Kapitel 13

Sprachunterstützende Architekturen

13.1. Einleitung

Es gibt eine **semantische Lücke** zwischen der Maschinensprache und höheren Programmiersprachen, die vom Compiler geschlossen werden muß.

Immer dann, wenn die Maschinensprache nur wenig angepaßte Befehle für die Konstrukte höherer Sprachen zur Verfügung stellt, geht das auf Kosten der Leistung des Rechners.

Die folgende Liste zeigt die Lücken.

Sprachkonstrukt	Operationen	Maschinenbefehle
Datenstrukturen		
boolean	log.Operationen	AND, OR, XOR, NOT , shift
integer	+, -, *, div	add, sub, mul, div
	Pixelmanipulationen	(MMX-Befehle)
Gleitkomma	+, -, *, /	FP add, sub, mul, div
	sin, cos, tg, atan exp, log, ln	Folgen von Befehlen
record	a.b	indizierte Adressierung
array	a (i)	
	A(i,j)	Adressrechnungen
	Skalarprodukt	Folge von Befehlen in einer Schleife
	Matrixmultiplikation	geschachtelte Schleife
Listen	car, cdr, cons	Adressrechnungen
Stack	push, pop	Folgen von Befehlen
Queue	einlagern, auslagern	Folgen von Befehlen
Kontrollstrukturen		test & jump, jump
	Schleifen	Folge von Befehlen
	case-Konstruktion	Folge von Befehlen
	Prozeduraufruf	call
	Parameter bergabe	Folge von Befehlen
	Rekursion	lange Folge von Befehlen
Prozeß		enable, disable interrupt
	P(S), V(S)	Folge von Befehlen
	create, terminate, resume, wait, activate	lange Folgen von Befehlen

Überall dort, wo Konstrukte höherer Sprachen auf komplexe Befehlsfolgen abgebildet werden müssen, kann man mit Vorteil spezielle Hardware zur Beschleunigung einsetzen.

13.2. Unterstützung spezieller Softwarekonstruktionen

13.2.1. Unterstützung für Semaphore

Man sieht einen Maschinenbefehl vor für die unteilbare Operation

P (S): `if S ≠ 0 then S := S - 1`
`else wait` (= Interrupt und Sprung ins B. S.)
dieser Befehl ist unteilbar; S ist z. B. ein Register.

Der korrespondierende Befehl

V (S): `S := S + 1` ist ein normales Inkrement.
Die Alternative wäre eine komplexe Befehlsfolge

```
  Disable Interrupts
  Lade S
  Vergleiche auf Null
wenn ja: springe zu einer Adresse im B. S.
wenn nein: S := S - 1; speichere S zurück
  Enable Interrupts
```

Im Betriebssystem muß die Umschaltung in den hochprioren Modus des B. S. erfolgen.

13.2.2. Unterstützung für Stacks

Man sieht Maschinenbefehle für "push" und "pop" vor, die automatisch die Grenzabfragen machen.

Zu einem Stack S_n gehören fünf Register: ein Stackpointer SP_n und eine obere und untere Grenze; OG_n und UG_n und zwei Ausweichadressen OV_n und UF_n für Über- und Unterlauf.

push (R_x, S_n): $S_n(SP_n) := R_x$
$SP_n := SP_n + 1$
if $SP_n = OG_n$ then Sprung nach OV_n

pop (S_n, R_x): $SP_n := SP_n - 1$
if $SP_n = UG_n$ then Sprung nach UF_n
$R_x := Stack_n (SP_n)$

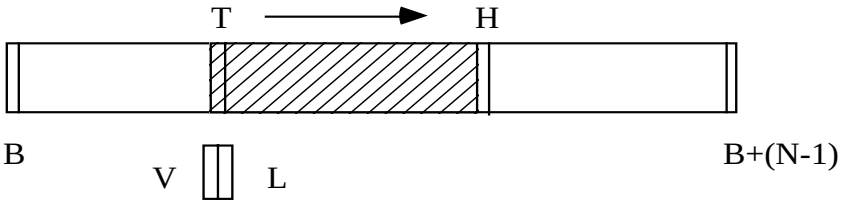
Der Sprung an die Adressen OV_n bzw. UF_n muß nicht notwendigerweise ins B. S. führen; auch innerhalb des Programms kann ein Stacküberlauf z. B. Umspeicheroperationen veranlassen (Wechselpuffertechnik).

13.2.3. Unterstützung für Warteschlangen

Wird eine Warteschlange Q_n als Kreisliste der Länge N implementiert, dann sollte der Maschinencode Befehle addiere/subtrahiere mod N kennen und mit Hardware die Ein- und Auslageroperationen unterstützen.

Eine Warteschlange Q_n wird kontrolliert durch ein Basisregister B_n, zwei Pointer H_n (head) und T_n (tail), zwei Flags V_n (voll) und L_n (leer) und zwei Sprungadressen OV_n und UF_n. Initialisiert wird die Warteschlange mit

$H_n = T_n = 0$ und $V_n = 0$ und $L_n = 1$.



Die beiden Operationen sind

- einlagern (R_x, Q_n)

```
if (Vn = 0) then Qn (Bn + Hn):= Rx ;
                 Hn  :=  (Hn + 1) mod N ;
if (Hn= Tn) then Vn  :=  1 ;
                 Ln  :=  0 ;
             else    Sprung nach OVn
```

- auslagern (Q_n, R_x)

```
if (L = 0)  then Rx := Qn (Bn + Tn) ;
                 Tn := (Tn + 1) mod N ;
if (T = Hn) then  L := 1 ;
                 Vn := 0 ;
             else   Sprung nach UFn
```

Für Realzeitsysteme gibt es Zusatzchips, die als "real-time unit, RTU", hardwaremäßig die Warteschlangen für Prozeßumschaltungen und ihre Semaphore verwalten.

13.2.4. Unterstützung für Prozeduraufrufe

Hier sieht eine RISC-Maschine eine Unterstützung durch ein Registerband vor, siehe dort, die die Parameterübergabe einfacher machen.

Das Abspeichern der Rückkehradresse kann in einem eigenen Stack erfolgen (siehe Kapitel 6).

Auch kann die Parameterübergabe durch hardwareverwaltete Stacks unterstützt werden.

13.2.5. Unterstützung der Bildung von Skalarprodukten : digitale Signalprocessoren

In sehr vielen Anwendungen müssen Skalarprodukte gebildet werden

$$c = \sum_{i=1}^{n} a_i * b_i$$

Hardwareunterstützung kann durch drei Maßnahmen geschehen:

- MAC-Operationen (multiply & accumulate)

  Die Bildung von

  $c := c + a_i * b_i$

  ist eine einzige Operation in einem relativ langen Akkumulator.

- Der Zugriff auf die Komponenten a_i und b_i wird hardwaremäßig unterstützt durch ein Adressrechenwerk:

```
for  i = 1 to limit do
     adr_a = j_a + i
     adr_b = j_b + i
```

- Unterbringung der beiden Vektoren in getrennten Speichern M_a und M_b :

  $a_i = M_a (adr_a)$; $b_i = M_b (adr_b)$

Diese Operationen werden hardwaremäßig realisiert in **Digitalen Signalprozessoren** (digital signal processor, DSP) mit typischem Aufbau:

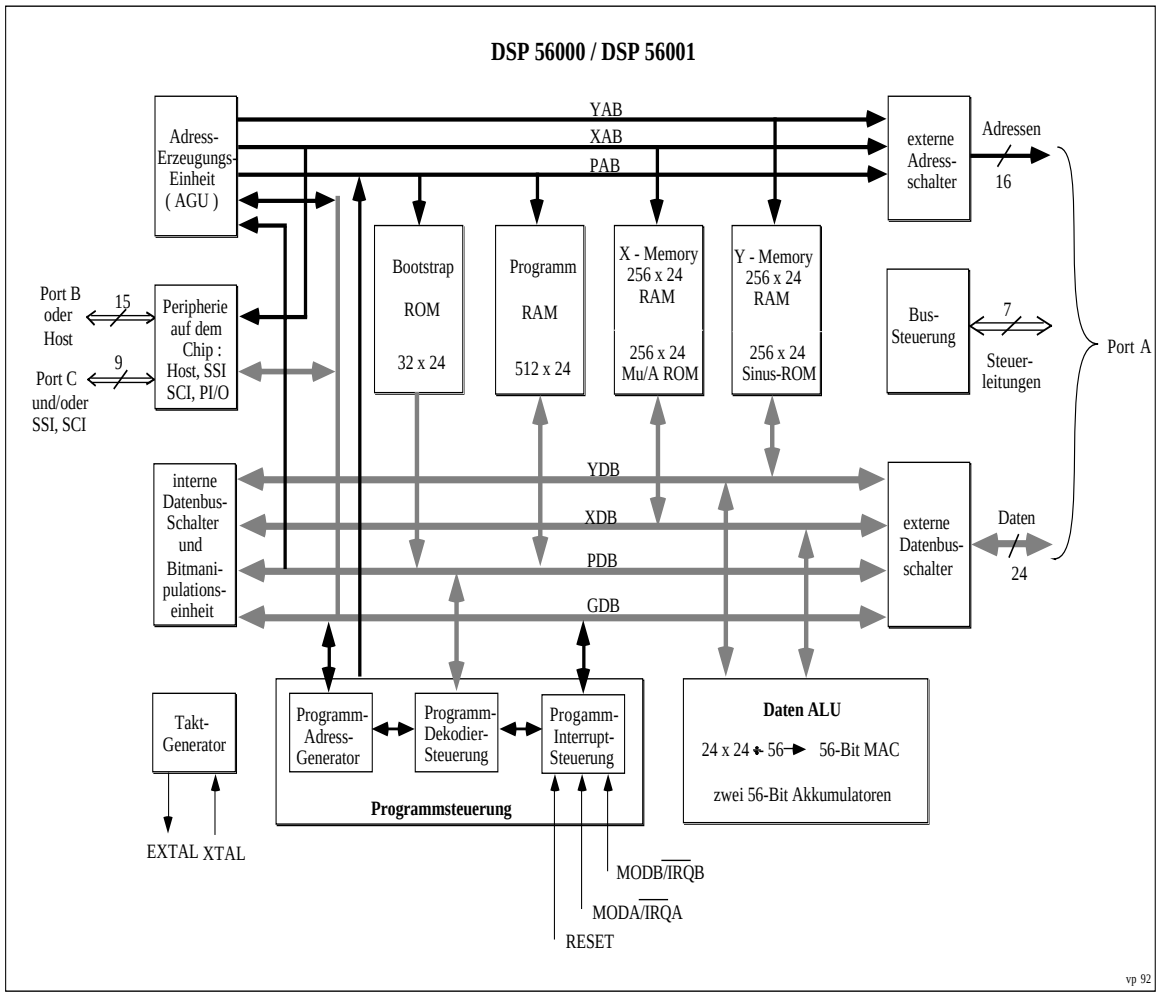
- drei Speicher M_a, M_b, P für zwei Vektoren und das Programm
- drei Adressbusse $(j_a + i)$, $(j_b + i)$, PC
- drei Datenbusse (a_i, b_i, Befehl)
- ein Ergebnisbus
- ein Akkumulator für die MAC-Operation
- ein komfortables Adressrechenwerk, mit Schleifenzähler zur automatischen Berechnung einer for-Schleife.

  Schleifen der Form

```
for i = 1 to n do
   c := c + a(i) * b(i)
```

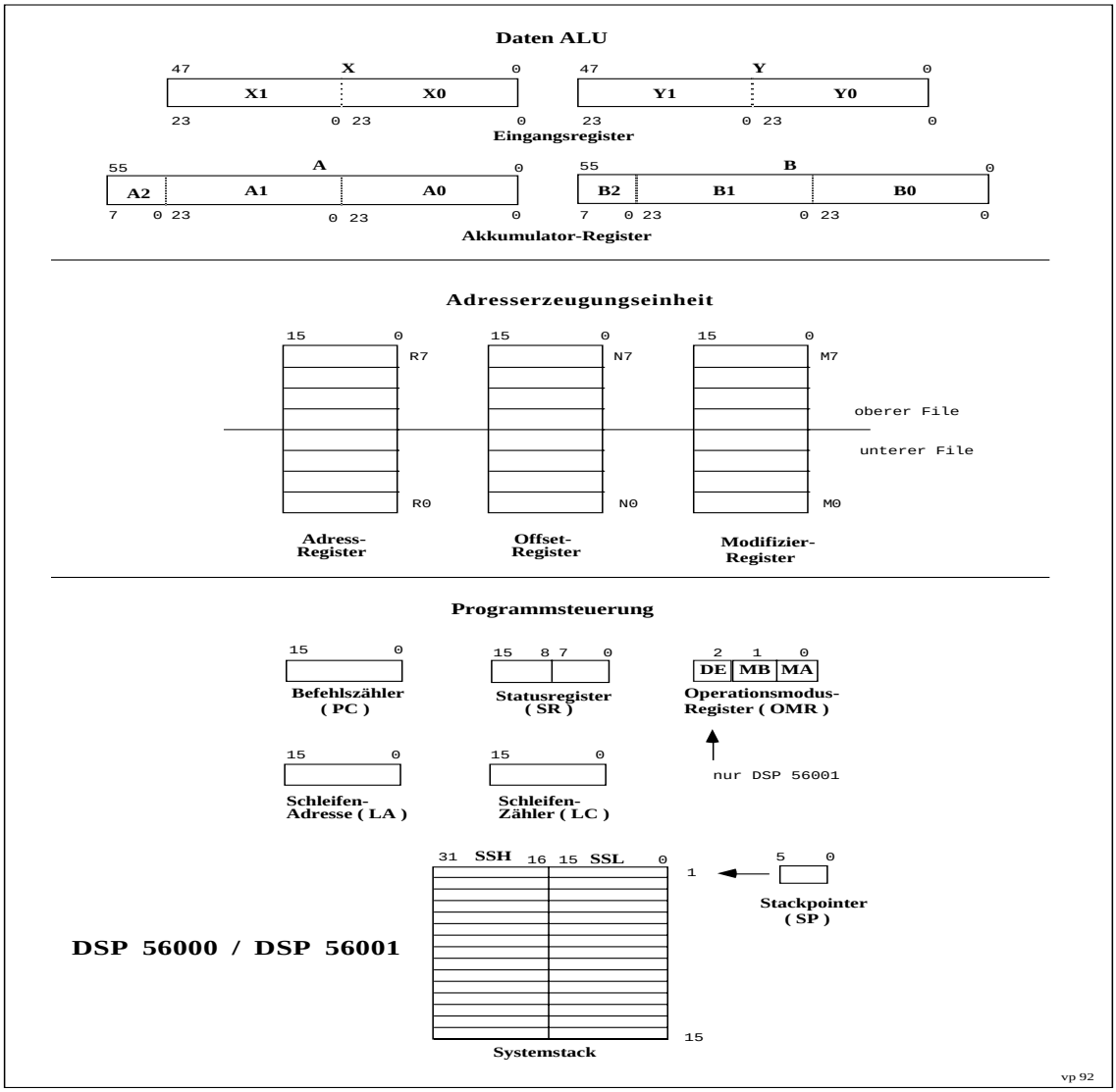
  laufen automatisch ab, sind ein Maschinenbefehl Befehl in einem DSP.

Beispiel Motorola DSP 56000.



Die beiden Datenspeicher x-Memory und y-Memory enthalten hier noch einen ROM, in dem eine Sinusfunktion und eine Kompressionskennlinie für Anwendungen der Nachrichtentechnik vertafelt sind.

Die Registerstruktur zeigt das folgende Bild.



Es wird hardwaremäßig die Schleifenabarbeitung unterstützt durch ein extra Register für die Schleifenadresse (loop address, LA) , einen Schleifenzähler (loop counter, LC) und durch drei Adressregisterfiles für Adresse R, Offset N und Modifikation M für die Operationen

adr := R + N ; N := N + M .

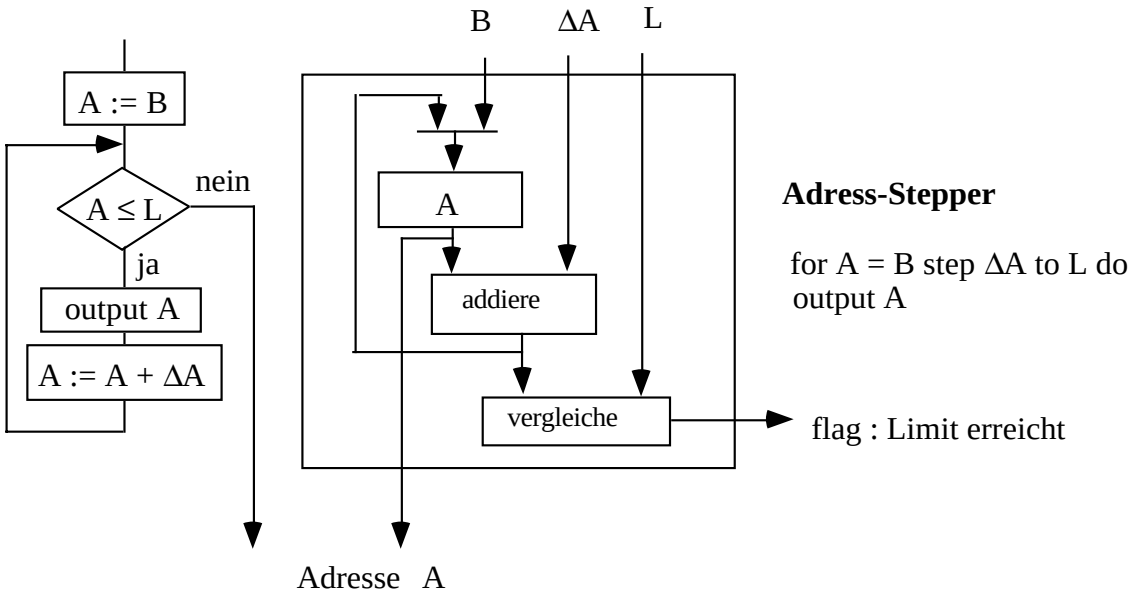
Dazu kommt ein eigener Stack an Bord für die Unterstützung von Prozeduraufrufen.

13.2.6. Unterstützung von Filteroperationen auf Matrizen - X-Puter

Bei Filteroperationen auf Matrizen läßt man ein n x m - Fenster mit Filterkoeffizienten über eine N x M - Matrix laufen.

