MC SYLLABUS 45

COLLOQUIUM HOGERE PROGRAMMEERTALEN EN COMPUTERARCHITECTUUR

P. KLINT (red.)

MATHEMATISCH CENTRUM AMSTERDAM 1980

1980 Mathematics subject classification: 68-06, 68A05, 68B20, 68F99, 68G99

ACM-Computing Reviews-category: 4.21, 4.22, 4.35, 6.21, 6.22, 6.34

ISBN 90 6196 206 4

INHOUD

INHOUD i

VOORWOORD v

A. DE SYMBIOSE VAN PROGRAMMEERTAAL EN COMPUTER *door* P. Klint 1
0. Inleiding 1
1. Invloed van programmeertalen op architectuur 2
1.1. Von Neumann architectuur 2
1.2. Taal-gerichte architectuur 4
1.3. Conclusies 6
2. Invloed van architectuur op programmeertalen 7
2.1. Technologische vernieuwingen 7
2.2. Systemen met zichtbaar parallellisme 9
2.2.1. Vector- en arrayprocessors 9
2.2.2. Een programmeertaal voor vector- en arrayprocessors 10
2.3. Systemen met onzichtbaar parallellisme 12
3. Functionele talen en architecturen 13
3.1. Functionele programmeertalen 13
3.2. Overspecificatie 15
3.3. De Barton/Clark architectuur 17
4. Conclusie 20

B COMPUTERORGANISATIE VOOR STRINGMANIPULATIE *door* J. Heering 23
1. Inleiding 23
1.1. Algemeen 23
1.2. Stringmanipulatie-faciliteiten op de DEC VAX-11 24
2. De Fairchild SYMBOL computer 26
2.1. Inleiding 26
2.2. De SYMBOL brontaal 26
2.3. SYMBOL geheugenbeheer 29
2.3.1. De verschillende geheugenniveau's van SYMBOL 29
2.3.2. De geheugenbeheerder en de geheugenopruimer 31
2.3.3. De referentie-processor 35
3. Een onconventionele geheugen-organisatie 36
3.1. Algemene organisatie 36
3.2. Het descriptor-geheugen 38
3.3. De geheugenopruimer 39

4. Organisatie van systemen met grote (virtuele) geheugens 40
4.1. Inleiding 40
4.2. Implicaties van grote snelle geheugens voor systeem-organisatie 41
4.3. Integratie van commando- en programmeertaal 42
4.4. Implementatie 43
5. Conclusie 44

C. TOEPASSINGEN VAN ASSOCIATIEVE GEHEUGENS *door* W. Scheepens 47
1. Inleiding 47
2. Associatief datageheugen (APP) 49
3. Associatief werkgeheugen (AAM) 58
4. Conclusie 71

D. REALISATIE VAN ASSOCIATIEVE GEHEUGENS *door* C. Stork 75
1. Begripsomschrijving 75
2. Een realisatie 79
2.1. Kenmerkregister 79
2.2. Het associatief array 80
2.3. Response store 82
2.3.1. Tijdens het vergelijken 82
2.3.2. Tijdens het schrijven 83
2.3.3. Tijdens het lezen 83
2.3.4. Ripple-schakeling 84
3. Variaties op het principe 86
3.1. Partieel associatief geheugen 86
3.2. Het bitseriële associatieve geheugen 88
3.3. Andere vergelijkingskriteria 88
4. Praktische realisatie 90
4.1. De bitcel 90
4.2. De besturing (C.C.U.) 93

E. DATAFLOW COMPUTERS *door* A.H. Veen 99
1. In het begin 99
2. Motivatie 100
2.1. Hardware 100
2.2. Software 101

3. Machinetaal 102
3.1. Een eenvoudige graaf 102
3.2. Een vereenvoudigde machine 105
3.3. Overige instructies 107
4. Kringloop en kleuren 110
5. De Manchester dataflow machine 112
6. Funktionaliteit en stromen 115
7. Hogere programmeertalen 118
7.1. Lapse 120
7.2. Value-oriented algorithmic language 122
7.3. Irvine dataflow 124
8. Een hybride taal 125
9. Konklusies 129

F. LISP MACHINES *door* B. Wielinga 133
1. Inleiding 133
2. Architectuur van het LISP systeem 135
2.1. De struktuur van een typisch LISP systeem 135
2.2. Interlisp als LISP machine 136
3. LISP machines 138
3.1. LISP en de von Neumann machine 138
3.2. Niet specifieke LISP machines 140
3.3. Specifieke LISP machines: LISP chips 142
4. Multi-processor systemen 144
5. Konklusies 146

G. HOGERE PROGRAMMEERTALEN GEZIEN VANUIT DE MACHINETAAL *door* G.A. Blaauw 149
0. Inleiding 149
1. Termen 149
1.1. Hogere programmeertalen 149
1.2. Computer-architektuur 150
2. Von Neumann architektuur 151
2.1. Gemeenschappelijk geheugen 151
2.2. Binair grondtal 152
2.3. Registers 152
2.4. Adrestype 153
2.5. Indicering 153
2.6. Bewerkingsrepertoire en datatype 153

2.7. Bewerkingsspecificatie 154
2.8. Vaste-lengte-operands 154
2.9. Datastrukturen 154
2.10.Direkte geheugenadressering 154
2.11.Adresvertaling 155
2.12.Gelijktijdigheid 155
2.13.Conclusie 156
3. Machinetaal 157
3.1. De eentalige machine 157
3.2. Kenmerk van de machinetaal 157
3.3. Microprogrammering 157
3.4. Beschikbare microcode 158
3.5. Verborgen machinetaal 158
4. De omhoogstrevende uitrusting 158
4.1. Lage architektuur 158
4.2. Instruktiebit-besparing 159
4.3. De uitrusting kan het beter 159
5. De neerdrukkende programmatuur 159
5.1. Compilatie versus interpretatie 159
5.2. Som versus product 160
5.3. Ontkoppeling 160
5.4. Grootste gemene deler 160
5.5. De vertaler weet het beter 160
5.6. Mogelijkheden voor hoge talen 161
6. Conclusie 161

H. ONTWERPREGELS VOOR MACHINE-ARCHITECTUREN *door* R. Hoogerwoord 163
1. Inleiding 163
2. Het belang van "Separation of Concerns" 165
3. Het waarom van vertalers 169
4. Naamgeving van variabelen 171
5. Het inperken van combinatorische vrijheid 176
6. De rol van registers 180
7. Slotopmerkingen 182

VOORWOORD

In de eerste helft van 1980 is op het Mathematisch Centrum een serie lezingen gehouden rond het thema "Hogere Programmeertalen en Computerarchitectuur". De resultaten hiervan liggen nu voor U.

In de eerste voordracht ("De symbiose van programmeertaal en computer" door P. Klint) wordt een poging gedaan om de wederzijdse beïnvloeding van programmeertalen en computers te analyseren. Enige gevolgen van het introduceren van parallellisme worden nader onder de loupe genomen. Deze bijdrage is bedoeld als inleiding op het onderwerp van dit colloquium.

De tweede voordracht ("Computerorganisatie voor stringmanipulatie" door J. Heering) belicht de vraag hoe een computer georganiseerd moet worden om efficiënte operaties op strings mogelijk te maken. Grote aandacht wordt hierbij besteed aan de geheugenorganisatie in dergelijke computers.

De derde en vierde voordracht ("Toepassingen van associatieve geheugens" door W. Scheepens en "Realisatie van associatieve geheugens" door C. Stork) behandelen toepassing en realisatie van associatieve geheugens. Beide voordrachten benaderen deze geheugens vanuit het vakgebied van de digitale techniek. De voordracht van Scheepens behandelt enkele voorbeelden van het zoeken in een associatief (data)geheugen. Verder wordt ingegaan op consequenties van het gebruik van associatieve geheugens als programmageheugen. De voordracht van Stork behandelt verschillende manieren om associatieve geheugens te realiseren. Naast een meer theoretische beschouwing over ontwerpbeslissingen die hierbij genomen moeten worden, komt ook een specifieke realisatie van een, met schilfertechnologie geproduceerd, associatief geheugen ter sprake.

De vijfde voordracht ("Dataflow computers" door A.H. Veen) geeft een overzicht van de huidige stand van zaken op het gebied van dataflow computers. In dergelijke computers worden programmadelen uitgevoerd zodra alle benodigde gegevens bekend zijn en niet (zoals in het klassieke geval) wanneer dat programmadeel volgens een of andere sequentiële ordening aan de beurt is. De gevolgen van dit principe voor computerarchitectuur en hogere programmeertalen worden uitvoerig besproken.

In de zesde voordracht ("LISP machines" door B. Wielinga) komt de opzet van interactieve LISP systemen aan de orde. Momenteel worden speciale LISP computers ontworpen en gebouwd om aan de eisen van

dergelijke interactieve systemen en aan de steeds toenemende behoefte aan rekencapaciteit te voldoen. Enige recente ontwikkelingen op dit gebied worden besproken.

In de zevende voordracht ("Hogere programmeertalen gezien vanuit de machinetaal" door G.A. Blaauw) wordt een begrippenkader voor computerarchitectuur gedefinieerd. Verder wordt de ontwikkeling van steeds hoger niveau architecturen bekeken uitgaande van de noodzaak om verschillende hogere programmeertalen op dergelijke architecturen uit te voeren. Hieruit volgt de conclusie dat hogere programmeertalen uiteindelijk leiden tot verlaging van het niveau van de architectuur.

In de achtste en laatste lezing ("Ontwerpregels voor machinearchitecturen" door R. Hoogerwoord) wordt de invloed van concepten uit een hogere programmeertaal op het machinetaalniveau bestudeerd door een strenge scheiding te maken tussen architectuur en realisatie van een machine. Dit wordt geïllustreerd aan de hand van het begrip "variabele".

Tijdens lezing van de hierna volgende bijdragen zult u tientallen malen de uitdrukking "von Neumann computer" tegenkomen. Dit is wellicht de plaats om er nog eens op te wijzen dat het hier een historische vergissing betreft, aangezien alle concepten die een dergelijke computerorganisatie karakteriseren reeds eerder door Eckert en Mauchly geïntroduceerd waren. Blaauw noemt dit feit al in zijn bijdrage. Zie [1] voor een uitvoeriger beschrijving van de feitelijke loop der gebeurtenissen.

Paul Klint

[1] N. Metropolis & J. Worlton, A Trilogy of Errors in the History of Computing, Annals of the History of Computing, 2(1980) 1, 49-59.

DE SYMBIOSE VAN PROGRAMMEERTAAL EN COMPUTER

P. KLINT
Mathematisch Centrum

0. INLEIDING

Von Neumann's ontwerp was zo revolutionair, dat zelfs na ruim dertig jaar de werking van vrijwel alle computers gebaseerd is op principes die door hem bedacht zijn, en dat er pas de laatste jaren fundamentele kritiek op deze zogenaamde von Neumann architectuur is gekomen. Deze kritiek komt voort uit verbeterde inzichten in programmeertaalontwerp en uit impulsen van technologische vernieuwing.

Aan de ene kant hebben de voortdurend toenemende complexiteit en omvang van problemen die met behulp van computers opgelost worden ertoe geleid dat hogere programmeertalen steeds krachtiger primitieven gaan bevatten om de oplossing van bepaalde klassen problemen te vereenvoudigen. Tegelijkertijd wordt hierdoor echter de afstand tussen hogere programmeertaal en de onderliggende von Neumann architectuur steeds groter. Bovendien is er een conflict tussen het streven naar veiligheid en redundantie in grote programma's en de onveiligheid van de niet-redundante von Neumann architectuur. In de volgende sectie wordt beschreven hoe dit semantische gat overbrugd kan worden door complexiteit en niveau van architecturen af te stemmen op de eisen die door hogere programmeertalen gesteld worden.

Aan de andere kant leiden technologische vooruitgang en de daarmee samenhangende verandering van economische factoren ertoe dat herwaardering nodig is van ontwerpbeslissingen die op louter economische gronden genomen zijn. Dit betekent dat de verdeling van taken over hardware en software drastisch zal veranderen. Bovendien is een direct gevolg van de technologische vooruitgang zelf dat bepaalde oplossingen en methodes, die vroeger ondenkbaar waren nu tot de mogelijkheden gaan behoren. De impulsen die uitgaan van dergelijke vernieuwingen komen in sectie 2 ter sprake.

Merkwaardig genoeg leidt kritiek uit deze totaal verschillende bronnen vaak tot dezelfde conclusies. Het blijkt namelijk dat de psychologische factoren die de conceptuele complexiteit van programma's begrenzen

een directe tegenhanger hebben in technologische factoren die de complexiteit van computerarchitecturen begrenzen! In sectie 3 wordt hierop verder ingegaan.

1. INVLOED VAN PROGRAMMEERTALEN OP ARCHITECTUUR

Welke stappen zijn nodig om een, in een hogere programmeertaal geschreven, programma tot uitvoering te brengen op een computer met gegeven architectuur? Het is duidelijk dat het aantal stappen afneemt naarmate de afstand tussen hogere programmeertaal en computer architectuur kleiner is. Ik zal dit illustreren aan de hand van de von Neumann architectuur en taal-gerichte architecturen.

1.1. Von Neumann architectuur

De globale organisatie van een von Neumann computer is weergegeven in figuur 1.



Fig. 1. De von Neumann architectuur.

