W. Matthes

S Y S T E M .0 0 8

- Vorläufige Architekturbeschreibung -

Ausgabe: Nr. 1 vom 11. 3. 1986

Inhalt

1. Einführung

Das System .008 ist eine Spezialprozessor- Architektur für die effiziente rechentechnische Behandlung von Problemen, die sich auf den Umgang mit BOOLEschen Gleichungen, BOOLEschen Differentialgleichungen usw. zurückführen lassen.
Wesentlich ist, daß zur internen Repräsentation der Informationsstrukturen vorzugsweise Ternärvektorlisten (TVL) verwendet werden. Hinsichtlich der elementaren Operationen, die damit möglich sind, besteht eine weitgehende Anlehnung an das in /4/, /7/, /16/, /17/ ausführlich beschriebene Algorithmen- und Programmsystem.
Damit sollen komplexe Anwendungsprobleme rechenpraktisch beherrschbar werden, namentlich solche, die durch eine NP- vollständige oder exponentielle Komplexität (im Sinne der Komplexitätstheorie, vgl. /4/) gekennzeichnet sind.
Einsatzmöglichkeiten bestehen z. B. auf folgenden Gebieten (s. a. /7/, /21/):

- Entwurf und Optimierung digitaler Schaltungen
- Berechnung von Testbelegungen für digitale Schaltungen
- Behandlung von Graphen
- Layout- Rückerkennung integrierter Schaltkreise
- Entwurf programmierbarer Steuerungen.

Die Spezialprozessoren des System .008 sind vorzugsweise Hardware- Anordnungen aus spezifischen Speicher-, Verarbeitungs-, Adressierungs- und Steuerschaltungen ("Algorithmen- Realisierungs- Maschine" ARM), die mit frei programmierbaren Einrichtungen ("Architektur- Implementierungs- Maschine" AIM, z.B. durch einen Mikrorechner oder ein mikroprogrammierbares Steuerwerk realisiert) gekoppelt sind.
Eine typische Hardware- Struktur ist in /12/ umfassend beschrieben.
Einen Überblick über charakteristische interne Operationsprinzipien gibt weiterhin der Abschnitt 5.

Es ist jedoch auch möglich, die .008- Architektur durch Software (bzw. durch eine Kombination konventioneller Software mit mikroprogrammierten Abläufen) auf herkömmlichen Rechenanlagen zu implementieren. In diesem Sinne repräsentiert die .008- Architektur das "Laufzeitsystem" für die Anwendungsprogramme, die die gegebenen Datenstrukturen und Algorithmen nutzen. Sie kann somit als Äquivalent zu bekannten Architekturdefinitionen angesehen werden, deren Hauptzweck die "Portabilität" ist und die für das Formulieren von Anwendungsprogrammen im eigentlichen Sinne nicht vorgesehen sind (Beispiel: P- Code für die Implementierung der Programmiersprache Pascal, s. /22/). Über die bekannten Beispiele hinausgehend muß die Architektur nicht nur die Portabilität zwischen verschiedenen Rechnertypen gewährleisten, sondern auch zwischen verschiedenen Ausführungsformen spezieller Hardware sowie zwischen Software- und Hardware- Realisierungen (so muß es z. B. möglich sein, einen Ablauf, der als Unterprogramm existiert, durch eine spezielle Schaltungsanordnung ausführen zu lassen, wobei lediglich der Programmaufruf durch eine Hardware- Aktivierung ersetzt wird, die Konventionen der Parameter- Übergabe usw. aber unverändert bleiben).
In dieser Schrift wird ein erster Ansatz zu einer solchen Architekturdefinition vorgestellt. Als Basis für die weitere Diskussion werden die grundlegenden Informationsstrukturen und Datenzugriffsprinzipien, die Organisation des Befehlsablaufes und wesentliche Operationen beschrieben. Der vorläufige Charakter schließt Vollständigkeit von vornherein aus. Eine vollständige, für die praktische Anwendung verwendbare Architekturdefinition erfordert sowohl weitere Vorarbeiten, um alle praxisrelevanten Aspekte zu berücksichtigen, als auch eine andere Art und Weise der Darstellung (vgl. /23/).

In dieser Schrift wird eine Architekturdefinition formal als 5- Tupel betrachtet:

$$A = (D, I, S, A, C)$$

Dabei bedeutet
D: Menge der Datenstrukturen
I: Menge der Befehlsstrukturen ("Befehlsliste")
S: Prinzipien der Ablauforganisation
A: Prinzipien des Datenzugriffs
C: Prinzipien der Externkommunikation ("Ein- Ausgabe")

Die Mengen D und I bestehen jeweils wieder aus zwei Teilmengen, wobei die erste die Eigennamen der Datenstrukturen bzw. Befehle umfaßt und die zweite die jeweiligen Beschreibungen.
Die Beschreibung selbst erfolgt überwiegend verbal; für Deklarationen wird eine formalisierte Schreibweise benutzt, die stark an die Programmiersprache Ada (/10/) angelehnt ist. Die benutzten bzw. neu eingeführten Sprachkonstrukte sind zusammengefaßt in Abschnitt 5 dargestellt; ansonsten werden sie jeweils dort erklärt, wo dies notwendig ist.
Hinweis:
Es handelt sich nicht darum, eine bestimmte Programmiersprache zu benutzen, sondern darum, die notwendigen Deklarationen in einer exakten und verständlichen Form darzustellen. Die praktische Nutzung der Architektur ist nicht an den Einsatz der hier verwendeten Sprachkonstrukte (etwa im Sinne einer "TVL- Spezialsprache") gebunden.
In den folgenden Darstellungen wird eine gewisse Kenntnis der in /16/, /17/ beschriebenen Algorithmen vorausgesetzt (in der Literatur bereits beschriebene Algorithmen werden hier nicht näher erläutert).

2. Allgemeine Prinzipien

2.1. Grundlagen

Die Architektur des Systems .008 ist objektorientiert (im Sinne von /9/, /11/, /14/, /25/).
Jede Informationsstruktur wird als ein Objekt betrachtet. Jedes Objekt ist eine geordnete Menge seiner Komponenten (vgl. /25/), so daß Begriffe der Mengenlehre (s. z. B. /8/) zur Charakterisierung von Objekten verwendet werden können.
Die Anzahl der Elemente einer Menge (Komponenten eines Objekts) wird durch eine Kardinalzahl ausgedrückt. Solche Zahlen haben für eine Menge aus maximal n Elementen Werte zwischen 0 und n (Wert 0: "leere Menge", kein Element vorhanden).
Ein bestimmtes Element der Menge wird durch seine Ordinalzahl identifiziert (die Ordinalzahl 1 bezeichnet das erste Element, die Ordinalzahl n das letzte).
Die maximale Anzahl der Komponenten, aus denen ein Objekt zusammengesetzt sein kann, wird stets endlich sein. Es erweist sich jedoch als zweckmäßig, von vornherein eine Unterteilung in finite und transfinite Objekte einzuführen:
Finite Objekte haben eine begrenzte Maximalzahl von Komponenten. Diese ist jeweils durch eine Architekturdefinition gegeben.
Transfinite Objekte können prinzipiell beliebig viele Komponenten haben. Die Begrenzung ist lediglich durch technische Beschränkungen (der jeweiligen Implementierung) gegeben.
Hinweise:

1. Die Kardinalität finiter Objekte ist auf maximal $2^{12} = 4096$ beschränkt.
2. Die Obergrenze der Kardinalität transfiniter Objekte ist auf 2^{32} beschränkt. Jede Implementierung kann (aus technisch- ökonomischen Gründen) einen kleineren Wert vorsehen; dieser Wert ist programmseitig abfragbar.

3. Kardinal- und Ordinalzahlen werden durch positive ganze Binärzahlen repräsentiert.
4. Der Wertebereich einer Kardinalzahl für ein Objekt aus maximal k Komponenten reicht von 0 bis k.
5. Der Wertebereich einer Ordinalzahl für ein Objekt aus maximal k Komponenten reicht von 1 bis k (erste....k-te).
6. Finite Kardinalzahlen sind 13 bit lang (Bereich 0...4096).
7. Finite Ordinalzahlen sind 12 bit lang; eine Ordinalzahl k wird intern durch den Wert k-1 repräsentiert (Wert 0 identifiziert die erste Komponente).
8. Transfinite Kardinalzahlen sind aus der Sicht der Architektur 32 bit lang (implementierungsspezifisch kürzere werden durch Voranstellen von Nullen auf diese Länge gebracht). Wertebereich: $0..2^{32}-1$.
9. Transfinite Ordinalzahlen sind aus der Sicht der Architektur 32 bit lang (erforderlichenfalls Voranstellen von Nullen). Wertebereich: $1..2^{32}$ (repräsentiert durch $0..2^{32}-1$).
10. Die vorstehenden Festlegungen sind so getroffen, daß eine 32- Bit- Binärarithmetik (2er- Komplement) zur Interpretation der Architektur hinreichend ist.

Objekte können auf zwei verschiedene Weisen aus ihren Komponenten gebildet werden:

1. Durch Aggregation der maschineninternen Repräsentationen der Komponenten. Solche Objekte heißen zusammengesetzte Objekte ("composite objects").
2. Die Komponenten sind ihrerseits Objekte. Ein derart gebildetes Objekt heißt Verbundobjekt ("compound object").

Jedes Objekt hat einen bestimmten Typ. Elementare Typen sind im Rahmen der Architektur definiert. Hat ein Objekt nur Komponenten identischen Typs, so wird es als homogenes Objekt bezeichnet, ansonsten als heterogenes.

Für die Deklarationen von Objekten werden folgende Sprachkonstrukte vorgesehen:

1. array bezeichnet ein homogenes Verbundobjekt
2. record bezeichnet ein heterogenes Verbundobjekt
3. area bezeichnet ein homogenes zusammengesetztes Objekt
4. record area bezeichnet ein heterogenes zusammengesetztes Objekt.

Elementare Objekte sind lediglich binäre (BOOLEsche) Variable und ganze Zahlen. Daraus werden alle anderen Objekte der 9008- Architektur definiert.

Die vorstehenden Definitionen sind in Bild 1 übersichtsweise zusammengefaßt (wesentliche Anregungen stammen aus /11/, /14/; eine ausführlichere Diskussion des Problemkreises ist in /13/ zu finden).

Einschränkungen bei der Bildung von Objekten:

1. Transfinite Objekte können nur aus finiten aufgebaut werden.
2. Die Deklaration zusammengesetzter Objekte ist nur im Rahmen der Architekturdefinition möglich. In Anwendungsprogrammen definierte Objektstrukturen werden grundsätzlich als Verbundobjekte implementiert.

Hinweise:

1. Die erste Einschränkung existiert im Sinne der Effizienz der Speicherverwaltung (transfinite Objekte, die transfinite Komponenten haben, können schnell so groß werden, daß sie in keinem technisch realisierbaren Speicher unterzubringen sind).
2. Die zweite Einschränkung existiert im Sinne der Effizienz der Zugriffe zu den Objekten und der Konsistenz der Zugriffsprinzipien (diese sollen für alle Objekttypen unterschiedslos Gültigkeit haben).
3. Zusammengesetzte Objekte werden maschinenspezifisch durch die jeweils dichteste Codierung repräsentiert (d.h. faktisch durch die direkte Aneinanderreihung der binären Codierung). Maschinenspezifische Besonder-

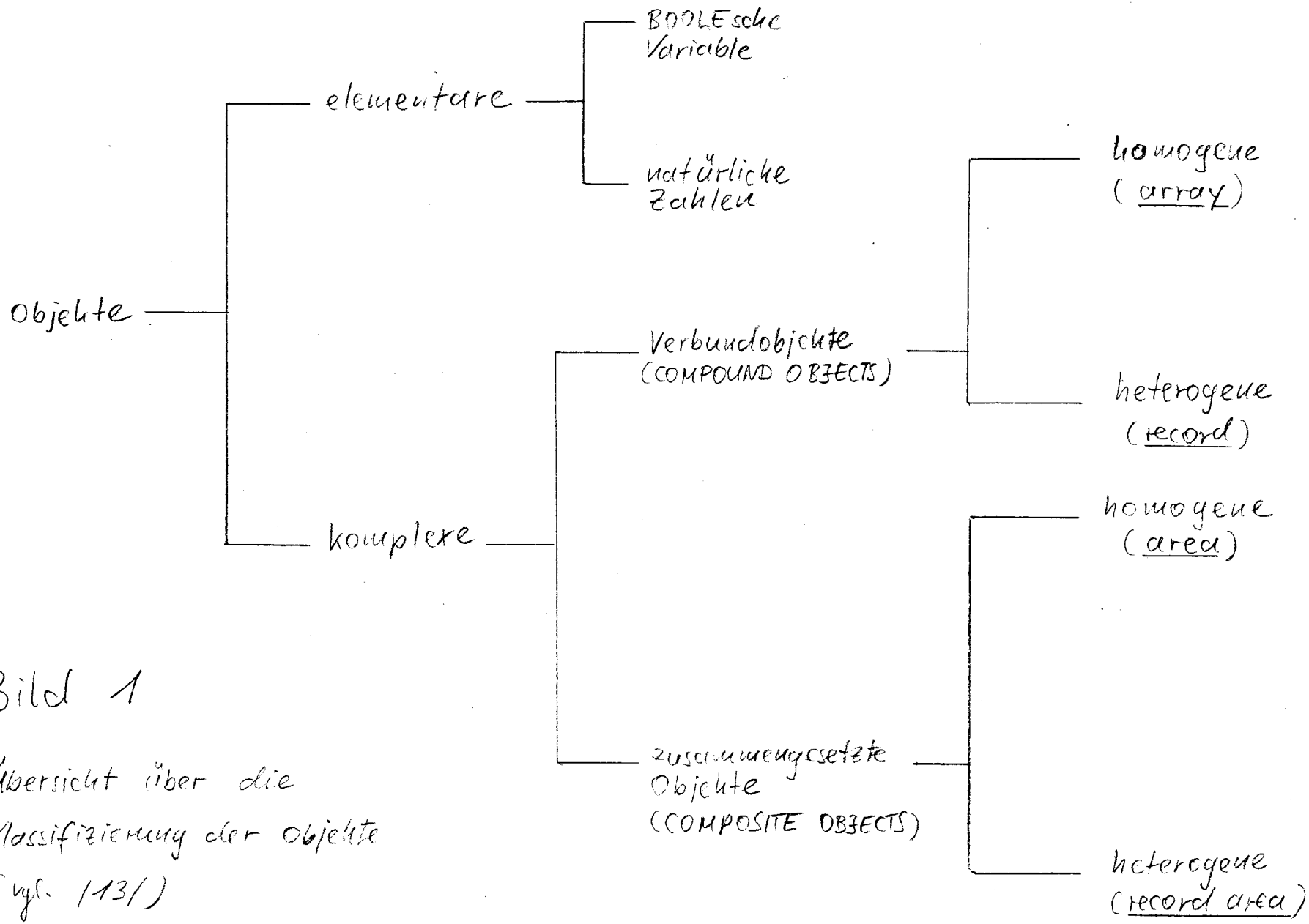


Bild 1

Übersicht über die Klassifizierung der Objekte (vgl. /13/)

heiten (z. B. die Bildung von Maschinenworten, die Unterbringung in bestimmten Speichermitteln usw. (vgl. /12/ sowie Abschnitt 6) sind für die Architektur transparent.

2.2. Ablauforganisation

Die Architektur des System .008 ist dadurch gekennzeichnet, daß es keine Adressierung im üblichen Sinne gibt, sowie dadurch, daß Datenzugriffe und Operationen völlig voneinander getrennt sind.
Zugriffe sind nur zu Objekten bzw. deren Komponenten möglich, und dazu werden keine Adressen, sondern ausschließlich Ordinalzahlen (Objektidentifier) benutzt.
Vor der Ausführung einer Operation (Operationsbefehl bzw. Programmaufruf) müssen alle benötigten Argumentobjekte bereitgestellt werden. Nach der Operationsausführung stehen die Resultatobjekte zur Verfügung (z. B. zur Verwendung in weiteren Operationen oder zum Abtransport). Grundsätzlich handelt es sich um eine Modifikation des seit längerem bekannten Prinzips der Stack- Maschine (s. /15/):
Vor Ausführung eines Operationsbefehls müssen alle erforderlichen Argumentwerte in korrekter Reihenfolge am Anfang des Stack bereitgestellt werden. Nach der Operation befindet sich das Resultat in der ersten Stackposition, und die Argumente wurden aus dem Stack entfernt.
Die wesentliche Modifikation beim System .008 besteht darin, daß der eigentliche Stack keine Objekte aufnimmt, sondern nur deren beschreibende (deskriptive) Information (d. h. üblicherweise die Objektidentifier).
Beispiel:
Es ist ein Ablauf der Form C := A <u>op</u> B auszuführen.
Dies geschieht in folgenden Schritten:

1. Der Objektidentifier des Objekts, das der Variablen A zugeordnet ist, wird in den Stack gebracht.

2. Der Objektidentifier des Objekts, das der Variablen B zugeordnet ist, wird in den Stack gebracht.
3. Die Operation op wird ausgeführt. Dabei wird ein Resultatobjekt aufgebaut. Dessen Identifier wird in den Stack gebracht, wobei zuvor die Identifier für A und B aus dem Stack entfernt wurden.
4. Der Variablen C wird dieses neue Objekt zugeordnet. Dessen Identifier wird aus dem Stack entfernt.