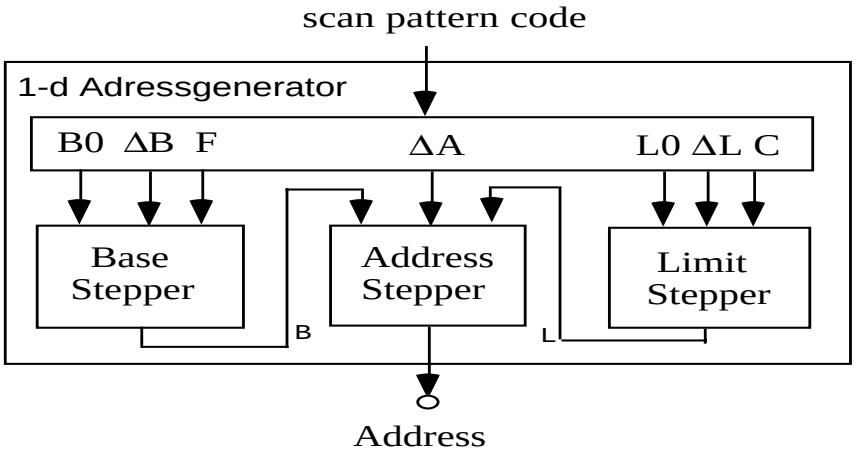
Die Beschleunigung der Operationen kann durch zwei Maßnahmen geschehen: eine Adressrecheneinheit, die die Daten aus einem Scan-Cache lädt und einer Nachfüll-operation für den Scan-Cache.

Die Adressierung des Scan-Cache kann dadurch unterstützt werden, daß drei Adressrecheneinheiten benutzt werden: Jede für sich bildet Adressen A aus einer Basisadresse B, einem Inkrement ΔA und einem Limit L.



Hier stehen nebeneinander das Flußdiagramm, die Hardwarerealisierung und eine Formulierung als Programm.

Drei solche Adressrecheneinheiten bilden eine Scan-Einheit



die die Operation Scan : (B0, ΔB, L0, ΔL, F, C, ΔA) automatisch ablaufen läßt

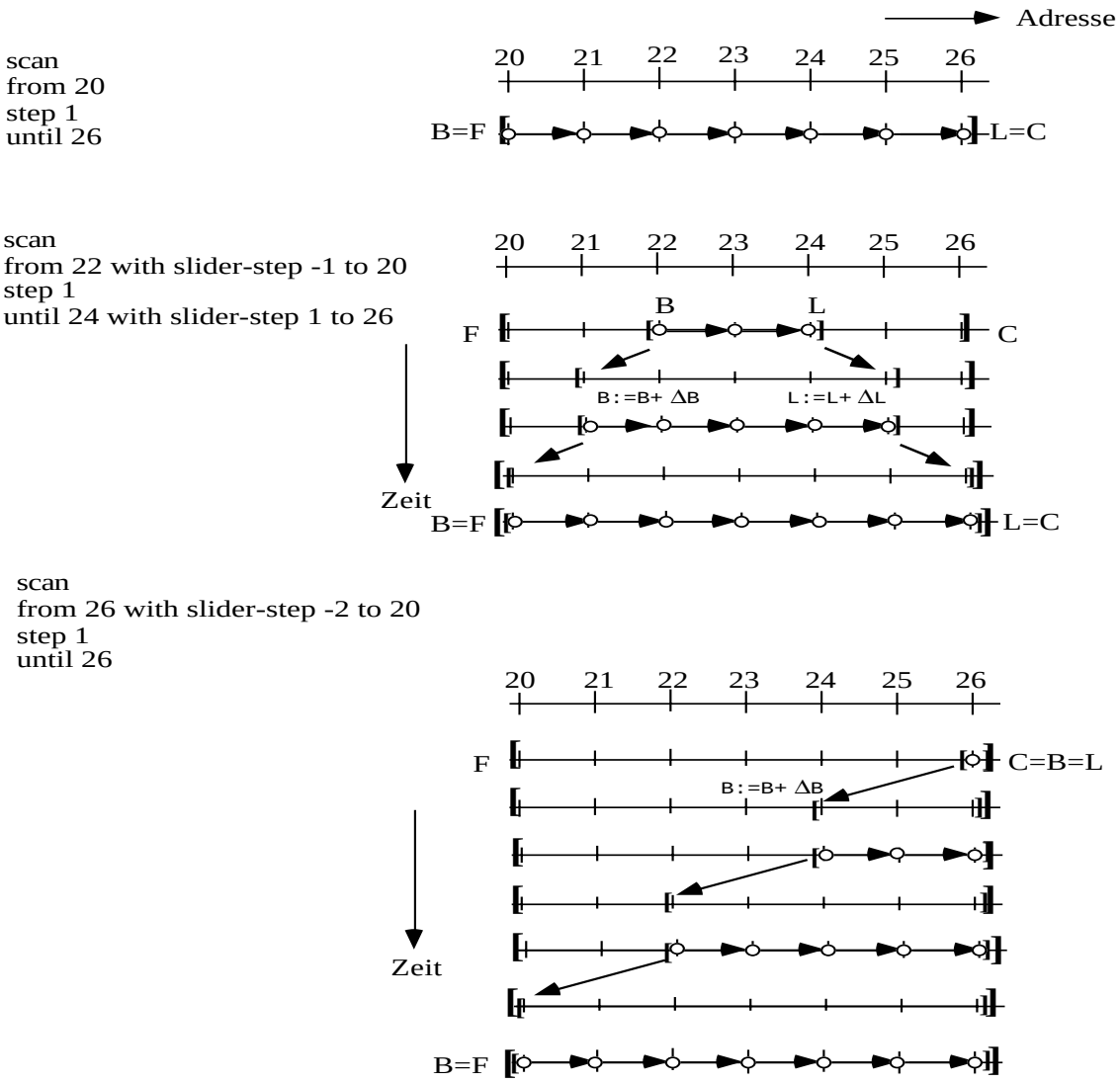
die in Kurzform beschrieben wird durch

```
scan from B0 (with slider-step DB to F )
          step DA
     until L0 (with slider-step DL to C)
```

Als Programm beschrieben:

```
 procedure scan ( B0, DB, L0, ΔL, F, C, ΔA )
var A ,B : address
begin
   B =: B0;
   repeat
         A := B;
         output A;
         repeat
               A := A + ΔA ;
               output A ;
         until (A ≥ L)            (* Adress Stepper *)
         if (L < C)
               then L := L + ΔL ; (* Limit Slider *)
         B := B + DB ;            (* Base Slider *)
   until  (B ≤ F)
end.
```