Men kan hierin drie onderdelen onderscheiden:

1) Een geheugen dat bestaat uit een lineaire rij cellen waarnaar wordt gerefereerd door middel van hun rangnummer ("adres") in de rij. Een cel kan zowel data als instructies bevatten. Bovendien is de interpretatie van de inhoud van een geheugencel niet onverbrekelijk met de celinhoud verbonden, maar hangt af van de context waarin de cel gebruikt wordt. Springen naar een floating-point getal en delen door een characterstring behoren zo tot de mogelijkheden.

2) Een centrale verwerkingseenheid (cpu), die één voor één cellen uit het geheugen inspecteert, hun inhoud als instructie opvat en deze vervolgens uitvoert. Hierbij kan de inhoud van andere cellen gewijzigd worden, kunnen aritmetische bewerkingen uitgevoerd worden, kan bepaald worden welke cel na uitvoering van de huidige instructie ter uitvoering geïnspecteerd moet worden en kunnen opdrachten aan randapparaten gegeven worden.

3) Randapparaten, die de communicatie tussen computer en buitenwereld regelen (schijven, magneetbanden, eindstations).

In de loop der jaren is de von Neumann architectuur uitgebreid en verbeterd (onder andere met registers, indirecte adressering, indexering, interrupts, asynchrone in- en uitvoer, virtueel geheugen,

geheugenbescherming en scheiding van instructies en data) maar de basisopzet is ongewijzigd gebleven.

Vergelijk nu eens de eigenschappen van de von Neumann architectuur met de eisen die hogere programmeertalen stellen:

- De toegang tot waarden is streng gereglementeerd; de regels hangen bovendien af van de organisatievorm van de waarden (arrays, records, etc.) en de plaats in het programma waar ze gedeclareerd respectievelijk gebruikt worden (blokstruktuur).

- De betekenis van waarden is onverbrekelijk met de waarden zelf verbonden.

- De aritmetiek in hogere programmeertalen is decimaal. De binaire aritmetiek van de von Neumann computer leidt tot afrondfouten en vereist veelvuldige conversie van en naar externe (decimale) representatie.

Dergelijke verschillen maken duidelijk hoe groot de afstand tussen hogere programmeertalen en von Neumann architectuur is. Het overbruggen van deze afstand leidt tot een sterke toename van de complexiteit (en ook onbetrouwbaarheid) van het computersysteem als geheel.

Figuur 2. geeft de verschillende stappen weer die meestal nodig zijn om een programma in een hogere programmeertaal (HT) uit te voeren op een von Neumann computer.

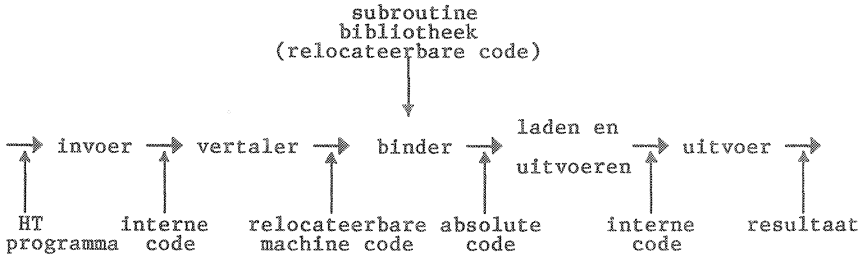


Fig. 2. Uitvoering op von Neumann computer.

Een bronprogramma wordt eerst door een invoerapparaat omgezet naar een interne code en vervolgens door een vertaler getransformeerd naar relocateerbare machinecode. Vervolgens wordt deze code door een binder ("linkage editor") aangevuld met de benodigde subroutines (sinus, invoer/uitvoer, etc.) tot absolute code die vervolgens geladen en uitgevoerd kan worden. Eventuele resultaten van de uitvoering van het

programma worden door een uitvoerapparaat omgezet van interne code naar een (extern) leesbare vorm.

Er is in deze opzet geen enkele overeenkomst meer tussen bronprogramma en uiteindelijk uit te voeren code. De programmeur wordt voortdurend met deze afstand tussen bronprogramma en uit te voeren programma geconfronteerd. In de allereerste plaats al door het feit dat er een "vertaal"-stap (wat is dat eigenlijk?) nodig is, en verder onder andere bij het opsporen van fouten in programma's. In de opzet zoals hierboven gepresenteerd dient dit te gebeuren met behulp van (hexadecimale) geheugendumps. Het is uiteraard zeer wel mogelijk om foutmeldingen en dergelijke te geven in termen van het oorspronkelijke programma, maar ook dit verhoogt de complexiteit van het totale systeem.

1.2. Taal-gerichte architectuur

De eerste stap op weg naar hoger niveau architecturen werd gezet door de introductie van een hardware stapel en een op postfix operaties gebaseerd instructierepertoire in de B5500 [4]. Bovendien zijn de adresseringsmechanismen in deze machine geheel gericht op de uitvoering van ALGOL 60 programma's, zoals onder andere blijkt uit het onderscheiden van code- en datasegmenten, relatieve adressering ten opzichte van segmenten, hardware ondersteuning van de ALGOL 60 blokstruktuur, voorzieningen voor de adressering van arrays van variabele lengte en het feit dat de hardware primitieve datatypes kent.

Een op segmenten gebaseerde geheugenorganisatie (zoals bijvoorbeeld in het Multics systeem [5] gerealiseerd is) maakt een aparte link-edit stap overbodig; referenties naar bibliotheeksubroutines worden tijdens uitvoering van het programma opgelost. Hierdoor vervalt de noodzaak om een onderscheid te maken tussen relocateerbare en absolute machinecode.

Figuur 3. geeft de stappen weer die nodig zijn om een HT programma uit te voeren op een taal-gerichte computer.

Een bronprogramma wordt via invoerapparaat en vertaler omgezet in (bijvoorbeeld) postfixcode. Aangezien het relatief eenvoudig is om postfixcode te genereren, kan een vertaler voor een taal-gerichte architectuur eenvoudiger zijn dan voor een von Neumann architectuur.

Het is mogelijk om de afstand tussen hogere programmeertaal en architectuur nog verder te verkleinen. Dit is bijvoorbeeld gebeurd in de SYMBOL [6] computer, die een hogere programmeertaal als machinetaal heeft. Deze architectuur voert programma's indirect uit, omdat er eerst vertaling naar een interne vorm plaats vindt.

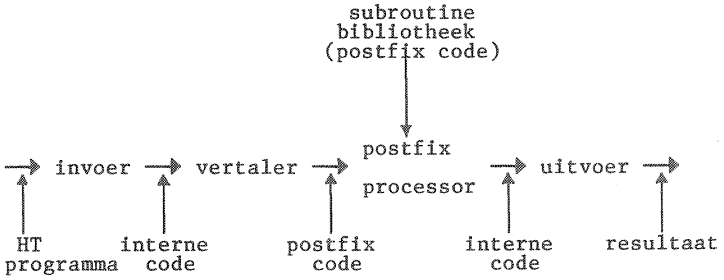


Fig. 3. Uitvoering op taal-gerichte computer.

De voornaamste verschillen met een architectuur zoals die van de B5500 zijn dat de postfixcode een kleinere lexicale en semantische afstand tot de hogere programmeertaal vertoont en dat de vertaler onderdeel van de architectuur als geheel uitmaakt. Deze is in het geval van de SYMBOL computer zelfs geheel in hardware uitgevoerd. In dit colloquium zullen twee voorbeelden van indirect uitvoerende architecturen behandeld worden (SYMBOL [7], LISP [8]).

De meest verregaande stap in de hier geschetste ontwikkeling is om hogere programmeertaal en machinetaal identiek te maken. In een dergelijke direct uitvoerende architectuur bestaat zelfs geen interne vertaalfase meer, maar worden HT programma's symbool voor symbool ingelezen en uitgevoerd. Figuur 4. geeft dit proces weer.

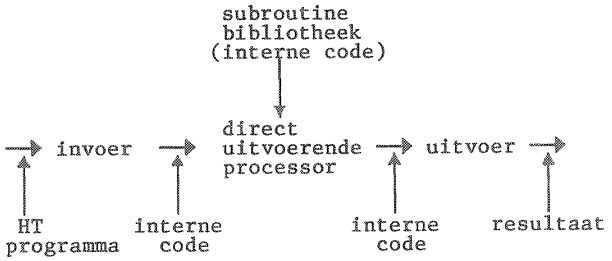


Fig. 4. Uitvoering op direct uitvoerende computer.

In een dergelijke architectuur is de lexicale, syntactische en semantische afstand tussen hogere programmeertaal en machinetaal minimaal. Het meest in het oog springende voordeel van deze architectuur is dat de totale complexiteit van het systeem (een complexiteit die zoals we hierboven al zagen voor de gebruiker duidelijk waarneembaar is) drastisch verminderd wordt. Hier staat tegenover dat in het algemeen analyse van een programma voorafgaande aan de feitelijke uitvoering gewenst is,

bijvoorbeeld om syntactische of semantische fouten op te sporen. Een dergelijke analyse kan echter ook onafhankelijk van de feitelijk uitvoering van het programma verricht worden.

1.3. Conclusies

Door ontwikkelingen in de richting van hoog niveau architecturen is meer bereikt dan dat de complexiteit bóven het machinetaalniveau is afgenomen en de complexiteit daaronder is toegenomen. De volgende overwegingen maken dit duidelijk:

1) Vertalers worden eenvoudiger en sneller.

2) De betrouwbaarheid van het totale computersysteem neemt toe.

3) Door het niveau van tussencodes te verhogen wordt de uiteindelijke interne representatie van programma's kleiner. Hierdoor wordt het gegevensverkeer tussen centrale processor en geheugen verminderd en de capaciteit dienovereenkomstig vergroot. Om dezelfde reden hoeven voor de realisering van constructies in de hogere taal minder instructies uitgevoerd te worden.

4) Door hoger niveau adresseringsmechanismen kunnen veel adresberekeningen en onder andere de controle op het overschrijden van arraygrenzen door de hardware uitgevoerd worden. Bovendien vermindert de informatie die nodig is om effectieve adressen van variabelen te specificeren.

5) Het opsporen van fouten in programma's wordt vereenvoudigd.

Nu het machinetaalniveau is aangepast aan het hogere programmeertaalniveau, kan men zich afvragen of er niet nog meer taalniveaus in een computersysteem zijn die beter op elkaar afgestemd kunnen worden. Drie candidaten die zich hiervoor lenen zijn:

- commandotalen, waarmee opdrachten aan het bedrijfssysteem gegeven worden,

- editortalen, waarmee wijzigingen in teksten aangebracht worden, en

- vraagtalen, waarmee gegevensbestanden doorzocht worden.

Het is momenteel niet duidelijk in hoeverre één taal kan voldoen aan eisen die door deze verschillende toepassingsgebieden gesteld worden. Vergelijk bijvoorbeeld zeer compacte, onleesbare en éénmaal gebruikte editor-opdrachten met leesbare, vaak gebruikte programmateksten.

2. INVLOED VAN ARCHITECTUUR OP PROGRAMMEERTALEN

Een paar voorbeelden van de invloed van architectuur op programmeertalen zijn:

1) Nadruk op het gebruik van arrays en pointers.

2) Onderscheid tussen intern en extern geheugen, d.w.z. het verschil tussen het wijzigen van een array-element en wijzigen van een gegevensbestand.

3) Eindige geheugencapaciteit (overlays).

4) Overspecificatie van evaluatievolgorde en nadruk op sequentiële uitvoering.

5) Het beschouwen van "variabelen" als hokjes waar een waarde ingestopt en weer uit gehaald kan worden.

6) Operaties voor de synchronisatie van parallelle processen, voorzover dergelijk parallellisme niet essentieel samenhangt met de probleemstelling.

7) Onderscheid tussen verschillende soorten geheugentoekenning (stapel, hoop).

8) Zoveel mogelijk vermijden van objecten van variabele grootte; eindige integercapaciteit en bijvoorbeeld hiermee samenhangende taalconstructies zoals ON condities in PL/I.

9) Vermijden van associatief zoeken.

Deze opsomming geeft een indruk hoe uiteenlopend de invloed van computerarchitectuur op programmeertalen is. Daarnaast is zelfs het hele berekeningsmodel van hogere programmeertalen door de von Neumann architectuur bepaald. Hierop kom ik in sectie 3 nog nader terug. Eerst worden nu een aantal recente technologische vernieuwingen en ontwikkelingen gesignaleerd, die invloed (kunnen) hebben op toekomstige architecturen en in tweede instantie op toekomstige programmeertalen.

2.1. Technologische vernieuwingen

De spectaculaire ontwikkeling van de micro-electronica in de afgelopen jaren heeft mogelijkheden binnen bereik gebracht die tot voor kort ondenkbaar waren. Ik zou deze mogelijkheden globaal willen karakteriseren als het introduceren van "gedistribueerde intelligentie". De

huidige fabricagemethodes zijn namelijk uitermate geschikt om systemen te construeren die uit veel identieke componenten bestaan. De maximale complexiteit per component heeft momenteel de complexiteit van de klassieke von Neumann architectuur al geëvenaard. Deze ontwikkeling wijst in de richting van:

1) Multiprocessorsystemen met een (potentieel) hoge graad van parallellisme.