Wesentlich ist, daß die Zugriffe zu den eigentlichen Objekten (die sehr groß sein können) erst bei der Operationsausführung erfolgen (Pkt. 3.).
Die Speicherverwaltung ist für die .008- Architektur grundsätzlich transparent. Es wird stets angenommen, daß die jeweilige Implementierung gewährleistet, daß für die Ausführung einer Operation die erforderlichen Objekte zugriffsfertig in den jeweiligen Speichermitteln (allgemeinen bzw. solchen der speziellen Hardware) bereitstehen (dies erfordert üblicherweise eine Kombination aus einer virtuellen Speicherverwaltung und einem "garbage collector"; manche Implementierungen können auch dafür unterstützende Hardware vorsehen). Da bei jedem aktuellen Operationsaufruf bekannt ist, wieviel Objekte benötigt werden und wie groß diese sind, können sehr effiziente Speicherverwaltungsverfahren implementiert werden.
Es gibt keinen grundsätzlichen Unterschied zwischen Operationsbefehlen und Programmen: Jeder entsprechende Aufruf erfordert, daß alle Argument- Identifier im Stack bereitstehen. Derartige Blöcke von Identifiern im Stack werden als "activation records" bezeichnet (d. h. jeder Operationsaufruf erfordert den Aufbau des entsprechenden activation record). Damit ist gewährleistet, daß Programmaufrufe und Operationsbefehle gegeneinander ausgetauscht werden können. Sowohl Programme als auch Operationsbefehle sind Prozeduren bzw. Funktionen im Sinne üblicher Programmiersprachen (vgl. /10/, /26/).

Bei Programmen können Argumente anfänglich fest zugewiesen sein, wobei diese Zuweisung aufrecht erhalten bleibt, sofern sie beim aktuellen Aufruf nicht explizit geändert wird ("default"- Vorkehrungen, vgl. /10/).
Programme sind prinzipiell heterogene zusammengesetzte Objekte, die zunächst aus zwei Komponenten bestehen:

1. einer ACCESS REFERENCE TABLE (ART)
2. dem eigentlichen Codebereich.

Die ART enthält:

- alle Objektidentifier, die initial als "default"- Belegung zugewiesen wurden
- Konstanten
- beschreibende Information für bestimmte Operationsbefehle
- Selektoren
- Zeiger für die Ausnahmebehandlung und für die Übergabe von Parametern.

Prinzipiell ist die ART das Muster ("template") für den activation record, und im allgemeinen Sinne wird ein activation record dadurch aufgebaut, daß die ART auf den Stack kopiert wird und daß anschließend die Objektidentifier der übergebenen Argumente in diese Kopie transportiert werden.
Aus Effizienzgründen gibt es jedoch Abweichungen von diesem allgemeinen Schema:

1. Sind keine "default"- Belegungen vorgesehen und werden keine Konstanten, Selektoren usw. benötigt, so enthält das Programm keine ART.
2. Ist eine ART vorhanden, ohne daß "default"- Belegungen vorgesehen sind, so unterbleibt das Kopieren der ART (stattdessen werden die Identifier der Argumentobjekte direkt in den Stack transportiert).

Die Operationsbefehle der .008- Architektur sind somit äquvalent zu Programmaufrufen gemäß Pkt. 1.

Wird im Rahmen der Architekturdefinition völlig auf Adressierungsmöglichkeiten im üblichen Sinne verzichtet, so sind als Ersatz dafür spezifische Vorkehrungen zu treffen, um jene Teile in den Programmabläufen formulieren zu können, die üblicherweise durch Verwalten bzw. Manipulieren von Adressen implementiert werden (z. B. die Auswahl von Komponenten eines Objekts, den Aufbau von Objekten, das Abarbeiten von Schleifen usw.). Dies läuft darauf hinaus, in der Architektur nicht nur eine "Datenabstraktion" zu gewährleisten, sondern auch für alle anderen Aspekte des Verarbeitungsprozesses Mittel bereitzustellen, die eine "abstrakte" Formulierung erlauben (vgl. /11/). Die wesentlichen Abstraktionen sind im folgenden aufgeführt:

1. Operation
Eine Operation ist lediglich durch ihre Wirkung gekennzeichnet, d. h. durch das aus den Argumenten gebildete Resultat. Dabei ist es völlig belanglos, auf welche Weise die Argumente bereitgestellt wurden und wie das Resultat weiterverwendet wird.

2. Invocation (Beschaffung von Argumenten)
Das Resultat einer Invocation ist, daß das betreffende Objekt (bzw. eine Komponente eines Objekts) zur weiteren Verarbeitung bereitsteht. Dabei ist es nicht erforderlich, daß das Objekt bereits physisch "geladen" wird (im Sinne des Transportierens in Arbeitsspeicher o. dergl.).

3. Assignment (Zuordnung von Resultaten)
Im Ergebnis eines solchen Ablaufs wird ein Objekt einer Variablen zugeordnet. Dabei gibt es zwei Varianten:
1) Das Objekt hat einen neuen Objektidentifier, der der Variablen zugeordnet wird (die Variable zeigt auf das "neue" Objekt anstatt auf das "alte", vgl. /11/, /13/).
2) Der Inhalt des neuen Objekts wird physisch in das Ob-

jekt kopiert, das der Variablen aktuell zugeordnet ist (bzw. gleichbedeutend: das neue Objekt erhält keinen neuen Identifier, sondern jenen, der der Variablen aktuell zugeordnet ist). Damit wird der Inhalt des betreffenden "alten" Objekts geändert. Diese Änderung ist "sichtbar" für alle Variablen, denen der betreffende Identifier zugeordnet ist (vgl. /11/, /13/).

3. Selektion (Auswahl von Komponenten eines Objekts)
Das Ergebnis einer Selektion ist, daß die selektierte Komponente für die Verarbeitung bereitsteht. Wie diese Komponente ermittelt wird, bestimmt der jeweilige Selektor (einige Selektoren sind in der Architektur vordefiniert, es ist aber auch möglich, Selektionsabläufe in Anwendungsprogrammen zu formulieren).

4. Iteration
Die Iteration dient zum Bereitstellen von Objekten bzw. Komponenten innerhalb von Schleifen ("for- loops", vgl. /11/). Wie das jeweils nächste Objekt ermittelt wird, bestimmt der jeweilige Iterator (einige sind vordefiniert, es können aber auch Iteratoren in Anwendungsprogrammen formuliert werden).

5. Konstruktion (Aufbau von Objekten)
Aus gegebenen Objekten (die Komponenten anderer Objekte sein können) lassen sich neue Objekte aufbauen.
Für den Konstruktionsablauf ist es dabei unwesentlich, auf welche Weise die Argumentobjekte bereitgestellt werden.

6. Ausnahmen (Exceptions)
Spezifische Bedingungen, die während des Programmablaufs auftreten, führen zur Signalisierung von Ausnahmen (vgl. /9/, /11/). Einige Ausnahmebedingungen sind vordefiniert, andere können spezifisch in Anwendungsprogrammen formuliert werden.

2.3. Datenzugriffe

Objekte sind nur über ihre Objektidentifier zugänglich. Ein Objektidentifier ist die Ordinalzahl eines Objekts aus der Menge aller Objekte. Die Abbildung vom Objektidentifier auf das jeweilige physische Objekt erfolgt über Objekttabellen (OBJECT REFERENCE TABLE ORT). Diese enthalten die beschreibende Information für das Objekt. Wesentliche Komponenten dieser beschreibenden Information ("Objektdeskriptor") sind:

- Typ
- Status
- Position
- Größe
- Anzahl der Variablen- Zuordnungen ("reference count").

Der Objektdeskriptor enthält somit die Information, die für den Zugriff zum physischen Objekt notwendig ist. Bei zusammengesetzten Objekten ermöglicht der Objektdeskriptor physiche Zugriffe bis zu den elementaren Komponenten (die Struktur- Information ist im Deskriptor selbst gespeichert).

Bei Verbundobjekten identifiziert der Objektdeskriptor eine weitere Tabelle (COMPOUND OBJECT REFERENCE TABLE CORT), die ihrerseits die Objektidentifier der Komponenten enthält. Jede Komponente eines Verbundobjekts wird durch eine Ordinalzahl selektiert, mit der zur jeweiligen CORT zugegriffen wird.

Bei allen Arten von Objekten besteht die Möglichkeit, zusätzliche kennzeichnende Angaben direkt im Objektdeskriptor zu hinterlegen (diese Angaben können programmseitig abgefragt bzw. modifiziert werden). Damit ist eine Art von Klassenbildung innerhalb eines Objekttyps möglich.

Beispiel:

Ternärvektorlisten sind Verbundobjekte aus einem Objekt,das die Zuordnung der Variablen zu den TVL- Spalten beschreibt

(Variablenliste) und aus der eigentlichen Liste der Ternärvektoren. Ein weiteres Kennzeichen von TVL ist die "Problemform", d. h. die Angabe, welche Art von BOOLEscher Gleichung repräsentiert wird (alternative Form, konjunktive Form, Lösung = 1, Lösung = 0 usw., vgl. /16/). Diese Kennzeichnung ist im Objektdeskriptor selbst codiert (andernfalls wäre ein weiterer Eintrag in die CORT erforderlich).
Für das Deklarieren solcher Kennzeichen wird das Sprachkonstrukt "Attribut" eingeführt. Die Syntax wird anhand eines Beispiels erläutert:

<u>type</u> TVL_FORM <u>is</u> <u>attr</u> (AF0, AF1, DF0, DF1, KF0, KF1, EF0, EF1)

Hinweise:

1. Attribute sind eine Erweiterung des Typenkonzepts. Damit kann die Typenvielfalt in Grenzen gehalten werden (ohne Attribute müßten gem. obigen Beispiel 8 TVL- Typen deklariert werden, wodurch sich auch die Anzahl der zu deklarierenden Operationen usw. sinngemäß erhöhen würde).
2. Manche Befehle der .008- Architektur enthalten eine implizite Typen- bzw. Attributprüfung (bei Verletzung der jeweiligen Kriterien werden entsprechende vordefinierte Ausnahmebedingungen wirksam).
3. Typen und Attribute können zur Laufzeit getestet werden.
4. In der Sprache, die zur Formulierung der Anwendungsprogramme dient, sollte es möglich sein, bei der Deklaration von Prozeduren, Funktionen usw. die Typangabe um eine Attributspezifikation zu erweitern.

Der generelle Weg von einer Variablen in einem Programm zum jeweiligen Objekt ist folgender:

- Variablenangaben in den Befehlen sind Ordinalzahlen für den activation record des Programms.
- Der activation record enthält den Objektidentifier.

- Mit dem Objektidentifier erfolgt ein Zugriff zur ORT.
- Bei zusammengesetzten oder elementaren Objekten ermöglicht der Objektdeskriptor allein den physischen Zugriff.
- Bei Verbundobjekten ermöglicht der Objektdeskriptor den Zugriff zur jeweiligen CORT (diese "ist" das Verbundobjekt). Soll eine Komponente des Verbundobjekts selektiert werden, so wird mit der betreffenden Ordinalzahl zur CORT zugegriffen. Daraus resultiert der Objektidentifier der Komponente, mit dem wiederum zur ORT zugegriffen wird.

Details der Objektzugriffe sind in den Bildern 2-4 dargestellt.

Hinweise:

1. Die Menge aller Objekte wird als "Umgebung" (environment) bezeichnet.
2. In einem System können mehrere Umgebungen (maximal 4096) existieren. Die direkte Kommunikation zwischen den Umgebungen ist nicht möglich (es ist lediglich das physische Kopieren von Objekten durch System- "utilities" vorgesehen).
3. Eine Umgebung umfaßt maximal $2^{24}-1$ Objekte.
4. Eine Umgebung ist in maximal 4096 Regionen zu 4096 Objekten aufgeteilt. Diese Aufteilung erfolgt lediglich aus Effizienzgründen (Verwaltung der ORT) und ist für die Benutzung transparent.