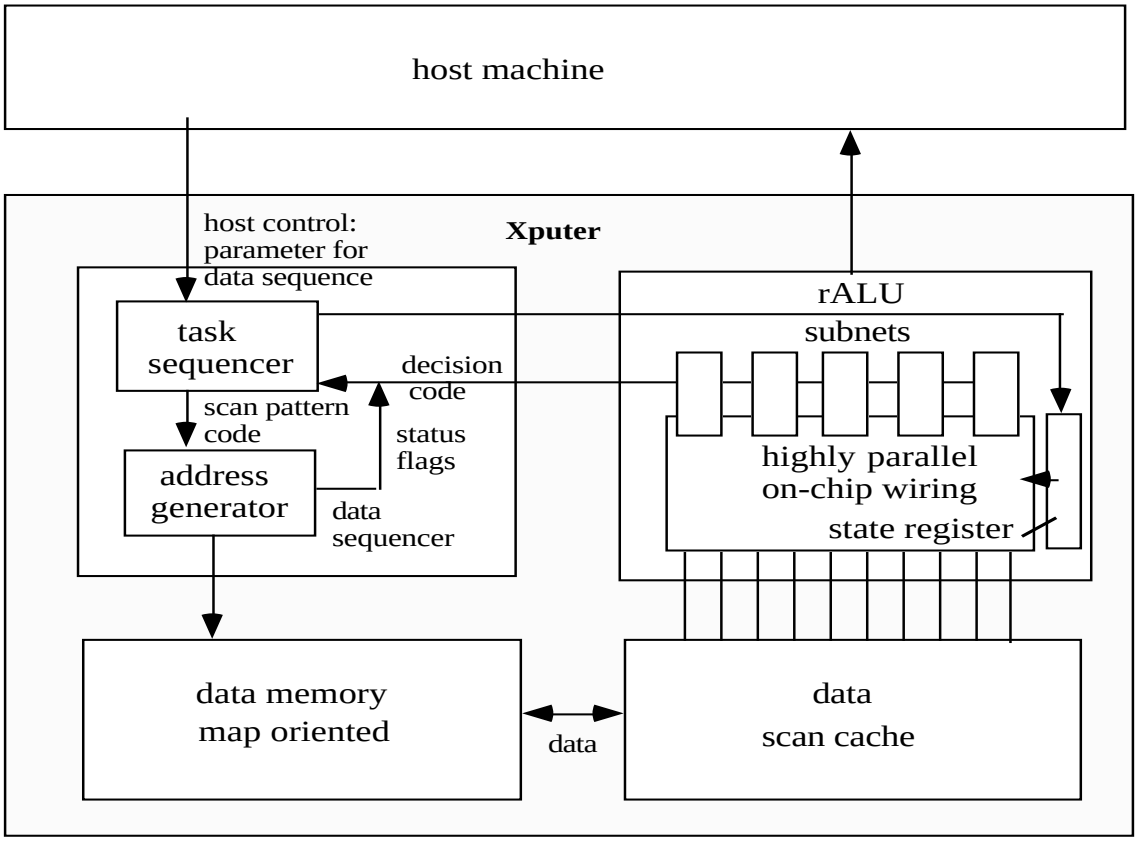
Beispiele für Scan-Pattern sind



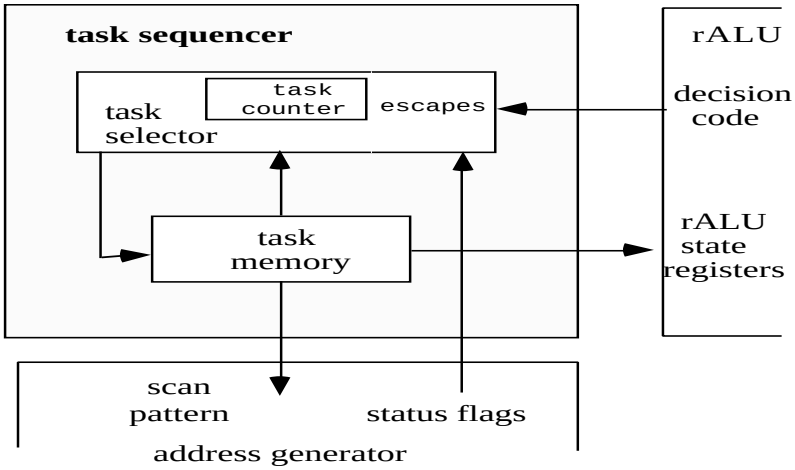
Eine Scan-Einheit erzeugt die Adressen für einen eindimensionalen Vektor. Mit zwei Scan-Einheiten läßt sich eine Matrix fast beliebig scannen.

Damit ist die Abarbeitung von Schleifen abgesenkt in die Hardware.

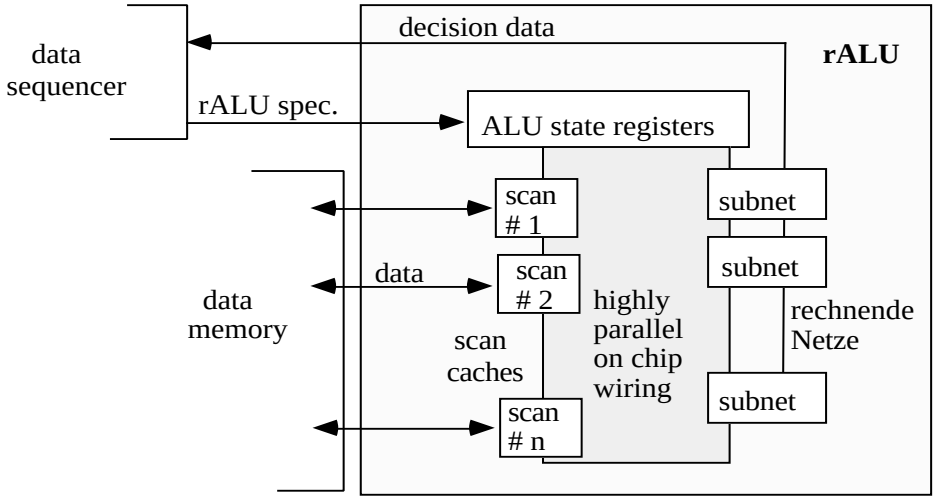
Der weiteren Beschleunigung der Filterberechnung dient ein Satz von rechnenden Einheiten, die parallel die Filterfunktion für mehrere Pixel berechnen. Die Daten einer Scanline müssen so nicht immer wieder aus einem Speicher nachgeladen werden, sondern können in einem Scan-Cache zwischengespeichert werden mit schnellem Zugriff ohne Umweg über einen Assoziativspeicher. Ein prototypischer Rechner ist der X-Puter.



Die Befehlsfortschaltung geschieht im Task-Sequencer, der Scan-Pattern und Datenverarbeitungsfolgen bereitstellt. Durch die Scan-Einheiten ist der Befehlscode sehr kurz.



Die Verwendung von FPLA's für die rechnenden Einheiten ermöglicht die Umkonfigurierung und Anpassung der rechnenden Einheiten an die jeweilige Aufgabe.



13.3. Lisp-Maschinen

Die Sprache Lisp kennt als Datenstrukturen Atome und Listen aus doppelt verpointerten Knoten. In Lisp sind Daten und Programme ununterscheidbar: beides sind Listen. Die Listenelemente übernehmen hier die Funktion der Bits im Speicher einer v. Neumann-Architektur. Auf der Ebene der Bitpattern (Listen) sind Daten und Programme im Speicher ununterscheidbar.

Lisp hat die Mächtigkeit der Turing-Maschine. Die Favorisierung der Rekursion zur Formulierung von Problemen als Lisp-Ausdrücke gibt der Sprache ihre Ausdrucksmächtigkeit, insbesondere für Aufgabenstellungen aus der KI.

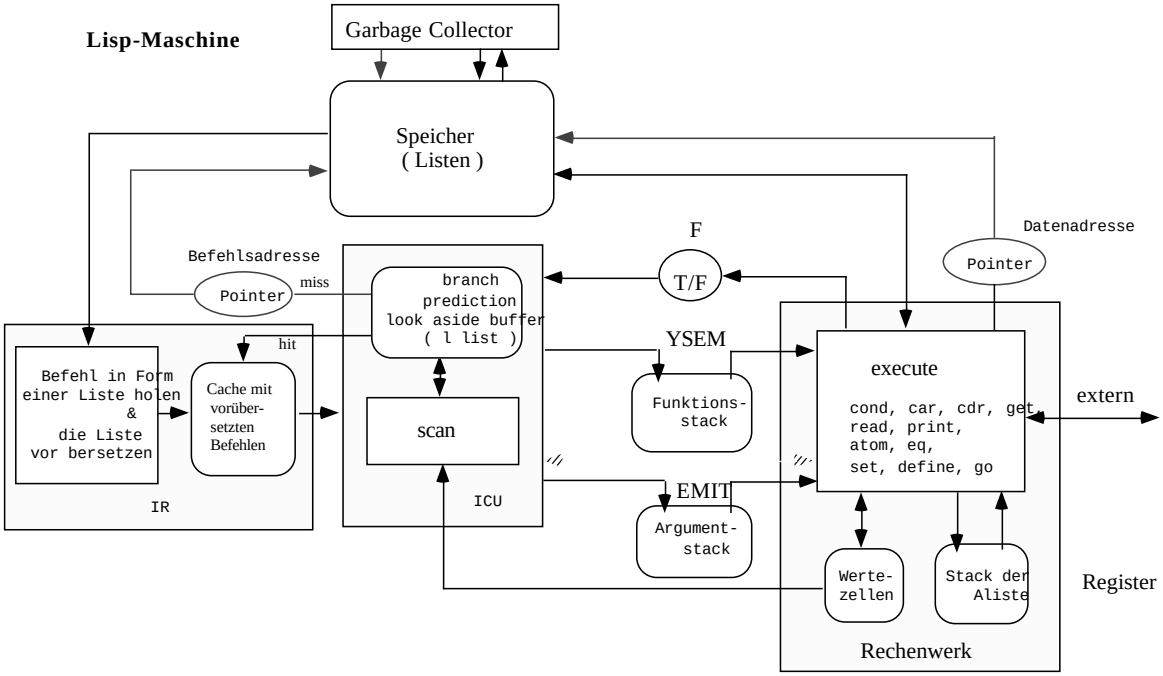
Die Verwendung von doppelt verpointerten Knoten in Listen verwandelt den Speicher in einen Spaghettihaufen von Pointern und läßt traditionelle Konzepte von Caches und sequentieller DV mit Befehlspipelining zusammenbrechen.

Tiefe Rekursionen erfordern spezielle Stacks, deren Implementierung in Software Programme sehr langsam laufen läßt. Die Interpretation von Listen als Programme läßt die Abarbeitung zusätzlich langsam werden, und eine Compilation zerstört die Eigenschaft, daß das Ergebnis einer Rechnung eine Liste sein kann, die dann als Programm weiterinterpretiert wird.

Eine der möglichen Ideen für eine Lisp-Maschine soll hier gezeigt werden (Eicher, 1985).

Der Speicher ist ein Listenspeicher, der Befehle und Daten in Form von Listen vorhält und über Pointer angesprochen wird. Die Zugriffsbreite sollte die der einzelnen Listenelemente sein: (4 - 8 Byte).

Ein Befehl, d. h. eine Liste, wird über einen Befehlspointer angesprochen und geladen. Sie wird vorübersetzt und in einem Cache abgespeichert, auf den über ein "branch prediction look aside buffer", die l-list, zugegriffen wird.



Der Cache enthält vorübersetzte Listen, die noch nicht abgearbeitet sind.

Eine Befehlssteuereinheit (instruction control unit, ICU) erkennt einen Befehl (= vorübersetzte Liste) und verteilt mit einer Funktion "scan" die Aufgaben: Funktionsaufrufe werden in einem Funktionsstack geparkt, die Argumente der Funktionen in einem Argumentstack gesammelt.