2) Associatieve geheugens, waarin iedere geheugencel een aparte processor bevat. Dit brengt ook databasecomputers met inhoudadressseerbare achtergrondgeheugens binnen bereik.

3) Intelligente eindstations met ingebouwde teksteditors en eigen geheugen. Uiteindelijk, hoog niveau persoonlijke computers met grafische faciliteiten (zoals bijvoorbeeld het Smalltalk [9] systeem van Xerox).

We zullen nu nagaan welke invloed deze ontwikkeling in de richting van "een hoge graad van parallellisme door middel van veel identieke componenten" kan hebben op hogere programmeertalen. Men kan hierbij een onderscheid maken tussen systemen waarin het parallellisme voor de gebruiker zichtbaar is en systemen waarin dit niet het geval is. In termen van een hogere programmeertaal kan dit onderscheid als volgt geformuleerd worden:

de parallelle operaties zijn direct beschikbaar in de hogere programmeertaal

of de programmeertaal bevat uitsluitend sequentiële (en geen parallelle) constructies en een compiler detecteert inherent parallellisme in programma's en beeldt dit af op parallelle operaties van de gebruikte computer.

De eerste methode leidt tot efficientere programma's maar heeft als nadeel dat programma's machine-afhankelijk worden. Bovendien wordt de programmeur voor aanzienlijke problemen geplaatst als hij een graad van parallellisme nodig heeft die de capaciteit van de gebruikte computer te boven gaat. De tweede methode heeft als nadeel dat parallellisme moeilijk (statisch) te detecteren is en dat op deze manier gegenereerde code minder efficiënt is. De programmeur moet zijn programma vaak herstructureren opdat de compiler meer parallellisme kan ontdekken!

2.2. Systemen met zichtbaar parallellisme

2.2.1. Vector- en arrayprocessors

Vector- en arrayprocessors zoals de CRAY-1 [10] kunnen gezien worden als een tussenschakel in de hierboven geschetste ontwikkeling. De problemen die zich voordoen bij de programmering van dergelijke machines kunnen daarom leerzaam zijn en misschien geëxtrapoleerd worden.

Globaal gesproken, bevat een dergelijke computer een aantal vectorregisters $V_1, \ldots, V_N$. Ieder vectorregister kan maximaal M reële getallen $V_i[1], \ldots, V_i[M]$ bevatten. Typische waarden voor N en M zijn 8 en 64. Op deze vectorregisters zijn operaties gedefinieerd zoals

$$V_i[1{:}s] := V_j[1{:}s] * V_k[1{:}s]$$

wat equivalent is met

$$\text{FOR } p := 1 \text{ TO } s \text{ DO } V_i[p] := V_j[p] * V_k[p] \text{ OD}$$

Ook kunnen parallel operaties uitgevoerd worden op een deelverzameling van de vectorelementen. Dit komt overeen met

$$\text{FOR } p := p_1, \ldots, p_s \text{ DO } V_i[p] := V_j[p] * V_k[p] \text{ OD}$$

Niet genoemde vectorelementen blijven hierbij ongewijzigd.

De CRAY-1 is in staat om tot maximaal 250 miljoen floating-point operaties per seconde uit te voeren. Figuur 5. laat executietijden zien voor de uitvoering van enkele typische FORTRAN DO loops.

Executietijd in nanoseconden per geproduceerd resultaat voor het statement

DO 10 I = 1,N
10 "loop body"

loop body	N=1	N=10	N=100	N=1000	1000 scalair
A(I)=1.0	512.5	68.8	32.5	31.3	281.3
A(I)=B(I)+C(I)	737.5	102.5	48.8	46.3	512.5
A(I)=SIN(B(I))	5775.0	762.5	416.3	392.5	2476.3

Fig. 5. Typische executietijden CRAY-1.

Alleen voor korte vectoren met minder dan vier elementen loont het de moeite om scalaire operaties te gebruiken. Dit heeft te maken met de tijd die nodig is voor het initialiseren van de vectorregisters. Bij gebruik van vectoroperaties wordt de executietijd per berekend resultaat

10 maal zo klein als bij het gebruik van scalaire operaties, terwijl de CRAY-1 met zijn cyclustijd van 12.5 nanoseconde al de snelste scalaire processor ter wereld is!

2.2.2. Een programmeertaal voor vector- en arrayprocessors

Actus [11] is een op Pascal gebaseerde programmeertaal voor vector- en arrayprocessors, waarin de programmeur controle heeft over de graad van parallellisme in zijn programma zonder dat hij zich hoeft bezig te houden met de soms bizarre eigenschappen van een specifieke processor. De graad van parallellisme komt tot uiting in declaraties en controlestructuren.

Een scalair array van M bij N getallen wordt gedeclareerd door:

```
VAR scalara : ARRAY[1..M, 1..N] OF integer
```

Een graad van parallellisme wordt geïntroduceerd door één paar arraygrenzen te noteren als "a:b" in plaats van "a..b". Zo definieert

```
VAR parb : ARRAY[1:M, 1..N] OF real
```

een array van M bij N getallen met een maximale graad van parallellisme ter grootte M. Dit betekent dat tegelijkertijd op maximaal M elementen van parb bewerkingen uitgevoerd kunnen worden. Op vergelijkbare wijze kan men bij het gebruik van arrayvariabelen naar een aantal elementen tegelijkertijd refereren. Zo komt bijvoorbeeld

```
a[1:4, 2] := 0
```

overeen met

```
a[1,2] := 0; a[2,2] := 0; a[3,2] := 0; a[4,2] := 0
```

Dit mechanisme is verder verfijnd door de introductie van zogenaamde indexverzamelingen, die het mogelijk maken om bij ieder gebruik van een "parallelle" variabele aan te geven op welke elementen tegelijkertijd operaties uitgevoerd moeten worden. De indexverzameling

```
INDEX even = 2:(2)100;
```

laat bijvoorbeeld operaties toe op alle arrayelementen met even index. Er zijn verschillende bewerkingen gedefinieerd op dergelijke indexverzamelingen: vereniging, doorsnede, verschil, complement, verschuiving en

rotatie. De eerste vier bewerkingen hebben een voor de hand liggende betekenis. Verschuiving (SHIFT) leidt tot verplaatsing van arrayelementen binnen de grenzen van de gedeclareerde graad van parallellisme en rotatie (ROTATE) leidt tot circulaire verschuiving. Als bijvoorbeeld gedeclareerd zijn:

```
VAR parb : ARRAY[1:100] OF integer
INDEX helftl = 1:50;
```

dan heeft

```
parb[helftl] + parb[helftl SHIFT 1]
```

de betekenis dat de elementen 1 tot 50 van parb (parallel) opgeteld worden bij de elementen 2 tot 51, terwijl in

```
parb[helftl] + parb[helftl ROTATE - 1]
```

element 1 en element 50 bij elkaar opgeteld worden, en verder elementen 2 tot 50 respectievelijk bij elementen 1 tot 49 opgeteld worden.

De controlestructuren zijn ook aangepast om manipulatie van de graad van parallellisme mogelijk te maken. Alleen het IF statement zal hier nu ter spake komen. Als de boolean expressie van een IF statement parallelle variabelen bevat dan wordt deze expressie uitgevoerd voor alle aangegeven elementen van de parallelle variabelen. Alle boolean expressies die TRUE opleveren definiëren samen een nieuwe graad van parallellisme die binnen het THEN gedeelte van het IF statement gebruikt kan worden (dit wordt aangegeven door het symbool "#"). Bijvoorbeeld in

```
IF a[0:49] > b[0:49] THEN a[#] := a[#] - 1;
```

worden alle 50 elementen van array a paarsgewijs vergeleken met de elementen van array b; de waarde van die elementen van a waarvan de waarde groter is dan die van het corresponderende element van b worden vervolgens met 1 verminderd.

Deze beschrijving doet vanwege zijn beknoptheid uiteraard geen recht aan de ideeën die aan Actus ten grondslag liggen. Toch kunnen we proberen om een oordeel uit te spreken over de invloed van deze benadering van parallellisme op de totale complexiteit van een systeem. Als men de eigenschappen van individuele arrayprocessors vergelijkt met de abstracties die Actus ter beschikking stelt dan is het duidelijk dat de programmeur zich niet meer hoeft te bekommeren om allerlei laag niveau details

die voor de oplossing van zijn probleem niet van belang zijn. Vanuit dit gezichtspunt is de gekozen benadering effectief en kan succes hebben. Hier staat echter tegenover dat de programmeur nu te maken krijgt met hoog niveau details die nog steeds weinig of niets met zijn probleem te maken hebben. Samenvattend, leidt dit tot vermindering maar niet tot opheffing van de complexiteit die voortvloeit uit het gebruik van parallelle processors.

2.3. Systemen met onzichtbaar parallellisme

In sectie 2.1. werd al opgemerkt dat het moeilijk is om door middel van statische analyse inherent parallellisme in programma's te ontdekken. Bovendien zijn dergelijke analyses noodzakelijkerwijs pessimistisch en kan parallellisme zelden maximaal benut worden.

Voorbeelden van inherent parallellisme in conventionele programmeertalen zijn:

```
a := (b+1) * (c-1) + d[i]
```
(b+1), (c-1) en de adresberekening van d[i] kunnen parallel uitgevoerd worden.

```
a := sin(b) + cos(c)
```
de functies sin en cos kunnen parallel uitgevoerd worden.

```
FOR i := 1 TO n DO a[i] := b[i] * c[i] OD
```
de berekeningen per iteratie kunnen parallel uitgevoerd worden; bovendien kunnen binnen iedere iteratie de adressen van a[i], b[i] en c[i] tegelijkertijd berekend worden.

De mate van inherent parallellisme wordt fundamenteel begrensd door het berekeningsmodel dat ten grondslag ligt aan een programmeertaal. Bewerkingen worden meestal gedefinieerd in termen van veranderingen in de programmatoestand vóór en ná de bewerking. Aangezien de programmatoestand zelf een complexe grootheid is wordt het al snel onmogelijk om vast te stellen of een aantal bewerkingen elkaars effect op de programmatoestand kunnen beïnvloeden. Zelfs als het mogelijk is om dit vast te stellen en de genoemde bewerkingen tegelijkertijd uitgevoerd kunnen worden dan nog is het nodig om na afloop de verschillende (deel)toestanden te combineren tot één (eind)toestand. Dit leidt tot onnodige begrenzing van parallellisme. Het vergt niet veel fantasie om dit op toestandsovergangen gebaseerde berekeningsmodel en de von Neumann architectuur met elkaar in verband te brengen. De grote boosdoener is het assignment statement (vergelijk: het geheugen in de von Neumann architectuur) dat neveneffecten en onnodige sequentialisering tot gevolg heeft.

Het onderzoek naar programmeertalen die niet aan bovenstaande beperkingen onderhevig zijn beweegt zich in twee richtingen:

1) Dataflowtalen. Hierin staat centraal dat een expressie uitgerekend kan worden zodra de waarden van alle operanden bekend zijn. Vergelijk dit met de situatie in "von Neumann talen" waarin een expressie pas berekend kan worden zodra de waarde van de vorige expressie berekend is.

2) Functionele talen. Hierin wordt een op de lambdacalculus gebaseerd berekeningsmodel gebruikt.

Dataflowtalen en functionele talen zijn op vergelijkbare principes gebaseerd alleen wordt in eerstgenoemde talen geprobeerd om de gelijkenis met conventionele programmeertalen zoveel mogelijk te handhaven, bijvoorbeeld door een beperkte vorm van assignment statement toe te laten (zogenaamde "single assignment" talen). Dataflowtalen en -computers worden in [12] uitvoerig behandeld. Functionele talen en functionele computers komen hieronder aan de orde.

3. FUNCTIONELE TALEN EN ARCHITECTUREN

De invloed die functionele programmeertalen en functionele computers op elkaar hebben is dusdanig dat met recht van "symbiose" gesproken kan worden. Enerzijds zijn functionele programmeertalen gebaseerd op een klein aantal wiskundige begrippen en leggen de programmeur strenge beperkingen op ten aanzien van het gebruik van variabelen en controlestructuren. Anderzijds wordt de drang om deze beperkingen van functionele talen niet op te heffen gedeeltelijk door architectuuroverwegingen ingegeven. De functionaliteit van dergelijke talen maakt het namelijk mogelijk om speciale architecturen te ontwerpen die in hoge mate van parallellisme gebruik (kunnen) maken. Sectie 3.3. behandeld een dergelijke architectuur voor functionele talen.

3.1. Functionele programmeertalen

Een functionele programmeertaal, zoals bijvoorbeeld door Backus [13] gepropageerd, is gebaseerd op lambda expressies met een geassocieerde substitutieregel voor variabelen. Bovendien kunnen functionele vormen gebruikt worden om bestaande functies tot nieuwe functies te combineren. Enkele voorbeelden van dergelijke functionele vormen zijn:

compositie $(f.g) : x = f : (g : x)$

insertie
$$/f : x = \text{IF } x \simeq \langle x_1 \rangle \text{ THEN } x_1$$
$$\text{ELIF } x \simeq \langle x_1, \ldots, x_n \rangle \ \&\ n \geq 2$$
$$\text{THEN } f : \langle x_1, /f : \langle x_2, \ldots, x_n \rangle \rangle$$
$$\text{ELSE undefined}$$
$$\text{FI}$$

constructie
$$[f_1, \ldots, f_n] : x = \langle f_1:x, \ldots, f_n:x \rangle$$

allemaal
$$@f : x = \text{IF } x \simeq \text{null then null}$$
$$\text{ELIF } x \simeq \langle x_1, \ldots, x_n \rangle$$
$$\text{THEN } \langle f:x_1, \ldots, f:x_n \rangle$$
$$\text{ELSE undefined}$$
$$\text{FI}$$

Toepassen van functie "f" op argument "x" wordt aangegeven door "f:x", terwijl "$\langle x_1, \ldots, x_n \rangle$" een rij waarden en "$[f_1, \ldots, f_n]$" een rij functies aangeeft. "a $\simeq$ b" controleert structurele gelijkheid (is a van de vorm b?). Bij wijze van voorbeeld volgen nu definitie en berekening van een inprodukt in een klassieke programmeertaal en in een functionele programmeertaal.