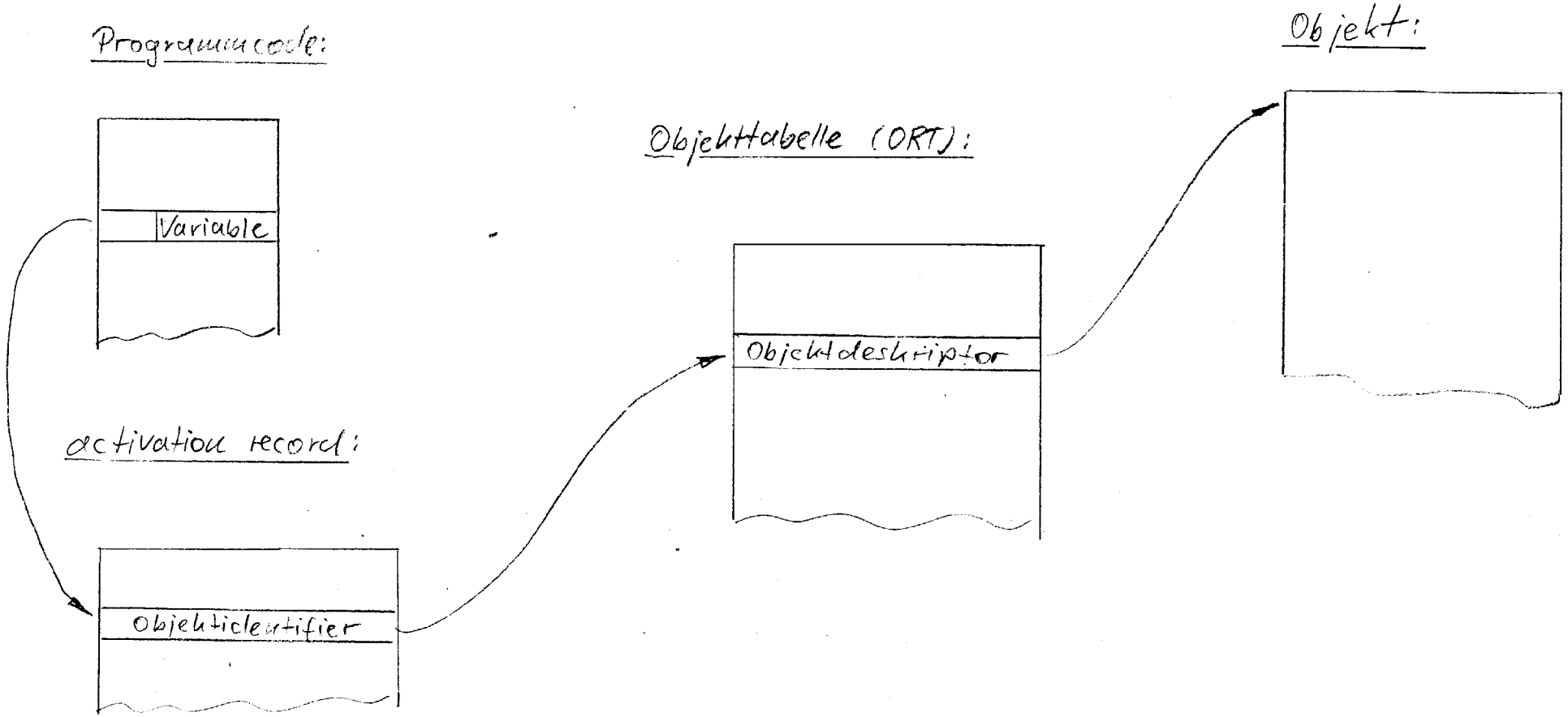


Bild 2

Prinzipielles Schema eines Objektzugriffs

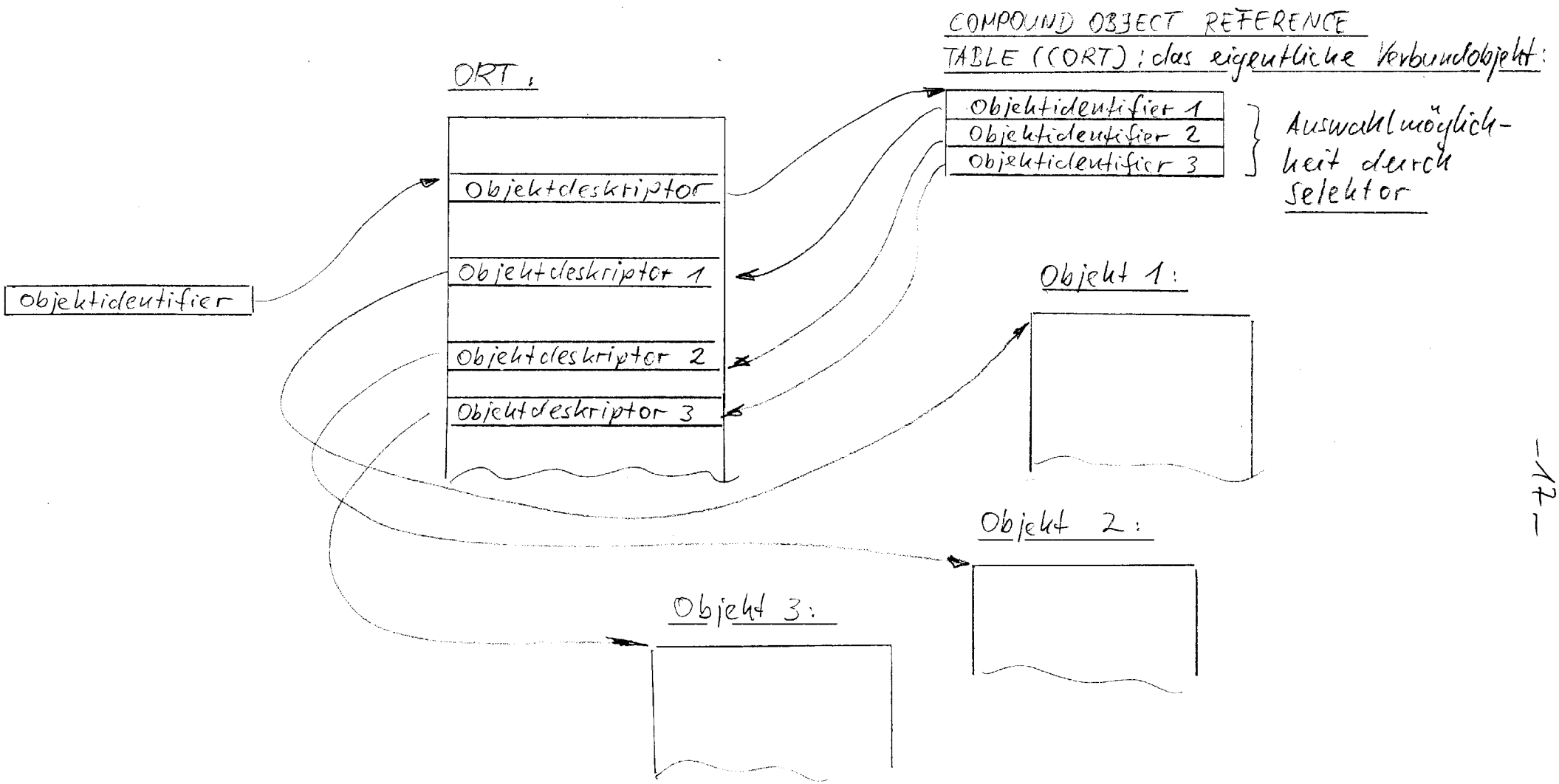


Bild 3

Zugriff zu Komponenten von Verbundobjekten

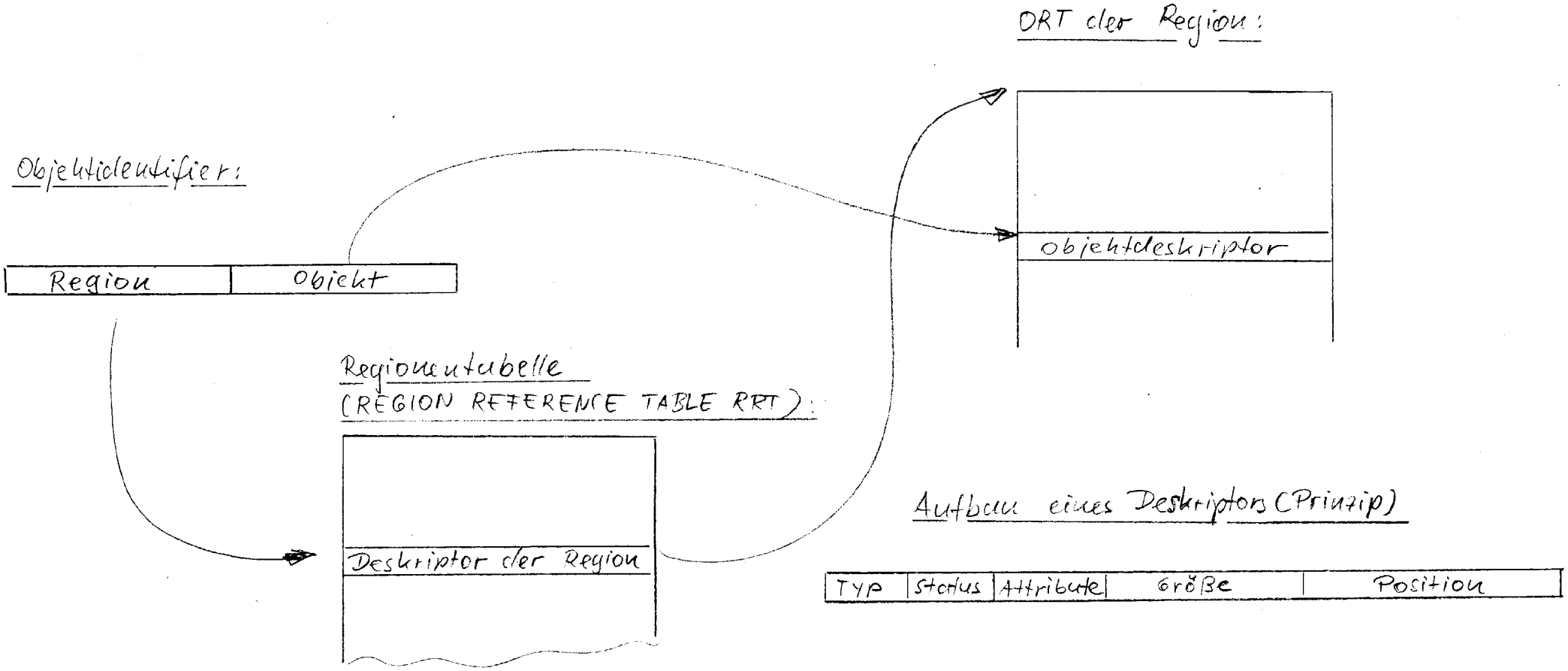


Bild 4

Details der Objektidentifier und -tabellen

2.4. Struktur der Befehle

Es gibt 16 verschiedene Typen von Befehlen. Jeder Befehl wird durch ein 16- Bit- Wort repräsentiert. Dabei bilden vier Bit den Funktionscode (FC), der den Typ des Befehls kennzeichnet. Mit den verbleibenden 12 Bit ist eine finite Ordinalzahl codiert, womit ein Element aus einer spezifischen Menge (abhängig vom Typ des Befehls) selektiert wird.
Es gibt folgende Typen:

1. Organisatorische Operationen (ORGANIZATION OPERATOR; FC 0)
   Es wird eine Operation aus der Menge der organisatorischen Operationen ausgeführt. Beispiele für solche Operationen: Starten und Beenden von Abläufen, Speicherverwaltung, Bilden von Lambda- Ausdrücken, Eintritt in Programme, Verlassen von Programmen usw.

2. Prozedurale Operationen (PROCEDURAL OPERATOR; FC 1)
   Es wird eine Operation aus der Menge der Prozeduren bzw. Funktionen ausgeführt, die auf der Architekturebene verfügbar sind. Derartige Operationen sind mit den z. B. in /16/, /17/ beschriebenen Basis- und Mengenalgorithmen vergleichbar. Beispiele: Durchschnittsbildung, Vereinigung, Komprimieren, Expandieren von TVL usw.

3. Zugriffe zu Argumenten (INVOCATION; FC 2)
   Die Ordinalzahl im Befehl bezeichnet eine Position im activation record. Handelt es sich um ein Objekt, das seinem Typ nach als Argument für Operationen in Frage kommt, so wird dessen Identifier auf den Stack transportiert. Handelt es sich um einen Iterator, so wird dieser für eine nachfolgende Benutzung initialisiert.

4. Zuordnung (ASSIGNMENT; FC 3)
   Die Ordinalzahl im Befehl bezeichnet eine Position im activation record. Diese Position wird mit dem

Objektidentifier geladen, der sich in der ersten Position des Stack befindet. War bereits ein Objektidentifier in der Position des activation record vorhanden, so wird dieser dereferenziert, d. h. er wird freigegeben, so daß das betreffende Objekt vernichtet werden kann, sofern es nicht noch anderen Variablen zugewiesen ist.

5. Modifikation (MODIFICATION; FC 4)
   Die Ordinalzahl im Befehl bezeichnet eine Position im activation record. Befindet sich dort ein Objektidentifier, so wird das betreffende Objekt mit dem Wert des Objekts überladen, das durch den Identifier in der ersten Position des Stack bezeichnet wird.

6. Finiter Direktwert (FINITE IMMEDIATE; FC 5)
   Die 12- Bit- Angabe im Befehl bezeichnet einen Direktwert, der als solcher in den Stack gebracht wird.

7. Transfiniter Direktwert (TRANSFINITE IMMEDIATE; FC 6)
   Die Ordinalzahl im Befehl bezeichnet eine Position im activation record. Deren Inhalt wird als 32- Bit- Direktwert interpretiert.

8. Auswertung (YIELD; FC 7)
   Die Ordinalzahl im Befehl bezeichnet eine Position im activation record. Die betreffende Position enthält entweder einen Objektidentifier oder ein Steuerwort für eine programminterne Steuerabstraktion (Selektor, Iterator, Lambda- Ausdruck). Es wird der entsprechende aktuelle Wert ermittelt.

9. Ausführung (EXECUTE; FC 8)
   Die Ordinalzahl im Befehl bezeichnet eine Position im activation record. Diese enthält entweder einen Objektidentifier oder ein Steuerwort, das eine spezifische interne Operation beschreibt. Die betreffende Operation wird ausgeführt.

10. Bedingte Ausführung (EXECUTE ON TRUE; FC 9)
    Die Wirkung gem. Pkt. 9 wird ausgeführt, wenn für die erste Position des Stack eine (zuvor selektierte oder durch Testoperationen ermittelte) Bedingung den Wert TRUE hat. Ansonsten wird der folgende Befehl abgearbeitet.

11. Bedingte Ausführung (EXECUTE ON FALSE; FC 10)
    Wirkung analog zu Pkt. 10; Ausführung dann, wenn die Bedingung den Wert FALSE hat.

12. Verzweigung (BRANCH; FC 11)
    Die Ordinalzahl im Befehl wählt den nächsten Befehl aus der Menge der Befehle des Programms aus.

13. Bedingte Verzweigung (BRANCH ON TRUE; FC 12)
    Die Verzweigung gem. Pkt. 12 wird nur dann ausgeführt, wenn die Bedingung gem. Pkt. 10 den Wert TRUE hat. Ansonsten wird der folgende Befehl abgearbeitet.

14. Bedingte Verzweigung (BRANCH ON FALSE; FC 13)
    Wirkung analog zu Pkt. 13; Ausführung der Verzweigung dann, wenn die Bedingung den Wert FALSE hat.

15. Ausnahmebedingung (RISE EXCEPTION; FC 14)
    Die Ordinalzahl im Befehl bezeichnet eine Ausnahmebedingung aus der Menge aller Ausnahmebedingungen (sowohl der in der Architektur gegebenen als auch der zusätzlich von Anwendern definierten).
    Für diese Bedingung wird der "innerste" Behandler gesucht und gestartet (die Suche beginnt im activation record des Programms).

16. Reserviert (RESERVED; FC 15)
    Dieser Funktionscode ist für künftige Erweiterungen reserviert.

2.5. Struktur der Programme

Programme sind grundsätzlich heterogene zusammengesetzte Objekte (vg. Abschnitt 2.2.), die aus einer ART und dem eigentlichen Codebereich bestehen. Dieser umfaßt eine Befehlsfolge oder mehrere davon. Jede Befehlsfolge kann maximal 4096 Befehle haben.
Zur Laufzeit des Programms repräsentiert die ART nach der Belegung mit den aktuellen Parametern den activation record des Programms. Dieser kann maximal 4096 Positionen umfassen (Objektidentifier + Steuerworte + Direktwerte usw.). Jede Position wird durch ein 32- Bit- Wort repräsentiert.
Das Prinzip der Codierung von Programmabläufen besteht darin, daß nur kurze und elementare Befehle vorgesehen sind und daß zusätzliche Information prinzipiell aus dem activation record entnommen wird.

Hinweise zu den einzelnen Befehlstypen:

1. ORGANIZATION OPERATOR
   Es sind maximal 4096 verschiedene Operationen möglich. Die Menge dieser Operationen umfaßt alle organisatorischen Abläufe, die in der Architektur definiert sind sowie allgemein übliche elementare Funktionen (Ein- und Ausgabe, Rechnen mit Binärzahlen usw.).

2. PROCEDURAL OPERATOR
   Es können maximal 4096 Operationen definiert werden. In die betreffende Menge werden nur jene Abläufe aufgenommen, die Operationen über die Datenstrukturen der Architektur darstellen (mit Ausnahme elementarer Funktionen gem. Pkt. 1).

3. YIELD
   Enthält der activation record in der betreffenden Position einen Objektidentifier, so wird dieser wie folgt ausgewertet:
   - im Falle eines Objekts mit zugewiesenem Wert: keine Wirkung (NO OP)
   - im Falle eines Lambda- Ausdrucks: Berechnung des Wertes
   - im Falle eines Programms, das eine Iteration bzw. Selektion beschreibt, wird dies aufgerufen, und die jeweils ermittelten Werte werden zurückgegeben.

   Enthält der activation record ein Steuerwort für einen intern definierten Iterator bzw. Selektor, so werden die aktuellen Werte zurückgegeben.

4. EXECUTE
   Enthält der activation record in der betreffenden Position einen Objektidentifier, der ein anderes Programm identifiziert, so wird dies im Sinne eines Unterprogramms ausgeführt.
   Alternativ dazu kann der activation record ein Steuerwort enthalten. Ein solches Steuerwort kann beschreiben:
   - eine Verzweigung zu einem Befehl einer anderen Befehlsfolge des Programms
   - den Code des auszuführenden Befehls ("Befehlssubstitution")
   - einen erweiterten Befehl, der aus einem Operationscode und weiteren Steuerbits besteht.

5. BRANCH
   Es kann nur zu Befehlen in der aktuellen Befehlsfolge verzweigt werden (andernfalls ist die EXECUTE- Funktion mit einem entsprechenden Steuerwort im activation record zu verwenden. Die Position des Folgebefehls wird bei der Verzweigung gerettet, so daß eine Rückkehr (im Sinne eines Unterprogrammaufrufs) möglich ist. Dies

wird durch die "angesprungene" Befehlsfolge gesteuert: Ein Befehl TERMINATE (FC 0) veranlaßt die Rückkehr zum besagten Folgebefehl, ein Befehl CANCEL löscht die Rückkehrposition (Übergang zum Ausgangsniveau) und ein Befehl END veranlaßt das Beenden des gesamten Programms.

Hinweise zu typischen Abläufen:

1. Iteration

   Diese wird in üblichen Programmiersprachen durch "FOR-loops" beschrieben, z. B. in der Form

   <u>for</u> X <u>do</u> A <u>end</u>

   Dabei ist X ein Iterator, der den Schleifenkörper ("loop body") A mit aktuellen Argumenten versorgt und über das Beenden der Schleife entscheidet. Ein solcher Ausdruck wird innerhalb eines Programms wie folgt codiert:

   <u>invoke</u> X --Initialisieren des Iterators
   <u>execute</u> A --Ausführen des "loop body"
   <u>yield</u> X -- Auswerten des Iterators

   Der YIELD- Befehl hat dabei folgende Wirkung: Ist die Iteration noch nicht beendet, so wird zum vorhergehenden Befehl verzweigt, andernfalls zum nachfolgenden.
   Für X und A gibt es jeweils eine Position im activation record. Jede dieser Positionen kann wahlweise enthalten:
   - einen Objektidentifier
   - ein spezifisches Steuerwort (Befehlssubstitution, Befehle mit zusätzlichen Modifikationsmöglichkeiten, in der Architektur definierte elementare Iteratoren)
   - ein Steuerwort, das den Eintritt in eine spezifische Befehlsfolge des Programms veranlaßt (Iterator bzw. "loop body" sind innerhalb des Programms separat codiert).

2. Befehle mit Modifikationsmöglichkeiten

   Derartige Befehle werden durch EXECUTE- Befehle zur Wirkung gebracht. Dabei bestimmt die Position des activation record sowohl die Operation als auch deren Modifikation.

Der Zweck besteht in der effizienten Codierung von Operationen, die sich auf jeweils einen grundlegenden Ablauf zurückführen lassen, wobei eine Vielzahl von Modifikationen bzw. Kombinations- Möglichkeiten existiert. Ein typisches Beispiel bilden die Suchoperationen (Durchmustern von Listen), wobei Suchkriterium, Suchrichtung, die Typen der Argument- Objekte usw. auf vielfältige Weise modifiziert werden können.
Ohne Vorkehrungen zur Befehlsmodifikation müßte für jede wünschbare Variante ein Operationsbefehl vorgesehen werden, und für die Programmierung selbst wäre es erforderlich, die Vielzahl der entsprechenden Mnemonics zu kennen. Hingegen gibt es bei der Befehlsmodifikation nur jeweils eine Mnemonic, an die die einzelnen Spezifikationen angefügt werden.
Um derartige Befehle deklarieren zu können, wird die übliche Prozedur- bzw. Funktionsdefinition um ein Sprachkonstrukt <u>depending on</u> erweitert. An diesen Ausdruck schließt unmittelbar eine "specifier list" an, womit die einzelnen Möglichkeiten der Modifikation deklariert werden (z. B. als Aufzählungen). Die Liste selbst ist (implizit) vom Typ <u>record area</u>.
Nach dieser Liste werden die Argumente bzw. Resultate in üblicher Weise deklariert.
Beispiel (fiktiv):

```
functions SEEK depending on
          SEEK_DIRECTION: (FORWARD,BACKWARD)
          END_CRITERION: (LESS, EQUAL, GREATER)
          (SEEK_TVL: TVL_AREA, SEEK_VECTOR: TVEC,
           SEEK_CRITERION: BOOLEAN_FORMULA)
           return CONDITION
```

3. Lambda- Ausdrücke

Ein Lambda- Ausdruck ist ein heterogenes zusammengesetztes Objekt, das anstelle eines Wertes aus einem activation record mit den Identifiern der aktuellen Argumente und dem Programmcode besteht, der den Resultatwert erzeugt. Damit wird vom Wert eines Objekts an sich abstrahiert: stattdessen wird die Rechenvorschrift gespeichert, deren Auswertung (YIELD- Befehl) den Wert ergibt. Der Zweck besteht in der Einsparung von Speicherplatz. für Objekte, deren Wert nicht unmittelbar benötigt wird (ein Lambda- Ausdruck benötigt wesentlich weniger Platz als etwa eine umfangreiche Ternärvektorliste).
Beispiel:

a) konventionell	b) mit Lambda- Ausdruck
A:= UNION (B,C)	A:= <u>lambda</u> UNION (B,C)
.....	.....
weitere Operationen, in denen A nicht benötigt wird	
.....	.....
X:= INTERSECTION (A,Y)	X:= INTERSECTION (YIELD (A),Y)

Im konventionellen Fall (a) belegt das Resultatobjekt auch dann den Speicherplatz für seinen Wert, wenn es für weitere Operationen nicht benötigt wird.