Immer wenn eine Liste komplett gescannt ist - meist nur eine Teilliste des Befehls in Arbeit - wird die Ausführungseinheit aktiviert, die mit einer internen Funktion "execute" Funktionen vom Stack lädt, die zugehörigen Argumente vom Argumentstack sammelt und die Funktion ausführt.

Wertzuweisungen werden in einem eigenen Stack (A list) und einem Speicher (Wertezellen) gemacht. Sie übernehmen die Rolle der Register einer v.Neumann-Maschine

Die Kontrolle wechselt zwischen "scan" und "execute" hin und her. Beim Abstieg in eine Rekursion wird der Name der Funktion an die Befehlssteuereinheit zurückgereicht, die die vorübersetzte Liste über die l-list im Befehlscache findet und mit "scan" weiterarbeitet.

Wird die Funktion nicht im Cache gefunden, muß sie aus dem Listenspeicher nachgeladen werden. Dazu dient der Speicher der Wertezellen: bei dem Namen ist hier der Befehlspointer auf die Liste, die die Funktion repräsentiert, mit verzeichnet.

Primitivfunktionen von Lisp können in der Ausführungseinheit direkt ausgeführt werden. Die Argumente sind Pointer auf Listen im Listenspeicher.

Die Vorübersetzung erkennt die Namen der Primitive und kennzeichnet sie entsprechend.

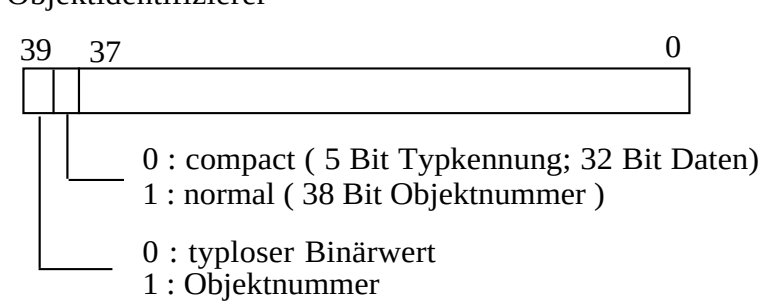
Auf dem Listenspeicher arbeitet parallel ein Garbage-Collektor: Er sucht Listen, die nicht mehr referenziert werden und verkettet sie in einer Freiliste, aus der bei Bedarf Knoten für neu zu bauende Listen geholt werden. Im Initialzustand ist der Speicher eine einzige Freiliste.

Die Stacks, der Befehlscache und die Wertezellen sind nicht Teil des Speichers, sondern sind unabhängige Einheiten, die von eigener Hardware verwaltet werden. Sie erlauben tiefe Rekursionen.

13.4. Unterstützung der Rekursion und objektorientierter Programmierung - WISC

Man unterstützt die Konstrukte höherer Programmiersprachen durch Verwendung von Microcode in der Maschine, der ggf. nachgeladen werden kann (writable instruction set computer, WISC).

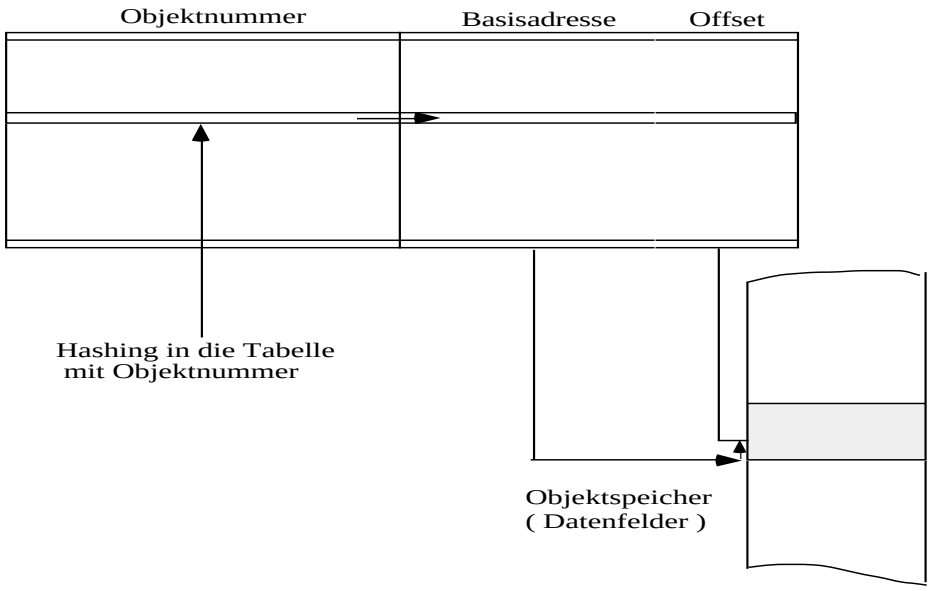
Beispiel: **Rekursiv-Maschine** (Linn, 1988)
Rechner für Manipulation von Objekten.
Jedes Objekt in der Maschine ist durch eine 40-Bit-Nummer gekennzeichnet. Sie begleitet das Objekt von der Entstehung an und ist der "Name" des Objekts. Jeder Zugriff auf ein Objekt geschieht über diese Nummer.



In den 40 Bit sind Binärwerte, direkt zugreifbare Daten und Objektnummern unterschieden, die zusammengesetzte Objekte kennzeichnen.

Die Objekte werden im "object store" gehalten (in der Rekursiv-Maschine 128 k Worte à 40 Bit), einem schnellen Halbleiterspeicher von 4 M Byte und 100 ns Zugriffszeit. Er ist nur zur Hälfte belegt, um Platz für "garbage collection" zu haben.

Der Zugriff auf den Speicher geschieht durch eine Pager-Tabelle von 64 k Worten: eine Objektnummer wird durch eine Hasching-Funktion auf 16 Bit komprimiert als Adresse in die Pager-Tabelle.



In der Tabelle steht eine 40-Bit-Objektnummer als Header und ein 40-Bit-Datenfeld: entweder direkt der Wert oder der Verweis in den Objektspeicher, in dem die Datenfelder stehen.

Der Objektspeicher wird mit 17 Bit für den Tabellenanfang und 17 Bit für die Tabellenlänge adressiert.

Die garbage collection im Objektspeicher geschieht automatisch. Die Maschine verarbeitet Befehle einer höheren Sprache, wie z. B. Smalltalk.

Die Befehle der Sprache sind in 40-Bit Befehlen codiert mit 10 Bit Befehlscode und 30 Bit für Argumente.

Sie stehen in einem eigenen Speicher, dem

New Language Abstract Memory, NAM & NAMARG,

einem schnellen Befehlscache mit 500 k Worten zu 40 Bit (in der Rekursiv ein SRAM mit 55 ns Zugriffszeit).

Ein Op-Code greift über eine Tabelle (control store map, CSMAP) von 2048 Worten zu 14 Bit auf einen Kontrollspeicher zu, in dem in 16 k Worten zu 128 Bit die Mikroprogramme zu den Befehlen stehen.

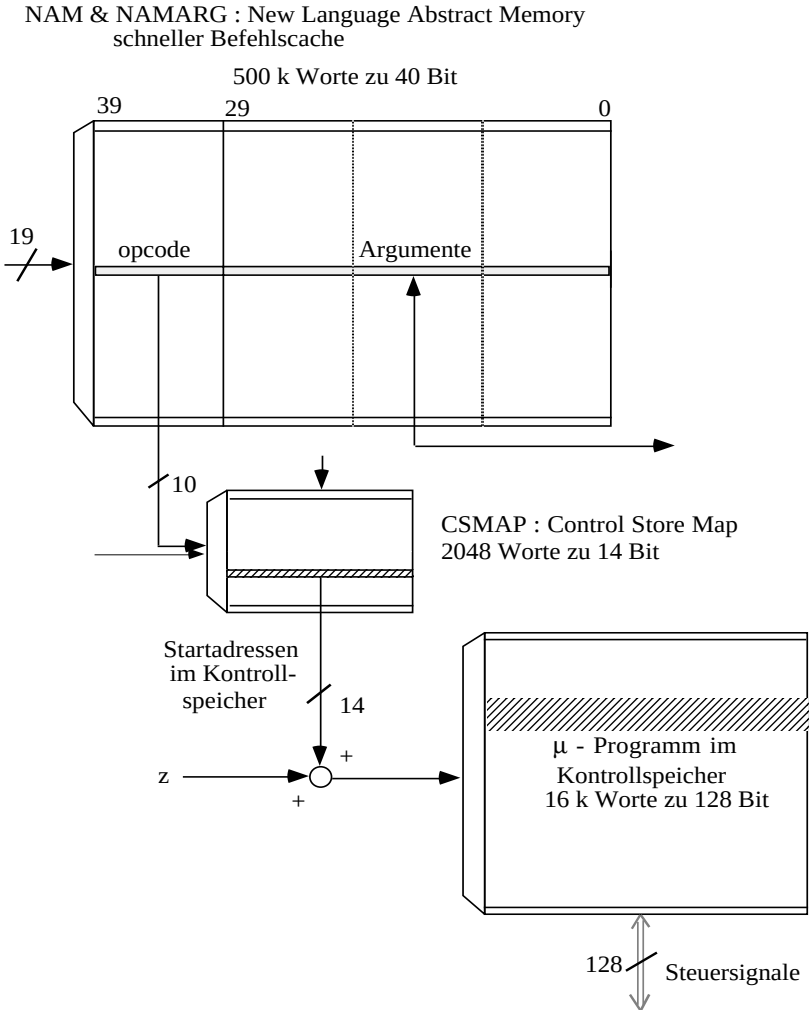
Ein Op-Code springt die Anfangsadresse der Mikrobefehlsfolge an.

