmäßige Überlegenheit zweckgerichtet entworfener Sonderschaltungen ist aber beeindruckend: es lohnt sich also zu untersuchen, wie solche Schaltmittel in die Struktur eines Universalrechners eingefügt werden können.

4. Die gleichsam ganzheitliche Betrachtungsweise, die verschiedene Ebenen im Zusammenhang untersucht: von der Anweisung in einer höheren Programmiersprache über die compilierte Befehlsfolge bis zum Ablauf in der Hardware. Dieses Vorgehen wird beispielsweise in /233/ als notwendig angesehen, um das Leistungsvermögen von Rechnerstrukturen deutlich zu verbessern. Hier wird zunächst auf eine Erfahrungsbasis Bezug genommen, die durch eingeführte Programmiersprachen, Maschinenarchitekturen und typische Schaltungslösungen gegeben ist.

---

1 Auf Basis von Messungen und statistischen Auswertungen allein können lediglich bestehende Konzepte in sich optimiert, aber keine neuen gefunden werden (abgesehen davon, daß aus solchen Ergebnissen Ziele für erfinderisches Handeln ersichtlich bzw. ableitbar sind).

3. Das elementare Modell der sequentiellen Informationsverarbeitung

3.1. Von der Verarbeitungsschaltung zum Universalrechner

Im folgenden geht es ausschließlich um die Verarbeitung binär codierter Information, die in Speichermitteln mit wahlfreiem Zugriff gespeichert ist. Dabei entstehen binär codierte Ergebnisse; diese werden ebenfalls in Speichermitteln mit wahlfreiem Zugriff abgelegt. (Der Stand der Technik erlaubt es, Fragen der Ein- und Ausgabe als grundsätzlich lösbar anzusehen, beispielsweise durch Anschluß der Speichermittel an ein universelles Bussystem, so daß übliche Mikrorechner- Baugruppen für die E/A- Funktionen nutzbar sind.)
Die Ergebnisse werden stets durch Anwendung von Wandlungsvorschriften mit eindeutiger Wirkung in diskreten Zeitschritten (Taktzyklen) gebildet; aus gleichen Angaben entstehen durch Anwendung gleicher Wandlungsvorschriften stets gleiche Resultate. Jede solche Wandlungsvorschrift heißt im folgenden Algorithmus.[1] Ein Algorithmus kann beispielsweise implementiert werden durch einen Programmkomplex, ein einzelnes Programm, ein Unterprogramm, einen Maschinenbefehl, eine Schaltungsanordnung usw. Das Ziel besteht darin, universell nutzbare und überdurchschnittlich leistungsfähige Schaltungsanordnungen im Zusammenhang mit den ihnen zugeordneten Informationsstrukturen zweckgerichtet auszubilden. Deshalb werden hier lediglich Beziehungen zwischen Algorithmen und technischen Mitteln zu deren Implementierung untersucht; das Gebiet der Programmierung bleibt außer Betracht. Um Klarheit über grundlegende technische Sachverhalte und Zusammenhänge zu gewinnen, soll, mit dem einfachsten Fall beginnend, beschrieben werden, wie einzelne Verarbeitungsschaltungen systematisch in Universalrechnerstrukturen überführt werden können. Der einfachste Fall betrifft die Implementierung eines einzigen Algorithmus, der aus n Argumenten m Resultate erzeugt.[2] Das grundsätzliche Schema zeigt Bild 4. Es sind folgende technische Mittel vorgesehen:

1. Speichermittel für Argumente und Resultate
2. Verknüpfungsschaltungen
3. Verbindungen zwischen 1. und 2.

Dieses Schema hat keinerlei Erkenntniswert, wenn zugelassen wird, daß die Verknüpfungsschaltungen in beliebiger Weise ausgestaltet sein können. Im folgenden sind Verknüpfungsschaltungen deshalb ausschließlich rückwirkungsfreie kombinatorische Schaltungen bzw. Zuordner, die aus allen Argumenten in einem einzigen Zeitintervall alle Resultate bilden.[3]

---

1 Zur Algorithmentheorie vgl. etwa /46/, /47/, /74/, /274/.
2 In den Bildern ist der Einfachheit halber - ohne Beschränkung der Allgemeinheit - n = 2 und m = 1.
3 Solche Überlegungen sind bereits in /243/ zu finden; allerdings ausdrücklich auf Spezialmaschinen beschränkt. Demgegenüber wird hier - zwecks Kenntnisgewinn - zunächst die bekannte v. Neumann- Struktur angestrebt.

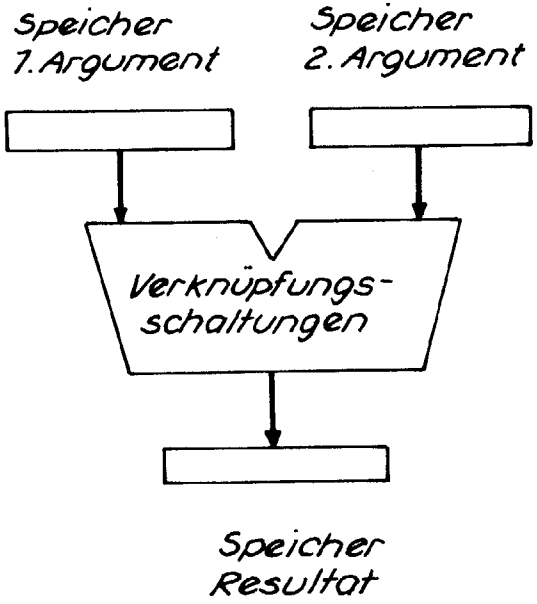


Bild 4

Allgemeines Schema der unmittelbaren Ausführung eines Algorithmus

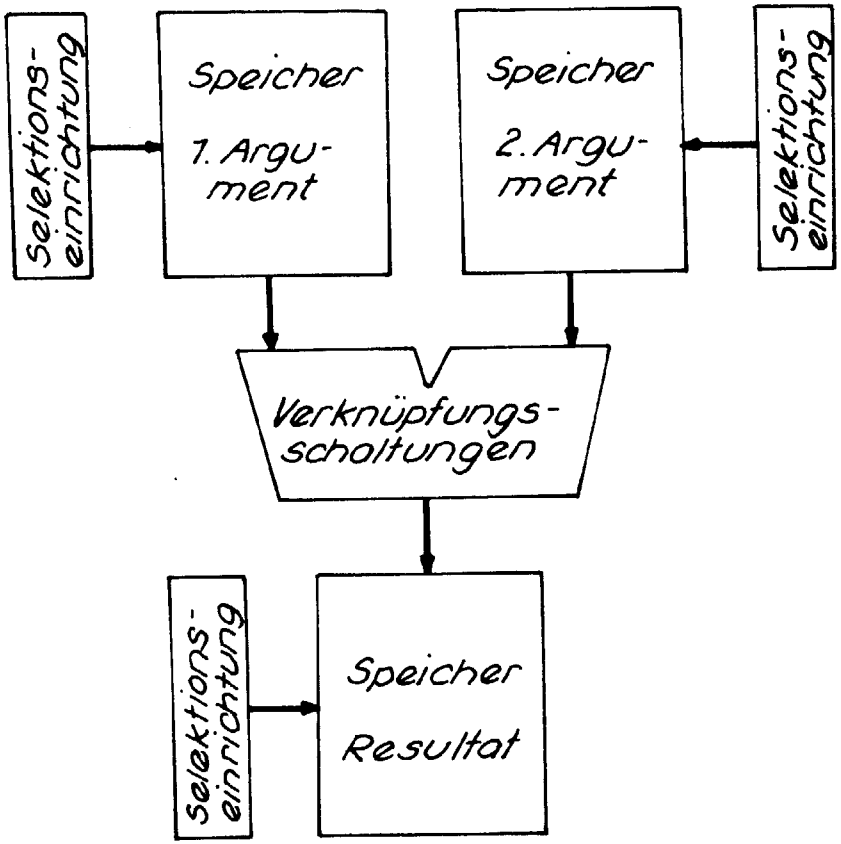


Bild 5

Abschnittsweise unmittelbare Ausführung eines Algorithmus

Dieses Zeitintervall entspricht einem Taktzyklus vom Auslesen der Argumentspeicher bis zum Laden der Resultatspeicher. Eine solche Implementierung eines Algorithmus ist in Bezug auf die Ausführungsgeschwindigkeit die wünschbarste überhaupt[1], und diese Vorstellung wird den weiteren Betrachtungen gleichsam als Richtbild zugrunde gelegt, - es ist aber klar, daß der technischen Durchführbarkeit der unmittelbaren Zuordnung enge Grenzen gesetzt sind.[2]
Algorithmen, die nicht für die unmittelbare Zuordnung geeignet sind, müssen gewissermaßen stückweise in mehreren Taktzyklen ausgeführt werden. Die Informationswandlungen in diesen Zyklen werden hier allgemein als Aktionen bezeichnet. Den Verknüpfungsschaltungen werden in den einzelnen Zyklen Teile der Argumente zugeführt, und es werden Teile der Resultate gebildet. Aktionen, die die jeweils benötigten Abschnitte der Argumente auswählen bzw. Abschnitte von Resultaten entsprechend einordnen, heißen Selektionen; Aktionen, die Resultat- Abschnitte erzeugen, heißen Operationen. Bild 5 zeigt die Modifikation des Schemas von Bild 4. Alle Speichermittel sind zur abschnittsweisen Speicherung der Argumente bzw. Resultate ausgebildet. Sie sind an weitere Schaltmittel (Selektionseinrichtungen) angeschlossen, so daß in jedem Taktzyklus ein Zugriff zu einem Abschnitt in jeder Speichereinrichtung möglich ist. Dieses Schema ist nicht für alle Algorithmen anwendbar, sondern nur für solche, die unter 2 Einschränkungen implementiert werden können:

1. In allen Taktzyklen ist nur eine einzige Operation auszuführen.
2. Es ist nicht notwendig, Teile von Resultaten in Operationen einzubeziehen (keine Rückführung von Resultaten).

Um die erste Einschränkung aufzuheben, müssen die Verknüpfungsschaltungen gemäß Bild 6 so erweitert werden, daß Schaltmittel für mehrere Operationen verfügbar sind. Soll in einem bestimmten Taktzyklus eine bestimmte Operation ausgeführt werden, wird das betreffende Operationswerk aktiviert. Aktionen, die Operationen auswählen, heißen Aktivierungen. In technischer Hinsicht ist dazu notwendig, Auswahlschaltungen für die Resultate vorzusehen und deren Auswahlsteuerleitungen an eine Ablaufsteuerung anzuschließen. Diese besteht im Beispiel aus einem Zähler, der die Nummer des aktuellen Zyklus liefert und aus einem nachgeschalteten Zuordner, der die Auswahlsteuerleitungen erregt.

---

1 Der besagte Zyklus muß natürlich hinreichend kurz sein; aus praktischen Erwägungen heraus sind etwa 2 µs als Obergrenze anzusetzen: damit ist die Zuordnung oft schneller als die sequentielle Verarbeitung, und es ist technisch auch mit umfangreichen Zuordnungsschaltungen problemlos erreichbar (Beispiel: Tabelle der Winkelfunktionen in ROM- oder DRAM-Speichern mit nachgeordnetem Interpolationsnetzwerk).

2 Bei vergleichsweise wenigen Argumentbits spielt die Kompliziertheit keine Rolle (ROM bzw. RAM als Zuordner); ansonsten müssen sich die Verknüpfungen mit Gatternetzwerken verwirklichen lassen (Näheres s. S. 52 ff.).

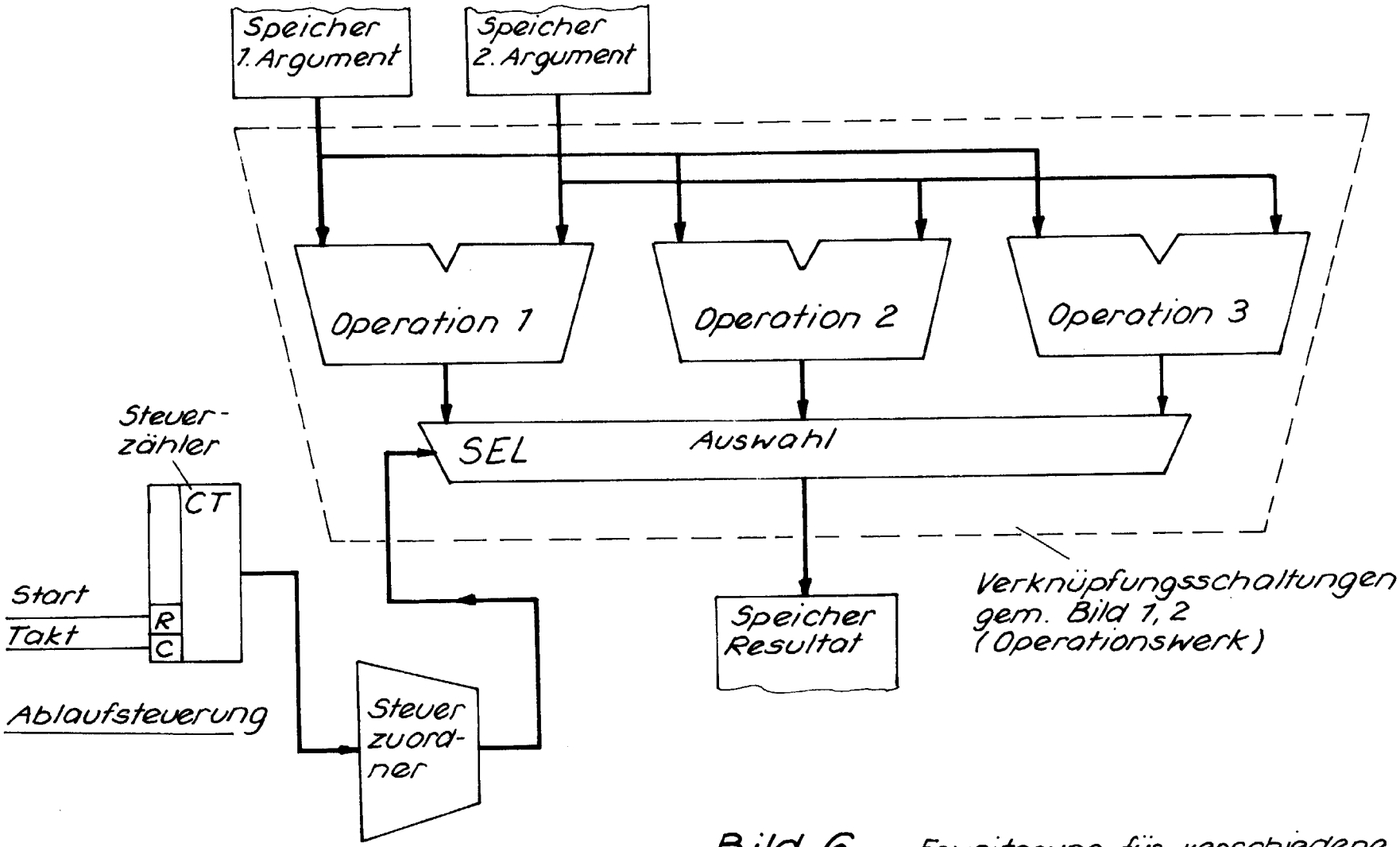


Bild 6 Erweiterung für verschiedene Operationen

Die zweite Einschränkung läßt sich beseitigen durch gemeinsame Speichermittel für alle Angaben des Algorithmus: Übergang zur Registermaschine nach Bild 7. Es werden zusätzliche Taktzyklen benötigt , da die Abschnitte von Argumenten und Resultaten nur nacheinander selektiert werden können; die Register sind notwendig, um die selektierten Abschnitte zu halten. Diese Anordnung ist allerdings nur für Algorithmen geeignet, die nicht erfordern, in bestimmten Zyklen zwischen verschiedenen Selektionen bzw. Operationen zu wählen.[1] Um diese Einschränkung aufzuheben, sind die Steuerschaltungen zu einem Steuerautomaten nach Bild 8 weiterzubilden, der zusätzlich über Bedingungsleitungen an Teile der Ausgänge von Operationswerken und Selektionseinrichtungen angeschlossen ist. Von hier aus erfordert es nur die folgenden Schritte, um zur wirklichen Universalmaschine zu kommen:

1. Anordnung eines ladbaren Steuerspeichers im Steuerautomaten (Bild 9 zeigt ein Ausführungsbeispiel).

2. Gestaltung der Operationswerke und Selektionseinrichtungen derart, daß anstelle der speziellen Verknüpfungen für den jeweiligen Algorithmus elementare, allgemein nutzbare Verknüpfungen[2] für Operationen und Selektionen vorgesehen werden.

3. Anordnung eines einzigen gemeinsamen Speichers für Argumente, Resultate und Steuerinformation (Bild 1Ø). Damit laufen alle Aktionen in mehreren aufeinanderfolgenden Zyklen ab (Lesen der Steuerinformation -> Lesen der Argumente -> Verknüpfung -> Schreiben der Resultate), und es müssen Register vorgesehen werden, um die in den einzelnen Zyklen benötigten Angaben zu halten.

4. Nutzung der Operationswerke, um die Selektionsangaben für Argumente und Resultate zu bestimmen. Statt der Selektionseinrichtungen gibt es lediglich ein Adressenregister, das in jedem Zyklus mit jeweils einer Selektionsangabe (Adresse) geladen werden kann.

5. Weiterbildung der Steuermittel für die beschriebene zyklusweise sequentielle Nutzung der Schaltungsstruktur (Befehlsablaufsteuerung; "Sequencer").

Der Übergang zum klassischen v. Neumann- Rechner ist damit vollzogen. Bild 11 zeigt die Struktur; die Benennung der Schaltmittel ist bereits auf die allgemein übliche Ausdrucksweise umgestellt. Man gelangt also von der zweckgebundenen Schaltung zum Universalrechner, indem die technischen Mittel

---

1 In Abhängigkeit von bestimmten Teilresultaten (Bedingungen).

2 Welche Verknüpfungen unbedingt notwendig sind, ist umfassend untersucht worden; im besonderen haben van der Pohl und Fromme gezeigt, daß ein einzige Verknüpfungsart ausreicht (/74/, /279/, /28Ø/). Gemäß dem Stand der Technik sind solche Minimalprinzipien kaum mehr von Interesse; vielmehr kann eine umfangreiche Erfahrungsbasis im Hinblick auf Zweckmäßigkeit und Nutzungshäufigkeit ausgewertet werden.

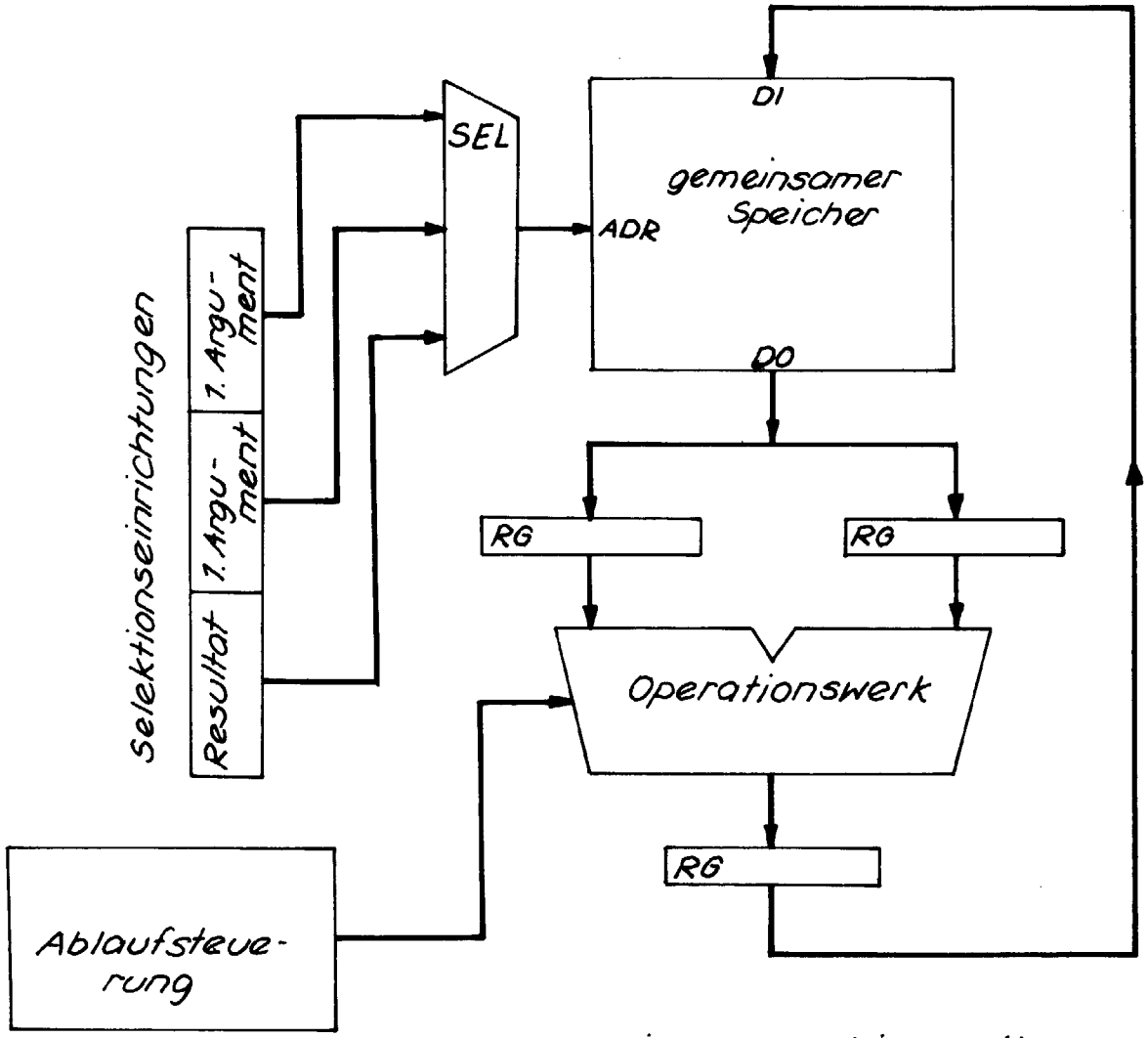


Bild 7

Registermaschine mit gemeinsamen Speicher für Argumente und Resultat

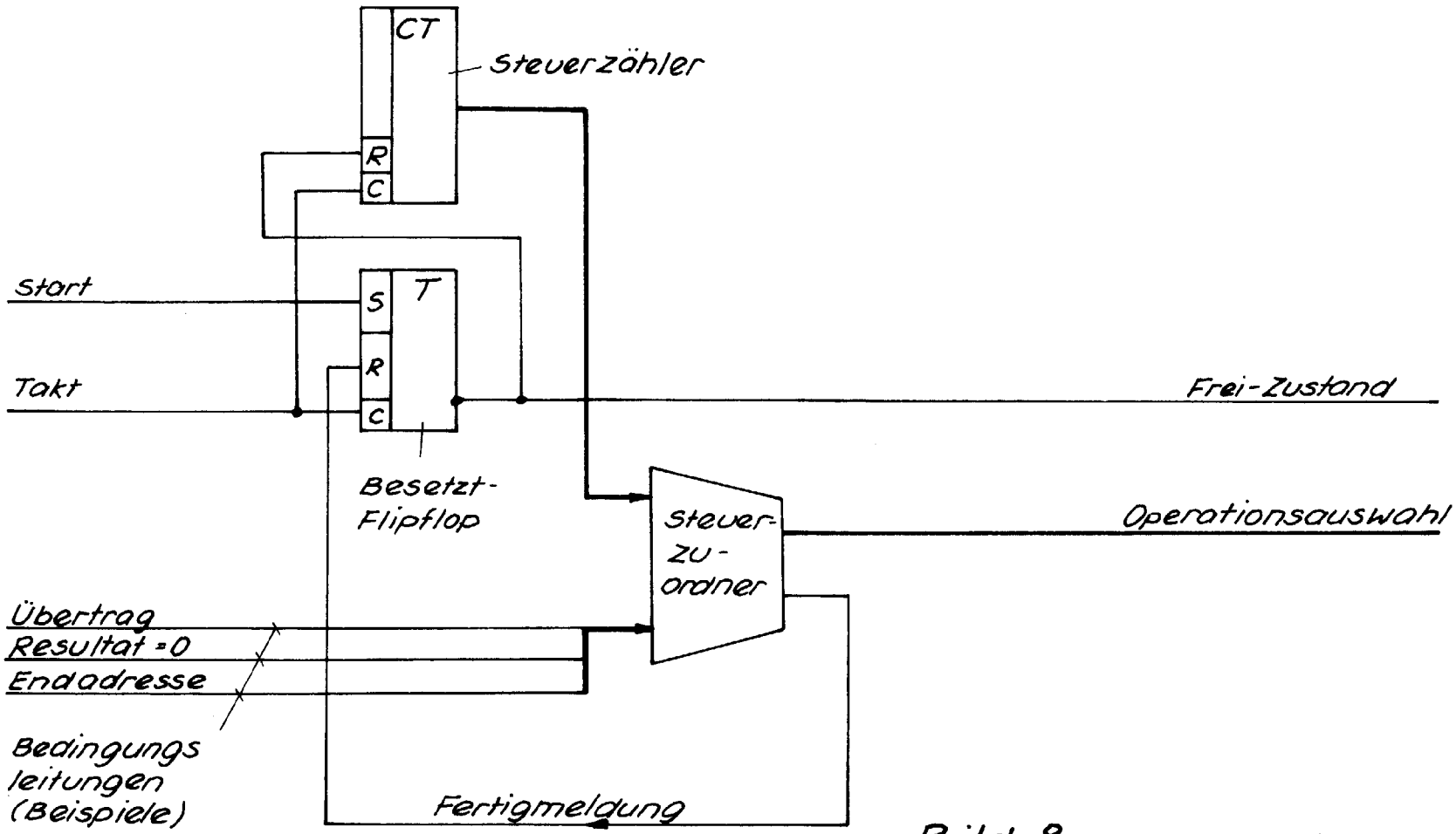


Bild 8 Steuerautomat mit Bedingungsauswertung

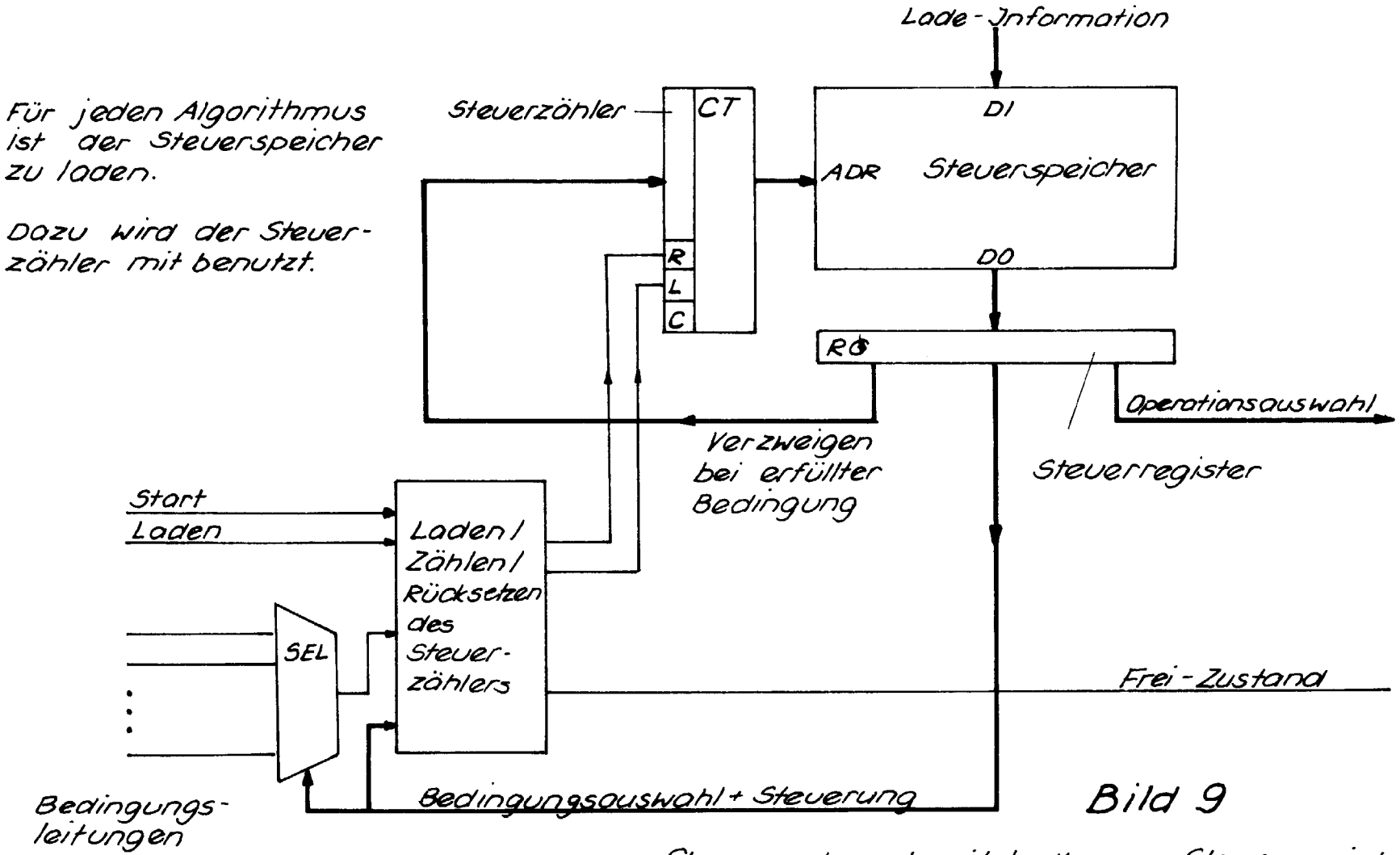


Bild 9

Steueroautomat mit ladbarem Steuerspeicher

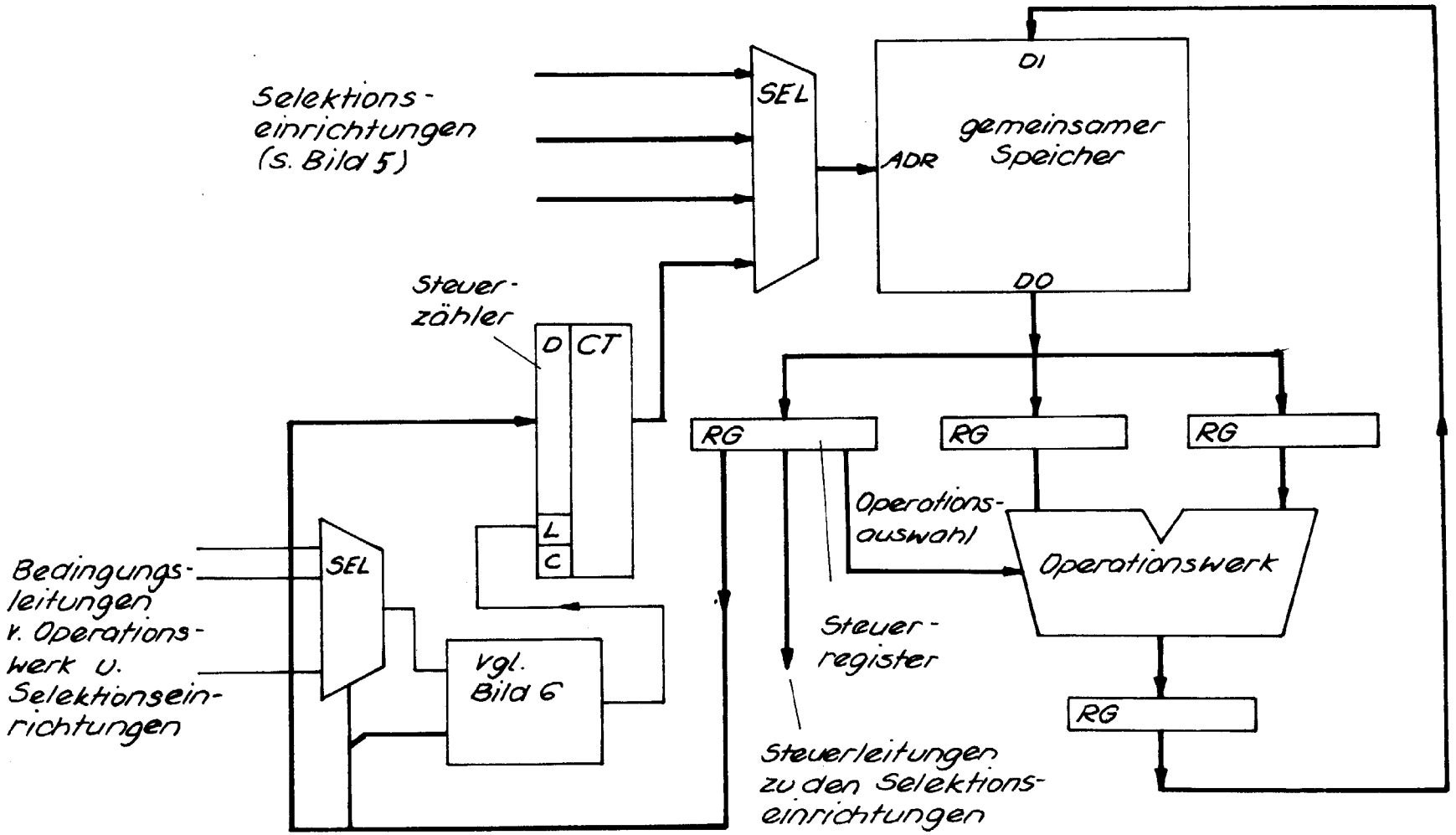
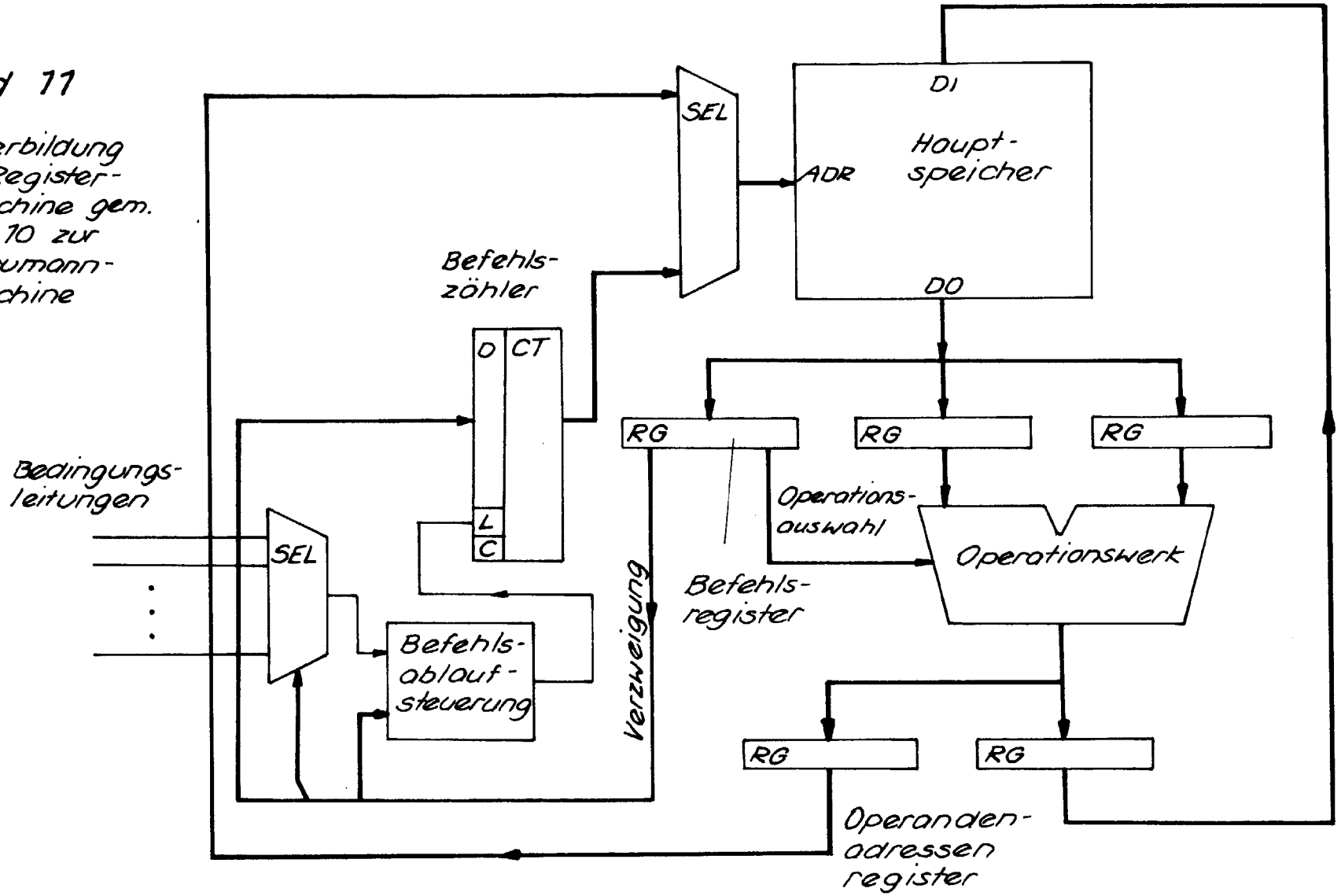


Bild 10 Registermaschine mit gemeinsamen Speicher für Argumente, Resultate und Steuerinformation

Bild 11

Weiterbildung der Registermaschine gem. Bild 10 zur v. Neumann-Maschine



zur Implementierung von Algorithmen systematisch zunehmend universeller ausgebildet werden. Ein solcher Weg zeigt wichtige Sachverhalte auf:

1. Mit zunehmender Universalität der Schaltungsstruktur vermindert sich die Ausführungsgeschwindigkeit des einzelnen Algorithmus, da in leistungsbestimmende Signalwege Auswahlschaltungen eingefügt (vgl. Bild 6) und alle Abläufe auf Folgen elementarer Aktionen zurückgeführt werden müssen.

2. Jede technische Einrichtung bedarf grundlegender Steuermittel. Diese werden mit zunehmender Universalität komplizierter; sie reichen vom einfachen Taktgeber für den Resultatspeicher in Bild 4 bis zur Befehlsablaufsteuerung (Sequencer) der v. Neumann- Maschine gemäß Bild 11.

3.2. Formale Darstellung

Um die Beziehungen zwischen Algorithmen und Schaltungsstrukturen zu untersuchen, wird nachfolgend eine formale Notation eingeführt. Eine Formalisierung kann gegebene Sachverhalte eindeutiger, bestimmter, genauer ausdrücken als eine auf Anschauung beruhende Darstellungsweise; sie kann aber selbst keine neuen Erkenntnisse hervorbringen, also auch beim Ausarbeiten neuartiger Schaltungslösungen und Architekturkonzepte die erfinderische Tätigkeit, die ohne ganzheitliche Anschauung schwer vorstellbar ist, nicht ersetzen. Deshalb wird die Formaldarstellung hier nur soweit ausgebildet, daß sie im heuristischen Sinne nutzbar ist.[1]

3.2.1. Allgemeine Vereinbarungen

Alle Strukturen, die hier betrachtet werden, sind gekennzeichnet durch einen Eigennamen, eine Beschreibung ihres inneren Aufbaus und eine Beschreibung ihrer Bedeutung bzw. Wirkungsweise.[2] Deshalb wird eine beliebige Menge aus n einzelnen Strukturen als Tripel gleichmächtiger Teilmengen definiert:

$$\mathfrak{M} = \left\{ N^{\mathfrak{M}},\ D^{\mathfrak{M}},\ M^{\mathfrak{M}} \right\} \qquad (3.1)$$

---

1 Es sei darauf hingewiesen, daß sich eine algebraische Modellierung zur Verfeinerung und exakteren Fassung des hier skizzierten Ansatzes anbietet. Vgl. S. 2, Punkt 2. Um ein formales System für konkrete Architekturen, Schaltungen usw. wirklich nutzen zu können, ist dessen programmseitige Implementierung unerläßlich.

2 Das ist methodisch durch die von Tarski (/3ØØ/) begründete Vorgehensweise geprägt: es wird untersucht, wie die sprachlichen Mittel zur Beschreibung der in Frage stehenden Sachverhalte beschaffen sein müssen.

Darin bedeuten:

$N^{(m)} = \{N_1^{(m)} \ldots N_p^{(m)}\}$	die Menge der Eigennamen
$D^{(m)} = \{D_1^{(m)} \ldots D_p^{(m)}\}$	die Menge der internen Strukturbeschreibungen
$M^{(m)} = \{M_1^{(m)} \ldots M_p^{(m)}\}$	die Menge der Beschreibungen der Bedeutung bzw. Wirkungsweise.

Die Eigennamen dienen lediglich der eindeutigen Bezeichnung, ihnen kommt sonst keine weitere Bedeutung zu.

Informationsstrukturen sind bis aufs Bit ausdefiniert[1]; Schaltungsstrukturen werden letztlich durch Verbindungen von Speichermitteln und kombinatorischen Schaltungen beschrieben.

Die Bedeutung bzw. Wirkungsweise wird, je nach Zweckmäßigkeit, in einer geeigneten Metasprache ausgedrückt bzw. durch Rückführung auf Abläufe in wohlbekannten Schaltungsanordnungen erklärt.
Als geeignete Metasprache ist beispielsweise die Kombination von umgangssprachlicher, formaler und bildlicher Darstellung anzusehen, in der üblicherweise Rechnerarchitekturen gegenüber dem Nutzer dokumentiert werden.[2] Es sei bemerkt, daß es keineswegs um triviale Systeme geht, so daß pragmatische Gesichtspunkte nicht außer Acht gelassen werden sollten. So hat die Erfahrung gezeigt, daß es unerläßlich ist, komplexe Rechnerarchitekturen in vollem Umfang umgangssprachlich zu beschreiben.[3]
Die Rückführung auf wohlbekannte Schaltungsanordnungen ist ihrem Wesen nach eine Methode der operationalen Semantikdefinition.[4] Die Absicht besteht darin, eine bis auf die einzelne Boolesche Gleichung ausgearbeitete Schaltungsbeschreibung einer solchen Semantikdefinition zugrunde zu legen. Die Schaltungsanordnungen sollen mit Evidenzkriterien verifizierbar sein; sie sollen deshalb die Algorithmen und Datenstrukturen möglichst direkt mit überschaubaren technischen Mitteln[5] widerspiegeln. Beispielsweise wird für die Multiplikation von Binärzahlen eine Anordnung geschichteter Addierwerke vorgesehen (Schaltungsanordnung MULTIPLY). Die Bedeutung eines Ausdruckes C := A * B wird dann so erklärt, daß C jenes Resultat- Bitmuster zugeordnet erhält, das an den Ausgängen von MULTIPLY entsteht, wenn die Werte von A, B an deren Eingängen anliegen (das Resultat ist durch die Booleschen Gleichungen, die MULTIPLY beschreiben, eindeutig bestimmt).

---

1 Diese Absicht wurde bereits von Zuse mit dem Plankalkül verfolgt (/1ØØ/).
2 Als Beispiele können die bekannten Architekturhandbücher angesehen werden, z. B. /36/, /38/.
3 Umfassende Erfahrungen eines führenden Anbieters werden in /332/ vermittelt.
4 Eine Übersicht über verschiedene Möglichkeiten zur formalen Definition von Bedeutungen ist z. B. in /9Ø/ zu finden.
5 Überschaubarkeit geht vor Aufwand!

3.2.2. Algorithmen

Ein Algorithmus $\mathcal{A}$ wird durch ein geordnetes Paar aus Daten $\mathcal{D}$ und einer Wandlungsvorschrift $\mathcal{W}$ dargestellt:

$$\mathcal{A} = \{\mathcal{D}, \mathcal{W}\} \qquad (3.2)$$

Das entspricht direkt der Formulierung eines Algorithmus in einer hinreichend ausgestatteten Programmiersprache.[1]
Die Menge $\mathcal{D}$ der Daten ist in die Menge der Argumente $\mathcal{D}^a$ und in die Menge der Resultate $\mathcal{D}^r$ zerlegbar:

$$\mathcal{D} = \{\mathcal{D}^a, \mathcal{D}^r\}$$

Die Wandlungsvorschrift $\mathcal{W}$ ist gegeben durch eine Ressourcenbeschreibung $\mathcal{R}$ und eine Ablaufbeschreibung $\mathcal{P}$:

$$\mathcal{W} = \{\mathcal{R}, \mathcal{P}\}$$

So wird deutlich, daß es stets notwendig ist, zu jeder Ablaufbeschreibung $\mathcal{P}$ (die man sich z. B. als den Quelltext eines Programms vorstellen kann), die jeweiligen Voraussetzungen mitzudenken: wird Hardware direkt programmiert, ist $\mathcal{R}$ die Beschreibung der Maschinenarchitektur; wird $\mathcal{P}$ in einer höheren Programmiersprache formuliert, ist $\mathcal{R}$ die Sprachbeschreibung.

3.2.3. Ressourcen

Ressourcen sind hier allgemein die Mittel zur Implementierung von Algorithmen, also die unmittelbar gegebenen grundlegenden Datenstrukturen, elementaren Algorithmen und explizit nutzbaren technischen Einrichtungen. Die Ressourcenbeschreibung $\mathcal{R}$ wird folgendermaßen definiert:

$$\mathcal{R} = \{d, c, h, r, s, b\} \qquad (3.3)$$

Die Bedeutung der Symbole ist in Tafel 3 erklärt.
Die Mengen sind gemäß (3.1) zerlegbar; dabei gilt für jede Menge q; q = (d, c, b, r, s):

D^q	beschreibt die Kardinalitäten (Anzahl der Bits bzw. der elementareren Strukturen)
M^q	beschreibt die Wirkungen.

Beispielsweise beschreibt D^d die Datenformate, D^c die Befehlsformate und M^c die Wirkungen der Befehle.
b darf nur Ausdrücke aus d, c, h, r, s sowie allgemein- logische[2] und Ausdrücke des zeitlichen Folgens enthalten. Im besonderen müssen sich alle Wirkungen durch Nennung der jeweili-

---

1 Die Programmiersprache muß Ausdrucksmittel für Datentypen, Verknüpfungen und Abläufe enthalten. Beispiele: C, Clu, Ada.
2 Die Ausdrucksweise folgt /300/. Allgemein- logische Ausdrükke sind UND, ODER, NICHT, ELEMENT VON, FÜR ALLE...GILT, WENN...SO usw.

Symbol	Bedeutung	praktische Entsprechung in Maschinenarchitekturen	praktische Entsprechung in höheren Programmiersprachen
$d = \{d_1 \ldots d_\mu\}$	Menge der Datenstrukturen	Datenformate (Byte, Maschinenwort usw.)	elementare Datentypen (Integer, Real usw.)
$c = \{c_1 \ldots c_\nu\}$	Menge der Steuerstrukturen	Befehle und Steuerworte	elementare Operatoren
$h = \{h_1 \ldots h_\zeta\}$	Menge der technischen Einrichtungen	architekturseitig definierte Register u. ä.	- (leere Menge)
$r = \{r_1 \ldots r_\varrho\}$	Menge der Arten des Datenzugriffs	Adressierungsprinzipien	Konstrukte für Zugriffe zu Datenstrukturen
$s = \{s_1 \ldots s_\sigma\}$	Menge der Prinzipien der Ablauforganisation	Steuerung der Befehlsfolge, Verzweigungen, Unterbrechungsbehandlung usw.	Konventionen der Abarbeitungsreihenfolge, Verzweigungen, Prozeduraufruf, Ausnahmebehandlung usw.
b	Beschreibung des Zusammenwirkens	beschreibt alle Einzelheiten, die nicht im Rahmen von d...s erklärbar sind	

Tafel 3 Symbolerklärungen für die Ressourcendefinition

gen Elemente von M^c bzw. h und der zulässigen Aktivitäten (aus r, s) ausdrücken lassen. Man beachte, daß keine physischen Verbindungen definiert sind: (3.3) ist eine funktionelle Beschreibung, keine strukturelle.
Ist ein Algorithmus $\mathcal{U}$ mit gegebenen Ressourcen $\mathcal{R}$ zu implementieren, so sind alle Datenstrukturen $\mathcal{S}$ aus Datenstrukturen aufzubauen, die in $\mathcal{R}$ definiert sind, und alle Informationswandlungen von $\mathcal{U}$ sind auf Folgen von elementaren Algorithmen zurückzuführen, die in c, r, s bzw. b vorgesehen sind.[1]
Die Ablaufbeschreibung von $\mathcal{U}$ muß also mit $\mathcal{R}$ auskommen, sie darf höchstens noch Eigennamen von Daten und Ausdrücke aus deren Strukturbeschreibung $D^{\mathcal{U}}$ enthalten.

3.2.4. Schaltungsanordnungen

Für die weiteren Betrachtungen liefert ein Strukturgraph $\Sigma = \{E, V, \Gamma, \Gamma^L, \Gamma^b, \Pi\}$ ein hinreichend genaues Modell einer Schaltungsanordnung.[2] Im einzelnen bezeichnet:

E die Liste der Schaltelemente (Knoten des Strukturgraphen), wofür entweder Speicher (S- Knoten) oder kombinatorische Schaltungen (K- Knoten) in Frage kommen. E ist gemäß (3.1) zerlegt; dabei beschreibt $D^E = \{D^E_1 \ldots D^E_\mu\}$ die Art des Knotens (S oder K) und $M^E = \{M^E_1 \ldots M^E_\mu\}$ dessen Ausgestaltung im einzelnen.

V die Liste der Verbindungen (Kanten des Strukturgraphen). In der Zerlegung nach (3.1) beschreibt $D^V = \{D^V_1 \ldots D^V_\nu\}$ die jeweilige Anzahl an Verbindungsleitungen, und $M^V = \{M^V_1 \ldots M^V_\nu\}$ kennzeichnet deren Verwendung (z. B. Taktleitungen, Datenleitungen, Adressenleitungen usw.).

Γ die Verbindungsmatrix. Es ist eine quadratische Matrix mit μ Zeilen. Besteht eine Verbindung von einem Knoten i zu einem Knoten j, so ist an der Position Γ_{ij} die Ordinalzahl in V eingetragen, die die Art der Verbindung kennzeichnet, sonst eine Ø.

Γ^L die Leitungszahlmatrix. Sie hat dieselbe Struktur wie Γ und enthält für jede Verbindung von einem Knoten i zu einem Knoten j in der Position Γ^L_{ij} die Anzahl der Verbindungsleitungen.

Γ^b die binäre Verbindungsmatrix. Sie hat dieselbe Struktur wie Γ und enthält für jede Verbindung von einem Knoten i zu einem Knoten j in der Position Γ^b_{ij} eine 1.

---

1 Die elementaren Algorithmen (s. Tafel 3) sind innerhalb von $\mathcal{R}$ nicht weiter zurückführbar. Ihre Ressourcen sind die Mittel, mit denen $\mathcal{R}$ implementiert ist.

2 In der Praxis sind zur Dokumentation komplexer Schaltungsanordnungen umfangreiche Datenmassive erforderlich. Σ entspricht dem Schaltplan auf der Ebene der Funktionalelemente.

Π die Funktionsbeschreibung der Anordnung.[1] Sie enthält neben Ausdrücken aus E und V nur noch allgemein- logische Ausdrücke und solche des zeitlichen Folgens.

An sich bestimmen E, V, Π den Strukturgraphen vollständig. Die Matrizen Γ, Γ^L, Γ^b wurden eingeführt, um bestimmte Eigenschaften von Σ elegant auswerten zu können.[2]

3.2.5. Abbildungsfragen

Um einen Algorithmus $\mathfrak{A}$ tatsächlich auszuführen, ist eine Schaltungsanordnung Σ erforderlich, die die Anforderungen der Ressourcenbeschreibung $\mathfrak{R}$ erfüllt.[3] Dazu muß die Schaltungsanordnung Σ folgenden Anforderungen entsprechen:

1. Σ muß S- Knoten enthalten, die zur Aufnahme aller in $\mathfrak{R}$ beschriebenen Informationsstrukturen geeignet sind, und zwar in dem Umfang, wie er durch die in D^{Γ}, D^{S} angegebenen Kardinalitäten erforderlich ist.

2. Σ ist so auszugestalten, daß alle in M^{c}, M^{Γ}, M^{S} beschriebenen Wirkungen durch zeitliche Folgen von Signalflüssen eintreten, die sich im Rahmen von Π beschreiben lassen, sowie durch Verknüpfungen in K- Knoten und Informationsspeicherung in S- Knoten.

Gelingt es, einen Algorithmus $\mathfrak{A}$ so in eine Schaltungsanordnung Σ abzubilden, daß alle Daten $\mathfrak{D}$ eindeutig S- Knoten zugeordnet werden können und die Ablaufbeschreibung $\mathfrak{P}$ der Wandlungsvorschrift $\mathfrak{W}$ nur eine einzige Aktivierung umfaßt (die Anwendung einer einzigen Steuerstruktur $c_i \in c$ im Rahmen eines einzigen Prinzips $p_j \in p$ der Ablauforganisation), so heißt der Algorithmus $\mathfrak{A}$ vergegenständlicht in der Schaltungsanordnung Σ.
Erfüllt eine Schaltungsanordnung Σ die Anforderungen einer Ressourcenbeschreibung $\mathfrak{R}$, so sind alle in $\mathfrak{R}$ definierten Informationswandlungen in Σ vergegenständlichte Algorithmen.
Von nun an wird ausdrücklich zwischen Vergegenständlichung und Implementierung von Algorithmen unterschieden, das heißt zwischen der direkten Umsetzung in Schaltungsanordnungen und der Umsetzung in zeitliche Folgen anderer Algorithmen.

---

1 Dafür reichen die Mittel der Automatentheorie an sich aus. Die Praxis zeigt aber, daß ein einzelnes Beschreibungsmittel (Graph, Übergangsmatrix, Phasenliste usw.) nicht allen Anforderungen gerecht wird, so daß verschiedene Darstellungsweisen im Verbund genutzt werden müssen (vgl. /51/).

2 Das wird z. B. in Abschnitt 4.3. genutzt.

3 Die Schaltungsstruktur darf einen größeren Funktionsumfang aufweisen als gemäß $\mathfrak{R}$ erforderlich ist. Nur sind bei gegebenem $\mathfrak{R}$ die zusätzlichen Möglichkeiten nicht nutzbar, um $\mathfrak{A}$ zu implementieren (typisches Beispiel: zusätzliche Spezialhardware, die von einer höheren Programmiersprache ($\mathfrak{R}$) aus nicht zugänglich ist).

4. Grundlagen der Bewertung

Um leistungsfähige und technisch- ökonomisch sinnvolle Schaltungsanordnungen zu schaffen, sind verschiedene Lösungsansätze zu erarbeiten und zu bewerten. Teillösungen, die sich als zweckmäßig erwiesen haben, sind zu Systemlösungen zusammenzufassen. Dafür sind Bewertungsmaßstäbe erforderlich: Schaltungsanordnungen sind nach ihrer Leistungsfähigkeit zu bewerten, Algorithmen nach der Eignung zur Vergegenständlichung und Aufwendungen nach ihrer Nützlichkeit. Weiterhin sind die verschiedenen Lösungsansätze untereinander zu vergleichen.

4.1. Bewertung der Schaltungsanordnungen

Zunächst wird das absolute Leistungsvermögen von Hardware untersucht; die Aufwendungen bleiben also außer Betracht. Allgemein übliche Bewertungsgrundlagen sind:[1]

1. Ausführungszeiten bestimmter Algorithmen (Anwendungsprogramme)
2. Ausführungszeiten "repräsentativer" Algorithmen ("benchmark"- Programme)
3. Ausführungszeiten vergegenständlichter Algorithmen (bei üblichen Rechnern in herkömmlicher Redeweise: die Ausführungszeiten der Maschinenbefehle).

Es ist klar, daß brauchbare Angaben nur selten analytisch oder durch einmalige Probeläufe zu erhalten sind, sondern daß Mittelwerte bzw. Erwartungswerte gebildet werden müssen.

Die Messung der Ausführungszeiten von Anwendungsprogrammen liefert einem Nutzer, der nur an bestimmten Algorithmen interessiert ist, gut auswertbare Vergleichsdaten über die Eignung verschiedener Maschinen. Sie hat aber folgende Nachteile:

- Anwendbarkeit nur auf fertige Hardware- Software- Komplexe (Maschinen und Programme müssen vorhanden sein); kaum geeignet für die Bewertung von Lösungsansätzen neuer Schaltungsstrukturen (Simulation ist aufwendig und langsam)

- sehr beschränkte Aussagekraft hinsichtlich des allgemeinen Leistungsvermögens

- vergleichsweise hohes Bewertungsrisiko: geringe Änderungen in den Algorithmen, Programmen bzw. Compilern können sich erfahrungsgemäß in manchen Maschinen deutlich auf die Laufzeiten auswirken, in anderen nicht.

Derartige Messungen testen nicht nur die Hardware, sondern auch den Compiler und die Kunstfertigkeit des Programmierers.

---

1 Derzeitig gibt es noch kein gesichertes Fachwissen darüber, wie ausgewogene ("well-balanced") Hardware- Software- Systeme zu entwerfen sind, die hohe Leistungen für normale Nutzer liefern. Man kann sich für die Leistungsbewertung nur auf empirische Resultate verlassen. (Nach: /213/)

So ist in /264/ dargestellt, daß auf der CRAY-1 die rechts angegebene Schleife um 25% schneller läuft als die linke[1]:

```
  DO 1Ø   I = 1,N                    DO 1Ø   I = 1,N
      Y(I) = A*X(I)+Y(I)                 Y(I) = A*X(I)+(Y(I))
 1Ø CONTINUE                        1Ø CONTINUE
```

Praktische Erfahrungen zeigen, daß allein ein Wechsel des Compilers das Laufzeitverhalten der Programme beträchtlich verändern kann.