4. Übergang zwischen den Programmen
Werden dem zu rufendem Programm alle Argumente übergeben (kein "default" erforderlich), so werden diese in der erforderlichen Reihenfolge auf dem Stack bereitgestellt. Das Programm selbst wird mit einem EXECUTE-

Befehl aufgerufen. Dabei bleibt das aufrufende Programmniveau zunächst erhalten (zum Beenden des Programms s. den Hinweis zu den BRANCH- Befehlen).
Werden nicht alle Argumente übergeben (d. h. Nutzung der "default"- Vorkehrungen), so wird folgender Ablauf benutzt:

INVOKE pgm_x --Zugriff zum Programm pgm_x

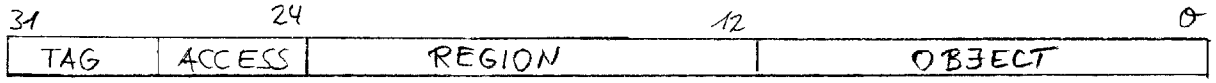
COPY_ART -- Kopieren der Argumentangaben aus der ART auf den Stack

EXECUTE -- Transport der aktuellen Parameter aus dem Stackbereich des rufenden Programms in den des gerufenen. Dafür ist im activation record des rufenden Programms ein Steuerwort vorgesehen. Dieses zeigt auf eine spezielle Befehlsfolge im Codebereich, die den Transport im einzelnen steuert (s. Bild 8).

ENTER -- Übergang zum gerufenen Programm.

Details des Programmaufbaus zeigen die Bilder 5- 8.
Einige Gesichtspunkte für die Wahl dieser Lösung sind im folgenden angeführt:

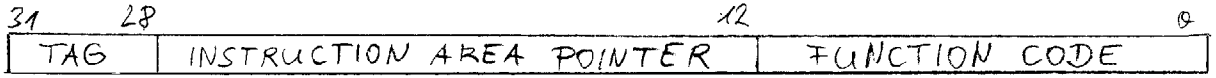
1. Es soll eine weitgehende Abstraktion von den konkreten Operationen erreicht werden. Befehlsformate und Zugriffsprinzipien sollen so allgemeingültig sein, daß die Architektur auch für andere Aufgabenstellungen brauchbar ist.
   Dies wird dadurch erreicht, daß alle problemspezifischen Operationen über ein Befehlsformat aufrufbar sind.
   (Bei einer anderweitigen Nutzung der Architekturprinzipien werden lediglich die Ordinalzahlen in den Befehlen vom Typ PROCEDURAL OPERATOR anderweitig interpretiert.)



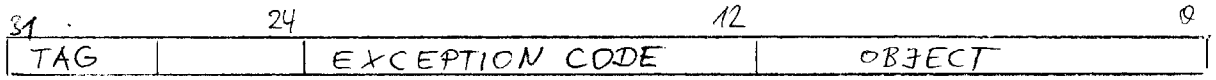
a.) Objektidentifier

b.) Direktwert

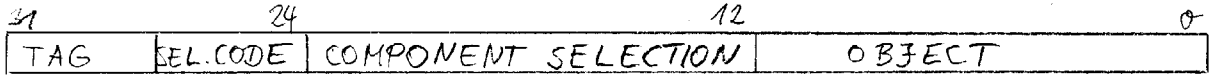
c.) Steuerwort für Befehlsmodifikation



d.) Steuerwort für programm-interne Abläufe



e.) Pointer für Ausnahmebehandlung



f.) elementarer Selektor

Bild 5 Formate in der ART

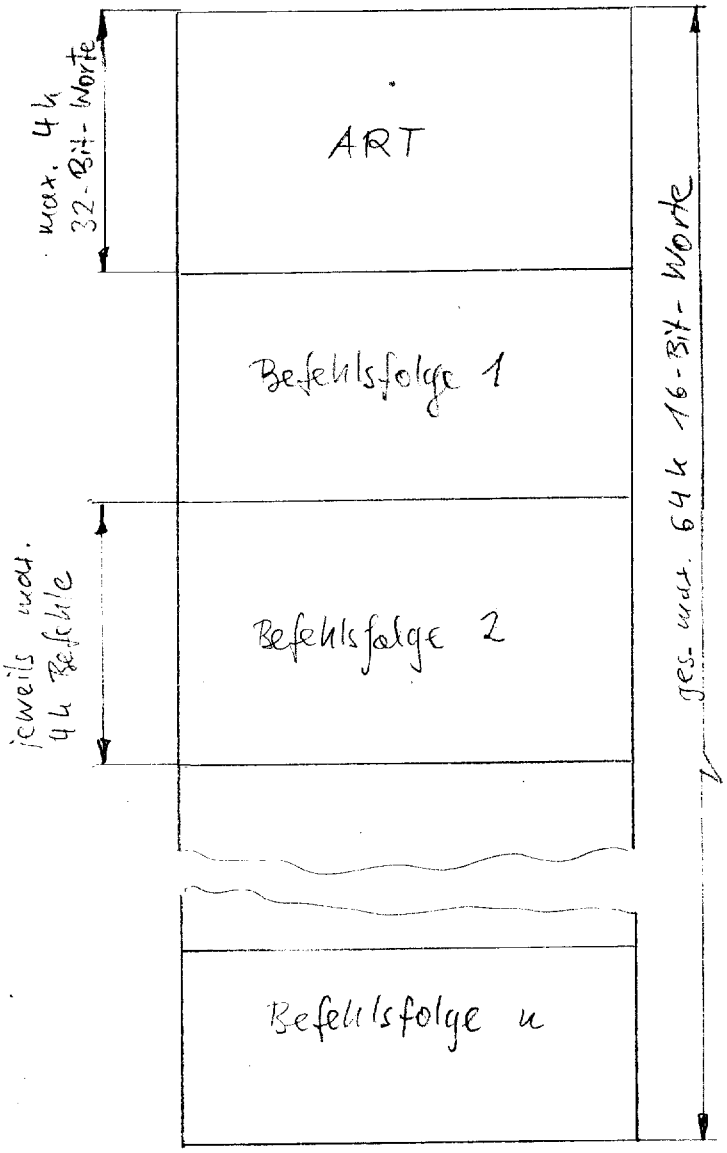
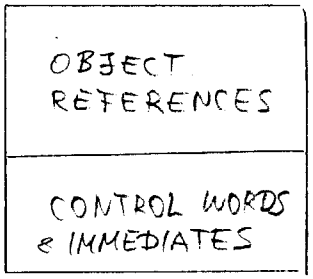


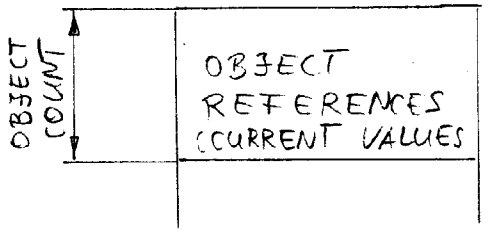
Bild 6 Prinzipieller Aufbau eines Programms

ART:

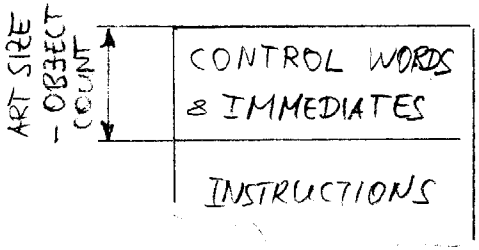


ACTIVATION RECORD:

a.) on stack:



b.) program code:



PGM OBJECT DESCRIPTOR LAYOUT:

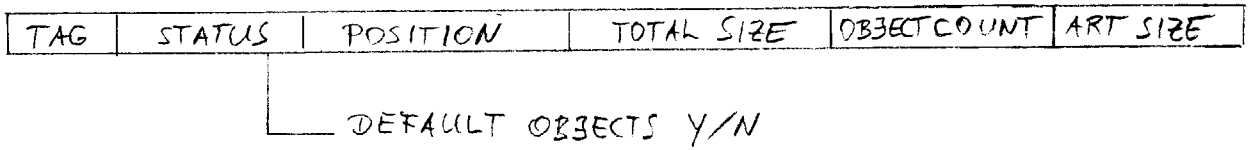
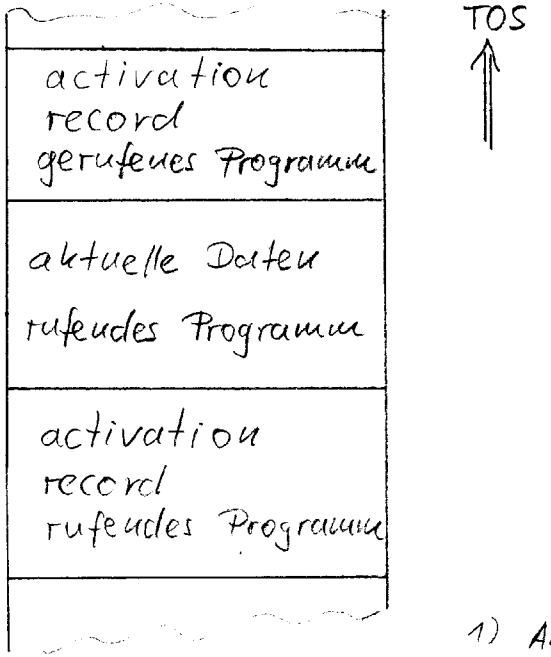


Bild 7

Beziehung zwischen ART und activation record (Übersicht)

Belegung des Stack:



Kopieren von Objekt-Referenzen:

a.) Befehl

EXEC	

Detail CONTROL WORD:

TAG	CODE AREA POINTER	OBJECT COUNT[1]

b.) ART

CONTROL WORD

c.) Code für Transport (im rufenden Programm)

INVOKE	Obj a	
ASSIGN	Obj x	
INVOKE	Obj b	
ASSIGN	Obj y	

Transporte

x := a

y := b

1) Anzahl der Paare INVOKE/ASSIGN

a, b : bezogen auf activation record des rufenden Pgm.

x, y : bezogen auf activation record des gerufenen Pgm.

Bild 8

Details der Parameterübergabe zwischen Programmen

2. Durch die 12- Bit- Direktwerte in den Befehlsformaten ist zum einen eine ausreichende Vielfalt von Codierungsmöglichkeiten gegeben (etwa im Gegensatz zu den in vielen Architekturen verwendeten 8- Bit- Operationscodes).
   Zum anderen ist die Vielfalt nach oben hin so begrenzt, daß für die Implementierung einfache Verzweigungstabellen noch technisch realisierbar sind (eine solche Tabelle würde maximal 4096 Einträge haben, z. B. um zu den ausführenden Abläufen für 4096 verschiedene Operationen zu verzweigen).

3. Es soll möglich sein, eine Vielzahl komplexer Operationen in speziellen Hardwaremitteln auszulösen. Dies führt zu folgenden Problemen:
   - Viele in der Hardware auslösbare Abläufe entsprechen aus der Sicht der konventionellen Programmierung eher recht komplexen Unterprogrammen als elementaren Maschinenbefehlen.
   - Die Hardware stellt eine Vielzahl von Sonderwirkungen, Kombinations- und Variationsmöglichkeiten zur Verfügung, die mit üblichen Operationscodes kaum für die Programmierung zugänglich gemacht werden können (die Anzahl der Varianten wäre zu groß).

   Deshalb wurde neben einem an sich sehr großen Operationscode- Vorrat die Möglichkeit zur Befehlsmodifikation (mit Steuerworten in der ART) eingeführt.

4. Es ist möglich, Substrukturen (z. B. <u>for</u>- loops) innerhalb eines Programmes selbst zu codieren (als Alternative wäre die Erzeugung jeweils neuer Programm- Objekte denkbar). Der Vorteil besteht darin, daß die Objektvielfalt kontrollierbar bleibt: es ist nicht nötig, neben den explizit deklarierten Objekten im Verlauf der Compilierung neue Objekte einzuführen.
   Im einzelnen ist es über die ART möglich, innerhalb eines Programms zu jeder Code- Position (Befehl, Steuerwort usw.) dirkt zuzugreifen.

3. Deklarationen

3.1. Elementare Datenstrukturen

```
type FINITE_CARDINAL is INTEGER range 0 .. 4096;
type FINITE_ORDINAL is INTEGER range 1 .. 4096;
type TRANSFINITE_CARDINAL is INTEGER range 0 .. MACHINE_BOUND;
type TRANSFINITE_ORDINAL is INTEGER range 1 .. MACHINE_BOUND;
-- MACHINE_BOUND ist eine maschinenspezifische Obergrenze;
-- deren Maximalwert ist 2^32-1

type BOOLEAN is (FALSE, TRUE);
type TERNARY is (DONT_CARE, LO, HI, INVALID);
-- DONT_CARE: - ; LO: 0; HI: 1; die Bezeichnungen der Werte
-- wurden lediglich aus formalen Gründen so eingeführt

-- es folgt die Deklaration von Parametern für Vektoren und
-- Listen
type VAR_COUNT is FINITE_CARDINAL;
type ROW_COUNT is TRANSFINITE_CARDINAL;

-- es folgen die elementaren Vektoren und Listen- Bereiche
type BVEC is area (1 .. VAR_COUNT) of BOOLEAN;
type BVL_AREA is area (1 .. ROW_COUNT) of BVEC;
type TVEC is area (1 .. VAR_COUNT) of TERNARY;
type TVL_AREA is area (1 .. ROW_COUNT) of TVEC;
```

```
-- zum Kennzeichnen von Variablen- Zuordnungen wird die
-- Variablenliste eingeführt:

type VAR_NAME is record area

        TAG: BOOLEAN ( 4 ):= 0;
        DERIVATION_CODE: INTEGER range 0 .. 15:= 0;
        VAR_NAME_PROPER: INTEGER range 0 .. (2 exp 24)-1;

-- damit ist der einzelne Variablenname als 32- Bit- Wort
-- eingeführt. Mit dem gegebenen Format wird eine von
-- maximal 2^24-1 Variablen beschrieben, wobei von jeder
-- Variablen bis zu 16 Ableitungen gebildet werden können.
-- Generell ist es möglich, auch andere Formate zu definieren.
-- Diese müssen sich durch die Belegung des TAG- Feldes voneinander unterscheiden.

type VARLIST is area (1 .. VAR_COUNT) of VAR_NAME;

--damit kann die TVL als Verbundobjekt definiert werden:

type TVL is record

      PROBLEM_FORM: attr (KF0, KF1, DF0, DF1, AF0, AF1,
                          EF0, EF1, UNCOMMITTED);

      VAR_ASSIGNMENT: VARLIST;

      TVL_PROPER: TVL_AREA;

-- sinngemäß wird die Binärvektorliste eingeführt:

type BVL is record

      PROBLEM_FORM: attr (KF0, KF1, DF0, DF1, AF0, AF1
                          EF0, EF1, UNCOMMITTED);

      VAR_ASSIGNMENT: VARLIST;

      BVL_PROPER: BVL_AREA;

-- eine weitere elementare Struktur ist die Indexliste:

type INDEX_LIST is area (1 .. VAR_COUNT) of FINITE_ORDINAL;
```

```
-- BOOLEsche Gleichungen können weiterhin durch Formel-
-- ausdrücke beschrieben werden (namentlich zu Ein- und
-- Ausgabezwecken).
-- Derartige Formelausdrücke sind als Verbundobjekte
-- definiert:

type FORMULA is record
     EQUATION_RESULT: attr (LO, HI);
     VAR_ASSIGNMENT: VARLIST;
     FORMULA_PROPER: FORMULA_EXPRESSION;

-- Der eigentliche Formelausdruck (FORMULA_EXPRESSION)
-- besteht aus 16- Bit- Worten, die analog zu den
-- Maschinenbefehlen strukturiert sind.
-- Damit wird die jeweilige BOOLEsche Gleichung in
-- invertierter polnischer Notation (RPN) beschrieben.
-- Zu Details der Formelausdrücke s. Bild 9.
```

3.2. Elementare Selektoren

```
-- Selektoren ermöglichen Zugriffe zu Komponenten von
-- Objekten.
-- Um die betreffende Komponente direkt zu erhalten,
-- ist ein YIELD- Befehl auf den betreffenden Selektor
-- anzuwenden.
-- Manche Prozeduren bzw. Funktionen können Selektoren
-- zurückgeben.

-- Mit den folgenden Selektoren kann jeweils eine
-- enzelne Variable in einem Vektor identifiziert
-- werden (YIELD ergibt den Wert dieser Variablen):

type VAR_IN_BVEC is record
     B_VECTOR: BVEC;
     VAR_POSITION: FINITE_ORDINAL;
```

Struktur eines 16-Bit-Wortes:

15 … 12	11 … 0
Operation	Variable

a.) Variablenauswahl und Verknüpfung

1 1 1 0	Operation

b.) Verknüpfung von Variablen (im Stack)

(Opcodes 2–9: Verkn. <TOS> mit <Variable> ⇒ TOS

Operationscode (hex.):

0 - PUSH
1 - PUSH INVERTED
2 - PUSH AND
3 - PUSH AND INVERTED
4 - PUSH OR
5 - PUSH OR INVERTED
6 - PUSH EQUIV
7 - PUSH EQUIV INVERTED
8 - PUSH XOR
9 - PUSH XOR INVERTED
A - D : reserviert
E - Verknüpfung (Variable im Stack)
F - Ende der Formelausdruckes

Bild 9

Details von Formelausdrücken

```
type VAR_IN_TVEC is record
     T_VECTOR: TVEC;
     VAR_POSITION: FINITE_ORDINAL;

-- Mit den folgenden Selektoren werden Zeilen in Listen
-- ausgewählt (YIELD liefert den Zeilen- Vektor):
type BVL_ROW is record
     BV_LIST: BVL_AREA;
     ROW_NUMBER: TRANSFINITE_ORDINAL;
type TVL_ROW is record
     TV_LIST: TVL_AREA;
     ROW_NUMBER: TRANSFINITE_ORDINAL;

-- Weiterhin können auch Variable in Listen ausgewählt
-- werden:
type VAR_IN_BVL is record
     BV_LIST: BVL_AREA;
     ROW_NUMBER: TRANSFINITE_ORDINAL;
     VAR_POSITION: FINITE_ORDINAL;

type VAR_IN_TVL is record
     TV_LIST: TVL_AREA;
     ROW_NUMBER: TRANSFINITE_ORDINAL;
     VAR_POSITION: FINITE_ORDINAL;
```

3.3. Prozedurale Operationen

-- Zunächst werden die "TVL- Mengenoperationen" dargestellt:

```
function ORTHOGONALIZATION (P1: TVL) return TVL;
function UNION (P1; P2: TVL) return TVL;
function INTERSECTION (P1, P2: TVL) return TVL;
function DIFFERENCE (P1, P2: TVL) return TVL;
function SYMM_DIFF (P1, P2: TVL) return TVL;
function COMPL_SYM_DIFF (P1, P2: TVL) return TVL;
function COMPLEMENT (P1: TVL) return TVL;
```

-- Die angeführten 7 Operationen gibt es noch mit drei
-- weitern Formen der Parameter- Übergabe:
-- 1.: jeweils mit einem zusätzlichen Parameter
-- MASK_VECTOR: BVEC
-- 2.: ohne diesen zusätzlichen Parameter, aber statt TVL steht TVL_AREA (d. h., es findet keine Angleichung der Variablenpositionen vor der eigentlichen Operation statt)
-- 3.: wie 2., aber mit zusätzlichem Maskenvektor gem. 1.
-- insgesamt gibt es also 28 entsprechende Befehle.

-- Zur Komprimierung von TVL ist verfügbar:

```
function ROW_REDUCTION (P1: TVL_AREA) return TVL_AREA;
```

-- dies entspricht dem "Blocktausch"

```
function ROW_REDUCT_AUGMENTED (P1: TVL_AREA) return TVL_AREA;
```

-- dies ist eine Kombination von Blockbildung und Blocktausch.

```
-- Die folgenden Durchmusterungen von TVL werden bevor-
-- zugt mit besonders hoher Geschwindigkeit aosgeführt:

function SEEK_ORTHOGONALITY (P1: TVL_AREA, P2: TVEC)
                                return TVL_ROW;

function SEEK_NONORTHOGONALITY (P1: TVL_AREA, P2: TVEC)
                                return TVL_ROW;

function SEEK_ROW_REDUCTION (P1: TVL_AREA, P2: TVEC)
                                return TVL_ROW;

function SEEK_ROW_SWAPPING (P1: TVL_AREA, P2: TVEC)
                                return TVL_ROW;

function SEEK_LOGIC_EQUALITY (P1: TVL_AREA, P2: BVEC)
                                return TVL_ROW;

function SEEK_LOGIC_UNEQUALITY (P1: TVL_AREA, P2: BVEC)
                                return TVL_ROW;

function SEEK_BINARY_EQUALITY (P1: TVL_AREA, P2: BVEC)
                                return TVL_ROW;

function SEEK_BINARY_UNEQUALITY (P1: TVL_AREA, P2: BVEC)
                                return TVL_ROW;

-- Die angeführten 8 Operationen gibt es nochmals mit je-
-- weils einem zusätzlichen Parameter: MASK_VECTOR: BVEC
-- insgesamt gibt es also 16 entsprechende Befehle.

-- Allgemeine Durchmusterungen werden über ein Steuer-
-- wort modifiziert. Die Struktur dieses Steuerwortes wird
-- zunächst beschrieben:

type SCAN_CONTROL is record area
     SCAN_CRITERION: (ORTHOGONALITY, ORTHOGONALIZATION_
                      POSSIBLE, EQUALITY, UNEQUALITY,
                      NEGATION, N_ELEMENT, NO_N_ELEMENT,
                      L_ELEMENT, NO_L_ELEMENT, H_ELEMENT,
                      NO_H_ELEMENT);
```

```
    SCAN_MODE: (NUMBER_OF_OCCURRENCES, FIRST_OCCURRENCE);
    SCAN_DIRECTION: (FORWARD, BACKWARD);
    SCAN_ORIENTATION: (HOR, VERT);
    SCAN_RELATION: (EQU, UNEQU, LESS, GREATER, LE, GE);

-- Damit kann die allgemeine Durchmusterungsoperation
-- wie folgt eingeführt werden:
functions SCAN depending on SCAN_CONTROL (P1: TVL_AREA,
                P2: TVEC, SCAN_VALUE: INTEGER)
                return TVL_ROW;

-- Bei manchen Durchmusterungen ist der zweite Parameter
-- nicht erforderlich.
-- Weiterhin gibt es diese Operationen noch mit einem
-- zusätzlichen Maskenoperanden.
-- Es handelt sich also um insgesamt 4 Befehle.
-- Der Typ von SCAN_VALUE ist im einzelnen:
-- für "Quersumme" (NUMBER_OF_OCCURRENCES): Kardinalzahl
-- für "Index" (FIRST_OCCURRENCE): Ordinalzahl
-- bei horizontaler (HOR) Orientierung: finiter Wert
-- bei vertikaler (VERT) Orientierung: transfiniter Wert.

-- Im folgenden werden elementare variablenweise Wand-
-- lungen beschrieben:
-- Dazu wird zunächst ein Steuerwort definiert:
type CONV_CONTROL is record area
    CONV_MODE: (NEG, NL, NH, LN, LH, HN, HL, MERGE,
                SWAP_BITS);

-- NEG repräsentiert die "Negation nach DeMorgan"
-- LN repräsentiert die "Shegalkin- Transformation"
-- (Wandlung "O - N").
```

```
-- Damit ergibt sich die Wandlungsoperation wie folgt:
functions CONVERT depending on CONV_CONTROL
          (P1: TVL_AREA, TAG_VECTOR: BVEC) return TVL_AREA;
-- Eine weitere Version ist:
functions CONVERT depending on CONV_CONTROL
          (P1: TVEC, TAG_VECTOR: BVEC) return TVEC;
-- Beide Versionen gibt es weiterhin mit zusätzlichem
-- Maskenvektor.
-- Eine weitere Operation dieser Art ist die Zerlegung
-- einer TVL in eine Erlaubnis- und eine Wertliste:
procedure TVL_SEPARATION (P1: in TVL_AREA; R1, R2:
                                  out BVL_AREA);
-- Weiterhin ist es möglich, unter Steuerung eines
-- Markierungsvektors Variablen in zwei Ternärvektoren
-- auszutauschen:
procedure SWAP_VAR (P1, P2: in TVEC; TAG_VECTOR: in BVEC;
                         R1, R2: out TVEC);

-- Die beiden Operationen gibt es auch mit zusätzlichem
-- Maskenvektor.

-- Sowohl Durchmusterungs- als auch variablenweise Ver-
-- knüpfungsoperationen sind nicht nur mit den beschrie-
-- benen (fest vorgesehenen), sondern auch mit frei
-- programmierten Verknüpfungen ausführbar. Die Ver-
-- knüpfungen selbst betreffen dabei jeweils eine Variab-
-- le von zwei Ternärvektoren, eines Maskenvektors und
-- eines Markierungsvektors.
-- Die Verknüpfungen werden im einzelnen durch einen
-- Verknüpfungsdeskriptor beschrieben, der ein Objekt
```

```
-- im Rahmen der .008- Architektur ist (seine interne Struk-
-- tur ist implementierungsspezifisch).
-- Jeder Verknüpfungsdeskriptor kann bis zu 32 verschiedene
-- Verknüpfungen beschrieben.
-- Ein elementarer Satz von Verknüpfungen ist in der Archi-
-- tektur vordefiniert.

-- Damit werden folgende Steuerworte eingeführt:

type PROGRAMMABLE_SCAN_CONTROL is record area

     SCAN_CRITERION: FORMULA_NUMBER;
     FORMULA_SELECTION: COMBINATION_DESCRIPTOR;

     SCAN_MODE: (NUMBER_OF_OCCURRENCES, FIRST_OCCURRENCE);
     SCAN_DIRECTION: (FORWARD? BACKWARD);
     SCAN_ORIENTATION: (HOR, VERT);
     SCAN_RELATION: (EQU, UNEQU, LESS, GREATER, LE, GE);

type PROGRAMMABLE_CONV_CONTROL is record area

     CONV_CRITERION: FORMULA_NUMBER;
     FORMULA_SELECTION: COMBINATION_DESCRIPTOR;

-- Darin ist

type FORMULA_NUMBER is INTEGER range 1 .. 32

Mit den angegebenen Steuerworten werden die Operationen
-- SCAN bzw. CONVERT wie bereits beschrieben definiert.

-- Mit den gegebenen Verknüpfungsmöglichkeiten können auch
-- einzelne Vektoren untersucht werden, um die Anzahl des
-- Auftretens bestimmter Bedingungen bzw. die Position des
-- ersten Auftretens zu bestimmen.
```

```
-- Die Steuerworte dafür sind:

type TEST_CRITERION_1 is record area
     CRIT_SEL_1: (ORTHOGONALITY, ORTHOGONALIZATION_POSSIBLE,
                  EQUALITY, UNEQUALITY, NEGATION, N_ELEMENT,
                  NO_N_ELEMENT, L_ELEMENT, NO_L_ELEMENT,
                  H_ELEMENT, NO_H_ELEMENT);

type TEST_CRITERION_2 is record area
     CRIT_SEL_2: FORMULA_NUMBER;
     FORMULA_SELECTION: COMBINATION_DESCRIPTOR;

-- Damit ergeben sich die Operationen in allgemeiner
-- Schreibweise:

functions NUMBER_OF_OCCURRENCES depending on TEST_CRITERION
          (P1, P2: TVEC; TAG_VECTOR, MASK_VECTOR: BVEC)
          return FINITE_CARDINAL;

functions FIRST_OCCURRENCE depending on TEST_CRITERION
          (P1, P2: TVEC; TAG_VECTOR, MASK_VECTOR: BVEC)
          return FINITE_ORDINAL;

-- Jede Operation ist für beide Test- Kriterien definiert.
-- Insgesamt gibt es folgende Modifikationen bei der
-- Parameter- Übergabe:
-- 1.: 2 Ternärvektoren, Maskenvektor, Markierungsvektor
-- 2.: 1 Ternärvektor, Maskenvektor, Markierungsvektor
-- 3.: 2 Ternärvektoren, Markierungsvektor
-- 4.: 1 Ternärvektor, Markierungsvektor
-- 5.: 2 Ternärvektoren, Maskenvektor
-- 6.: 1 Ternärvektor, Maskenvektor
-- 7.: 2 Ternärvektoren
-- 8.: 1 Ternärvektor
-- Weiterhin kann statt des ersten Ternärvektors der Typ
-- TVL_ROW definiert sein.
-- Insgesamt gibt es somit 32 entsprechende Befehle.
```

```
-- Es ist möglich, für orthogonale Ternärvektorlisten
-- Lösungszahlen zu berechnen und verschiedene Lösungszahlen
-- miteinander zu vergleichen.
-- Es wird zunächst die Lösungszahl als Datenstruktur ein-
-- geführt:

type NUMBER_OF_SOLUTIONS is INTEGER range 0 .. 2 exp 4096;

-- Die Operationen sind:
function COMPUTE_NOS (P1: TVL_AREA) return NUMBER_OF_SOLUTIONS;

function COMPARE_NOS (P1, P2: NUMBER_OF_SOLUTIONS)
                      return COMPARE_CRITERION;

-- COMPARE_CRITERION: (LESS, EQUAL, GREATER)
-- Dieses ist mit Testbefehlen abfragbar.

function TEST_NOS (P1: NUMBER_OF_SOLUTIONS) return CONDITION;
-- CONDITION = TRUE, wenn die Norm der BOOLEschen Gleichung
-- größer oder gleich 0,5 ist.

-- Für das Umbauen existierender Vektoren bzw. Listen
-- gibt es drei prinzipielle Operationen:
-- 1.: Aufbau des Resultats aus selektierten Positionen
--     des Arguments (GATHER)
-- 2.: Aufbau des Resultats derart, daß in die selektier-
       ten Positionen des ersten Arguments nacheinander
       die aufeinanderfolgenden Positionen des zweiten
       Arguments eingefügt werden, wobei die betreffenden
       ursprünglichen Positionen des ersten Arguments zu
       den jeweils nächsten (höherwertigen) Stellen hin
       verdrängt werden (EXPAND)
--3.: Ersetzen der selektierten Positionen des ersten
      Arguments durch die aufeinanderfolgenden Positionen
      des zweiten (INSERT).
```

```
-- Zur Selektion (Argument SELECTION_CONTROL) kann wahl-
-- weise ein Markierungsvektor oder eine Indexliste ver-
-- wendet werden.
-- Beispiele für entsprechende Operationen sind:

function GATHER (P1: TVL_AREA, SELECTION_CONTROL: INDEX_LIST)
         return TVL_AREA;

function EXPAND (P1, P2: TVL_AREA, SELECTION_CONTROL:
                 INDEX_LIST) return TVL_AREA;

function INSERT (P1, P2: TVL_AREA, SELECTION_CONTROL:
                 INDEX_LIST) return TVL_AREA;

-- Die Operationen sind weiterhin definiert für Argumente
-- der Typen: BVL_AREA, TVEC, BVEC.
-- Insgesamt gibt es 24 entsprechende Befehle.

-- Die Argumente zur Selektion können ineinander umge-
-- wandelt werden:

function VECTOR_TO_INDEX (P1: BVEC) return INDEX_LIST;
function INDEX_TO_VECTOR (P1: INDEX_LIST) return BVEC;

-- Binärvektoren können invertiert werden:
function INVERT (P1: BVEC) return BVEC;

-- Weiterhin können Indexlisten aus Variablenlisten er-
-- zeugt werden, und aus einer Variablenliste läßt sich
-- mittels einer Indexliste eine weitere Variablenliste
-- aufbauen.
```

```
function VAR_TO_INDEX (P1, P2: VARLIST) return INDEX_LIST;

-- P1 ist die ursprüngliche Variablenliste, P2 die
-- Liste der Variablen, für die die Indexliste zu er-
-- zeugen ist.

function INDEX_TO_VAR (P1: INDEX_LIST) return VARLIST;

-- Binäre oder ternäre Vektoren können erzeugt werden:

function BINARY_VECTOR (LENGTH: FINITE_CARDINAL,
                        FILLER: BOOLEAN) return BVEC;

function TERNARY_VECTOR (LENGTH: FINITE_CARDINAL,
                         FILLER: TERNARY) return TVEC;

-- Weiterhin kann ein Indexwert und ein Belegungswert
-- angegeben werden, der in der betreffenden Position
-- den FILLER ersetzt:

function BINARY_VECTOR (LENGTH: FINITE_CARDINAL,
                        INDEX: FINITE_ORDINAL,
                        VALUE, FILLER: BOOLEAN) return BVEC;

function TERNARY_VECTOR (LENGTH: FINITE_CARDINAL,
                         INDEX: FINITE_ORDINAL,
                         VALUE, FILLER: TERNARY) return TVEC;

-- Binäre bzw. ternäre Listen können auf ähnliche Weise
-- erzeugt werden:

function BINARY_LIST (VARS: FINITE_CARDINAL,
                      SIZE: TRANSFINITE_CARDINAL,
                      FILLER: BOOLEAN) return BVL_AREA;

function TERNARY_LIST (VARS: FINITE_CARDINAL,
                       SIZE: TRANSFINITE_CARDINAL,
                       FILLER: TERNARY) return TVL_AREA;
```

```
-- Eine "leere" Liste kann wie folgt erzeugt werden:
function CREATE_BVL (VARS: FINITE_ORDINAL) return BVL_AREA;
function CREATE_TVL (VARS: FINITE_ORDINAL) return TVL_AREA;

-- Die Listen haben zunächst die Zeilenzahl 0.
-- Sinngemäß können "leere" Vektoren (Variablenzahl 0)
-- erzeugt werden:
function CREATE_BVEC return BVEC;
function CREATE_TVEC return TVEC;

-- Um Listen bzw. Vektoren aus ihren Komponenten zusammen-
-- zufügen, ist die Operation CONCATENATE (CAT) vorgesehen.
-- Es wird zunächst eine Variante definiert:
function CAT (LIST: TVL_AREA, VECTOR: TVEC) return TVL_AREA;

-- Generell gibt es folgende Kombinationen von Argument- Typen:
-- Liste/Liste, Liste/Vektor, Liste/Zeile (z. B. TVL_ROW),
-- Vektor/Variable, Vektor/Variable in Vektor
-- Dies gilt sowohl für ternäre als auch für bonäre
-- Argumente. Somit sindinsgesamt 10 entsprechende Befehle
-- vorgesehen.
-- Weiterhin gibt es eine CAT- Operation für Variablenlisten:
function CAT (LIST1, LIST2: VARLIST) return VARLIST;

-- Aus Listen können Zeilen ausgewählt werden:
function SEL_ROW (LIST: TVL_AREA, ROW: TRANSFINITE_ORDINAL)
                  return TVL_ROW;

-- Sinngemäß können aus Vektoren Variable ausgewählt werden:
function SEL_VAR (VECTOR: TVEC, POS: FINITE_ORDINAL)
                  return VAR_IN_TVEC;
```

```
-- Analoge Operationen gibt es auch für binäre Listen bzw.
-- Vektoren.

-- Die einzelnen Formen der Darstellung BOOLEscher
-- Gleichungen können ineinander überführt werden:
function FORMULA_TO_TVL (P1: FORMULA) return TVL;
function TVL_TO_FORMULA (P1: TVL) return FORMULA;

-- TVL können aus Erlaubnis- und Wertlisten erzeugt
-- werden:
function MERGE_ENABLE_VALUE (P1, P2: BVL_AREA) return TVL_AREA;

-- Weiterhin können TVL durch "Auflösen" der N- Belegungen
-- in Binärvektorlisten gewandelt werden:
function TVL_TO_BVL (P1: TVL_AREA) return BVL_AREA;

-- Ebenso ist die umgekehrte Wandlung vorgesehen:
function BVL_TO_TVL (P1: BVL_AREA) return TVL_AREA;

-- Für das Erzeugen von Verknüpfungsdeskriptoren sind
-- spezielle Operationen vorgesehen:
function CREATE_DESCRIPTOR return COMBINATION_DESCRIPTOR;
-- Damit wird ein "leerer" Deskriptor aufgebaut.
function ADD_FORMULA (P1: COMBINATION_DESCRIPTOR,
                      P2: TVL) return COMBINATION_DESCRIPTOR;
-- Damit wird eine durch eine TVL gegebene BOOLEsche
-- Gleichungdem betreffenden Deskriptor hinzugefügt
-- (ein Deskriptor kann bis zu 32 Gleichungen aufnehmen).
```

```
-- Listen können auch durch Zusammenfügen von Spalten
-- gebildet werden:
function ADD (P1, P2: TVL_AREA) return TVL_AREA;
function ADD (P1, P2: TVL) return TVL;
-- Bei der letzteren Operation werden die Variablenlisten
-- ebenfalls zusammengefügt (unter Berücksichtigung
-- von Variablen, die in beiden Argument- Listen vorkommen).

-- Aus Listen bzw. Vektoren können Teile selektiert werden:
function SUBLIST (LIST: TVL_AREA, FIRST_ROW: TRANSFINITE_ORDINAL,
                  ROWS: TRANSFINITE_CARDINAL) return TVL_AREA;
function SUBVEC (VECTOR: TVEC, FIRST_VAR: FINITE_ORDINAL,
                 VARS: FINITE_CARDINAL) return TVEC;
-- Derartige Operationen sind auch für binäre Listen bzw.
-- Vektoren vorgesehen.

-- Für den Umgang mit Ableitungen (im Sinne des "BOOLEschen
-- Differentialkalküls") sind folgende elementare Operationen
-- vorgesehen:

function ADD_DERIVATION (LIST: TVL, TAGVEC: BVEC) return TVL;
-- Für die durch den Markierungsvektor gekennzeichneten
-- Variablen wird in der Variablenliste jeweils eine neue
-- Position mit erhöhtem (um 1) DERIVATION_CODE angefügt.
-- Die eigentliche TVL wird um die entsprechende Anzahl an
-- Spalten erweitert, wobei diese Spalten mit N- Elementen
-- belegt werden.
```

```
function SWAP_DERIVATION (LIST: TVL, TAGVEC: BVEC)
                                return TVL_AREA;

-- Für die im Markierungsvektor gekennzeichneten Variablen
-- werden in der Variablenliste die abgeleiteten Variablen
-- gesucht. Die Belegungen der entsprechenden Spalten in der
-- eigentlichen TVL werden untereinander getauscht.
-- Der Algorithmus "Verschiebung/Köpplung" wäre also
-- wie folgt zu formulieren:
-- CAT ( SWAP_DERIVATION ( ADD_DERIVATION ( TVL, TGVECTOR),
        TGVECTOR))

-- Weitere Operationen mit Ableitungen sind:

function DELETE_DERIVATION (LIST: TVL, TAGVEC: BVEC,
                            DERIVATION_CODE: INTEGER range
                            1 .. 15) return TVL;

-- In der resultierenden TVL fehlen die Spalten, deren
-- Variable sowohl durch den Markierungsvektor gekenn-
-- zeichnet sind als auch den entsprechenden DERIVATION_CODE
-- haben.
-- Eine weitere Operation liefert eine TVL, die genau jene
-- Spalten enthält:

function EXTRACT_DERIVATION (LIST: TVL, TAGVEC: BVEC,
                             DERIVATION_CODE: INTEGER range
                             1 .. 15) return TVL;

-- In Variablenlisten läßt sich der DERIVATION_CODE in
-- markierten Positionen auf Null zurückstellen:

function  ZERO_DERIVATION (P1: VARLIST, TAGVEC: BVEC)
                                return VARLIST;

-- Zum "Streichen von Lösungspaaren in Unterräumen"
-- dient folgende Operation:

function DELETE_TAGGED_ROWS (LIST: TVL_AREA, TAGVEC: BVEC)
                                return TVL_AREA;
```

```
-- In der resultierenden TVL fehlen alle Zeilen, in
-- denen an wenigstens einer markierten Position ein
-- N̸ Element vorkommt.

-- Komplette TVL können mit folgender Operation aus ihren
-- Bestandteilen zusammengesetzt werden:

function COMPOSE_TVL (LIST_PROPER: TVL_AREA, VAR_ASSIGNMENT:
                      VARLIST, ATTRIBUTES: FINITE_ORDINAL)
                      return TVL;

-- Logische Verknüpfungen sind zwischen den Variablen
-- eines binären bzw. ternären Vektors möglich.
-- Dabei wird ein Formelausdruck abgearbeitet (es kann sich
-- sowohl um binäre als auch um ternäre Verknüpfungen handeln).

function COMBINE_IN_VECTOR (VECTOR: TVEC, EXPRESSION:
                            FORMULA_EXPRESSION) return TERNARY;

-- Die Varianten sind im einzelnen:
-- ternärer Vektor/ternäres Resultat, ternärer Vektor/binä-
-- res Resultat und binärer Vektor/binäres Resultat
-- Somit ergeben sich insgesamt 4 Befehle.
```