In een klassieke programmeertaal ziet een programma voor de berekening van het inprodukt van vectoren a en b er alsvolgt uit:

```
resultaat := 0;
FOR i := 1 STEP 1 UNTIL n
DO resultaat := resultaat + a[i] * b[i] OD
```

In een functionele taal ziet de definitie er uit als

```
Inprodukt = (/+) . (@*) . Transp
```

Toepassing van deze functie op <<1,2,3>,<6,5,4>> leidt tot de volgende berekeningsstappen:

```
                                  Inprodukt:<<1,2,3>,<6,5,4>>
definitie van Inproduct    => (/+).(@*).Transp:<<1,2,3>,<6,5,4>>
effect van compositie (.) => (/+):((@*):
                                      (Transp:<<1,2,3>,<6,5,4>>))
pas Transp toe             => (/+):((@*):<<1,6>,<2,5>,<3,4>>)
effect van allemaal (@)    => (/+):<*:<1,6>, *:<2,5>, *:<3,4>>
pas * toe                  => (/+):<6,10,12>
effect van insertie (/)    => +:<6, +:<10,12>>
pas + toe                  => +:<6,22>
pas + toe                  => 28
```

Uit dit voorbeeld zal duidelijk zijn dat hier geen complexe toestandsovergangen aanwezig zijn en dat de berekening gebaseerd is op een klein aantal eenvoudige regels. Het is ook duidelijk dat een dergelijk berekeningsmodel een hoge graad van parallellisme toelaat zonder dat dit voor de programmeur zichtbaar is.

Functionele talen persen de programmeur in een keurslijf dat vaak knelt. Ze hebben echter als voornaamste bezwaar dat de oplossing van geschiedenis-afhankelijke problemen moeilijk is. Om deze reden wordt dan ook gezocht naar talen die lokaal gezien functioneel zijn, maar die toch globale toestandsveranderingen kunnen beschrijven.

3.2. Overspecificatie

Implementaties van niet-functionele talen kunnen maar in beperkte mate van parallellisme gebruik maken, met name doordat mogelijke neveneffecten de vaststelling van een éénduidige berekeningsvolgorde vereisen. Zo kan de waarde van

```
f(x, g(3), h(x))
```

afhangen van de berekeningsvolgorde van de argumenten indien de functies g en h de waarde van de variabele x kunnen wijzigen. *) In functionele talen kan deze situatie zich principieel nooit voordoen omdat iedere berekeningsvolgorde (en ook de "volgorde" waarbij alle expressies tegelijkertijd berekend worden) hetzelfde resultaat oplevert.

*) Sommige programmeertalen verbieden het gebruik van procedures met neveneffecten in expressies of definiëren de waarde van dergelijke expressies als "ongedefinieerd". Deze benadering levert echter geen nieuwe gezichtspunten op voor de huidige beschouwing over het gebruik van parallellisme.

Tot nu toe is alleen overspecificatie van de volgorde van controlestructuren aan de orde gekomen. Hoe is het met datastructuren gesteld? Beschouw lineaire lijsten die voldoen aan:

Als L een lijst $\langle e_1, \ldots, e_n \rangle$ is dan geldt
cons(e, L) = $\langle e, e_1, \ldots, e_n \rangle$
car(L) = e_1
cdr(L) = $\langle e_2, \ldots, e_n \rangle$

Het is eenvoudig in te zien dat de argumenten van de lijstconstructor cons nooit onmiddellijk geëvalueerd hoeven worden, mits de programmeertaal functioneel is. De berekening van deze argumenten kan uitgesteld worden tot het moment dat de waarde van het corresponderende lijstelement nodig is. Deze methode heet luie evaluatie. Men kan nog een stap verder gaan door zelfs de volgorde van de lijstelementen te laten afhangen van het gebruik van de lijst. Deze interpretatie verandert de betekenis van cons en de zo verkregen datastructuur is uiteraard geen lineaire lijst meer maar een verzameling waarin hetzelfde element meermalen voor kan komen ("bag"). De volgorde kan helemaal bepaald worden door elementen waarvan de waarde (inmiddels) al berekend is vóór aan de lijst te plaatsen en elementen waarvan de berekening nog niet voltooid is achteraan. Op deze manier kunnen lijsten zelfs elementen bevatten waarvan de berekening nooit eindigt (toepassing: oneindige verzamelingen). Figuur 6. geeft een overzicht van de drie manieren om tegen de constructor cons aan te kijken.

	bepaling inhoud	bepaling volgorde	berekening inhoud
cons (LISP)	moment van constructie	moment van constructie	moment van constructie
cons (lui)	moment van constructie	moment van constructie	moment van gebruik
cons (verzameling)	moment van constructie	moment van gebruik	moment van gebruik

Fig. 6. Drie interpretaties van cons.

De ervaring met deze verschillende methodes [15] is nog gering, maar het valt te verwachten dat het principe van minimale volgorde-specificatie invloed zal hebben op de verdere ontwikkeling van functionele talen.

3.3. De Barton/Clark architectuur

De Barton/Clark architectuur [14] heeft tot doel om een functionele computer te realiseren met behulp van één soort, herhaalde cellen, die onderling lineair verbonden zijn. In 2.1. werd al opgemerkt dat het gebruik van véél, identieke componenten zeer aantrekkelijk is vanuit technologisch oogpunt. Minimaliseren van het aantal onderlinge verbindingen is daarbij van minstens even groot belang. In principe kan een rij lineair verbonden cellen, samen met een arbitrage- en distributienetwerk (voor de werkverdeling en retournering van resultaten) gecombineerd worden tot een volledig systeem. De Barton/Clark cel (zie figuur 7) is bedoeld om een cel te verkrijgen die niet alleen berekeningen uitvoert maar ook voor communicatie zorgdraagt.

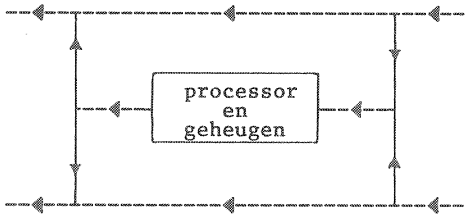


Fig. 7. De Barton/Clark cel.

De processor in iedere cel voert een functionele machinetaal uit die voldoende dicht bij de brontaal staat om de afbeelding van brontaal naar machinetaal eenvoudig te maken. Het lokale geheugen in iedere cel wordt gebruikt voor een stapel die dient voor het uitvoeren van functieaanroepen en onthouden van gedeeltelijk berekende waarden. Zelfs de machinetaal bevat geen instructies die directe toegang tot het geheugen verschaffen; naar waarden kan alleen gerefereerd worden door middel van, intern gegenereerde padnamen. Op betekenis en gebruik van dergelijke padnamen kom ik nog terug.

Evenals in het geval van functionele talen zijn met de machinetaal een aantal functionele vormen geassocieerd zoals compositie, constructie en selectie. Compositie maakt gelijktijdige berekening volgens het "lopende band"-principe (pipelining) mogelijk (figuur 8) en constructie laat gelijktijdige berekening door parallellisme toe (figuur 9).

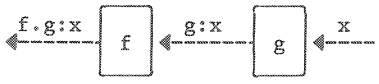


Fig. 8. Functionele compositie en pipelining.

Iedere cel verwerkt zijn invoer sequentieel maar "lopende band"

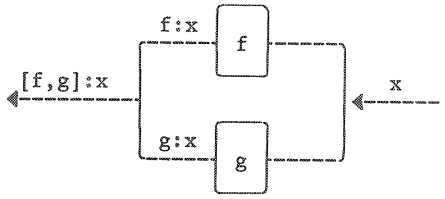


Fig. 9. Functionele constructie en parallellisme.

bewerkingen en parallelle bewerkingen worden automatisch doorgestuurd naar volgende cellen. Deze delegatie van taken wordt volledig bepaald door de functionele structuur van het programma en vereist geen bijzondere voorzieningen, zoals een bedrijfssysteem dat asynchrone processen beheert. Merk op dat deze (conceptuele) schema's voor compositie en constructie gerealiseerd moeten worden door expliciete adressering (padnamen) van cellen in de lineaire keten.

Het communicatienetwerk in de Barton/Clark cel is opgebouwd uit twee meer primitieve cellen met complementaire betekenis. Een selector (zie figuur 10) heeft één ingang en twee uitgangen. Een binaire boodschap op de ingang wordt, afhankelijk van de waarde van de eerste component, naar één van beide uitgangen gestuurd. Hierbij wordt de eerste component van de boodschap verwijderd.

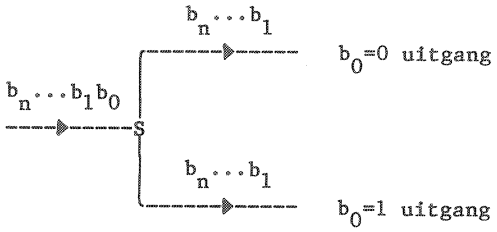


Fig. 10. Selector.

Een arbiter (zie figuur 11) heeft twee ingangen en één uitgang. Een boodschap op één van beide ingangen wordt naar de uitgang doorgestuurd. Hierbij wordt aan het einde van de boodschap een extra component toegevoegd, die aangeeft van welke ingang de boodschap afkomstig was. Figuur 12. geeft weer hoe een Barton/Clark opgebouwd kan worden uit selectors en arbiters.

De padnaam speelt dezelfde rol als het "adres" in de von Neumann architectuur. Markant verschil hierbij is dat een padnaam bepaald wordt door de cel die een vraag beantwoordt en niet door de cel die de vraag

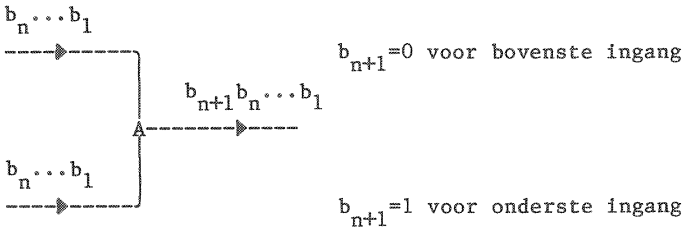


Fig. 11. Arbiter.

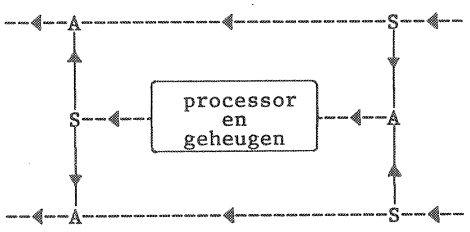


Fig. 12. Communicatie in Barton/Clark cel.

stelt. Arbiters en selectors worden gebruikt voor de codering en decodering van padnamen. Als voorbeeld van een dergelijke "adresberekening" laat ik nu zien hoe een boomgestructureerd geheugen opgebouwd kan worden uit arbiter/selector paren (figuur 13).

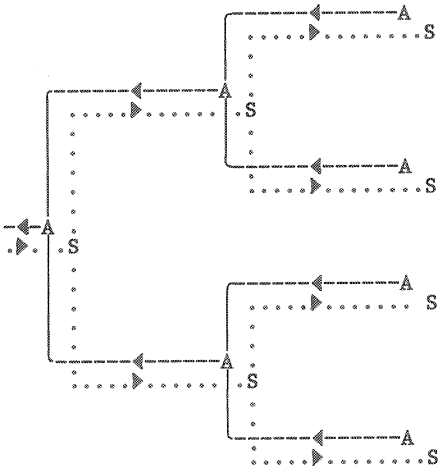


Fig. 13. Adressering met arbiter/selector paren.

Als een blad een boodschap stuurt naar de wortel van de boom, dan wordt deze boodschap automatisch voorzien van een achtervoegsel dat het hele

pad van blad naar wortel beschrijft. Dit achtervoegsel kan gebruikt worden om een boodschap van de wortel naar hetzelfde blad terug te sturen.

Met behulp van een dergelijk adresseringsmechanisme kan iedere cel verzoeken naar zijn buren sturen om bepaalde berekeningen uit te voeren. Als een andere cel besluit om aan dat verzoek gehoor te geven, dan is de inmiddels opgebouwde padnaam precies voldoende om een antwoord naar de oorspronkelijke cel terug te sturen.