Repräsentative Algorithmen, die eindeutig dokumentiert sind (z. B. in einer verbreiteten Programmiersprache), sind für das überschlägige Vergleichen zweckmäßiger; sie gestatten es, die Eignung von Maschinen für bestimmte Klassen von Algorithmen zu beurteilen ("benchmark"- Tests). Mit einigen Tests kann man das Leistungsvermögen bezüglich bestimmter Operationen recht genau erfassen.[2] So kann man beim bekannten LINPACK- Benchmark (Lösung linearer Gleichungssysteme) die Gleitkommaoperationen (Addition und Multiplikation) auszählen: ein System aus n Gleichungen erfordert

$$\frac{2}{3}n^3 + 2n^2 + O(n)$$

solche Operationen.
Die gemessenen Werte weichen erheblich von den Angaben der Maximalleistung ("peak performance") ab (Tafel 4).[3]

Hingegen lassen sich die Ausführungszeiten der vergegenständlichten Algorithmen ("Maschinenbefehle") an sich recht einfach aus einem hinreichend detaillierten Schaltungsentwurf bestimmen (durch Auszählen der Taktzyklen und einige statistische Annahmen, z. B. hinsichtlich der Vermittlungszeiten von Speicherzugriffen und der Trefferraten bei Cache- Speichern). Besonders beliebt ist in der Praxis die Annahme der jeweils günstigsten Verhältnisse: so kommen die meisten der üblichen MIPS- bzw. MFLOP- Angaben zustande.
Ein Blick in Tafel 4 zeigt die Fragwürdigkeit solcher Angaben. Etwas bessere Vergleichswerte liefern die bekannten statistischen Befehlsverteilungen (Mix- Werte, z. B. Gibson- Mix, GPO-Mix usw.). Sie können zu Vergleichszwecken recht einfach berechnet werden; man sollte aber nicht errechnete Zahlen mit gemessenen vergleichen, und die betreffenden Maschinenarchitekturen sollten einigermaßen vergleichbar sein.[4]
Des weiteren hat sich gezeigt, daß eine auf gute Mix- Leistung hin entworfene Maschine in der Anwendungsleistung nicht immer im erwarteten Maße überlegen ist.[5]

---

1 Gilt für Übersetzung mit Fortran- Compiler Level 1.Ø9.
2 Auch dieses Verfahren ist ohne fertige Maschine (mit Betriebssystem und Compiler) kaum anwendbar.
3 Die Darstellung (einschließlich Tafel 4) stammt aus /161/; s. weiterhin /159/, /16Ø/.
4 Man vergleiche also nur CISC- Maschinen, RISC- Maschinen, Vektorprozessoren usw. jeweils untereinander.
5 Z. B. EC 1Ø55 im Vergleich mit EC 1Ø4Ø.

lfd. Nr.	Maschine	Zyklus (ns)	Prozessoren
1	Culler PSC	200	1
2	Multiflow TRACE 7/200	130	1
3	Convex C-1	100	1
4	SCS-40	45	1
5	FPS 264	38	1
6	Alliant FX/8	170	8
7	Amdahl 500	7,5	1
8	CRAY-1	12,5	1
9	CRAY X-MP-1	9,5	1
10	IBM 3090/VF-200	18,5	2
11	Amdahl 1100	7,5	1
12	NEC SX-1E	7	1
13	CDC CYBER 205	20	1
14	CRAY X-MP-2	9,5	2
15	IBM 3090/VF-400	18,5	4
16	Amdahl 1200	7,5	1
17	NEC SX-1	7	1
18	CRAY X-MP-4	9,5	4
19	Hitachi S-810/20	14	1
20	NEC SX-2	6	1
21	CRAY-2	4,1	4

lfd. Nr.	Leistung (MFLOP) maximal	LINPACK	Effizienz
1	5	2	0,4
2	15	6	0,4
3	20	3	0,15
4	44	8	0,18
5	54	5,6	0,1
6	94	7,6	0,08
7	133	14	0,11
8	160	12	0,075
9	210	24	0,11
10	216	12*	0,11 (0,056)
11	267	16	0,06
12	325	35	0,11
13	400	17	0,043
14	420	24*	0,11 (0,057)
15	432	12	0,11 (0,028)
16	533	18	0,034
17	650	39	0,06
18	840	24*	0,11 (0,029)
19	840	17	0,02
20	1300	46	0,035
21	2000	5*	0,03 (0,0075)

* Angabe für einen Prozessor

Tafel 4 Das Leistungsvermögen von Hochleistungsrechnern; mit dem LINPACK- "benchmark" **gemessen**

Hier sollen nicht nur bekannte bzw. von vornherein leicht vergleichbare Prinzipien untersucht werden, und es ist wünschenswert, Schaltungslösungen in einem frühen Bearbeitungsstand[1] überschlagsmäßig beurteilen zu können. Dafür wird ein Verfahren vorgeschlagen, das auf folgenden Überlegungen beruht:

Für einen einzelnen Algorithmus interessiert an sich nur die reine Ausführungszeit; die Zeit, die die Maschine braucht, um die gewünschten Resultate zu liefern.
Weiterhin kann man sich vorstellen, für den besagten Algorithmus eine Sondermaschine zu bauen. Daß programmgesteuerte Universalmaschinen verwendet werden, hat unter diesem Gesichtspunkt nur den Grund, daß es nicht durchführbar ist, für jeden Algorithmus eine Sondermaschine zu bauen. Die programmgesteuerte sequentielle Ausführung eines Algorithmus ist also nur ein Notbehelf, eben wegen der technisch- ökonomischen Gegebenheiten. Man kann deshalb die Abarbeitung von Befehlen eher als Störfaktor auffassen und nur die Argumente und Resultate der Algorithmen betrachten. Diese Werte sind binär codiert, und sie werden in aufeinanderfolgenden Taktzyklen von Speichermitteln zu Speichermitteln über Verbindungsleitungen und kombinatorische Netzwerke bewegt.
Für eine Sondermaschine ist die Frage nach "MIPS" an sich gegenstandslos. Solche Angaben sind nur sinnvoll, wenn es darum geht, den Geschwindigkeitszuwachs einer Sondermaschine im Vergleich zur Nutzung einer Universalmaschine für den selben Zweck zu beurteilen. Beispiel: Der leistungsentscheidende Ablauf eines Algorithmus erfordere 5Ø Befehle einer geläufigen Rechnerarchitektur. Eine Sondermaschine leiste dasselbe in 1 µs. Es müßte also ein Universalrechner mit 5Ø MIPS beschafft werden, um das Leistungsvermögen der Sondermaschine zu erreichen[2] (an diese Überlegung wird sich üblicherweise eine Kostenabschätzung anschließen). Solche Betrachtungen wurden z. B. in /243/ angestellt, um die Sinnfälligkeit konkret entworfener Sondermaschinen im Vergleich zu Universalrechnern beurteilen zu können. Hier stellt sich die Aufgabe gerade anders herum: das Leistungsvermögen der Sondermaschine ist bekannt, und es ist die Universalmaschine zu bewerten. Die Sondermaschine verkörpert die leistungsfähigste technisch beherrschbare Vergegenständlichung des Algorithmus. Sie erzeugt die Resultate in einer minimalen Anzahl von Maschinenzyklen.

Um ein Maß für die Verarbeitungsleistung zu gewinnen, ist es mithin ausreichend, zu zählen, wieviele Bits an Nutz- Information (Argumente und Resultate der jeweils betrachteten Algorithmen) in einer bestimmten Zeiteinheit verarbeitet bzw. erzeugt werden. Bezeichnungsvorschlag: "Effektive Bits je Sekunde" (EB/s; mit Faktoren $1Ø^6$ bzw. $1Ø^9$ dann MEB/s bzw. GEB/s).

---

1 Z. B. nach Ausarbeitung der Funktionsprinzipien und des Blockschaltbildes bzw. der Register- Transfer- Struktur.
2 Die Aussage "die Sondermaschine leistet x MIPS" ist falsch! Zutreffend ist vielmehr: "Die Sondermaschine ersetzt für den betreffenden Algorithmus einen Universalrechner von x MIPS".

Um dieses Leistungsmaß (im folgenden mit PM bezeichnet) zu bestimmen, wird in einem Intervall t_x gezählt, wieviele Zugriffe zur Nutz- Information (die durch ψ in (3.2) beschrieben wird) dabei ausgeführt werden (die Anzahl sei r).[1] Die Anzahl der Nutzbits in Zugriff i ($1 \leqslant i \leqslant r$) sei CARDBi. Dann gilt:

$$PM = \frac{1}{t_x} \sum_{i=1}^{r} CARDBi. \qquad (4.1)$$

Da die Ausführungszeiten der meisten Algorithmen datenabhängig sind, müssen in der Praxis Mittelwerte bzw. Erwartungswerte gebildet werden. Die Angaben müssen stets auf den jeweiligen Algorithmus bzw. Komplex von Algorithmen bezogen werden.
Für überschlägige Abschätzungen und Vergleiche eignet sich beispielsweise der Vorrat an elementaren Operatoren (numerische und nichtnumerische), der in eingeführten Programmiersprachen (z. B. Fortran, C, Ada) definiert ist.[2] Oft lassen sich die Operatoren direkt auf Maschinenbefehle abbilden; gelegentlich erfordert ein Operator eine Folge von Maschinenbefehlen (z. B.: Multiplikation in RISC- Architekturen). Es wird von Verzweigungen usw. abgesehen und eine lückenlose Folge von Operatoren angenommen, die gespeicherte Argumente verarbeiten und die Resultate wieder in einem allgemein zugänglichen Speicher ablegen.[3] Dem eigentlichen Vergleich wird dann eine sinnfällige Folge ("Mix") solcher Operatoren zugrunde gelegt, wobei die Ausführungszeit (t_x) und die unbedingt notwendigen Argument- und Resultattransporte (CARDBi) bestimmt werden.[4] Bei Hochleistungssystemen reicht es bisweilen aus, den Idealfall anzunehmen, daß alle Datenpfade und Verarbeitungswerke voll ausgelastet sind: es wird dann die maximale Datenrate betrachtet.[5]

4.2. Bewertung der Algorithmen

Um Algorithmen zur Vergegenständlichung auszuwählen, ist deren Eignung zu bewerten. Dabei geht es nicht um die Nützlichkeit oder universelle Anwendbarkeit - eine solche Vorauswahl wird als gegeben angenommen -, sondern um die grundsätzliche Möglichkeit, leistungsmäßig überlegene Schaltmittel entwerfen zu können. Die zeiteffektivste Form der Vergegenständlichung ist dann gegeben, wenn die Argument- und Resultatbits nur jeweils einmal transportiert werden müssen.[6]

---

1 Für künftige Maschinen sollte auch an technische Vorkehrungen zur Leistungsmessung gedacht werden.
2 Vgl. etwa Tafel 8 (S. 72).
3 Die Speicher müssen für andere Prozessoren bzw. für die Ein- und Ausgabe zugänglich sein. Register-Register-Verknüpfungen verlängern t_x, zählen aber nicht für CARDBi.
4 Datentransporte aus Emulationsgründen (z. B. zu Hilfsbereichen im RAM) verlängern t_x, zählen aber nicht für CARDBi.
5 Zugriffe zu Befehlen, Deskriptortabellen usw. verlängern manchmal (wenn nicht parallelisierbar) t_x, zählen aber nicht für CARDBi.
6 Dieser Ansatz wurde in /243/ erstmals beschrieben.

Um diesen Sachverhalt zu bewerten, wird der Begriff der Implementierungseffizienz [1] e_i wie folgt eingeführt:

$$e_i = \frac{\sum_{i=1}^{n} CARDB(A_i) + \sum_{j=1}^{m} CARDB(R_j)}{z \cdot (ARG\_LINES + RES\_LINES)} \qquad (4.2)$$

Bedeutung der Symbole:

- CARDB(A_i), CARDB(R_j): Anzahl der Bits, mit denen die Argumente A_i ($1 \leqslant i \leqslant n$) bzw. Resultate R_j ($1 \leqslant j \leqslant m$) jeweils codiert sind
- ARG_LINES, RES_LINES: Anzahl der genutzten Leitungen für die Zuführung der Argumente bzw. den Abtransport der Resultate
- z: Anzahl der Maschinenzyklen, die für die Bildung aller Resultate benötigt werden ($z \geqslant 1$).

Wird kein Resultat gespeichert (wenn beispielsweise der Algorithmus lediglich eine Bedingung prüft), so sind die Resultat-Terme gemäß der jeweiligen binären Codierung (z. B.: einzelne Bedingungsleitung, 2-bit- Bedingungscode o. ä.) anzusetzen.
Ist eine technisch gegebene Leitungszahl (z. B. aus ⌐¬ ablesbar) größer als die betreffende Anzahl an Bits, so gilt letztere.
Die Implementierungseffizienz ist eine dimensionslose Zahl im Intervall $\emptyset < e_i \leqslant 1$, und es ist ersichtlich, daß im günstigsten Fall $e_i = 1$ ist. Um das zu erreichen, müssen die beiden folgenden Sätze erfüllt sein:

1. $e_i = 1$ kann grundsätzlich nur dann erreicht werden, wenn für jeden Abschnitt des Resultats gilt, daß dieser durch Zuordnung (d. h. gemäß dem Schema von Bild 4) aus den jeweils aktuellen Abschnitten der Argumente erzeugt werden kann, wobei in diese Zuordnung höchstens noch Zustands- Information einbezogen ist, die eindeutig durch die zuvor verarbeiteten Abschnitte bestimmt ist.

2. Um $e_i = 1$ praktisch verwirklichen zu können, müssen die Abläufe des Algorithmus so zerlegbar sein, daß jeder Abschnitt der Resultate sowie erforderlichenfalls die Zustands- Information im Sinne von Satz 1 ausschließlich durch kombinatorische Verknüpfungen gemeinsam selektierbarer Abschnitte der Argumente sowie der besagten Zustands- Information des jeweils vorausgegangenen Verarbeitungszyklus gebildet wird.

Satz 1 muß stets erfüllt sein, da nur so gewährleistet ist, daß zur Gewinnung eines Resultatabschnittes keine zusätzlichen Zustandswechsel erforderlich sind. Man braucht hingegen solche Zustandswechsel, wenn auch nur ein Resultatabschnitt von Argumentabschnitten abhängt, die nicht im jeweils aktuellen Zyklus zugänglich sind oder in voraufgegangenen Zyklen bereits verarbeitet wurden: diese Argumentabschnitte sind nur mit

---

1 Die Bezeichnung aus /243/ wird zunächst beibehalten; sie wäre ggf. künftig durch Vergegenständlichungseffizienz ("efficiency of incarnation") zu ersetzen.

zusätzlichen Zugriffen erreichbar. Für $e_i = 1$ ist es aber nicht notwendig, daß ein Resultatabschnitt nur von den aktuellen Argument- Abschnitten abhängt: es können auch Abhängigkeiten von zuvor verarbeiteten Abschnitten bestehen; sofern diese über die Zustands- Information vermittelt werden können. Bild 12 zeigt, wie dafür das Schema von Bild 4 bzw. 5 zu erweitern ist
Die praktische Konsequenz von Satz 2 besteht darin, daß $e_i = 1$ nur von bestimmten Verarbeitungsbreiten an realisierbar ist, d. h. nur im Rahmen technischer Auslegungen, die gewährleisten, daß die jeweils erforderlichen Argument- und Zustandsbits parallel in den Speichermitteln selektiert und den Verknüpfungsnetzwerken zugeführt werden können.
Grundsätzlich ist die Implementierungseffizienz kleiner als 1, wenn:

1. die abschnittsweise Zuordnung technisch nicht zu verwirklichen ist (Aufwand, Kompliziertheit), so daß bestimmte Folgen von Maschinenzuständen notwendig sind, um einen Resultatabschnitt zu erzeugen

2. Resultatbits von Argumentbits abhängen, die sich in verschiedenen Abschnitten befinden können, so daß Zugriffe zu mehreren Argumentabschnitten nötig sind, um diese Resultatbits zu bestimmen

3. bestimmte Argumentbits veranlassen, daß Teile bereits erzeugter Resultatabschnitte geändert werden müssen, so daß erneute Zugriffe zu diesen Abschnitten auszuführen sind.

Ein Sachverhalt nach 1. ist nicht immer ein unüberwindliches Hindernis: es ist letztlich eine Ermessensfrage, was man als "zu aufwendig" oder "zu kompliziert" ansieht.
Hingegen bezeichnen die Sachverhalte 2. oder 3. objektive Grenzen: solche Algorithmen können nie mit $e_i = 1$ vergegenständlicht werden. Ein plausibles Beispiel dafür ist das Umordnen eines Vektors gemäß den Angaben einer Indexliste: jede Argumentposition kann grundsätzlich in jede Resultatposition transportiert werden, so daß es notwendig sein kann, bereits abgespeicherte Resultatabschnitte nochmals aufzurufen, um neue Werte einzufügen.

Alle diese Überlegungen gehen von der grundsätzlichen Wünschbarkeit der direkten Zuordnung aus. Durch abschnittsweise Zuordnung soll dieser Ansatz technisch verwirklicht werden, und es ist klar, daß damit Schaltmittel in bestmöglicher Weise ausgenutzt werden können, nämlich durch lückenlose Folgen von Nutzoperationen. Läßt sich $e_i = 1$ von einer gewissen Mindest-Verarbeitungsbreite an verwirklichen, so heißt das, daß jede so ausgelegte Schaltung in jedem ihrer internen Zyklen einen Anteil zum Endresultat beiträgt, der ihrer Verarbeitungsbreite entspricht; man erhält also das Maximum an Verarbeitungsleistung, das mit den vorgesehenen Aufwendungen überhaupt zu verwirklichen möglich ist.

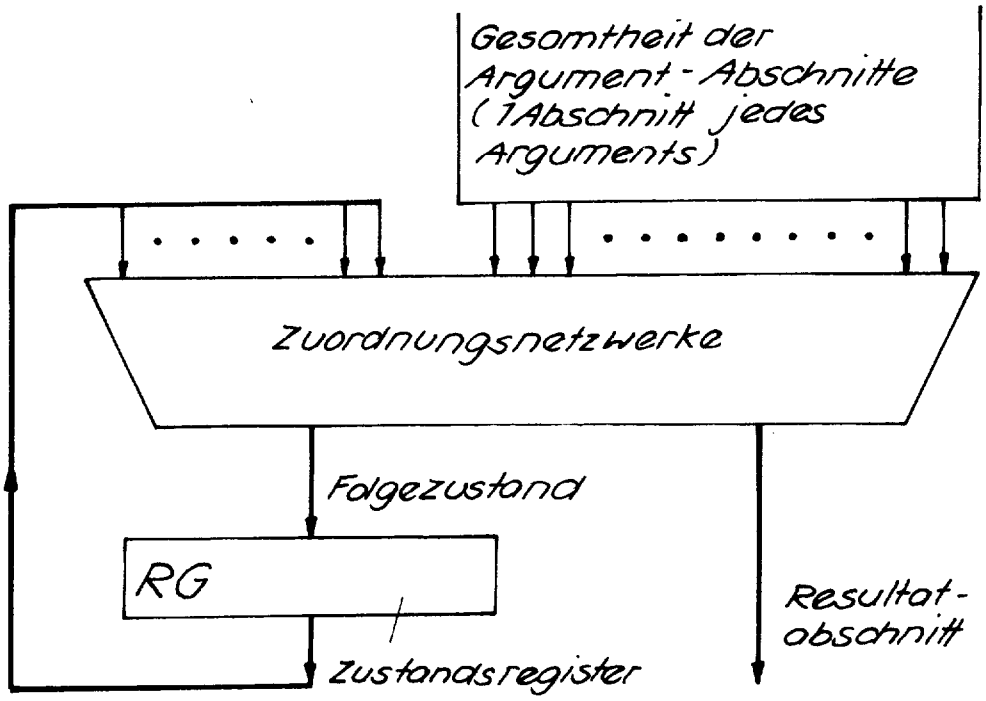
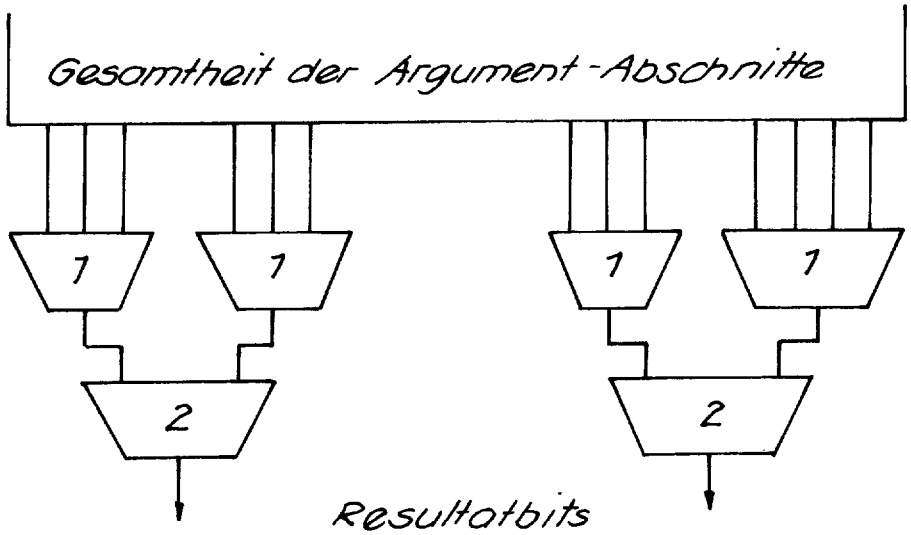


Bild 12 Prinzip der abschnittsweisen Zuordnung unter Einbeziehung von Zustands-Information



1: Netzwerke zur Verknüpfung unabhängiger Gruppen von Argumentbits
2: Netzwerke zur kombinatorischen Zusammenfassung

Bild 13 Illustration der gruppenweisen Zerlegbarkeit

Die Frage, ob für einen gegebenen Algorithmus eine technisch-ökonomisch sinnvolle Vergegenständlichung gelingt, ist letzten Endes nur durch zielgerichtete erfinderische Bemühungen entscheidbar, also durch Versuche, entsprechende Schaltungsanordnungen auszuarbeiten. Im besonderen sollte man sich nicht darauf beschränken, einen prozedural beschriebenen Algorithmus lediglich schaltungstechnisch umzusetzen, sondern man sollte gleichsam vorurteilsfrei nach Lösungen suchen, die die gewünschten Wirkungen hervorbringen.[1] Ein solcher Weg kann recht langwierig sein; deshalb seien einige Kriterien angeführt, die oft ein überschlagsmäßiges Urteil ermöglichen:

1. Beherrschbare Leitungszahlen liegen in der Größenordnung von 32 bis etwa 512. Dem Schema von Bild 4 entsprechen direkt arithmetisch- logische Einheiten, die aus 2 Argumenten zu 32 oder 64 bit ein Resultat gleicher Länge erzeugen und zusätzlich einige Bedingungsleitungen erregen (z. B. "Übertrag", "Resultat = Ø"). Die Grenzen sind im wesentlichen durch die beschränkten Kontaktzahlen an Schaltkreisen und durch Störeinflüsse gegeben (kapazitive und induktive Kopplung; impulsförmige Beeinflussung der Speisespannung, wenn viele Ausgangstreiber gleichzeitig schalten).[2]

2. Ob sich ein Zuordner technisch verwirklichen läßt, hängt von der Anzahl der Verknüpfungen, den technologischen Voraussetzungen und der Kompliziertheit der Booleschen Gleichungen ab, die die Verknüpfungen beschreiben.

3. Beliebig komplizierte Verknüpfungen können mit ROM- oder RAM- Zuordnern vergegenständlicht werden; die Variablenzahlen sind allerdings beschränkt:

- 8 bit/Verknüpfung: unproblematisch
- 12 bit/Verknüpfung: noch beherrschbar
- 16 bit/Verknüpfung: noch möglich.

Die absolute Obergrenze (wenn Kosten keine Rolle mehr spielen) liegt vielleicht bei 2Ø...24 bit/Verknüpfung.[3]

4. Die Chancen der Realisierbarkeit steigen an, wenn es gelingt, die notwendigen Verknüpfungen in Gruppen zu zerlegen, die jeweils für sich realisierbar sind und die entweder voneinander unabhängig sind oder mit einfachen Mitteln verknüpft werden können.[4] Das ist in Bild 13 veranschaulicht.

---

1 Das wurde in /241/ bzw. /243/ gezeigt. Man vergleiche die prozedurale Beschreibung des Algorithmus (/281/, /297/), deren technische Umsetzung in /123/ und die Lösung nach /241/, die durch abschnittsweise Parallelarbeit mit $e_i = 1$ deutlich (wenigstens 8-16 mal) schneller ist.

2 512-bit- Datenpfade sind schon ausgeführt worden (/151/).

3 1...16 Mbit je Resultat- Bitposition sind technisch durchaus noch beherrschbar (z. B. mit 1 bzw. 4Mbit DRAM).

4 Für Weiteres s. /243 /. Beispiel: die Schaltung nach /123/, die aus einem beliebigen Binärvektor einen Vektor erzeugt, der nur die erste Eins enthält (Indexvektor).

Die Verhältnisse lassen sich genauer untersuchen, wenn man für den betreffenden Algorithmus eine Abhängigkeitsmatrix |D| aufstellt. Das ist eine binäre Matrix, deren Zeilen den Bitpositionen aller Argumente entsprechen und deren Spalten den Bitpositionen aller Resultate. Hängt ein Resultatbit i von einem Argumentbit j ab (andersherum: übt das Argumentbit j Einfluß auf das Resultabit i aus), so steht in der Position ij der Matrix |D| eine Eins, sonst eine Null. (|D| repräsentiert nicht eine aktuelle Abhängigkeit, sondern alle grundsätzlich möglichen Abhängigkeiten bei beliebigen Werten.) Beispielsweise sieht die Matrix einer bitweise unabhängigen Verknüpfung von zwei 3-bit- Argumenten zu einem 3-bit- Resultat so aus:

$$|D| = \begin{array}{c|ccc} & r_1 & r_2 & r_3 \\ a_{11} & 1 & & \\ a_{12} & & 1 & \\ a_{13} & & & 1 \\ a_{21} & 1 & & \\ a_{22} & & 1 & \\ a_{23} & & & 1 \end{array}$$

Die Spaltensummen von |D| geben für jedes Resultatbit an, von wievielen Argumentbits es abhängt. Damit läßt sich abschätzen, ob direkte Zuordnungen realisierbar sind. Die Zeilensummen von |D| geben für jedes Argumentbit an, auf wieviele Resultatbits es Einfluß hat. Damit läßt sich abschätzen, welche Aufwendungen für die Informationswege bzw. für das Selektieren der Argumentbits zu veranschlagen sind. Bildet man die elementweise Konjunktion zwischen 2 Spalten und erhält dabei keine einzige Eins, so können die beiden Resultatbits unabhängig voneinander gebildet werden: folglich lohnt es sich, die Schaltung so auszulegen, daß die benötigten Argumentbits parallel für die Verknüpfungen zugänglich sind bzw. abschnittsweise parallel selektiert werden können.

4.3. Bewertung der Aufwendungen

Aufwendungen, wie Datenwege, Speicher, Verknüpfungsschaltungen, sind nach ihrer Nützlichkeit zu bewerten. Ein Schaltungsentwurf wird auch danach beurteilt, wie gut die Schaltmittel im praktischen Betrieb ausgelastet werden. Eine bestimmte Aufwendung ist um so nützlicher, je mehr sie zum Leistungsvermögen der gesamten Anordnung beiträgt. Um diese Nützlichkeit beurteilen zu können, wird der Begriff der Mehraufwandseffizienz eingeführt. Dieser geht auf /243/ zurück, wo gezeigt wurde, daß damit plausible Überschlagsbetrachtungen möglich sind, um in der Konzeptionsphase schnell Entscheidungen zu treffen.[1] Zunächst wird für die einzelne Schaltungsanordnung eine Hardware- Effizienz HE wie folgt definiert:

$$HE = \frac{\text{Leistungsangabe}}{\text{Aufwandsangabe}}$$

1 Z. B. auf Grundlage von Probeentwürfen bis zur Blockschaltbild- oder Register- Transfer- Ebene.

Es wird angenommen, für eine bestimmte Aufgabenstellung seien verschiedene Schaltungsanordnungen entworfen worden. Diese werden nach steigenden Aufwendungen geordnet, und es werden sinnfällige Übergänge von einfacheren zu aufwendigeren Schaltungen betrachtet. Für jeden Übergang von einer Anordnung 1 zu einer Anordnung 2 wird die Mehraufwandseffizienz ME_{12} folgendermaßen ermittelt:

$$ME_{12} = \frac{HE_2}{HE_1}$$

Es ist durchaus sinnvoll, sich auch die Absolutwerte vor Augen zu führen; dazu wird das Aufwandsverhältnis R_a und das Leistungsverhältnis R_p bestimmt:

$$R_a = \frac{\text{Aufwand 2}}{\text{Aufwand 1}} \qquad R_p = \frac{\text{Leistung 2}}{\text{Leistung 1}}$$

Damit ergibt sich: $ME_{12} = \frac{R_p}{R_a}$

Um diese einfachen Formeln anwenden zu können, ist es erforderlich, eine Vielzahl von Probeentwürfen zu erarbeiten. Die Ergebnisse sind aber keineswegs trivial. So läßt sich das Verfahren gut nutzen, wenn man von der Vorstellung einer Reihe verschiedener, aber untereinander kompatibler Hardware- Modelle für den gleichen Verwendungszweck ausgeht. Es dürfte sich dann oft zeigen, daß es besonders sinnfällige Modelle gibt , aber auch solche, deren Verwirklichung sich nicht lohnt.[1]

Im folgenden wird nur eine einzige Schaltungsanordnung betrachtet, um zu untersuchen, wie gut deren Schaltmittel ausgenutzt sind. Dazu wird die binäre Verbindungsmatrix Γ^b des Strukturgraphen $\sum$ verwendet. In der Anordnung seien n Algorithmen $\mathfrak{A}_1 \ldots \mathfrak{A}_n$ vergegenständlicht. Für jeden Algorithmus $\mathfrak{A}_\nu$, $\nu = 1 \ldots n$, wird eine Bedeckungsmatrix $|C_\nu|$ aufgestellt. Sie hat dieselbe Struktur wie Γ^b . In ihr ist jede Verbindung (Kante des Strukturgraphen) mit einer 1 bezeichnet, die vom betreffenden Algorithmus benutzt wird; es werden also alle Verbindungen markiert, die für die Ausführung des Algorithmus notwendig sind; dadurch sind auch die jeweils miteinander verbundenen Knoten als notwendig gekennzeichnet. Praktisch kann dieses Markieren so vonstatten gehen, daß die Bedeutung der einzelnen Schaltmittel und Verbindungen, d. h. deren Aufgabe bei der Vergegenständlichung der Algorithmen, in der Entwurfsphase formal beschrieben und rechentechnisch erfaßt wird. Es ist also nicht nur anzugeben, wie die Schaltmittel untereinander verbunden sind, sondern auch, wozu sie vorgesehen sind.[2]

---

1 So wurde in /243/ genau ein "Einstiegs"- Modell identifiziert, das bei etwa doppeltem Aufwand gegenüber dem vorgeordneten Modell eine 7-fache Leistung aufweist.

2 Wurde der Entwurf in herkömmlicher Weise erfaßt, sind diese Angaben kaum mehr zu ermitteln. Bemerkung: Ein ausgebauter Kalkül über Bedeckungsmatrizen könnte eine Grundlage dafür bilden, Testbelegungen automatisch zu erzeugen.

In der gesamten Anordnung sollte jede Komponente irgendeinen Zweck erfüllen, d. h. die elementweise disjunktive Verknüpfung aller Bedeckungsmatrizen sollte der binären Verbindungsmatrix Γ^b gleich sein:

$$\bigvee_{\nu=1}^{n} |C_\nu| = \Gamma^b$$

Ist das nicht der Fall, enthält die Anordnung überflüssige Verbindungen[1]; diese sind durch elementweise Antivalenzverknüpfung sofort ersichtlich.

Für einen bestimmten Algorithmus $\mathcal{A}_\nu$ ist die Schaltung dann gut ausgenutzt, wenn $|C_\nu|$ möglichst viele Einsen enthält. Das wird durch den Ausnutzungsgrad Ψ_ν für den jeweiligen Algorithmus $\mathcal{A}_\nu$ gekennzeichnet:

$$\Psi_\nu = \frac{\sum_{ij} |c_{ij}^\nu|}{\sum_{ij} \Gamma_{ij}^b} \quad ; \quad 0 < \Psi_\nu \leq 1$$

Diese Bewertungsweise allein wird ausgesprochenen Hochleistungsschaltungen nicht gerecht: jede kombinierte Auslegung bzw. Mehrfachnutzung von Schaltmitteln erhöht zwar den Ausnutzungsgrad, ist aber grundsätzlich mit Leistungsbeschränkungen verbunden. Für höchste Leistungsanforderungen sind erfahrungsgemäß die Vergegenständlichungen der einzelnen Algorithmen getrennt voneinander auszuführen[2] und in sich so auszubilden, daß das jeweils höchste Leistungsvermögen erreicht wird. Deshalb wird aus der Häufigkeit der Nutzung der einzelnen Algorithmen eine Ausnutzungsmatrix |UE| über den Erwartungswert bestimmt:

$$|UE| = \sum_{\nu=1}^{n} p_\nu |C_\nu| \quad ; \quad \sum_{\nu=1}^{n} p_\nu = 1$$

Aus |UE| lassen sich wichtige Schlüsse für die weitere Verfeinerung der Schaltungslösung ziehen. Dazu wird der Mittelwert $\overline{UE}$ gebildet, und es wird eine Abweichungsmatrix $|\triangle|$ aufgestellt:

$$\overline{UE} = \frac{\sum_{ij} |UE_{ij}|}{i \cdot j} \quad ; \quad |\triangle| = |UE| - \overline{UE} \quad \text{(elementweise Subtraktion).}$$

Komponenten, die sich durch eine große negative Abweichung hervorheben (bzw. absolut gesehen: die in |UE| Ø nahekommen), sind in ihrer Nützlichkeit grundsätzlich fragwürdig: es lohnt sich zu überprüfen, ob die so gekennzeichneten Schaltungsteile weggelassen werden können, wobei deren Aufgaben durch andere, an sich gegebene Einrichtungen übernommen werden.[3]

---

1 Fehlende Verbindungen sind ebenso erkennbar; dieser (praktisch bedeutsame) Gesichtspunkt wird hier vernachlässigt.

2 Meist werden gewisse Algorithmen zusammengefaßt und jeweils gemeinsame Schaltmittel vorgesehen (z. B. für Gleitkommaoperationen, Adressenrechnung usw.).

3 Z. B. durch mikroprogrammtechnische Emulation.

Komponenten mit außergewöhnlich großer positiver Abweichung (bzw. absolut: in |UE| nahe bei 1) haben eine besondere Bedeutung für das Leistungsvermögen des Systems. Es lohnt sich deshalb zu untersuchen, ob durch Verfeinerungen oder gezielten Einsatz zusätzlicher Mittel die Gesamtleistung weiter verbessert werden kann.

4.4. Bewertung des Wirkungsgrades

Das Ziel ist die universelle Maschine, die vergegenständlichte Algorithmen als Ressourcen bereitstellt, um damit eine Vielfalt von - üblicherweise recht komplexen - Anwendungsalgorithmen implementieren zu können. Mit den bisher vorgeschlagenen Bewertungsgrundlagen läßt sich das absolute Leistungsvermögen angeben, es läßt sich beurteilen, welche Algorithmen sich zur Vergegenständlichung eignen, und es läßt sich bewerten, wie gut die Schaltmittel ausgenutzt sind. Hingegen ist der Anwender ausschließlich daran interessiert, wie schnell seine Aufgaben praktisch bearbeitet werden. Solche Angaben sind durch Schätzungen, Simulation oder Probeläufe zu ermitteln. Damit sind beispielsweise Preis- Leistungs- Vergleiche mit anderen Maschinen möglich.
Wenn man universelle Maschinen mit höchstem Leistungsvermögen schaffen will, braucht man aber den Vergleich mit absoluten Leistungsgrenzen. Anhand solcher Vergleichswerte ist dann die Sinnfälligkeit des Lösungsvorschlages beurteilbar. Im besonderen wird sich herausstellen, ob die vergegenständlichten Algorithmen tatsächlich zweckmäßig ausgewählt wurden. Dazu wird vorgeschlagen:

Es wird für jeden wesentlichen Algorithmenkomplex eine fiktive Sondermaschine ausgearbeitet (so detailliert, daß deren Leistungsvermögen beurteilbar ist). Unterstellt man für die Sondermaschine eine technisch gerade noch beherrschbare Auslegung[1], so wird deren Leistungsvermögen den absoluten Leistungsgrenzen sehr nahe kommen.
Dann läßt sich ein Wirkungsgrad η einführen, der das Verhältnis der Leistungen der zu beurteilenden Universalmaschine und der fiktiven Sondermaschine repräsentiert[2]:

$$\eta = \frac{P_{UNIV}}{P_{SPEZ}}$$

Damit läßt sich ein Ziel für das Entwerfen neuartiger Universalmaschinen angeben: ihr Leistungsvermögen sollte für die anwendungspraktisch wichtigsten Algorithmenkomplexe dem von einschlägigen Sondermaschinen soweit wie möglich entsprechen.

---

1 Es bietet sich an, dafür die Technologien und Aufwendungen der jeweils leistungsfähigsten kommerziell verfügbaren Supercomputer anzusetzen (in /243/ bereits ausgeführt).

2 Je nach Zweckmäßigkeit kann der Wirkungsgrad auf eine bestimmte Technologie oder Größenordnung des Aufwandes bezogen werden; es ist dann für die fiktive Sondermaschine dieselbe Technologie oder ein ähnlicher Kostenrahmen wie bei der zu vergleichenden Universalmaschine anzunehmen.

5. Tiefenstrukturen des Verarbeitungsmodells

Im folgenden werden wesentliche Einzelheiten des Verarbeitungsmodells überblicksmäßig beschrieben, um jene Tatsachen, Zusammenhänge und Notwendigkeiten zu erkennen, die den technischen Ausgestaltungen zugrunde zu legen sind.
Bekannte, weit verbreitete Programmiersprachen und Rechnerarchitekturen liefern dafür das Erfahrungsmaterial[1]; sie sind gleichsam Ausdruck der Oberflächenstrukturen, woraus die Tiefenstrukturen abstrahiert werden.[2] Dazu werden die Erfahrungsbereiche der numerischen und nichtnumerischen Informationsverarbeitung mit Universalrechnern genutzt, und es werden Datenstrukturen, Programmstrukturen sowie Prinzipien der Informationsspeicherung betrachtet.[3]
Das ist auf den ersten Blick eine eher konservative Grundlage; deshalb sind einige Anmerkungen zur Einbeziehung neuerer Forschungsrichtungen notwendig:

1. Für elementare ("low level") Operationen gibt es leistungsfähige und flexible Prinzipien der Befehlsgestaltung und Steuerung.[4] Mit darauf beruhenden optimierten technischen Lösungen dürften sich Konzepte wie Lisp, Prolog oder Smalltalk wenigstens so effizient implementieren lassen, wie es dem Stand der Technik entspricht.

2. In Weiterführung der Arbeiten lassen sich künftig einschlägige Forschungsergebnisse (z. B. zu Prolog- Maschinen) bewerten und ggf. einbeziehen.

---

1 Nach /146/ kommen als Erfahrungsbasis für neue Rechner-Befehlslisten in Betracht: Programmiersprachen, Anwendungslösungen, Betriebssysteme und die elementaren Befehle ("low level instructions"), die in fast allen Systemen zu finden sind. Betriebssysteme werden hier über hinreichend ausgestattete Programmiersprachen (z. B. Ada) mit erfaßt. Die Analyse von Anwendungslösungen bleibt weiterführenden Arbeiten vorbehalten.

2 Die Begriffe stammen aus der Sprachwissenschaft; sie gehen u. a. auf Wittgenstein und Chomsky zurück (vgl. etwa die Erläuterungen in /45/, /54/). Die Oberflächenstrukturen sind durch die jeweilige Sprache bzw. durch das jeweilige konkrete Repräsentationssystem gegeben; die Tiefenstrukturen bringen das Wesen, die innewohnenden - bei Transformationen invarianten - Bedeutungen zum Ausdruck.

3 Für diesen Abschnitt wurden hauptsächlich folgende Quellen verwendet: /10/, /22/, /34/-/38/, /59/-/61/, /76/, /77/, /86/, /88/, /91/, /92/, /129/, /169/, /229/, /285/ zu Rechnerarchitekturen; /33/, /55/, /62/, /70/-/72/, /102/, /225/ zu Programmiersprachen.
Architektur- und Sprachbeschreibungen haben einen beachtlichen Umfang, so daß es unmöglich ist, in den folgenden Übersichten auf Einzelheiten einzugehen; dafür sei auf die jeweilige Original- Dokumentation verwiesen.

4 Beispiele: VLIW- Architekturen; Mikroprogrammsteuerungen großer EDV- Anlagen (370/168, 3081 u. a.).

3. Zur Vergegenständlichung von Funktionen der "Künstlichen Intelligenz" in Universalrechnern wird folgende vorläufige Arbeitshypothese vertreten:
Die Beschränkung auf den Prädikatenkalkül 1. Stufe (vgl. Prolog) ist völlig unzureichend, wenn man den Begriff ("KI") auch nur einigermaßen wörtlich nimmt. Vielmehr erscheint es notwendig, wenigstens einen n-stufigen Prädikatenkalkül zu implementieren[1] und darüber hinaus Vorkehrungen zu treffen, um auf einen gegebenen Informationsbestand ("Wissensbasis") verschiedene nichtklassische Logiken, wie modale, deontische, mehrwertige usw.[2] anwenden zu können.
Als Grundlage für die Implementierung eines n-stufigen Prädikatenkalküls eignen sich Operationen über binär codierte halbgeordnete Mengen. Dazu ist die Isomorphie zwischen dem Booleschen Verband ($B^k, \wedge, \vee, \neg$) und dem Potenzmengenverband ($P(M), \cap, \cup, \overline{\ }$) sowie zwischen dem Booleschen Ring ($B^k, \wedge, \oplus$) und dem Potenzmengenring ($P(M), \cap, \triangle$) technisch ausnutzbar.[3]
Wesentlich ist, daß sich die innersten Schleifen aller leistungsentscheidenden Algorithmen mit $e_i = 1$ vergegenständlichen lassen.[4] International wird diese Richtung (Durchmustern von Mengen statt Anwenden von Regeln) mit dem Stichwort "memory based reasoning" bezeichnet.[5]

5.1. Datenstrukturen

5.1.1. Numerische Datenstrukturen

In Maschinenarchitekturen sind nur wenige elementare Datenstrukturen definiert:

- natürliche bzw. ganze Binärzahlen fester Länge; Bild 14 zeigt einige Beispiele
- Näherungsdarstellungen für reelle Zahlen; zumeist als binäre Gleitkommazahlen (Bild 15)
- stellenweise binär codierte Dezimalzahlen (Bild 16).

Daraus können Vektoren, Matrizen und andere zusammengesetzte Strukturen gebildet werden; das wird aber von den meisten

---

1 Ein entsprechender Vorschlag ist in /244/ bzw. in /245/, Anhang 5, näher skizziert.
2 Zu nichtklassischen Logiken s. /5/, /18/, 93/.
3 Zu den theoretischen Grundlagen s. etwa /52/.
4 Das ist bereits in /240/ bzw. /243/ gezeigt worden. Für das hier skizzierte Vorhaben sind die Datenstrukturen, Algorithmen und Schaltungslösungen natürlich in den Einzelheiten zweckgerecht abzuwandeln.
5 Eine solche Lösung (mit einfacherer Zielstellung) wurde auf der Connection Machine implementiert (/311/). Ähnliche Ansätze zum parallelen Suchen in semantischen Netzen auf Grundlage spezieller Parallelverarbeitungs- Strukturen werden von anderen Forschungsgruppen bearbeitet (/190/, /193/). Die bisherigen Datenbasismaschinen bzw. Assoziativprozessoren sind praktisch als elementare Vorstufen für solche Lösungen anzusehen (vgl. z. B. /119/, /127/, /288/).

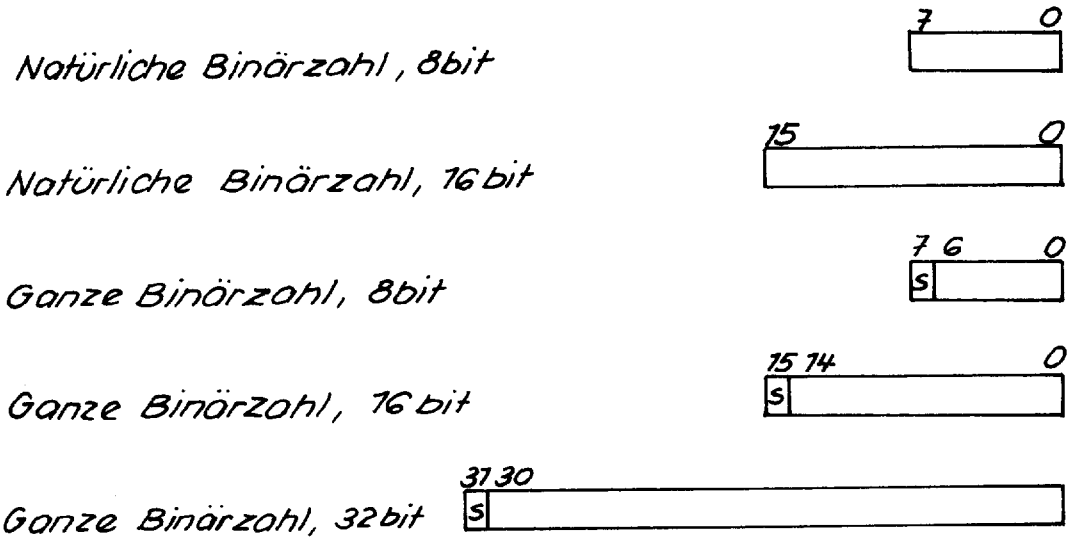


Bild 14 Beispiele für Formate natürlicher und ganzer Binärzahlen

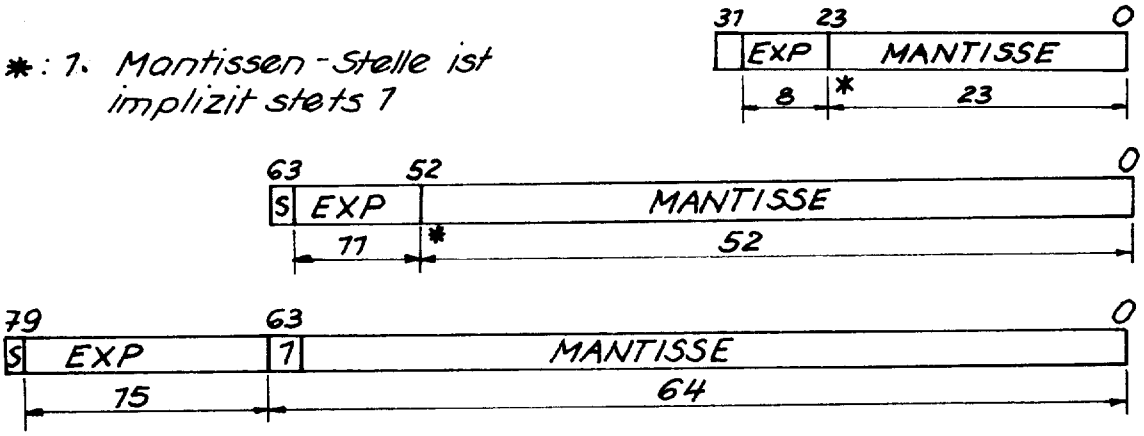


Bild 15 Beispiele für Gleitkommaformate (gemäß IEEE 754)

Je Byte 2 Dezimalstellen,
Länge 1-16 Bytes

Bild 16 Beispiel für das Format binär codierter Dezimalzahlen

Maschinenarchitekturen nicht direkt unterstützt. Nur bei Superrechnern ist die Datenstruktur "Vektor" in der Architektur definiert. Die Verarbeitungsgeschwindigkeiten unterscheiden sich beträchtlich:

- natürliche oder ganze Binärzahlen werden am schnellsten verarbeitet (zumeist in allen Stellen parallel; Addition bzw. Subtraktion in einem Taktzyklus)

- Gleitkommazahlen werden oft in allen Stellen parallel verknüpft, die Operationen erfordern aber mehrere Verarbeitungsschritte; höhere Leistungen sind nur mit speziellen Hardwarestrukturen zu erreichen (im Rahmen von Superrechnern oder als ergänzende Beschleunigungseinrichtungen)

- parallel wirkende Schaltmittel für binär codierte Dezimalzahlen sind zwar vorgeschlagen worden (so in /35/), sie haben sich aber nicht durchgesetzt. Manche Architekturen bieten lediglich Hilfsbefehle, um die Implementierung einer Dezimalarithmetik zu unterstützen; sie wirken aber auch bei großen Verarbeitungsbreiten oft nur für eine Dezimalstelle. Selbst in Maschinen mit ausgebauter Dezimalarithmetik sind die Grundoperationen meist nur seriell (byte- oder tetradenweise) implementiert.[1]

- Vektoren werden in üblichen Rechnern sequentiell mittels Programmschleifen verarbeitet. In Superrechnern bzw. speziellen Zusatzprozessoren wird das Pipelining- Prinzip genutzt, so daß in jedem Maschinenzyklus eine Komponente als Resultat erzeugt wird.[2] Bei manchen Architekturen sind die vergegenständlichten Vektor- Strukturen starken Beschränkungen unterworfen[3], um die Maschinenzyklen extrem kurz halten zu können; bei anderen wurde hingegen Wert auf sehr flexible Strukturen gelegt[4].

Tafel 5 zeigt eine Übersicht über numerische Datenstrukturen, die in eingeführten Programmiersprachen vorgesehen sind.

Tiefenstrukturen

Numerische Datenstrukturen sind sowohl für das eigentliche numerische Rechnen als auch für strukturell- deskriptive Angaben[5] von Bedeutung, so beispielsweise für:

- Angaben zur Beschreibung und Lokalisierung von Datenstrukturen (Adressen, Längen, Grenzen usw.)
- Kardinalzahlen (Mächtigkeit von Mengen bzw. Anzahlen)
- Ordinalzahlen.

---

1 Beispiele: 360/65, 370/158, EC 1040 und Nachfolger.
2 Die Vektor- Komponenten sind binäre Gleitkommazahlen.
3 Beispiel: Cray-1 (maximal 64 Komponenten je Vektor).
4 Beispiel: Matrixmodul zu EC 1055.
5 Der Begriff wurde aus /300/ entlehnt, hat aber eine gewandelte Bedeutung: er bezeichnet codierte Angaben, die Strukturen anderer Angaben beschreiben (z. B. Felder nach Typ, Speicheradresse und Größe).

PL/1

Numerische Daten können nach Modus, Basis, Skala und Genauigkeit spezifiziert werden.

Modus: REAL oder COMPLEX
Basis: BINARY oder DECIMAL
Skala: FIXED oder FLOAT

Die Genauigkeit wird durch die Stellenzahl beschrieben (Gesamtanzahl und Stellen nach dem Komma).

Ada

Es können ganze Zahlen sowie Gleitkomma- und Festkommazahlen deklariert werden. Für alle Datentypen ist eine Bereichsangaben (**range**) möglich; für Gleit- und Festkommazahlen auch eine Genauigkeitsangabe:

- **digits** (relative Genauigkeit; Gleitkomma)
- **delta** (absolute Genauigkeit; Festkomma).

C

Es sind ganze (**int**), natürliche (**unsigned int**) und Gleitkommazahlen (**float**) verschiedener Länge vorgesehen:

- short int
- int
- long int
- unsigned short int
- unsigned int
- unsigned long int
- float
- double

Tafel 5 Übersicht: Numerische Datenstrukturen in eingeführten Programmiersprachen

Dafür reichen natürliche Binärzahlen angemessener Länge aus.

Für das eigentliche numerische Rechnen sind Anforderungen zu erfüllen hinsichtlich der Genauigkeit, des Umfangs der Zahlenbereiche, des Speicherbedarfs und der Verarbeitungsgeschwindigkeit. Aus der Mathematik sind folgende grundlegende Zahlenbereiche bekannt:

1. die natürlichen Zahlen
2. die ganzen Zahlen
3. die rationalen Zahlen
4. die reellen Zahlen
5. die komplexen Zahlen.

Über jeden Zahlenbereich sind alle 4 Grundrechenarten definiert. Die entscheidende Aufgabe besteht darin, diese Bereiche auf finite Mengen binärer Codierungen abzubilden. Das ist mit 2 Beschränkungen verbunden:

1. man kann aus jedem Bereich nur ein endliches Intervall abbilden

2. die Intervalle der rationalen, reellen und komplexen Zahlen können nicht völlig exakt abgebildet werden.

Die Unzulänglichkeiten der üblichen Implementierungen waren Anlaß zu umfassenden Forschungsarbeiten (/22/, /77/, /134/). Deren Ergebnisse sollen im folgenden genutzt werden.

Es ist notwendig, numerische Rechnungen in folgenden Zahlenbereichen ausführen zu können:

1. natürliche sowie ganze Zahlen
2. reelle Zahlen
3. komplexe Zahlen
4. Intervalle über 2. und 3.
5. Vektoren und Matrizen über 2., 3. und 4.

In Tafel 6 sind die Bereiche 2. - 5. zusammen mit den grundlegenden Operationen dargestellt.
Jede Operation # (# $\in \{+, -, *, /\}$) in jedem Zahlenbereich M (gemäß Tafel 6) wird auf eine korrespondierende maschineninterne Operation ▦ im jeweiligen Bereich der Maschinendarstellung N unter Wahrung folgender Eigenschaften abgebildet:

(5.1) $x \boxplus y = \square(x \# y)$ für alle $x, y \in N$.

(5.2) $\square x = x$ für alle $x \in N$ (Rundung).

(5.3) $x \leqslant y$ impliziert $\square x \leqslant \square y$ für alle $x, y \in M$ (Monotonizität).

(5.4) $\square(-x) = -\square x$ für alle $x \in M$ (Antisymmetrie).

Zahlenbereich	Struktur	Operationen
reell	Skalar	+ - * / *
	Vektor	+ - *
	Matrix	+ - *
reelles Intervall	Skalar	+ - * / *
	Vektor	+ - *
	Matrix	+ - *
komplex	Skalar	+ - * / *
	Vektor	+ - *
	Matrix	+ - *
komplexes Intervall	Skalar	+ - * / *
	Vektor	+ - *
	Matrix	+ - *

Tafel 6 Zahlenbereiche, Strukturen und Operationen für das numerische Rechnen

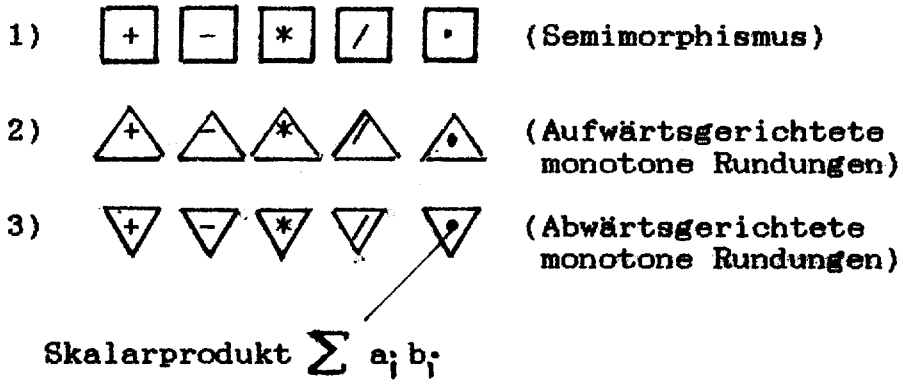


Bild 17 Grundlegende Operationen als Voraussetzung des numerischen Rechnens

Ist M eine Menge von Intervallen so bezeichnet $\leqslant$ die mengentheoretische Inklusion $\subseteq$, und die Rundung hat die zusätzliche Eigenschaft

(5.5) $x \subseteq \square x$ für alle $x \in M$ (aufwärtsgerichtete Rundung).

Eine solche Abbildung $\square$ stellt eine Annäherung an einen Homomorphismus dar. (5.2) ist eine natürliche Eigenschaft, die jede Rundung haben sollte. Es kann gezeigt werden, daß (5.1), (5.3) und (5.5) notwendige Bedingungen für einen Homomorphismus zwischen geordneten algebraischen Strukturen sind. Eine solche Abbildung $\square$ wird deshalb als Semimorphismus bezeichnet.

Bild 17 zeigt, welche elementaren Operationen notwendig sind, um alle Verknüpfungen gemäß Tafel 6 mit der erforderlichen Genauigkeit ausführen zu können. Wesentlich ist, daß zwischen dem korrekten Resultat und seiner Annäherung durch die maschineninterne Darstellung kein weiterer Wert der maschineninternen Darstellung vorkommt. Werden diese Anforderungen erfüllt, so sind an sich beliebige Codierungen zulässig. Die Codierung kann also ausschließlich im Hinblick auf eine hohe Implementierungseffizienz gewählt werden.