3.4. Organisatorische Operationen (Ausschnitt)

Im folgenden werden wesentliche organisatorische Operationen in verbaler Darstellung (ohne formale Deklaration) angeführt:

a.) Befehle für die Programm- Organisation

COPY_ART: Kopieren der ART des betreffenden Programms auf den Stack als activation record des zu rufenden Programms (mit entsprechender Verwaltung der "stack frame pointer" usw.)

ENTER: Übergang zum gerufenen Programm

TERMINATE: Abbruch des aktuellen Programms und Rückkehr zum rufenden Programm (im Sinne einer Unterprogramm- Rückkehr)

CANCEL: Abbruch des aktuellen Programms und Rückkehr zum Ausgangs- Niveau

END: Vollständiges Beenden eines Programmlaufs (Rückgabe der Steuerung an den "Scheduler" des Betriebssystems)

CONTINUE: Übergang zum "loop body" in einem Iterator- Ausdruck

BREAK: Abbruch der Schleifen- Ausführung in einem Iteratorausdruck (Übergang zum nächsten Befehl nach dem betreffenden YIELD)

OPEN_LAMBDA: Eröffnen eines Lambda- Ausdruckes: die nachfolgenden Befehle werden Bestandteil eines Lambda- Ausdruckes (sie konstituieren diesen zusammen mit den aktuellen Parametern)

CLOSE_LAMBDA: Abschließen eines Lambda- Ausdruckes. Der Lambda- Ausdruck ist gegeben durch die Befehlsfolge zwischen einem Befehl OPEN_LAMBDA und einem Befehl CLOSE_LAMBDA sowie durch die aktuellen Parameter.

CAUSE: Auslösen eines Ereignisses (Ereignisse sind spezielle Objekte zum Starten bzw. Synchronisieren von Prozessen)

WAIT: Warten auf das Eintreten bestimmter Ereignisse

SWITCH: Zuteilung von Laufzeit an andere Prozesse

CAUSE_AND_TERMINATE: Beenden des aktuellen Prozesses und Auslösen eines Ereignisses

HI_BOUND: Liefern eines Wertes, der die Obergrenze transfiniter Kardinalzahlen anzeigt

RELEASE_ID: Liefern eines Wertes, der Nummer und Ausgabedatum des aktuellen Standes der Architektur- Implementierung anzeigt

MACHINE_ID: Liefern eines Wertes, der den Typ der zugrunde liegenden .008- Hardware, die Seriennummer sowie einen "feature code" anzeigt, aus dem die Ausstattung der Hardware ersichtlich ist (z. B., welche Klassen von Operationen in Hardware ausgeführt werden, welche Geschwindigkeiten zu erwarten sind usw.).

Hinweis: Mit den letztgenannten drei Befehlen kann von Anwendungsprogrammen aus geprüft werden, ob die aktuelle Hardware für deren Ausführung geeignet ist (ausreichend ausgestattet, ausreichend schnell usw.).

b.) Rechnen mit natürlichen Zahlen

Die vier Grundrechenarten sind sowohl für finite als auch für transfinite Zahlen vorgesehen. Dabei wird die allgemeine Bedingung (CONDITION) gesetzt, wenn

- bei der Addition finiter Zahlen der Wert 4096 überschritten wird,
- bei der Addition transfiniter Zahlen der Wert $2^{32}-1$ überschritten wird,

- bei einer Subtraktion ein negatives Resultat entsteht,
- bei einer Division ein Rest entsteht (dieser ist ebenfalls für die weitere Verarbeitung zugänglich),
- bei einer Multiplikation die Werte 4096 bzw. 2^{32}-1 überschritten werden (das Gesamt- Resultat steht in jedem Fall für die weitere Verarbeitung zur Verfügung).

c.) Abfragen von Bedingungen

Verzweigungsbefehle (BRANCH) können lediglich eine einzige allgemeine Bedingung (CONDITION) auswerten. Bei manchen Operationen wird die Bedingung gemäß dem jeweiligen Resultat gestellt (die genaue Darstellung bleibt künftigen Beschreibungen der .008- Architektur vorbehalten).
Weiterhin ist es möglich, bestimmte Bedingungs- Aussagen über Testbefehle abzufragen (CONDITION wird aktiv, wenn die abgefragte Bedingung zutrifft). Beispiele für Testbefehle:

```
TEST_ARITHMETIC_EQUAL
                _UNEQUAL
                _LESS
                _GREATER
                _LESS_OR_EQUAL
                _GREATER_OR_EQUAL
```

Damit können sowohl finite als auch transfinite Zahlen miteinander verglichen werden.

TEST_BOOLEAN: die Bedingung entspricht dem Wert der betreffenden BOOLEschen Variablen (die z. B. durch einen Selektor ausgewählt wurde)

```
TEST_TERNARY_N
            _L
            _H
            _I
            _NOT_N
            _NOT_L
            _NOT_H
            _NOT_I
```

Die Bedingung ist dabei erfüllt, wenn die betreffende ternäre Variable dem Testkriterium genügt.

TEST_ORTHOGONALITY: Prüfen einer TVL_AREA auf Orthogonalität

TEST_EMPTY: Prüfen, ob das betreffende Objekt "leer" ist (z. B. eine TVL mit Zeilenzahl 0)

TEST_DONT_CARE: Prüfen, ob eine gegebene TVL_AREA durch eine "Strichzeile" repräsentiert wird

TEST_NOS_EQUAL
_UNEQUAL
_LESS
_GREATER
_LESS_OR_EQUAL
_GREATER_OR_EQUAL

Damit kann das Resultat einer Operation COMPARE_NOS getestet werden.

TEST_OBJECT: Es werden zwei Objekte daraufhin verglichen, ob sie gleichen Typ und gleichen Wert haben.

TEST_ZERO: Die Bedingung wird aktiviert, wenn der Wert des betreffenden Objekts ausschließlich durch binäre Nullen repräsentiert wird.

Weitere (hier nicht näher beschriebene) Testbefehle sind zur Abfrage von Typen, Attributen und anderen Kennzeichen von Objekten vorgesehen.

d.) Iteratoren (Auswahl)

NEXT_ROW: liefert die nächste Zeile einer Liste (Abbruch nach Lieferung der letzten Zeile)

NEXT_VAR: liefert die nächste Variable eines Vektors (Abbruch nach Lieferung der letzten Variablen)

NEXT_OBJECT: liefert das nächste Objekt einer Verbundobjekttabelle CORT (Abbruch nach Lieferung des letzten Objekts)

4. Sonstige Vorkehrungen und weitere Probleme

a.) Ausnahmebehandlung

Ausnahmebedingungen werden bei der Ausführung mancher Befehle wirksam, wenn dabei Inkorrektheiten festgestellt werden (z. B. Argumente mit inkorrektem Typ, Argumente, die hinsichtlich ihrer Größe nicht zusammenpassen usw). Die genaue Beschreibung bleibt künftigen Darstellungen der .008- Architektur vorbehalten. Weiterhin können Ausnahmebedingungen explizit mit Befehlen RISE EXCEPTION signalisiert werden. Grundsätzlich gilt, daß zunächst in der ART des auslösenden Programms nach einem Zeiger für den entsprechenden Behandler gesucht wird (dieser wird vom Compiler auf Grund einer on- Anweisung dort eingetragen). Wird kein solcher Zeiger gefunden, so werden alle aktiven Programme des betreffenden Prozesses in der Reihenfolge ihres Aufrufs daraufhin durchsucht. Wird dabei auch kein Zeiger gefunden, so wird eine systeminterne Behandlung aktiviert (in der Regel: Aufzeichnung der Situation und Abbruch des Prozesses).

b.) Realzeitbetrieb

Die Architektur schließt Vorkehrungen für Realzeitbetrieb und Parallelverarbeitung mehrerer Prozesse ein (vgl. Abschnitt 3.4. a.)). Dies ist im wesentlichen dafür vorgesehen, einen "multi user"- Betrieb (z. B. von mehreren Terminals aus) zu ermöglichen. Die Prozeßumschaltung ist ereignisgesteuert (ein auftretendes Ereignis, z. B. durch eine Bedienhandlung an einem Terminal bewirkt, startet den zugeordneten Prozeß). Für die Laufzeitverteilung ist eine einfache "round robin"- Strategie (zyklisches Weiterschalten zwischen den aktiven Prozessen) vorgesehen.

c.) Speicherverwaltung

Die Speicherverwaltung ist für die Architektur grundsätzlich transparent. Objekte befinden sich entweder komplett in Speichermitteln der jeweiligen Hardware, oder sie sind auf Externspeicher ausgelagert. Vor der Ausführung einer jeden Operation wird gewährleistet, daß die benötigten Objekte in den jeweiligen Speichermitteln bereitgestellt werden.
Dabei wird gleichzeitig ein zusammenhängender Freispeicherbereich für die Resultate bereitgestellt. U. U. sind Objekte, die nicht unmittelbar benötigt werden, zuvor auszulagern.
Die Strategie des Auslagerns ist implementierungsspezifisch (und Gegenstand der Erprobung). Wesentlich ist die ausschließliche Orientierung an Objekten und nicht (im Gegensatz zu üblichen Virtualspeicher- Organisationen) an Adressenbereichen. Konsequenz: Manche Probleme können auf Implementierungen der .008- Architektur nicht bearbeitet werden, wenn die Speicherausstattung nicht ausreichend ist (diese kann programmseitig abgefragt werden); bei einer üblichen Virtualspeicherorganisation wäre dies nicht der Fall (genügend Externspeicher- Kapazität vorausgesetzt), die Rechenzeiten würden sich jedoch infolge der häufigen Seitenwechsel intolerabel verlängern.
Die Objekttabellen (ORT) können implementierungsspezifisch um objektspezifische Steuerinformation für die Speicherverwaltung erweitert sein; ebenso können manche .008- Implementierungen dafür spezifische Hardwaremittel bereitstellen.

d.) Externkommunikation

Das System .008 ist für die Externkommunikation auf die Zusammenarbeit mit konventionellen Rechnersystemen angewiesen. Prinzipiell werden Objekte (einschließlich deren deskriptiver Information) über Interfaceanschlüsse bzw. gemeinsam gekoppelte Externspeicher ausgetauscht.

Das Starten bzw. Beenden der Zusammenarbeit wird aus der Sicht der .008- Architektur ausschließlich über Ereignisse gesteuert. Das prinzipielle Schema der Zusammenarbeit ist folgendes:

1. Der Universalrechner bereitet die Quell- Objekte für das jeweilige Problem auf und übermittelt diese an die .008- Implementierung.
2. Der Universalrechner löst ein Ereignis aus, um den Abarbeitungsprozeß im System .008 zu starten (das Ereignis selbst beschreibt den Prozeß an sich, das erste zu startende Programm usw.; es ist selbst ein Objekt, das zuvor übertragen wurde).
3. Das System .008 bearbeitet den übergebenen Auftrag und erzeugt dabei die Resultat- Objekte.
4. Nach der Bearbeitung des Auftrags wird im System .008 ein Ereignis wirksam (Auslösung durch CAUSE- Befehl). Dieses bewirkt eine Fertigmeldung an den Universalrechner.