In de Barton/Clark architectuur is volledige decentralisatie van taken bereikt. Een individuele cel bezit alleen kennis over zichzelf, zijn directe buren en over de vragen die hij aan andere, anonieme cellen gesteld heeft. Berichten worden in serie verstuurd, waardoor de lengte ervan in principe onbegrensd kan zijn, maar de verwerking van berichten geschiedt parallel. Expressies worden berekend zodra hun waarde nodig is ("demand-driven"), in tegenstelling tot het standaard dataflow-model waarin expressies berekend worden zodra alle daarvoor benodigde gegevens beschikbaar zijn ("data-driven", zie [12]).

4. CONCLUSIE

De voorafgaande (noodzakelijk schetsmatige) beschouwing over de onderlinge verstandhouding tussen computers en programmeertalen laat maar één conclusie toe: bij ieder verder onderzoek op deze beide gebieden moet de vraag centraal staan hoe de totale complexiteit van computersystemen verkleind kan worden, anders zullen we nooit het stadium van "computers ondanks programmeertalen, programmeertalen ondanks computers" te boven komen. Het feit dat onderzoekers op beide gebieden zich langzaam maar zeker van elkaars bestaan bewust lijken te worden, geeft enige grond voor de verwachting dat een "symbiose van programmeertaal en computer" misschien toch verwezenlijkt kan worden.

LITERATUUR

[1] Myers, G.J., Advances in computer architecture, John Wiley, 1978.

[2] Chu, Y. (Ed.), High-level language computer architecture, Academic Press, 1975.

[3] Foster, C.C., Content adressable parallell processors, Van Nostrand Reinhold, 1976.

[4] Organick, E.I., Computer system organisation. The B5700/B6700 series. Academic Press, 1973.

[5] Organick, E.I., The MULTICS system, MIT Press, 1972.

[6] Smith, W.R, et. al., SYMBOL - A large experimental system exploring major hardware replacement of software, AFIPS Spring Joint Computer Conference, 38(1971) 601-616.

[7] Heering, J., Computerorganisatie voor stringmanipulatie (dit colloquium).

[8] Wielinga, B., Lisp machines (dit colloquium).

[9] Ingalls, D.H.H., The Smalltalk-76 programming language design and implementation, Conference Record of the Fifth Annual ACM symposium on Principles of Programming Languages, Jan. 1977, 9-15.

[10] Russell, R.M., The CRAY-1 computer system, Communications of the ACM 21(1978) 63-72.

[11] Perrott, R.H., A language for Array and Vector processors, Transactions on Programming Languages and Systems (TOPLAS) 1(1979), 177-195.

[12] Veen, A.H., Dataflow computers (dit colloquium).

[13] Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Communications of the ACM, 21(1978) 613-641.

[14] Organick, E.I., New directions in computer systems architecture, EUROMICRO journal 5(1979) 190-202.

[15] Friedman, D.P. & Wise, D.S., An indeterminate constructor for applicative programming, Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Jan. 1980, 245-250.

COMPUTERORGANISATIE VOOR STRINGMANIPULATIE

J. HEERING
Mathematisch Centrum

1. INLEIDING

1.1. Algemeen

Volgens een recente Engelse studie [1] zal de voortschrijdende ontwikkeling van de microelectronica de komende 10 tot 20 jaar zijn grootste invloed doen gelden op het terrein van de kantoor-automatisering. Het kantoor zal, in de vorm waarin het nu bestaat, verdwijnen en plaats maken voor het "electronische kantoor". De schrijfmachines worden vervangen door interactieve tekstverwerkende systemen (word-processors), elk met zijn eigen ingebouwde computer en massageheugen, de archiefkasten en kaartsystemen worden vervangen door bestandenbeheer-systemen en dit alles wordt onderling en met de buitenwereld verbonden via een datacommunicatie-netwerk. Daarmee is tevens het kader geschetst voor de ontwikkeling en de belangrijkste toepassing van stringmanipulatie, want tekstverwerking en stringmanipulatie zijn zo nauw met elkaar verweven, dat de lezer ze zonder bezwaar als synoniem mag beschouwen. Het corrigeren van tekst (editing), een bezigheid waarmee elke computergebruiker vertrouwd is, kan gezien worden als de oervorm van alle stringmanipulatie.

Een string is een geordende rij symbolen (letters, cijfers, etc.) en de belangrijkste operatie op strings is concatenatie. Dit is een dyadische operatie, die als resultaat zijn linker operand gevolgd door zijn rechter operand aflevert. Bijvoorbeeld

conc("AAP","NOOT") = "AAPNOOT".

(Stringwaarden worden steeds tussen aanhalingstekens geschreven om ze van de rest van de tekst te onderscheiden.) Daarnaast spelen operaties voor patroonherkenning een belangrijke rol. Deze bieden de mogelijkheid in een gegeven string een gedeelte (substring) te localiseren, dat aan bepaalde eisen voldoet. Dergelijke operaties zijn nuttig bij het opzoeken van een woord in een tekst of bij de syntactische analyse van

een programma. Enkele voorbeelden van patroonherkennings-operaties zullen in de volgende paragraaf aan de orde komen.

Tot nu toe zijn de computerfabrikanten er met redelijk succes in geslaagd verborgen te houden, dat er willekeurig grote getallen bestaan, maar computers, waarin strings niet langer kunnen worden dan 8 symbolen (en met als optie voor de veeleisende gebruiker dubbele-precisie-strings van 16 symbolen), zijn niet acceptabel. Het toelaten van strings van willekeurige lengte is echter op zich nog niet voldoende, want, alhoewel het in eenvoudige gevallen mogelijk is de lengte van de in een berekening optredende strings van te voren te bepalen en de benodigde ruimte statisch toe te wijzen, is in het algemene geval dynamische geheugentoewijzing vereist. Dat wil zeggen, dat de voor een string benodigde ruimte moet worden toegewezen op het moment dat de string wordt geproduceerd. Het toewijzingsproces kan gezien worden als onderdeel van de operatie, die de string produceert, en er hoeven op machinetaalniveau geen expliciete instructies voor te zijn. Anders gezegd: de dynamische geheugentoewijzing kan zich onder het machinetaalniveau bevinden.

Dynamische geheugentoewijzing en geheugenorganisatie zullen in het volgende de centrale onderwerpen zijn. Naast de geheugenorganisatie van de Fairchild SYMBOL computer in hoofstuk 2, komt in hoofdstuk 3 een onconventionele geheugenorganisatie ter sprake, waarin gebruik gemaakt wordt van associatief geheugen en een parallelle geheugenopruimer (garbage collector). Vervolgens worden in hoofstuk 4 de implicaties van grote geheugens voor systeemorganisatie besproken.

1.2. Stringmanipulatie-faciliteiten op de DEC VAX-11

Alvorens het SYMBOL systeem, dat speciaal voor stringmanipulatie ontworpen werd, te bespreken is het misschien nuttig bij wijze van contrast eerst eens te kijken wat voor stringmanipulatie-faciliteiten geboden worden door een representatieve bestaande machine, de DEC VAX-11. Dit is een 32-bits computer met een instructieset van een relatief hoog niveau [2]. De VAX heeft geen zelf-identificerende data (evenmin als de meeste andere machines) en kent eigenlijk geen strings. Het type van een object ligt niet vast, maar wordt op elk moment geheel bepaald door de operatie, die op het object wordt uitgevoerd. Er is geen type-informatie geassocieerd met het object zelf. Ook heeft de VAX geen ingebouwd mechanisme voor dynamische geheugentoewijzing (en verschilt daarin weer niet van andere commercieel verkrijgbare machines). Als gevolg van deze twee beperkingen, die kenmerkend zijn voor de von Neumann architectuur, moeten de adressen en lengtes als parameters worden meegegeven aan de VAX string-instructies. Zo heeft een typische versie van de "copieer string" instructie (MOVC) 3 parameters, namelijk het adres en de lengte van de te

copiëren string en het adres van de bestemming. De VAX heeft geen operatie voor stringconcatenatie. Deze moet worden gerealiseerd door middel van MOVC instructies. Hij heeft echter wel een aantal interessante operaties voor patroonherkenning, namelijk MATCHC, SPANC en SCANC.

MATCHC zoekt in zijn eerste argument (de subjectstring) "van links naar rechts" naar een substring, die gelijk is aan zijn tweede argument. (Beide argumenten worden in overeenstemming met het voorgaande gekarakteriseerd door expliciet gegeven adres en lengte.) Als de operatie slaagt levert MATCHC het adres van de gevonden substring af.

SPANC bepaalt in een gegeven subjectstring "van links naar rechts" de langste string, die geheel bestaat uit symbolen uit een gegeven verzameling. Het subject wordt weer door adres en lengte gekarakteriseerd, terwijl de verzameling toegestane symbolen in de vorm van een karakteristieke vector moet worden meegegeven. De vector, die aan de SPANC operatie wordt meegegeven, dient een lengte van 256 te hebben, want een symbool op de VAX is 8 bits lang. Elk element van de vector is 1 of 0 al naar gelang de bijbehorende index een toegestaan symbool is of niet. SPANC gebruikt de opeenvolgende symbolen van het subject als index in de vector en stopt op het moment, dat het corresponderende vectorelement 0 is. (Tijdens deze operatie wordt, zonder dat er veel ophef over gemaakt wordt, een symbool als geheel getal gebruikt). Het is duidelijk, dat de vector-representatie van de verzameling van toegestane symbolen een efficiënte zoekactie mogelijk maakt, maar het aanmaken van de vector is tamelijk veel werk. Dit is vooral een bezwaar als het dynamisch moet gebeuren en als de resulterende vector slechts door weinig SPANC instructies als argument wordt gebruikt. In de praktijk is dit veelal niet zo en wordt een gegeven vector zeer vaak benut (bijvoorbeeld in de lexicale scan van een vertaler om een vast aantal lexicale typen te onderscheiden). In dat geval weegt de tijdwinst, die ten gevolge van de representatie in de vorm van een vector wordt behaald ruimschoots op tegen de eraan verbonden hoge aanmaakkosten.

SCANC is het complement van SPANC. Hij bepaalt "van links naar rechts" de langste string, die geheel bestaat uit symbolen die niet tot een gegeven verzameling behoren.

Wat betreft string-instructies komt de VAX er in vergelijking met andere machines niet slecht af, maar er valt desondanks nog veel te verbeteren.

2. DE FAIRCHILD SYMBOL COMPUTER

2.1. Inleiding

De SYMBOL computer [3-11] is in de periode 1964-1970 ontwikkeld door Fairchild Instrument and Camera Co. De ontwerpers hadden zich tot doel gesteld een systeem te produceren, dat speciaal geschikt zou zijn voor niet-numerieke berekeningen (lees: stringmanipulatie) en waarin het grootste deel van de systeemtaken zoals editing, vertaling en binding van programma's, multiplexing van processen (time-sharing) en dynamische geheugentoewijzing, verricht zou worden door speciaal daartoe ontwikkelde hardware subsystemen. Het resultaat, dat begin 1971 voor evaluatie en verdere ontwikkeling aan Iowa State University werd overgedragen, was een heterogeen multiprocessor-systeem, dat in vele opzichten uniek was en is (ook in de zin dat er slechts één van is gemaakt) en dat, qua concept, na meer dan 10 jaar nog steeds interessant is. Alle subsystemen van SYMBOL, waaronder de vertaler, zijn geheel in hardware uitgevoerd zonder dat van microprogrammering gebruik is gemaakt.

Achtereenvolgens zullen nu de SYMBOL brontaal en het geheugenbeheer van het SYMBOL systeem besproken worden. Zie figuur 1 voor een blokdiagram van het systeem.

2.2. De SYMBOL brontaal

Het SYMBOL systeem is toegespitst op de executie van programma's in een speciaal ontworpen hogere taal. Het is een zg. indirect uivoerende processor [12], dat wil zeggen, dat hij een hoog niveau machinetaal heeft, die alleen wat betreft representatie verschillen vertoont met de brontaal, maar niet wat betreft de semantiek van zijn operaties. De SYMBOL brontaal lijkt qua expressie- en controle-structuur op ALGOL60, zij het dat er geen FOR-statement is, en dat recursieve procedure/functie-aanroepen niet zijn toegestaan. Procedures en functies gebruiken het call-by-name parameter-mechanisme.