5.1.2. Nichtnumerische Datenstrukturen

In den verbreiteten Maschinenarchitekturen sind nur Zeichenketten und Binärvektoren (Bitfelder) vorgesehen.
Zeichen sind üblicherweise in 8-bit- Bytes codiert[1] (wichtige Codes sind ASCII und EBCDIC bzw. DKOI II). Zeichenketten sind Aneinanderreihungen von Bytes.[2]
Ein Binärvektor (Bitfeld) entspricht normalerweise einem Maschinenwort (z. B. von 32 bit Länge), das als Ansammlung einzelner Bits interpretiert wird.
Höhere Programmiersprachen bieten darüber hinaus Aufzählungstypen an und gestatten es, vielfältige zusammengesetzte Datenstrukturen (records) zu deklarieren.

Tiefenstrukturen
An sich sind nur Binärvektoren variabler Länge (vom einzelnen Bit an) und Aufzählungstypen zu betrachten. Datenstrukturen vom Aufzählungstyp sind geordnete endliche Mengen. Ist eine solche Menge einmal beschrieben, läßt sich jedes ihrer Elemente durch seine Ordinalzahl codieren.
Auch Zeichen können als Datenstrukturen vom Aufzählungstyp angesehen werden; die geordnete endliche Menge ist dann das jeweilige Alphabet (so ist beispielsweise der übliche ASCII-Zeichensatz im Standard- Package der Sprache Ada definiert). Mit diesem Konzept ist man nicht zwingend an eine starre Byte-

---

1 In älteren Architekturen (z. B. ICL 19ØØ, CDC 36ØØ) waren 6-bit- Codes üblich; einige Maschinen haben bei 36 bit Wortlänge 9- bit- Bytes (Univac 11ØØ, S1).

2 Die Länge der Zeichenkette ist üblicherweise in Deskriptoren (iAPX 432) oder direkt im Befehl (S/36Ø) angegeben; früher waren auch Endemarken üblich (14Ø1, R 3ØØ).

Struktur gebunden, sondern man kann für jedes Alphabet eine jeweils angemessene Länge des Zeichencode- Formates festlegen.[1] Dies dürfte die praktischen Schwierigkeiten mit mehrsprachigen bzw. landesspezifischen Zeichensätzen radikal und elegant beheben.[2]
Alle komplexeren Strukturen (records) sind heterogener Art. Es ist eine Angelegenheit der Implementierung, ob die innere Struktur ausschließlich durch progammseitige Zugriffe repräsentiert wird (prozedurale Darstellung[3]) oder durch zusätzliche strukturell- deskriptive Angaben (deklarative Darstellung[4]).

5.2. Programmstrukturen

In Verallgemeinerung bekannter Ansätze lassen sich alle Programmstrukturen auf 4 Abstraktionen zurückführen:

1. Operatoren liefern Resultate durch Verarbeitung gegebener Argumente
2. Selektoren wählen bestimmte Angaben aus gespeicherten Informationsstrukturen aus
3. Iteratoren liefern nacheinander Folgen von Angaben
4. Aktivatoren bestimmen, unter welchen Bedingungen und in welcher zeitlichen Reihenfolge Operatoren, Selektoren und Iteratoren nacheinander zur Wirkung kommen.

Der Begriff des Operators ist in Mathematik und Informatik seit langem üblich (vgl. beispielsweise /74/).

Der Begriff des Selektors wird in grundlegenden Arbeiten zu abstrakten Objekten verwendet (so in /335/). Hier wird er so aufgefaßt wie in der Programmiersprache Clu: als Abstraktion jeglicher Auswahlvorgänge.

Der Begriff des Iterators wurde in der Programmiersprache Clu als verallgemeinerte Abstraktion für Programmschleifen (for-loops) eingeführt.

Sinngemäß wird der Begriff des Aktivators als zusammenfassende Bezeichnung angewendet, die beispielsweise Verzweigungen, Unterprogrammaufrufe und Unterbrechungen einschließt.

---

1 Es geht hier um die Effizienz der internen Speicherung und Verarbeitung. Wandlungen zu Ein- und Ausgabezwecken bereiten keine Schwierigkeiten; sie sind z. B. durch Blocktransporte mit Tabellenzugriffen ("Move Translated") zu implementieren.
2 Einen Eindruck von diesen Schwierigkeiten gewinnt man anhand der Firmenschriften einschlägiger Gerätesysteme, wie IBM 327Ø oder EC 792Ø.
3 Der Compiler erzeugt mit den Angaben der record- Deklaration unmittelbar die Befehlsfolgen.
4 Der Compiler erzeugt aus der record- Deklaration Deskriptoren (andere Bezeichnung: Dope- Vektoren, z. B. beim PL/1-Laufzeitsystem), die von Unterprogrammen oder von Befehlen (z. B. bei iAPX 432) genutzt werden.

5.2.1. Operatoren

In Tafel 7 sind die Operatoren einer verbreiteten Maschinenarchitektur angeführt.[1]
Tafel 8 zeigt die Operatoren der Programmiersprache Ada.

Tiefenstrukturen
Verglichen mit Programmiersprachen enthalten Maschinenarchitekturen oft mehr elementare Operatoren.[2] Die Tiefenstrukturen werden also durch die Befehlslisten der bewährten Rechner gut repräsentiert. Die Vielfalt der Operatoren läßt sich folgendermaßen ordnen:

1. arithmetische Operatoren (+, -, *, /, Divisionsrest, Vorzeichenwechsel, Vergleich)
2. logische Operatoren, die bitweise unabhängig wirken (AND, OR, XOR usw.)
3. sonstige (einschließlich Transporte, Bittests, Setzen von Einzelbits, Transporte mit tabellengesteuerter Umcodierung, Ermitteln der ersten Eins in einem Bitfeld usw.).

5.2.2. Selektoren

In den üblichen Programmiersprachen ist die Selektion durch Benennung bzw. Indizierung (bei Array- Strukturen) gegeben. Es lassen sich nur Informationsstrukturen selektieren, die in der Sprache definiert sind.
In den Maschinenarchitekturen wird die Selektion durch die Adresssierung verwirklicht. Im allgemeinen ist das Byte die kleinste adressierbare Einheit; Wort- oder Bitadressierung sind wenig verbreitet. In Bild 18 und Tafel 9 sind eingeführte Adressierungsprinzipien dargestellt. Es gibt eine beachtliche Vielfalt von Auffassungen darüber, welche Adressierungsprinzipien in einer Rechnerarchitektur vorgesehen werden sollten; die Verfahren nach Bild 18 bzw. Tafel 9 repräsentieren nur eine dieser Auffassungen. Beachtenswerte Extreme sind:

1. Beschränkung auf die Form "Basisregister + vorzeichenbehafteter 16-bit- Offset". Das wird z. B. in /139/ als ausreichend angesehen, da viele Programme von Ø verschiedene ("non zero") Offsets verwenden.[3]

2. Adressierung unter Nutzung eines Kellerspeichers, wobei ein vollständiger Satz von Adressierungsmodi definiert ist (Tafel 1Ø). Das Ziel besteht darin, Konstrukte höherer Programmiersprachen so direkt wie möglich in die Befehlsliste abzubilden: Beziehungen zwischen Namen im Befehl und temporären Speicherplätzen sollen redundanzfrei und ohne Zwang zur Einführung zusätzlicher Befehle hergestellt werden können.[4]

---

1 VAX 11 (CISC mit besonders reichhaltigem Befehlsvorrat).
2 Deshalb ist es manchmal unumgänglich, auf die Maschinensprache zurückzugreifen.
3 Es ist eine Basisadresse von 32 bit vorgesehen.
4 Das Konzept geht namentlich auf Flynn zurück (/168/, /169/).

Add
Add packed decimal string
Add with carry
Add one and branch
Arithmetic shift
Arithmetic shift and round packed decimal string
Bit clear
Bit set
Bit test
Clear
Compare numeric
Compare character string
Compare packed decimal string
Compare variable bit field to integer
Convert data types
Decrement
Divide
Divide packed decimal string
Extented Divide
Extented Multiply
Extract variable bit field
Find first bit in variable bit field
Increment
Index (calculate array index)
Insert entry into queue
Insert integer into variable bit field
Locate character
Match character string
Move complemented
Move negated
Move numerical quantities
Move address
Move character string
Move packed decimal string
Move translated characters
Move zero extented numerical quantities
Multiply
Multiply packed decimal string
Remove entry from queue
Rotate longword
Scan character string
Span character string
Subtract
Subtract packed decimal strings
Subtract with carry
Subtract one and branch
Test
Exclusive OR

Tafel 7 Operatoren einer verbreiteten Maschinenarchitektur (Maschinenbefehle, die Daten verändern bzw. Bedingungen abfragen)

Logische Operatoren

and **or** **xor**
and then: verkürzte Konjunktion[1]
or else: verkürzte Disjunktion[1]

1 Der 2. Operand wird erst dann ausgewertet, wenn das Ergebnis nicht bereits durch den 1. Operanden zu bestimmen ist.

Relationale Operatoren

= /= < <= > >=
in **not in**: Prüfung, ob ein Wert zu einem Typ oder Subtyp gehört bzw. in einem Bereich (**range**) liegt.

Zweistellige additive Operatoren

\+ -
&: Konkatenation von 2 **arrays**; Erweiterung eines **array** mit einem Element; Bilden eines **array** durch Konkatenation von 2 Elementen

Einstellige additive Operatoren

\+ -

Multiplikative Operatoren

* / **mod** **rem**

Operandenkombination für * und /: integer/integer; float/float; fixed/integer; integer/fixed; fixed/fixed
mod und **rem**: integer/integer

Operatoren mit höchstem Vorrang

abs: Absolutwert
not: logische Negation
**: Potenzierung; Operandenkombinationen: integer/positive integer; float/integer

Tafel 8 Operatoren der Programmiersprache Ada

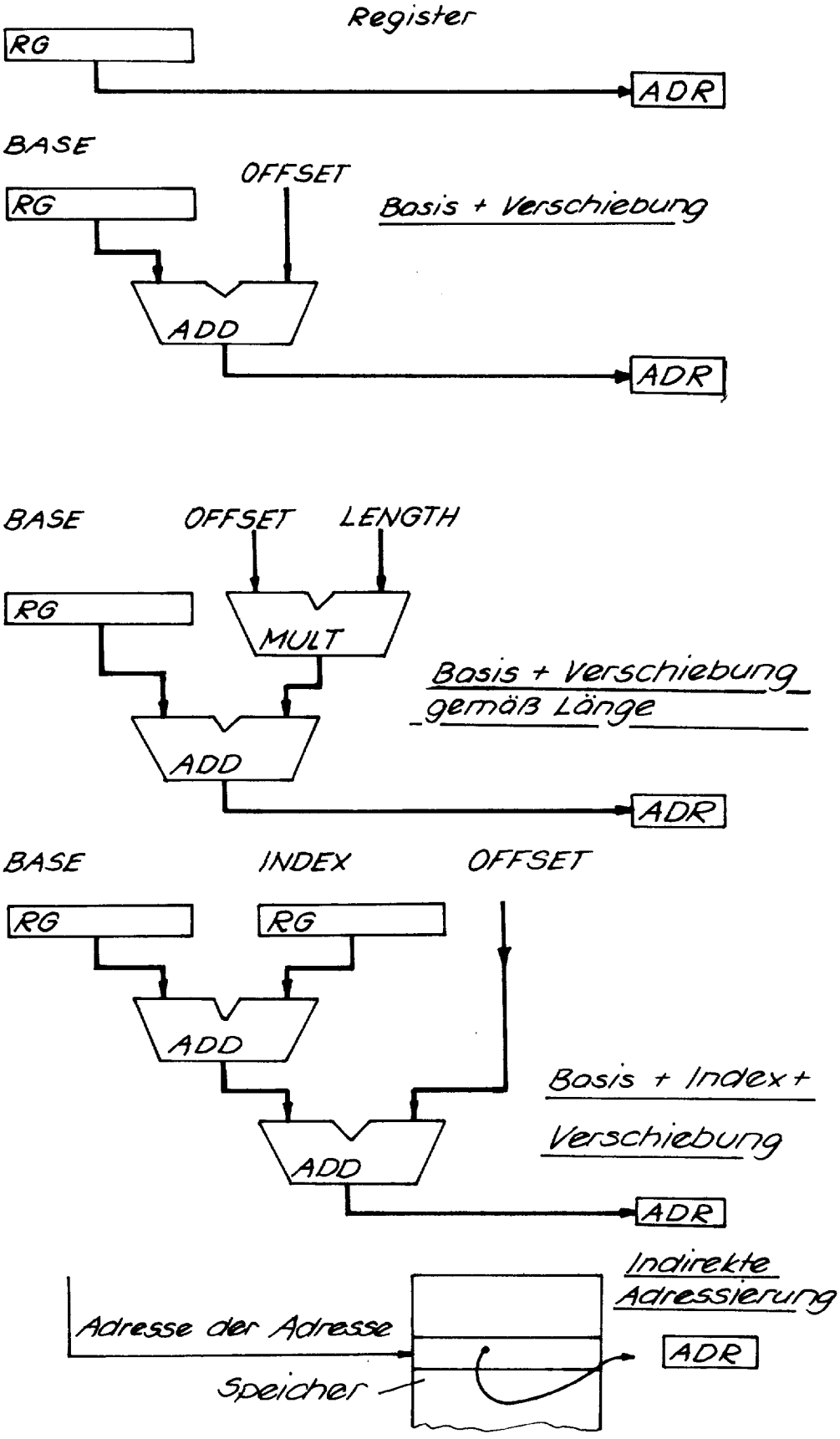


Bild 18 Adressierungsprinzipien

Bezeichnung	Der Operand ist gegeben durch
Short Literal	N (Direktwert im Befehl)
Index	<(b + s * <Rn>)>
Register	<Rn>
Register deferred	<<Rn>>#
Autodecrement	<<Rn>>; zuvor <Rn> := <Rn>-s #
Autoincrement	<<Rn>>; danach <Rn> := <Rn>+s #
Autoincrement deferred	<<<Rn>>>; danach <Rn> := <Rn>+s #
Displacement	<(<Rn> + D)> #
Displacement deferred	<<(<Rn> + D)>> #
- Adressierung auf Befehlszähler bezogen (R 15) -	
Immediate	N
Absolute	<A>
Relative	<(<PC> + D)>
Relative deferred	<<(<PC> + D)>>

<...>	Inhalt von...
Rn	Register
N	Direktwert
A	Absolutadresse (im Befehl)
s	Operandengröße (1, 2, 4, 8, 16 Bytes)
D	Displacement (Byte, Wort , Langwort)
b	Basisadresse des Index Mode
#	An die Adressenangabe im Befehl kann noch ein Indexregister (für den Index Mode) angefügt werden.

Tafel 9 Adressierungsverfahren einer verbreiteten Maschinenarchitektur

Format	Verknüpfung	Explizite Operanden
A B C	A op B -> C	3
A B B	A op B -> B	2
A B A	A op B -> A	2
A A A	A op A -> A	1
A B T	A op B -> T	2
A T B	A op T -> B	2
T A B	T op A -> B	2
T A A	T op A -> A	1
A T A	A op T -> A	1
A A T	A op A -> T	1
A T T	A op T -> T	1
T A T	T op A -> T	1
T T A	T op T -> A	1
T U A	T op U -> A	1
T U T	T op U -> T	Ø

A, B, C explizite Operanden
T erste Stack- Position (Top of Stack)
U zweite Stack- Position (auf T folgend)

Besondere PUSH- und POP- Befehle sind nicht nötig:

- T als Resultat veranlaßt automatisch PUSH
- T als Argument veranlaßt automatisch POP.

Tafel 10 Ein vollständiger Satz von Adressierungsmodi (nach Flynn)

3. objektorientierte Zugriffsorganisation; in den Bildern 19 und 2Ø ist ein Beispiel veranschaulicht.[1] Zur Implementierung höherer Programmiersprachen sind solche Zugriffsverfahren in irgendeiner Form stets notwendig (je nach Sprachkonzept mit unterschiedlicher Bedeutung und Nutzungshäufigkeit): wenn die Maschinenarchitektur sie nicht zur Verfügung stellt, müssen sie im Laufzeitsystem programmtechnisch vorgesehen werden.[2]

Tiefenstrukturen
Letztlich geht es stets darum, für jeden Zugriff eine einzige Angabe zum Speicher zu liefern, die faktisch die Ordinalzahl der gewünschten elementaren Informationsstruktur darstellt. Diese Ordinalzahl kann nach verschiedenen Algorithmen ermittelt werden, wozu vornehmlich Additionen und Tabellenzugriffe gehören.

5.2.3. Iteratoren

Einen Überblick über Iterator- Konstrukte in verbreiteten Programmiersprachen gibt Tafel 11.
In der Sprache Clu ist die Iterator- Abstraktion explizit enthalten. So kann in den eigentlichen Programmschleifen davon abstrahiert werden, wie die Angaben bei aufeinanderfolgenden Durchläufen herangeschafft bzw. abgespeichert werden.
In Maschinenarchitekturen sind meist nur recht elementare Iteratoren vorgesehen, z. B.:

- Blockoperationen (Transporte und Vergleiche)

- Wiederholungsbefehle[3]

- Blocktransporte für Graphik- Bitmuster: Bitblock- Transporte BITBLT, CLIP (nur Null- Werte werden geschrieben), DRAW (nur von Null verschiedene Werte werden geschrieben), Transporte mit Hintergrund- Unterdrückung (Werte, die kleiner sind als der aktuelle Inhalt der jeweiligen Zielposition, werden nicht geschrieben) usw.[4]

- automatische Adressenrechnungen im Rahmen der Befehlsausführung, wobei die Inhalte von Adressenregistern geändert werden (Autoincrement, Autodecrement usw.). Besonders bedeutsam sind Zugriffe zu Kellerspeichern (Stacks): meist ist PUSH ein "Predecrement" vor dem Schreiben, POP ein "Postincrement" nach dem Lesen.[5]

---

1 Die Bilder betreffen iAPX 432. Solche Prinzipien gibt es auch in anderen Architekturen (z. B. B 55ØØ...69ØØ).
2 Zur Geschwindigkeit vgl. S. 2Ø, Punkt 6.
3 Beispiel: Befehl DJNZ des Mikroprozessors Z 8Ø.
4 Neuerdings in verschiedenem Umfang vergegenständlicht (Intel 8Ø86Ø, "Transputer" T 8ØØ, Stellar u. a.).
5 Bei manchen Architekturen mit jedem Universalregister ausführbar, bei manchen nur mit den Stackpointer- Registern. Die Überwachung der Bereichsgrenzen ist erst neuerdings vergegenständlicht (68 Ø3Ø).

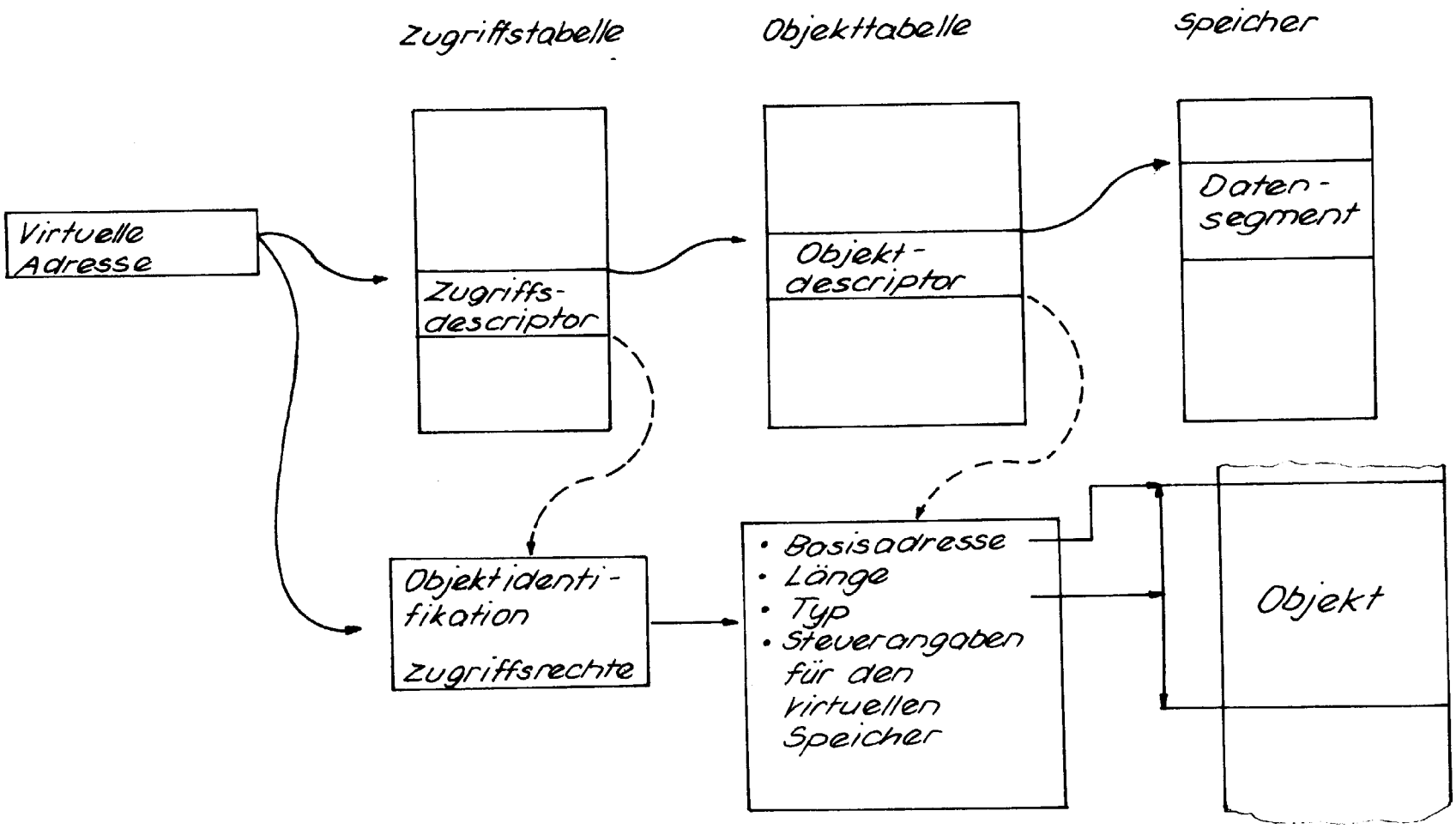


Bild 19 Prinzip der objektorientierten Zugriffsorganisation (Bsp. iAPX 432)

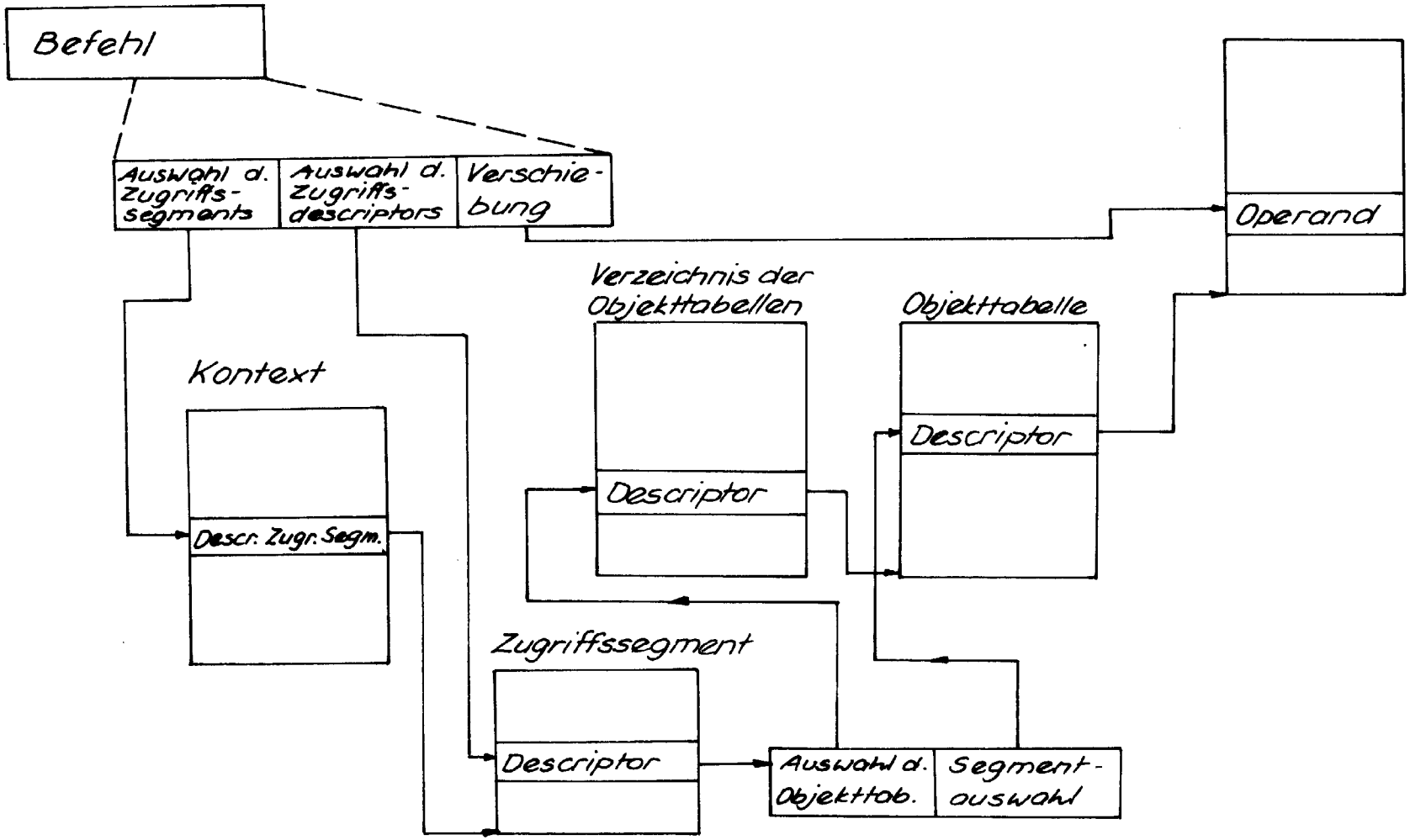


Bild 20 Schema der Adressierung eines Operanden beim Mikroprozessor iAPX 432

PL/1

```
DO

DO WHILE (Bedingung)

DO Variable = Anfangswert TO Endwert BY Schrittweite
```

Ada

```
while Boolescher Ausdruck loop
                .....

end loop

for Variable in (reverse) Bereich loop
              .....

end loop

loop
 .....

end loop
```

C

```
while (Ausdruck)
      Anweisung...

for (Initialisierung, Endebedingung, Zählanweisung)
Anweisung...

do
Anweisung...
while (Ausdruck)
```

Tafel 11 Iterator- Konstrukte in verbreiteten Programmiersprachen

• Vektoroperationen. Tafel 12 gibt einen Überblick über solche Operationen, die in einem Spezialprozessor[1] vorgesehen sind; weitere Anregungen sind aus den Tafeln 13 - 15 ersichtlich. Tafel 13 zeigt einige elementare Schleifen, die beim numerischen Rechnen häufig verwendet werden (nach /29Ø/). In Tafel 14 sind jene Muster für den Zugriff auf zusammengesetzte Strukturen aufgezählt, die in der numerischen Analysis am meisten gebraucht werden (nach /199/).[2] Tafel 15 wurde nach den Vorschlägen in /61/ zusammengestellt, die von der Sprache APL beeinflußt sind.

Tiefenstrukturen
Für einen Iterator ist kennzeichnend, auf welche Weise die aufeinanderfolgenden Angaben selektiert werden; der Iterator ist praktisch ein Algorithmus zum Erzeugen von Selektionsangaben. Solche Algorithmen können aus der Sicht des Anwenders recht komplex ausfallen, so daß tatsächlich nur ganz wesentliche Abläufe vergegenständlicht werden können. Diese betreffen in der Regel das Erhöhen bzw. Vermindern von Adressenangaben, wobei die Endebedingung durch das Erreichen eines Endwertes, durch Ausschöpfen eines Zählwertes oder eine eine in der Schleife ermittelte Bedingung (z. B. durch arithmetischen Vergleich) gegeben ist. Als elementare Iteratoren bieten sich an:

• Blockoperationen mit konstanter Adressenerhöhung bzw. -verminderung (auch besondere; beispielsweise Zeichenketten-Transporte mit Umcodierung, Transporte für Graphik- Bitmuster, Blockvergleiche)

• elementare Schleifen, wie sie beim numerischen Rechnen gebräuchlich sind (vgl. die Tafeln 12- 15)

• Zugriffsprinzipien für Kellerspeicher (LIFO) und Warteschlangen (FIFO).

5.2.4. Aktivatoren

Es ist zwischen synchronen und asynchronen Aktivatoren zu unterscheiden. Synchrone Aktivatoren sind Verzweigungen verschiedener Art, einschließlich der Unterprogrammaufrufe, der Supervisorrufe und der programmseitig ausgelösten Unterbrechungen; asynchrone Aktivatoren sind externe Unterbrechungen oder Maßnahmen zur Behandlung unvorhersehbarer Ausnahmebedingungen.
In Programmiersprachen sind synchrone Aktivatoren in Form der Prozedur- bzw. Funktionsaufrufe und in Form von Konstrukten wie IF...THEN...ELSE, CASE..., GOTO... u. a. gegeben. Asynchrone Aktivatoren sind in vielen Programmiersprachen nicht ohne weiteres zugänglich. Manche Sprachen (z. B. PL/1 und Ada)

1 IBM 2938.
2 Tafel 14 soll dazu anregen, sowohl die Speicherorganisation als auch die vergegenständlichen Iteratoren anwendungsgerecht auszugestalten.

1. Elementweise Multiplikation von Vektoren

   $Y(i) := X(i) * U(i)$

2. Elementweise Addition von Vektoren
   $Y(i) := X(i) + U(i)$

3. Skalarprodukt von Vektoren

   $Y := \sum_i X(i) * U(i)$

4. Summe der Elemente eines Vektors

   $Y := \sum_i X(i)$

5. Summe der Quadrate der Elemente eines Vektors

   $Y := \sum_i X(i) * X(i)$

6. Multiplikation mit Faltung

   $Y(i) := \sum_j Uj * X(i + j - 1)$

Tafel 12

Elementare Vektoroperationen eines Spezialprozessors

```
A(i) := 1                          A(i) := sin (B(i))
A(i) := B(i)                       A(i) := arc sin (B(i))
A(i) := B(i) + 1Ø                  A(i) := abs (B(i))
A(i) := B(i) + C(i)
A(i) := B(i) * 1Ø                  Umordnen:
A(i) := B(i) / 1Ø                  {C(i) := A(i);
A(i) := B(i) / C(i)                {A(i) := B(i);
A(i) := max (B(i), C(i))           {B(i) := C(i).
A(i) := B(i) * C(i) + D(i)
A(i) := B(i) * C(i) + D(i) * E(i)
```

Tafel 13

Elementare Schleifen des numerischen Rechnens

1. eine Zeile einer Matrix

2. eine Spalte einer Matrix

3. die Hauptdiagonale einer quadratischen Matrix

4. der Zeilenabschnitt einer quadratischen Matrix, der der oberen Dreiecksmatrix entspricht ("half row")

5. der Zeilenabschnitt einer quadratischen Matrix, der der unteren Dreiecksmatrix entspricht ("half row")

6. der Vektor, der durch die geraden Elemente eines Vektors gebildet wird

7. der Vektor, der durch die ungeraden Elemente eines Vektors gebildet wird

8. die transponierte Matrix

9. die "Fläche" eines Würfels

1Ø. eine Matrix, die durch Extraktion der ungeraden Elemente der ungeraden Zeilen einer Matrix gebildet wird

11. Untermatrizen

Tafel 14

Wichtige Zugriffsmuster für das numerische Rechnen

Operationen an binären Vektoren

FIRST OCCURRENCE: Index der ersten Eins

NUMBER OF OCCURRENCES: Anzahl der Einsen; "Quersumme"

ALL OCCURRENCES: Index- Vektor, der alle mit Eins belegten Positionen des Binärvektors angibt

AND; OR: Logische Operationen "quer" über alle Komponenten des Vektors

Operationen mit beliebigen Vektoren

SUBVEC: Auswahl eines Teilvektors

SAMPLE: Auswahl aller Komponenten des Quellvektors, die einen bestimmten Abstand zueinander haben

SELECT: Auswahl von Komponenten aus einem Vektor über eine Indexmenge

COMPRESS: Auswahl von Elementen des Quellvektors über einen binären Maskenvektor

EXPAND: Gegenteil von COMPRESS; der binäre Maskenvektor hat die Länge des Zielvektors und zeigt an, an welchen Stellen die Komponenten des Quellvektors eingefügt werden. Der Zielvektor ist anfänglich gleich Null.

MERGE: Mischen zweier Vektoren gemäß einem binären Vektor: eine Eins veranlaßt die Übernahme der jeweiligen Komponente aus dem ersten Quellvektor, eine Null sinngemäß aus dem zweiten

GATHER: Zusammenbau eines Vektors mit den Angaben einer Indexmenge

Weitere Operationen

DECLARE:	erzeugt einen Vektor
INSERT:	fügt eine Komponente in einen Vektor ein
DELETE:	löscht eine Komponente in einem Vektor
SWAP:	tauscht 2 Komponenten eines Vektors aus
ALTER:	ändert den Wert einer Komponente
CONCATENATE:	verkettet 2 Vektoren zu einem neuen
LENGTH:	gibt die Länge des Vektors zurück
OCCURS:	gibt Quersumme bzw. Index zurück

Tafel 15 Vorschläge für elementare Vektoroperationen

gestatten es, die Ausnahmebehandlung zu steuern; Konstrukte zur Reaktion auf externe Bedingungen (bezogen auf den aktuellen Programmablauf) sind nur in wenigen Sprachen (z. B. Ada) oder in herstellerspezifischen Spracherweiterungen (z. B. Burroughs Algol) zu finden.
Synchrone Aktivatoren in Maschinenarchitekturen sind zumeist Verzweigungen und Unterprogrammaufrufe, wobei wesentliche Unterschiede in den Einzelheiten der Adressenrettung und der Parameterübergabe bestehen. So wird beim System/360 ein jeweils explizit anzugebendes Register für die Adressenrettung genutzt; mit diesem ist das aufrufende Programm adressierbar, so daß Parameter beispielsweise dem Aufrufbefehl nachgeordnet werden können. In neueren Architekturen wird das Stack- Prinzip bevorzugt, und es sind manchmal komplexe Abläufe zur Rettung des Verarbeitungszustandes und zur Parameterübergabe vergegenständlicht.[1] In manchen Architekturen sind mehrere Zustandsebenen vorgesehen (meistens 2: für Anwender und System; teils sind vollständige umschaltbare Registersätze vorhanden, teils für jede Ebene nur ein Stackpointer und ein Flagregister).

Asynchrone Aktivatoren sind gegeben durch die verschiedenen Formen der Unterbrechungs- und Ausnahmebehandlung, die meist auf dem Kellerspeicherprinzip, auf dem Prinzip des Umschaltens von Registersätzen oder des Austauschs von Statusworten beruhen.

Tiefenstrukturen
Aus der Sicht der Algorithmischen Logik[2] kommt man für ein zusammengesetztes Programm mit 3 Formen synchroner Aktivatoren aus:

1. Aneinanderreihung P1;P2

2. Verzweigung if B then P1 else P2

3. Schleife while B do P1.

(P1 und P2 sind Programme; B ist ein prädikatenlogischer Ausdruck.)
In den meisten Rechnerarchitekturen ist die Ausführungsreihenfolge durch Befehlszählung vergegenständlicht[3]; in den meisten Programmiersprachen gibt die Reihenfolge der Anschreibung die Reihenfolge der Ausführung an. Verzweigungen bedeuten eine Änderung der Bearbeitungsreihenfolge, also ein Abweichen von der vergegenständlichten Befehlszählung durch Wirksammachen eines codierten Selektors für den Folgebefehl; ohne Rettung des Maschinenzustandes.
Unterprogrammaufrufen und asynchronen Aktivatoren ist gemeinsam, daß beim Eintritt der Maschinenzustand gerettet und beim Austritt gezielt wiederhergestellt wird (Rückkehr zum verlas-

1 Z. B. CALLG und CALLS bei VAX 11.
2 Für einen einführenden Überblick s. /274/, S. 293-326.
3 Die explizite Angabe von Folgebefehlen (ohne Befehlszählung) ist in manchen Mikroprogrammsteuerungen üblich.

senen Zustand oder Übergang zu einem neuen[1]). Asynchrone Aktivatoren erfordern zusätzlich übergeordnete Steuerungsmaßnahmen (z. B. Erlaubnis bzw. Verhinderung der Annahme). Eine wirkungsvolle und elegante Abstraktion ist das "Ereignis" ("event").[2]

5.3. Speicherung

Dem Stand der Technik entsprechend sind Rechner mit folgenden Speichermitteln ausgestattet:

1. Flipflops bzw. aus diesen gebildete Hardware- Register in den Schaltungsanordnungen
2. Registerspeicher
3. Schnellzugriffsspeicher (Cache)
4. Arbeitsspeicher (RAM)
5. Massenspeicher (vorzugsweise Magnetplatten).

Diese Speichermittel sind in Maschinenarchitekturen bzw. Programmiersprachen auf unterschiedliche Weise zugänglich. Maschinenarchitekturen sind meist auf Grundlage des Registerspeichers und der Adressierungsprinzipien des Arbeitsspeichers definiert. In Hochleistungsmaschinen sind oft alle architekturseitig definierten Register auch als solche in der Hardware ausgeführt; ansonsten unterscheidet sich die Registerstruktur der Hardware deutlich von jener der Architekturdefinition, namentlich dann, wenn die Architektur mikroprogrammtechnisch emuliert wird.[3] Die Massenspeicher sind üblicherweise nur über das Ein/Ausgabe- Subsystem zugänglich.[4] Sie werden in modernen Systemen zweifach genutzt:

1. als Mittel zur Implementierung virtueller Speicher
2. als Mittel zur Implementierung von Dateisystemen; in diesem Sinne sind sie von höheren Programmiersprachen aus (indirekt) zugänglich.[5]

---

1 Beispiele für Übergänge zu einem neuen Zustand: Aufrufen einer anderen Task, Fortsetzen mit einem Ausnahmebehandler.

2 Z. B. in PL/1 und Burroughs Algol (/88/) definiert und in Realzeit- Betriebssystemen genutzt (/234/, /237/, /249/).

3 So sind in einer typischen Implementierung (EC 1040, /40/) die architekturseitigen 16 Universalregister (32 bit) und 4 Gleitkommaregister (64 bit) in einem Speicherarray zusammengefaßt; die eigentliche Verarbeitungshardware enthält 4 Register zu 64 bit, 5 Register zu 8 bit sowie 3 Adressen- bzw. Längenzähler.

4 Auch wenn die entsprechenden Steuereinheiten direkt über den Speicheradressenraum zugänglich sind ("memory mapped I/O"), bedarf der Betrieb der Massenspeicher der programmseitigen Unterstützung (Gerätetreiberroutinen).

5 Das dem Nutzer zugängliche Dateisystem abstrahiert von physischen Geräten und Datenträgern ("logische" Dateiorganisation). Die Verbindung zwischen logischen und physischen Dateien wird beispielsweise über eine "Jobsteuersprache" hergestellt.

Die gespeicherten Angaben sind zu technisch beherrschbaren Verpackungseinheiten fester Länge zusammengefaßt (vom Byte bis zum Datenblock des Dateisystems). Tafel 16 gibt einen Übersicht über solche Verpackungseinheiten.

Tiefenstrukturen
Alle gespeicherten Informationsstrukturen müssen:

- sicher gehalten werden (das betrifft die Bereitstellung von Speicherplatz und den elementaren Schutz gegen Überschreiben; technische und übergeordnete organisatorische Fragen werden hier vernachlässigt)

- schnell auffindbar sein (betrifft die Selektionseinrichtungen)

- schnell zur Verarbeitung bereitgestellt oder (als Resultate) eingespeichert werden (betrifft die Arbeitsgeschwindigkeit der Speichermittel).

Bei der Verarbeitung stehen letztlich alle Angaben in Registern, die direkt mit den Verknüpfungsschaltungen verbunden sind. Aus dieser Sicht sind alle zwischen Registern und Massenspeichern liegenden Ebenen lediglich erforderlich, um mit den stets beschränkten technischen Mitteln die Forderungen nach praktisch unbegrenztem Speichervermögen näherungsweise erfüllen zu können.
Neben dem Speichervermögen an sich ist der Codierung der gespeicherten Information besondere Aufmerksamkeit zu schenken: der technische Fortschritt auf dem Gebiet der Speicher darf nicht zu der Annahme verleiten, Bemühungen um sparsame, dichte Codierungen seien nicht mehr notwendig. Dazu einige Gesichtspunkte:

1. Große Speicher haben naturgemäß größere Zugriffszeiten als kleine; die aktuell zu bearbeitenden Angaben sollten deshalb in möglichst kleine Speicher passen (Register, Cache).

2. Je länger die Adresse, um so mehr Adressenbits müssen in Befehlen und strukturell- deskriptiven Angaben (z. B. Objektdeskriptoren) mitgeführt werden, um so länger dauert die Adressenrechnung (z. B. Offset- Addition).

3. Die Datenwege bestimmen letzten Endes - bei zweckmäßigster Ausgestaltung aller Verarbeitungs- und Steuerschaltungen - die Leistung der Maschine: es ist also danach zu trachten, im Rahmen des technisch gegebenen Leistungsvermögens ("bit/s") ausschließlich signifikante Information (d. h.: effektive Bits, vgl. Abschnitt 4.1., S. 51 f.) zu übertragen (wenn eine Angabe einen von n möglichen Werten repräsentiert, sollte dies nur CEIL (ld n) Bits erfordern).

4. Viele wesentliche Anwendungsalgorithmen haben einen NP-vollständig mit der Problemgröße wachsenden Speicherbedarf; deshalb dürften auch künftig die Wünsche der Anwender den technischen Möglichkeiten vorauseilen.

Verpackungseinheit	Größe
Byte	8 Bit (selten 9)
Maschinenwort	16, 32 o. 64 Bit (selten 24, 36, 48, 6Ø)
Universal-registersatz	8 - 32 Maschinenworte (häufig 16)[1]
Cache- Block	64 - 512 Bytes
Cache insgesamt	4 - 64 kBytes
Segment im Arbeitsspeicher	64k - 1M Bytes
Arbeitsspeicher insgesamt	64k - 256M Bytes (ständig zunehmend)
Seite im virtuellen Speicher	512, 1k, 2k, 4k Bytes
virtueller Speicher insgesamt[2]	16M, 2G, 4G Bytes[3]
Datenblock im Massenspeicher	128 - 1k Bytes (UNIX: 512 Bytes)

1 Extremes Beispiel: 192 Register (AMD 29ØØØ).
2 Manche Betriebssysteme unterstützen mehrere unabhängige virtuelle Speicher in einem Prozessor.
3 Die Angaben entsprechen virtuellen Adressen zu 24, 31 bzw. 32 bit; zunehmend werden in neuen Architekturen virtuelle Adressen von 48 bzw. 64 bit vorgesehen.

Tafel 16 Übersicht: Verpackungseinheiten auf verschiedenen Ebenen der Speicherhierarchie

6. Vergegenständlichte Abstraktionen

Aus der Vielfalt der Tiefenstrukturen sind bestimmte Abstraktionen auszuwählen, um sie in Schaltungsanordnungen zu vergegenständlichen.
Diese Abstraktionen müssen hinreichend bedeutsam sein (universelle Anwendbarkeit, hohe Nutzungshäufigkeit), und es muß möglich sein, beherrschbare, leistungsfähige Schaltungsstrukturen dafür anzugeben. Im Interesse der praktischen Durchsetzbarkeit und Akzeptanz müssen bewährte, weit verbreitete Konzepte im Rahmen der Vergegenständlichungen implementiert werden können. Die Abstraktionen sind sorgfältig auszuwählen und aufs genaueste festzulegen: höchstes Leistungsvermögen erfordert direkte Umsetzung in Schaltmittel, ohne zwischengeordnete Ebenen der programmierten Steuerung. Ziel ist der Hochleistungsrechner, nicht der universelle Emulator; diesem Ziel ist es äußerst abträglich, wenn konzeptionelle Unzulänglichkeiten durch Programmierung nachgebessert werden müssen.

Dazu wird folgender Ansatz gewählt:

1. Alle Festlegungen werden ohne Rücksicht auf Vorhandenes mit Blick auf Leistungsvermögen und technische Durchführbarkeit getroffen.

2. Im Rahmen dieser Festlegungen werden Rückzugsmöglichkeiten auf Bewährtes angeboten. Beispiele dafür sind aus Tafel 17 ersichtlich. Beispielsweise wird angestrebt, auf virtuelle Speicherorganisation und Dateiverwaltung im herkömmlichen Sinne zu verzichten (stattdessen: konsequente Objektorientierung). Die technischen Mittel können aber ohne weiteres genutzt werden, die herkömmlichen Konzepte zu implementieren; einerseits zur Anpassung an gegebene Anwendungs- und Programmierumgebungen, andererseits als Rückzugsmöglichkeit, falls sich die neuen Prinzipien als nicht hinreichend überlegen erweisen.

Für die neuen Architekturdefinitionen gilt es, von wirtschaftlich erfolgreichen Rechnerarchitekturen[1] zu lernen und einige praxisbezogene Gesichtspunkte zu beachten:

1. Jede Code- bzw. Formatfestlegung sollte hinreichend großzügig bemessen sein, beispielsweise so, daß wenigstens das 4-fache des größten Wertes, der als technisch- ökonomisch sinnfällig angesehen wird, noch codiert werden kann (das betrifft z. B. Adressenformate).

2. Jede Code- Festlegung sollte eine Ausweichvorkehrung ("escape") haben, um künftige Erweiterungen zu ermöglichen (das betrifft z. B. Deskriptor- und Steuerwortformate; dort sollte wenigstens ein Escape- Bit vorgesehen werden).

---

1 Die folgende Darstellung wurde im wesentlichen von Schriften und Erzeugnissen der Fa. IBM angeregt (bes. S/360 und S/370 einschließlich peripherer Geräte).

Beispiele für Innovationen	Rückzug auf ...
Allgemeine Operationen über Binärvektoren	Wortverarbeitung; Bitfeld-operationen gemäß Stand der Technik
Universelle Hochgenauigkeits- Numerik	Gleitkomma- Numerik, z. B. nach IEEE 754; BCD- Numerik (S/37Ø bzw. VAX)
Datenflußsteuerung von Operationswerken	herkömmliche Mikro-programmsteuerung
Objektorientierte Daten-organisation	herkömmliches Laufzeit-system (softwareseitig); Variablenbindung zur Compilierzeit
neuartige Speicherver-waltung dazu	bekannte Prinzipien des virtuellen Speichers
Maschinenworte zu 96 bit (mit zusätzlichen TAG- Bits)	Maschinenworte zu 32 bzw. 64 bit
heterogene Schnellspei-cheranordnungen	herkömmliche Befehls- und Daten- Caches
relationale Wissensbasis	herkömmliches Dateisystem (z. B. UNIX)
neuartige Steuerprinzi-pien bzw. Befehlsformate	herkömmliche Mikropro-grammsteuerung; ggf. Über-nahme einer eingeführten Befehlsliste

Tafel 17 Beispiele für Innovationen und Rückzugsmöglichkeiten auf Bekanntes

3. Sind in einer Architektur Bezüge zu maschinenspezifischen Hardware- Ressourcen notwendig, so sollten diese regulär ausgestaltet werden (s. etwa den recht eleganten Ansatz beim System/37Ø, die vielen irregulären Angaben zur Steuerung der Fehlerbehandlung, zur Adressenumsetzung usw. in einem regulären Satz von 16 Steuerregistern zusammenzufassen).

4. Hardware- Ressourcen (technische Einrichtungen h in der Ressourcenbeschreibung ℜ) sollten wohlüberlegt vorgesehen werden, das Prinzip "Viel hilft viel" ist keineswegs angebracht. Wichtige Einflußgrößen sind: Funktionsvielfalt, Zykluszeit, Nutzungshäufigkeit, Kosten, Wechselwirkungen mit anderen konzeptionellen Festlegungen und "scalability" (die Möglichkeit, bei voller Kompatibilität leistungs- und kostenseitig abgestufte Hardware- Modelle in verschiedenen Technologien fertigen zu können). Beispielsweise ist eine große Anzahl von Universalregistern auf den ersten Blick sinnvoll. Eine solche Festlegung bedeutet aber u. a. mehr Chipfläche und mehr Adressenbits in den Befehlen; sie führt zu einem längeren Maschinenzyklus[1], und sie ist nur in Technologien mit hohem Integrationsgrad zu implementieren[2].

5. Die aktuelle Ausstattung der Hardware sollte programmseitig abfragbar sein (z. B. mit einem Befehl "Laden Maschinenidentifikation").

Im Sinne von Anschaulichkeit und Kürze werden nachfolgend verschiedene Angaben zahlenmäßig festgelegt. Diese Festlegungen haben den Charakter von Beispielen, sie können beliebig geändert werden und beeinträchtigen die Allgemeinheit nicht.
Die Vorstellungen für eine künftige Architekturdefinition werden zunächst im Überblick (ohne nähere Begriffserklärungen) dargestellt:

Datenstrukturen
Die einzige elementare Datenstruktur ist der Binärvektor. Ein Binärvektor kann zwischen 1 Bit und 64k-1 bit lang sein und als numerische oder als nichtnumerische Angabe interpretiert werden. Numerische Datenstrukturen werden stets auf natürliche bzw. ganze Binärzahlen zurückgeführt.

Codierungen
Es wird ausschließlich die binäre Codierung verwendet, und zwar wahlweise je nach Zweckmäßigkeit:

- durch Aneinanderreihen von Bits in einem Binärvektor
- durch binär codierter Ordinalzahlen in endliche Mengen
- durch Kombination dieser Möglichkeiten ("Gleitkommadarstellung")
- durch Strukturbeschreibung
- durch ternäre Angaben.

---

1 Nach /158/ bedeutet Verdoppeln des Registerspeichers eine Verlängerung des Maschinenzyklus um 3Ø% (Erfahrungswert).
2 Alternative: Auslagerung in externe Speicherbereiche (wie bei 36Ø/3Ø), d. h. aber beträchtliche Leistungsminderung.

Eine binär codierte Ordinalzahl von n bit kann ein Element aus einer Menge der Mächtigkeit 2^n auswählen. Ein solcher Binärvektor von n bit wird als Codon der Länge n bezeichnet.

Logische Datenorganisation
Die grundlegende Abstraktion ist das Objekt. Jedes Objekt wird durch seine Ordinalzahl in der geordneten Menge aller Objekte bezeichnet. Für jedes Objekt gibt es strukturell- deskriptive Angaben, die seine Position in den technischen Speichermitteln, seine Größe und seinen inneren Aufbau beschreiben.

Speicherorganisation
Es ist eine Hierarchie von Speichermitteln verschiedener Zweckbestimmung vorgesehen. Die grundlegende Abstraktion zu deren Verwaltung ist der Behälter. Der kleinste Behälter ist das einzeln adressierbare Maschinenwort, hier allgemein als Eimer bezeichnet. Es gibt Behälter verschiedener Größe; das Sortiment ist in der Architektur festgeschrieben. Jedes Objekt belegt genau einen Behälter. Alle Objekte, die erhalten bleiben sollen, gehören einer gemeinsamen Datenbasis an; es gibt kein Dateisystem im herkömmlichen Sinne.

Ablauforganisation
Alle Algorithmen werden als funktionelle Zuweisungen aufgefaßt. Die Vergegenständlichungen bilden die unterste Ebene solcher Zuweisungen. Alle Parameter werden durch ihren Wert übergeben ("call by value"). Sie werden in den jeweiligen Speichermitteln zur Ausführung der Zuweisung bereitgestellt (Register, Schnellspeicher, Speichermittel der Laufzeitumgebung).

Programmorganisation
Die grundlegenden Abstraktionen sind Operatoren, Selektoren, Iteratoren und Aktivatoren. Sie sind für die meistbenutzten wesentlichen Aktionen vergegenständlicht.
Elementare Operatoren betreffen numerische und nichtnumerische Operationen über Binärvektoren wählbarer Länge. Aus beliebigen Binärvektoren können beliebige Subvektoren ausgewählt, verknüpft und in beliebige Binärvektoren zurückgeschrieben werden. Mit den numerischen Grundoperatione können beliebige arithmetische Systeme implementiert werden (ganze Zahlen, Näherungsdarstellungen reeller Zahlen, rationale Zahlen).

Diese Ansätze werden im folgenden genauer beschrieben.

6.1. Prinzipien der Codierung

Alle Angaben werden ausschließlich binär codiert. Die Informationsstrukturen werden systematisch vom Bit an aufgebaut und grundsätzlich als geordnete Mengen betrachtet.[2]
Es ist wesentlich, mit wenig Bits auszukommen: kann eine Angabe jeweils einen von a Werten annehmen, so sollen dafür möglichst nur CEIL (ld a) Bits erforderlich sein.

---
1 Das Prinzip geht auf Zuse zurück (/1ØØ/, /1Ø1/).

Eine Aneinanderreihung von n Bits, mit der einer von 2^n Werten codiert ist, wird als Codon bezeichnet.[1]
Jede (gespeichert vorliegende) Aneinanderreihung von n Bits kann als Belegung eines binären Raumes B^k (mit k = CEIL (ld n)) aufgefaßt werden.[2] Davon ausgehend können systematisch die Möglichkeiten untersucht werden, eine bestimmte Belegung des Raumes B^k anzugeben:

1. Die Belegung wird durch einen Binärvektor der Länge 2^k angegeben, in dem jede in B^k belegte Position mit einer 1 gekennzeichnet ist.

2. Die Belegung wird durch eine Liste der Codons angegeben, die die belegten Punkte in B^k bezeichnen.

3. Die Belegung wird durch einen Lambda- Ausdruck angegeben (Algorithmus mit aktuellen Parametern), der bei Abarbeitung eine Darstellung nach 1. oder 2. erzeugt.

Hier interessiert die explizite Angabe (1. oder 2.), und es gilt, technisch sinnvolle kurze Codierungen zu finden. Dabei geht es nicht um die Codierung des Anwendungsproblems, sondern darum, daß bereits gegebene Codierungen weiter zu verdichten sind.[3]

Zunächst wird untersucht, ob ein Codon der Länge k weiter verkürzt werden kann. Dazu ist eine isomorphe Abbildung von Binärvektoren der Länge k in Binärvektoren der Länge m (m < k) zu finden. Das gelingt in folgenden Fällen:

1. Es gibt nur vergleichsweise wenige, aber bekannte Codons (d. h. es ist nur ein Unterraum in B^k belegt). Dann läßt sich jede Angabe eines solchen Codons ersetzen durch dessen Ordinalzahl in der Menge aller zulässigen Codons. Diese Menge muß explizit gespeichert werden. n Codons zu k bit werden so ersetzt durch

- die Repräsentation der Menge selbst (wenigstens k * n bit)
- strukturell- deskriptive Angaben, die die Längen und Bereiche von n und k sowie die Abbildung nach Strukur und Position in den Speichermitteln beschreiben
- jede Angabe eines Codons von k bits durch ein kürzeres Codon, das eine Ordinalzahl der Länge CEIL (ld n) Bits darstellt.

---

1 Die Terminologie geht auf die recht einprägsamen Begriffsbildungen der Molekularbiologie zurück (vgl. etwa /67/, /96/).
2 Zu den theoretischen Grundlagen s. /52/.
3 Daß der Nutzer sein Problem im Rahmen der jeweiligen Sprachmittel zweckmäßig codiert, wird als gegeben betrachtet; hier geht es darum, welche Vorkehrungen in der Maschinenarchitektur zu treffen sind, um die Problem- Codierung in kompakte binäre Darstellungen isomorph wandeln zu können.

Werden die strukturell- deskriptiven Angaben vernachlässigt, so lohnt sich das Verfahren, wenn für b Codons gilt:

$$b \cdot CEIL(ld\ n) + nk < bk.$$

(Man sieht sofort, daß b deutlich größer sein muß als n; die einzelnen Codons müssen also öfter als nur jeweils einmal gebraucht werden.)

2. Codons für Ordinal- bzw. Kardinalzahlen lassen sich verkürzen, wenn der zulässige Wertebereich bekannt und kleiner als 2^k-1 ist. Es ist dann eine strukturell- deskriptive Angabe vorzusehen, die das Bereichs- Intervall beschreibt, so daß jeder Wert im Intervall der Länge 1 mit CEIL (ld 1) bits codiert werden kann.

3. Der Binärvektor ist gekennzeichnet durch viele zusammenhängende Nullen bzw. Einsen.
Dann läßt sich die explizite Codierung ersetzen durch einen Deskriptor, der Position, Belegung und Länge der der mit gleichen Werten belegten Abschnitte enthält und durch die explizite Codierung der verbleibenden Abschnitte. Es gibt 2 besondere Fälle, in denen diese Form der Codierung oft verwendet wird:

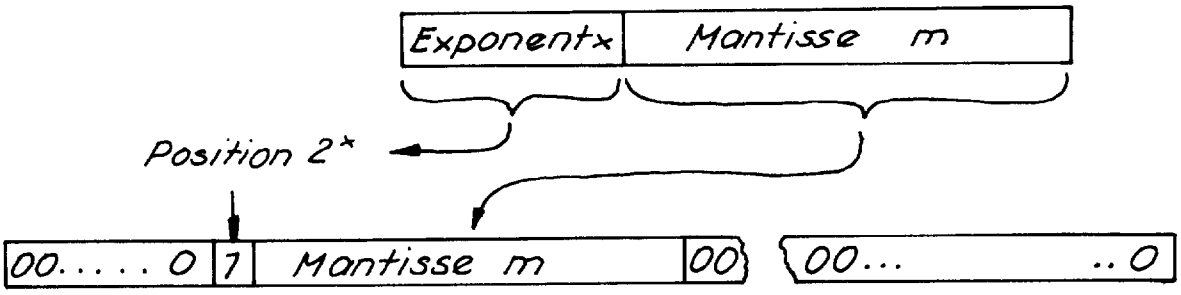
a) Binärvektoren der Form 2^n (es ist nur das Bit an der Position 2^n gesetzt) bzw. deren bitweise Negation. Solche Vektoren werden als Indexvektoren bezeichnet.

b) Gleitkommadarstellung. Sie läßt sich folgendermaßen auffassen (Bild 21): Der Exponent gibt die erste (höchstwertige) Bitposition im Vektor an, also den Index in der Form 2^n für die Mantisse; diese enthält die Belegung der nachfolgenden Bits; die verbleibenden Bits sind mit Ø belegt.