Die Gestaltung der physischen Kommunikation ist implementierungsspezifisch (abhängig von Typ und Konfiguration des Universalrechners sowie von der Implementierung des System .008).

e.) Fehlersuchhilfen

Manche .008- Implementierungen können spezifische Bedien- und Anzeigemittel haben, mit denen direkte Eingriffe in die Hardware möglich sind (Anzeigen und Ändern von Speicherinhalten, Vergleichsstop usw.).
In der Architektur sind verschiedene Typen von Ereignissen und zusätzliche Befehle vorgesehen, die Eingriffe in laufende Programme ermöglichen. Zur Bedienung und Anzeige ist dabei ebenfalls eine Zusammenarbeit mit dem gekoppelten Universalrechner erforderlich.
Charakteristisch ist, daß durch spezielle Ereignisse besondere Monitor- Prozesse aktiviert werden können, von denen aus Eingriffe in andere Prozesse möglich sind (die Monitor- Prozesse erhalten vorrangig Laufzeit und können

die Laufzeit- Zuteilung zu den zu überwachenden Prozessen steuern). Typische Funktionen sind:

- Anhalten/Fortsetzen des zu überwachenden Prozesses
- Anzeigen/Ändern von Objekten (sowohl Werte als auch deskriptive Angaben)
- Anzeigen/Ändern der aktuellen Prozeßumgebung
- Registrierung von Laufzeit bzw. Benutzungshäufigkeit bestimmter Befehle
- Registrierung (im Sinne von "Vergleichsstop" der Benutzung bestimmter Befehle oder des Zugriffs zu bestimmten Objekten (so, daß ein "schrittweises" Abarbeiten von Programmen möglich ist).

In Abhängigkeit von der Ausstattung der .008- Implementierung kann diese Überwachung die Laufzeit mehr oder weniger verlängern.

f.) Typenkontrolle zur Laufzeit und Operationsvorrat

Diese Aspekte sind für die weitere Bearbeitung der .008- Architektur von besonderer Bedeutung.
Für die Typenkontrolle gibt es zwei Alternativen:

1. Strikte Typenkontrolle: Zur Ausführung einer Operation müssen die Argumente exakt der jeweiligen Typenspezifikation entsprechen.
2. "Default"- Verhalten: Entspricht ein angegebenes Argument-Objekt nicht exakt der Typenspezifikation, so wird ein Argument mit dem geeigneten Typ wie folgt bereitgestellt:
   a.) als Komponente des Objekts, wenn die Komponente exakt den erforderlichen Typ hat und die einzige Komponente dieses Typs ist
   b.) durch eine implizite YIELD- Operation, wenn so ein Argument des benötigten Typs erzeugt werden kann.

Die Alternative 1. bedeutet dabei nicht, daß zur Laufzeit exzessive Typenkontrollen stattfinden (das kann zur Compilierzeit erledigt werden), sondern im Gegenteil, daß jeder Operationsbefehl die Argumente ohne weitere

Prüfung, Umrechnung usw. sofort benutzen kann.

Vorteil von 1.: Effizienz zur Laufzeit, was den eigentlichen Operationsbefehl angeht (dessen Laufzeit wird nicht durch "default"- Maßnahmen verlängert.

Nachteile von 1.: a.) Die Anzahl der Operationscodes wird sehr groß. Im Sinne dieses Prinzips reichen die in Abschnitt 3.3. definierten Operationen keinesfalls aus (für weitere interessante Argument- Typen wären zusätzliche Operationen - mit prinzipiell jeweils gleicher Wirkung- zu deklarieren).

b.) Kommt es oft vor, daß die Argumente Komponenten anderer Objekte sind, so müssen jeweils Selektoren vorgesehen werden (Speicherbedarf und Auswertungszeit).

Mit der Einführung von 2. könnte die Vielfalt der Deklarationen gem. Abschnitt 3.3. deutlich eingeschränkt werden.

Beispiel: Für eine Operation sei ein Ternärvektor (TVEC) als Argument angegeben. Soll mit dieser Operation eine Zeile aus einer TVL (TVL_ROW) verarbeitet werden, so wäre bei Variante 1. erforderlich:

- eine weitere Operation mit dem Argumenttyp TVL_ROW einzuführen

  oder

- dem Operationsbefehl einen YIELD- Befehl für den TVL_ROW- Selektor voranzustellen.

Hingegen würde bei Variante 2. der YIELD- Ablauf implizit ausgeführt werden.

Ebenso bedarf die Erweiterung (bzw. Einschränkung) des Operationsvorrates einer eingehenden Untersuchung.
Zum einen muß der Operationsvorrat reichhaltig genug sein, um zusammen mit den anderen Befehlen, Steueworten usw. alle in Frage kommenden Anwendungs- Algorithmen formulieren zu können.
Zum anderen ist der Operationsvorrat soweit wie möglich einzuschränken, damit die Implementierung und Verifizierung der Architektur auch praktisch gewährleistet ~~sind~~.
Je umfassender der Operationsvorrat, um so größer die Aufwendungen und Schwierigkeiten bei Implementierung und Testung, besonders dann, wenn mehrere Implementierungen erforderlich sind (Software, spezielle Hardware verschiedener Ausgestaltung).
Je elementarer der Operationsvorrat, um so geringer die Effizienz, namentlich dann, wenn häufig vorkommende Abläufe durch eine Vielzahl von Befehlen realisiert werden müssen.
Unmittelbar einsichtige Kriterien für die Definition des Operationsvorrates sind:

1. Alle elementaren Algorithmen, die ganze Listen betreffen, sollten als Befehle vorgesehen werden.
2. Alle Algorithmen mit exponentiell oder polynomial wachsender Laufzeit sollten als Befehle vorgesehen werden, sofern sie ihrem Prinzip nach elementar sind und die Anzahl der Argumente überschaubar ist.
3. Es sollten alle die Abläufe als Befehle vorgesehen werden, für die sich wirtschaftliche Hardware- Realisierungen angeben lassen (unabhängig davon, ob diese in den bevorzugten Implementierungen vorhanden sind oder nicht).
4. Aus den gegebenen anderen Algorithmen sollten elementare Teilabläufe als Befehle vorgesehen werden, so daß sich die betreffenden Algorithmen mit überschaubaren Befehlsfolgen daraus formulieren lassen.

5. Sprachkonstrukte der Deklarationen (Übersicht)

array: kennzeichnet ein homogenes Verbundobjekt
record: kennzeichnet ein heterogenes Verbundobjekt
area: kennzeichnet ein homogenes zusammengesetztes Objekt
record area: kennzeichnet ein heterogenes zusammengesetztes Objekt

Hinweis: Zusammengestzte Objekte werden stets in der "dichtesten' maschineninternen Codierung dargestellt.

attr: kennzeichnet Attribute, die einem Objekt zur näheren Charakterisierung beigefügt werden.
Attribute werden als Aufzählung der einzelnen Merkmale als record area implementiert und direkt im jeweiligen Objektdeskriptor gespeichert.

depending on: kennzeichnet eine Befehlsmodifikation.
Es wird eine Klasse von Funktionen oder Prozeduren dargestellt, deren konkrete Wirkung durch ein Steuerwort bestimmt wird.
Das Steuerwort wird als record area deklariert, und zwar wahlweise gesondert oder unmittelbar im Anschluß an depending on.

Ansonsten entsprechen die Deklarationen der Programmiersprache Ada, so wie sie in /10/ dargestellt ist.Aus Ersparnisgründen wurden lediglich die Ende- Kennzeichnungen (z. B. end record) weggelassen.

6. Übersicht über interne Operationsprinzipien der 008.B-Hardware

6.1. Allgemeine Eigenschaften

Es handelt sich um eine Reihe von Spezialprozessoren für die Implementierung der .008- Architektur, wobei sich die einzelnen Modelle bei identischer Grundstruktur in der technologischen Realisierung, der Arbeitsgeschwindigkeit, der Speicherkapazität und in Details der Ausstattung voneinander unterscheiden.
Grundsätzlich besteht jeder der Spezialprozessoren aus einer "Architektur- Implementierungs- Maschine" (AIM) und einer "Algorithmen- Realisierungs- Maschine" (ARM) sowie aus Schaltmitteln zur Kommunikation mit Externspeichern und Universalrechnern (Bild 10). Alle diese Einrichtungen sind in sich programmierbar (bzw. mikroprogrammierbar) und über ein universelles (aber modellspezifisches) Bussystem untereinander verbunden.
Das primäre Mittel zur Implementierung der .008- Architektur ist die AIM, die ihrerseits einen auf übliche Weise programmierbaren Universalrechner darstellt, der mit den anderen Schalt- und Speichermitteln des Spezialprozessors verbunden ist. In den kleineren Modellen ist die AIM durch eine Mikrorechner- Anordnung üblicher Art realisiert, die mit entsprechender Speicher- und Peripherieausstattung (ROM, RAM, Floppy Disk, Tastatur, Bildschirm) auch die Initialisierung, Fehlerbehandlung, Testung usw. übernimmt. Bei größeren Modellen ist der Mikrorechner durch spezielle Hardware zur Speicherverwaltung, zur Beschleunigung der Objektzugriffe usw. erweitert bzw. die gesamte AIM ist als mikroprogrammiertes schnelles Steuerwerk ausgelegt (optimiert für die Aufgaben der Architektur-Interpretation).

Die ARM enthält die Speichermittel für die zu verarbeitenden Objekte sowie die Verknüpfungs- und Steuerschaltungen für die zeitkritischen Operationen.
Die entscheidende Einrichtung ist dabei ein "aktives Memory Array" (AMA), das neben Speichern relativ großer Kapazität (äquivalent zu mehreren MBytes) mehrere parallele Sucheinrichtungen (zum Durchmustern von Listen) enthält sowie verschiedene Arten von Zugriffen ermöglicht. Damit können die entscheidenden Such- und Umordnungsabläufe mit TVL jeweils mit maximaler Geschwindigkeit ausgeführt werden. Für die verbleibenden Such- bzw. Verknüpfungsoperationen sind zentralisierte Schaltmittel vorgesehen, wobei sich das Verhältnis von parallel mehrfach angeordneten und zentralisierten Schaltmitteln zwischen den einzelnen Modellen unterscheiden kann.
Prinzipiell ist die ARM so ausgelegt, daß elementare Abläufe stets die jeweils maximale Geschwindigkeit haben, d. h. gemäß der jeweils vorgesehenen Verarbeitungsbreite wird in jedem internen Zyklus ein entsprechender Beitrag zum Resultat geliefert (die Steuer- und Adressierungsschaltungen sind so ausgelegt, daß keine Zeitverluste durch Organisations- oder Entscheidungsprozesse o. dergl. eintreten). Auf diese Weise werden die leistungsentscheidenden Abschnitte der wesentlichen Befehle gesteuert (z. B. Transporte von Listen, Durchmustern usw.).
Die Befehle selbst werden von der AIM interpretiert.
Bei den kleineren Modellen veranlaßt die AIM die Ausführung eines entsprechenden Befehls dadurch, daß eine Folge elementarer Aufträge an die ARM gegeben wird. In leistungsfähigeren Modellen verfügt die ARM über weitere Steuermittel, die es ermöglichen, nach Übergabe der aktuellen Parameter die betreffenden Befehle (z. B. eine Durchschnittsbildung zweier TVL) autonom auszuführen, so daß die AIM parallel dazu die Ausführung der Folgebefehle vorbereiten kann.

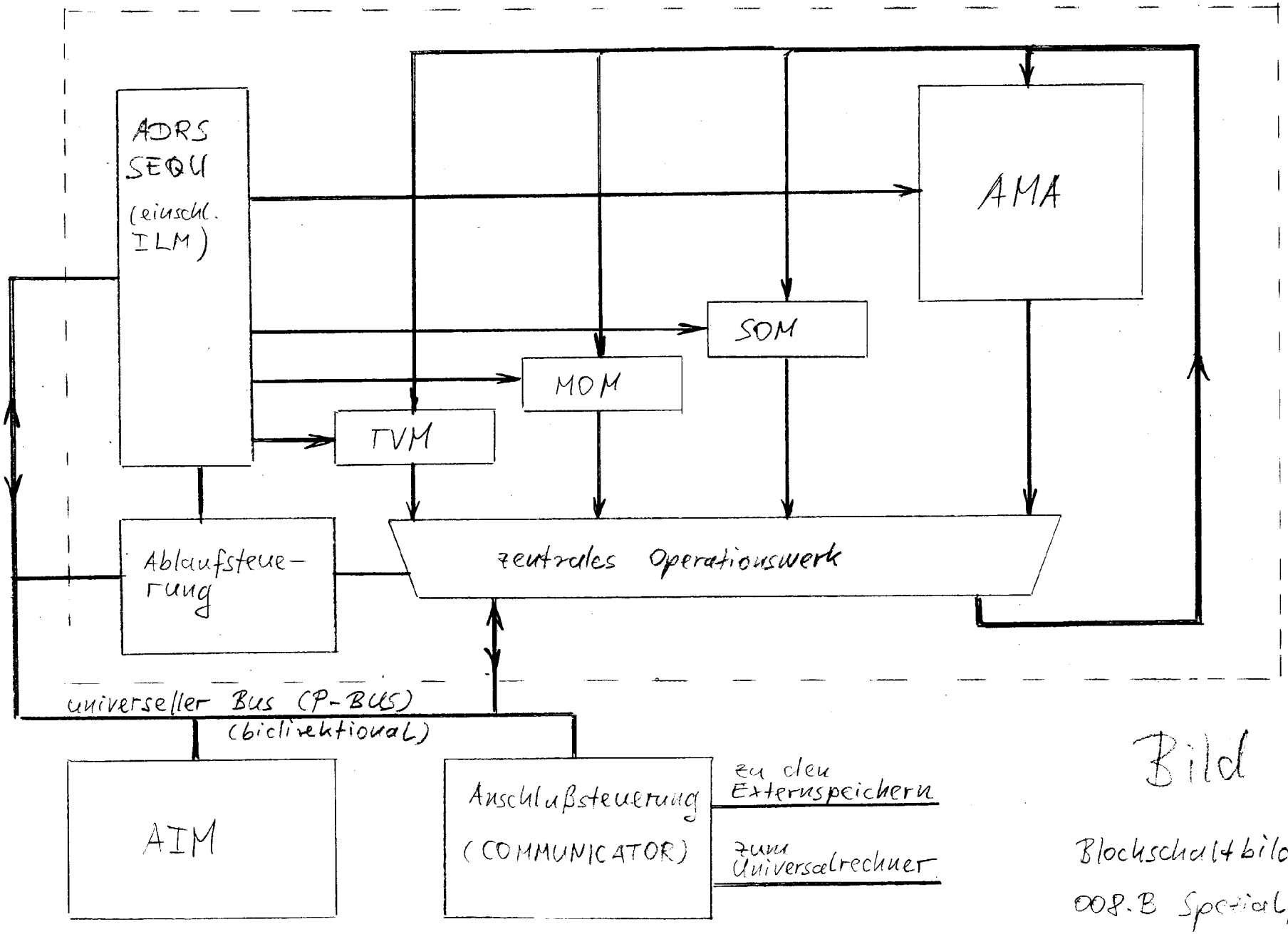


Bild 10

Blockschaltbild der OOB.B Spezialprozessoren

6.2. Übersicht über die Speichermittel der ARM

a.) ACTIVE MEMORY ARRAY (AMA)

Das AMA ist eine hinsichtlich der logischen Organisation matrixförmige Anordnung von Speicher-, Zugriffs- und Durchmusterungsschaltungen. Eine Reihe von Speichermitteln, die zusammen mit anderen Schaltungen hinsichtlich der logischen Organisation waagerecht nebeneinander angeordnet sind, wird als "Horizontalelement" bezeichnet. Entsprechend stellt ein "Vertikalelement" eine Reihe entsprechender Schaltmittel dar, die hinsichtlich der logischen Organisation senkrecht untereinander angeordnet sind.
Die grundlegende Einheit der Speicherung ist das Maschinenwort zu 32 Bit (bzw. 16 Ternärvariablen). Jedes Maschinenwort wird in einem Horizontalelement gespeichert.
Vektoren bzw. Zeilen von Listen sind Folgen von Maschinenworten auf fortlaufenden Positionen in einem Horizontalelement. Die einzelnen Zeilen einer Liste sind in aufeinanderfolgenden Horizontalelementen gespeichert.
Das AMA hat die Organisationsform "8x8" (8 Horizontalelemente x 8 Vertikalelemente, vgl. Bild 11). Die Speichermittel in jedem "Kreuzungspunkt" haben eine Aufrufbreiet von 2 Bit bzw. einer Ternärvariablen ("Variablenposition"). Zur Verarbeitung eines Maschinenwortes sind somit zwei Zugriffe notwendig (bei größeren Modellen können diese zeitlich ineinander "verschachtelt" sein: "interleaving"- Organisation der Speicher).
Die Speicherkapazität pro Horizontalelement reicht von 128k bis 1M Maschinenworte (Gesamtkapazität des AMA somit 4MBytes...32MBytes).
Zum AMA gibt es prinzipiell folgende Zugriffsmöglichkeiten:
a.) horizontal parallel: alle Horizontalelemente werden parallel adressiert, die Information wird gelesen und