Variabelen in SYMBOL zijn niet statisch getypeerd en er zijn geen declaraties, dat wil zeggen variabelen kunnen tijdens uitvoering van een programma achtereenvolgens waarden van verschillend type hebben. Het atomaire datatype is "string". De lengte van strings ligt niet statisch vast en wordt alleen begrensd door de hoeveelheid beschikbaar geheugen. Er zijn de volgende operaties op strings gedefinieerd:

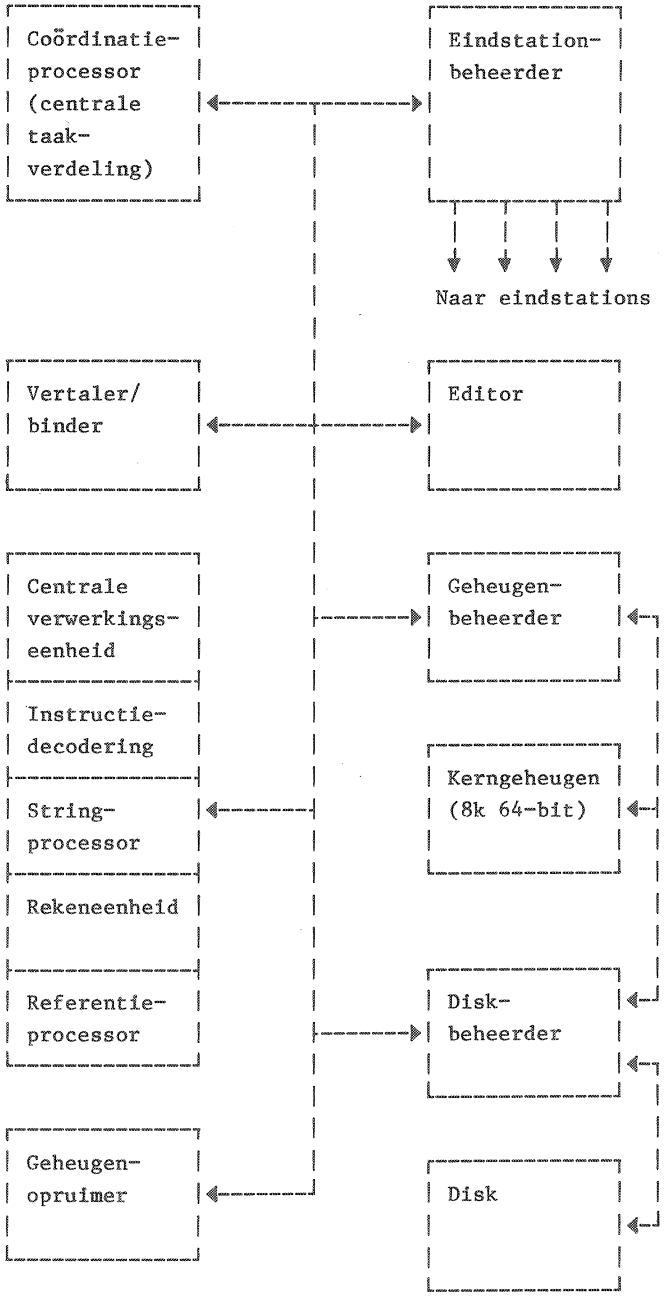


Fig. 1 - SYMBOL blokdiagram

JOIN - concatenatie
BEFORE, SAME, AFTER - strings zijn lexicografisch geordend - deze predicaten testen de volgorde van hun operanden

Er zijn geen operaties voor patroonherkenning. Naast string-instructies zijn er aritmetische operaties beschikbaar op strings, die voldoen aan de syntax voor gehele of floating point getallen, en logische operaties op strings, die slechts de symbolen 0 en 1 bevatten. In tegenstelling tot andere strings hebben getallen een maximale lengte, namelijk een mantisse van maximaal 99 cijfers, eventueel gevolgd door een exponent tussen -100 en +100. Om te voorkomen dat berekeningen met onnodig grote precisie (en dus onnodig traag) worden uitgevoerd heeft de programmeur de beschikking over de pseudo-variabele LIMIT, waarmee hij een bovengrens kan stellen aan het aantal significante cijfers van de in een berekening voorkomende getallen.

Naast strings zijn er in SYMBOL gestructureerde waarden, namelijk lijsten. Een element van een lijst is of een string of zelf weer een lijst. Bijvoorbeeld (in enigszins aangepaste notatie)

```
<"AAP","NOOT","MIES">,
<"AAP",<"NOOT","MIES">>,
<<"AAP","NOOT">,<"WIM","ZUS">>.
```

Het aantal elementen van een lijst kan dynamisch variëren en is niet aan een statische bovengrens gebonden. Componenten van een lijst kunnen door middel van indicering benoemd worden. Bijvoorbeeld na uitvoering van de toekenning

```
LEESPLANK := <"AAP","NOOT",<"MIES",<"WIM","ZUS">>>;
```

heeft LEESPLANK[1] de waarde "AAP", LEESPLANK[3,1] de waarde "MIES" en LEESPLANK[3,2] de waarde <"WIM","ZUS">. Bij toekenning van een waarde aan een niet bestaand lijstelement, wordt dit element automatisch gecreëerd, evenals alle tussenliggende elementen. Deze laatste krijgen de waarde 0. Bijvoorbeeld na de toekenningen

```
LEESPLANK := <"AAP">;
LEESPLANK[4,2] := "MIES";
```

heeft LEESPLANK de waarde <"AAP",0,0,<0,"MIES">>. Dezelfde automatische uitbreiding vindt plaats als een niet bestaand element van een lijst

wordt gelezen i.p.v. geschreven. Bijvoorbeeld na de toekenning

```
RIJ := <0,1,2>;
VOLGENDE := RIJ[4];
```

heeft VOLGENDE de waarde 0 en RIJ de waarde <0,1,2,0>. (Het is de vraag of automatische lijstuitbreiding in geval van schrijfoperaties gewenst is (het loopt gemakkelijk uit de hand), maar in geval van leesoperaties is het zeker een ongewenst neveneffect.)

Bij toekenningen wordt een strikte copieerregel toegepast, dat wil zeggen dat het niet mogelijk is meerdere verwijzingen naar een lijst of lijstelement te creëren. Bijvoorbeeld na

```
VERVOLG := <"NOOT","MIES">;
LEESPLANK := <"AAP",VERVOLG>;
VERVOLG[2] := "JET";
```

heeft VERVOLG de waarde <"NOOT","JET"> en LEESPLANK de waarde <"AAP",<"NOOT","MIES">> (niet <"AAP",<"NOOT","JET">>).

Er zijn, behalve toekenning, indicering en in- en uitvoer, geen operaties op lijsten gedefinieerd in SYMBOL. Het is dus niet mogelijk alle elementen van een lijst in één enkele operatie bij elkaar op te tellen of iets dergelijks.

2.3. SYMBOL geheugenbeheer

2.3.1. De verschillende geheugenniveau's van SYMBOL

Geheugenbeheer in de SYMBOL-machine is tamelijk ingewikkeld tengevolge van het sterk dynamische karakter van de objecten, die door het syteem gemanipuleerd worden. Er zijn verschillende geheugenniveau's te onderscheiden (zie figuur 2). Niveau (1) is het geheugen, zoals dit gezien wordt door de SYMBOL programmeur. Het bestaat uit strings en lijsten (zie vorige paragraaf). Het geheugenmodel op machinetaal-niveau (2) is gelijk aan dat op niveau (1), in overeenstemming met het feit, dat SYMBOL een hoog niveau machinetaal heeft.

Niveau (3) is het geheugen, zoals dit gezien wordt door alle processors, die deel uitmaken van het systeem (zie figuur 1), behalve de geheugenbeheerder zelf. Het geheugen op dit niveau bestaat uit een (praktisch) onbeperkt aantal woordstrings (8 symbolen/64 bits per woord), die dynamisch in lengte toe en af kunnen nemen. De afbeelding van (2) naar (3) wordt tot stand gebracht door de referentie-processor, een subsysteem

van de centrale verwerkingseenheid, dat zich voornamelijk met indicerings-operaties bezighoudt.

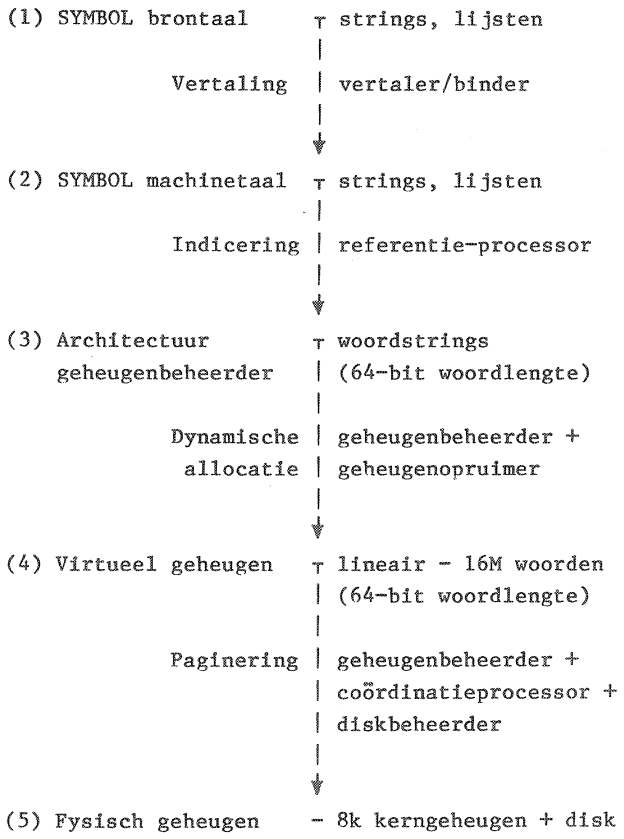


Fig. 2 - SYMBOL geheugenniveau's

De geheugenbeheerder beeldt niveau (3) af op een lineair geheugen (4) door middel van dynamische geheugenallocatie. Deallocatie wordt door een aparte processor, de geheugenopruimer, gedaan. Deze werkt parallel met de geheugenbeheerder. De stap van (4) naar (5) (paginering) wordt eveneens door de geheugenbeheerder gedaan in samenwerking met de coördinatie-processor en de diskbeheerder.

Achtereenvolgens zullen nu de stappen (3)-->(4) en (2)-->(3) in meer detail besproken worden.

2.3.2. De geheugenbeheerder en de geheugenopruimer

De woordstrings van onbeperkte lengte, die de geheugenbeheerder de overige subsystemen biedt, zien er intern uit als dubbel gelinkte lijsten. Elk lijstelement is een groep van 8 opeenvolgende woorden (samen 64 symbolen). De voornaamste instructies van de geheugenbeheerder zijn:

- Wijs groep toe en geef adres van eerste woord terug. (Creëer eerste groep voor nieuwe string.)

- Schrijf/lees woord op/van gegeven adres en geef adres van volgende woord terug. (Hierbij wordt de voorwaartse link gevolgd. Dit kan eventueel impliciet allocatie van een nieuwe groep met zich meebrengen.)

- Lees woord van gegeven adres en geef adres van vorige woord terug. (Hierbij wordt de achterwaartse link gevolgd.)

- Dealloceer string vanaf gegeven adres. (Zie bespreking geheugenopruimer.)

De regel is, dat een adres, dat aan de geheugenbeheerder als onderdeel van een geheugeninstructie wordt aangeboden, bij een eerdere instructie door deze geretourneerd moet zijn. Op deze wijze kan een subsysteem zonder problemen een woordstring in het geheugen opbouwen of lezen. Het feit, dat de string niet op opeenvolgende adressen opgeslagen is, doet verder niet ter zake. De vertaler/binder leest bijvoorbeeld op deze wijze de programmatekst, die als woordstring in het geheugen is gerepresenteerd, en produceert een bijbehorende objectcode-string en naamlijst. Deze wordt vervolgens door de centrale verwerkingseenheid geïnterpreteerd. Elke keer als deze een woord leest uit de objectcode-string, levert de geheugenbeheerder het adres van het volgende woord af. Zolang er geen sprongen voorkomen levert de geheugenbeheerder dus zelf de nieuwe waarde van de programmawijzer. Daarnaast gebruikt de centrale verwerkingseenheid een woordstring als stapel. De "schrijf en volg voorwaartse link" operatie voegt een woord aan de stapel toe en levert de nieuwe stapelwijzer af. Tevens wordt, indien nodig, een nieuwe groep aan de stapel toegewezen. Omgekeerd levert de "lees en volg achterwaartse link" operatie (met de stapelwijzer als adres) het topelement van de stapel af, samen met de nieuwe stapelwijzer. De stapel kan in deze opzet vrijwel onbeperkt dynamisch groeien.

De afbeelding van het string-geheugen op niveau (3) naar het lineaire geheugen op niveau (4) is ingewikkelder dan op het eerste gezicht nodig lijkt, omdat het lineaire geheugen geïmplementeerd is als een

gepagineerd virtueel geheugen en het om redenen van snelheid noodzakelijk is het paginaverkeer tussen het kerngeheugen en de disk tot een minimum te beperken. Om dit te bereiken moeten bij elkaar horende groepen zo veel mogelijk op dezelfde pagina geplaatst worden. De geheugenbeheerder houdt daartoe de volgende gelinkte lijsten bij:

- Globale lijst van vrije pagina's.
- Per pagina een lijst van vrije groepen. (Er zijn 28 groepen per pagina.)
- Per gebruiker een lijst van pagina's, die toegewezen zijn aan zijn bronprogramma.
- Per gebruiker een lijst van pagina's, die toegewezen zijn aan de door zijn programma gegenereerde objecten en de bij zijn programma horende naamlijst en stapel.
- Per gebruiker een lijst van pagina's, die toegewezen zijn aan zijn object-programma.
- In elk van de 3 voorgaande lijsten een deellijst van de pagina's, waarop nog groepen vrij zijn.
- Per gebruiker een lijst van niet meer gebruikte, maar nog niet opgeruimde, groepen.

Bij allocatie van een nieuwe groep wordt eerst geprobeerd deze op dezelfde pagina als zijn voorganger in de keten (als die er is) te plaatsen. Als dit niet lukt wordt geprobeerd hem op een pagina in dezelfde lijst toe te wijzen. (Om de geheugenbeheerder onder alle omstandigheden in staat te stellen te bepalen welk lijst hij hebben moet, worden bij elke geheugeninstructie de identificaties van de gebruiker en de opdrachtgevende processor meegestuurd.) Als ook dit faalt, wordt een vrije pagina onttrokken aan de globale lijst van vrije pagina's en wordt de groep op deze pagina toegewezen.