Jetzt wird der Fall betrachtet, daß B^k in mehreren Punkten belegt ist.
Liegt dieser Punktmenge eine Wohlordnung zugrunde, so läßt sie sich durch eine Intervallangabe eindeutig beschreiben.
Hingegen erfordert eine Halbordnung die Angabe aller betreffenden Binärvektoren, beispielsweise in Form einer Binärvektorliste. Solche Listen lassen sich gelegentlich durch ternäre Codierung weiter verdichten. In dieser Codierung sind für jede Bitposition 3 Werte vorgesehen, nämlich Ø, 1 und "-" (im folgenden mit "N" für "Neutral" bezeichnet).[1] Das erfordert 2 Bits für jede Bitposition der Binärvektorliste. Ein ternärer Vektor mit einer einzigen N- Angabe steht für 2 Binärvektoren, die sich nur an der betreffenden Bitposition unterscheiden. Allgemein ersetzt ein ternärer Vektor mit a N- Positionen 2^a Binärvektoren, belegt aber nur den Speicherplatz von 2 Binär-

1 Das Prinzip ist an sich seit längerem bekannt; es wurde z. B. für Assoziativspeicherkonzepte vorgeschlagen (/252/). Durch neuere Arbeiten (hier seien /281/ und /298/ genannt) hat es seine exakte theoretische Grundlage bekommen. Die Bezeichnung "N- Element" stammt aus /122/ bzw. /243/.

Darstellung



Binärvektor zur Repräsentation des gesamten Zahlenbereiches

Bild 21 Gleitkommadarstellung

Ternärvektor mit 2 N-Elementen

1 N 1 0 0 N 1 1 0 1	repräsentiert 4 Binärvektoren, belegt aber nur soviel Speicherplatz wie 2 Binärvektoren
1 0 1 0 0 0 1 1 0 1	
1 0 1 0 0 1 1 1 0 1	
1 1 1 0 0 0 1 1 0 1	
1 1 1 0 0 1 1 1 0 1	

Bild 22 Speicherplatzersparnis bei ternärer Codierung (Beispiel)

vektoren (Bild 22). Das erweist sich für manche Problemen als sehr wirkungsvoll (z. B. Rechnen mit Booleschen Gleichungen, Beschreibung des Verhaltens endlicher Automaten, Abfrage von Relationen), für manche nicht (z. B. für die Speicherung von Relationen).[1] Von diesem Ansatz aus lassen sich neue Prinzipien der Datenkompression für binär codierte Halbordnungen erschließen.[2] Die ternäre Codierung wird weiterhin - für rechnerinterne Zwecke - sehr interessant, wenn man die N- Belegungen als "don't cares" interpretiert und für die Maskierung von Bitpositionen bei bestimmten Verknüpfungen benutzt. Solche Verknüpfungen (maskierte Antivalenz, maskierte Konjunktion usw.) sind für Steuerprogramme aller Art von Bedeutung. Sie müssen oft durch Befehlsfolgen emuliert werden. In manchen Architekturen sind maskierte logische Verknüpfungen in der Befehlsliste vorgesehen; das erfordert allerdings 3 Quelloperanden.[3] Die ternäre Codierung gestattet hier eine elegante Lösung: man kommt mit 2 Quelloperanden aus, braucht also kein besonderes Befehlsformat, und löst mit einer Datenstruktur, einem Konzept und denselben Schaltmitteln Aufgaben der maskierten Verknüpfung, des Umgangs mit Booleschen Gleichungen, der Vergegenständlichung von Automaten[4] und der Abfrage von Relationen[5].

6.2. Objekte

Der Objektbegriff hat hier eine einzige, genau bestimmte Bedeutung: Objekte sind Einheiten gespeicherter Information; sie sind für die einzelnen Angaben gleichsam die Behälter, die aus abstrakter Sicht betrachtet werden können.[6] Die wichtigste Abstraktion ist die der geordneten Menge: die Gesamtheit der Objekte wird als geordnete Menge angesehen, so daß jedes Objekt durch seine Ordinalzahl eindeutig bezeichnet werden kann. Damit wird es möglich, auf Adressenangaben weitgehend zu verzichten. Ordinalzahlen werden nur dann in Adressen umgewandelt, wenn tatsächlich zu den jeweiligen Objekten zugegriffen wird.[7]

---

1 Die ternäre Codierung braucht offensichtlich weniger Speicherplatz als die binäre, wenn im Mittel jeder Ternärvektor wenigstens 2 N- Elemente enthält. Nach bisherigen experimentellen Erfahrungen (/48/) wird sie überlegen, wenn wenigstens 10% der Punkte des Raumes B^k belegt sind. Bei Relationen ist aber k so groß (z. B. k > 4096), daß dieser Wert praktisch nie erreicht wird.

2 Beispiel: Unterraumzerlegung (/297/). Eine Liste mit k Spalten wird durch 2^a Listen mit jeweils k-a Spalten ersetzt, die meist deutlich weniger Speicherplatz brauchen und sich zudem parallel verarbeiten lassen. Daraus könnten sich neue Lösungen für das "memory based reasoning" ergeben.

3 Beispiel: Mikroprozessor Am29116 (/140/, /185/).

4 Zu Verfahren und Schaltungen s. /250/ und /251/.

5 S. den Vorschlag nach /244/ bzw. /245/, Anhang 5.

6 Diese Darstellung geht auf /258/ zurück; vgl. auch /246/.

7 Das Prinzip wurde z. B. in den Systemen iAPX 432 und Burroughs 5500...6900 architekturseitig vorgesehen.

Eine objektorientierte Datenorganisation ermöglicht es also, die Datenabstraktion bis zur Laufzeit aufrecht zu erhalten. Ein Objekt ist im wesentlichen gekennzeichnet durch:

- seine Ordinalzahl in der Menge aller Objekte
- eine aktuelle (veränderliche) Größe
- eine aktuelle (veränderliche) Position in den technischen Speichermitteln (RAM bzw. Plattenspeicher)
- einen aktuellen (veränderlichen) Inhalt
- einen Typ (als Ordinalzahl in der Menge aller Typen angegeben), der bestimmte Eigenschaften kennzeichnet, z. B. die Menge der zulässigen Operationen,die auf das Objekt angewendet werden dürfen
- weitere Angaben technisch- organisatorischer Art (z. B. Zugriffsrechte von Nutzern).

Aus abstrakter Sicht sind folgende Grundoperationen mit Objekten erforderlich:[1]

- Aufbauen
- Ändern (Inhalt bei gleichbleibender Struktur)
- Umbauen (Ändern der Struktur)
- Selektieren von Komponenten (bis zum einzelnen Bit)
- Vernichten.

Weiterhin ist es notwendig, die technischen Speichermittel zu verwalten und die jeweils benötigten Objekte zur Verarbeitung bereitzustellen.
Solche Vorkehrungen sind für komplexe Systeme unumgänglich. Unterschiede gibt es an sich nur in der Bezeichnungsweise und in der Implementierung (Unterstützung durch Hardware oder reine Software- Lösung im Rahmen des Laufzeitsystems; Variablenbindung zur Compilierzeit oder zur Laufzeit). Der Objektbegriff gewährleistet einen exakten einheitlichen Zugang zu diesem Problemkreis.
Hier geht es nicht um ein bestimmtes Verarbeitungsmodell oder eine bestimmte Sprachkonzeption. Vielmehr soll eine einheitliche technische Grundlage geschaffen werden, die nutzbar ist, um verschiedene Verarbeitungsmodelle bzw. Sprachkonzeptionen zu implementieren, wobei es gilt, die Unzulänglichkeiten der bekannten Lösungen zu vermeiden. Dazu folgende Überlegungen:

1. "Was sein muß, muß sein." Die Datenabstraktion zur Laufzeit wird zunehmend unverzichtbar, auch wenn das zugrunde liegende Sprachkonzept dies an sich nicht erfordert.[2] In den betreffenden Funktionsabläufen sind aufeinanderfolgene Zugriffe zu Deskriptor- Angaben in verschiedenen Tabellen auszuführen (vgl. die Bilder 19, 2Ø), wobei ein Zugriff die Parameter für den nächsten liefert. Oft bestimmen Angaben, die in einem Zugriff gelesen wurden, den folgende Zugriff. Bei solchen

---

1 Die Anregungen hierfür gehen auf grundlegende Arbeiten von Zemanek und Dijkstra zurück (/37/, /335/).
2 Muß der Compiler alle Bezüge auflösen, so ist bei jeder Änderung ein zeitaufwendiger Compilerlauf für den gesamten Programmkomplex nötig (vgl. S. 2Ø, Punkt 4.).

Folgen voneinander abhängender Lese-, Verzweigungs- und Rechenabläufe sind die üblichen Cache- Anordnungen und Vorkehrungen zur Ablaufüberlappung gar nicht oder nur in geringem Maße wirksam.[1] Es ist deshalb sinnvoll, die entscheidenden Abläufe mit höchstmöglicher Effizienz zu vergegenständlichen, so daß zu deren Steuerung keine Befehlsfolgen nötig sind und jeder verfügbare Speicherzyklus unmittelbar für den Zugriff zu den notwendigen deskriptiven Angaben genutzt werden kann.

2. Diese Zugriffsorganisation muß umgehbar sein für die Fälle, wo sie nicht benötigt wird. Man kann beispielsweise die Vorkehrungen zur Objektverwaltung lediglich dazu nutzen, die jeweils benötigten Programme und Datenbereiche zwecks Verarbeitung im Speicher bereitzustellen und die einzelnen Informationsstrukturen dann auf herkömmliche Weise adressieren, d. h. die Objektverwaltung auf die Behälterverwaltung beschränken.[2] Auch wäre bei einem Sprachkonzept wie Ada daran zu denken, in der Erprobungsphase die Variablenbindung bis zur Laufzeit aufzuschieben und für die Nutzungsfreigabe die Programme erneut - diesmal mit Variablenbindung - zu compilieren.

3. Für die Vergegenständlichungen sind hinreichend leistungsfähige Schaltmittel vorzusehen[3], und die Umgehungen sind so einzuführen, daß die jeweils zu umgehenden Schaltmittel tatsächlich nicht durchlaufen werden (Vermeidung unnötiger Signallaufzeiten).

Kompliziertere Objekte werden aus einfacheren aufgebaut.[4] Ein solches Objekt ist dann durch die geordnete Menge seiner Komponenten gegeben. Für die Implementierung gibt es grundsätzlich 2 Möglichkeiten:

1. Zusammengesetztes Objekt: das Objekt wird als Ganzheit behandelt, in der die einzelnen Datenstrukturen aneinandergereiht sind. Um zu einer bestimmten Komponente zuzugreifen, muß ein entsprechender Selektor angewendet werden.

2. Verbundobjekt: jede einzelne Komponente ist ihrerseits ein Objekt, auf das auch einzeln zugegriffen werden kann. Das eigentliche Verbundobjekt ist praktisch nur eine Tabelle der Identifier (Ordinalzahlen) seiner Komponenten.

Diese beiden Möglichkeiten sind in Bild 23 an einem Beispiel veranschaulicht. Des weiteren ist es sinnvoll, zu unterscheiden, ob ein Objekt aus gleichartigen oder aus verschiedenartigen Datenstrukturen aufgebaut ist. Daraus ergibt sich eine Einteilung in homogene und heterogene Objekte. Bild 24 zeigt die verschiedenen Objektstrukturen im Überblick.

---

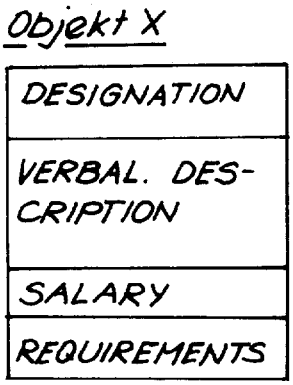
1 Das gilt besonders für RISC- Maschinen.
2 Das wurde z. B. in den Realzeit- Betriebssystemen nach /234/; /237/ bzw. /249/ verwirklicht; der 80286 leistet im Protected Mode Ähnliches.
3 Vgl. S. 20, Punkt 6.
4 Die Ausführungen gehen auf /246/ zurück; wesentliche Anregungen stammen aus /225/ und /258/.

Beispiel eines Objekts, das aus verschiedenen Datenstrukturen besteht:

```
type JOB_CHARACTERISTICS is record
DESIGNATION: array (1..20) of CHARACTER;
VERBAL_DESCRIPTION: array (1..2000) of CHARACTER;
SALARY: INTEGER range 100..9000;
REQUIREMENTS: (UNSPECIFIED, LOW_LEVEL, GRADUATE, PHD);
end record;

X: JOB_CHARACTERISTICS;
```

a) Repräsentation von X als zusammengesetztes Objekt



b) Repräsentation von X als Verbundobjekt

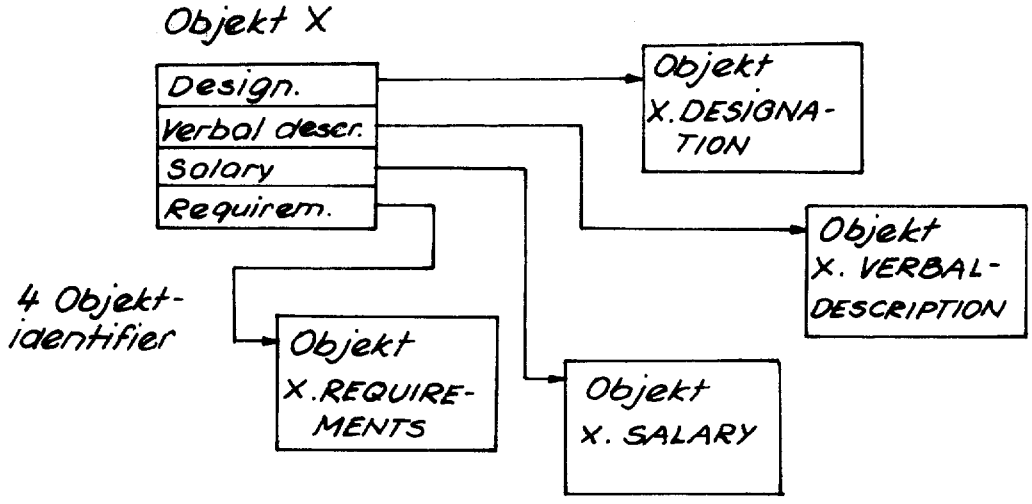


Bild 23 Verschiedene Implementierungen von Objektstrukturen

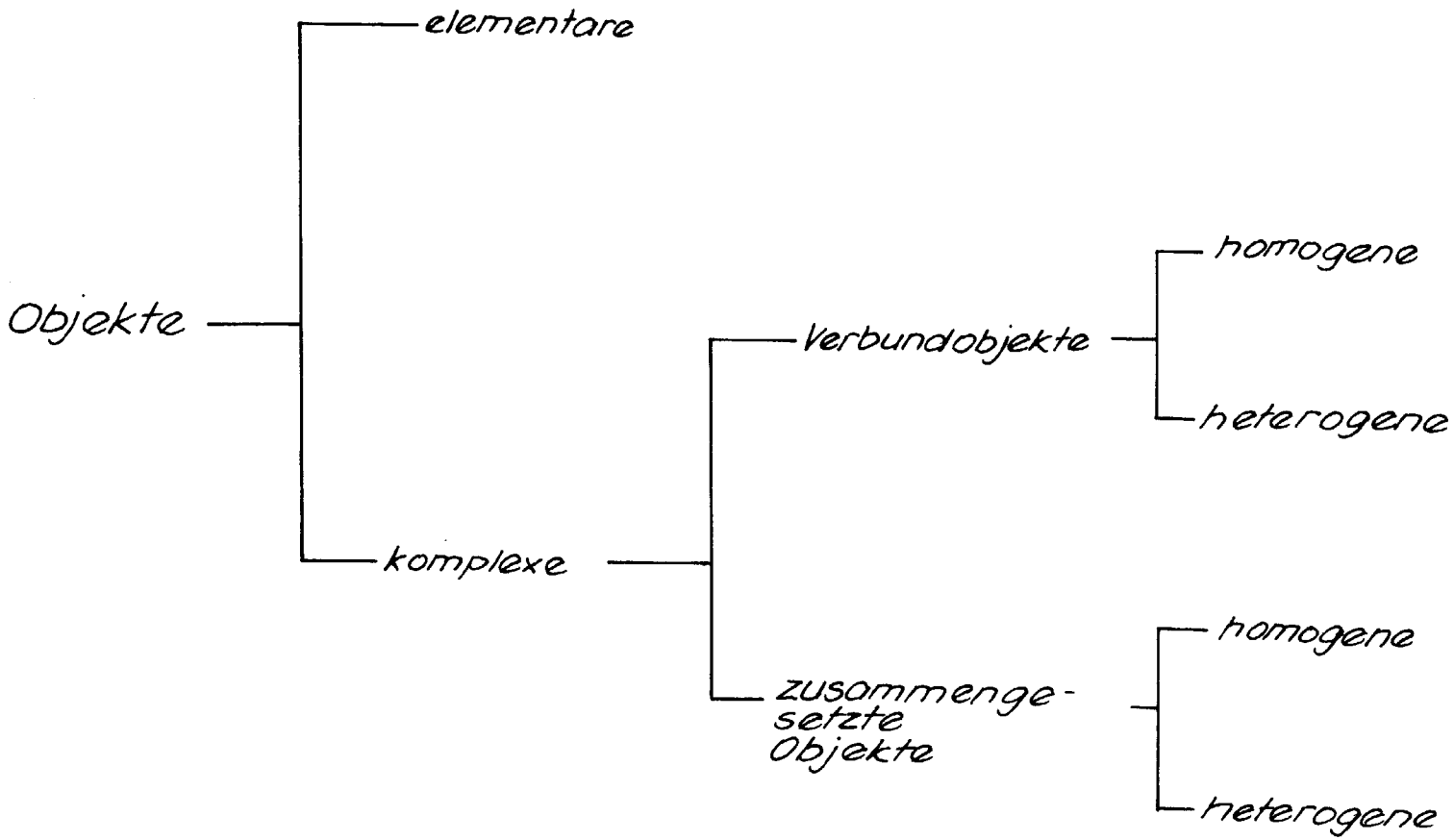


Bild 24 Übersicht über die Einteilung der Objektstrukturen

6.3. Zugriffsorganisation

6.3.1. Objektbeschreibungen und -zugriffe

Um zu den gespeicherten Datenstrukturen zugreifen zu können, müssen die Ordinalzahlen der jeweiligen Objekte bzw. der Komponenten von Objekten in die aktuellen technischen Positionsangaben (RAM- Adressen bzw. Spur-, Zylinder- und Sektorangaben bei Magnetplatten) umgewandelt werden. Die Gesamtheit der Objekte umfaßt vielgestaltige und oft recht umfangreiche Datenstrukturen, so daß - auch bei Nutzung der modernsten Technologien - ein sparsamer Umgang mit der verfügbaren Speicherkapazität geboten ist. Assoziative Zugriffe zu Strukturen der Form "Ordinalzahl: Inhalt" sind, zumindest in großem Maßstab, technisch nicht ohne weiteres zu verwirklichen. RAM- Anordnungen sind bedeutend kostengünstiger als assoziative Speicher[1]; so erscheint es sinnvoll, Ordinalzahlen direkt als Adressen für RAM- Anordnungen zu verwenden und auf diese Weise die wichtigste Form des assoziativen Zugriffs technisch nachzubilden. Damit dies durchführbar wird, sind folgende Voraussetzungen zu erfüllen:

1. Die betreffenden Ordinalzahlen müssen in einem beherrschbaren Bereich liegen; die absolute Obergrenze liegt vielleicht bei 24 bit (bzw. bei einer Menge mit 16 777 216 Elementen)[2]; vorzuziehen sind 12 bit oder weniger[3].

2. Alle auszuwählenden Angaben müssen die gleiche Länge haben; diese muß der Aufrufbreite des Speichers entsprechen (obere Grenze 64...512 bit).

Um die 2. Voraussetzung erfüllen zu können, muß eine an sich beliebige Datenstruktur, die aus n Komponenten unterschiedlicher und veränderlicher Länge besteht, zerlegt werden in eine Tabelle aus n gleich großen, in sich fest formatierten Angaben und - erforderlichenfalls - in einen variablen Teil.
Im allgemeinen enthält die Tabelle für jede der n Komponenten einen Deskriptor, der Beginn und Länge des eigentlichen Inhalts im variablen Teil beschreibt. Bild 25 zeigt das Prinzip.

Die wichtigsten Prinzipien der Objektzugriffe sind zu vergegenständlichen. Die Festlegung der Tabellenstrukturen und Deskriptorformate ist dabei eine entscheidende Aufgabe, namentlich in Hinsicht auf eine hohe Verarbeitungsgeschwindig-

---

1 Um n Bits zu speichern, braucht man stets n binäre Speicherzellen (Flipflops, MOS- Transistoren o.ä.). In RAM- Strukturen wächst der Adressierungsaufwand in Näherung gemäß $O(\sqrt{n})$ (Koinzidenzauswahl). Assoziative Auswahl erfordert hingegen für jede zu speichernde Angabe: 1. das Mitspeichern der Kennung , 2. Vergleichshardware. Der Aufwand dafür wächst wenigstens linear mit der Speicherkapazität.

2 Diese Speicherkapazität ist noch mit beherrschbaren RAM-Anordnungen aufzubauen: 16 MWorte zu 96 bit + ECC erfordern weniger als 2ØØØ 1Mbit- DRAMs (künftig < 12Ø 16Mbit- DRAMs).

3 Dafür sind schnelle statische RAMs nutzbar.

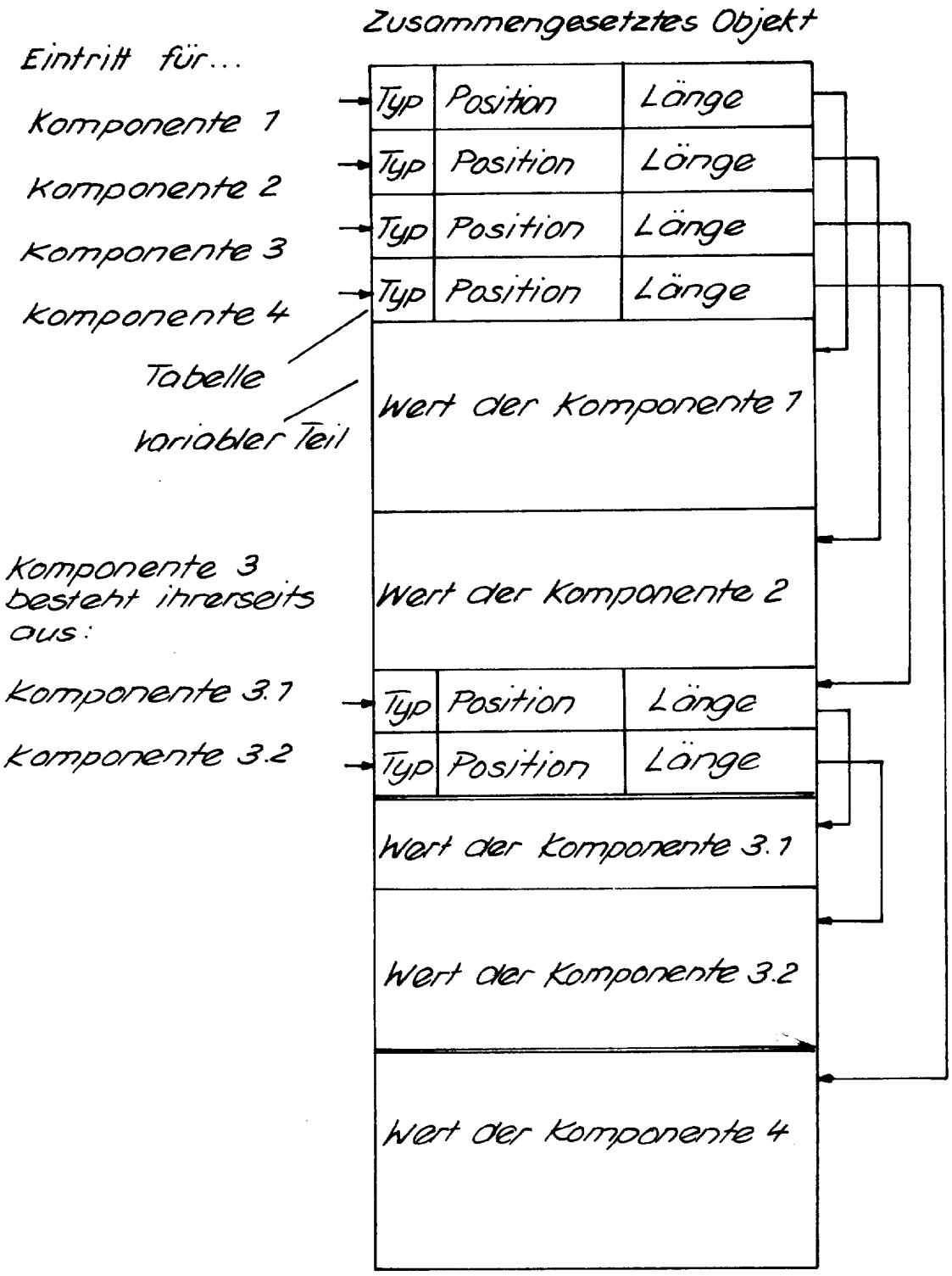


Bild 25 Prinzip des Zugriffs zu Komponenten eines zusammengesetzten Objekts

keit (Mangel aller bisherigen Ansätze!). Dazu einige Überlegungen:

1. An sich muß der Deskriptor nur einen Zeiger auf den Anfang des Inhalts enthalten. Dort kann die weitere beschreibende Information (Länge, Zugriffsrechte usw.) untergebracht werden.
Vorteil: die Tabelle wird kurz.
Nachteil: deutlicher Leistungsverlust, da die Größe des Objekts bzw. der Komponente und die jeweiligen Berechtigungs- und Zustandsangaben frühestens nach einem zweiten Zugriff (zum Anfang des Inhaltes) zugänglich sind. Diese Angaben werden aber für die Speicherverwaltung unbedingt benötigt. Daraus folgt:
Ein Deskriptor sollte alle Angaben enthalten, die die Speicherverwaltung braucht, um die Informationsstruktur zugriffsfähig bereitstellen zu können (d. h. Position, Größe und Zustand) .

2. Jede Informationsstruktur hat einen bestimmten Typ. Wesentliche Typen allgemeiner Verwendbarkeit sind vergegenständlicht. Diese Typen heißen Fundamentaltypen. Jede Informationsstruktur enthält eine Kennung ihres Fundamentaltyps, so daß dieser direkt durch Schaltmittel ausgewertet werden kann.
Die Vielfalt der Typen, die in konkreten Anwendungsumgebungen vorkommen können, wird nicht berücksichtigt. Es werden aber Unterstützungsmaßnahmen für den Fall vorgesehen, wo es notwendig ist, zur Laufzeit zusätzliche deskriptive Information zu den Objekten mitzuführen.

3. Die Gesamtheit der Objekte wird in einer Objekttabelle (Object Reference Table ORT) beschrieben.
Es sind mehrere Objekttabellen in einem System zugelassen. Jeder Komplex aus Objekttabelle und zugehörigen Objekten wird als Umgebung (Environment) bezeichnet. Beispiel für Grenzwerte: 4Ø96 Umgebungen zu 16 777 216 Objekten.

4. Die Objektdeskriptoren enthalten allgemeine Positionsangaben. Sie sind mit einer Kennung versehen, die die Art des jeweiligen Speichers bezeichnet. Die einzelne Positionsangabe wird von der Speicherverwaltung sinngemäß interpretiert, z. B. als Adresse des in der Kennung angegebenen RAM oder auch als Laufwerks-, Zylinder- Spur- und Sektornummer, wenn in der Kennung ein Plattenspeicher angegeben ist[1].

5. Es ist eine Angelegenheit der jeweiligen Anwendung (bestimmt durch Programmiersprache, Compiler und Laufzeitsystem), wann der Bezug auf codierte deklarative Datenbeschreibungen verlassen wird (für solche Zugriffe sind dann in den Programmen Adressenwerte relativ zur Anfangsadresse des Objekts angegeben). Beispielsweise kann für zusammengesetzte Objekte, deren Struktur zur Compilierzeit bekannt ist (und die sich zur Laufzeit nicht ändert!), die Adressenumsetzung bereits vom Compiler erledigt werden.

---

1 Das verkürzt die Zugriffszeiten (direktes Positionieren ohne Zugriff auf ein Directory); vgl. /88/, /237/, /249/.

6.3.2. Ablauforganisation

Der Ablauforganisation liegt eine 4- stufige Speicherhierarchie zugrunde:

1. Gemeinsame Datenbasis. Sie nimmt alle Objekte auf, die als Datenbestand des Systems erhalten bleiben müssen. Dafür sind beispielsweise Plattenspeicher vorgesehen. Teile der Datenbasis können zwecks schnellerem Zugriff in RAM- Anordnungen bereitgestellt werden. In großen Maschinen kann die gesamte Datenbasis im RAM gehalten werden; Plattenspeicher dienen dann nur noch zur Datensicherung.[1] Mit diesem Konzept sollen die üblichen Dateisysteme ersetzt werden.[2]

2. Laufzeitumgebung. Diese umfaßt alle Objekte (auch temporäre), die gerade verarbeitet werden.

3. Operanden- und Steuerspeicher. Diese sind den Operations- bzw. Steuerwerken unmittelbar vorgeordnet.

4. Register in der Hardware.

Programme werden ausschließlich als funktionelle Zuordnungen aufgefaßt.[3] Zur Abarbeitung eines Programms werde alle Objekte, die es benötigt, in die Speichermittel der Laufzeitumgebung überführt. Als Resultat des Programmlaufs entstehen temporäre Objekte. Diese können durch Zuweisung in die Datenbasis übernommen werden, sei es in Form neuer Objekte, sei es als Änderung des Inhalts bzw. der Struktur bereits existierender Objekte. Das Programm selbst besteht aus Folgen vergegenständlichter Algorithmen ("Maschinenbefehle" in der üblichen Redeweise). Jeder vergegenständlichte Algorithmus leistet in sich wiederum funktionelle Zuordnungen, deren Argumente in Operandenspeichern bzw. Registern bereitzustellen sind und deren Resultate ebenfalls in Operandenspeichern bzw. Registern abgelegt werden. Dieses Prinzip hat wesentliche Vorteile:

1. Es ist frei von Seiteneffekten.

2. Die Informationssstrukturen, die verarbeitet werden sollen, können stets in jeweils technisch zweckmäßig ausgebildeten Speichermitteln untergebracht werden, die direkt mit den Verarbeitungsschaltungen verbunden sind.

---

1 Das ist bei der Connection Machine verwirklicht (/303/, /311/).

2 Vorbilder: IBM System/38 und AS/400 (/39/).

3 Anregungen sind z. B. in /97/, /100/, /102/ gegeben. Eine solche Auslegung bedeutet nicht, daß prozedurale Konzepte (z. B. Algol) nicht implementierbar sind. Vielmehr lassen sich die notwendigen Zuweisungen ohne weiteres programmtechnisch emulieren, und zwar auf einem Befehlsniveau, das in Geschwindigkeit und Flexibilität dem Mikroprogrammniveau üblicher Maschinen gleichwertig ist.

3. Fragen der Fehlerbehandlung werden wesentlich besser beherrscht. Wird bei der Abarbeitung eines Algorithmus ein Fehler festgestellt, so ist der Ablauf ohne weiteres wiederholbar, solange die Resultate noch nicht zugewiesen wurden.

4. Es ist ein einheitliches Konzept für die Ablauforganisation, vom Programmkomplex bis zum Maschinenbefehl. Da es nur auf die funktionelle Zuordnung ankommt (Heranschaffen der Argumente, Zuordnung der Resultate und deren Zuweisung sind völlig voneinander entkoppelt), lassen sich codierte und vergegenständlichte Implementierungen desselben Algorithmus (also: Programme und spezielle Hardware) gegeneinander austauschen.

5. Jedes Programm ist als Unterprogramm nutzbar.

6. Man hat stets die Wahl, die Resultate zuzuweisen oder nicht und kann somit jede Funktion für das Abtesten von Bedingungen ausnutzen (besondere Testbefehle können also entfallen).

Der einzige wesentliche Nachteil besteht darin, daß alle Parameter durch ihren Wert übergeben werden müssen. Die Vorteile sind aber so bedeutsam, daß es sich lohnt, die Auswirkungen des Nachteils durch technisch- erfinderische Maßnahmen zu mindern; u. a. beispielsweise durch:

- Vorkehrungen für extrem schnelle Blocktransporte
- Vorkehrungen zur Anzeige der Belegung von Speichermitteln, um unnötige Transporte zu vermeiden
- Optimierung der Speichernutzung durch die Compiler.

6.3.3. Programmstrukturen

Programme sind heterogene zusammengesetzte Objekte. Sie bestehen aus einer Zugriffstabelle (Access Reference Table ART) und wenigstens einer Befehlsfolge.
Beim Aufruf eines Programms wird in einem Stackbereich ein Activation Record erzeugt. Im allgemeinen Fall wird dazu die ART auf den Stack kopiert, dann werden die Parameterangaben eingetragen. Die Befehle des Programms beziehen sich ausschließlich auf den Activation Record, der Direktwerte oder Zugriffsdeskriptoren für Objekte enthält (Bild 26).
Die Prinzipien sind an sich bekannt und bewährt (vgl. Bild 19 und 2Ø). Hier soll versucht werden, durch technische Maßnahmen eine hohe Verarbeitungsgeschwindigkeit zu erreichen, beispielsweise durch Schnellspeicher für die aktuell benötigten Tabellen.

6.4. Speicherorganisation

6.4.1. Organisation der Ressourcen

Alle technisch gegebenen Ressourcen sind stets endlich. Man hat nun die Wahl, diese Tatsache in den Architekturfestle-

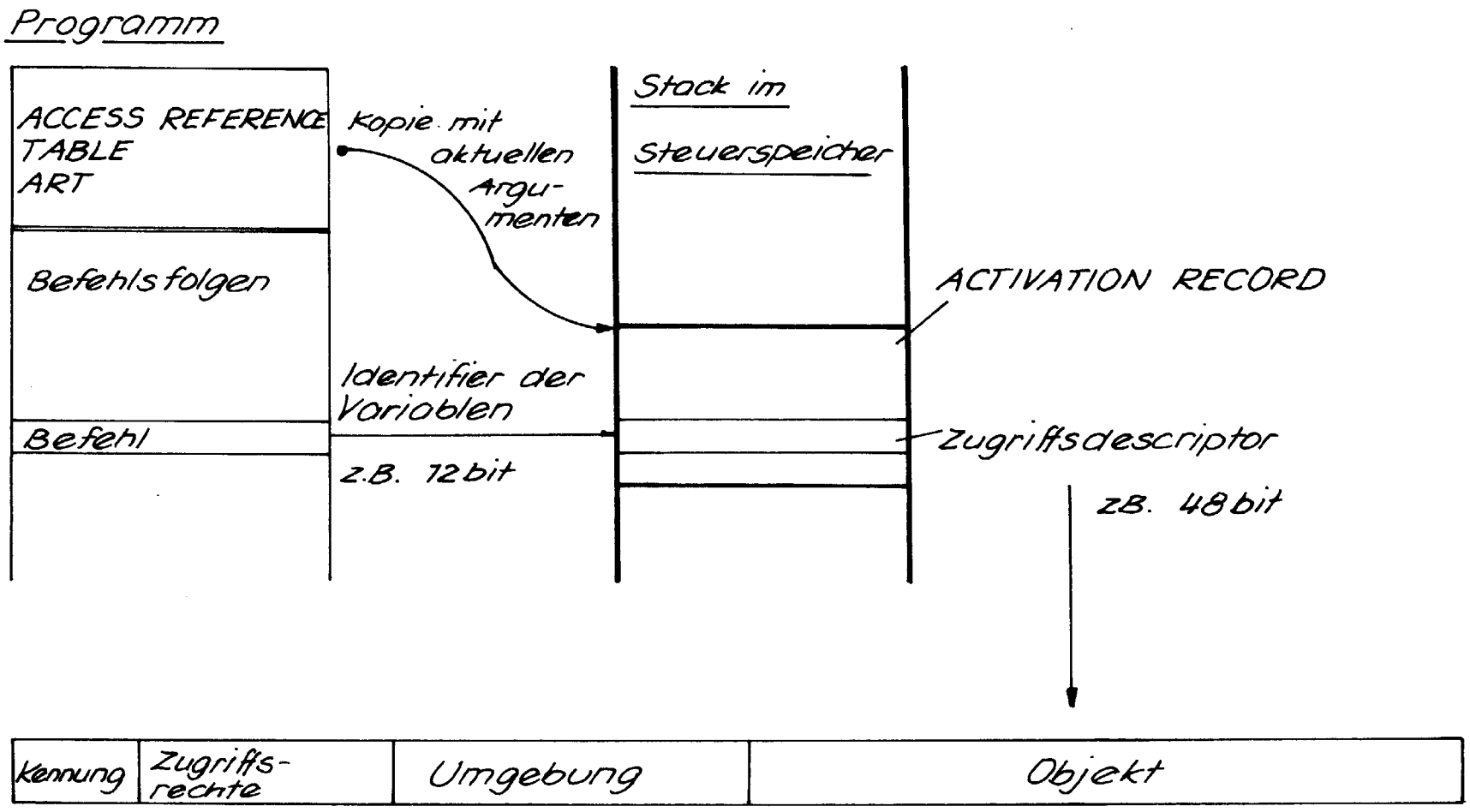


Bild 26 Grundsätzliche Programmstruktur

gungen widerzuspiegeln, oder weitere Mittel vorzusehen, um die Endlichkeit gleichsam so gut wie möglich zu verstecken (Beispiel: virtueller Speicher).
Es ist klar, daß eine Programmiersprache soweit wie möglich von den technischen Ressourcen abstrahieren sollte. Hier geht es aber um die Ebene der Hardware, um das Leistungsvermögen der technischen Mittel: deshalb erscheint es vorteilhaft, die Endlichkeit der Ressourcen bewußt herauszustellen. Dieser Ansatz soll als Prinzip der kontrollierten Kardinalität bezeichnet werden.
Sowohl Informationsstrukturen als auch technische Mittel werden als geordnete endliche Mengen aufgefaßt. Die maximale Kardinalität einer jeden solchen Menge wird nach gewissen Kriterien der Sinnfälligkeit festgelegt. Dafür werden angemessene Speicher-, Verarbeitungs- und Steuermittel bereitgestellt. Es wird gewährleistet, daß alle Ressourcen der programmseitigen Steuerung zugänglich sind. Damit wird es möglich, über optimierende Compiler das Leistungsvermögen der Hardware besser auszunutzen.[1] An sich ist dieses Prinzip eine Verallgemeinerung der Gepflogenheiten beim Entwerfen von Sondermaschinen, jeweils nach Maßgabe der Zweckmäßigkeit verschiedenartige Speicheranordnungen sowie Verknüpfungs- und Steuerschaltungen "an Ort und Stelle" vorzusehen.[2]

6.4.2. Organisation der Speicherebenen

Register
Die unterste Ebene bilden die Register, die in der Architektur definiert sind. Sie sind praktisch die kostbarste Ressource: sie bestimmen maßgeblich das Leistungsvermögen der Hardware, und ihre Anzahl (bis hin zur einzelnen Bitposition gesehen) wird stets vergleichsweise gering sein.
Höchste Verarbeitungsgeschwindigkeiten können nur dann erreicht werden, wenn die in der Architektur definierten Register auch tatsächlich direkt (z. B. mit Flipflops, nicht als RAM- Bereich) ausgeführt und mit den Verknüpfungsschaltungen unmittelbar verbunden sind. Jede zwischengeordnete Auswahlschaltung, jedes Bussystem verlängert die Dauer der Verarbeitungszyklen (Register - Register- Verknüpfungen).
Daraus folgt sofort die Notwendigkeit, für höchste Leistungsansprüche das übliche Prinzip der Universalregister zu verlassen. Andererseits scheint aus intuitiver Sicht eine Sammlung von Schaltungsanordnungen der Form "Register - Verknüpfungsschaltung - Register" auch - trotz kürzester Verarbeitungszyk-

---

1 Hier wird auf Forschungen zu RISC- und VLIW- Machinen sowie zu optimierenden Compilern Bezug genommen. Beim beschriebenen Prinzip geht es darum, zwar die Ressourcen der Hardware dem Compiler zugänglich zu machen, aber nicht deren Feinstruktur. Der Compiler soll Schnellspeicher explizit nutzen können, es soll aber nicht damit belastet werden, z. B. Zeitabläufe in Pipelines zu analysieren.

2 Als Beispiel s. /240/, wo neben einem Aktiven Memory- Array Speicher für Zweitoperanden, Indexvektoren, Indexlisten, Lösungszahlen und Iterationsangaben vorgesehen sind.

len - nicht die beste Lösung zu sein[1]: je weniger Register verfügbar sind, um so öfter muß zur übergeordneten Speicherebene zugegriffen werden; das verlängert die Verarbeitungszeit wieder.
Über die beste Nutzung einer Vielzahl von Registern gibt es verschiedene Auffassungen:

1. als Speicher von Variablen der aktuellen Prozedur (viele eingeführte Architekturen und vor allem RISC- Maschinen)[2]

2. als oberster Teil eines Stack ("Stack Cache"). So wurde im CRISP- Mikroprozessor (/158/) ein "Stack Cache" von 32 Worten zu 32 bit implementiert. Zu dessen Größe wurden experimentelle Ergebnisse angegeben (Tafel 18). Das Füllen und Entleeren kostet etwa 12% der E/A- Aktivitäten des Schaltkreises.

3. als Sammlung von Zeigern, um auf wichtige Datenstrukturen (Daten- Stack, Prozedur- Stack, Heap usw.) schnell zugreifen zu können.[3]

Es ist davon auszugehen, daß die Register- Ressourcen auf jeden Fall beschränkt sind, also irgendwann ein Ausweichen auf übergeordnete Speicherebenen unvermeidbar sein wird. Es sind deshalb so viele Register vorzusehen, daß

- alle Parameter für die jeweiligen vergegenständlichten Zuordnungen gehalten werden können

- für jeden Parameter Ausweichmöglichkeiten in die übergeordnete Ebene vorgesehen sind (Zeigerregister in einen Rettungsbereich, Stacks o. dergl.).

Schnellspeicher
Nach dem Prinzip der kontrollierten Kardinalität sind die Schnellspeicher nicht als Cache im übliche Sinne gedacht, sondern als direkt zugängliche, in ihrer Größe genau definierte Ressourcen. Mit diesem Ansatz soll versucht werden, schnelle RAM- Schaltkreise mittlerer Speicherkapazität (Richtwert: 4kbit...64kbit) in heterogenen, aus verschiedenen zweckgerecht verschalteten Speicherblöcken gebildeten Anordnungen wesentlich wirksamer auszunutzen als in regulären Befehls- und Daten- Caches. Die Zykluszeit eines jeden solchen Speichers sollte nicht größer sein als die der Verarbeitungsschaltungen. Die Adressierung sollte direkt wirken, ohne den Zeitverlust von Adressenumsetzungen bzw. -prüfungen.

---

1 S. auch Abschnitt 7.1., S. 132 f.
2 Das bringt nur dann Vorteile, wenn die Variablen in den meisten Fällen vom passenden Typ sind (z. B. 32-bit- Binärzahlen).
3 Sind nur sehr wenige Register vorgesehen, können diese kaum anders genutzt werden, selbst wenn sie an sich universell ausgelegt sind (Beispiel: 8Ø86).

Größe des "Stack Cache" (32-bit- Worte)	Trefferrate ("on-chip references")
Ø	Ø%
4	42%
8	5Ø%
16	73%
32	81%
64	82%
128	82%
256	82%

Tafel 18 Trefferraten eines "Stack Cache"

Angabe im Befehl	relative Länge des Codes
bitvariabel	1,Ø
4/8/12 bit	1,18
6/12/18 bit	1,13
8/16/24 bit	1,43
nur 12 bit	1,78
nur 16 bit	2,12
S/37Ø	3,3Ø

Tafel 19 Versuchsergebnisse zur Effektivität verschiedener Codelängen für Selektionsangaben (Variablen- Identifier) in Maschinenbefehlen

Solche Anordnungen sind erforderlich für:

- die Befehlsfolgen des aktuellen Programms
- den Activation Record des aktuellen, des rufenden und des gerufenen Programms
- Teile der Objekttabellen
- wichtige Systemroutinen und Steuertabellen.

Beispiel: Es gibt statische 4 kbit- RAMs mit 10 ns Zugriffszeit.[1] Damit läßt sich die Schaltungsanordnung für einen Zyklus von 25 ns auslegen. Wird die Ordinalzahl des gewünschten Eintrags des Activation Record an die Adresseneingänge gelegt, so hat man nach 25 ns den Objektidentifier, der seinerseits weiteren Speicherschaltungen als Adresse zugeführt werden kann. Im günstigsten Fall ist die reale Objektadresse nach 50 ns bekannt. Das leistet keine der üblichen Cache- Anordnungen.[2]

Für die Größenfestlegung solcher Speicher können experimentelle Befunde (/309/) herangezogen werden.[3] In keinem der untersuchten Programme wurden mehr als 4096 Angaben adressiert; am häufigsten waren Werte zwischen 17 und 64. Diese Ergebnisse werden durch eine einfache Überlegung gestützt: In modernen Programmiersprachen (z. B. Ada) müssen alle Variablen deklariert werden. Wer übersieht aber 4096 Variable gleichzeitig (d. h. in einer einzigen Funktion oder Prozedur)?

Speicher der Laufzeitumgebung

Diese Speichermittel müssen die aktuellen Programme zusammen mit den jeweiligen Datenbeständen aufnehmen können. Sie werden mit dynamischen RAMs großer Speicherkapazität aufgebaut.

Speicher der Datenbasis

Für den Datenerhalt sind beim Stand der Technik Magnetplattenspeicher unerläßlich. Es kommen weiterhin optische Plattenspeicher in Frage, für Archivierungszwecke vielleicht auch Magnetbandsubsysteme.[4] Im Interesse der Leistung sollte die Datenbasis während des Betriebs weitgehend in RAM- Anordnungen gehalten werden.

---

1 Nach /107/ hat der derzeit schnellste RAM (GaAs; 1k*4 bit) eine Zugriffszeit von 2,5 ns (ECL- RAM: 5 ns).

2 Vgl. auch die schlechte Leistung von Caches bei Vektorverarbeitung oder häufiger Kontextumschaltung.

3 Es wurde untersucht, wieviele Variablen in Programmen tatsächlich adressiert werden, um eine zweckmäßige Codierung der Selektionsangaben (Variablen- Identifier) im Befehl zu bestimmen. Da solche Werte für die künftige Gestaltung von Befehlslisten bedeutsam sind, werden sie in Tafel 19 mitgeteilt.

4 Die Gesichtspunkte der Datenverwaltung, Archivierung usw. sind hier nicht weiter von Interesse.

Forderungen an Struktur und Zusammenwirken

Bild 27 zeigt den grundsätzlichen Aufbau der Speicherhierarchie. Es wird eine weitgehende logisch- funktionelle Trennung zwischen den Hierarchie- Ebenen angestrebt. Die architekturseitigen Wirkprinzipien in den einzelnen Speicherebenen sollten so unabhängig wie möglich definiert werden; beispielsweise sollte die Datensicherung in der Datenbasis von den Transportabläufen zwischen den Speichern der Laufzeitumgebung und den Schnellspeichern vollkommen unabhängig sein.
Der Stand der Technik rechtfertigt es, in den verschiedenen Ebenen programmierbare Einrichtungen einzusetzen, um diese Wirkprinzipien zu implementieren. Beispielsweise können in den "höheren" Speicherebenen autonom arbeitende Mikroprozessoren für Funktionen vorgesehen werden, die einer besonderen Beschleunigung nicht bedürfen, die eigentliche Verarbeitungshardware aber möglichst nicht belasten sollen (Datensicherung, zyklische Kontrollen usw.).
Es sollte aber auch möglich sein, in kleineren Konfigurationen Speicherebenen technisch (bei logisch- funktionell weiterhin bestehender Unabhängigkeit!) zusammenzufassen oder wegzulassen (Beispiel: gemeinsame RAM- Anordnungen für Datenbasis- und Laufzeitspeicher).
Von leistungsentscheidender Bedeutung sind schnelle Transportabläufe zwischen den einzelnen Speicherebenen.

6.4.3. Speicherverwaltung

Die Aufgabe der Speicherverwaltung besteht darin, Objekte zur Verarbeitung bereitzustellen und zu gewährleisten, daß ein gewisser Bestand an Objekten (die Datenbasis) unabhängig von den Betriebszuständen des Systems zuverlässig erhalten bleibt. Dafür gibt es bekannte und bewährte Lösungen, z. B. virtuelle Speicher nach dem Prinzip des Seitenwechsels und Systeme zur Dateiverwaltung (z. B. im Betriebssystem UNIX)..
Um künftigen Anforderungen entsprechen und moderne Technologien gut ausnutzen zu können, muß aber nach neuen Ansätzen gesucht werden. So befinden sich DRAM- Schaltkreise mit 16 Mbit in der Entwicklung, 64 Mbit sind zu erwarten. Es gilt, diese Bauelemente wirksam zu nutzen, und es erscheint sinnvoll, zwar von bewährten Prinzipien auszugehen, aber alle Vorstellungen kritisch zu bewerten, die noch unter Bedingungen der relativen Kostbarkeit von RAM- Ressourcen geschaffen wurden.[1] Einige Überlegungen dazu:

1. Objekte sollten nicht nur gedanklich als Ganzheiten betrachtet , sondern auch technisch zusammenhängend gespeichert werden. Vorteile:

- keine komplizierten Zeigersysteme
- Verkürzung der Transportzeiten
- bessere Übersicht über die Speicherbelegung
- schnellere Verarbeitung durch Zugriffszeitverkürzung.

---

1 Die gängigen Konzepte des virtuellen Speichers wurden Mitte der 60er Jahre entwickelt (Atlas, 360/67); UNIX ab 1968.

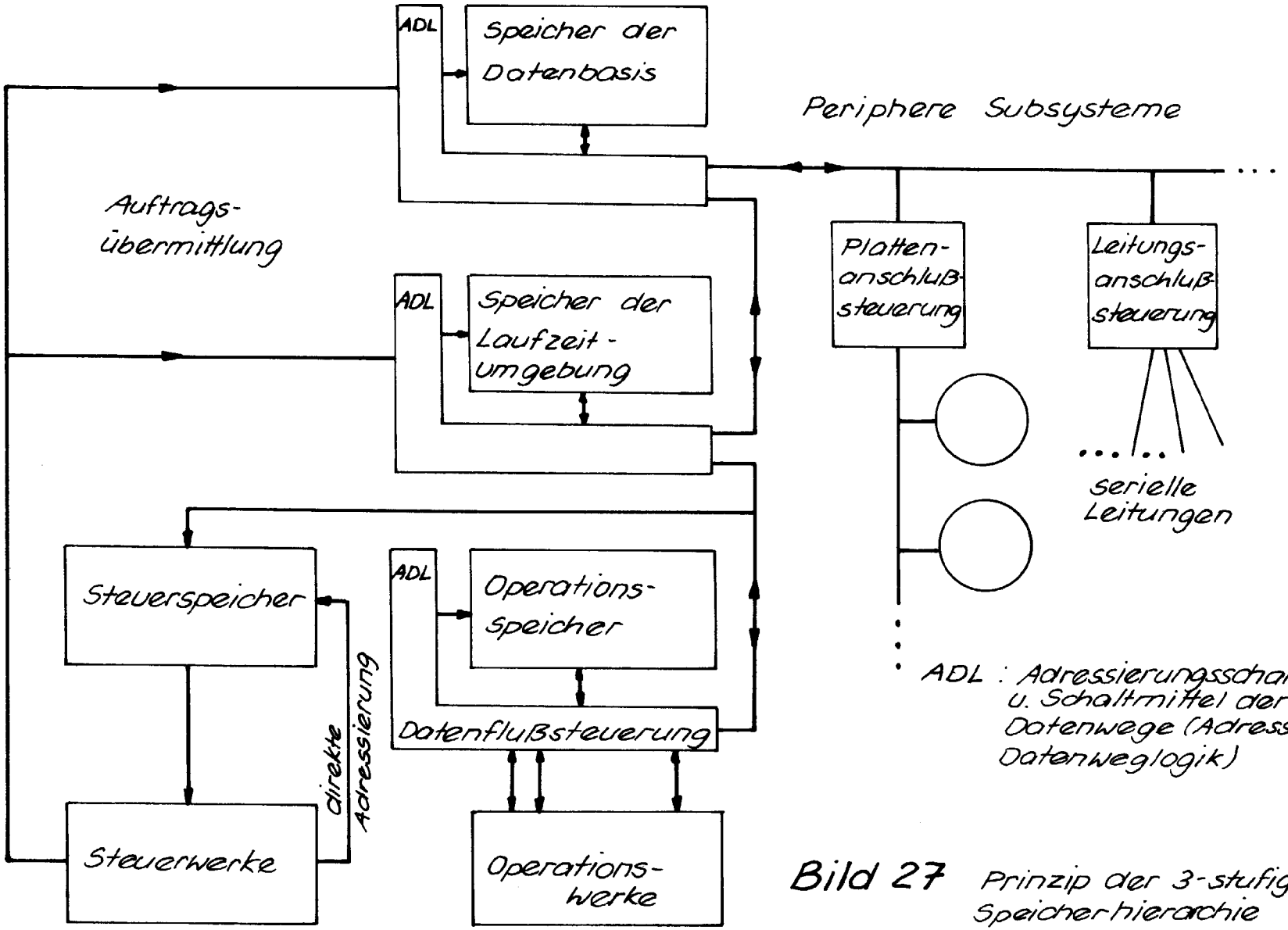


Bild 27 Prinzip der 3-stufigen Speicherhierarchie

Zu allen Komponenten eines zusammengesetzten Objekts kann direkt zugegriffen werden; Zugriffe zu Adreßumsetzungsspeichern, Speicherbelegungstabellen o. dergl. sind nicht erforderlich. Des weiteren sind beim Datenaustausch zwischen RAM und Plattenspeicher keine Positionierabläufe zu gestreuten Datenblöcken notwendig.

2. Die Speicherverwaltung stellt gleichsam die technischen Verpackungseinheiten zur Verfügung, in denen die Objekte untergebracht werden (die ihrerseits die logischen Verpackungseinheiten der Daten sind).
Könnte man hinreichend viele gleich große technische Verpakkungseinheiten vorsehen, wovon jede ein Objekt maximaler Größe aufnehmen kann, wäre das Problem gegenstandslos.
Das ist aber technisch nicht ausführbar. Die Speichermittel müssen so gut wie möglich ausgenutzt werden.
Wird den Objekten nur die zusammenhängende Speicherkapazität zugewiesen, die sie tatsächlich benötigen, ist die Speicherbelegung von Zeit zu Zeit zu verdichten, um hinreichend große freie Bereiche zur Verfügung zu stellen ("garbage collection"). Das kann sehr zeitaufwendig sein.

3. Bei Verpackungseinheiten fester Größe ist stets mit Verlusten zu rechnen, da die Objekte die Speicherbereiche nur selten vollständig ausfüllen.

4. Die Zerlegung in gleich große Verpackungseinheiten geringerer Speicherkapazität ist ein gewisser Ausweg (virtueller Speicher[1], z. B. mit Seiten von 512 Bytes - 4 kBytes; Dateisysteme[2] mit Blöcken von z. B. 512 Bytes; verkettete Listen[3] mit Blöcken von z. B. 128 Bytes). Das erfordert aber zwischengeordneten Abbildungsmechanismen für die Adressierung, die erfahrungsgemäß nur bei jeweils bestimmten Zugriffsweisen effektiv arbeiten (virtueller Speicher bei wahlfreien Zugriffen und "Lokalität" der Information; Dateisysteme und verkettete Listen bei sequentiellen Zugriffen).

5. Als Alternative soll deshalb ein gewisses Sortiment an Verpackungseinheiten mit verschiedener, aber genau festgelegter Speicherkapazität vorgesehen werden. Die Anzahl der Verpackungseinheiten ist maschinenspezifisch. Sie wird durch die jeweilige Speicherausstattung bestimmt und kann programmseitig abgefragt werden.

6. Die elementare Verpackungseinheit ist ein Maschinenwort von 96 bit Länge sowie einigen Zusatzbits (TAG- Bits), im folgenden als Eimer[4] bezeichnet. In diesen Eimer passen viele kleinere Informationseinheiten hinein. In Tafel 2Ø sind einige Aufteilungen angegeben.

---

1 Ein verbreitetes System ist in /36/ beschrieben.
2 Für einen Überblick zu UNIX s. z. B. /55/.
3 Eine solche Implementierung ist in /299/ beschrieben.
4 Die Bezeichnung entspricht Gepflogenheiten der Literatur, wo von "chunks", "buckets", "bags" usw. die Rede ist.