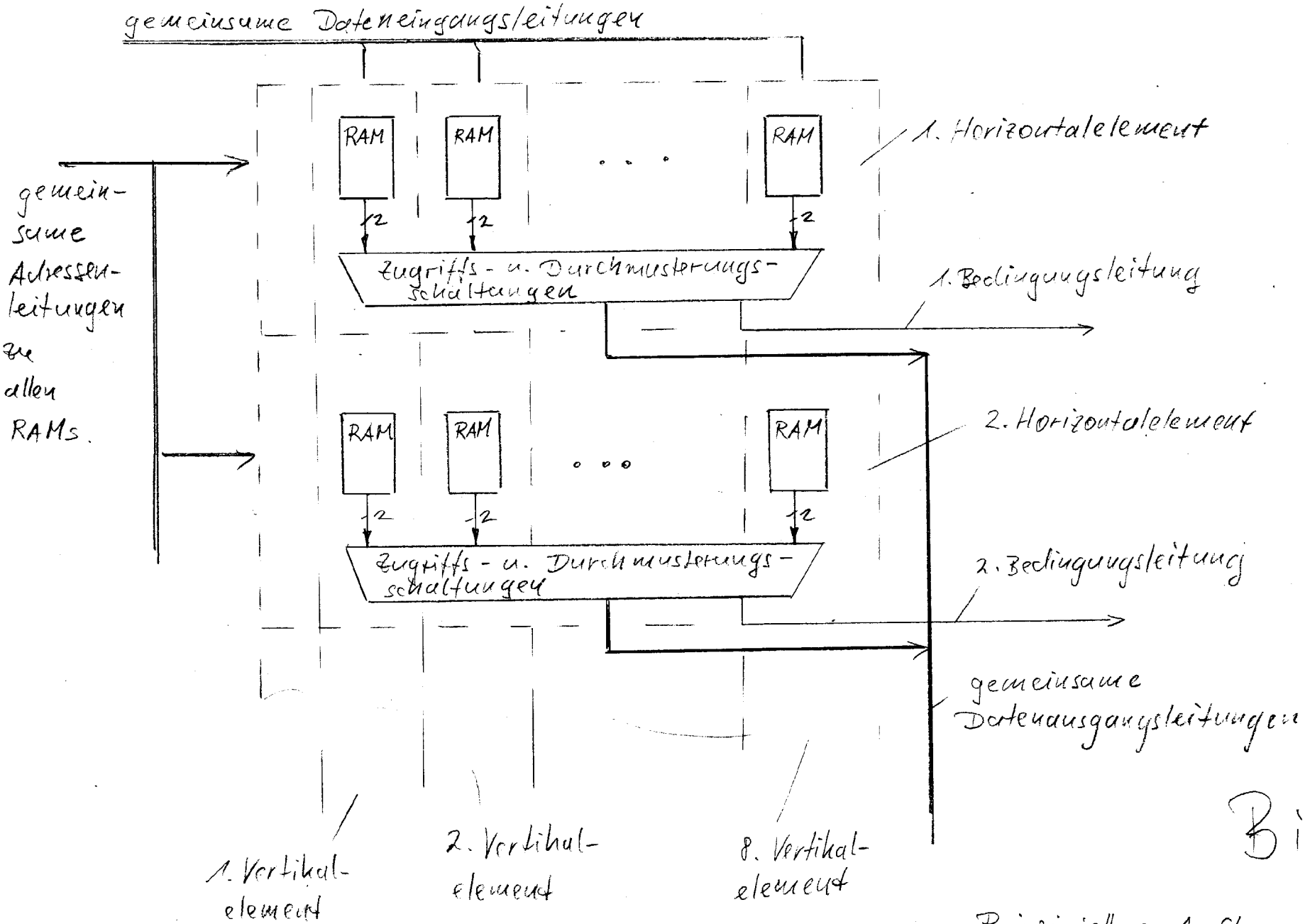


Bild 11

Prinzipieller Aufbau des AMA

von den parallel vorhandenen Durchmusterunsschaltungen ausgewertet.

b.) horizontal seriell: es wird ein Horizontalelement adressiert, so daß Lese- und Schreibzugriffe zu einzelnen Maschinenworten möglich sind

c.) vertikal: es wird ein Vertikalelement adressiert. Damit sind Zugriffe zu jeweils einer Variablenposition in 8 Zeilen einer Liste gleichzeitig möglich.

d.) variablenweise: durch gleichzeitiges Adressieren eines Vertikal- und eines Horizontalelements sind selektive Zugriffe zu einzelnen Variablen möglich.

Die Zykluszeiten der Zugriffe variieren zwischen 500 und 125ns (abhängig von der Art des Zugriffs und der Ausstattung des jeweiligen Modells).

b.) Zweitoperandenspeicher (SECOND OPERAND MEMORY SOM)

Der SOM hat eine Kapazität von zwei Ternärvektoren (jeweils maximal 4096 Variable).
Er verfügt weiterhin über Schaltmittel zur direkten Erzeugung fester Vektoren auf Grund vorgegebener Parameter (Länge, FILLER usw.). Derartige Vektoren brauchen also nicht gespeichert zu werden.

c.) Maskenoperandenspeicher (MASK OPERAND MEMORY MOM)

Der MOM hat eine Lese/Schreib- Kapazität von zwei Binärvektoren (jeweils maximal 4096 Variable) sowie eine ROM-Kapazität für 16 verschiedene Festvektoren zu jeweils maximal 256 Bit.

d.) Markierungsvektorspeicher (TAG VECTOR MEMORY TVM)

Der TVM hat eine Kapazität von zwei Binärvektoren (je 4096 Bit).

e.) Indexlistenspeicher (INDEX LIST MEMORY ILM)

Der ILM hat eine Kapazität von 4k Worten zu 16 Bit. Dies ist zur Aufnahme einer kompletten Indexliste maximal möglicher Länge ausreichend. Alternativ wird der ILM zur Speicherung von Formelausdrücken benutzt.

f.) Speichermittel für Parameter- Information

Es sind besondere Speichermittel vorgesehen, um die aktuellen Parameter der zu verarbeitenden Objekte zu halten. Es können jeweils 16 verschiedene Parameter gespeichert werden:

- AMA- Adresse: 20 Bit
- Horizontaladresse: 3 Bit
- Vertikaladresse: 3 Bit
- Endadresse: 16 Bit (wird zur Kontrolle der Maximalgröße eines Objekts mit den höchstwertigen 16 Bit der AMA- Adresse verglichen)
- Zeilenzahl: 24 Bit
- Variablenzahl: 13 Bit

Bei manchen elementaren Aktionen der ARM (z. B. Transporte, Durchmusterungen usw.) können mehrere aufeinanderfolgende Positionen der Parameterspeicher belegt werden (z. B.: Anfangsadresse der Liste, Anfangsadresse der aktuellen Zeile, Adresse des aktuellen Maschinenwortes).
Die Auswahl des aktuellen Vektors in den Vektoroperandenspeichern (MOM, SOM, TVM) erfolgt über Steuerregister.

6.3. Objekttypen der ARM

a.) Variable

Es gibt binäre und ternäre Variable. Eine Variable ist alternativ:
- ein einzelnen Objekt
- Teil eines Vektors
- Teil einer Liste

Einzelne Objekte bzw. Listen werden stets im AMA gespeichert. Als Teil eines Vektors kann eine Variable auch in einem der Vektoroperandenspeicher selektiert werden (nur Lesen; selektives Schreiben ist nur mit dem AMA möglich).

b.) Vektoren

Es gibt binäre und ternäre Vektoren. Diese können im AMA oder in einem der Vektoroperandenspeicher gespeichert sein. Parameter: Horizontaladresse, AMA- Adresse, Variablenzahl. Vektoren werden stets in abgeschlossenen Maschinenworten gespeichert. Freie "Restpositionen" im jeweils letzten Maschinenwort eines Vektors werden automatisch (ohne explizit angegebenen Maskenvektor) bei der Verarbeitung maskiert.

c.) Listen

Es gibt binäre und ternäre Listen. Sie werden nur im AMA gespeichert, wobei aufeinanderfolgende Vektoren (Zeilen) aufeinanderfolgende Horizontalelemente belegen.
Parameter: AMA- Adresse, Variablenzahl, Zeilenzahl.
(Das "Auffüllen" auf ganze Maschinenworte wird bei den Zugriffen automatisch berücksichtigt.)
Sonderformen von Listen sind:
- Variablenlisten
- Indexlisten
- UNCOMMITTED AREAS (Listen aus Maschinenworten ohne explizite weitere Unterteilung; auf diese Weise wird das AMA zur Speicherung beliebiger Information genutzt, etwa als Arbeitsspeicher für die AIM).

6.4. Operationen der ARM (Auswahl)

a.) Horizontal paralleles Suchen

Argumente: AMA- Operand (Liste)
Zweitoperand (Vektor in SOM)
Maskenvektor (MOM)

Suchkriterien:

1. Orthogonalität
2. Nichtorthogonalität
3. Blockbildung möglich
4. Blocktausch möglich
5. logische Gleichheit
6. logische Ungleichheit
7. binäre Gleichheit
8. binäre Ungleichheit

Als Resultat liegen im Parameterspeicher vor:
AMA- Anfangsadresse und Horizontaladresse der ersten Zeile der Liste, für die das Suchkriterium erfüllt ist.

b.) Variablenweise parallele Verknüpfungen

Es werden maskierte und markierte Verknüpfungen unterschieden. Bei markierten Verknüpfungen kennzeichnet der Markierungsvektor die Positionen, an denen die jeweilige Verknüpfung bzw. Umwandlung stattfindet.

Argumente: AMA- Operand (Vektor oder Liste)
Zweitoperand (nicht bei Umwandlungen)
Maskenvektor
Markierungsvektor (TVM); nur bei markierten Verknüpfungen

Die maskierten Verknüpfungen sind im einzelnen:

1. Direkter Transport
2. Durchschnittsbildung
3. Blockbildung

4. Blocktausch

Die folgenden Verknüpfungen liefern eine binäre 1 an den Positionen, an denen die jeweilige Bedingung erffüllt ist:

1. Orthogonalität
2. Orthogonalisierung möglich
3. Gleichheit
4. Ungleichheit
5. Negation
6. N- Element
7. kein N- Element ("Unterraumauswahl")
8. 0- Element
9. kein 0- Element
10. 1- Element
11. kein 1- Element

Mit den folgenden Verknüpfungen können verschiedene Darstellungsformen von TVL wechselseitig umgewandelt werden:

1. Wert- und Erlaubnisvektor zu Ternärvektor (interne Darstellung
2. Transportieren niedere Bitpositionen
3. Transportieren der höheren Bitpositionen

Es sind folgende markierte Verknüpfungen fest vorgesehen:

1. Orthogonalisierung
2. Negation nach De Morgan
3. Wandlung N-0
4. " N-1
5. " 0-N ("Shegalkin- Transformation")
6. " 0-1
7. " 1-N
8. " 1-0
9. Mischen SOM/AMA
10. Bittausch
11. Variablentausch
12. Invertieren Maskenvektor
13. Invertieren Markierungsvektor

Weiterhin sind bis zu 32 verschiedene Verknüpfungen über ladbare Verknüpfungsspeicher programmierbar.

c.) Bewertung von Verknüpfungsresultaten

Für binäre Verknüpfungsresultate können berechnet werden:
- die Anzahl der Einsen ("Quersumme", NUMBER OF OCCURRENCES)
- die Position der ersten Eins ("Index", FIRTS OCCURRENCE).

Beide Werte werden gleichzeitig ermittelt. Für jeweils einen dieser Werte kann ein Vergleich mit einem Sollwert ausgeführt werden (Vergleichskriterium einstellbar).

d.) Transporte

Diese sind möglich
- innerhalb des AMA (MOVE)
- vom AMA zu einem der anderen Speicher (LOAD)
- von einem der anderen Speicher zum AMA (STORE).

e.) Erzeugung von Festvektoren

1. Maskenvektoren aus dem ROM- Teil des MOM
2. Erzeugung fester Zweitoperanden. Die Argumente sind:
   - Variablenzahl
   - Filler (fester Wert aller Variablen).

   Weiterhin ist es möglich, zusätzlich anzugeben:
   - Insert- Wert
   - Position.

   Dann wird an der angegebenen Position anstelle des Filler der Insert- Wert eingefügt.
3. Festwertaufschaltung (N, 0, 1, I) bei AMA- Schreibzugriffen (je nach Art des Zigriffs werden Lsiten, Vektoren oder einzelne Variable mit dem Festwert belegt).

f.) sonstige Funktionen

Dazu gehören:

1. Selektieren und Bewerten binärer und ternärer Variablen aus dem AMA bzw. einem der Vektoroperandenspeicher
2. Berechnen von Variablenwerten gemäß einem Formelausdruck im ILM ("Variablenprozessor")
3. Wandlung Markierungsvektor/Indexliste
4. Wandlung Indexliste/Markierungsvektor
5. Vektor- Umbauoperationen GATHER, EXPAND, INSERT.

Allgemeiner Hinweis:

Die ARM- Hardware ermöglicht eine Vielzahl von Kombinationen und Modifikationen der hier angeführten Grundabläufe (so z. B. durch verschiedene Modi der Adressierung und Ablaufsteuerung). Die Details sind zudem modellspezifisch.
Eine umfassendere Beschreibung für ein typisches Modell ist in /12/ zu finden.

7. Literatur

/1/: Berka, K.; Kreiser, L.: Logik- Texte.
Berlin: Akademie- Verlag 1971

/2/: Biehl, G.; Ditzinger, A.: LOGE- Programmierbare Logik problemlos entwerfen. Elektronik, München 32 (1983) 7, S. 93- 98

/3/: Bochmann, D.; Posthoff, Ch.: Binäre dynamische Systeme.
Berlin: Akademie- Verlag 1981

/4/: Bochmann, D.; Zakrevskij, A. D.; Posthoff, Ch. (Herausg.): Boolesche Gleichungen. Berlin: Verlag Technik 1984

/5/: Brunner, J.: Lösung Boolescher Gleichungen auf der Basis von Mengenalgorithmen für implizite Funktionen als Grundlage des Master- Slice- Entwurfs.
Diplomarbeit TH Karl- Marx- Stadt Sektion IT 1982

/6/: Cyrol, R.: Strukturelle Dekomposition BOOLEscher Funktionen- Komponente der Entwurfsautomatisierung.
Diplomarbeit TH Karl- Marx- Stadt Sektion IT 1984

/7/: Fehmel, J.; Posthoff, Ch.; Steinbach, B.: Binäre Systeme- rechnergestützter Schaltungsentwurf.
Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx- Stadt 7/ 1982

/8/: Haupt, D.: Mengenlehre.
Leipzig: Fachbuchverlag 1966

/9/: Kahn, K. C.: Object- oriented languages tackle massive programming headaches. Electronics, Nov. 17, 1982
S. 141- 145

/10/: Ledgard, H.: ADA- An Introduction/ Ada Reference Manual (July 1980). New York Heidelberg Berlin: Springer-Verlag 1981

/11/: Liskov, B. et al.: CLU Reference Manual. Cambridge/Mass.: Massachusetts Institute of Technology MIT/LCS/TR-225 1979

/12/: Matthes, W.: WP- Anmeldung G06F/282 506.4 Spezialprozessoranordnung zur Verarbeitung von Ternärvektorlisten

/13/: Matthes, W.: Datenzugriffsprinzipien bei objektorientierten Rechnerarchitekturen. Unveröff. Manuskript 1986

/14/: Moss, J. E. B.: Abstract Data Types in Stack Based Languages. Cambridge/Mass.: Massachusetts Institute of Technology MIT/LCS/TR-190 1978

/15/: Organick, E. I.: Computer System Organization- The B 5700/B 6700 Series. New York London: Academic Press 1973

/16/: Posthoff, Ch.; Steinbach, B.: Binäre Gleichungen-Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx- Stadt 1/1979

/17/: Posthoff, Ch.; Steinbach, B.: Binäre dynamische Systeme-Algorithmen und Programme. Wissenschaftliche Schriftenreihe der Technischen Hochschule Karl- Marx- Stadt 8/1979

/18/: Rattner, J.; Lattin, W. W.: Ada determines architecture of 32- bit- microprocessor. Electronics February 24, 1981 Vol. 4 No. 54 S. 119- 126

/19/: Schütz, A.: Analyse von Einflußfaktoren auf die Rechenzeit bei Problemlösungen mit Ternärvektorlisten. TH Karl- Marx- Stadt Sektion IT Großer Beleg 1982

/20/: Schwartz, R. L.; Melliar- Smith, P. M.: The Suitability of Ada for Artifical Intelligence Applications. SRI International Menlo Park California 1980

/21/: Steinbach, B.: Theorie, Algorithmen und Programme für den rechnergestützten logischen Entwurf digitaler Systeme. TH Karl- Marx- Stadt Dissertation B 1983

/22/: -: PASCAL- Anwendungen für ESER und SKR. Berlin: rechentechnik datenverarbeitung Beiheft 3/1981

/23/: Wright, R. E.: Documenting a Computer Architecture. IBM J. Res. Dev. Vol. 27 No. 3 May 1983 S. 257- 264

/24/: Zeigler, S. et al.: Ada for the Intel 432 Microcomputer. Computer June 1981, S. 47- 56

/25/: Zemanek, H.: Abstrakte Objekte. Elektronische Rechenanlagen 10 (1967), H. 5, S. 208- 211