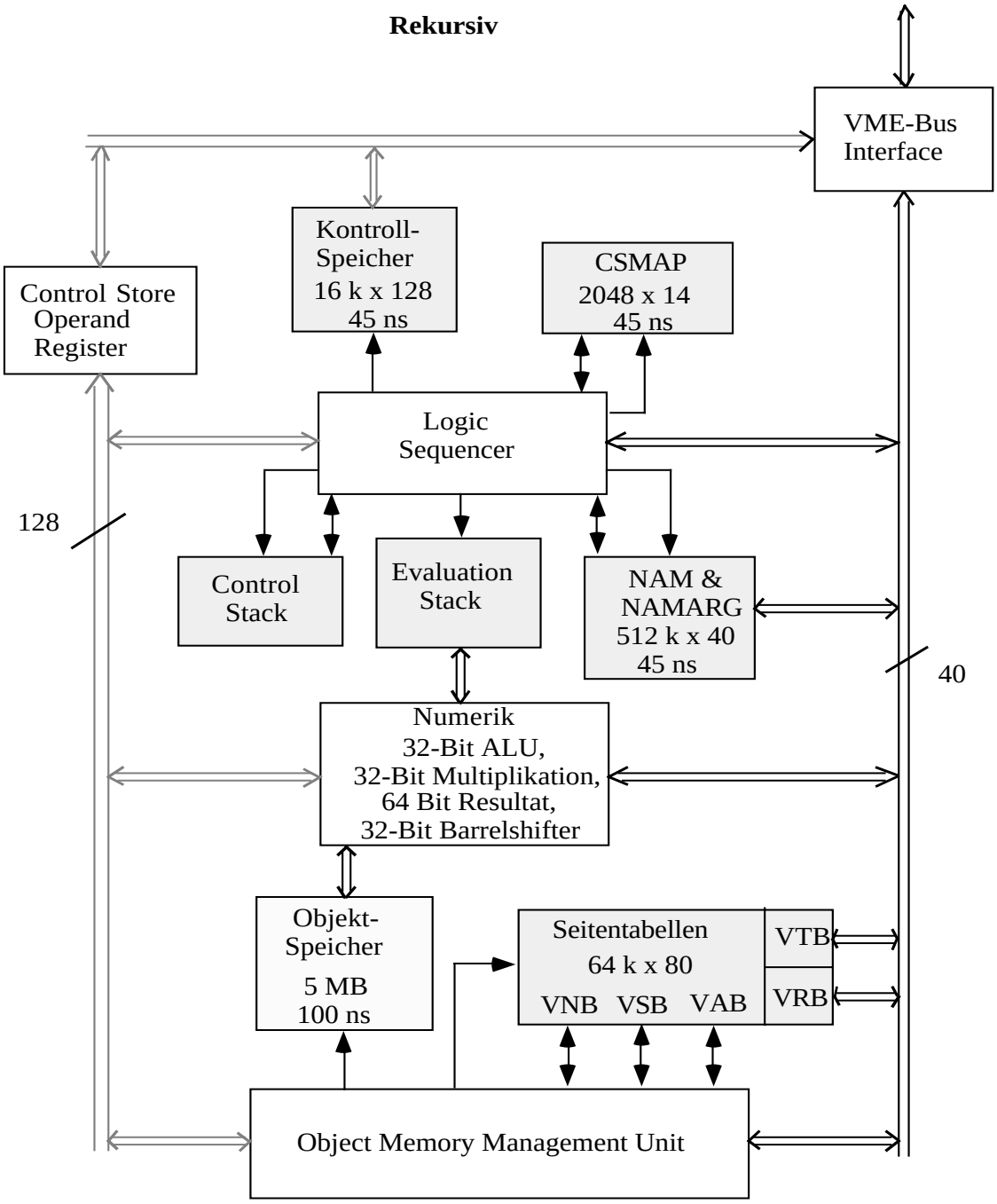
Die Steuersignale aus dem Kontrollspeicher betreiben über einen logischen Sequencer die Numerik, einen Control-Stack für Befehle, einen Evaluation Stack, sprechen den NAM & NAMARG an und veranlassen den Zugriff auf den Objektspeicher und die Seitentabellen für den Zugriff auf den Massenspeicher.

Als Beispiel für ein Programm auf der Rekursiv diene ein Lisp-Programm für das Kopieren einer Liste

```
( DEFUN ( COPYTREE ( ITEM ) )              - Funktionskopf
    ( COND ( ( ATOM ITEM ) ITEM )          - Funktionskörper
           ( T ( CONS ( COPYTREE ( CDR ITEM ) )
                      ( COPYTREE ( CAR ITEM ) )
               )
           )
    )
)
```

Es ist ein rekursives Programm mit zwei Rekursionen im Funktionskörper. Sie rufen die Funktion jeweils auf mit anderen Argumenten. Die Maschine Rekursiv verwaltet diese Aufrufe automatisch.





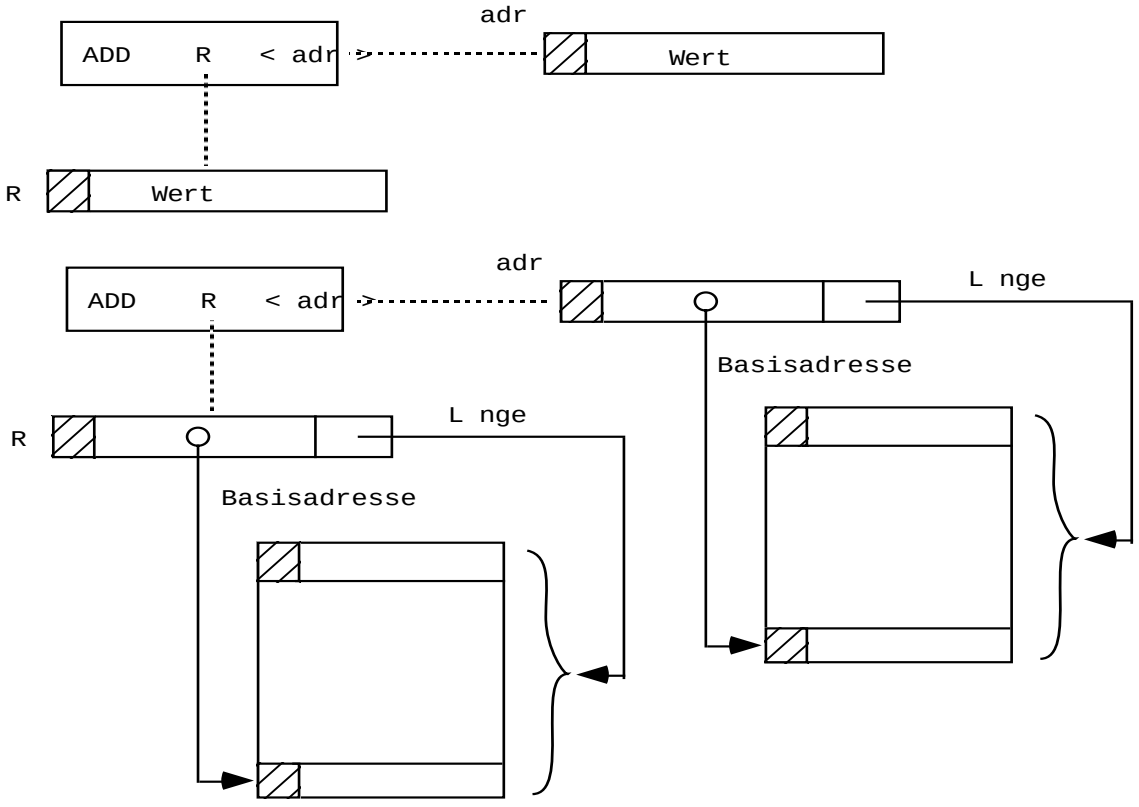
13.5. Architektur mit Typenkennung

13.5.1. Tags

Man verzichtet auf die Ununterscheidbarkeit von Daten und Programmen im Speicher und kennzeichnet mit einigen Kennungsbits (tag) bei jedem Wort im Speicher den Typ des Inhalts bei dem Wort.

Vorteile:

- typunabhängige Befehlsausführung wird möglich: das Tag bestimmt die spezielle Art des Schaltnetzes, das die Operation durchführt: z. B. integer oder GK-Addition
- Fehlermeldungen werden erzeugt beim Versuch, Programme zu manipulieren oder Integer-Größen als Programm zu interpretieren.
- Überprüfung von Feldgrenzen ist möglich.
- Zugriff auf nichtinitialisierte Variable löst eine Fehlermeldung aus.
- Struktur- Datentypen können durch eine Kennung "Deskriptor" gekennzeichnet werden
  Beispiel:
  < vektor - descriptor > :: =
  < typkennung > < basisadresse> < zahl-der-elemente >
  Die Angabe eines Deskriptors stößt einen Adressgenerator an, der die Adressen automatisch erzeugt. Damit werden z.B. die Addition von Skalaren und Vektoren gleich behandelt.