1 * 96 bit	12 * 8 bit
2 * 48 bit	16 * 6 bit
3 * 32 bit	24 * 4 bit
4 * 24 bit	32 * 3 bit
6 * 16 bit	48 * 2 bit
8 * 12 bit	96 * 1 bit

Tafel 20

Beispiele für die Aufteilung eines Eimers von 96 bit

96 ist ein Vielfaches von 12; 12 ist durch 2, 3 und 4 teilbar. Deshalb sind Angaben vom Aufzählungstyp[1] und record- Strukturen meist günstiger in diesem Format unterzubringen als in Wortformaten, deren Länge einer Zweierpotenz entspricht. Auch ist zu erwarten, daß sich diese Format für Deskriptoren, Steuerworte usw. gut eignet.[2] So läßt sich in einem 48-bit-Wort ein Zugriffsdeskriptor mit einem Objektidentifier von wenigstens 32 bit nebst Angaben über Zugriffsrechte usw. bequem unterbringen, und 96 bit gestatten die Formatierung von Objektdeskriptoren mit sehr langen Adressen (im Extremfall: 64 bit Adresse und 32 bit Längenangabe). Schließlich können Daten üblicher Formatierung zusammen mit anwendungsspezifischen Markierungsangaben (z. B. Typkennungen) effektiv verpackt werden (Beispiel: für je 32 Informationsbits 16 Markierungsbits), wenn Konzepte wie Lisp oder Smalltalk zu implementieren sind.

7. Die Eimer werden zu größeren Verpackungseinheiten (Behältern) zusammengefaßt (Größenbereich: 1 Eimer...2^{31}-1 Eimer). Es gibt eine feste, überschaubare Anzahl solcher Behälter (beispielsweise 64 in einer logarithmischen Größen- Staffelung). Anfänglich werden die Datenbasis- und Laufzeitspeicher eines Systems nach Erfahrungswerten so aufgeteilt, daß ein gewisses Sortiment an Behältern verschiedener Größe vorhanden ist. Für jede Behältergröße wird ein Verzeichnis der freien Behälter geführt. Ein Objekt wird stets in einem Behälter untergebracht, der dazu gerade noch ausreicht. Ist der Vorrat an freien Behältern dieses Formats bereits verbraucht, so wird ein nächstgrößerer Behälter in eine entsprechende Anzahl kleinerer zerlegt. Wird ein großer Behälter benötigt und sind nur kleinere frei, so wird versucht, den größeren aus zusammenhängenden kleineren zu bilden. Die "garbage collection" kann solange aufgeschoben werden, bis große Behälter benötigt werden und nur verstreute kleinere frei sind.

9. Da alle Inhalts- und Strukturveränderungen von Objekten auf funktionellen Zuweisungen beruhen, werden entweder nur Komponenten selektiv geändert (ohne Größenänderung) oder das Objekt entsteht bei Zuweisung völlig neu. Damit wird es stets in einem seiner Größe angemessenen Behälter untergebracht. Mit zunehmender Objektgröße wird es zunehmend seltener notwendig, das Objekt in einen neuen Behälter zu überführen.

---

1 Die meisten Deklarationen vom Aufzählungstyp enthalten nur wenige Elemente (meist << 255). Folglich entstehen nach den Prinzipien von Abschnitt 6.1. meist nur kurze Binärzahlen.

2 32 bzw. 64 bit sind meist recht knapp (iAPX 432 Zugriffsdeskriptor, S/370 PSW und CCW); 128 bit nicht immer sinnvoll ausnutzbar (iAPX 432 Objektdeskriptor).

6.5. Datenstrukturen

Alle Datenstrukturen (numerische und nichtnumerische) werden auf Binärvektoren variabler Länge zurückgeführt. Grundsätzlich ist jeder Binärvektor bis aufs Bit selektierbar. Dieses Konzept hat bedeutsame Vorteile:

1. Maximale Codeverdichtung, da es nicht notwendig ist, auf starre Formate Rücksicht zu nehmen. Die Prinzipien von Abschnitt 6.1. lassen sich so praktisch verwirklichen.

2. Verschiebe- und Rotationsbefehle können entfallen. Solche Befehle werden häufig nur genutzt, um einzelne Angaben aus dicht gepackten Datenstrukturen zur Verarbeitung aufzubereiten, z. B. um Zahlenwerte rechtsbündig für Rechenoperationen bereitzustellen.[1]

3. Vereinfachungen in Compilern. Um zu beliebigen Informationsstrukturen zugreifen zu können, reicht es aus, entsprechende Selektionsangaben zu deklarieren. Es ist nicht erforderlich, dafür besondere Unterprogramme zu erzeugen.

4. Alle Datenstrukturen werden maschinenintern ausschließlich deklarativ repräsentiert. Sie sind somit der Anwendung von Inferenzregeln oder relationalen Operatoren zugänglich.

Derartige Vorteile waren wiederholt Anlaß, das Prinzip zu implementieren.[2] In der Vergangenheit hat es sich aber nicht durchsetzen können. Das liegt im wesentlichen an den Aufwendungen: Technisches Kernstück dieses Konzepts ist der Bitfeldextraktor. Bei einer Verarbeitungsbreite von n bit kostet er wenigstens $3n^2$ Gatter (2-fach NAND). Für n = 32 bit braucht man also 3168 Gatter. Das entspricht 8ØØ SSI- bzw. etwa 16Ø MSI- Schaltkreisen (Multiplexern).
Bei VLSI- Technologien liegen die Verhältnisse allerdings günstiger. Die Aufwendungen sind bei Integrationsgraden von 2Ø ØØØ Gattern und mehr durchaus tragbar. Zudem lohnt es sich, den Bitfeldextraktor auf dem Transistorniveau zu optimieren (z. B. mit CMOS- Transfergates).[3]
Der Aufwand wird durch den Leistungsgewinn in der Anwendung gerechtfertigt, da für wesentliche Aufgaben deutlich weniger Befehle benötigt werden. Das war beispielsweise für einen Hersteller Anlaß, in einer an sich auf RISC- Konzepten aufgebauten neuen Architektur entsprechende Lösungen vorzusehen.[4]

---

1 Weiterhin sind Verschiebungen für arithmetische Zwecke nutzbar. Das wird aber nicht auf der Architekturebene vorgesehen. Dort sind vielmehr alle elementaren arithmetischen Operationen unmittelbar definiert ("Shift" ist keine Grundoperation des numerischen Rechnens; vgl. Bild 17, S. 67).
2 Beispiele: IBM Stretch, Burroughs 17ØØ.
3 Zur Schaltungstechnik s. etwa /85/.
4 In der HP Precision Architecture (/229/) gibt es EXTRACT-Befehle, die ein Bitfeld aus einem Maschinenwort rechtsbündig in Register laden und DEPOSIT- Befehle, die ein Bitfeld aus einem Register in ein Maschinenwort einfügen.

6.5.1. Numerische Datenstrukturen

Es ist zu unterscheiden zwischen strukturell- deskriptiven Angaben und Näherungsdarstellungen für Zahlenbereiche des numerischen Rechnens.

Numerische strukturell- deskriptive Angaben sind binär codierte Ordinal- bzw. Kardinalzahlen, die geordnete endliche Mengen betreffen. Die maximale Mächtigkeit einer jeden solchen Menge wird auf $2^{31}-1$ Elemente festgesetzt. Damit reicht für jede Angabe ein 32-bit- Wort (mit Bit 31 = Ø als "Escape"- Vorkehrung) aus. Im Bereich von 1...32 bit sind beliebige Codon-Größen zugelassen, je nach der maximalen Mächtigkeit der betreffenden Menge.

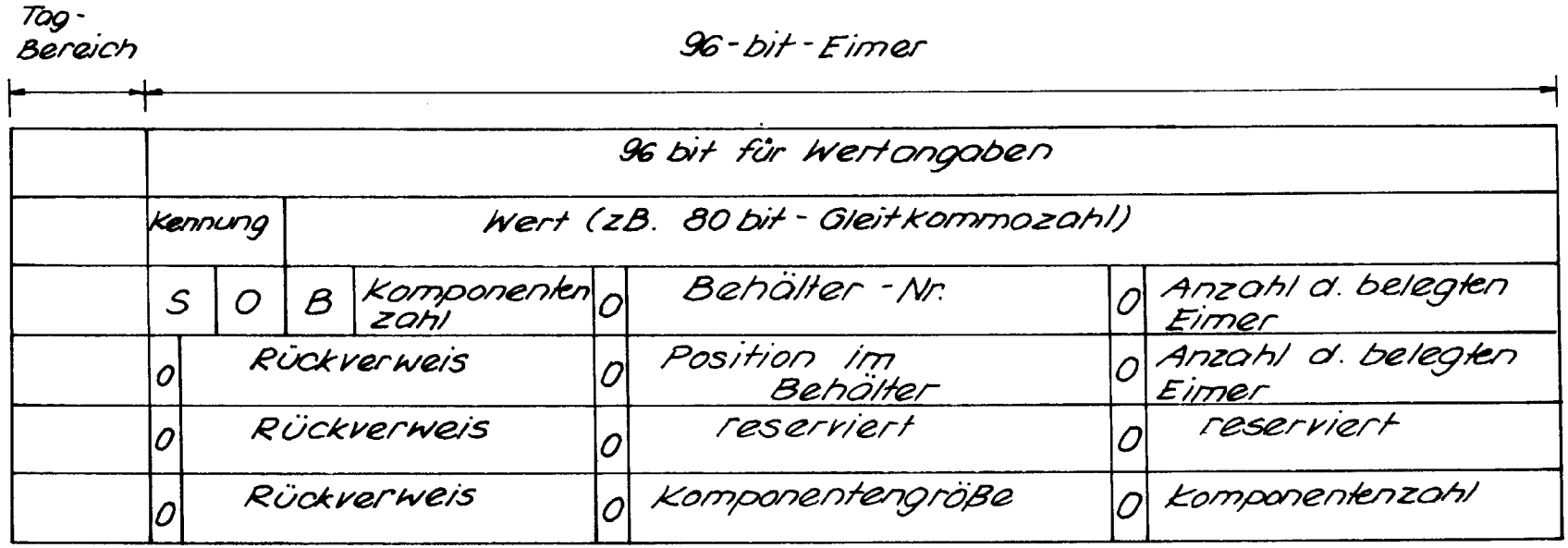
Für das numerische Rechnen wird ein Zahlenbereich angesetzt, wie er durch die üblichen Gleitkommadarstellungen überstrichen wird. Üblich sind 14 Exponentenstellen zuzüglich Vorzeichen.[1] Mit 15 bit kann man 32 768 diskrete Werte aufzählen. Jeder Wert wird als Bitposition in einem entsprechend langen Vektor aufgefaßt. Die Mantisse wird jeweils von der Position an in den Vektor eingefügt, die durch den Exponenten adressiert wird (vgl. Bild 21, S. 94). Alle Datenstrukturen beschreiben endliche Teilmengen aus diesem Intervall.
Ganze Zahlen sind Angaben mit Exponent Ø. Sie können 16, 24, 32, 48, 64 oder 96 bit lang sein. Interne Gleitkommazahlen haben ein 16-bit- Codon aus Mantissenvorzeichen, Exponentenvorzeichen und Exponenten sowie eine Mantisse von 32 oder 64 bit. Auf Grundlage der ganzen Zahlen werden rationale Zahlen (Form: a + b/c) eingeführt.[2]
Die Binärvektoren gemäß Bild 21 sind bis aufs Bit zugänglich, so daß beliebige Wandlungs- und Anpassungsroutinen geschrieben werden können.
Zur Vergegenständlichung vorgesehen sind die vier Grundrechenarten über Binärvektoren (die als natürliche bzw. ganze Binärzahlen aufgefaßt werden) sowie Operationen des Auswählens, Transportierens und Rundens. Damit läßt sich jede gewünschte Arithmetik implementieren.

6.5.2. Nichtnumerische Datenstrukturen

Die einzige grundlegende nichtnumerische Datentruktur ist der Binärvektor variabler Länge, der in Abschnitten von ebenfalls variabler Länge (1..64 bit) verarbeitet werden kann. Jeder beliebige Teilvektor ist selektierbar.
Strukturell- deskriptive Angaben werden im Rahmen des Eimer-Formats durch Aneinanderreihung der jeweiligen Codons gebildet. Bild 28 zeigt einige Beispiele.

---

1 Beispiele: IEEE 754 8Ø-bit- Format; Cray-1- Format.
2 Der Nenner (c) gibt die Genauigkeit an. Er gehört nicht zum Wert, sondern zum jeweiligen Deskriptor (die Genauigkeit wird deklariert, z. B. in Ada mit der delta- Angabe). Binär codierte Dezimalzahlen werden somit überflüssig. Dezimalzahlen sind rationale Zahlen mit einer Zehnerpotenz als Nenner.



1) Descripter enthält Wert des Objekts in 96 bit (z. B. 2 Gleitkommazahlen zu 48 bit)
2) Descripter enthält eine Kennung sowie den Wert des Objekts
3) Objektdescripter
4) Descripter einer Komponente in einem zusammengesetzten Objekt
5) 1. Eimer eines zusammengesetzten Objekts
6) 1. Eimer eines homogenen zusammengesetzten Objekts

Hinweis: Die Unterscheidung zwischen 1)...6) ist im TAG- Bereich codiert.

Bild 28 Beispiele für Descriptorformate (Auswahl)

6.6. Programmstrukturen

6.6.1. Operatoren

Allgemeine Transporte
Alle Binärvektoren sind bis aufs Bit adressierbar. Aus jedem Binärvektor können Abschnitte beliebiger Länge selektiert werden, und es lassen sich beliebige Abschnitte einfügen.

Strukturanalyse
Von jedem Binärvektor läßt sich ermitteln:

- die Position der ersten (niedrigstwertigen) Eins
- die Position der letzten (höchstwertigen) Eins
- die Anzahl der Einsen
- die Lage in einem Intervall, dessen Grenzen durch 2 weitere Binärvektoren gegeben sind (numerische Interpretation).[1]

Strukturwandlungen[2]
• Erzeugen eines Festwertes: Argumente sind eine Ordinalzahl n und ein Füllbit b. Es wird ein Vektor erzeugt, der an der Position 2 mit der Negation von b belegt ist und sonst mit dem Wert von b.

• Erzeugen eines Indexvektors: es wird ein Vektor erzeugt, der wahlweise nur die höchstwertige oder die niedrigstwertige Eins enthält.[3]

• Indexwert: es wird wahlweise die Position der höchstwertigen oder der niedrigstwertigen Eins ermittelt, und es entsteht weiterhin ein Resultatvektor, in dem die betreffende Postion gelöscht ist.

• Mischen: Bilden eines Resultatvektors durch bitweises Mischen aus 2 Argumentvektoren unter Steuerung eines Markierungsvektors (Übernahme des Bits aus dem 1. Vektor, wenn Maskenbit = Ø, sonst aus dem 2.) .

• Expandieren: Einfügen eines Vektors in die geraden bzw. ungeraden (wahlweise) Positionen eines anderen.

• Extrahieren: das Resultat wird wahlweise aus den geraden oder ungeraden Positionen eines Vektors gebildet.

Bitweise logische Verknüpfungen

- AND
- OR
- XOR.

---

1 Schaltungsvorschlag für Position der 1. Eins und Anzahl der Einsen in /239/.
2 Schaltungsvorschläge bzw. -anregungen in /240/.
3 Schaltungsvorschlag (für niedrigstwertige Eins) in /123/.

Jedes Argument kann selektiv komplementiert werden (damit sind auch NOR, NAND und Äquivalenz möglich). Weiterhin ist die Maskierung der Verknüpfung anschaltbar (nachgeordnete Konjunktion mit einem Maskenvektor). Es werden folgende Sonderbedingungen erkannt:

- Resultat enthält ausschließlich Nullen
- Resultat enthält ausschließlich Einsen.

Ternäre Verknüpfungen[1]
- Testen auf Orthogonalität
- Erzeugen eines Markierungsvektors
- variablenweises Wandeln
- ternäres Laden bzw. Transportieren.

Numerische Verknüpfungen
Es sind die 4 Grundrechenarten vorgesehen[2], wobei jeweils wählbar ist, ob der Binärvektor als natürliche oder als ganze Zahl aufzufassen ist. Bei der Division lassen sich Quotient und Rest wahlweise als Resultat zuordnen. Es werden folgende Sonderbedingungen erkannt:

- Resultat = Ø
- Überlauf
- Divisor = Ø
- Divisionsrest = Ø.

6.6.2. Selektoren

Es wird zunächst die Auswahl der kleinsten maschineninternen Verpackungseinheit (Eimer) beschrieben. Ist der Eimer einmal selektiert, so kann daraus jeder beliebige Binärvektor ausgewählt werden. Die Beschreibung der Selektoren ist nach aufsteigender Abstraktion (zunehmender Entfernung von der Maschinenebene) geordnet.

Adressierung auf Maschinenebene
Gemäß Bild 29 wird ein Eimer direkt in den Speichermitteln adressiert. Im Eimer wird ein Binärvektor ausgewählt.

Behälterangabe
Gemäß Bild 3Ø wird die Eimeradresse aus der Ordinalzahl des Behälters (Behältertyp + laufende Nummer) durch Tabellenzugriffe ermittelt.

Objektangabe
Gemäß Bild 31 wird aus dem Objektidentifier durch Zugriff zur Objekttabelle die Ordinalzahl des Behälters ermittelt. Diese wird gemäß Bild 3Ø weiterverarbeitet.

---

1 Als Quelle für Anregungen und Schaltungslösungen s. /24Ø/. Es ist notwendig, die erweiterten Anforderungen (z. B. für relationale Operationen) zu berücksichtigen.

2 Alle Operationen sind für das Abtesten von Bedingungen nutzbar (dann wird das Resultat nicht zugewiesen).

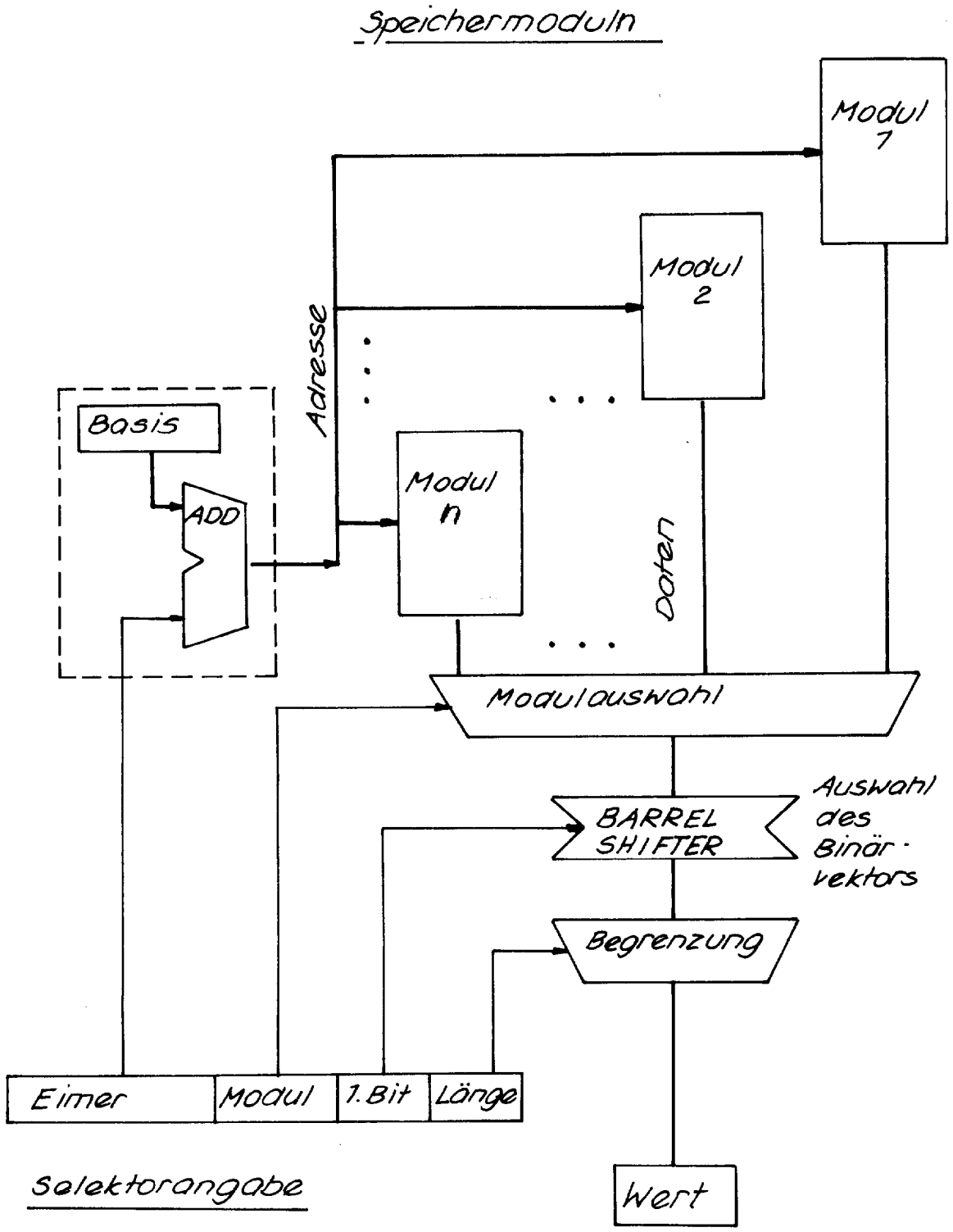
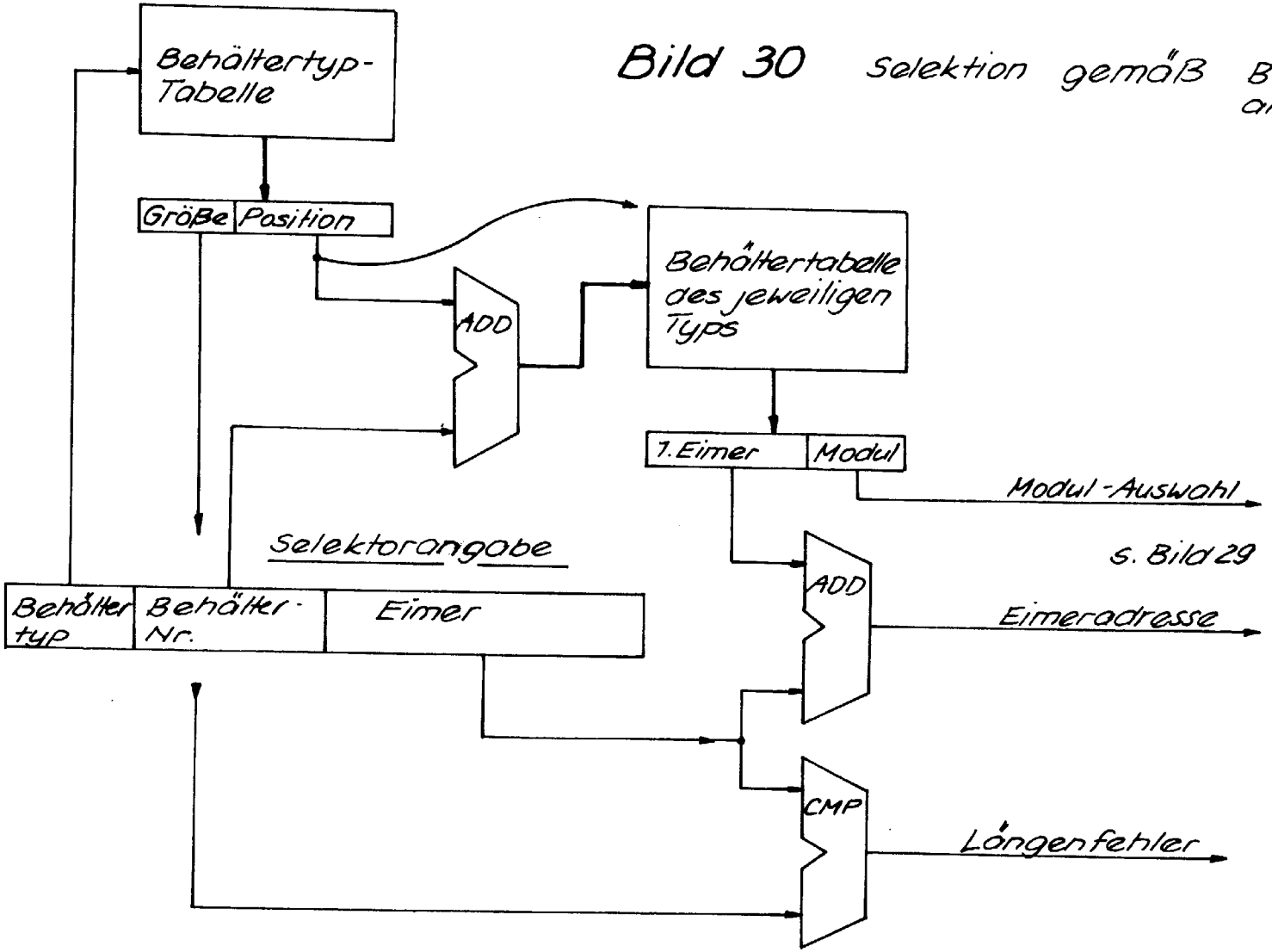


Bild 29 Adressierung auf Maschinenebene

Bild 30 Selektion gemäß Behälter-angabe



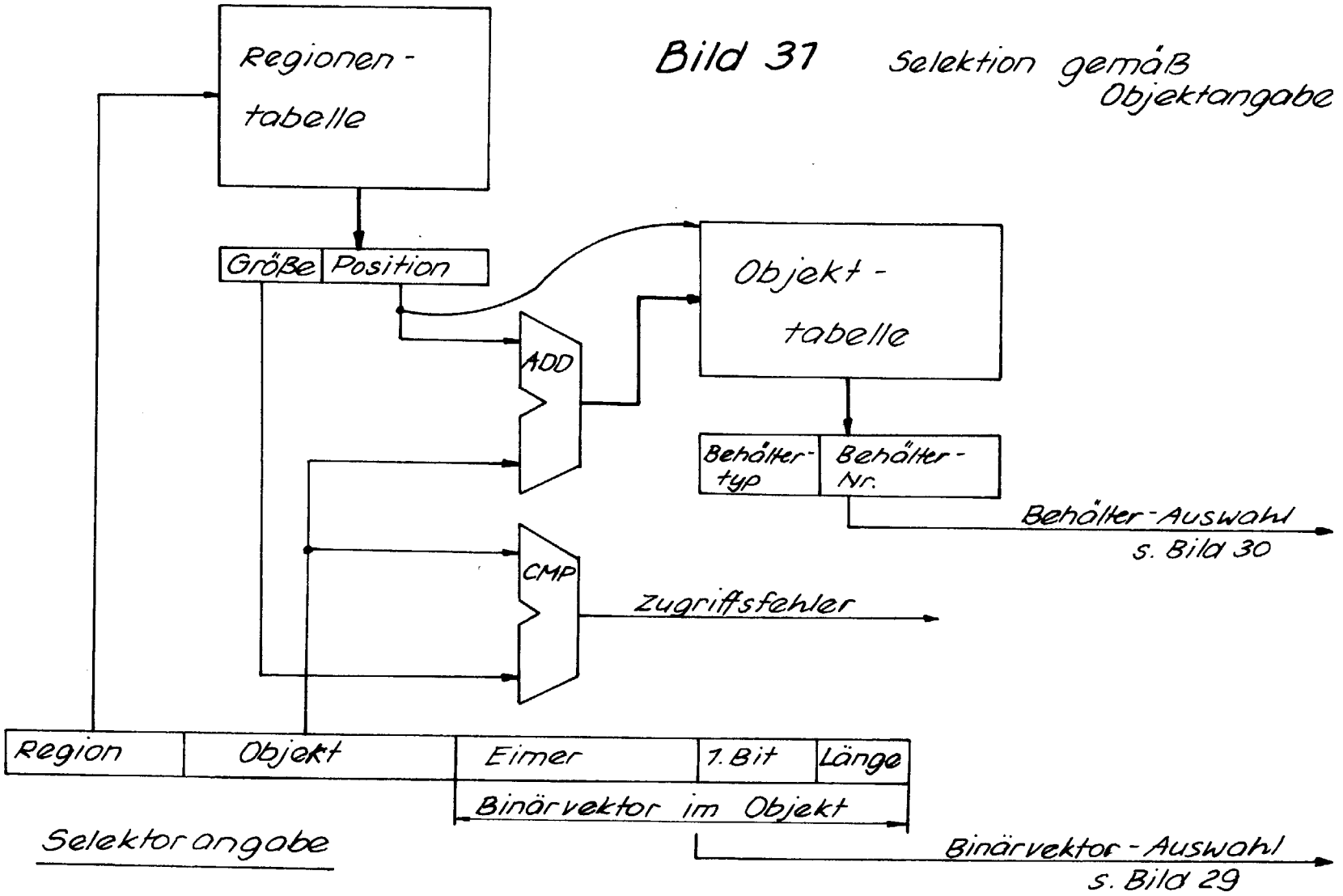


Bild 31 Selektion gemäß Objektangabe

Bezug auf Activation Record

Bild 32 zeigt, wie ein Objektidentifier aus einem Activation Record (im Bild mit ART bezeichnet) entnommen wird, der in einem Stackbereich gespeichert ist (weitere Umsetzung nach Bild 31). In jedem Programm sind 4 solche Bereiche zugänglich: der aktuelle, der des aufrufenden Programms, der des jeweils nächsten zu rufenden Programms und der des Systems).

Komponenten eines zusammengesetzten Objekts

Um eine ausgewählte Komponente adressieren zu können, sind Folgen von Zugriffen zu Deskriptoren notwendig. Jeder Deskriptor bestimmt Position und Größe des jeweils beschriebenen Bereichs (vgl. Bild 25). Aus TAG- Bits ist erkennbar, ob der betreffende Bereich wiederum Deskriptoren enthält oder bereits den Inhalt der jeweiligen Komponente darstellt. Dieser Ablauf ist vergegenständlicht, so daß alle Speicherzyklen für die unumgänglich notwendigen Deskriptorzugriffe nutzbar sind. Als Parameter wird eine Zugriffsliste (Liste von Ordinalzahlen zur Identifikation der betreffenden Komponente) übergeben. Der Ablauf wird beendet, wenn die Adresse des Inhalts gefunden bzw. wenn die Liste abgearbeitet wurde.

Allgemeine Tabellenzugriffe

Von wesentlicher Bedeutung sind Tabellen, die aus zusammenhängend gespeicherten gleich großen Einträgen bestehen (homogene zusammengesetzte Objekte). In einem bestimmten Eintrag ist ein Binärvektor zu selektieren. Im allgemeinen Fall umfaßt der Eintrag (hier als Element bezeichnet) eine gewisse Anzahl von Eimern. Die Anordnung nach Bild 33 ermittelt die Position des ersten Eimers aus der Ordinalzahl des Elements und dessen Größenbeschreibung. Der weitere Zugriff ist gemäß Bild 29 organisiert.[1]

Allgemeine Zugriffe zu beliebigen Grundstrukturen

Um der Forderung gewachsen zu sein, zu eingeführten Architekturen kompatible Adressierungsprinzipien nutzen zu können, wird das allgemeine Schema gemäß Bild 34 verwendet. Die grundlegende adressierbare Einheit ist der Beutel mit einer festen Länge von n Bit (bei n = 8: Byte- Adressierung). Die Adresse ist praktisch die Ordinalzahl des Beutels in der Menge aller Beutel. Daraus ermittelt die Anordnung nach Bild 34 die Adresse des Eimers und die erste Bitposition des betreffenden Beutels (weitere Verarbeitung gemäß Bild 29). Bei bekannter Beutelgröße ist nur eine Division zu vergegenständlichen; die Anzahl der Beutel im Eimer kann als Festwert bereitgestellt werden. Die Divisionsschaltung wird besonders einfach, wenn diese Anzahl eine Zweierpotenz ist.

---

1 In bekannten Architekturen werden die meisten Integer- Multiplikationen für solche Zwecke benutzt (eine Ursache dafür, daß in VLIW- Maschinen Gleitkomma- und Integer- Werke gleichzeitig beschäftigt werden können). Meist ist der Multiplikator kurz, weil das Element nur wenige Eimer umfaßt, und er steht auch häufig bereits zur Compilierzeit fest (in RISC- Konzepten ausgenutzt (/228/); für Vergegenständlichung aber genauer zu untersuchen).

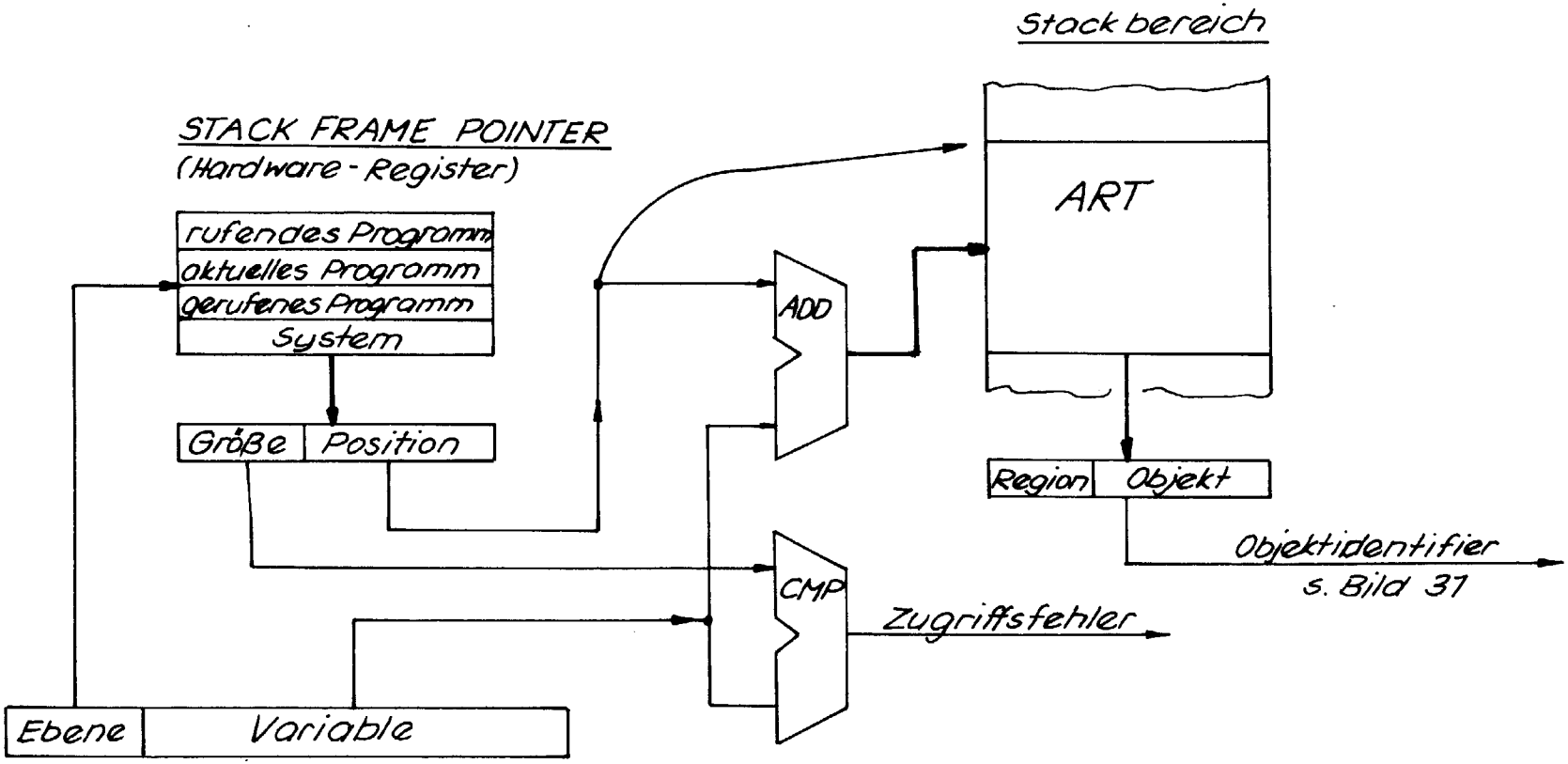


Bild 32 Selektion über ART

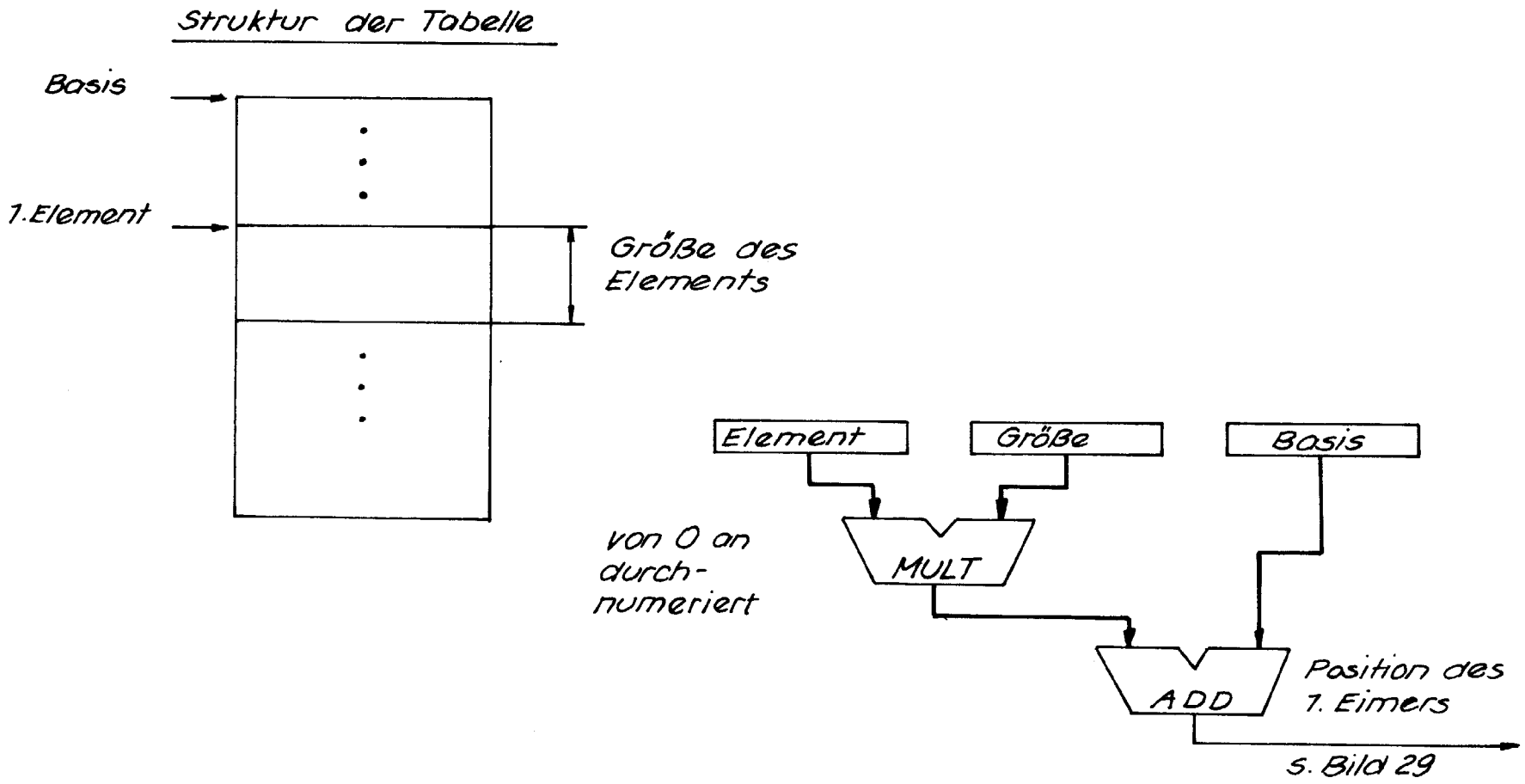


Bild 33 Tabellenzugriff

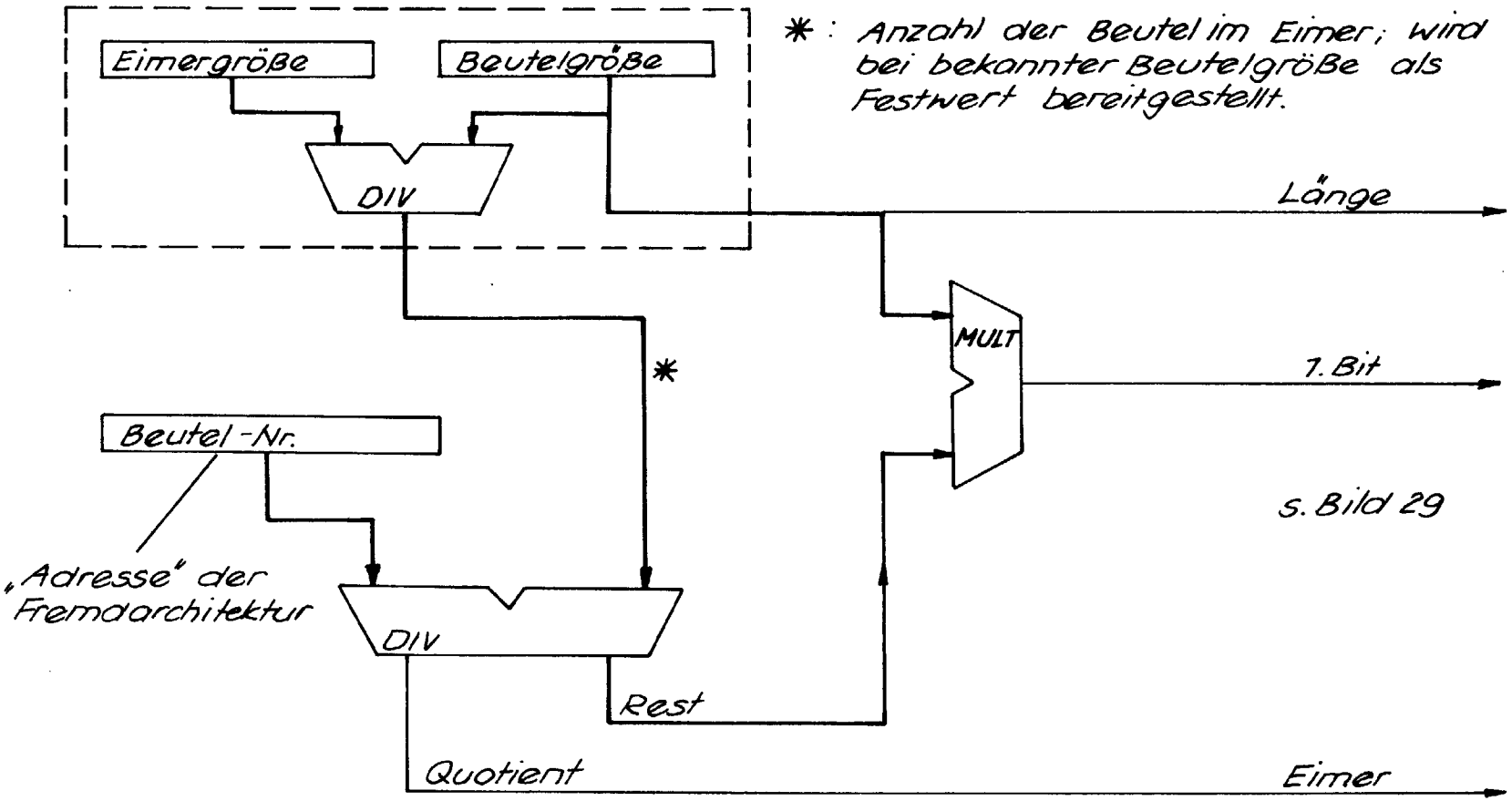


Bild 34 Zugriffe zu beliebigen Grundstrukturen (Beuteln)

6.6.3. Iteratoren

Ein einstufiger elementarer Iterator (der praktisch einer for-Schleife in üblichen Programmiersprachen entspricht), ist durch folgende Angaben bestimmt:

- Wert
- Schrittweite (vorzeichenbehaftet mit voller Länge gemäß Adressenbereich; das ist trotz des höheren Aufwandes der +1-Zählung vorzuziehen)[1]
- Begrenzung (Endwert oder Durchlaufzahl).

Bild 35 zeigt die im allgemeinen Fall notwendigen Schaltmittel. Beim Aufruf des Iterators wird der Anfangswert zum aktuellen Wert (AW -> i). Nach jedem Zugriff wird die Schrittweite zum aktuellen Wert addiert (i := i + S). Bei Erreichen der Begrenzung wird die Iteration abgebrochen.
Die Begrenzung läßt sich auf 3 verschiedene Weisen bestimmen:

1. Erhöhen eines Schrittzählers und Vergleichen mit einer vorgegebenen Schrittzahl (gleichwirkend: Vermindern der Schrittzahl auf Ø; das hat aber den Nachteil, daß die Schrittzahl- Angabe verlorengeht)

2. Vergleich des aktuellen Wertes i mit einer Bereichsgrenze T

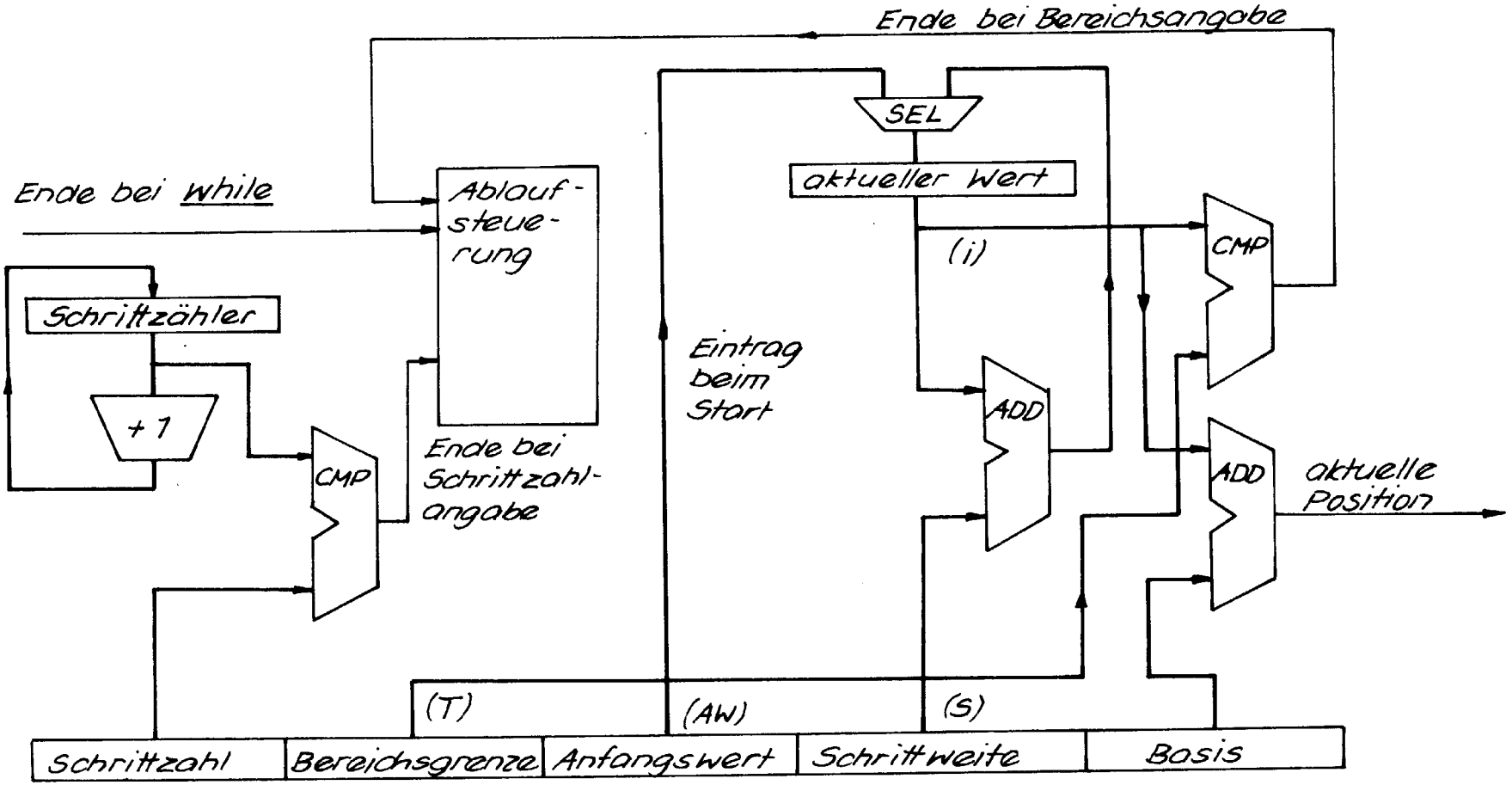
3. Auswerten von Abbruchbedingung, die von anderen Schaltmitteln (z. B. einem arithmetischen Vergleicher) geliefert werden (in datenabhängigen while- Schleifen).

Für ineinandergeschachtelte Schleifen sind mehrstufige Iteratoren erforderlich (Bild 36). Hier wird bei Erreichen der Grenze in einer Stufe die nächste Stufe aktiviert und die Grenzbedingung der vorhergehenden Stufe wieder zurückgestellt. Bild 36 zeigt einen 3-stufigen Iterator, wobei Anfangswert, Bereichsgrenze und Basis nicht dargestellt sind (s. dazu Bild 35). Zu Beginn der Nutzung des Iterators wird der Anfangswert (AW in Bild 35) in alle Register für aktuelle Werte eingetragen (Leitungen a) in Bild 36). Jede Stufe hat ein solches Register, aber nur das Register der ersten Stufe (entspricht der innersten Schleife) liefert eine verwertbare Adressenangabe (b) in Bild 36). Ist die innerste Schleife abgearbeitet (Vergleichsbedingung C1 wird aktiv), so wird der Schrittzähler der 1. Stufe auf Ø gesetzt, und in der 2. Stufe wird ein Schritt ausgeführt. Der dort ermittelte aktuelle Wert wird in die erste Stufe überführt, so daß er dort für den nachfolgenden Durchlauf der innersten Schleife als neuer Anfangswert

---

1 Das kostet einen kompletten Addierer und ein Schrittweitenregister im Vergleich zum Zählnetzwerk bei +1- Zählung. Damit lassen sich aber z. B. Zugriffe zu Matrizen in Zeilen-, Spalten- oder Diagonalrichtung gleich schnell ausführen (weitere Voraussetzung: entsprechende Speicherorganisation). Maschinen, in denen nur die +1- Iteration beschleunigt ist (z. B. CDC 2Ø5) bereiten größere Schwierigkeiten bei der Programmoptimierung.

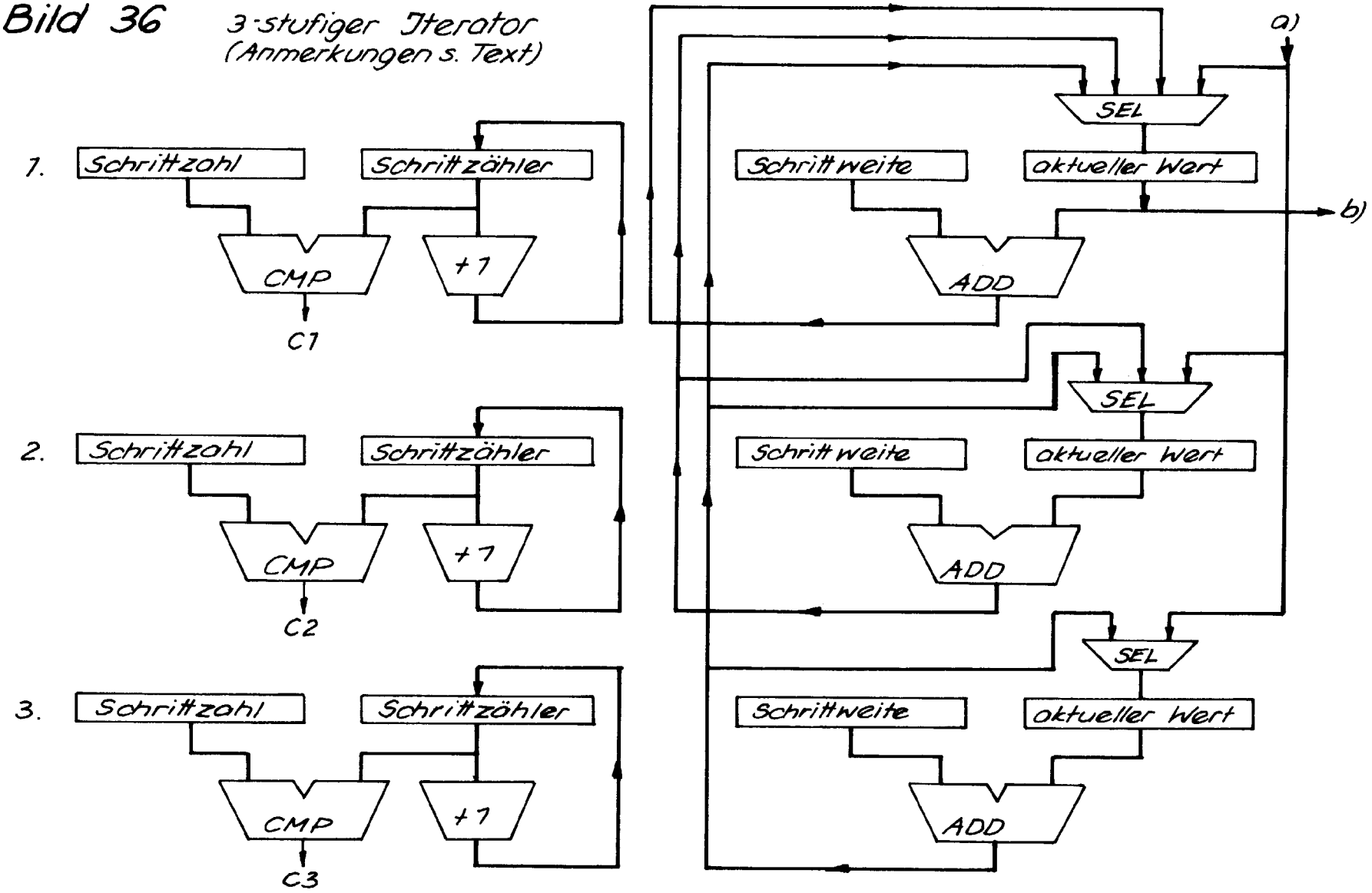
Beispiel: for i = AW step S until T



Iterationsangaben

Bild 35 Einfacher Iterator (Anmerkungen s. Text)

Bild 36 3-stufiger Iterator
(Anmerkungen s. Text)



verfügbar ist. Wurde in der 2. Stufe die angegebene Schrittzahl abgearbeitet (C2 wird aktiv), so wird der Schrittzähler gelöscht und in der 3. Stufe ein Schritt ausgeführt. Der dort ermittelte aktuelle Wert wird in die beiden vorhergehenden Stufen zurückgeführt. Die Iteration wird abgebrochen, wenn nacheinander alle 3 Vergleichsbedingungen C1, C2, C3 aktiv werden (bzw. durch Erreichen der absoluten Bereichsgrenze T gem. Bild 35 oder durch externe Bedingungen).
Nach diesem Prinzip lassen sich auch Iteratoren mit mehr Stufen aufbauen. Wesentlich ist, daß der in einer höheren Stufe errechnete aktuelle Wert in alle jeweils niederen Stufen zurückgeführt wird.
Es ergibt sich die Frage, wieviele Stufen zu vergegenständlichen sind. Von der Erfahrungstatsache ausgehend, daß häufig 2-dimensionale Strukturen vorkommen, deren Komponenten abschnittsweise zu verarbeiten sind, kann man zunächst 3 Stufen vorsehen:

1. Stufe: wählt die Abschnitte der einzelnen Komponente
2. Stufe: wählt die Komponenten der ersten Richtung (z. B. in einer Spalte)
3. Stufe: wählt einen Komponenten- Vektor der zweiten Richtung.

Es ist zwischen technischen und logischen Iteratoren zu unterscheiden. Technische Iteratoren verarbeiten Maschinenangaben, und sind den Selektor- Vergegenständlichungen nachgeordnet (sie beziehen ihre Anfangsangaben von Schaltmitteln ähnlich Bild 29...33). Logische Iteratoren verarbeiten Ordinalzahlen, und sind diesen Schaltmitteln vorgeordnet; sie liefern ihnen die Werte, aus denen die Maschinenadressen ermittelt werden.
Erfahrungsgemäß sind Iterationen über Komponenten zusammengesetzter Objekte weit mehr leistungsbestimmend[1] als solche über eine Vielzahl einzelner Objekte; somit sind vorzugsweise technische Iteratoren zu vergegenständlichen.

6.6.4. Aktivatoren

Synchrone Aktivatoren sind in bekannter Weise als Verzweigungen vorgesehen. Es sind die Ordinalzahlen der möglichen Nachfolgebefehle angegeben sowie Auswahlcodons für die Bedingungen, unter denen jede dieser Ordinalzahlen wirksam wird. Verzweigungen sollten keine zusätzlichen Maschinenzyklen kosten; Unterprogramme der untersten Ebene (d. h. solche, die ihrerseits keine Unterprogramme mehr rufen) sollten ohne Zeitverlust aufrufbar sein.
Es ist nicht sinnvoll, Verzweigungs- Konstrukte höherer Programmiersprachen unmittelbar zu vergegenständlichen.[2]

---

1 Das ist die häufigste Nutzung der innersten Schleifen.
2 Ein solcher Vorschlag wurde in /3Ø1/ dargelegt. Kritik:
  a) if...then...else ist mit üblichen Verzweigungen verlustlos zu implementieren
  b) case ist unproblematisch, wenn schnelle Tabellenzugriffe vergegenständlicht sind.