De copieerregel, die op brontaal niveau geldt (zie vorige paragraaf), wordt ook op het niveau van de geheugenbeheerder gehanteerd. Dit betekent, dat deallocatie in principe eenvoudig is, omdat er nooit meer dan één enkele verwijzing naar een object kan bestaan. Op het moment, dat deze verwijzing wordt vernietigd, kan ook het object zelf opgeruimd worden. Dit gebeurt onder de volgende omstandigheden:

- In geval van toekenning aan een variabele kan de oude waarde van de linker operand opgeruimd worden.

- In geval van procedure- en blokuitgang kunnen de waarden van de lokale variabelen opgeruimd worden.

- Op het moment, dat een programma klaar is, kunnen de naamlijst, de objectcode-string en de stapel vernietigd worden. (Compilatie op de SYMBOL-machine gaat zo snel, dat alleen de brontekst van programma's en procedures bewaard wordt.) Het opruimen van de naamlijst en de stapel leidt automatisch tot deallocatie van de door het programma gegenereerde objecten, omdat deze alle hun "wortel" in één van deze twee objecten hebben.

Het sturen van de vereiste "dealloceer string" instructie naar de geheugenbeheerder is de verantwoordelijkheid van de processor, die de operatie uitvoert, welke er toe leidt dat de string niet meer bereikbaar is. De geheugenbeheerder doet dit dus niet zelf. Het gebruikte systeem van gelinkte groepen en pagina's brengt echter met zich mee, dat het effectief opruimen van een string tamelijk veel werk is. Daarom is deze operatie in tweeën gesplitst. Op het moment, dat de geheugenbeheerder de opdracht krijgt een bepaalde string te dealloceren, wordt deze slechts toegevoegd aan een lijst van nog op te ruimen strings. Zo'n lijst wordt per gebruiker bijgehouden (zie boven). De echte deallocatie wordt vervolgens door de geheugenopruimer gedaan. Deze werkt parallel met de geheugenbeheerder. De geheugenbeheerder beschikt ten behoeve van de opruimer over een aantal operaties, die speciaal dienen om synchronisatie-problemen te voorkomen:

- Lees woord van gegeven adres en schrijf er een 0 in.

- Voeg groep toe aan lijst van vrije groepen.

- Voeg lijst van vrije pagina's toe aan globale lijst van vrije pagina's.

Als hij niets te doen heeft kijkt de opruimer (die de laagste prioriteit heeft op de centrale databus) naar de toppen van de lijsten van de op te ruimen strings. Zodra er iets op te ruimen valt wordt een top ongelijk 0 en neemt de opruimer hem over. Daarbij zet hij de oorspronkelijke locatie weer op 0, zonder dat dit door een "dealloceer string" instructie van een andere processor onderbroken kan worden. Daarvoor gebruikt hij de "lees en schrijf 0" instructie. Vervolgens begint hij de keten te volgen en successievelijk alle groepen weer toe te voegen aan de vrije groep-lijsten van de pagina's, waarop ze zich bevinden.

Daartoe gebruikt de opruimer weer een speciale instructie van de geheugenbeheerder en wel de tweede in bovenstaand lijstje. Deze instructie sluit synchronisatie-problemen met andere processors uit. Zonodig werkt de geheugenbeheerder, als onderdeel van deze instructie, tevens de deellijst van gedeeltelijk vrije pagina's bij. Op het moment, dat een hele lijst van aan een gebruiker toegewezen pagina's ongeldig wordt (bijvoorbeeld als een programma afgelopen is - zie boven), voegt de opruimer deze lijst in zijn geheel toe aan de globale lijst van vrije pagina's.

Sommige objecten, zoals naamlijsten, stapels en lijsten op geheugenniveau (2), kunnen zelf verwijzingen naar andere objecten of subobjecten bevatten. Deze verwijzingen zijn zelf-identificerend en, als de opruimer zo'n verwijzing tegenkomt, zendt hij eenvoudig een "dealloceer string" instructie voor het bijbehorende (sub)object naar de geheugenbeheerder. Vervolgens komt het vanzelf aan de beurt om opgeruimd te worden.

Kan de opruimer het werk aan of worden de lijsten van nog op te ruimen groepen langer en langer? Hier spelen een aantal factoren een rol. Enerzijds heeft de opruimer de laagste prioriteit op de centrale bus (die door alle andere processors ook gebruikt wordt) en dit is natuurlijk niet bevorderlijk voor de snelheid waarmee het opruimwerk gebeurt. Anderzijds is het zo, dat naarmate er meer nog op te ruimen groepen zijn, de gemiddelde hoeveelheid bruikbare informatie per pagina afneemt, zodat het aantal verwijzingen naar niet aanwezige pagina's (pagina-uitzonderingen - pagefaults) toeneemt. Dit is gunstig voor de opruimer, want daardoor neemt het verkeer op de centrale bus af. (De processor, die de pagina-uitzondering heeft veroorzaakt, heeft niets te doen totdat de pagina in het snelle geheugen aanwezig is.) Daar staat weer tegenover, dat de opruimer zelf ook pagina-uizonderingen veroorzaakt. Deze worden op zeer lage prioriteit behandeld. Alles bij elkaar is de situatie niet duidelijk en er zijn geen meetresultaten gepubliceerd.

Een opmerkelijk facet aan het woordstring-geheugen op niveau (3) is, dat het gebruikt wordt door alle processors in het systeem. Enerzijds wijst de geheugenbeheerder ruimte toe voor objecten, die tijdens uitvoering van een SYMBOL programma aangemaakt worden, anderzijds doet hij ook de allocatie voor wat in andere systemen files zouden heten, namelijk de objectcode-string van de vertaler, de brontekst, etc. Deze pseudo-files worden "by reference" van de ene processor naar de andere doorgegeven (via de coördinatie-processor). Op het niveau van de SYMBOL brontaal is het verschil tussen files en interne, door het programma gegenereerde, objecten weer (gewoon) aanwezig. Het is niet mogelijk een programma een permanente string te laten afleveren en deze vervolgens weer aan een volgend programma te binden. Een dergelijke communicatie tussen programma's moet door middel van files en expliciete in- en uitvoer-operaties

plaatsvinden. Dit gaat geheel buiten de geheugenbeheerder om. Hierop zal in hoofdstuk 4 in een algemenere context nog uitvoerig worden ingegaan.

2.3.3. De referentie-processor

De referentie-processor verzorgt de afbeelding van niveau (2) (strings en lijsten) naar niveau (3) (woordstrings). Het moeilijkste hierbij is de uitvoering van indicerings-operaties, omdat er geen simpele indexberekening mogelijk is. De referentie-processor vindt het bij een index horende adres door componenten te tellen. Bijvoorbeeld

```
LEESPLANK := <"AAP","NOOT",<"MIES",<"WIM","ZUS">>>;
FAMILIE   := LEESPLANK[3,2];
```

Voor de toekenning FAMILIE := LEESPLANK[3,2] genereert de vertaler de volgende postfixcode:

```
push    bij FAMILIE horende wijzer naar de naamlijst
push    bij LEESPLANK horende wijzer naar de naamlijst
push    3
push    2
indiceer
ken toe
```

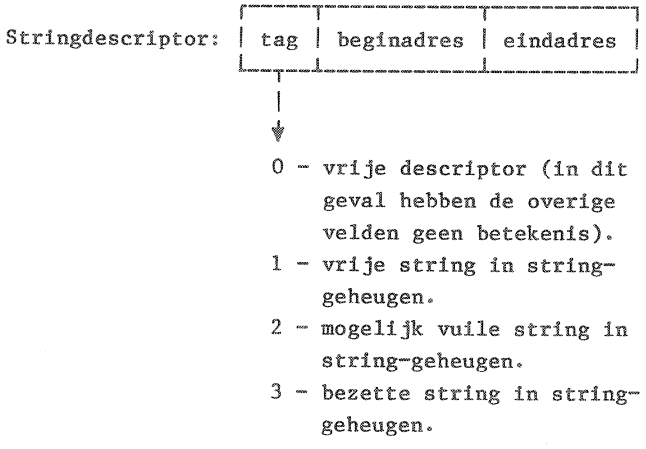
De objectcode verwijst nooit direct naar de waarde van een variabele (het is immers tijdens vertaling niet bekend waar die zich bevindt), maar altijd indirect via de naamlijst. De indiceer-operatie heeft een variabel aantal operanden, maar kan dit aantal bepalen, omdat alle indices van type "getal" zijn, terwijl de variabele van type "wijzer naar naamlijst" is. Verwijzingen en strings zijn eveneens zelf-identificerend. De indiceer-operatie begint met de eerste index (dat is de laagste op de stapel - 3 in dit geval) en zoekt lineair naar het adres van de bijbehorende component. Dit herhaalt hij voor elke index. Om het zoeken te versnellen worden bij een indiceer-operatie de gebruikte index-waarden geregistreerd in de bijbehorende verwijzingen in de lijst. Als bij een volgende indicering index-waarden voorkomen, die groter zijn dan of gelijk aan overeenkomstige index-waarden van de vorige operatie, kan het bijbehorende adres gebruikt worden om het zoekproces te versnellen.

3. EEN ONCONVENTIONELE GEHEUGEN-ORGANISATIE

3.1. Algemene organisatie

In het volgende wordt een experimentele geheugenorganisatie besproken, waarbij de voor het geheugenbeheer noodzakelijke administratie in een apart associatief descriptor-geheugen wordt bijgehouden, terwijl een parallel met de centrale verwerkingseenheid werkende geheugenopruimer tot taak heeft strings waarnaar geen verwijzingen meer bestaan, vrij te maken. Om de taak van de opruimer te verlichten markeert de centrale verwerkingseenheid de strings, die mogelijkerwijs in aanmerking komen om opgeruimd te worden, als zodanig. De opruimer gaat na in hoeverre deze strings inderdaad opgeruimd kunnen worden en dealloceert zoveel van elke string als mogelijk is. Zie figuur 3 voor een blokdiagram van het systeem.

Het string-geheugen bestaat uit een verzameling strings van variabele lengte. Een string is een niet lege rij van opeenvolgende geheugenplaatsen. Met elke string is een descriptor van vaste lengte geassocieerd, die het begin- en eindadres van de string bevat en verder informatie over de status van de string, namelijk vrij, mogelijk vuil of bezet. Bezette en mogelijk vuile strings mogen onderling overlappen, maar strings mogen geen verwijzingen bevatten.



Overlappende strings zijn toegestaan om het aantal copieer-acties te minimaliseren. Als een patroonherkennings-operatie een substring oplevert (zie inleiding), kan dit in deze organisatie gebeuren in de vorm van een

verwijzing naar een substring-descriptor. De prijs, die voor dit schema betaald moet worden, is hoog: er is een geheugenopruimer nodig om strings vrij te maken. Het bijhouden van verwijzingstellers helpt immers niet, omdat het niet duidelijk is, welke teller verhoogd moet worden als er een verwijzing naar een substring wordt gecreëerd.

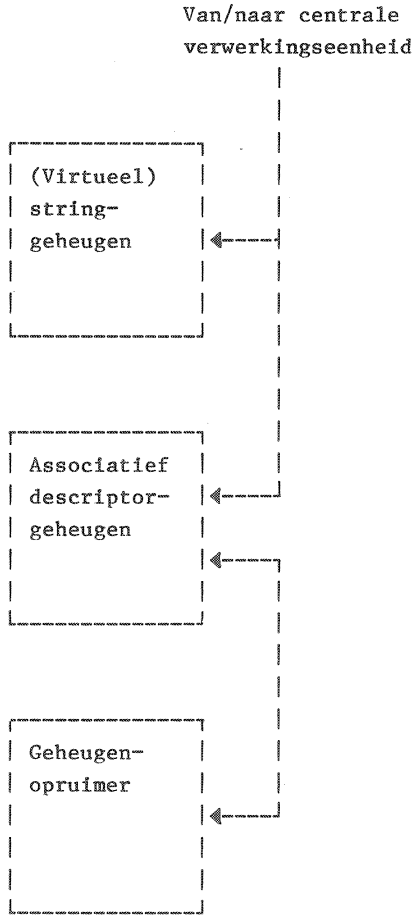


Fig. 3 - Blokdiagram string-geheugen

Het string-geheugen is georganiseerd als gepagineerd virtueel geheugen, net als in de SYMBOL computer. Het is dus belangrijk om pagina-uitzonderingen te minimaliseren. Er zijn een aantal maatregelen genomen om dit te bereiken:

- De string-descriptors bevinden zich in een apart geheugen, zodat toegang tot een descriptor geen pagina-uitzonderingen kan veroorzaken.

- De geheugenopruimer compactificeert niet en heeft geen toegang tot het string-geheugen nodig. Om te voorkomen, dat de strings, die in gebruik zijn, zich teveel over het string-geheugen gaan verspreiden (wat een toename van het aantal pagina-uitzonderingen met zich mee zou brengen), wordt geprobeerd ruimte voor nieuwe strings steeds op een zo laag mogelijk virtueel adres toe te wijzen. Op die manier worden opengevallen gaten weer opgevuld. Er wordt bij toewijzing van ruimte geen rekening gehouden met paginagrenzen. Het pagineringsmechanisme is geheel onzichtbaar buiten het string-geheugen zelf.

- Copieeracties zijn in een aantal gevallen niet nodig, omdat strings geheel of gedeeltelijk mogen overlappen (zie boven).