- Unterstützung der Mechanismen höherer Sprachen
  Beispiel: Typ "Label"
  < label > :: = < typkennung > < befehlsadresse > < environment-pointer >
  < environment-pointer > zeigt auf die Liste, mit der die Umgebung nach dem Sprung arbeiten soll.
- Unterstützung der Implementierung von ereignisgesteuerten Konstrukten z. B. durch einen Datentyp "parameter-set".
- Unterstützung des Betriebssystems durch eigene Datentypen "Keller" und "Stack".

- Kennung ob Kopiererlaubnis besteht, wichtig für die "garbage collection" bei Listen.

Nachteile:

Die Adressierung erfolgt nicht mehr byteweise, sondern wortweise. Die Bits für eine Kennung nehmen Speicherplatz fort: z. B. Kennung 4 Bit, Daten 32 Bit erfordert 12 % mehr Speicherplatz. Bei 64 Bit Daten reduziert sich der Overhead auf 6%.

Hierarchische Typkennung

Man wird versuchen, nur generische Typen zu definieren

- signals : Daten von außen
- scalar data : skalare Größen
- data-structures : zusammengesetzte Daten wie arrays oder records
- code-elements : Programmteile
- status-information : langlebige Statusanzeigen

und wird versuchen, mit möglichst wenigen Typen auszukommen.

Als Beispiel einer Typkennung im 36-Bit-Speicherwort mit 4-Bit-Kennung (16 verschiedene Typen

bit	vector descriptor	processor-statuswort
char	segment descriptor	record-descriptor
BCD-digit	data	
integer	code	
real	stack	

Eine sparsame Alternative ist eine Aufteilung der Kennung:

Data-Primitive	⎫	
Vektor-Deskriptor	⎬	2 Bit
Segment-Deskriptor	⎪	
Programmstatuswort	⎭	

Innerhalb der Data-Primitive weitere Unterteilung

Bits	⎫	
Bytes	⎬	2 Bit
Integer	⎪	
Real	⎭	

Ein erster Schritt hin auf Kennung von Daten ist die Speichersegmentierung mit einer Typkennzeichnung in den Segmentdeskriptoren als "data", "code" und "stack".

In die gleiche Richtung weist die Einführung von Objektdeskriptoren mit eindeutigen Objektnummern.

13.5.2. Datentyp-Architekturen

13.5.2.1 DRAMA-Architektur

Verwendet man Datentyp-Deskriptoren als eigenständige Objekte in einer Maschine, mit eigenen Standardoperationen kommt man zu einer **DRAMA-Architektur** :
(descriptor referenced autonomous memory allocation, DRAMA).

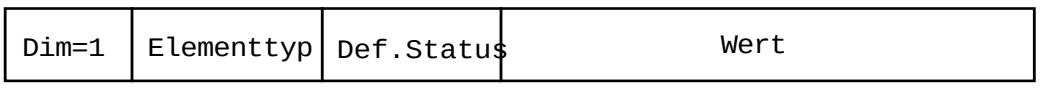
Der Zugriff auf Objekte erfolgt nur über Deskriptoren. Von da an übernimmt Hardware und erzeugt den konsekutiven Zugriff auf Adressen.

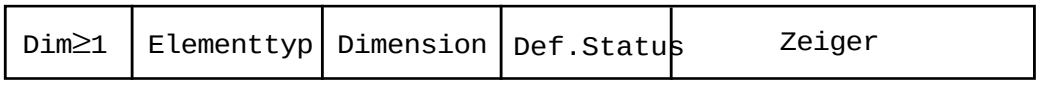
Bei skalaren Größen steht die Wertangabe im Deskriptor selber. Der Benutzer sieht nur Deskriptoren, die er als Objektnamen deuten kann.

Es wird der Variablenbegriff neu definiert:
< variable > :: = (< name >, < wert >)
< wert > :: = (< Struktur-Definition >, < Datenmenge >)

In der Maschine sind dann zwei Arten von Transformationen denkbar:
- Strukturändernde Transformationen
- Datenverändernde Transformationen.

Die Strukturdefinition wird gegeben durch Datendeskriptoren





Der Definitionsstatus ist definiert / nicht definiert.

Elementtypen sind

clear (kein Inhalt)
integer ⎫
real ⎬ der Wert steht direkt im Deskriptor drin
char ⎪
boolean ⎭
long integer
long real
reference (Pointer)
Identifier
E-Typ (Elementtyp), der Wert steht im Deskriptor drin
Module
Procedure
Systemobjekt
Label .

Im Stack stehen Objektklassen "Variable" und "Parameter" mit Lebensdauern über die Prozeduraktivierung und als Bereich eine Prozedur; im Heap stehen Pointer auf Variable und Parameter mit Lebensdauern über eine Prozeßaktivierung und mit einem Gültigkeitsbereich über alle Prozeduren eines Moduls.

Strukturtransformationen:

Ein Datenvektor der Länge L, bestehend aus Elementen der Menge S, wird definiert durch eine **Zugriffsfunktion** .

$$I : M \supset \{ b : b - 1 + c \cdot L \} \rightarrow S$$

mit M : Menge der Speicheradressen
$b \in M$ eine Basisadresse
c : ein Faktor, der das Vielfache/den Bruchteil von Worten angibt, der von einem Speicherwort zur Speicherung eines Elements $S_i \in S$ benötigt wird.

Für ein rechteckiges homogenes Feld A, dessen Komponenten Elemente $S_i \in S$ sind, ist eine **Strukturabbildungsfunktion**

$$\sigma_r : N^r \rightarrow N \qquad \text{definiert}$$

mit $(n_0, ..., n_{r-1}) \in N^r$ als Koordinaten des Feldes der Dimension r und $\alpha \in N$ als Adresse.

Für die Komponenten von S_i gilt

$1 \leq n_i \leq d_i$ (Dimension in Richtung i)
$0 \leq i \leq r-1$ Anzahl der Richtungen

Das r-Tupel $(d_0, ..., d_{r-1})$ ist der Dimensionsvektor des Feldes.

Der Wert einer Variablen wird dann beschrieben durch

< wert > :: = < strukturdeskriptor >, < dimensionsvektor >, < datenvektor >

I beschreibt als Zugriffsfunktion den Zugriff auf die Elemente von S im Speicher M .

σ_r beschreibt als Strukturabbildungsfunktion die logische Struktur von S .

Dazwischen vermittelt die **Adressfunktion**

$\alpha : N \rightarrow M$.

Die vollständige **Indizierungsfunktion** ist dann die Konkatenation der Abbildungen

$I \circ \alpha \circ \sigma_r : N^r \rightarrow M \rightarrow S$

und liefert den Zugriff auf Elemente der r-dimensionalen Struktur.

In höheren Programmiersprachen treten die Indizierungs-r-Tupel meist auf in Form einer Indexliste am Variablennamen.

Typische Adressfunktionen sind dann von der Form

$\alpha (i) : b + c \cdot i$ mit $i \in N$.

Eine standardisierte Speicherabbildung

$\alpha \circ \sigma_r : N^r \rightarrow M$

spezifiziert eine "kanonische" Ordnung für die Positionen der Struktur, gegeben durch N^r auf die Positionen des Datenvektors

$$\{ b : b-1 \leq \ldots \leq u \} \subset M \quad \text{mit} \quad \prod_{i=0}^{d-1} = d_j$$

Strukturoperationen bilden Indextupel ineinander ab:

$I : N^r \rightarrow N^r$.

13.5.2.2. STARLET-Maschine (Giloi 1982)

Als Beispielarchitektur diene hier die STARLET von Giloi (1982)

Sie unterstützt den Zugriff auf zweidimensionale Arrays

$\sigma (i, j) = i \cdot d_1 + j \; ; \; i \in [0, ..., d_{0-1}]$

$j \in [0, ..., d_{1-1}]$

mit d_0 = Anzahl der Zeilen

d_1 = Anzahl der Spalten des Array.

Eine Strukturoperation ist dann von der Form $I : N^r \rightarrow N^r$

$I = (\sigma \circ S)(i, j) = f_0(p_0, t(i, j)) \cdot d_1 + f_1(p_1, t(i, j))$

$= i' \cdot d_1 + j'$

mit der Zeilenstrukturierung f_0 mit Parameter p_0

und der Spaltenstrukturierung f_1 mit Parameter p_1

und einer gemeinsamen Transposition t (i, j) .

Die **verallgemeinerte Speicherzugriffsfunktion** (generalized storage access function, GSAF) ist

$$\gamma(i, j) = \left| \left[f_0(p_0, t(i, j)) \cdot d_1 + f_1(p_1, t(i, j)) \right] \cdot \frac{E}{w} \right| + b$$

mit : b = Basisadresse, E = Anzahl Bit eines Datenelements

w = Wortlänge im Speicher .

13.5.2.3. Variablenreferenzierung im STARLET

Das Befehlsformat sieht aus wie das einer 3-Adressmaschine

OPcode	VD3	VD2	VD1

nur sind die Felder Variablendeskriptoren, für das Ergebnis, VD3, den 1. Operanden, VD1 und den 2. Operanden VD2, d. h. Indices in einer Variablendeskriptorliste.