Für die Gestaltung asynchroner Aktivatoren gibt es hinreichend Anregungen seitens der eingeführten Architekturen. Schließlich kommt es nur darauf an, den Kontext des aktuellen Programms zu retten und die Abarbeitung eines anderen Programms zu starten. Die Kontextumschaltung ist unproblematisch, wenn alle architekturseitig definierten Register in einem Universalregistersatz zusammengefaßt sind (wie bei den meisten RISC- Maschinen): man sieht mehrere Registersätze vor und braucht nur noch zwischen ihnen umzuschalten.[1]
Sind alle Register direkt mit den jeweiligen Verknüpfungsschaltungen verbunden (für höchste Leistung an sich unumgänglich), so bereitet die Kontextumschaltung allerdings einige Schwierigkeiten. Jede technische Sondervorkehrung[2] braucht Platz auf den Schaltkreisen, das bedeutet eine Verlängerung der Zykluszeit.
Daher soll das bekannte Prinzip genutzt werden, den Kern eines Hochleistungsrechners weitgehend von der Unterbrechungsbearbeitung freizuhalten.[3]
Für die Behandlung von Programmausnahmen kann man die Vorkehrungen so treffen, daß der einleitende Ablauf nur wenige Register gezielt retten muß (der Ablauf sucht im aktuellen Activation Record nach, ob im Programm ein Behandler definiert ist; er startet diesen oder leitet eine systemseitige Maßnahme ein). Externe Unterbrechungen werden nur dann zugelassen, wenn die meisten Register frei sind. Das ist der Fall nach der Zuweisung errechneter Resultate auf Objekte im Laufzeitspeicher. Die Unterbrechung wird also nicht nach jedem Befehl, sondern nur nach dem Beenden von Funktionsaufrufen angenommen.

6.6.5. Befehlsgestaltung

Es ist naheliegend[4], Konzepte, die für leistungsfähige Mikroprogrammsteuerungen entwickelt wurden, direkt für die Architekturdefinition zu nutzen.[5] Im besonderen sind folgende Eigenschaften von Interesse:

1. Jeder Befehl steuert nur einen Maschinenzyklus.[6]

2. Alle Angaben zur Ablaufsteuerung im betreffenden Zyklus sind im (hinreichend langen) Befehlswort parallel vorgesehen.

3. Jeder Befehl enthält alle Angaben zur Auswahl seines Nachfolgers.

---

1 Wie z. B. beim Mikroprozessor SAB 8Ø199 (/129/).
2 Z. B.: Austauschregistersätze, "Schattenregister" in RAM-Arrays, Anschluß aller Register an ein Bussystem.
3 Voraussetzung: Der Hochleistungsrechner muß mit unabhängig arbeitenden E/A- Prozessoren ausgestattet sein. Das wurde bereits in der CDC 66ØØ verwirklicht.
4 Vgl. S. 21 und S. 61, Punkt 1.
5 Dieser Abschnitt bietet einige praxisbezogene Überlegungen. Für allgemeinere Betrachtungen s. Abschnitt 7.2., S. 136 ff.
6 Zumindest für die leistungsentscheidenden Abläufe in den größeren Hardware- Modellen.

4. Spätverzweigung[1]. Bei diesem Prinzip hat jeder Befehl nur eine kleine Anzahl möglicher Nachfolger (z. B. 2 oder 4). Diese werden im aktuellen Zyklus parallel gelesen. Steht am Ende des Zyklus die Verzweigungsbedingung fest, so kann der betreffende Folgebefehl ohne Zeitverlust wirksam werden, da nur ein einfaches Auswahlnetzwerk zu durchlaufen ist.

Solche Prinzipien müssen auf der Architekturebene nutzbar werden. Bisher wurden Mikrobefehlsformate vorwiegend unter dem Gesichtspunkt ausgelegt, die Hardware für eine bestimmte Aufgabe (z. B. die Emulation einer CISC- Architektur) leistungsfähig und kostengünstig gestalten zu können. Damit waren oft Einschränkungen verbunden, die z. B. einem Compilerautor nicht zugemutet werden können (geringe zulässige Entfernungen von Sprungzielen, Zwang zur Voreinstellung von Befehlswirkungen usw.).[2] Um solche Einschränkungen aufzuheben, braucht man an sich nur einige Bitpositionen mehr.[3]

---

1 Das Prinzip wurde in den Zentraleinheiten und Gerätesteuerungen der Systeme /360 und /370 ausgiebig genutzt. Erläuterungen z. B. in /53/ und /235/, Folge 17.

2 Beispiel: EC 1055. Selbst wenn es gelingt, für solche Befehlsgestaltungen Compiler zu schaffen: die Effektivität wird unbefriedigend bleiben, da viele Befehle in den compilierten Mikroprogrammen nur deshalb erforderlich sind, um die Beschränkungen zu umgehen.

3 Auch ein Grund für 96 bit gegenüber 64 bit. Die Zusatzaufwendungen in den Steuerschaltungen sind unbedeutend.

7. Wirkprinzipien und Schaltungsstrukturen

7.1. Grundlagen des Zusammenfassens von Vergegenständlichungen

Ein einzelner Algorithmus läßt sich in einer Einzweckmaschine gemäß Bild 37 vergegenständlichen. In einer solchen Maschine sind Speichermittel für Argumente und Resultate, Selektionseinrichtungen, Verknüpfungsschaltungen und Steuermittel in algorithmenspezifischer Weise zusammengeschaltet. Durch einen gemeinsamen Datenpfad (z. B. ein Bussystem) ist gewährleistet, daß die Speicherinhalte untereinander austauschbar sind; damit ist das Schema grundsätzlich auf beliebige Algorithmen anwendbar. Alle Schaltmittel sind an die Erfordernisse des betreffenden Algorithmus (Datenstrukturen, Operationen, Steuerabläufe usw.) angepaßt, so daß im Rahmen der technischen Aufwendungen das theoretisch höchste Leistungsvermögen in Bezug auf den vergegenständlichten Algorithmus erbracht wird (Datenstrukturmaschine; Abschnitt 2.3.5., S.25 f.). Demgegenüber muß eine Universalmaschine es gestatten, eine Vielzahl vergegenständlichter Algorithmen im Verbund anzuwenden.

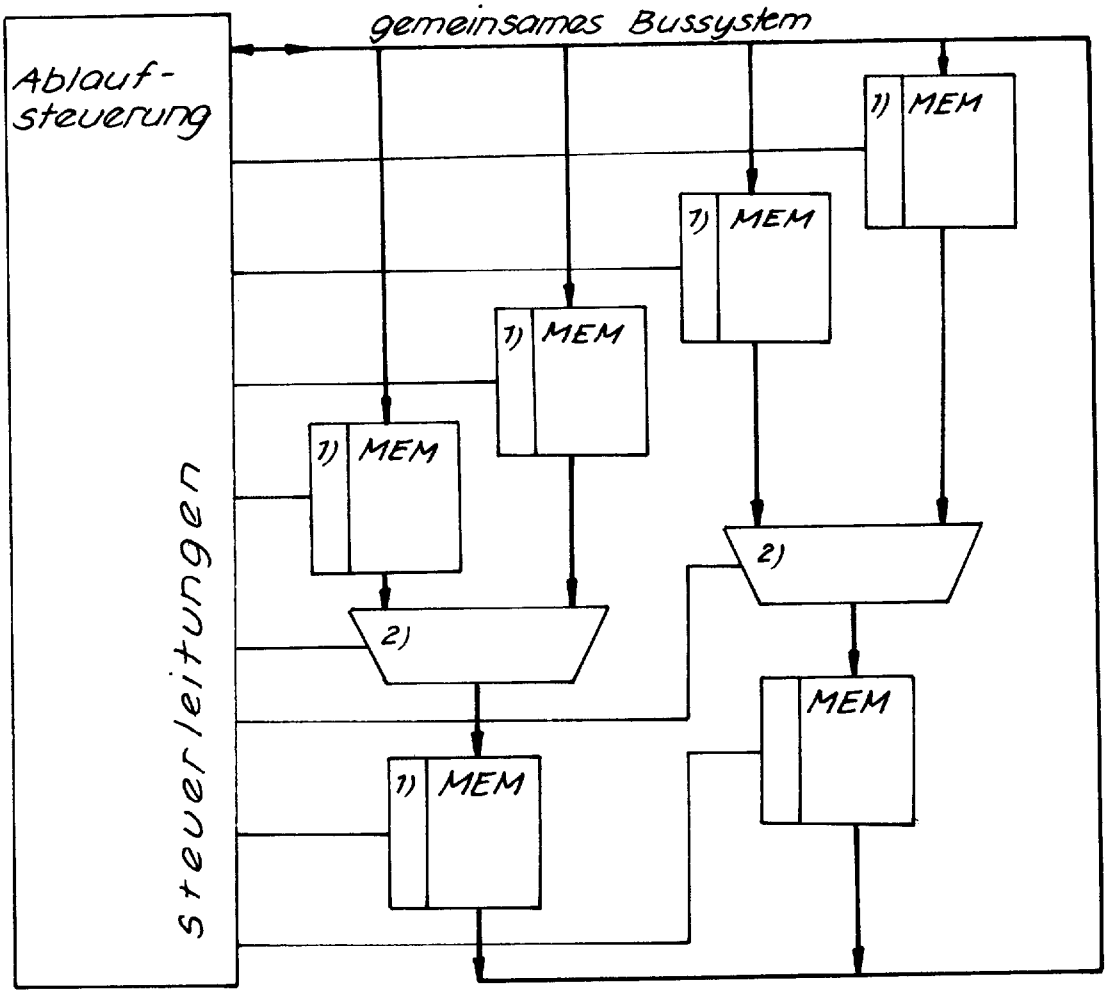
Zunächst soll untersucht werden, ob sich eine Universalmaschine bauen läßt, indem eine Anzahl von Einzweck- bzw. Spezialmaschinen (Datenstrukturmaschinen) unterschiedlicher Zweckbestimmung miteinander verbunden wird (Bild 38). Jede einzelne Maschine läßt sich auf höchste Leistung hin auslegen. Speicherinhalte lassen sich unter zentraler Steuerung freizügig austauschen, so daß es möglich ist, alle vergegenständlichten Algorithmen in beliebiger Kombination anzuwenden. Die einzelnen Maschinen sind parallel nutzbar, wenn dafür unabhängige Verarbeitungsaufgaben vorhanden sind. Diese Struktur läßt sich in eine Maschine mit gemeinsamen Speichermitteln überführen (Bild 39).[1] Die Maschine nach Bild 38 sei mit A bezeichnet, jene nach Bild 39 mit B.
B hat eine größere Zykluszeit als A ($t_c B > t_c A$), da alle Zeitverhältnisse durch die gemeinsame (und folglich physisch ausgedehnte) Speicheranordnung bestimmt werden. Hingegen sind bei A Transportabläufe zwischen den verschiedenen Speichermitteln notwendig, bei B nicht. Voraussetzungsgemäß sind in A und B dieselben Algorithmen vergegenständlicht: die Ausführung einer Anwendungsaufgabe erfordert also in A und B gleichermaßen z Zyklen. Zusätzlich sind in A noch v Transportzyklen notwendig. A ist überlegen, wenn gilt[2]

$$(z + v)\, t_c A < z\, t_c B\,. \qquad (7.1)$$

---

1 Dazu werden die Überlegungen von Abschnitt 3.1. (S. 32 ff.) sinngemäß nachvollzogen. Hier entsteht aber keine gewöhnliche v. Neumann- Maschine, sondern eine Anordnung, in der die Verknüpfungs-, Auswahl- und Steuerschaltungen der Datenstrukturmaschinen von Bild 38 mit gemeinsamen Speicher- und Steuermitteln verbunden sind.

2 Für konkrete Vergleiche sind in (7.1) Mittel- bzw. Erwartungswerte aus einer Vielfalt repräsentativer Anwendungsaufgaben einzusetzen. (Betrifft auch $t_c A$ bzw. $t_c B$, falls ablaufspezifisch verschiedene Taktzyklen vorgesehen sind.)



MEM: Speicher

1) : Selektionseinrichtungen

2) : Verknüpfungsschaltungen

Bild 37 Einzweckmaschine

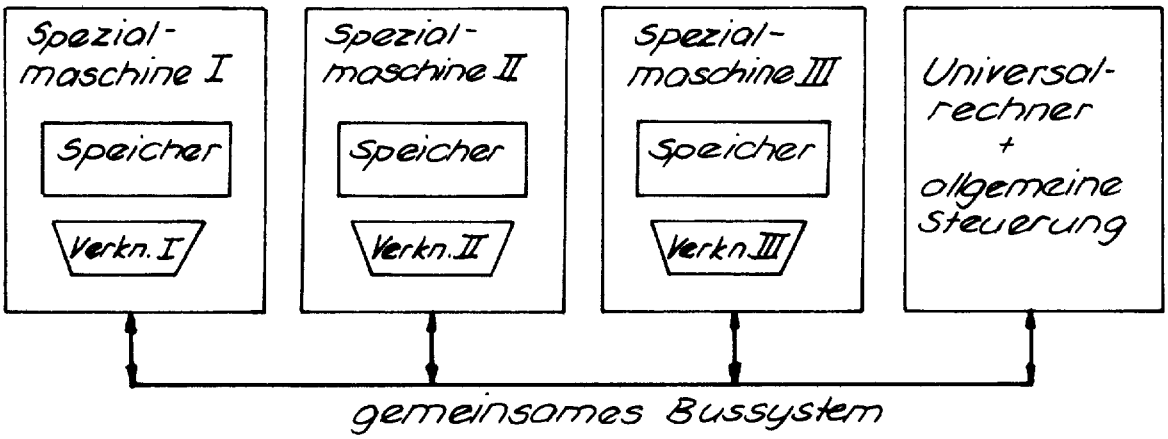


Bild 38 Datenstrukturmaschine aus gekoppelten Spezialmaschinen und Universalrechner

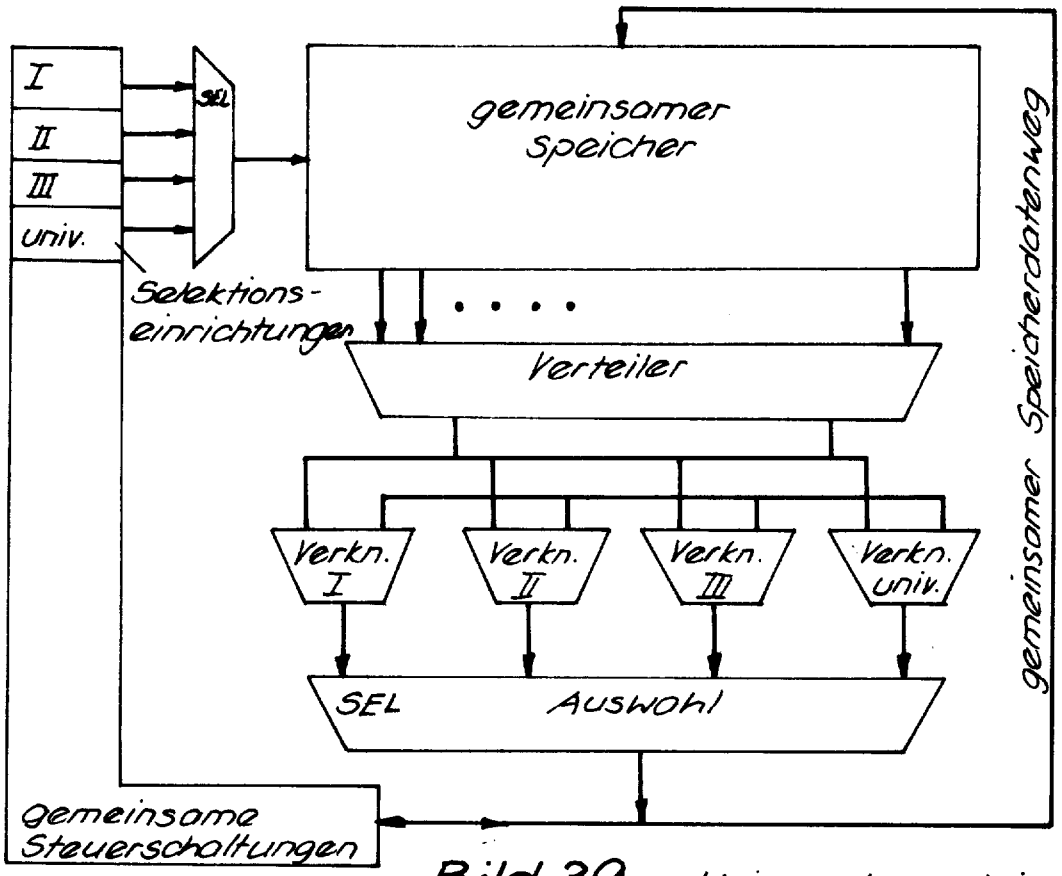


Bild 39 Universalmaschine

Es ist ersichtlich, daß A nur dann wirklich überlegen ist, wenn der Datenaustausch zwischen den einzelnen Maschinen vernachlässigt werden kann (also bei weitgehend unabhängiger Parallelarbeit); ansonsten müßte, um die Überlegenheit von A sicherzustellen, $t_cA << t_cB$ sein. Das ist aber praktisch kaum erreichbar (erfahrungsgemäßer Richtwert: $t_cA = 3/4..1/8\ t_cB$).[1] Im recht häufigen Fall, wenn die einzelnen Algorithmen in wenigen Zyklen ausgeführt werden können (z klein) und wenn ein intensiver Datenaustausch nötig ist (v groß), sind diese Verhältnisse kaum ausreichend, um eine eindeutige Überlegenheit für A sicherzustellen. Eine Vielzahl in sich optimierter Maschinen mit jeweils eigenen Speichern ist also nicht von vornherein eine besonders günstige Lösung für einen Universalrechner. Anzustreben sind vielmehr schnelle und universell nutzbare (d. h. für alle vergegenständlichten Algorithmen freizügig adressierbare) Speichermittel. Geschwindigkeit (kurze Zugriffszeiten für die jeweiligen Verknüpfungsschaltungen) und universelle Nutzbarkeit (jeder Speicherinhalt ist für alle Verknüpfungsschaltungen zugänglich) sind Forderungen, die einander widersprechen, da sie unterschiedliche technische Auslegungen bedingen (zum einen direkten Anschluß der Speicheranordnungen an die jeweiligen Verknüpfungsschaltungen, zum anderen Mittel zum wahlfreien Mehrfachzugriff, wie Vermittlungseinrichtungen, Koppelnetzwerke oder Bussysteme).
Der Lösungsansatz ist gekennzeichnet durch funktionelle Zuordnung und damit durch Parameterübergabe über Werte ("call by value"). Alle vergegenständlichten Algorithmen sind funktionelle Zuordnungen der untersten Ebene. Die erforderlichen Angaben werden zwecks Verknüpfung in technisch zweckgerecht ausgebildeten Speichermitteln (Registern, Schnellspeichern) gehalten. Um die Universalität zu gewährleisten, wird eine Speicherhierarchie aufgebaut (s. Bild 27, S.111), die in der obersten Ebene eine gemeinsame Datenbasis aufnimmt. Diesen Speichermitteln sind jene nachgeordnet, die die aktuell zu verarbeitenden Informationsstrukturen aufnehmen (Speicher der Laufzeitumgebung). An diese wiederum sind die Speichermittel angeschlossen, die die Argumente und Resultate der vergegenständlichten Algorithmen aufnehmen (die Hardware- Register bilden die unterste Ebene dieser Speicherhierarchie). Je höher die Speicherebene, umso größer die vorzusehende Speicherkapazität. Wesentlich ist, daß bei deren Verwaltung nicht mehr wahlfreie Zugriffe (wie in den untersten Ebenen) leistungsbestimmend sind, sondern Blocktransporte. Sie sind vergleichsweise einfach zu beschleunigen. Die Transportgeschwindigkeit wird durch die Anzahl der Datenleitungen bzw. zugänglichen Datenanschlüsse der Speicher[2] und durch die Zugriffszeit bestimmt, wobei viele Halbleiterspeicher für Zugriffe zu aufeinanderfolgenden Adressen eine Verkürzung ermöglichen (z. B. Column Access bei DRAMs).

---

1 Praktisch abhängig von physischer Ausdehnung, Kompliziertheit des Taktschemas und technologischen Aufwendungen. Man kann auch ausgedehnte, aus vielen Leiterplatten bestehende Anordnungen mit einem Taktzyklus von < 10 ns betreiben (Cray X-MP, Cray-2); t_cB kann also auch sehr kurz sein.

2 Vgl. die Ausführungen in /243/.

7.2. Ablaufsteuerprinzipien und deren Codierung

In den üblichen Rechnerarchitekturen werden die Abläufe durch Maschinenbefehle codiert. Ein Befehl enthält Codons für die Aktivierung der vergegenständlichten Algorithmen und für die jeweils notwendigen Parameter (Adressenangaben bzw. Direktwerte). Befehlslisten werden oft nach heuristischen Kriterien, in Anlehnung an ein Vorbild oder ein bestimmtes Verarbeitungsmodell bzw. auf Grund von Betrachtungen zur Nutzungshäufigkeit aufgestellt. Das in Abschnitt 3 beschriebene Modell der Informationsverarbeitung gestattet einen allgemeineren Zugang zu dieser Aufgabe:
Es sind bestimmte Ressourcen $\mathcal{R}$ gegeben. Für jeden Algorithmus $\mathcal{A}$, der damit ausgeführt werden soll, ist die gewünschte Informationswandlung durch eine zeitsequentielle Nutzung dieser Ressourcen $\mathcal{R}$ zu bewerkstelligen.
Allgemein läßt sich so eine beliebige Rechnerstruktur[1] im Rahmen des Verarbeitungsmodells auffassen als eine Ansammlung von Ressourcen (Speichermittel, Informationswege, Verknüpfungsschaltungen) mit Steuerschaltungen zu deren zeitsequentiellen Nutzung und mit Speichermitteln, die die Steuerangaben enthalten (Bild 40). Diese Auffassung schließt beide Extremfälle ein:

1. die v. Neumann- Maschine: der Steuerspeicher wird sequentiell abgefragt; in jedem diskreten Zeitschritt bestimmt der gelesene Inhalt des Steuerspeichers eindeutig die Nutzung der Ressourcen (bei der reinen v. Neumann- Maschine ist zudem der Steuerspeicher mit den allgemeinen Speicherressourcen technisch identisch)

2. die Datenflußmaschine: der Steuerspeicher wird assoziativ abgefragt, und zwar auf Grundlage der jeweils verfügbaren Ressourcen und verarbeitungsbereiten Argumentwerte.

Diese Extremfälle zeigen folgende Sachverhalte auf:

1. Beide Konzepte haben grundsätzliche Grenzen:
Der v. Neumann- Rechner ist leistungsbeschränkt, da ausschließlich die abgerufenen Steuerangaben über die Nutzung der Ressourcen entscheiden; der innewohnende Parallelismus ist zur Laufzeit nicht erkennbar. Die reine Datenflußmaschine erfordert einen voll assoziativen Steuerspeicher; das ist praktisch nicht zu verwirklichen: das Datenflußprinzip ist aufwandsbeschränkt; es ist nur teilweise realisierbar.

2. Es ist grundsätzlich möglich, technische Lösungen anzugeben, die zwischen beiden Extremen liegen.

Ein Rechner erbringt dann seine höchste Verarbeitungsleistung, wenn in jedem internen Zyklus alle Ressourcen mit nützlicher (d. h. zum angestrebten Endresultat beitragender) Arbeit beschäftigt sind. Eine ideale Datenflußmaschine würde dies ge-

---

1 Einzige Voraussetzungen: aussagenlogische Verknüpfungen, binäre Speicher, getaktete Arbeitsweise.

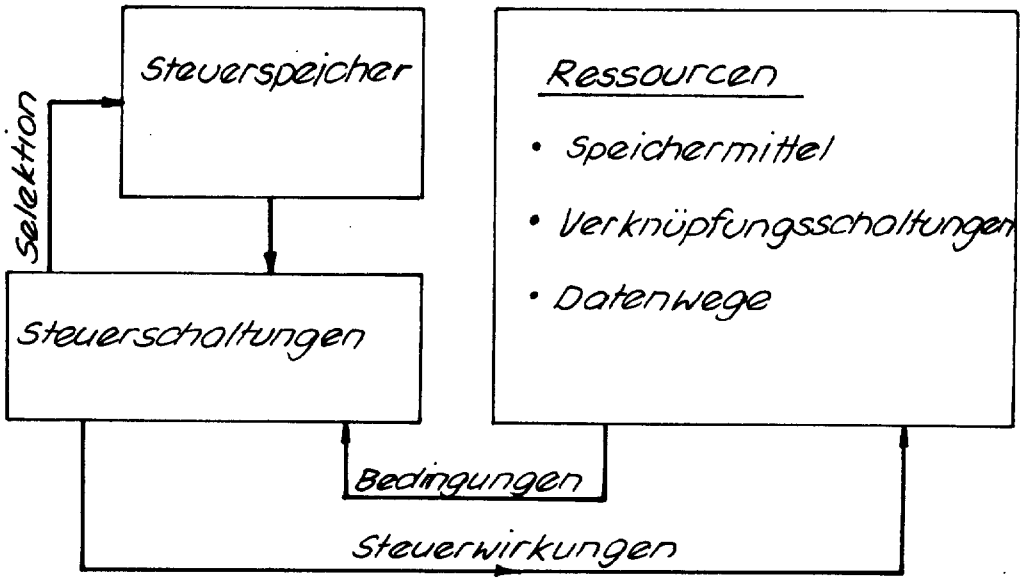


Bild 40 Allgemeines Modell einer Rechnerstruktur als Anordnung von Ressourcen

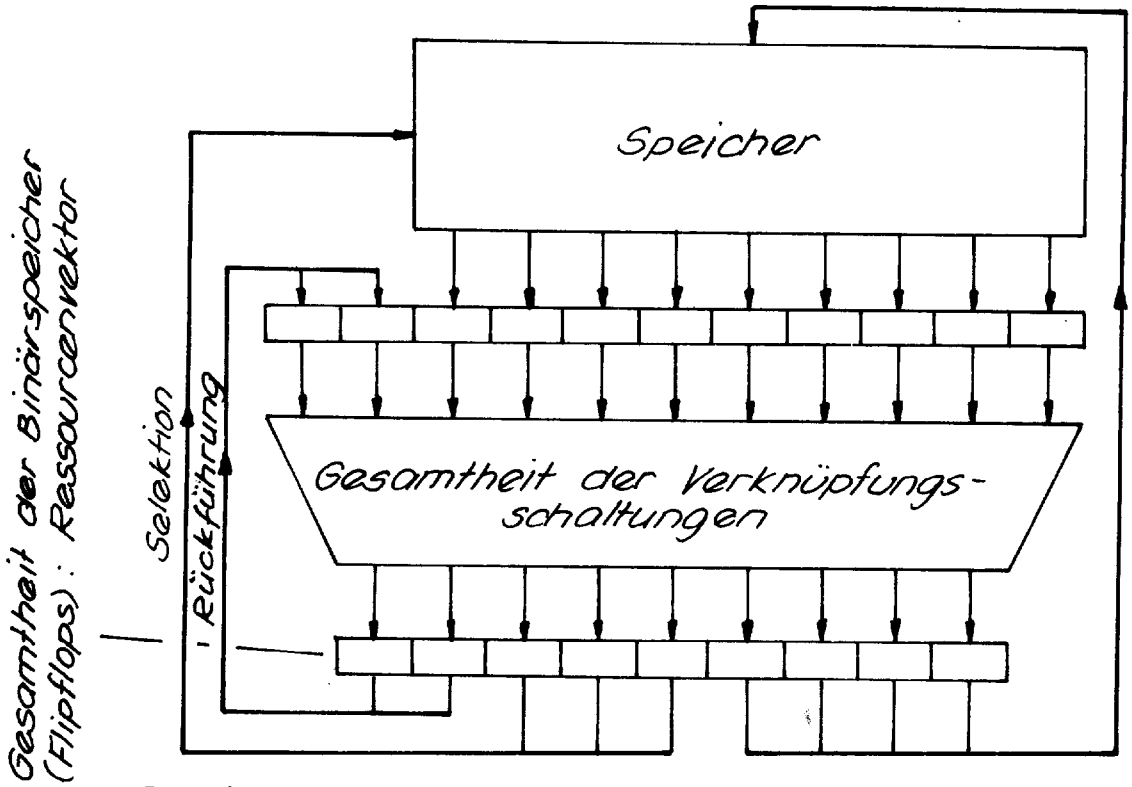


Bild 41 Grundsätzliche Gestaltung der Ressourcen

währleisten; sie läßt sich aber nicht bauen. Mit praktikablen technischen Mitteln kann man also nur eine näherungsweise Lösung anstreben. Das führt auf 2 Teilaufgaben:

1. die Bestimmung der Ressourcenbelegung in jedem Zeitschritt

2. die technische Steuerung der Ressourcen.

Dem Stand der Technik gemäß kann die Ressourcenbelegung zur Compilierzeit oder zur Laufzeit bestimmt werden. Wird diese Aufgabe in die Laufzeit verlegt, so sind zusätzliche Schaltmittel notwendig, deren Wirkung darauf hinausgeht, eine Datenflußsteuerung ausschnittsweise nachzubilden und dazu den tatsächlichen Datenfluß gewissermaßen "live" aus dem Befehlsstrom zu erkennen. Wird die Ressourcenbelegung zur Compilierzeit bestimmt, so erspart dies die einschlägigen Schaltmittel, die immer verlangsamend wirken, sei es durch zusätzliche Belastung und physische Vergrößerung (Signallaufzeiten), sei es deshalb, weil solche Einrichtungen Platz für eigentlich leistungstragende Schaltmittel (z. B. Rechenwerke) entziehen[1]. Zudem kann ein Compiler theoretisch den gesamten Programmtext übersehen: das läßt darauf hoffen, mehr innewohnenden Parallelismus erkennen zu können als durch Beobachten des Befehlsstroms zur Laufzeit.[2] Einwände[3] sind allerdings auch ernst zu nehmen: sie weisen auf die Notwendigkeit hin, technisch zweckmäßige Kompromißlösungen zu entwickeln.[4] Die Aufgabe des Compilers besteht dann darin, aus dem Programmtext die Steuerinformation für die jeweils gegebenen Ressourcen zu erzeugen. Besondere Schaltmittel haben die Aufgabe, zur Laufzeit jene Bedingungen zu verarbeiten, die ein Compiler nicht berücksichtigen kann (z. B. datenabhängige Schleifen) bzw. von denen er freigehalten werden sollte (z. B. Steuerung von Speicherzugriffs- Pipelines).
Sollen alle Ressourcen - wenn immer möglich - in jedem Zyklus parallel ausgenutzt werden, so ist es notwendig, alle Angaben gleichzeitig bereitzustellen. Die Ressourcen der untersten Ebene sind durch Flipflops bzw. Hardware- Register, Verknüpfungsschaltungen und die zugehörigen Informationswege gegeben. Gemäß Bild 41 wird die Gesamtheit der Speichermittel dieser Ebene (also Flipflops und Register) zu einem Ressourcenvektor zusammengefaßt. Teile des Ressourcenvektors benötigen Information von übergeordneten Speichermitteln, Teile liefern Resultate an übergeordnete Speichermittel, Teile stellen Selektionsangaben (Adressen) für übergeordnete Speichermittel bereit; schließlich sind Teile des Ressourcenvektors aufeinander zurückgeführt: dieses Schema gilt für die Feinstruktur jeder Rechenmaschine, die dem Verarbeitungsmodell von Abschnitt 3 entspricht.

---

1 Bezogen auf ein vorgegebenes Limit (z. B. Siliziumfläche).
2 Zur Laufzeit kann erfahrungsgenmäß nur der Parallelismus in einem Basisblock (Befehlsfolge zwischen zwei Verzweigungen) erkannt werden, das sind meist 5...1Ø Befehle.
3 Vgl. S. 19 bzw. /33Ø/.
4 Anregung: Milderung von Nachteilen (z. B. Compilierzeit) durch unterstützende Schaltmittel.

Von dieser Auffassung ausgehend kann man systematisch die Möglichkeiten untersuchen, die Ressourcen mit Information zu versorgen. Bild 42 veranschaulicht die allgemeinen Alternativen:

a) direkte Zuordnung. Alle Angaben gelangen parallel zu den jeweiligen Positionen des Ressourcenvektors. Das Prinzip ist technisch in Maschinen mit besonders langem Befehlswort (VLIW) bzw. besonders langen "horizontalen" Mikrobefehlen verwirklicht.

b) gruppenweise Auswahl. Es wird jeweils nur ein ausgewählter Teil des Ressourcenvektors geladen. Dazu ist eine Verteilerschaltung notwendig, die durch zusätzlich gespeicherte Auswahlcodons gesteuert wird. Das entspricht der üblichen Mikrobefehlsgestaltung mit Steuerfeldern, die selektiv jeweils bestimmte Schaltungskomplexe unmittelbar beeinflussen.

c) codierte Auswahl. Der Speicher liefert codierte Angaben, die über besondere Schaltmittel umcodiert und selektiv Teilen des Ressourcenvektors zugeführt werden. Das entspricht der herkömmlichen Auslegung der Maschinenbefehle.

Für genauere Untersuchungen müssen die Bedeutungen der einzelnen Angaben berücksichtigt werden. Steuerangaben für Ressourcen betreffen im einzelnen:

1. Operationen
2. Argumente
3. Resultate
4. die nachfolgende Steuerangabe
5. das eigene Format.

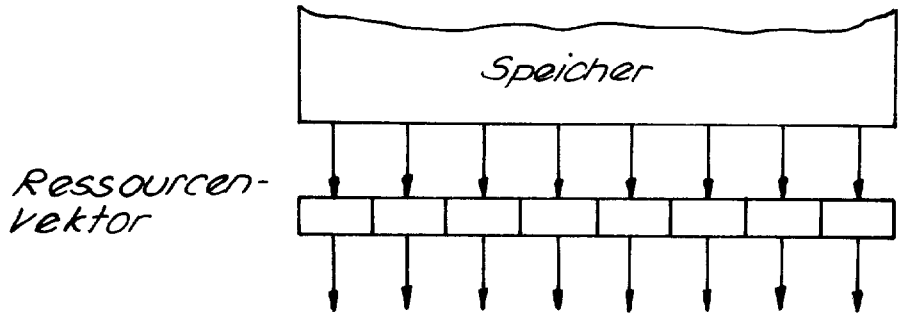
Operationsangaben sind letzten Endes (ihrer Tiefenstruktur nach, gleichgültig wie die Codierung aussieht) Ordinalzahlen in die Menge aller vergegenständlichten Operations- Verknüpfungen, die die im aktuellen Zeitschritt auszuführenden Operationen auswählen (übliche Masschinenbefehle haben nur einen binär codierten Operationscode, VLIW- Befehle haben mehrere Operationsangaben, bei Mikrobefehlen sind diese Angaben in Bitgruppen oder Einzelbits aufgelöst).

Für Argumentangaben gibt es die Alternativen

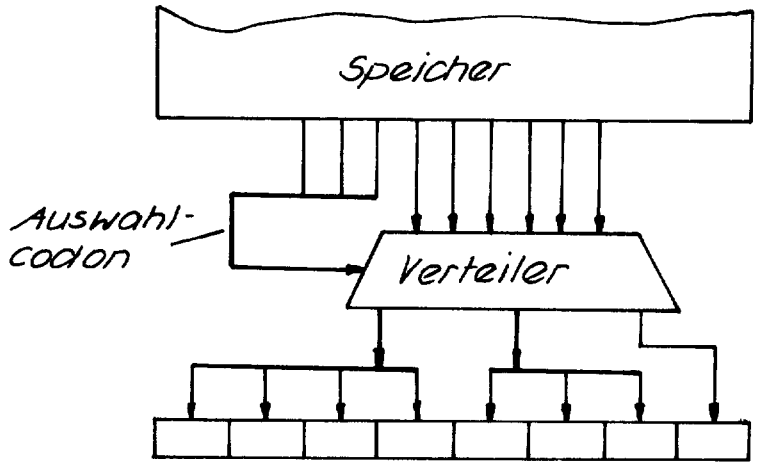
- Direktwerte
- Selektorangaben (Adressen).

Selektorangaben können auf Direktwerte beschränkt sein (Absolutadressen); zumeist handelt es sich aber um die Auswahl aus einer Menge vergegenständlichter Selektor- Abstraktionen (Adressierungsverfahren), die durch Direktwerte (Absolutadresse des Adressenregisters, Adressen- Offset) erweitert ist.
Für Resultate sind ebensolche Selektorangaben vorgesehen, es sei denn, das betreffende Resultat wird in vergegenständlichten Speichermitteln abgelegt (Akkumulatorregister, Top of Stack).

a) direkte Zuordnung



b) gruppenweise Auswahl



c) codierte Auswahl

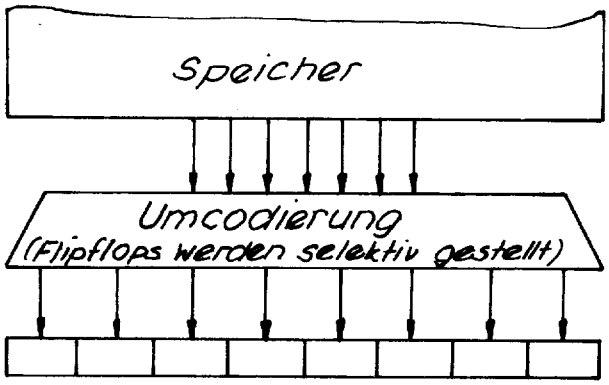


Bild 42 Transport von Information zu den Ressourcen

Die Nachfolge der aktuellen Steuerangabe ist oft durch +1-Zählung (Befehlszähler) vergegenständlicht. Abweichungen von dieser Reihenfolge erfordern codierte Angaben, das heißt Ordinalzahlen für die Auswahl der jeweiligen Aktivator- Vergegenständlichung (Art der Verzweigung, Bedingungsauswahl) sowie Selektorangaben für die nachfolgende Steuerinformation (Befehlsadresse).
Wenn die Zuordnung aller Steuerangaben zu den einzelnen Bitpositionen des Ressourcenvektors nicht fest vergegenständlicht ist, sind weitere Angaben notwendig, die deren Formatierung beschreiben.
Die direkte Zuordnung ist bei "horizontalen" Mikrobefehlen üblich und hat auch in Maschinen mit überlangem Befehlswort (VLIW) Anwendung gefunden. Nachteilig ist eine oft beachtliche Redundanz, da alle Formate für die jeweils längste Angabe eingerichtet sein müssen und es nur selten gelingt, alle Ressourcen wirklich parallel auszunutzen. Ein gewisser Ausweg besteht darin, die Angaben komprimiert zu speichern und Erweiterungs-, Auswahl- bzw. Umcodierungseinrichtungen im Datenweg zum Ressourcenvektor vorzusehen (vgl. Bild 42). Dafür seien einige Beispiele genannt:

1. Der "Transputer" T 8ØØ (/162/). Die Befehle sind nur 8 bit lang (4 Operationscode- und 4 Datenbits). Längere Steuerangaben werden in einem 32-bit- Operandenregister nacheinander mit PREFIX- Befehlen aufgebaut (diese übertragen nach entsprechender Verschiebung des Registerinhalts ihre 4 Datenbits in das Operandenregister). Die Wirkung wird mit einem Befehl OPERATE ausgelöst.

2. Der experimentelle CRISP- Mikroprozessor (/158/). Die Befehle desses 32-bit- Prozessors sind 1Ø, 6 oder 2 Bytes lang. Jeder Befehl wird in einen horizontal codierten Mikrobefehl von 192 bit umgeschlüsselt, der alle Datenwege und Verknüpfungsschaltungen direkt steuert. Dabei wird auch der nachfolgende Befehl mit berücksichtigt: ist dieser ein Sprungbefehl, wird er ebenfalls gelesen und im Mikrobefehl verschlüsselt. So sind intern Aktionen und Verzweigungen eine Einheit.

3. Der "Trace"- Computer (/147/). Diese VLIW- Maschine hat eine Befehlslänge von 1Ø24 bit. Es werden aber nur die Abschnitte jedes Befehls gespeichert, die tatsächlich belegt sind.

Eine weitere Lösung besteht darin, alle Angaben nur mit so vielen Bits zu codieren wie unbedingt erforderlich und die Art und Weise der Codierung in zusätzlichen Bitpositionen anzugeben. Das führt dazu, Befehle als Bitketten veränderlicher Länge zu gestalten (das Schema entspricht Bild 42 c).[1]

---

1 Das ist beim iAPX 432 verwirklicht worden; in /146/ wurde aber gezeigt, daß sich der Aufwand nicht lohnt. Des weiteren zeigen die Untersuchungen zur bitvariablen Codierung von Adressenangaben in Befehlen, daß mit 2 festen Längen (6 und 12 bit) ein befriedigender Kompromiß gegeben ist (vgl. S. 1Ø9, Fußnote 3 und Tafel 19, S. 1Ø8).

Allgemein üblich ist es, die Angaben auf mehrere Befehle zu verteilen. Wegen der sequentiellen Arbeitsweise ist die parallele Nutzbarkeit der Ressourcen stark eingeschränkt, und es sind kennzeichnende Codons ("Opcodes") erforderlich, da die einzelnen Angaben nicht mehr durch ihre Position im Binärvektor implizit gekennzeichnet sind. Deshalb wird die direkte Aneinanderreihung aller Codons bevorzugt. Zur unmittelbaren Steuerung der Ressourcen (in denen die Abstraktionen gemäß Abschnitt 6 vergegenständlicht sind) sind diese Angaben fest formatiert, und es ist keine Datenkompression vorgesehen. Die Formatgestaltung ähnelt der von üblichen Mikroprogrammsteuerungen. Die Information wird nach folgendem Schema genutzt:

1. die Steuerangaben werden in den Ressourcenvektor geladen
2. die Argumente werden herangeschafft
3. die Resultate werden ermittelt
4. die Resultate werden zugewiesen (erforderlichenfalls abtransportiert)
5. die folgenden Steuerangaben werden herangeschafft.

Im allgemeinen Fall lassen sich nicht alle Schritte 1...4 parallel ausführen. Der Zeitbedarf für das Ermitteln der Resultate wird durch die Ressourcen (strukturelle Gestaltung und Aufwand) bestimmt. Diese einmal gegeben, ist eine Ablaufbeschleunigung nur an folgenden Stellen möglich:

- Heranschaffen der Argumente: durch zeitliche Überlappung (voreilende Adressenrechnung, "look ahead"), durch Rückgriff auf Werte, die bereits in Teilen des Ressourcevektors (Registern) hinterlegt sind oder dadurch, daß sie zusammen mit den Operator- Codons gelesen werden (Datenflußprinzip).

- Zuweisung der Resultate: Vermeiden bzw. Hinausschieben der Zuweisung durch Hinterlegen der Resultate in Teilen des Ressourcenvektors

- Heranschaffen der folgenden Steuerangaben: durch Überlappung mit der Resultatbildung (voreilendes Befehlslesen).

Folglich sind die Prinzipien der Ablaufüberlappung ("look-ahead") und der Datenflußsteuerung[1] im einzelnen zu untersuchen. Besondere Maschinenzyklen für das Heranschaffen von Argumenten und das Zuweisen von Resultaten sind soweit wie möglich zu vermeiden. Unter diesem Gesichtspunkt sind Architekturen mit Top-of-Stack- bzw. Akkumulatorregistern eine bedenkenswerte Alternative[2] zu den gängigen Universalregisterkonzepten: die erstgenannten Register sind ohne weiters als Teil des Ressourcenvektors ausführbar, d. h. als Flipflops, die direkt mit den Verknüpfungsschaltungen verbunden sind; hingegen lassen sich große Universalregistersätze meist nur als (deutlich langsamere) RAM- Arrays verwirklichen.

---

1 Das betrifft elementare Konzepte, die sich technisch verwirklichen lassen (z. B. gem. /76/, /284, /306/).

2 Z. B. T 800 mit Top-of-Stack- Registern (3 * 32 bit für Integer- und 3 * 64 bit für Gleitkommaverarbeitung).

7.3. Schaltungsstrukturen eines Hochleistungsrechners

Ein Versuch, neue Prinzipien der Rechnerarchitektur einzuführen, wird eher Aussicht auf Erfolg haben, wenn sie nicht nur in einer einzigen technischen Ausführungsform nutzbar sind. Um den Grundsatz "ein Architekturkonzept, vielfältige Ausführungsformen" zu verwirklichen, gibt es mehrere Möglichkeiten. Wegen der technisch- ökonomischen Bedeutung dieses Problems werden vor den Erläuterungen zum Hochleistungsrechner einige Grundsatzlösungen dargelegt:

1. Aufwärts- bzw. Abwärtskompatibilität. In einer Reihe aufwands- und leistungsmäßig abgestufter Hardwarekomplexe sind in den kleineren Modellen Teilmengen des gesamten Funktionsumfangs vorgesehen. Anwendungslösungen kleinerer Modelle sind auf größeren unmittelbar lauffähig (Aufwärtskompatibilität). Wenn überhaupt vorgesehen ist, daß auf größeren Modellen implementierte Anwendungslösungen auf kleinere übertragbar sind (Abwärtskompatibilität), so ist dies meist durch softwaremäßige Emulation (Trap- Routinen) verwirklicht. Das Konzept der Aufwärtskompatibilität ist aus technischen Beschränkungen[1] und wirtschaftlichen Zwängen[2] heraus in den erfolgreichen Mikroprozessorfamilien verwirklicht worden (Intel 8086...486, Motorola 68 000...68 040, National NS 32000). Auf- und Abwärtskompatibilität (letztere durch softwareseitige Emulation) wurde in verschiedenen Serien von EDV- Anlagen vorgesehen (z. B. CDC 3100...3500).

2. Kompatible Systemfamilie. Alle Modellen haben denselben Funktionsumfang, der auf jeweils angemessene Weise implementiert ist. Dafür wird - namentlich in den kleineren Modellen - die Mikroprogrammierung umfassend genutzt (S/360, S/370, PDP 11, VAX).

3. Zwischensprache. Als Schnittstelle zu den Compilern und Anwendungsprogrammen wird eine hardware- unabhängige Zwischensprache definiert. Alle Anwendungsprogramme werden in diese Zwischensprache compiliert. In den einzelnen Modellen ist die Zwischensprache auf jeweils angemessene Weise implementiert (IBM S/38, AS/400), bzw. es ist eine modellspezifische Compilierung in den Maschinencode vorgesehen (Convex).

4. Verschiedene Technologien. In allen Modellen ist derselbe Funktionsumfang vorgesehen, und alle Modelle haben dieselbe Struktur, die mit verschiedenen technologischen Mitteln implementiert wird (z. B. in CMOS und in ECL). Beispiele: IBM 709/7090, Burroughs 3500, SPARC[3].

---

1 Die architekturseitige Auslegung der ersten Mikroprozessoren wurde wesentlich durch den Stand der Schaltkreistechnologie (Integrationsgrad) beschränkt.

2 Versuche, neue technologische Möglichkeiten ohne Rücksicht auf Bestehendes für höher entwickelte Architekturkonzepte zu nutzen, sind oft vom Markt nicht akzeptiert worden.

3 Dieser RISC- Mikroprozessor ist ausdrücklich dafür ausgelegt worden (Scalable Processor Architecture).

5. Konzeptionelle Verkleinerung. Die Architekturkonzepte der größeren Modelle werden gleichsam maßstäblich verkleinert auf die kleineren Modelle übertragen (so könnte beispielsweise statt einer 64-bit- Adresse eine 32-bit- Adresse vorgesehen werden, statt 64 Registern nur 16 usw.). Es ist aber gewährleistet, daß gleiche Operationen mit gleichen Datenstrukturen stets gleiche Resultate erbringen. Wichtig ist hier die konzeptionelle Einheitlichkeit und nicht eine umfassende Kompatibilität bis aufs Bit. Die Datenstrukturen müssen kompatibel sein, Steuerstrukturen, wie Adressen, Operationscodes usw. aber nicht. Datenbestände lassen sich zwischen den einzelnen Modellen austauschen, aber Programme bedürfen der Neucompilierung. Als Beispiel ist das Modell 360/20 in seiner Stellung zu den anderen Modellen des S/360 anzusehen (weiterhin Siemens 4004/15 und /25 gegenüber den größeren Modellen dieser Serie); aber auch der Mikroprozessor Intel 8086 gegenüber 80286 im Protected Mode.

Diese Konzepte sind in realen Systemen nur selten in reiner Form zu finden.[1] Auch für den eigenen Ansatz wird eine ingenieurmäßige Kompromißlösung auszuarbeiten sein. Als besonders zweckmäßig wird eine Kombination der Konzepte "Zwischensprache" und "konzeptionelle Verkleinerung" angesehen, und zwar aus folgenden Gründen:

- Das Hauptziel ist nicht ein neuer Mikoprozessor, sondern ein Hochleistungsrechner, der das Leistungsniveau von Mikroprozessoren deutlich übertrifft. Kleinere Hardwarekomplexe, die neuen Architekturprinzipien weitere Verbreitung sichern sollen, werden ebenfalls nicht als direkte Gegenstücke zu den eingeführten Mikroprozessorfamilien gesehen[2], sondern als Alternativen auf dem Gebiet der Mikrocontroller, der industriellen Steuerungen bzw. allgemein der "embedded systems". Für diesen Bereich muß die verfügbare Siliziumfläche[3] so gut wie möglich genutzt werden. Das ist aber nur durch direkte Vergegenständlichung der Architekturkonzepte zu gewährleisten und nicht durch mikroprogrammtechnische Emulation.[4]

- Die Architektur (gleich welcher Auslegung im einzelnen) wird nicht in einer "Assemblersprache" programmierbar sein. Deshalb ist davon auszugehen, daß es für alle Programme Quelltexte in höheren Programmiersprachen gibt.

---

1 So beim S/360: verschiedene Logikbaureihen und Schaltungsstrukturen, Speicher verschiedener Zykluszeit, Mikroprogrammsteuerung.

2 Weltweit sind Millionen einschlägiger Systeme verbreitet, und das Leistungsvermögen der eingeführten Mikroprozessoren ist bereits sehr hoch (80386, 68 030).

3 "Siliziumfläche" steht hier für logisch- funktionelle Möglichkeiten, Fertigungskosten, Ausbeute usw.

4 Eine komplexe Architektur braucht auf kleinen Hardwarestrukturen lange Mikroprogramme (ein /360- Befehl 20...50 (/154/)). Es hat sich gezeigt, daß kleinere /360- Modelle viel effektiver arbeiten, wenn Anwendungslösungen direkt in den Mikrocode umgesetzt werden (/4/).

• "Große" Anwendungen, die Hochleistungsrechner im Stundenbereich belegen (z. B. Simulationssysteme), werden nie auf die kleinen Modelle übertragen.

• Kleinere Anwendungsaufgaben brauchen keineswegs alle technischen Vorkehrungen des Hochleistungsrechners. So ist eine 64-bit- Adresse bei weitem nicht ausgenutzt, wenn nur 256 kBytes Speicher installiert sind und ausschließlich ein fest gegebener Komplex von Echtzeitprogrammen abzuarbeiten ist.

• Der Aufwand für die mikroprogrammtechnische Emulation der vollständigen Architekturdefinition, der sich auf jedem einzelnen Schaltkreis widerspiegelt[1], wird ersetzt durch die einmaligen Aufwendungen für die Compilerversionen (von der Zwischensprache in den Maschinencode) und für die Neucompilierung (Zwischensprachniveau -> Maschinencode) jener Programme, die auf andere Modelle zu übertragen sind.

7.3.1. Speicherstrukturen

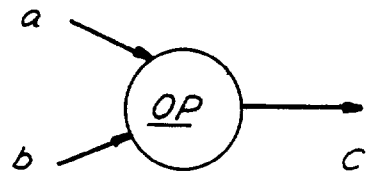
Die technischen Speichermittel sollten die in Abschnitt 6.4. (S. 104 ff.) erläuterte Speicherhierarchie möglichst direkt widerspiegeln. Für die leistunsgsfähige Vergegenständlichung läßt sich die Tatsache ausnutzen, daß im eigentliche Sinne wahlfreie Zugriffe mit Adressen, die von Zyklus zu Zyklus beliebig wechseln, nur in den untersten Hierarchie- Ebenen (Register, Steuerspeicher, Operationsspeicher) vorkommen. Ansonsten haben die meisten Speicherzugriffe der Charakter von Blocktransporten. Dafür sollen weitgehend autonome Steuer- und Adressierungsschaltungen vorgesehen werden, die möglichst unmittelbar mit den eigentlichen Speicherschaltkreisen zusammengeschaltet werden, um deren Eigenschaften bestmöglich nutzen zu können (Adressen- und Datenweglogik in Bild 27).

7.3.2. Operationswerke

Hier geht es grundsätzlich nur um Rechner, die zu einem Zeitpunkt ein Programm abarbeiten. Trotzdem ist es sinnvoll, über die Anordnung mehrerer Operationswerke nachzudenken, um den innewohnenden Parallelismus solcher Abläufe ausnutzen zu können. Die Anzahl der Werke soll auf ein vernünftiges Mindestmaß beschränkt bleiben (Ausnutzungsgrad!).
Für die numerische Informationsverarbeitung zeigt Bild 43 bekannte Anordnungen von Operationswerken (jedes Werk ist für alle Grundrechenarten eingerichtet).[2] Um zu bestimmen, wieviele Werke sinnvollerweise vorgesehen werden sollten, kann man eine Vielzahl einschlägiger Programme einer Datenflußanalyse unterziehen. Dazu werden die Quelltexte sowie entsprechende Compiler benötigt.
Hier wird ein anderer Ansatz versucht: es werden die Daten-

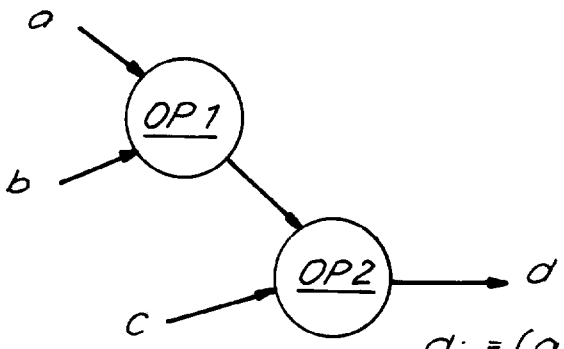
1 Betrifft Siliziumfläche für den Mikroprogramm- ROM und Laufzeitverlängerung.
2 Erläuterungen zu Bild 43 enthält Tafel 21.

1.



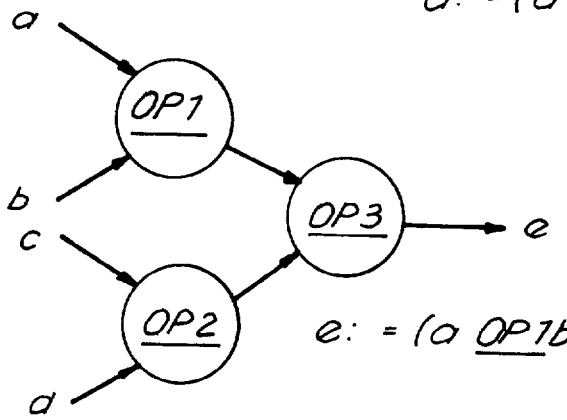
c: = a OP b

2.



d: = (a OP1 b) OP2 c

3.



e: = (a OP1 b) OP3 (c OP2 d)

4.

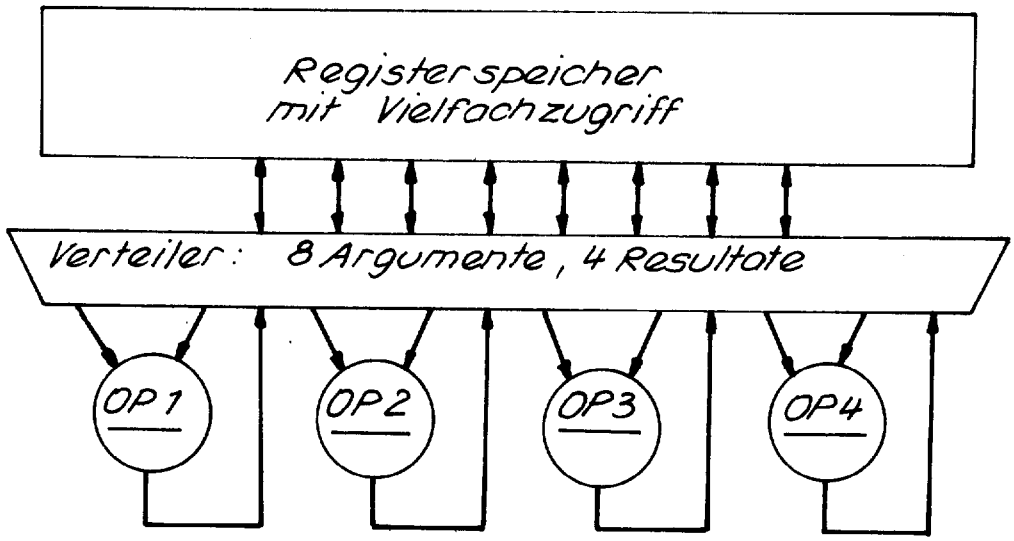


Bild 43 Verschiedene Anordnungen von Operationswerken

1. c := a OP b - repräsentiert den allgemeinen Stand der Technik.

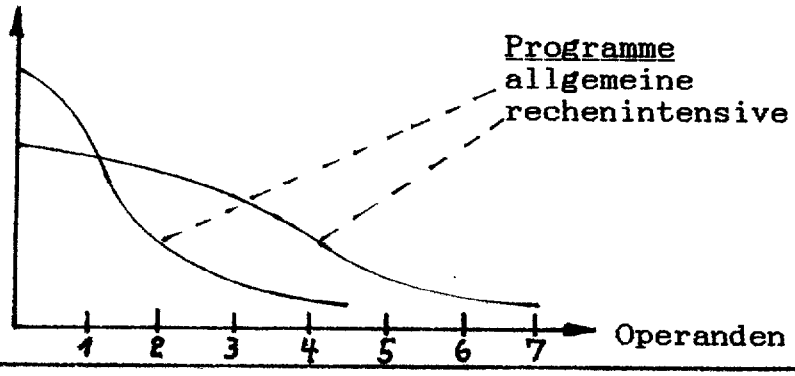
2. d := (a OP1 b) OP2 c - Vorschlag in /333/.

Beispiel einer Verknüpfung: die 5. "Livermoore loop":

```
      DO 5  i = 2,N
         X(i) := Z(i) * (Y(i) - X(i-1))
      5 CONTINUE
```

3. e := (a OP1 b) OP3 (c OP2 d)

Für einen GaAs- Mikroprozessor vorgesehen (/3Ø5/). Die Struktur wurde auf Grund folgender Feststellung gewählt: Anwendungsprogramme lassen sich in der Praxis sehr oft in allgemeine und besonders rechenintensive einteilen. Erstere haben in 93% der Zuordnungen keinen oder nur einen arithmetischen Operanden, letztere in 93% der Zuordnungen bis zu 3 Operanden. Dafür wird folgende näherungsweise Häufigkeitsverteilung angegeben:



4. 4 Werke + Universalregister: in VLIW- Architekturen vorgesehen (s. etwa /147/)

(Für Integer- und Gleitkommaoperationen ist je eine solche Struktur angeordet.)

Tafel 21

Erläuterungen zu Bild 43

flußschemata grundlegender Operationen des numerischen Rechnens mit Intervallen, komplexen Zahlen, rationalen Zahlen sowie Reihenentwicklungen betrachtet. Solche Schemata lassen sich ohne weiteres unter Rückgriff auf übliche Formelsammlungen (z. B. /44/, /47/) ausarbeiten. Das ist in den Bildern 44 - 49 anhand wichtiger Beispiele veranschaulicht.
In Tafel 22 ist zusammengefaßt, wieviele Rechenwerke für welche Operation benötigt werden, welche Zusatzeinrichtungen erforderlich sind und wie in manchen Fällen die Zahl der Werke sinnvoll durch Ausführung in mehreren Schritten verringert werden kann. Es ist ersichtlich, daß 4 Werke in den weitaus meisten Fällen voll ausgelastet werden können. 4 Rechenwerke (jedes für alle 4 Grundrechenarten ausgelegt, d. h. für c:=a+b, c:=a-b, c:=b-a, c:=a*b, c:=a/b, c:=b/a, c:=a, c:=b), 2 Zwischenspeicher, eine Minimum/Maximum- Auswahl und eine Einrichtung zur Delta- Erkennung sind also zweckmäßigerweise vorzusehen. Die Absicht besteht darin, stets das niedrigste Niveau ("finest grain") der Parallelisierung zu nutzen: wenn beispielsweise Vektoren von komplexen Zahlen zu verknüpfen sind, wird mit den 4 Werken zusammen jeweils eine komplexe Operation ausgeführt; die Werke werden nicht genutzt, um 4 unabhängige Resultate von 4 Element- Paaren der beiden Argumentvektoren zu berechnen. Vorteile:

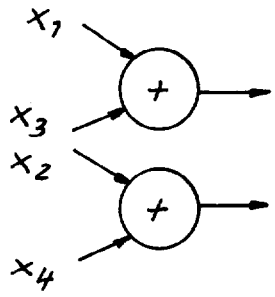
- die Anforderungen an die Speicherorganisation werden deutlich vermindert
- die Werke werden besser genutzt: es werden stets alle 4 belegt, unabhängig davon, wie lang die Vektoren sind
- Vereinfachung der Ablaufsteuerung: für bestimmte Abläufe, wie das Rechnen mit komplexen Zahlen, braucht man nur eine einzige Ablaufsteuerung und nicht 4 unabhängige.

In der nichtnumerischen Informationsverarbeitung sind Transporte einschließlich solcher mit Umcodierung und assoziative Suchoperationen parallelisierbar; letztere werden hier auf Durchmusterungen nach Orthogonalität bzw. Nichtorthogonalität (bei verschiedenen Kombinationen von Datentypen) zurückgeführt. Es bietet sich somit an, den Speichern im Rahmen der Adressen- und Datenweglogik solche Schaltmittel direkt zuzuordnen, zumal diese recht einfach sind (vgl. /24Ø/). Für andere Aufgaben gibt es derzeit keine Erkenntnisse, die eine Parallelanordnung mehrerer Werke rechtfertigen, so daß ein einziges, in sich leistungsfähig und universell ausgestaltetes Werk als ausreichend erscheint.

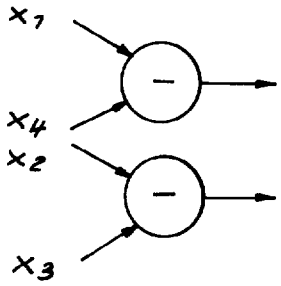
7.3.3. Selektions- und Iterationswerke

Die meisten vergegenständlichten Operationen erzeugen aus 2 Argumenten 1 Resultat. Weiterhin ist für das Heranschaffen des nächsten Befehls bzw. Steuerwortes zu sorgen . Somit sind mindestens 4 unabhängige parallele Zugriffspfade zu den Speichern der Laufzeitumgebung (bzw. zu Befehls- und Operandenspeichern) vorzusehen. Verschiedene Selektor- bzw. Iteratorfunktionen sind für Steuerzwecke notwendig (Unterprogrammaufruf, Parameterübergabe, Unterbrechungsbehandlung; das betrifft

Addition: $[x_1, x_2] + [x_3, x_4] = [x_1 + x_3, x_2 + x_4]$

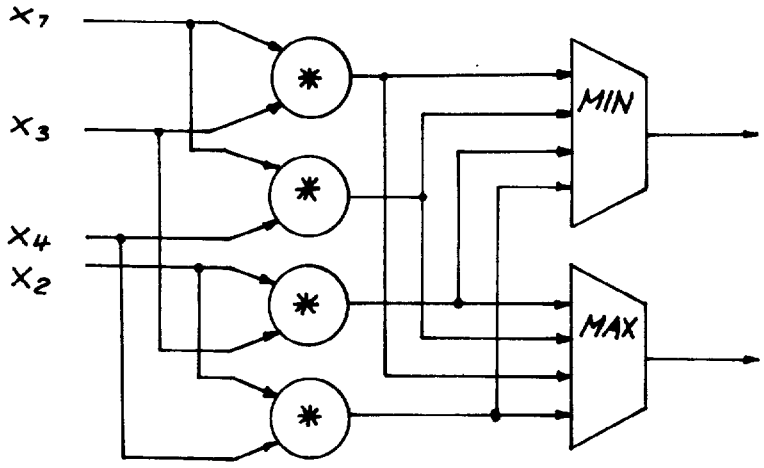


Subtraktion: $[x_1, x_2] - [x_3, x_4] = [x_1 - x_4, x_2 - x_3]$



Multiplikation: $[x_1, x_2] * [x_3, x_4] =$

$[\min(x_1x_3, x_1x_4, x_2x_3, x_2x_4), \max(x_1x_3, x_1x_4, x_2x_3, x_2x_4)]$



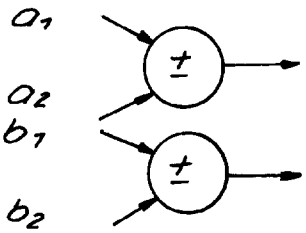
Division: $[x_1, x_2] / [x_3, x_4] = [x_1, x_2] * [1/x_4, 1/x_3]$

zurückführbar auf Multiplikation; mit 4 Werken in 2 Schritten ausführbar

Bild 44 Intervallrechnung

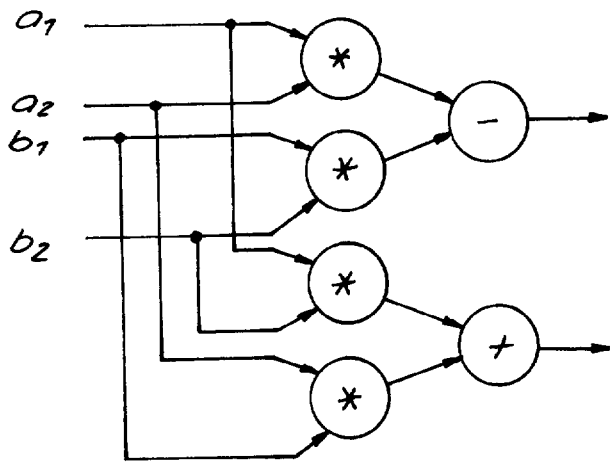
Addition / Subtraktion:

$(a_1, b_1) \pm (a_2, b_2) = (a_1 \pm a_2, b_1 \pm b_2)$



Multiplikation:

$(a_1, b_1) * (a_2, b_2) = (a_1 a_2 - b_1 b_2, a_1 b_2 + a_2 b_1)$

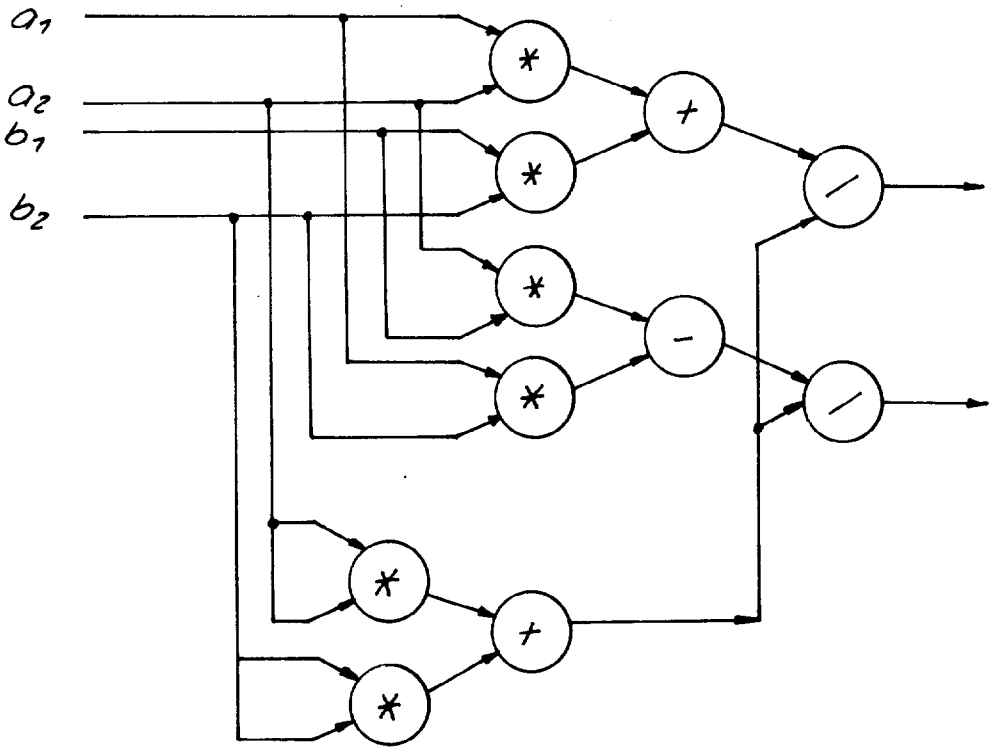


(In 2 Zyklen ausführbar mit einer Struktur aus 3 Werken)

Bild 45 Rechnen mit komplexen Zahlen (Addition, Subtraktion, Multiplikation)

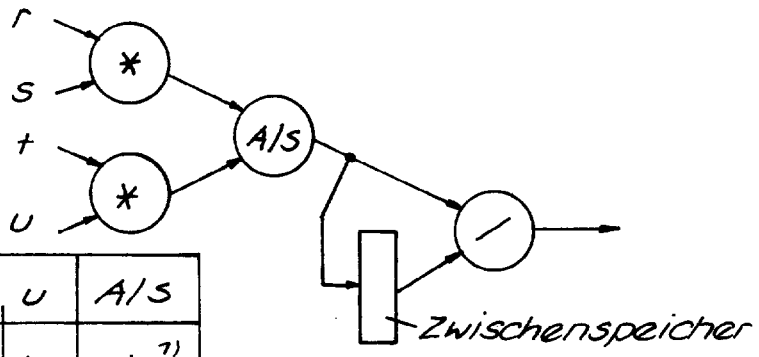
Division:

$$(a_1, b_1) / (a_2, b_2) = \left\{ \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} \; ; \; \frac{a_2 b_1 - a_1 b_2}{a_2^2 + b_2^2} \right\}$$



Überführung in Struktur aus 4 Werken

(erfordert 3 Ausführungszyklen):



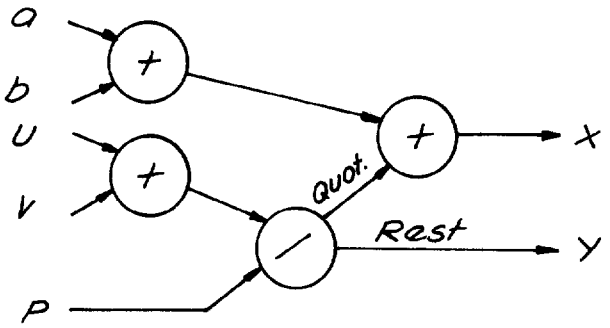
Zyklus	r	s	t	u	A/S
1	a_2	a_2	b_2	b_2	+ 1)
2	a_1	a_2	b_1	b_2	+
3	a_2	b_1	a_1	b_2	-

1) Resultat zum Zwischenspeicher

Bild 46 Division komplexer Zahlen

Addition rationaler Zahlen gleicher Genauigkeit:

$$x + \frac{y}{p} = a + \frac{u}{p} + b + \frac{v}{p}$$



Addition rationaler Zahlen unterschiedl. Genauigkeit:

$$x + \frac{y}{p} = a + \frac{u}{p_1} + b + \frac{v}{p_2}$$

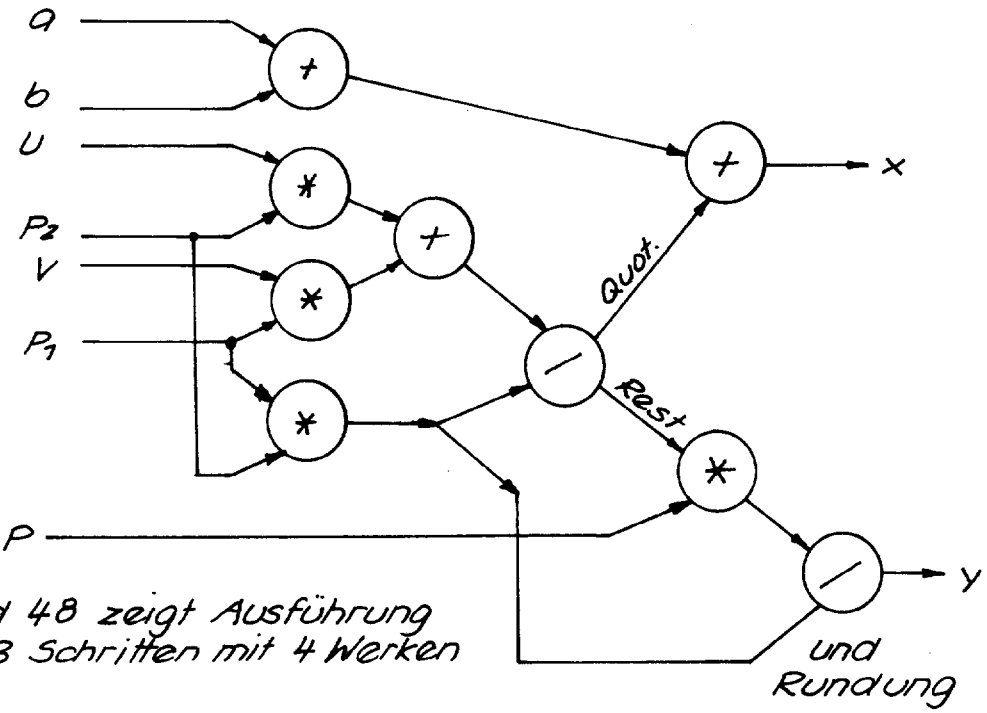
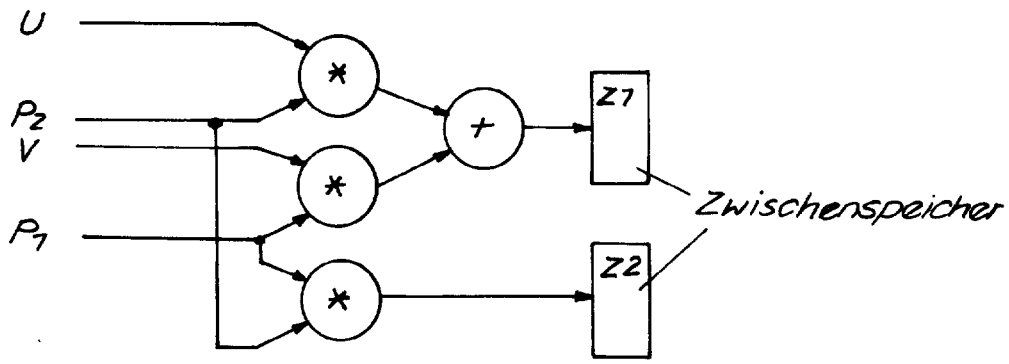
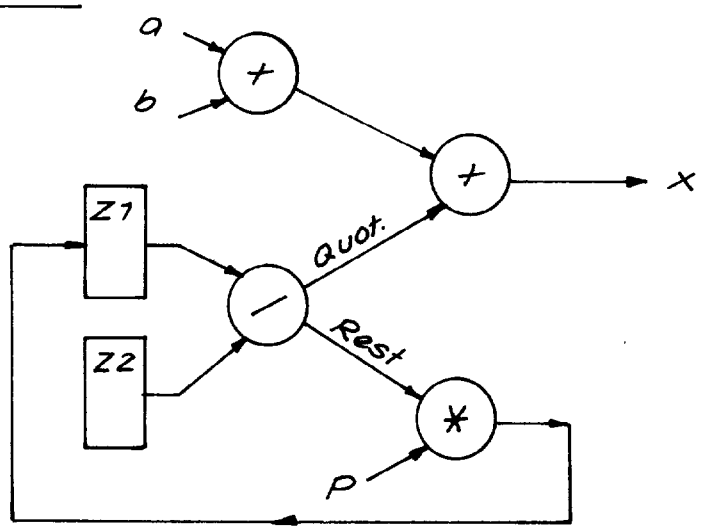


Bild 47 Rechnen mit rationalen Zahlen (Beispiele)

1. Schritt



2. Schritt



3. Schritt

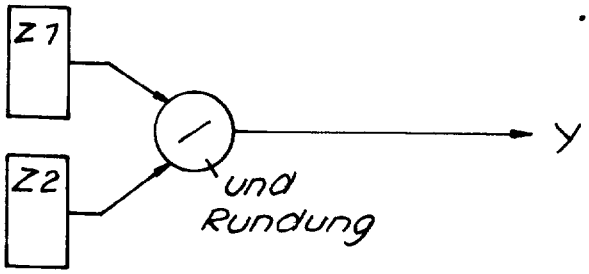


Bild 48 Allgemeine Addition rationaler Zahlen (s. Bild 47) in 3 Schritten

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} \ldots + \frac{x_4}{n!} \ldots$$

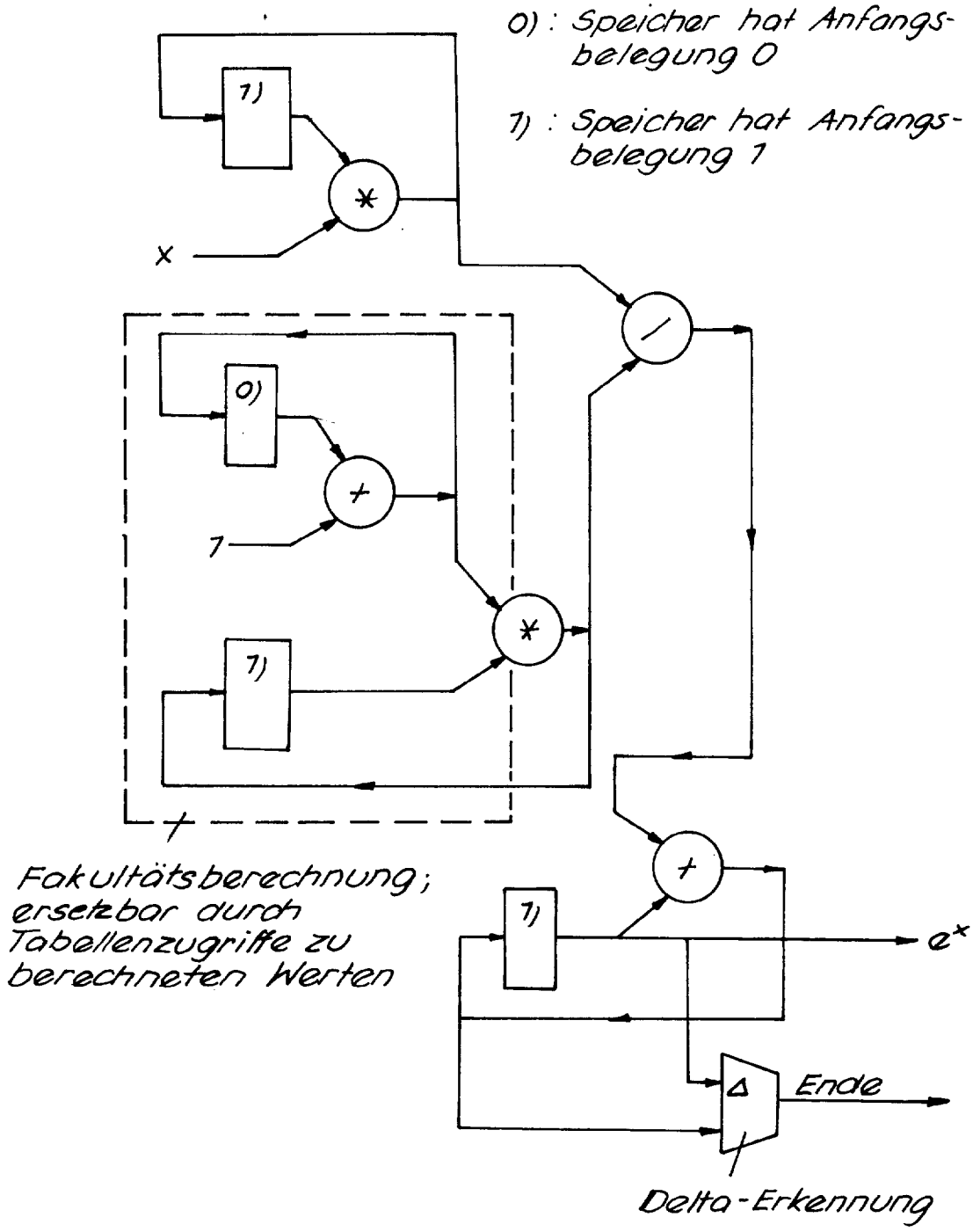


Bild 49 Berechnung einer Reihenentwicklung

Operation	Rechenwerke	Zusatz-einrichtungen	Aufwands-verringerung
Intervalle:			
Addition	2 ADD		
Subtraktion	2 SUB		
Multipli-kation	4 MUL	1 Minimumerk. 1 Maximumerk.	
Division	2 DIV (1/n) 4 MUL	1 Minimumerk. 1 Maximumerk.	2 Zyklen, 2 Zwischen-speicher f. $1/x_3$, $1/x_4$
komplexe Zahlen:			
Addition	2 ADD		
Subtraktion	2 SUB		
Multipli-kation	4 MUL 1 ADD 1 SUB		2 Zyklen, 3 Werke
Division	6 MUL 2 ADD 1 SUB 2 DIV		3 Zyklen, 4 Werke, 1 Zwischen-speicher
Addition rationaler Zahlen:			
gleiche Genauigkeit	3 ADD 1 DIV		
unterschied-liche Genauigkeit	4 MUL 3 ADD 2 DIV		3 Zyklen, 4 Werke, 2 Zwischen-speicher
Reihen-entwicklung (Bsp: e^x)	2 MUL 2 ADD 1 DIV	1 Delta-erkennung	4 Werke, wenn n! aus Tabelle

Tafel 22

Aufwendungen für die Vergegenständlichung von Datenflußschemata wichtiger numerischer Operationen

im wesentlichen Zugriffstabellen, Stacks und Warteschlangen). Hier wird vorgeschlagen, wenigstens 16 kombinierte Selektor- und Iterator- Vergegenständlichungen anzuordnen, wovon 4 gleichzeitig die genannten Zugriffspfade nutzen können.[1] Sie sind in Gruppen mit verschiedenen funktionellen Möglichkeiten eingeteilt (Schaltungs- Anregungen s. Bild 29...36). Eine solche Anordnung kann wesentlich mehr leisten als eine Maschine mit 16 Adressenregistern, da es für jede Selektions- bzw. Iterationsangabe einen unabhängigen Registersatz und für die Zugriffspade parallelwirkende Verknüpfungsschaltungen (z. B. Addierwerke und Vergleicher) gibt.[2]

7.3.4. Ausnutzung der Schaltmittel

Ein Universalrechner, der mehrere spezialisierte Funktionseinheiten enthält, ist wesentlich leistungsfähiger als ein solcher, der alle Funktionen durch Mehrfachausnutzung universeller Schaltmittel realisiert, der Ausnutzunsgrad der einzelnen Schaltungsanordnung ist aber deutlich geringer. Hier hängt er im wesentlichen von der Mischung zwischen numerischer und nichtnumerischer Informationsverarbeitung ab. Zur Verbesserung des Ausnutzungsgrades gibt es folgende Möglichkeiten:

- zeitmultiplexe Ausnutzung der Hardware durch mehrere unabhängige Aufgaben[3]
- Aufwandsverringerung in den einzelnen Schaltungskomplexen in Abhängigkeit von Erwartungswerten der Nutzungshäufigkeit.

7.3.5. Übersicht

Bild 5Ø vermittelt einen Eindruck von der Struktur eines Hochleistungsrechners[4], der nach den hier diskutierten Prinzipien aufgebaut ist. Aus Aufwandsgründen sind technisch nur 3 Speicherebenen vorgesehen:

- der kombinierte Datenbasis- und Laufzeitspeicher
- die Operanden- und Steuerspeicher
- die Register.

Datenbasis- und Laufzeitspeicher sind nur technisch vereinigt, sie sind aber logisch in 2 unabhängige Bereiche getrennt. Zur Verwaltung der Datenbasis, zur Steuerung der Ein- und Ausgabe und für Zwecke der Initialisierung, Diagnose und Fehlerbehandlung sind getrennte programmierbare Einrichtungen vorgesehen (dazu kann beispielsweise auf eingeführte 32-bit- Mikroprozessoren zurückgegriffen werden).

---

1 Jede Vergegenständlichung kann jeden Zugriffspfad belegen (z. B. über "crossbar"- Netzwerke).
2 Man braucht nur soviel parallele Verarbeitungshardware, wie Zugriffspade vorgesehen sind (als Anregung s. /24Ø/).
3 Vorbilder: CDC 66ØØ- Peripherieprozessoren, HEP, Stellar.
4 Bild 5Ø zeigt auch den Anschluß von Hardware zur Interpretation von Fremdarchitekturen (s. dazu S. 168).

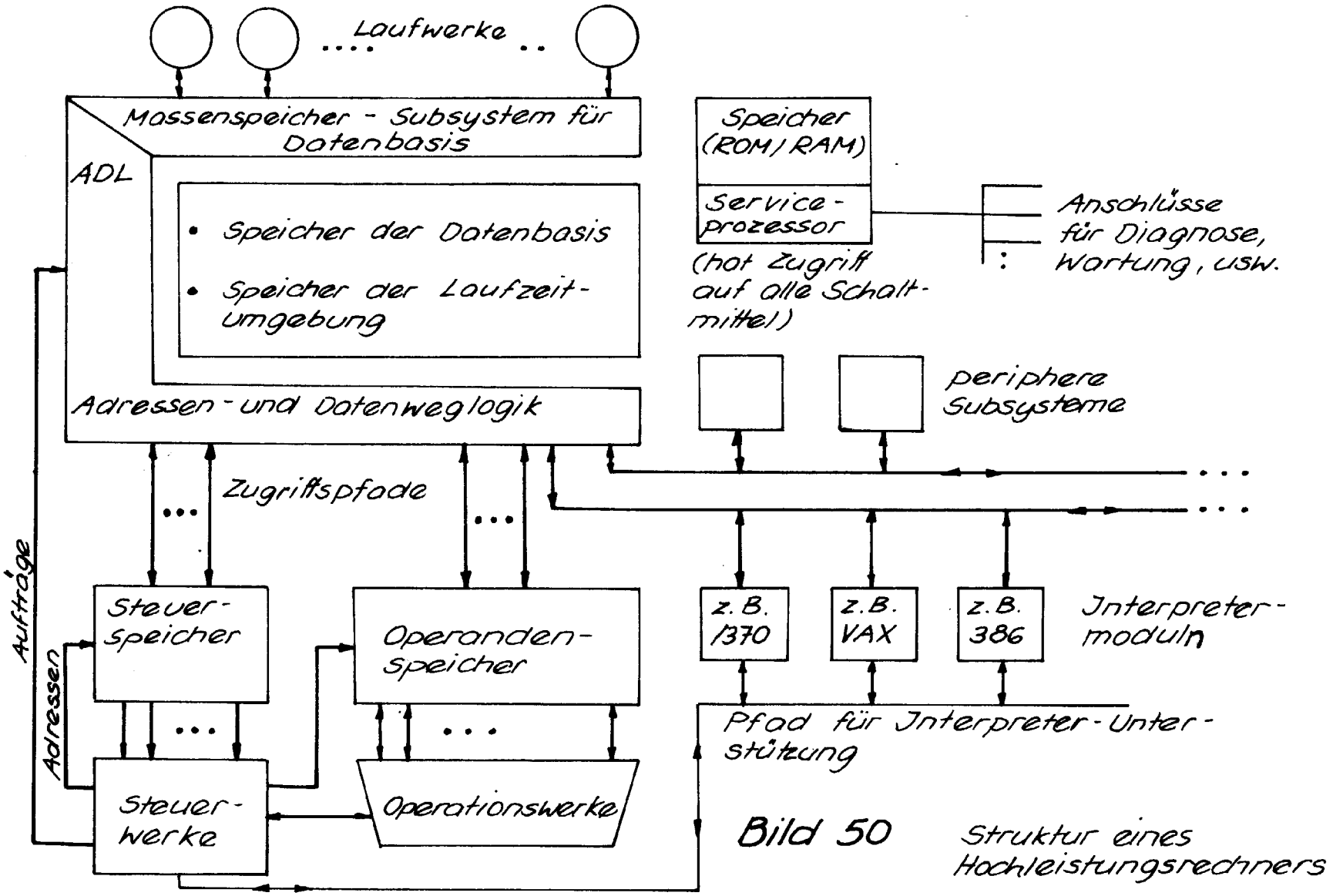


Bild 50 Struktur eines Hochleistungsrechners

7.4. Leistungsbetrachtungen

7.4.1. Absolute Grenzen

Im Rahmen überschlägiger Leistungsbetrachtungen kann man davon ausgehen, daß jede vergegenständlichte Aktion in einem Maschinenzyklus ausgeführt wird. Gemäß Abschnitt 4.1. werden Abläufe zur Selektion, Iteration und Aktivierung nicht mitgezählt, so daß die Maximalleistung praktisch von der Anzahl der Operationswerke abhängt. Bei k Operationswerken (je 2 Argumente, 1 Resultat), einer Eimergröße für Argumente und Resultate von b Bits und einer Zykluszeit t_c[1] ergibt sich somit die Maximalleistung zu

$$PMO = \frac{k}{t_c} \quad [Op/s] \quad \text{bzw.} \quad PM = \frac{3bk}{t_c} \quad [EB/s] \qquad (7.2)$$

Tafel 23 veranschaulicht einige Leistungsschätzungen bei verschiedenen technologischen Auslegungen und ermöglicht einen Vergleich zu bereits existierenden Prozessoren. Solche Angaben kommen den Verhältnissen der Praxis um so näher, je mehr die Anwendungsaufgaben durch Vektorverarbeitung oder nutzbaren internen Parallelismus gekennzeichnet sind. Bei weitgehend sequentiellen Abläufen sind Leistungsvorteile gegenüber den eingeführten Architekturen aus folgenden Gründen zu erwarten:

- leistungsfähige vergegenständlichte Operationen, die in einem Zyklus ausgeführt werden und Folgen herkömmlicher Befehle ersetzen
- auch bei Verzweigungen voll paralleles Befehlslesen
- voll parallele Adressenrechnung.

7.4.2. Vergleich mit herkömmlichen Architekturen

Die meisten herkömmlichen Rechner können den inneren Parallelismus in Programmen nicht ausnutzen. In jedem Befehl ist nur eine Operation codiert. Meist ist der vollständige Satz von Selektionen (vgl. Tafel 1Ø, S. 75) nicht vorgesehen, so daß zusätzliche Befehle für Adressenrechnung, Laden und Speichern notwendig sind. Verzweigungen kosten oft zusätzliche Zeit. Genauere Vergleichswerte sind nur durch systematische Messungen zu erlangen; vorab läßt sich aber eine plausible Abschätzung treffen: die Vorstellungen zur Befehlscodierung entsprechen näherungsweise Prinzipien der Mikroprogrammsteuerung in größeren Universalrechnern. Wenn man sequentielle Abläufe, z. B. Betriebssystemroutinen, von der üblichen Software in das Mikroprogrammniveau verlagert, so ergibt das erfahrungsgemäß eine 2..2Ø-fache Beschleunigung.[2] Für Programme ohne nutzbaren inneren Parallelismus läßt sich somit durchaus eine wenigstens 2...5 fache Beschleunigung abschätzen, und zwar aus folgenden Gründen: programmseitige Nutzung des schnellsten maschineninternen Zyklus, direkter Zugriff auf

---

1 Bezogen auf den Zyklus Speicher -> Verknüpfung -> Speicher.
2 Nach /146/ 8...15- fach; weitere Erfahrungen ergeben sich z. B. anhand der SVM- Assists bei ESER- Anlagen.

Prozessor	Datenweg-breite	Operations werke	Zyklus-zeit	Leistung MEB/s	Leistung Op/s MFLOP	Bemerkungen
eigener	3 • 64 bit	1	100 ns	1920	10	Standardzellen-technologie
			25 ns	7680 (5120)[1]	40	Technologie wie 80860
	12 • 64 bit	4	100 ns	7680 (5120)[1]	40	Standardzellen-technologie
			25 ns	30 720	160	WSI
Intel 80860	64 bit	2	25 ns	5120[2)]	80	Herstellerangabe
			50 ns	1280[3)]	6,7	Operationsschema 1
					10	Operationsschema 2
Stellar[4]	512 bit	1	50 ns	3456[2)]	36	Herstellerangabe

<u>Operationsschema 1</u>: $c_i := a_i \ \underline{OP} \ b_i$

<u>Operationsschema 2</u>: $c := \sum_i a_i \ \underline{OP} \ b_i$

1 Effektiver Wert für Operationsschema 2 (nutzt nicht alle Datenwege).
2 Aus Herstellerangaben zurückgerechnet für Operationsschema 2.
3 Schätzung für Zugriffe zu chip-externen Speichern (alle 5o ns 64 bit).
4 32-bit- Verarbeitung, mehrere Befehlsströme; die 36 MFLOP sind akkumuliert.

Tafel 23 Leistungsschätzungen und -vergleiche

Hardware- Register, unmittelbare Nutzung aller verfügbaren Datenwege, parallele Verzweigungen; also durch Vermeiden des "overhead" üblicher Maschinenbefehle (in einem kleineren Prozessor sind für einen Maschinenbefehl einer CISC- Befehlsliste im Durchschnitt 2Ø...5Ø Mikrobefehle zu veranschlagen[1]; sind für alle leistungsentscheidenden Abläufe besondere Schaltmittel im notwendigen Umfang vorgesehen, so braucht ein Maschinenbefehl 2...5 Mikrobefehlszyklen[2]).
Es ist aber darüber hinaus erwiesen, daß viele Anwendungsprogramme einen nutzbaren inneren Parallelismus aufweisen. Durch Parallelausführung der Adressenrechnungen, der Parameterübergaben sowie mehrerer Operandenverknüpfungen ist somit ein mehr als 1Ø-facher Leistungsanstieg (bei 3-4 fach höheren Aufwendungen in den Verarbeitungswerken für die parallel angeordneten Schaltmittel) als realisierbar einzuschätzen, das bedeutet bezüglich der gesamten Hardwarekosten eine wenigstens 3-fache Verbesserung des Preis-Leistungs- Verhältnisses.[3]

7.4.3. Vergleich mit RISC- Architekturen

Bei RISC- Maschinen wird versucht, durch einfache Gestaltung der Befehle kurze Zykluszeiten zu erreichen. Es sind aber auch Grenzen erkennbar: Innerhalb eines Schaltkreises sind die Signallaufzeiten deutlich geringer als bei Übergängen zwischen Schaltkreisen. Der Arbeitszyklus einer RISC- Maschine wird aber vom Datenaustausch zwischen Speicher und CPU bestimmt, der beim Stand der Technik mit Sicherheit über Schaltkreisgrenzen hinwegführt, d. h. die Verarbeitungsgeschwindigkeit wird durch einen prinzipiell eher langsamen Ablauf bestimmt.[4]
Schaltkreisanschlüsse und Datenwege sind vergleichsweise kostbare Ressourcen, die so gut wie möglich zu nutzen sind. Das erfordert kompakte, wirkungsvolle Codierungen. Die an sich sinnvolle Absicht, die weitaus meisten Zuordungen in einem einzigen Zyklus auszuführen, sollte nicht auf Kosten des Funktionsumfangs verwirklicht werden. Deshalb sind vergleichsweise komplizierte Zuordnungen vorgesehen, die aber beim Stand der Technik ohne weiteres in einzelnen Schaltkreisen unterzubringen sind. Ein typisches Beispiel ist der Verzicht auf Shift- und Rotate- Operationen durch konsequente Orientierung auf beliebig auswählbare Binärvektoren. Ähnliche Überlegungen sind überall dort angestellt worden, wo es darum ging, von RISC-Prinzipien ausgehend Rechnerarchitekturen für verkaufsfähige Erzeugnisse zu entwickeln, und zwar stets mit dem Resultat: RISC- Prinzipien sind einzubeziehen; sie sind aber nicht als alleinige Grundlage anzusetzen.[5]

---

1 Typisch für kleinere /37Ø- Implementierungen (/154/).
2 Z. B. 37Ø/168, EC 1Ø4Ø, EC 1Ø57.
3 Z. B. kostet ein Hochleistungsrechner 2/3 weniger als ein System aus 1Ø 32-bit- Mikroprozessoren, und der Nutzer spart Bemühungen um die Parallelisierung
4 Das wird auch noch bei WSI gelten (Leitungslängen, Zugriffszeiten großer Speicherarrays).
5 Beispiele: HP Precision Architecture (/229/), Intel 8Ø86Ø; 8Ø486 (/327/; /328/), Motorola 88ØØØ; 68Ø4Ø (/321/; /329/).

7.4.4. Vergleich mit VLIW- Architekturen

Architekturen mit sehr langem Befehlswort haben den eigenen Ansatz maßgeblich beeinflußt.
Ein bekanntes, kommerziell verfügbares System hat ein Befehlswort von 1Ø24 bit, womit 4 Gleitkomma- Rechenwerke und 4 Werke für Integer- Rechnungen gleichzeitig gesteuert werden können. Die Werke sind an einen gemeinsamen Registersatz angeschlossen (64 Integer- Register zu 32 bit, 32 Gleitkommaregister zu 64 bit). Jeder Befehl kann bis zu 28 Aktionen auslösen. Leistungsangabe: 215 VLIW MIPS + 6Ø MFLOP (/147/).
Es gibt keine Schaltmittel, die die Synchronisation der vielen parallelen Abläufe steuern, vielmehr muß der Compiler den Datenfluß aus dem Programmtext erkennen und - mit Kenntnis der Pipeline- Strukturen der Hardware - die Befehle entsprechend aufbereiten. Im System eines anderen Anbieters werden ähnliche Schaltungsstrukturen verwendet, diese sind jedoch durch zusätzliche Schaltmittel ergänzt, die den Datenfluß überwachen und entsprechende Steuerwirkungen ausüben. Der Compiler ist für eine gleichsam grobe Datenflußanalyse zuständig und die Hardware für die Konflikte während der Bearbeitung des Befehlsstromes (diese Lösung wird als "gerichtete Datenflußarchitektur" bezeichnet; /284/).
Die Konzeption solcher Maschinen ist im wesentlichen aus der Datenfluß- und Befehlshäufigkeitsanalyse üblicher Programme hervorgegangen. Hier wurde gezeigt, daß aus einer anderen Sicht praktisch die gleichen Ergebnisse bezüglich der Anzahl der Werke ableitbar sind: Man kann mit 4 Operationswerken den inneren Parallelismus bei wesentlichen numerischen Rechnungen sinnvoll nutzen; auch sind 4 unabhängige Werke zur Adressenrechnung meist ausreichend, sofern sie mit Registersätzen gekoppelt sind, die alle erforderlichen Angaben aufnehmen können. Weiterhin wurde gezeigt, daß ähnliche Gestaltungen der Befehlsformate naheliegend erscheinen, wenn die Hardware allgemein als Sammlung von Ressourcen angesehen wird, die es so vollständig wie möglich auszunutzen gilt.
Die obere Leistungsgrenze(in Op/s) einer gewissen Anzahl von Werken unter der Annahme der lückenlosen Auslastung und des verlustfreien Zugriffs zu den Speichermitteln läßt sich nach (7.2) berechnen: Anzahl der Werke/Zykluszeit. Kein Architekturprinzip, keine Gestaltung der Befehlsformate kann mehr herausholen. Die Erfahrung zeigt aber, daß es beträchtliche Unterschiede zwischen dieser Leistungsangabe und der Leistungsfähigkeit für eine konkrete Anwendung gibt. In diesem Sinne werden folgende Vorteile gegenüber den bekannten VLIW-Lösungen gesehen, die aus den voraufgegangenen Abschnitten teils direkt, teils als einsichtige Zielstellungen für erfinderische Anstrengungen erkennbar sind:

1. gezielter Einsatz parallel angeordneter Werke für Selektionen und Iterationen; damit ist es möglich, die Schaltungskomplexe in sich besser an ihre Aufgaben anzupassen und zu optimieren (die meisten Integer- Operationen sind Rechnungen mit Adressen bzw. Ordinalzahlen; folglich ist es sinnvoll, die betreffenden Schaltmittel von vornherein bevorzugt für diese Aufgaben auszubilden)

2. allgemein nutzbare, komplexe Grundoperationen (z. B. universelle Auswahl-, Transport- und Einfügungsoperationen für beliebige Binärvektoren)

3. parallele Verzweigungen, auch in mehrere Richtungen, sowie parallele Unterprogrammaufrufe

4. einheitliche Codierprinzipen (Binärvektoren variabler Länge als Grundlage der numerischen und nichtnumerischen Informationsverarbeitung, Ordinalzahlen in endliche Mengen als Mittel der Codeverdichtung)

5. schaltungstechnische Unterstützung der objektorientierten Datenorganisation

6. universelle Hochgenauigkeits- Numerikhardware, die mit gleichen Schaltmitteln Gleitkommazahlen, ganze Zahlen und rationale Zahlen zu verarbeiten gestattet

7. Abkehr vom Prinzip des gemeinsamen Hauptspeichers: stattdessen heterogene Speicherstrukturen, um die Eigenschaften der modernen Halbleiterspeicher so direkt wie möglich für die Steigerung der Verarbeitungsleistung einsetzen zu können

8. Entlastung der Compiler von Feinheiten der Ressourcenverwaltung und der Berücksichtigung schaltungsinterner Zeitverhältnisse.

7.4.5. Vergleich mit Sondermaschinen

Vergleiche mit Sondermaschinen sind für den eigenen Ansatz wesentlich: sowohl zur Leistungsbewertung als auch um Schaltmittel aufzufinden, die sich zur Vergegenständlichung im Universalrechner eignen. Im besonderen wird für entscheidende Aufgaben der numerischen und nichtnumerischen Informationsverarbeitung eine möglichst geringe Leistungsminderung im Vergleich zu einschlägigen Sondermaschinen angestrebt (betrifft z. B. das Lösen linearer Gleichungssysteme und elementare Operationen über Relationen).
Für numerische Aufgaben sind Supercomputer, Zusatzprozessoren und Signalprozessoren zu Vergleichszwecken nutzbar. Für höchste Leistungen wird seit längerem der Vektorrechner oder das Parallelverarbeitungssystem bevorzugt und nicht die ausgesprochene Datenstrukturmaschine für einen bestimmten Algorithmenkomplex. Allgemein nutzbare Anwendungsalgorithmen (z. B. zum Lösen linearer Gleichungssysteme) wurden nicht vergegenständlicht, sondern lediglich die leistungsentscheidenden innersten Schleifen daraus oder gar nur Teile solcher Schleifen.[1] Die wichtigsten dieser Abläufe haben die Form "Skalarprodukt" und "Vektor = Vektor + Konstante * Vektor" (SDOT bzw. SAXPY), und es ist anzustreben, sie mit der vollen Verarbei-

1 Für einen Überblick s. etwa /19/, /29Ø/; als konkretes Beispiel vgl. die Entwurfsentscheidungen zum EC1Ø55-Matrixmodul (Hinweis: unter den neuen Bedingungen neu bewerten).

tungsleistung ausführen zu können. Das bereitet mit mehreren Werken für numerische Operationen und Adressenrechnung keine grundsätzlichen Schwierigkeiten. Von Abläufen dieser Art abgesehen ist es offenbar besonders wichtig, die Leistungsfähigkeit der Skalar- Verarbeitung zu erhöhen[1]; dazu scheint die Anordnung mehrerer Werke, zwischen denen der Datenfluß flexibel gesteuert werden kann, in Verbindung mit paralleler Adressenrechnung und Ablaufverzweigung gute Voraussetzungen zu bieten (vgl.die VLIW- Architekturen).
Beispiele für Sondermaschinen der nichtnumerischen Informationsverarbeitung sind Bitprozessoren (für Binärsteuerungen), Datenbasismaschinen, Bilderkennungs- und Verarbeitungseinrichtungen sowie Maschinen zur Implementierung spezieller Sprachkonzepte, wie Lisp oder Prolog. Im folgenden werden die ausgearbeiteten Entwürfe zur Verarbeitung von Ternärvektorlisten (TVL)[2] als Fallbeispiel genutzt, und es werden 2 Aufgaben kurz erläutert:

1. Durchmustern von Relationen (Grundlage für JOIN und INTERSECTION). Das ist voll parallelisierbar, und der Universalrechner kann genau so schnell sein wie die Sondermaschine (Wirkungsgrad $\eta = 1$), sofern allen betreffenden Speichermitteln Durchmusterungsschaltungen nachgeordnet und 2-stufige Iteratoren vergegenständlicht sind (1. Stufe = innerste Schleife: abschnittsweises Durchmustern eines Tupels; 2. Stufe = äußere Schleife: Adressierung der einzelnen Tupel).

2. Orthogonalisierung. Dieser Ablauf ist in der Sondermaschine mit $e_i = 1$ vergegenständlicht; sie liefert in jedem Zyklus einen Abschnitt eines Resultatvektors.[3] Im Universalrechner lohnt sich die Vergegenständlichung nicht (zu geringe Nutzungshäufigkeit bzw. Nützlichkeit für andere Anwendungen). Die universellen vergegenständlichten Abstraktionen gemäß Abschnitt 6 gestatten es aber, den Algorithmus in folgenden Schritten zu implementieren:

I. Aufbauen eines Resultatvektors durch Transport der TVL-Zeile. Das erfordert nur einen Zyklus je Vektor- Abschnitt.

II. Finden der 1. Eins im Markierungsvektor und Löschen derselben. Diese Operation liefert einen Indexwert.

III. Transport des Elementes aus dem Zweitoperanden in den aufgebauten Resultatvektor. Die Position des Elements (Binärvektor aus 2 bit) ist durch den in Schritt II gefundenen Indexwert bestimmt. Das Element ist beim Transport zu negieren.

IV. gemäß besagtem Index Transport des Elementes aus der TVL-Zeile in den Zweitoperanden.

---

1 Vgl. Tafel 4, S. 5Ø.
2 Vgl. /24Ø/ bzw. /243/; für den Algorithmus s. /281/.
3 Die Schaltung nach /241/ liefert in n parallelen Anordnungen in jedem Zyklus je einen Abschnitt von n Resultatvektoren, sie ist aber für die meisten Anwendungen zu aufwendig.

V. Wenn der Markierungsvektor keine Einsen mehr enthält, Ende des Algorithmus, ansonsten weiter mit Schritt I (es wird der nächste Resultatvektor erzeugt).

Schritt I ist ein einfacher Blocktransport. Schritt II ist als vergegenständlichte Operation (1 Zyklus/Abschnitt) vorgesehen, da sie universell für Zwecke der Ablaufsteuerung nutzbar ist. Schritt III erfordert 1...3 Zyklen, je nach den Möglichkeiten des Befehlsformates (im ungünstigsten Fall: 1. Indexwert als Parameter übergeben; 2. Holen des Elements und Negation; 3. Einfügen). Schritt IV kostet höchstens 2 Zyklen; Schritt V läuft parallel ab (Bedingungsabfrage und Spätverzweigung). Der gesamte Algorithmus braucht also nur wenige Zyklen mehr als in der Sondermaschine (bei ansonsten gleichen Voraussetzungen bezüglich der Parameter- Bereitstellung). Diese Leistung wird nur mit allgemein nützlichen Vergegenständlichungen erreicht, also ohne algorithmenspezifische Schaltungen.

8. Empfehlungen und Ausblicke

Zunächst werden einige Möglichkeiten aufgezählt, die vorgestellten Ansätze weiter auszubauen:

1. Verfeinerung der Formalisierungs- Ansätze von Abschnitt 3.2. bis hin zum algebraischen Modell unter Einbeziehung der Schaltungsstrukturen. Damit dürfte sich ein formaler Zugang zu den Fragen der Funktions - Struktur - Abbildungen erschließen lassen. Das Ausarbeiten einer Struktur, die eine gegebene funktionelle Spezifikation erfüllt, wird gelegentlich als das eigentlich Schöpferische am Entwurfsprozeß bezeichnet.[1] Tatsächlich ist eine voll befriedigende Lösung noch nicht gelungen.[2] Für ein Vorwärtskommen auf diesem Wege ist z. B. folgendes Szenarium denkbar:

- Die grundlegenden Datenstrukturen werden bis aufs Bit definiert, und die Algorithmen werden durch Boolesche Gleichungen beschrieben. Dem Systementwerfer wird dafür eine komfortable Benutzerschnittstelle angeboten, die verschiedene Formen der Erfassung zuläßt (Hardware- Beschreibungssprachen, Wahrheitstabellen, Impulsdiagramme usw.).[3] Rechnerintern werden die Booleschen Gleichungen in einheitlicher Form dargestellt.[4]

- Durch Dekomposition für das gegebene Bauelementesortiment werden die Schaltungen für die einzelnen Algorithmen automatisch erzeugt.[5]

- Gemäß dem Vorgehen von Abschnitt 3.1. werden die Einzelschaltungen zu universellen Strukturen zusammengefaßt. Dazu werden systematisch Auswahlschaltungen , gemeinsame Speichermittel und Steuerschaltungen angeordnet. Die innere Struktur der Schaltungen wird analysiert, und homologe Strukturen werden vereinigt, wobei die Auswahl- und Steuerschaltungen entsprechend abgewandelt werden.

2. Ausbau der Bewertungskriterien von Abschnitt 4 und deren programmseitige Implementierung, so daß sie auf praxisübliche Verhältnisse (Algorithmen und Maschinen) anwendbar sind und quantitative Ergebnisse liefern. Die vorgeschlagenen Konzepte werden untersucht und mit gegebenen Lösungen sowie mit andern Forschungsansätzen verglichen.

3. Ausbau des Ressourcen- Konzepts (Abschnitte 3.2. und 7.2.) mit dem Ziel, Ressourcenkomplexe rechnerisch optimieren zu können.[6]

---

1 Vgl. z. B. entsprechende Aussagen in /14/.
2 Vgl. die Diskussion unter dem Stichwort "silicon compiler".
3 In modernen "silicon compiler"- Systemen bereits gegeben.
4 Beispielsweise durch Ternärvektorlisten (/6/, /282/); die Wandlung zwischen diesen und üblichen Entwurfsdatenmassiven wird beherrscht.
5 Das ist rechentechnisch lösbar (z. B. nach /164/).
6 Z. B. mittels linearer Optimierung über Aufwand, Leistung und Nutzungshäufigkeit (vgl. S. 8).

4. Nutzung der Betrachtungsweise von Abschnitt 5 für eine systematische Struktur- und Funktionsbeschreibung von Rechnerarchitekturen (im Sinne einer Gesamtschau).[1]

5. Systematisches Absuchen der Grundlagen der Informatik nach weiteren Tiefenstrukturen, die sich für die technische Umsetzung in vergegenständlichte Abstraktionen eignen. Beispiele: angewandte Mathematik im weitesten Sinne, formale Linguistik, verschiedene Logik- Kalküle (besonders solche, die über den Prädikatenkalkül 1. Stufe hinausgehen)[2], Signalverarbeitung, Simulationsprobleme verschiedener Art[3], Methoden zur Programmverifikation sowie Bilderkennung, -verarbeitung und -darstellung (farbig, bewegt, 3-dimensional).[4]

6. Weiterführung von Überlegungen der Abschnitte 6 und 7 zu konkreten Vorschlägen einer künftigen Architektur bzw. einer Familie von Architekturen.[5]

Es ist natürlich das vordringliche Ziel, die Architektur- und Schaltungsprinzipien eines Hochleistungsrechners auszuarbeiten. Um die neue Architektur bewerten zu können, müssen alle Prinzipen in ihren Einzelheiten dargestellt sein, und die Schaltungsstrukturen müssen so detailliert beschrieben sein, daß deren Funktionsfähigkeit und Realisierbarkeit beurteilbar ist.[6] Zu Bewertungs- bzw. Vergleichszwecken und auch als Quelle von Anregungen sind Sondermaschinen (reale und fiktive) für wichtige Anwendungsbereiche zu betrachten. Das betrifft herkömmliche Nutzungsfälle der numerischen und nichtnumerischen Informationsverarbeitung; es sind aber auch Forschungsarbeiten zu Maschinen für Konzepte wie Lisp, Prolog, Smalltalk u. a. zu berücksichtigen.
Eine neue Architektur wird nur dann akzeptiert werden, wenn sie ein außergewöhnlich gutes Ziel für Compiler darstellt; es ergeben sich also enge Beziegungen zu den einschlägigen Forschungsgebieten. Im besonderen sind Fragen der Datenflußanalyse und der optimalen Ressourcennutzung von Bedeutung; auch unter dem Gesichtspunkt, für diese wichtigen Aufgaben nach Ansätzen zur hardwareseitigen Unterstützung zu suchen.[7]

---

1 Die systematische Darstellung beschränkt sich meist auf elementare Zusammenhänge. Kompliziertere Sachverhalte bzw. Einzelheiten werden häufig nur anhand von Beispielen realer Maschinen erklärt; Erscheinungen werden also nicht auf das Wesen zurückgeführt (Beispiele für einschlägige Standardwerke: /4/, /19/ /27/, /86/, /88/).
2 Für Anregungen s. z. B. /5/, /18/, /93/.
3 Das betrifft Verfahren und Algorithmen für physikalische Phänomene, logische Funktionen, neuronale Netze usw.
4 Vgl. die "Graphic Supercomputer" (/126/, /151/, /157/, /318/), Intel 80860 (/327/) und andere Prozessoren (z. B. /162/, /276/).
5 Für einen ersten Überblick sei auf /248/ verwiesen.
6 /240/ ist ein Beispiel für diesen Grad an Detailliertheit.
7 D. h., Mittel zur schaltungstechnischen Unterstützung sind im Sinne einer Gesamtoptimierung auf Compilier- und Verarbeitungsprozesse zu verteilen.

Fragen der Parallelverarbeitung durch Mehrprozessorkonzepte vielfältigster Art sind bisher bewußt ausgespart worden; es wird aber notwendig werden, sie in umfassend zu bearbeiten: ein künftiger Einzelprozessor muß sich im Rahmen solcher Konzepte einsetzen lassen, er sollte sogar in besonderer Weise - besser als seine Vorgänger - dafür geeignet sein.[1]
Schließlich können die Fragen der Kompatibilität, der Nutzbarkeit vorhandener Programme und Datenbestände nicht weiter vernachlässigt werden. Sollten genauere Untersuchungen zeigen, daß das neue Architekturkonzept nicht so deutlich überlegen ist, daß sich also dessen Einführung nicht lohnt, so sollten Teilergebnisse daraufhin überprüft werden, ob sie sich zur Verfeinerung bzw. Ergänzung eingeführter Systeme eignen (für schaltungstechnische Verbesserungen in Steuerwerken, Speichern, Rechenwerken oder auch für Ergänzungs- und Beschleunigungsschaltungen).[2] Dann sind die Datenstrukturen und Wirkprinzipien an die Konventionen dieser Systeme anzupassen.
Erweist sich hingegen das neue Konzept als ausreichend überlegen, so bilden dessen Konventionen einen neuen Quasi- Standard, und für die praktische Nutzung sind die Fragen der "Aufwärtskompatibilität" mit Nachdruck zu bearbeiten. Dazu abschließend einige Überlegungen:

1. Der Hochleistungsrechner ist nicht vorrangig als Ersatz für Mikroprozessoren, Minicomputer oder EDV- Anlagen vorgesehen, sondern als technisches Mittel, um neue Gebrauchswerte schaffen zu können, die mit der bisherigen Ausrüstung nicht praktisch genutzt oder gar nicht implementiert werden konnten (z. B. 3D- Graphik mit Farbe und Bewegung; relationale wissensbasierte Systeme). Diese neuen Gebrauchswerte sind also von Grund auf zu erarbeiten, so daß sich Kompatibilitätsfragen klären lassen, indem entsprechende Compiler und allgemein akzeptierte Entwicklungsumgebungen bereitgestellt werden.