Die Variablendeskriptorliste enthält Elemente der Form

SD	d_0, d_1	Datenfeldpointer

VD	Struktur	Pointer in VD - Liste

S	wert

mit Datentyp-Attribut SD : Feld von Elementen eines elementaren Datentyps

und Dimensionsvektor d_0, d_1

und dem Zeiger auf den Anfang des Datenvektors

oder Strukturierungsangabe VD

und einen Pointer in die Variablendeskriptorliste

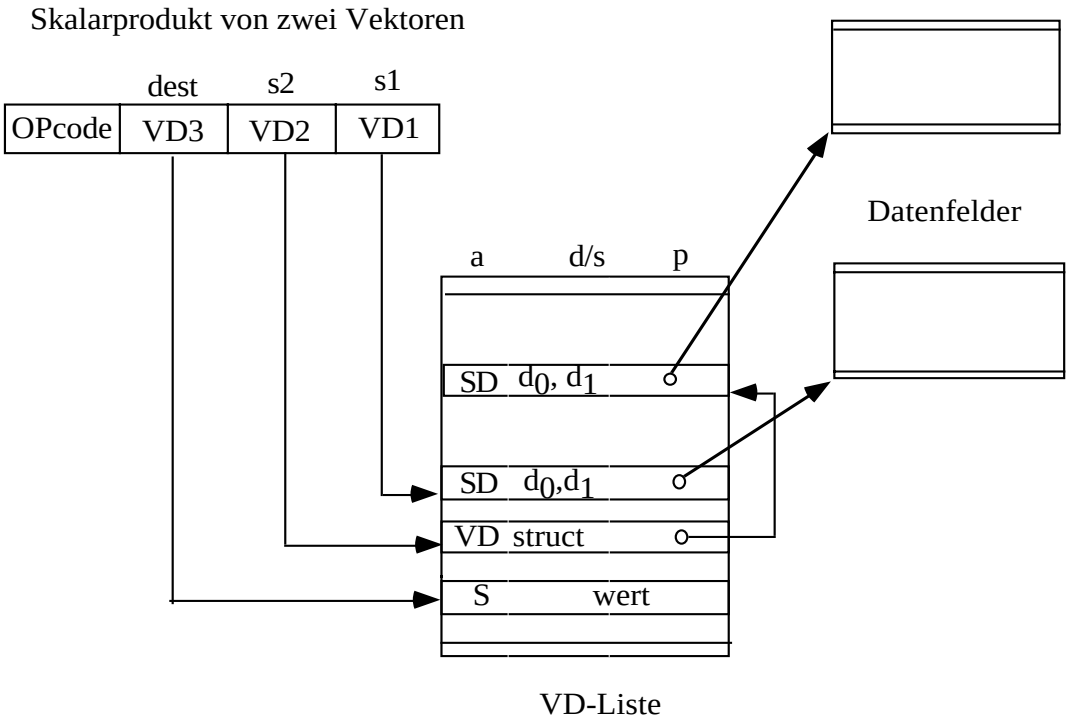
oder Kennzeichnung S als elementarer Datentyp

und Wert des Elementar-Datentyps

oder Kennzeichnung als Befehl

oder Kennzeichnung als "Typ" oder "Identifier"

Die folgende Abbildung zeigt als Beispiel die Operation "Skalarprodukt".



VD1 referenziert einen Vektor
VD2 referenziert indirekt einen Vektor
VD3 referenziert direkt einen Wert (skalare Variable)

Datentypen können sein:

Elementar-Datentypen S	"boolean", "integer", "real", "char"
Datentypen:	"type" T und "identifier" I
Struktur-Datentyp SD:	Vektor
Deskriptor-Datentyp VD:	Deskriptor
Befehls-Datentyp C:	Befehl .

Der Zugriff auf die Operanden über die Deskriptoren in der Vektordeskriptortabelle erfolgt unsichtbar für den Benutzer durch Hardware.

13.5.2.4. Vektoroperationen auf der STARLET

Zu den Datentypen sind jeweils Operatoren definiert. Insbesondere kennt der Struktur-Datentyp "Vektor" folgende (an die Sprache APL angelehnte) Operationen:

- Deklarationen
- Werteübertragung
- Konversion : ein beliebiger Elementar-Datentyp wird als Datentyp "boolean" interpretiert
- arithmetische Operationen, komponentenweise ausgeführt
  Grundrechenarten, Negation
  + - Reduktion (= Summe der Elemente)
  Skalarprodukt (= Summe der Produkte der Komponenten)

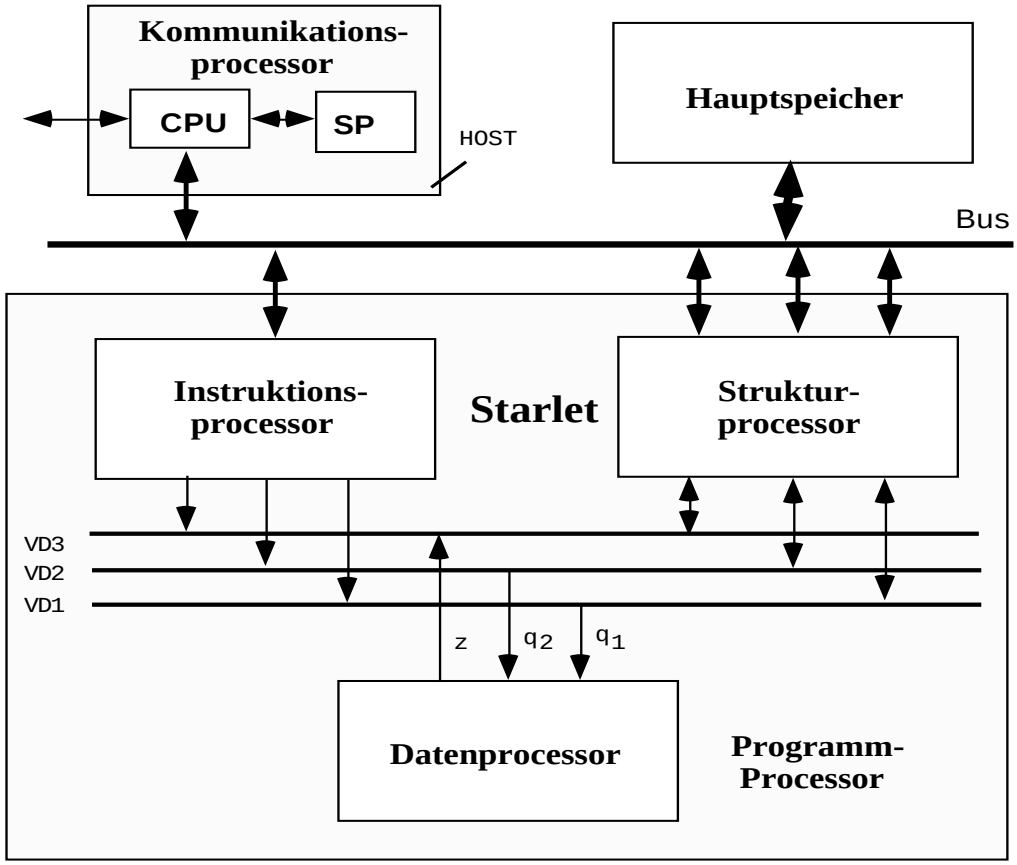
- Vergleichsoperationen
  $\neq, <, \leq, =, \geq, >$ komponentenweise angewandt.
  Das Resultat ist ein Bitvektor.
  L-Reduktion : das kleinste Element ist das Ergebnis
  Γ-Reduktion : das größte Element ist das Ergebnis.

- Suchoperationen : Suche in den Komponenten eines Vektors nach Gleichheit mit einem Skalar
  first occurance
  number of accurances
  all accurances.

- logische Operationen
  komponentenweise auf booleschen Vektoren : Negation, UND, ODER
  Λ-Reduktion : logisches Produkt aller Elemente
  $\vee$-Reduktion : logische Summe aller Elemente

- Selektionen
  Generierung eines Zielvektors durch Reihung von Elementen, die aus einem Quellvektor ausgewählt werden.
  SUBVEC : Untervektor vorgegebener Länge als vorgegebenes Element
  SAMPLE : Elemente in festem Abstand
  COMPRESS : Auswahl durch booleschen Vektor
  SELECT : Auswahl durch Angabe einer Indexmenge .

Den Aufbau der Maschine zeigt die folgende Abbildung.



Ein Kommunikationsprozessor aus CPU und Speicher SP bedient die Peripherie des Rechners. Er führt die Betriebssystemfunktionen aus und ist zuständig für die Programmübersetzung.

Im Hauptspeicher, gemeinsam genutzt vom Kommunikations- und Programmprozessor, stehen

- die Systemsoftware
- der Workspace mit
  - Keller (Stack)
  - Halde (Heap)
  - Programme
  - Datenfelder .

Der Programmprozessor führt die Programme aus, die aus einer Befehlsliste, einer Deskriptorliste und einer Datenliste bestehen.

Der Befehlsprozessor	interpretiert die Befehle
der Strukturprozessor	enthält die Adressgeneratoren zur Adresserzeugung aus den Deskriptoren
der Datenprozessor	enthält eine konfigurierbare Pipeline zur elementaren Datenverarbeitung.