2. Es gehört zum Stand der Technik, für neue, unkonventionelle Architekturen eine eingeführte Programmierumgebung (konkret: Unix + C) bereitzustellen. Eine Alternative besteht darin, die Programmierumgebungen und Compiler- Schnittstellen zu Systemen kompatibel zu gestalten, die bei den wichtigsten Anwender-Zielgruppen weit verbreitet sind.[3]

3. Vom Üblichen abweichende interne Datendarstellungen bereiten keine besonderen Schwierigkeiten, da die Ein- und Ausgabefunktionen grundsätzlich über wohldefinierte Hardware- und Software- Schnittstellen abgewickelt werden. Gerätetreiberroutinen können die Datenkonvertierung übernehmen, und die Nutzung von Mikrorechnern als E/A- Prozessoren gehört zum Stand der Technik.

---

1 Auf dem Gebiet der Mikroprozessoren ist das "Transputer"-Konzept eine solche Lösung. Es gilt, im oberen Leistungsbereich Ähnliches zu schaffen.
2 Beispiel: Nutzung von RISC- Konzepten bzw. Entwicklungszielen (Befehlsausführung in einem Zyklus) bei den Mikroprozessoren 80486 und 68 040 (/328/, /329/).
3 Etwa zu Cray- und VAX- Systemen (z. B. Fa. Convex).

4. Von besonderem Interesse ist es, vorhandene Programme und Datenbestände im Verbund mit neuen nutzen zu können. Das soll folgendermaßen gelöst werden:
Die Programme und Datenbestände werden (so wie sie sind) in die gemeinsame Datenbasis aufgenommen.[1]
Es ist naheliegend, Emulatorprogramme für die wichtigsten eingeführten Rechnerarchitekturen zu schaffen. Das ist aber meist mit einem deutlichen Leistungsverlust gegenüber der Zielmaschine verbunden. Deshalb werden besondere Schaltmittel vorgesehen und an entsprechender Stelle der Speicherhierarchie angeschlossen. Das ist in Bild 5Ø (S. 157) bereits dargestellt. Programmkomplexe, die für Mikroprozessor- Architekturen vorgesehen sind, werden lauffähig, indem entsprechende Mikroprozessor- Anordnungen an die Datenbasisspeicher angeschaltet werden (das bloße Nutzen eines zuhandenen Mikroprozessors erfordert gegenüber dessen programmtechnischer Emulation wesentlich geringere Entwicklungsaufwendungen und ist im Leistungsvermögen deutlich überlegen). Für eingeführte Minicomputer- bzw. EDVA- Architekturen, für die es keine Mikroprozessorversionen gibt, wird eine Kombination aus programmtechnischer Emulation und schaltungstechnischer Unterstützung vorgesehen, um mit beherrschbaren Aufwendungen die Leistung der jeweiligen Zielmaschine zu erreichen bzw. zu übertreffen.[2]
Für jede Zielarchitektur wird ein "Incarnated Interpreter Module" (I^2M) entwickelt. In diesem sind die Speicheradressierungsprinzipien und die Register der Zielarchitektur vergegenständlicht. Der I^2M übernimmt das Befehlslesen. Elementare Befehle werden vom I^2M unmittelbar ausgeführt; für kompliziertere Abläufe (z. B. Gleitkommarechnung) werden Emulatorroutinen im Hochleistungsrechner zu Hilfe gerufen.

---

1 Ein übergeordnetes Verwaltungssystem der Datenbasis gewährleistet die korrekte Zuordnung der gespeicherten Informationsstrukturen zu den verschiedenen Architektur- Implementierungen.

2 Es geht darum, ein gewisses mittleres Leistungsniveau zu halten, um vorhandene Programme genauso schnell abarbeiten zu können wie auf den jeweiligen Vorgänger- Maschinen. (Für neue Gebrauchswerte sollten die Eigenschaften der neuen Architektur genutzt werden.) In /1Ø5/ sind verschiedene Möglichkeiten beschrieben, wie die Funktionen einer CISC-Architektur auf mehrere mikroprozessor- ähnliche Schaltkreise aufgeteilt werden können. Aufbauend darauf wird hier eine Ausgestaltung vorgeschlagen, um mit erträglichen Aufwendungen das angestrebte Leistungsniveau zu gewährleisten.

9. Zusammenfassung

Die Arbeit dient dazu, für Forschungen, die zum Ziel haben, Architekturprinzipien und technische Lösungen für Hochleistungsrechner auszuarbeiten, die Arbeitsrichtung zu begründen, erste Vorstellungen zu umreißen und Anregungen für das wissenschaftlich- technische Handeln zu vermitteln.
Dazu werden Algorithmen, Datenstrukturen, Sprachkonstrukte und Schaltungsstrukturen aus ganzheitlicher Sicht betrachtet. Dem liegt ein Modell der Informationsverarbeitung zugrunde, das besonders unter dem Gesichtspunkt gewählt wurde, technische Mittel so leistungsfähig und zweckmäßig wie möglich gestalten zu können.
Für die Bewertung von Schaltungsstrukturen (Leistungsvermögen), Algorithmen (Eignung zur Vergegenständlichung), Aufwendungen (Nützlicheit) und Universalmaschinen insgesamt (Wirkungsgrad) werden Prinzipien angegeben.
In der Arbeit werden die bekannten Ansätze zur Vereinfachung von Befehlslisten und zur Nutzung des in üblichen Programmen gegebenen Parallelismus aufgegriffen und durch einen weiteren Ansatz ergänzt, der davon ausgeht, daß man für jeden einzelnen Algorithmus bzw. Komplex von Algorithmen und Datenstrukturen ohne grundsätzliche Schwierigkeiten Sondermaschinen (Datenstrukturmaschinen) angeben kann, die - bei Realisierbarkeit mit zuhandenen technischen Mitteln - im Leistungsvermögen jedem Universalrechner weit überlegen sind. Eine solche technische Umsetzung eines Algorithmus wird als Vergegenständlichung (im Gegensatz zur programmseitigen Implementierung) bezeichnet. Durch ein systematisches Absuchen der Grundlagen der Informatik sollen die Algorithmen gefunden werden, die dafür besonders geeignet sind. Das wird anhand von Beispielen vorgeführt, die aus der Betrachtung eingeführter Rechnerarchitekturen und höherer Programmiersprachen gewonnen wurden.
Es wird untersucht, wie man hochleistungsfähige Einzelschaltungen zu Universalmaschinen zusammenfassen kann. Die Universalmaschine wird als Sammlung von Ressourcen betrachtet, die in diskreten Schritten (Maschinenzyklen) zu steuern und so gut wie möglich auszunutzen sind. Das ermöglicht einen einheitlichen Zugang zu verschiedenen Architekturkonzepten. Auf diesem Wege wurden teils Ergebnisse erzielt, die denen anderer Forschungsansätze ähnlich sind, teils neue Ziele für das technisch- erfinderische Handeln erkannt.

1Ø. Literaturverzeichnis

Abkürzungen

Academic Press steht für Academic Press, New York-London bzw. London-Orlando-San Diego-Toronto-Montreal-Tokyo.
North Holland steht für North Holland Publishing Co., Amsterdam-New York-Oxford-Tokyo.
Mc Graw- Hill steht für McGraw- Hill, New York-London-Toronto.
Oldenbourg steht für Oldenbourg- Verlag, München-Wien.
Plenum steht für Plenum Press, New York-London.
Prentice Hall steht für Prentice Hall Inc., Englewood Cliffs.
Springer steht für Springer- Verlag, Berlin-Heidelberg-New York bzw. Berlin-Heidelberg-New York-Tokyo.
Wiley steht für Wiley & Sons, New York-Toronto bzw. New York-London-Sydney-Toronto.
IFB steht für Informatik- Fachberichte.
LNCS steht für Lecture Notes in Computer Sciences.

Sammelwerke

/1/ Barbacci, M. R.; Koomen, C. J. (eds.): Computer Hardware Description Languages and Their Applications. North- Holland, 1987.

/2/ Barbe, D. F. (ed.): Very Large Scale Integration (VLSI) Fundamentals and Applications. Springer, 198Ø.

/3/ Becker, J. D.; Eisele, I. (eds.): WOPPLOT 86. Parallel Processing: Logic, Organization and Technology. Springer, 1986 (LNCS 253).

/4/ Bell, C.; Newell, A. (eds.): Computer Structures: Readings and Examples. McGraw- Hill, 1971.

/5/ Berka, K.; Kreiser, L. (Herausg.): Logik- Texte. Akademie- Verlag, Berlin 1971.

/6/ Bochmann, D.; Zakrevskij, A. D.; Posthoff, Ch. (Herausg.): Boolesche Gleichungen. Verlag Technik, Berlin 1984.

/7/ Buchholz, W. (ed.): Planning a Computer System - Project Stretch. McGraw- Hill, 1962.

/8/ Dierstein, R.; Müller- Wichards, D.; Wacker, H. M. (eds.): Parallel Computing in Science and Engineering. Springer, 1988 (LNCS 295).

/9/ Dongarra, J. J. (ed.): Experimental Parallel Computing Architectures. North- Holland, 1987 (Special Topics in Supercomputing Vol.1).

/1Ø/ Enslow, P. H. (ed.): Multiprocessors and Parallel Processing. Wiley, 1974 (russ. Übers. isd-wo Mir, Moskau 1976).

/11/ Flynn, M. J.; Harris, N. R.; McCarthy, D. P. (eds.): Microcomputer System Design. An Advanced Course Dublin 1981. Springer, 1892 (LNCS 126).

/12/ Giloi, W. K. (ed.): Firmware Engineering. Springer, 198Ø (IFB 31).

/13/ Güth, R. (ed.): Computer Systems for Process Control. Plenum, 1985.

/14/ Hasselmeier, H.; Spruth, W. G. (Herausg.): Rechnerstrukturen. Vorträge des Informatik- Symposiums der IBM Deutschland Wildbad 1973. Oldenbourg, 1974.
/15/ Heyer, G.; Krem, J.; Görz, G. (Herausg.): Wissensbasen und ihre Darstellung. Springer, 1987 (IFB 169).
/16/ Houstis, E. N.; Papatheodoru, T. S.; Polychronopoulos, C. D. (eds.): Supercomputing. 1st International Conference Athens, Greece, June 1988 Proceedings. Springer, 1988 (LNCS 297).
/17/ Kowalik, J. S. (ed.): High- Speed Computation. Springer, 1984.
/18/ Kreiser, L.; Gottwald, S.; Stelzner, W. (Herausg.): Nichtklassische Logik. Akademie- Verlag, Berlin 1988.
/19/ Kuck, D. J.; Lawrie, D. H.; Samek, A. H. (eds.): High Speed Computer and Algorithm Organization. Academic Press, 1977.
/2Ø/ Kuhn, R. H.; Padua, D. A. (eds.): Tutorial on Parallel Processing. IEEE Computer Society, Los Angeles 1981.
/21/ Kung, H. T.; Sproull, B.; Steele, G. (eds.): VLSI Systems and Computations. Springer, 1981.
/22/ Miranker, W. L.; Toupin, R. A. (eds.): Accurate Scientific Computations. Springer, 1986 (LNCS 235).
/23/ Misunas, D. P. (ed.): Report on the Workshop on Data Flow Computer and Program Organization. MIT/LCS/TM-92, Cambridge, Mass. 1977.
/24/ Numrich, R. W. (ed.): Supercomputer Applications. Plenum, 1985.
/25/ Painke, H. (ed.): Digital Technology. Status and Trends. Oldenbourg, 1981 (Fachberichte und Referate Bd. 12).
/26/ Reijns, G. L.; Barton, M. H. (eds.): Highly Parallel Computers. North Holland, 1987.
/27/ Siewiorek, D.; Bell, G.; Newell, A. (eds): Computer Structures: Principles and Examples. McGraw- Hill, 1982.
/28/ Uhr, L. (ed.): Parallel Computer Visison. Academic Press, 1987.
/29/ Wallis, P. J. L. (ed.): Ada: managing the transition. Proceedings of the Ada- Europe International Conference Edinburgh 6-8 May 1986. Cambridge University Press, Cambridge 1986.
/3Ø/ Wasserman, A. I. (ed.): Tutorial: Programming Language Design. IEEE Computer Society, New York 198Ø.

<u>Konferenzberichte, Sonderausgaben, Standards, Firmenschriften, Neuigkeitsmeldungen</u>

/31/ AFIPS Conference Proceedings 1986 NCC.
/32/ AFIPS Conference Proceedings 1987 NCC.
/33/ The Programming Language Ada Reference Manual. American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983. Springer, 1983 (LNCS 155).

/34/ Second International Conference on Architectural Support for Programming Languages and Operating Systems. Operating Systems Review Vol. 21 No. 4 (October 1987).
/35/ Proceedings 5th Symposium on Computer Arithmetic May 18-19, 1981 University of Michigan, Ann Arbor, Michigan. IEEE Computer Society 1981.
/36/ IBM System/370 Principles of Operation (fourth Edition; January 1973). Russ. Übers. isd-wo Mir, Moskau 1975.
/37/ AP-120B Processor Handbook, February 1989 (Excerpts). In: /20/, S.41-47.
/38/ Motorola MC 68020 32-Bit Microprocessor User's Manual. Prentice-Hall, 1984.
/39/ IBM Applicaion System/400 Technology. IBM- Firmenschrift 1988.
/40/ Robotron EDVA R40. rechentechnik datenverarbeitung (9), Heft 10/11, 1972.
/41/ -: First TRON Microprocessor gets Japan into the 32-bit-fray. Electronics Vol. 61 No. 2 (January 21, 1988), S. 31, 32.
/42/ -: Sun Microsystems: Die Sonne strahlt weiter. bit Juli/August 1988, S. 50-52.
/43/ -: Der AMD 29 000. Chip 1, 1989, S. 106-108.
/44/ Bronstein, I. N.; Semendjajew, K. A. Taschenbuch der Mathematik. Teubner Verlag, Leipzig 1968.
/45/ Kleine Enzyklopädie Deutsche Sprache. Bibliographisches Institut, Leipzig 1983.
/46/ Kondakow, N. I.: Wörterbuch der Logik. Bibliographischen Institut, Leipzig 1978.
/47/ Steinbuch, K.; Weber, W. (Herausg.): Taschenbuch der Informatik (3. Aufl.). Springer, 1974.
/48/ Bochmann, D.: Persönliche Kommunikation 1982- 1989.

Lehrbücher und Monographien

/49/ Addis, T. R.: Designing Knowledge- Based Systems. Kogan Page, London 1985.
/50/ Baer, J. L.: Computer Systems Architecture. Computer Sciences Press 1980.
/51/ Bochmann, D.: Automatengraphen. Akademie- Verlag, Berlin 1982.
/52/ Bochmann, D.; Posthoff, Ch.: Binäre dynamische Systeme. Akademie- Verlag, Berlin 1982.
/53/ Boulaye, G. G.: Microprogramming. Carl Hanser Verlag, München; MacMillan, London 1975.
/54/ Chomsky, N.: Aspekte der Syntax- Theorie. Akademie-Verlag, Berlin 1970.
/55/ Claßen, L.; Oefler, U.: UNIX und C. Ein Arbeitsbuch. Verlag Technik, Berlin 1987.
/56/ Dasgupta, S.: The Design and Description of Computer Architectures. Wiley 1984.
/57/ Dijkstra, E. W.: A Discipline of Programming. Prentice- Hall, 1976.

/58/ Fountain, T.: Processor arrays: architectures and applications. Academic Press, 1987.
/59/ Frank, T. S.: Introduction to VAX-11 Architecture and Assembly Language. Prentice-Hall, 1987.
/60/ Gehani, N.: Ada. An Advanced Introduction. Prentice-Hall, 1983.
/61/ Giloi, W. K.: Rechnerarchitektur. Springer, 1981.
/62/ -: Programmieren in APL. Walter de Gruyter, Berlin - New York 1977.
/63/ Gluschkow, V. M.: Einführung in die technische Kybernetik. Verlag Technik, Berlin 1970.
/64/ Golovkin: Parallelnye vychislitelnye sistemy. Nauka, Moskau 1980 (russ.).
/65/ Grass, W.: Steuerwerke. Entwurf von Schaltwerken mit Festwertspeichern. Springer 1978.
/66/ Habermann, A. N.: Entwurf von Betriebsystemen - eine Einführung. Springer, 1981.
/67/ Hagemann, R. Allgemeine Genetik. Gustav Fischer Verlag, Jena 1984.
/68/ Haupt, D.: Mengenlehre. Fachbuchverlag, Leipzig 1966.
/69/ Hedtke, R.: Mikroprozessorsysteme. Zuverlässigkeit, Testverfahren, Fehlertoleranz. Springer, 1984.
/70/ Hellerman, H.; Smith, I. A.: APL/360. Programming and Applications. McGraw- Hill, 1976 (russ. Übers. Mashinostrojenie, Moskau 1982).
/71/ Hibbard, P. et al.: Studies in Ada Style. Springer, 1981.
/72/ Higman, B.: Programmiersprachen - eine vergleichende Studie. Teubner- Verlag, Leipzig 1971.
/73/ Hoffmann, R.: Rechenwerke und Mikroprogrammierung. Oldenbourg, 1977.
/74/ Kämmerer, W.: Digitale Automaten. Akademie- Verlag, Berlin 1969.
/75/ Katys, G. P.: Optiko- elektronnaja obrabotka informacii. Mashinostrojenie, Moskau 1973 (russ.).
/76/ Kogge, P. M.: The Architecture of Pipelined Computers. Hemisphere Publishing Co., Washington- New York- London 1981 (russ. Übers. radio i sviaz, Moskau 1985).
/77/ Kulisch, U. W.; Miranker, W. L.: Computer Arithmetic in Theory and Practice. Academic Press, 1981.
/78/ Kung, S. Y.: VLSI Array Processors. Prentice-Hall, 1988.
/79/ Levy, H. M.: Capability-Based Computer Systems. Digital Press, Digital Equipment Corporation 1984.
/80/ Levy, H. M.; Eckhouse, R. H.: Computer Programming and Architecture - The VAX 11. Digital Press, Digital Equipment Corporation 1980.
/81/ Loeper, H.;Jäkel, H.-J.; Otter, W.: Compiler und Interpreter für höhere Programmiersprachen. Akademie- Verlag, Berlin 1987.
/82/ Majorov, S. A.; Novikov, G. I.: Principy organizacii cifrovych mashin. Mashinostrojenie, Leningrad 1974 (russ.).

/83/ Mead, Conway: Introduction to VLSI Systems. Addison-Wesley, Reading, Mass. 1980.
/84/ Miller, R. E.; Thatcher, J. W.: Complexity of Computer Computations. Plenum Press, 1972.
/85/ Möschwitzer, A.; Rößler, F.: VLSI- Systeme. Verlag Technik, Berlin 1988.
/86/ Myers, G. J.: Advances in Computer Architecture. Wiley, 1982 (russ. Übers. isd-wo Mir, Moskau 1985.).
/87/ Newell, A.; Simon, H. A.: Mind Design. MIT Press, 1982.
/88/ Organick, E. I.: Computer Systems Organization. The B 5700/B 6700 Series. Academic Press, 1973.
/89/ -: A Programmers View of the Intel 432. McGraw- Hill 1982.
/90/ Riedewald, G.; Matuszynski, J.; Dembinski, P.: Formale Beschreibung von Pogrammiersprachen. Akademie-Verlag, Berlin 1983.
/91/ Schendel, U.: Einführung in die parallele Numerik. Oldenbourg, 1981.
/92/ Schiller, W.: Programmiersprache PL/1. Verlag Technik, Berlin 1971.
/93/ Sinowjew, A. A.: Über mehrwertige Logik. Deutscher Verlag der Wissenschaften, Berlin 1968.
/94/ Starke, P. H.: Abstrake Automaten. Deutscher Verlag der Wissenschaften, Berlin 1969.
/95/ Strauss, E.: Inside the 80286. Prentice-Hall, 1986.
/96/ Träger, L.: Einführung in die Molekularbiologie. Gustav Fischer Verlag, Jena 1975.
/97/ Wadge, W. W.; Ashcroft, E. A.: Lucid, the Dataflow Programming Language. Academic Press, 1985.
/98/ Wendt, S.: Entwurf komplexer Schaltwerke. Springer, 1974.
/99/ Wodjacho, A. I.; Smolov, W. B.; Plusnin, W. U.; Pusankov, D. W.: Funkcionalno- orientirovannye prozessory. Mashinostrojenie, Leningrad 1988 (russ.).
/100/ Zuse, K.: Beschreibung des Plankalküls. Oldenbourg, 1977 (GMD- Berichte Nr. 112).
/101/ -: Der Computer - Mein Lebenswerk. Springer 1986.

Einzeldarstellungen

/102/ Ackerman, W. B.: Data Flow Languages. AFIPS Conf. Proc. Vol. 48 (1979), S. 1087-1095 (in: /20/, S. 335-344).
/103/ Albrecht, A.: Komplexitätstheoretische Aspekte des Entwurfs von VLSI- Systemen. Humboldt- Universität, Sektion Mathematik, Berlin 1982 (Preprint Nr. 28).
/104/ -: Kompliziertheitsprobleme Boolescher Funktionen und Gleichungen. In: /6/, S. 151-159.
/105/ Agnew, P. W.; Kellerman, A. S.: Microprocessor Implementation of Mainframe Processors by Means of Architecture Partitioning. IBM J. Res. Dev. Vol. 26 No. 4 (July 1982), S. 401-412.

/1Ø6/ Amdahl, G.: The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS Conf. Proc. Vol. 3Ø (1967).
/1Ø7/ Andrews, W.: Static RAMs race to keep up with RISC. Computer Design Vol.28 No. 7 (April 1, 1989), S. 59-66.
/1Ø8/ Arvind; Iannucci, R. A.: Two Fundamental Issues in Multiprocessing. In: /8/, S. 61-88.
/1Ø9/ Aspinall, D.: Properties of Instruction Set Processor. In: /11/, S. 138-181.
/11Ø/ Annaronte, M. et al.: WARP architecture: From prototype to production. In: /32/, S. 133-14Ø.
/111/ Athas, W. C.; Seitz, C. L.: Multicomputers: Message-Passing Concurrent Computers. Computer, Vol. 21 No. 8 (August 1988), S. 9-24.
/112/ Atkinson, R. R.; McCreight, E. M.: The Dragon Processor. In: /34/, S. 65-69.
/113/ August, M. C. et al.: Cray X-MP: The Birth of a Supercomputer. Computer Vol. 22 No. 1 (January 1989), S. 45-52.
/114/ Azar, C.; Butts, M.: Hardware accelerator picks up the pace across full design cycle. Electronic Design Vol. 33 No. 24 (Oct. 17, 1985), S. 177-184.
/115/ Baskett, F.; Keller, T. W.: An Evaluation of the CRAY-1 Computer. In: /19/, S. 71-84.
/116/ Bassak, G.: Special Report: Designing Computers. Electronic Design Vol 32 No. 8 (April 19, 1984), S. 91-1Ø4.
/117/ Bectem, J. et al.: The GF 11 Parallel Computer. In: /9/, S. 255-297.
/118/ Behr, P. M.; Giloi, W. M.; Mühlenbein, H.: Rationale and Concepts for the SUPRENUM Supercomputer Architecture. In: /26/, S. 1-17.
/119/ Berra, P. B.; Oliver, E.: The Role of Associative Array Processors in Data Base Machine Architecture. Computer Vol. 12 No. 3 (March 1979), S. 53-61 (in: /2Ø/, S. 125-133.
/12Ø/ Birnbaum, J. S.; Worley, W. S.: Beyond RISC: High-Precision Architecture. Hewlett- Packard Journal Vol. 36 No. 8 (August 1985), S. 4-1Ø.
/121/ Block, E.; Galage, D. J.: Component Progress: Its Effect on High Speed Computer Architecture and Machine Organization. In: /19/, S. 13-39.
/122/ Bochmann, D.; Matthes, W.; Steinbach, B.: WP 235 744. Anordnung zur Verarbeitung von Ternärvektorlisten.
/123/ -: WP 235 743. Anordnung zur Orthogonalisierung von Ternärvektorlisten.
/124/ -: WP 236 822. Anordnung zur Bestimmung und Auswertung der Lösungszahlen orthogonaler Ternärvektorlisten.
/125/ Bode, A. et al.: A Highly Parallel Architecture Based on a Distributed Shared Memory. In: /26/, S. 19-27.

/126/ Bond, J.: New Architectures Give Supercomputers Power to Mirror Reality. Computer Design Vol. 27 No. 6 (March 15, 1988), S. 67-88.
/127/ Boral, H.; DeWitt, D. J.: Database machines: An idea whose time has passed. A critique of the future of database machines. Technical Report /288 Israel Institute of Technology, Department of Computer Science, Haifa 1983.
/128/ Branscomb, L. M.; Thomas, J. C.: Ease of use: A system design challenge. IBM Systems Journal Vol. 23 No. 3 (1984), S. 224-235.
/129/ Bunata, T.: Huber, B.: 16-Bit-Mikroprozessor für Echtzeit- Multitask- Betrieb. Elektronik Heft 7, 1983, S. 53-57.
/13Ø/ Bursky, D.: Multiple processors, memory merge on chip. Electronic Design Vol. 36 No. 13 (June 9, 1988), S. 26, 27.
/131/ -: Configurable Chip Eases Control- System Design. Electronic Design Vol. 36 No. 13 (June 9, 1988), S. 63-7Ø.
/132/ -: Digital GaAs: Slowly making inroads into large systems. Electronic Design Vol. 36 No. 13 (June 9, 1988), S. 29, 3Ø.
/133/ Campbell, J. E.; Tahmmoush, J.: Design Considerations for a VLSI Microprocessor. IBM J. Res. Dev. Vo. 26 No. 6 (July 1982), S. 454-463.
/134/ Cappello, P. R.; Miranker, W. L.: Systolic Super Summation. IEEE TC Vol. 37 No. 6 (June 1988), S. 657-677.
/135/ Case, B.: Der "Reduced-Instruction-Set-Computer" (RISC). Elektronik, Heft 23, 1985, S. 61-68.
/136/ Chang, D. Y.; Kuck, D. J.; Lawrie, D. H.: On the Effective Bandwidth of Parallel memories. IEEE TC-26 (May 1977), S. 48Ø-489 (in: /2Ø/, S. 174-184.
/137/ Chen, S. S.: Large-Scale and High-Speed Multiprocessor System for Scientific Applications: CRAY X-MP Series. In: /17/, S. 59-67.
/138/ Ching, W.-M.: An Extented von Neumann Model for Parallel Processing. In: /31/, S. 363-371.
/139/ Chow, F. et al.: How many Addressing Modes Are Enough? In: /34/, S. 117-121.
/14Ø/ Chu, P.; Kitson, B.; Tabler, O.: Smart controller meets disc- memory challenge. Electronic Design Vol. 3Ø No. 13, S. 133-142.
/141/ Chu, P.; New, B. J.: Mikroprogrammierbare 32-Bit-Bausteine. Elektronik, Heft 22, 1984 S. 81-88.
/142/ Chung, M. J.; Toy, E. J.; Aarti, G.: A parallel computer based on cube connected cycles for wafer scale integration. In: /31/, S. 325-335.
/143/ Ciarcia, S.: Why Microcontrollers? Byte Vol. 13 No. 8 (August 1988), S. 239-247.
/144/ Clementi, E.; Detrich, J.: Large Scale Parallel Computation on a Loosely Coupled Array of Processors. In: /9/, S. 141-175.

/145/ Cole, B. C.: Now 16-Bit Processors Can Handle the Speeds of Laser Printing. Electronics Vol. 61 No. 13 (July 1988), S. 101, 102.
/146/ Colwell, R. P.: The Performance Effects of Functional Migration and Architectural Complexity in Object- Oriented Systems. Department of Computer Science Carnegie- Mellon University, CMU-CS-85-159, Pittsburgh, Pa. 1985.
/147/ Colwell, R. P. et al.: A VLIW Architecture for a Trace Scheduling Computer. In: /34/, S. 180-192.
/148/ Conklin, J.: Hypertext: An Introduction and Survey. Computer Vol 20 No. 9 (September 1987), S. 17-41.
/149/ Cordero, H.: 4341's infrastructure is new from the substrate up. Electronics, November 8, 1979, S. 110-115.
/150/ Cragon, H. G.; Watson, W. J.: The TI Advanced Scientific Computer. Computer Vol. 22 No. 1 (January 1989), S. 55-64.
/151/ Curran, L.: Stellar's Graphic Machine Stresses Interactivity. Electronics Vol. 61 No. 6 (March 17, 1988), S. 97-99.
/152/ Dasgupta, S.; Agiiero, U.: On the Plausibility of Architectural Designs. In: /1/, S. 177-193.
/153/ Davidson, J. W.; Vaughan, R. A.: The Effect of Instruction Set Complexity on Program Size and Memory Performance. In: /34/, S. 60-64.
/154/ Davis, C. et al.: Gate array embodies System/370 processor. Electronics, October 9, 1980, S. 140-143.
/155/ Dennis, J. B.: The Varieties of Data Flow Computers. In: /20/, S. 210-219.
/156/ DeVane, C. J.; Lidington, G.: Boost processor performance with two- level cache memeory. Electronic Design Vol. 36 No. 13 (June 1988), S. 97-101.
/157/ Diede, T. et al.: The Titan Graphics Supercomputer Architecture. Computer Vol. 21 No. 9 (September 1988), S. 13-30.
/158/ Ditzel, D. R. et al.: Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor. In: /34/, S. 158-163.
/159/ Dongarra, J. J.: Some Linpack Timings on the CRAY-1. In: /20/, S.363-380
/160/ -: Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment. Computer Architecture News Vol. 16 No. 1 (March 1988), S. 47-64.
/161/ -: The LINPACK Benchmark: An Explanation. In: /16/, S. 456-474.
/162/ Eckelmann, P.: Transputer der 2. Generation. 1. Teil: Architektur und Merkmale. Elektronik, Heft 18, 1987, S. 61-70.
/163/ Farmwald, P. M.: The S-1 Mark II A Supercomputer. In: /17/, S. 145-155.
/164/ Fehmel, J.: Automatisierter Entwurf von kombinatorischen Schaltungen mit dekompositorischen Methoden. Dissertation (A), TU Karl-Marx-Stadt 1988.

/165/ Fiasconaro, J. G.: Instruction Set for a Single-Chip 32- Bit Processor. Hewlett- Packard Journal Vol. 34 No. 8 (August 1983), S. 9, 1Ø.
/166/ Flanders, P. M. et al.: Efficient High Speed Computing with the Distributed Array Processor. In: /19/, S. 113-128.
/167/ Flynn, M. J.: Some Computer Organizations and Their Effectiveness. IEEE TC-21 (September 1972), S. 948-96Ø (in: /2Ø/, S. 11-23.
/168/ -: The Interpretative Interface: Ressources and Program Representation in Computer Organization. In: /19/, S. 41-69.
/169/ -: Customized Microcomputers. In: /11/, S. 182-222.
Flynn, M. J.; Mitchell, C. L.; Mulder, J. M.: And Now a Case for More Complex Instruction Sets. Computer Vol. 2Ø No. 9 (September 1987), S. 71-83.
/17Ø/ Fox, G. C.: Questions and Unexpected Answers in Concurrent Computations. In: /8/, S. 97-121
/171/ Fritsch, G.: Numerical Simulation of Physical Phenomena by Parallel Computing. In: /3/, S. 4Ø-57.
/172/ Furht, B.; Milutinovich, V.: A Survey of Microprocessor Architectures for Memory Management. Computer Vol. 2Ø No. 3 (March 1987), S. 48-67.
/173/ Gehne, R.: Parallele Algorithmen in der Mathematischen Optimierung. Informatik Informationen Reporte 6, 1988. IIR Berlin, 1988.
/174/ Geyer, J.: 32-Bit-Mikrocomputer besitzt neuartige Architektur. Elektronik, Heft 5, 1981, S. 59-66.
/175/ Gibbons, J. F.; Shott, J. D.: Integrated Circuit Physics and Technology. In: /11/, S. 9-64.
/176/ Giloi, W. K.: Die Entwicklung der Rechnerarchitektur von der von Neumann- Maschine bis zu Rechnern der "fünften Generation". Elektronische Rechenanlagen 26. Jahrgang (1984), Heft 2, S. 55-7Ø.
/177/ -: Principles of Computer Architecture. In: /13/, S. 9-3Ø.
/178/ Gimarc, C. E.; Milutinovich, V. M: A Survey of RISC Processors and Computers of the Mid- 198Øs. Computer Vol. 2Ø No. 9 (September 1987), S. 59-69.
/179/ Goering, R.: New workstations magnify electronic CAE/CAD power. Computer Design Vol. 27 No. 6 (March 15, 1988), S. 36-39.
/18Ø/ Gottlieb, A.: An Overview of the NYU Ultracomputer Project. In: /9/, S. 25-95.
/181/ Griffin, G. G.: The Ultimate Ultimate RISC? Computer Architecture News Vol 16 No. 5 (December 1988), S. 26-31.
/182/ Gurd, J.; Kirkham, C.; Böhm, W.: The Manchester Dataflow Computing System. In: /9/, S. 177-219.
/183/ Hall, D. G.: Survey of Silicon- based Integrated Optics. Computer Vol. 21 No. 12 (December 1988), S. 25-32.
/184/ Händler, W.: Simplicity and Flexibility in Concurrent Computer Architecture. In: /17/, S. 69-87.

/185/ Harmon, W. J.; Miller, W. K.: Bipolarer 16-Bit-Prozessor übernimmt anspruchsvolle Controller- Aufgaben. Elektronik, Heft 8, 1980, S. 81-86.
/186/ Hayes, J. R. et al.: An Architecture for the Direct Execution of the Forth Programming Language. In: /34/, S. 42-48.
/187/ Herzog, U.: Performance Modelling and Evaluation For Concurrent Computer Architectures. In: /17/, S. 177-189.
/188/ Hext, J. B.: Data Abstraction Facilities. Technical Report No. 120 Basser Department of Computer Science, The University of Sydney, 1977.
/189/ Higbie, L.: A Vector Processing Tutorial. datamation Vol. 29 No. 8 (August 1983), S. 180-203.
/190/ Higuchi, T. et al.: The IX Supercomputer for Knowledge- Based Systems. In: /31/, S. 1041-1049.
/191/ Hilbert, D.: Die logischen Grundlagen der Mathematik. Mathematische Annalen 88 (1923), S. 151-165. Gekürzter Nachdruck in: /5/, S. 349, 350.
/192/ Hintz, R. G.; Tate, D. P.: STAR-100 Processor Design. In: /20/, S.36-40.
/193/ Hirakawa, M. et al.: A Relational Database Machine Organization for Parallel Pipelined Query Execution. In: /31/, S. 1233-1243.
/194/ Hoare, C. A. R.: An axiomatic approach to computer programming. Communications of the ACM Vol. 12 (1969), S. 576-580.
/195/ -: Proof of correctness of data representations. Acta Informatica Vol. 1 (1972), S. 271-281.
/196/ Hockney, R. W.: Classification and Evaluation of Parallel Computer Systems. In: /8/, S. 13-25.
/197/ -: Performance of Parallel Computers. In: /17/, S. 159-175.
/198/ Hon, R. W.; Reddy, D. R.: The Effect of Computer Architecture on Algorithm Decomposition and Performance. In: /19/, S. 411 ff.
/199/ Jegou, Y.: Access Patterns: A Useful Concept in Vector Programming. In: /16/, S. 377-391.
/200/ Jones, T.: Engineering design of the Convex C2. Computer Vol. 22 No. 1 (January 1989), S. 36-44.
/201/ Jones, D. W.: The Ultimate RISC. Computer Architecture News Vol. 16 No. 3 (June 1988), S. 48-55.
/202/ -: A Minimal CISC. Computer Architecture News Vol. 16 No. 3 (June 1988), S. 56-63.
/203/ Jones, A. K.; Gehringer, E. F. (eds.): The CM Multiprocessor Project: A Research Overview. Computer Science Department Carnegie- Mellon University CMU-CS-80-131, Pittsburgh, PA. 1980.
/204/ Jouppi, N. P.: Superscalar vs. Superpipelined Machines. Computer Architecture News Vo. 16 No. 5 (November 1988), S. 71-80.
/205/ Kahn, K. C.: Object- oriented languages tackle massive programming headaches. Electronics, Nov. 17, 1982, S. 141-145.
/206/ Kashiwagi, H.: Japanese Super-Speed Computer Project. In: /17/, S. 117-125.

/2Ø7/ Kascic, M. J.: A Performance Survey of the CYBER 2Ø5. In: /17/., S. 191-2Ø9.
/2Ø8/ Kober, R.; Kuznia, C.: SMS- A Multiprocessor Architecture for High Speed Numerical Calculations. In: /2Ø/, S. 236-242.
/2Ø9/ Koller, J.: Inhaltsbezogener Speicherzugriff durch Assoziativprozessor. Elektronik, Heft 8, 1983, S. 45- 48.
/21Ø/ Krause, F. L.; Spur, G. u. a.: Systemarchitektur für Geometrieverarbeitung. Bericht aus dem Sonderforschungsbereich 2Ø3 (Teilprojekt C2). Fraunhofer-Institut/TU Berlin(W), 1988.
/211/ Kriz, J.; Sugaya, H.: Logic Programming. In: /13/, S. 3Ø5-34Ø.
/212/ Kuck, D. J. et al.: Parallel Supercomputing Today and the Cedar Approach. In: /9/, S. 1-23.
/213/ Kuck, D. J.; Samek, A. H.: A Supercomputing Performance Evaluation Plan. In: /16/, S. 1-17.
/214/ Kuck, D. J.; Muraoka, Y.; Chen, S. C.: On the Number of Operations Simultaneously Executable in Fortran-like Programs. IEEE TC-12, 1972, S. 1293-131Ø.
/215/ Kuck, D. J. et al.: Measurements of Parallelism in Ordinary FORTRAN Programs. Computer Vol. 7 No. 1 (January 1974), S. 37-46 (in: /2Ø/, S. 346-355).
/216/ Lawrie, D. H.: Access and Alignment of Data in an Array Processor. IEEE TC-24 (December 1975), S. 1145-1155 (in: /2Ø/, S. 99-1Ø9).
/217/ Lee, R. B.: Precision Architecture. Computer Vol. 22 No. 1 (January 1989), S. 78-91.
/218/ Leonard, M.: RISC microprocessors: many architectures thrive. Electronic Design Vol 36 No. 17 (July 28, 1988), S. 49, 5Ø.
/219/ Lieberman, D.: Moderately parallel supercomputer avoids vector hardware. Computer Design Vol. 27 No. 1Ø (May 15, 1988), S. 29, 3Ø.
/22Ø/ Lincoln, N. R.: It's really not as much fun building a supercomputer as it is simply inventing one. In: /19/, S. 3-11.
/221/ -: Supercomputers = Collossal Computations + Enormous Expectations ; Renowed Risk. Computer Vol. 16 No. 5 (May 1983), S. 38-47.
/222/ Lineback, J. R.: TI's 64-Bit Processor Hooks into the SPARC RISC Chip. Electronics Vol. 61 No. 6 (March 17, 1988), S. 8Ø, 81.
/223/ -: Weitek's Pipelined Chip Offers Board- Size Savings. Electronics Vol. 61 No. 6 (March 17, 1988), S. 84-87.
/224/ Lipovski, G. J.; Doty, K. L.: Entwicklungen und Richtungen auf dem Gebiet der Rechnerarchitektur. Computer 11 (1978), S. 54-67 (dt. Übers. Kombinat Robotron).
/225/ Liskov, B. et al.: CLU Reference Manual. MIT Laboratory for Computer Science MIT/LCS/TR-225, Cambridge, Mass. 1979 bzw. Springer, 1981 (LNCS 114).

/226/ Lukes, J. A.: HP Precision Architecture Performance Analysis. HP Journal, August 1986, S. 3Ø-39.
/227/ Lutz, H.: Multibus- Coprozessor nach Maß. Elektronik, Heft 17, 1984, S. 64-68.
/228/ Magenheimer, D. J. et al.: Integer Multiplication and Division on the HP Precision Architecture. In: /34/, S. 9Ø-99.
/229/ Mahon, M. J. et al.: Hewlett-Packard Precision Architecture: The Processor. HP Journal, August 1986, S. 4-22.
/23Ø/ Manuel, T.: "Silverlake" Looks Good: Can it End IBMs Midrange Woes? Electronics Vol.61 No. 13 (July 1988), S. 37, 42.
/231/ -: Supermini Killer: First of New RISC Breed is Here from MIPS Computer. Electronics Vol. 61 No. 13 (July 1988), S. 12Ø, 121.
/232/ Massalin, H.: Superoptimizer: A Look at the Smallest Program. In: /34/, S. 122-126.
/233/ Matick, R. E.; Ling, D. T.: Architecture implications in the design of microprocessors. IBM Systems Journal Vol. 23 No. 3 (1984), S. 264-28Ø.
/234/ Matthes, W.: RTX.ØØ4 Reference Manual. Betriebsinterne Entwicklungsdokumentation VEB Robotron FG Geräte, Karl- Marx- Stadt 1981.
/235/ -: Multimikrorechnersysteme. radio fernsehen elektronik. Fortsetzungsreihe von Heft 4/1984 bis Heft 12/1985.
/236/ -: Diagnostik und Wartung an Multimikrorechnersystemen. radio fernsehen elektronik, Heft 8, 1986, S. 518-521 und Heft 9, 1986, S. 566, 567.
/237/ -: Echtzeit- Betriebssysteme für Multimikrorechnersysteme. radio fernsehen elektronik, Heft 11, 1986, S. 7ØØ-7Ø2 und Heft 12, 1986, S. 756-758.
/238/ -: WP 154 244: Speicheranordnung mit Fehlererkennungs- und Diagnoseeigenschaften, vorzugsweise für Mikrorechner.
/239/ -: WP 236 611: Anordnung zum Durchmustern binärer Information, vorzugsweise von Ternärvektorlisten.
/24Ø/ -: WP 242 299: Spezialprozessoranordnung zur Verarbeitung von Ternärvektorlisten.
/241/ -: WP 252 9Ø9: Aktive Speicheranordnung zur Orthogonalisierung von Ternärvektorlisten.
/242/ -: System.ØØ8 - Vorläufige Architekturbeschreibung. 1986 (unveröffentlicht).
/243/ -: Spezielle Hardware zur Verarbeitung von Ternärvektorlisten. Dissertation (A), Technische Universität Karl- Marx- Stadt 1987.
/244/ -: System.Ø14 Technische Anforderungen und Einführende Erläuterungen. 1987 (unveröffentlicht).
/245/ -: Zur künftigen Entwicklung von Hochleistungshardware im Institut für Informatik und Rechentechnik - eine Denkschrift. Institutsinternes Arbeitsmaterial, Berlin 1989 (unveröffentlicht).
/246/ -: Datenzugriffsprinzipen in objektorientierten Rechnerarchitekturen. Institut für Informatik und Rechentechnik, Berlin 1989 (Preprint 89.Ø1).

/247/ -: Ergänzungsschaltungen für Mikroprozessoren. In Vorbereitung für IIR Informatik Informationen Reporte.
/248/ -: Technische Lösungsansätze für Rechnerarchitekturen auf Grundlage Vergegenständlichter Abstraktionen. Internes Arbeitsmaterial des IIR, Berlin (in Vorbereitung).
/249/ Matthes, W.; Pietschmann, H.: Advanced Real Time Executive .Ø11 (ARTX.Ø11) Application Reference Manual. Betriebsinterne Entwicklungsdokumentation VEB Robotron FG Geräte, Karl- Marx- Stadt 1986.
/25Ø/ Matthes, W.; Steinbach, B.: WP 258 3ØØ. Elektronische binäre Steuereinrichtung, vorzugsweise zum Einsatz in Multimikrorechnersystemen.
/251/ Matthes, W.; Steinbach, B.; Wolf, M.: WP- Anmeldung: Verfahren und Anordnung zur technischen Implementierung von Steuerungsabläufen, die durch eine Automatenbeschreibung gegeben sind.
/252/ McKeever, B. T.: The associative memory structure. FJCC 1965 Proc. Vol. 27 Part 1, S. 371-388.
/253/ McWilliams, G.: RISC Gains, but its Role in Mainstream is still small. Datamation Vol 34 No. 11 (June 1, 1988), S.24-28.
/254/ Miller, Ch.: CISC, RISC, WISC: What's in a name ? IEEE Micro Vol. 8 No. 1 (February 1988), S. 6, 7.
/255/ Milutinovic, V. et al.: A GaAs- Based Microprocessor Architecture for Real- Time Applications. IEEE TC, June 1987, S. 714-727.
/256/ Miura, K.; Uchida, K.: FACOM Vector Processor VP-1ØØ/VP-2ØØ. In: /17/, . 127-137.
/257/ Moreland, J.: Workstations and mainframes must pool their resources for enginerering applications. Electronic Design Vol. 33 No. 16 (July 11, 1985), S. 68.
/258/ Moss, J. E. B.: Abstract Data Types in Stack Based Languages. MIT/LCS/TR-19Ø, Cambridge, Mass. 1978.
/259/ Moto-Oka, T.: Japanese Project on Fifth Generation Computer Systems. In: /17/, S. 99-115
/26Ø/ Mucha, J.: The Complexity Problem in Design, Verification and Testing. In: /25/, S. 2Ø5-227.
/261/ Müller- Wichards, D.: An Algebraic Approach to Performance Analysis. In: /8/, S. 159-185.
/262/ Muraoka, Y.; Marushima, T.: Major Research Activities in Parallel Processing in Japan. In: /31/, S. 836-853.
/263/ Myers, W.: Ada: First users- pleased; prospective users- still hesitant. Computer Vol. 2Ø No. 3 (March 1987), S. 68-73.
/264/ Neves, K. W.: Vectorization of Scientific Software. In: /17/, S. 277-291.
/265/ Nicolau, A.: The Cornell Parallel Supercomputing Effort. In: /9/, S. 221-253.
/266/ Nicolau, A.; Fisher, J.: Measuring the Parallelism Available for Very Long Instruction Word Architectures. IEEE TC November 1984.

/267/ Nicond, J.-D.: Video RAMs: Structure and Applications. IEEE Micro Vol. 8 No. 1 (February 1988), S. 8-27.
/268/ Nowak, L.: SAMP: A General Purpose Processor Based on a Self- Timed VLIW Structure. Computer Architecture News Vol. 15 No. 4 (September 1987), S. 32-39.
/269/ Ohr, S.: 1985 Technology Forecast: RISC Machines. Electronic **Design** Vol. 33 No. 1 (January 1Ø, 1985), S. 174-19Ø.
/27Ø/ Oppe, T. C.: Kincaid, D. R.: Parallel LU-Factorization Algorithms for Dense Matrices. In: /16/, S. 567-594.
/271/ Papadopoulos, G. M.: Implementation of a General Purpose Dataflow Multiprocessor. Ph.D. Thesis. Department of Electrical Engineering and Computer Science, MIT, Cambridge, Mass. 1897
/272/ Patt, Y. N.: Real Machines: Design Choices/Engineering Trade- Offs. Computer Vol. 22 No. 1 (January 1989), S. 8-1Ø.
/273/ Patterson, D. A.; Sequin, C. H.: RISC-I: A Reduced Instruction Set VLSI Computer. Proceedings of the Eighth Annual Symposium on Computer Architecture, May, 1981.
/274/ Petermann, U.: Algorithmische Logik. In: /18/, S. 293-326.
/275/ Pfister, G. F. et al.: An Introduction to the IBM Research Parallel Processor Prototype (RP3). In: /9/, S. 123-139.
/276/ Phillips, B. W.: Graphics engines tackle 3D image-processing jobs. Electronic Design Vol. 36 No. 17 (July 28, 1988), S. 1ØØ-1Ø6.
/277/ -: Adder algorithm speeds multiplication. Electronic Design Vol. 36 No. 19 (August 25, 1988), S. 32.
/278/ -: New Nonsaturating Logic Outruns ECL and Dissipates less Power. Electronic Design Vol.36 No. 2Ø (September 8, 1988), S. 21.
/279/ van der Poel, W. L.: A simple electronic computer. Applied Sciences Research Bullettin 2(1952), S. 367.
/28Ø/ Pösch, H.; Fromme, Th.: Programmorganisation bei kleinen Rechenautomaten mit innerem Programm. ZAMM 34 (1954), S. 3Ø7-3Ø8.
/281/ Posthoff, Ch.; Steinbach, B.: Binäre Gleichungen - Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx-Stadt, Heft 1, 1979.
/282/ -: Binäre dynamische Systeme - Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx- Stadt, Heft 8, 1979.
/283/ Radin, G.: The 8Ø1 Minicomputer. IBM J. Res. Dev. Vol. 27 No. 3 (May 1983), S. 237-246.
/284/ Ran, B. R. et al.: The Cydra 5 Departmental Supercomputer. Computer Vol. 22 No. 1 (January 1989), S. 12-34.

/285/ Rattner, W.; Lattin, W. W.: Ada determines architecture of 32-bit-microcomputer. Electronics Vol. 54 No. 4 (February 24, 1981), S. 119-126.
/286/ Reddaway, S. F.: Distributed Array Processor, Architecture and Performance. In: /17/, S. 89-97.
/287/ Rosenstein, L.; Doyle, K.; Wallace, S.: Object-Oriented Programming for Macintosh Applivcations. In: /31/, S. 31-35.
/288/ Rudolph, J. A.: A production implementation of an associative processor: STARAN. AFIPS Conf. Proc. FJCC 4, S. 229-241, 1972.
/289/ Runyon, S.: AMD's Floating-Point Chip Goes Beyond IEEE Specs. Electronics Vol. 61 No. 6 (March 17, 1988), S. 78, 79.
/29Ø/ Russel: The Cray 1 Computer System. CACM 21, 1 (January 1978), S. 63-72 (in: /2Ø/, S. 26-35).
/291/ Schwartz, R. L.; Melliar- Smith, P. M.: The Suitability of Ada for Artifical Intelligence Applications. SRI International, Menlo Park, Ca. 198Ø.
/292/ Schwermer, H. R.: Streamlined architecture achieves software compatibility. Electronics, November 8, 1979, S. 119, 12Ø.
/293/ Siegel, H. J. et al.: Communication Techniques in Parallel Processing. In: /8/, S. 35-6Ø.
/294/ Sluss, J. J. et al.: An Introduction to Integrated Optics for Computing. Computer Vol. 2Ø No. 12 (December 1988), S. 9-23.
/295/ Smith, B. J.: Shared Memory, Vectors, Message Passing, and Scalability. In: /8/, S. 29-34.
/296/ Smith, D. C. P.; Smith, J. M.: Relational Database Machines. Computer Vol. 12 No. 3 (March 1979), S. 28-37 (in: /2Ø/, S. 282-292).
/297/ Steinbach, B.: Theorie, Algorithmen und Programme für den rechnergestützten logischen Entwurf digitaler Systeme. Dissertation (B), Technische Hochschule Karl- Marx- Stadt, 1984.
/298/ Steinbach, B. u. a.: Rechentechnische Erfahrungen mit Booleschen Gleichungen. In: /6/, S. 8Ø-1Ø2.
/299/ Steinbach, B.; Le Trung Quoc: Tools für den Logikentwurf. Wissenschaftliche Schriftenreihe der Technischen Universität Karl- Marx- Stadt, Heft 9, 1988.
/3ØØ/ Tarski, A.: Der Wahrheitsbegriff in den formalisierten Sprachen. Lemberg 1935. Nachdruck in: /5/, S. 447-559.
/3Ø1/ Terry, J. M.: Flow- Control Machines: The Structured Execution Architecture (SXA). Computer Architecture News Vol. 15 No. 4 (September 1987), S. 58-69.
/3Ø2/ Thakkar, S. et al.: The Balance Multiprocessor System. IEEE Micro Vol. 8 No. 1 (February 1988), S. 57-69.
/3Ø3/ Tucker, L. W.; Robertson, G. G.: Architecture and Application of the Connection Machine. Computer, Vol. 21 No. 8 (August 1988), S. 26-38.
/3Ø4/ Vegdahl, S. R.: A Survey of Proposed Architectures for the Execution of Functional languages. IEEE TC-33 Vol. 12 (December 1984), S. 1Ø5Ø-1Ø74.

/3Ø5/ Vlakos, H.; Milutinovic, V.: GaAs Microprocessors and Digital Systems. An Overview of R&D Efforts. IEEE Micro Vol. 8 No. 1 (February 1988), S. 28-56.
/3Ø6/ Vlontzos, J. A.; Kung, S. Y.: A Wavefront Array Processor Using Data Flow Processing Elements. In: /16/, S. 744-767.
/3Ø7/ Völksen, G.; Wehrum, P.: Transition to Ada for Super Computers. In: /29/, S. 79-89.
/3Ø8/ Wacker, H. M.: Introduction to the Seminar. (Einführung zu /8/; dort S. 3-12).
/3Ø9/ Wakefield, S. P.; Flynn, M. J.: Reducing execution parameters through correspondence in computer architecture. IBM J. Res. Dev. Vol. 31 No. 4 (July 1987), S. 42Ø-434.
/31Ø/ Waller, L.: Makers soup up stand alones. Electronics Week, March 11, 1985, S. 2Ø-25.
/311/ Waltz, D. et al.: Very large database applications of the Connection Machine system. In: /32/, S. 159-166.
/312/ Wasserman, A. I.: Introduction to Sequential Control Structures. In: /3Ø/, S. 94-1Ø1.
/313/ Wegner, P.: Programming Languages - The First 25 Years. In: /3Ø/, S. 1Ø-28.
/314/ Weicher, R. P.: Dhrystone: A Synthetic Systems Programming Benchmark. Communications of the ACM Vol. 27 No. 1Ø (October 1984), S. 1Ø13-1Ø3Ø.
/315/ West, L. C.: Picosecond Integrated Optical Logic. Computer Vol. 2Ø No. 12 (December 1988), S. 34-47.
/316/ Widdoes, L. C.; Correll,, S.: The S 1 Project: Developing High- Performance Digital Computers. In: /2Ø/, S. 136-145.
/317/ Wiesemann, R.: Netzwerk oder Multiuser- System. mini micro magazin Nr. 7, 1988, S. 122, 123.
/318/ Williams, T.: Graphics supercomputer tackle real time visualization. Computer Design Vol. 27 No. 6 (March 15, 1988).
/319/ Wilson, K. G.: Science, Industry and the New Japanese Challenge. In: /17/, S. 9-55.
/32Ø/ Wilson, R.: New CPUs spearhead Intel attack on embedded computing. Computer Design Vol. 27 No. 8 (April 15, 1988), S. 22-28.
/321/ -: Motorola unveils new RISC microprocessor flagship. Computer Design Vol. 27 No. 9 (May 1, 1988), S. 21-32.
/322/ -: RISC architectures take on heavy weight applications. Computer Design Vol. 27 No. 1Ø (May 15, 1988), S. 59-79.
/323/ -: Application tuning: a key to RISC system development. Computer Design Vol. 27 No. 14 (August 1, 1988), S. 21, 22.
/324/ -: British microprocessors: a lesson in innovations. Computer Design Vol. 27 No. 2Ø (November 1, 1988), S. 32-42.
/325/ -: Intel debuts its super-performance microprocessor. Computer Design Vol. 29 No. 5 (March 1, 1989), S. 9.

/326/ -: Applications determine the Choice of RISC or CISC. Computer Design Vol. 29 No. 5 (March 1, 1989), S. 58-73.
/327/ -: 8Ø86Ø CPU positions Intel to take on minisuper-computers. Computer Design Vol. 28 No. 7 (April 1, 1989), S. 2Ø-23.
/328/ -: Intel 8Ø486 carries complex instruction set to RISC speeds. Computer Design Vol. 28 No. 9 (May 1, 1989), S. 18-2Ø.
/329/ -: 68Ø4Ø moves toward RISC camp with redesigned pipelines, caches. Computer Design Vol. 28 No. 9 (May 1, 1989), S. 22.
/33Ø/ Wirth, N.: Hardware Architectures for Programming Languages and Programming Languages for Hardware Architectures. In: /34/, S. 2-8.
/331/ Wood, L.: Does the mainframe have a Place in a PC World? mini micro magazin Nr. 7, 1988, S. 116, 117.
/332/ Wright, R. E.: Documenting a Computer Architecture. IBM J. Res. Dev. Vol. 27 No. 3 (May 1983), S. 257-264.
/333/ Wulf, Wm. A.: The WM Computer Architecture. Computer Architecture News Vol. 15 No. 4 (September 1987), S. 7Ø-84.
/334/ Young, J. L.: The Software Foundry: Almost to good to be true. Electronics Vol. 61 No. 2 (January 21, 1988), S. 47-51.
/335/ Zemanek, H.: Abstrakte Objekte. Elektronische Rechenanlagen 1Ø (1968), Heft 5, S. 2Ø8-211.