3.2. Het descriptor-geheugen

Om toewijzing van ruimte te versnellen en de taak van de opruimer te verlichten, zijn de string-descriptors in een geheugen geplaatst, dat zowel gewoon adressseerbaar als associatief is. In een dergelijk geheugen bezit elk woord (dat wil in dit geval zeggen elke descriptor) een zekere rekencapaciteit, zodanig, dat er eenvoudige operaties op alle woorden tegelijkertijd kunnen plaatsvinden. Zie [13,14] voor details.

Het descriptor-geheugen heeft de volgende instructies:

- Lees/schrijf gegeven veld van descriptor op gegeven adres.

- Zoek naar descriptor(s) waarvan de inhoud van een gegeven veld respectievelijk gelijk aan, niet groter dan of niet kleiner dan een gegeven waarde is en retourneer het adres van één van de descriptors, die aan de voorwaarde voldoet. Geef een faalsignaal als geen enkele descriptor voldoet.

- Zoek descriptor(s), die naar een string wijzen, waarvan de lengte groter is dan of gelijk aan een gegeven waarde en retourneer het adres van de descriptor, die behoort bij de string met het kleinste adres. Geef een faalsignaal als geen enkele descriptor voldoet.

- Zet alle antwoordbits (zie vervolg).

Met elke descriptor is een antwoordbit geassocieerd. Alleen descriptors, waarvan het antwoordbit gezet is, worden bij een zoek-operatie be-

trokken. Descriptors, die aan de in de zoek-operatie gespecificeerde voorwaarde voldoen, laten hun antwoordbit op 1 staan, de andere zetten hun antwoordbit op 0. Op deze wijze kan een deelverzameling van descriptors, die aan een aantal voorwaarden voldoen, geïsoleerd worden. Bijvoorbeeld

```
zet alle antwoordbits;
zoek descriptor met tag = bezet;
zoek descriptor met eindadres  >= eind;
zoek descriptor met beginadres <= begin;
```

Deze statements localiseren alle descriptors van strings, die een door begin en eind gespecificeerde interval omvatten en de laatste zoekactie levert het adres van één van deze descriptors af.

De derde instructie in het lijstje met geheugeninstructies op de vorige pagina is de moeilijkste, omdat hij een minimale waarde aflevert. Daarvoor is op beperkte schaal samenwerking tussen de verschillende woorden in het descriptorgeheugen noodzakelijk. Deze instructie wordt door de centrale verwerkingseenheid gebruikt om stringruimte toe te wijzen. Het associatieve geheugen heeft hier slechts één cyclus voor nodig.

3.3. De geheugenopruimer

Het programma, dat door de centrale verwerkingseenheid wordt uitgevoerd, adresseert waarden in het string-geheugen niet direct, maar via de stringdescriptors. Met de variabelen in het programma corresponderen vaste descriptors, die niet van plaats veranderen (afgezien van descriptors voor tussenresultaten en lokale variabelen van blokken en/of procedures). Als een nieuwe waarde aan een variabele wordt toegekend markeert de centrale verwerkingseenheid de oude waarde als mogelijk vuil. Dat wil zeggen, dat hij een nieuwe descriptor voor de oude waarde creëert van type mogelijk vuil. De oude descriptor gaat naar de nieuwe waarde wijzen. Het creëren van een nieuwe descriptor gaat snel: het descriptor-geheugen levert in één cyclus het adres van een vrije descriptor af.

De aldus gemarkeerde string hoeft niet of niet helemaal vrij te zijn, want er kunnen overlappende strings bestaan, die niet vrij zijn. Dit wordt door de opruimer uitgemaakt. Deze voegt ook naburige vrije strings weer tot één grotere samen. Het associatieve geheugen helpt bij deze beide acties door snel overlappende en aangrenzende strings op te zoeken (zie ook het voorbeeld in de vorige paragraaf).

Omdat strings, die eventueel voor deallocatie in aanmerking komen,

als zodanig door de centrale verwerkingseenheid gemarkeerd worden, hoeft de opruimer niet veel werk te doen. Hij heeft geen toegang tot het string-geheugen nodig, veroorzaakt geen pagina-uitzonderingen en kan werken op momenten, dat de centrale verwerkingseenheid data ophaalt uit of wegschrijft naar het string-geheugen. Wat dit betreft is de situatie in deze opzet veel gunstiger dan in de SYMBOL computer.

Synchronisatie tussen de centrale verwerkingseenheid en de opruimer is natuurlijk essentieel. Beide beschikken daartoe over speciale instructies, waarmee zij exclusief toegang tot het descriptor-geheugen kunnen krijgen.

De nadelen van bovenstaand schema zijn, dat associatief geheugen duur is en niet gemakkelijk te "virtualiseren". De geheugenopruimer is daarom niet alleen nodig om string-geheugen vrij te maken, maar ook om descriptors, die corresponderen met tussenresultaten en lokale variabelen, vrij te maken. Verder is het een nadeel, dat strings in aaneengesloten blokken geheugen zijn opgeslagen en geen verwijzingen mogen bevatten. Dit betekent, dat ze niet dynamisch kunnen groeien zonder dat er een tijdrovende copieeractie nodig is.

4. ORGANISATIE VAN SYSTEMEN MET GROTE (VIRTUELE) GEHEUGENS

4.1. Inleiding

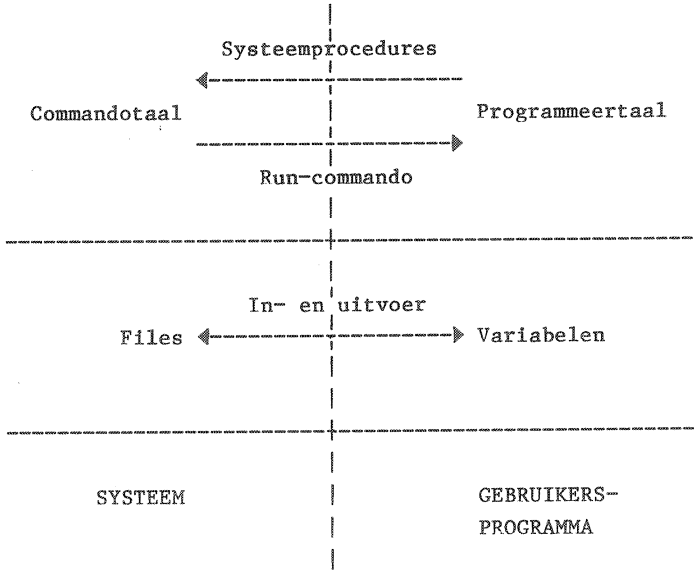
In het voorgaande is herhaaldelijk van virtueel geheugen sprake geweest en de lezer vraagt zich misschien af of, met het goedkoper worden van snel geheugen, niet tevens de noodzaak tot virtueel geheugen verdwijnt. Kan iedereen binnenkort niet zoveel geheugen kopen als hij denkt nodig te hebben en zijn daarmee niet alle problemen, die verband houden met de organisatie van virtueel geheugen, opgelost?

Het antwoord op deze vraag is, dat de geheugenhiërarchie, zoals die in bestaande systemen aanwezig is in de vorm van snel werkgeheugen enerzijds en relatief langzaam achtergrondgeheugen anderzijds, voorlopig om economische redenen niet zal verdwijnen. Er treedt alleen een schaalvergroting op. Werkgeheugens worden groter, maar achtergrondgeheugens ook. Virtuele-geheugentechnieken zijn onlosmakelijk met het bestaan van deze hiërarchie verbonden en blijven dus voorlopig eveneens bestaan. Niettemin is het interessant om na te gaan wat de implicaties zouden zijn voor organisatie en gebruik van computersystemen, indien snel geheugen tegen een zo geringe prijs voorhanden zou zijn, dat een meertraps geheugenorganisatie geen zin meer zou hebben. Immers, ondanks het feit dat dit niet zo is en voorlopig ook nog niet zo zal zijn, is het denkbaar, dat eventuele nieuwe concepten, die uit een dergelijke benadering zouden

voortvloeien, nu reeds implementeerbaar zouden blijken te zijn door van virtuele-geheugentechnieken gebruik te maken.

4.2. Implicaties van grote snelle geheugens voor systeemorganisatie

De gebruiker van een computersysteem wordt in het algemeen geconfronteerd met twee zeer verschillende werelden: de door hemzelf met behulp van een programmeertaal gecreëerde wereld van zijn programma tegenover de wereld van systeemcommando's en files daarbuiten.



Deze werelden staan op gespannen voet met elkaar. De gebruiker, die vanuit zijn programma een systeemcommando wil uitvoeren (bijvoorbeeld om één van zijn files op te ruimen), heeft daarvoor geen andere mogelijkheid dan een systeemprocedure aan te roepen. Of dit mogelijk is of niet, hangt van het gebruikte systeem en de gebruikte programmeertaal af.

Files vormen een hoofdstuk apart. De gebruiker kan er binnen zijn programma alleen via speciale in- en uitvoeracties toegang toe krijgen. Als hij zo naief is om te denken, dat het wel mogelijk zal zijn om een binnen zijn programmma bestaand object een permanente status te geven, zodat hij er later vanuit een ander programma weer aan kan refereren, dan komt hij bedrogen uit. Hij moet het object eerst naar een file schrijven en deze vervolgens in een ander programma weer inlezen. Het eigenaardige

hierbij is, dat het met het object in zijn programma geassocieerde type niet aan de file wordt meegegeven. Een object, dat via een file uit een programma wordt geëxporteerd, is vogelvrij. Het type ervan wordt bepaald door het programma, dat de file leest. In dit opzicht is de situatie een afspiegeling op een hoger niveau van een reeds eerder in de inleiding genoemd facet van de von Neumann architectuur: geheugen-elementen/files hebben geen intrinsiek type, maar hun interpretatie wordt bepaald door de instructie/het programma, die/dat de waarde als operand/data gebruikt. In het algemeen moeten objecten, die uit een programma geëxporteerd worden, eerst een conversie ondergaan. In het bijzonder zijn binnen een file verwijzingen naar bijvoorbeeld het werkgeheugen niet toegestaan en deze mogen dus in de "externe" representatie niet meer voorkomen.

Stel nu, dat een computer zou beschikken over een snel geheugen van voldoende omvang om alle tijdelijke en permanente informatie te bevatten, dan zou het in principe eenvoudig zijn om programma's objecten te laten exporteren in hun interne vorm, zodanig dat het bijbehorende type bewaard blijft. Dit zou op niet veel anders neerkomen dan het creëren van een verwijzing naar het object in een permanente naamlijst en het laten bestaan van het object (dat een zeer complexe structuur kan hebben) na afloop van het programma. Een volgend programma kan dan via de naamlijst weer bij het object komen. Omdat het geheugen zo groot is, is er geen noodzaak om het object "naar een file te schrijven". Dit betekent, dat er geen expliciete in- en uitvoer-acties meer nodig zijn en dat het object niet eerst naar een andere representatie hoeft te worden omgezet. Binnen en buiten het programma kunnen dezelfde objecten bestaan. Zij verschillen slechts wat betreft hun levensduur.

Het is duidelijk, dat dit een aantrekkelijk concept is, geheel onafhankelijk van de vraag of en, zo ja, hoe het geïmplementeerd kan worden.

Zoals eerder in paragraaf 2.3.2 uiteengezet, is het bovenstaande schema reeds in rudimentaire vorm te vinden in het SYMBOL systeem, zij het dat het niet op brontaal-niveau beschikbaar is. Een andere rudimentaire vorm ervan is te vinden in het MULTICS syteem [15], waar expliciete in- en uitvoer van/naar diskfiles niet meer nodig is. Files corresponderen met externe array's. Export van variabelen van willekeurig type is echter niet mogelijk en geëxporteerde array's verliezen hun type.

4.3. Integratie van commando- en programmeertaal

Omdat er binnen en buiten het programma in deze opzet dezelfde datatypes bestaan (namelijk de datatypes, die de programmeur in de oude opzet binnen zijn programma kon manipuleren zonder gebruik te maken van expliciete in- en uitvoer), kan de programmeertaal nu als commandotaal ge-

bruikt worden [12]. Deze combinatietaal dient aan een aantal eisen te voldoen, welke gedeeltelijk voortvloeien uit het feit, dat er geen files meer bestaan, en gedeeltelijk uit de verschillen, die er wat betreft implementatie en gebruik bestaan tussen commando- en programmeertalen:

- De functie van files moet overgenomen kunnen worden door datatypes, die geschikt zijn voor opslag van grote hoeveelheden informatie zonder dat de hoeveelheid van te voren statisch hoeft te worden vastgesteld. Tevens is het een groot voordeel als alle in het systeem bestaande objecten, zoals permanente naamlijsten, procestabellen en dergelijke, types hebben, die in de gebruikerstaal gedefinieerd zijn of kunnen worden. Als dit het geval is kan de taal ook als systeemprogrammeertaal gebruikt worden. Vele bestaande talen, zoals FORTRAN en PASCAL, voldoen niet aan deze eisen.

- Declaratie van (permanente) variabelen op commandoniveau lijkt op het eerste gezicht niet handig, omdat het teveel werk met zich mee brengt. Alhoewel het niettemin mogelijk is toch een type met een variabele te associëren (door het type in de naam te verwerken - vergelijk FORTRAN en sommige filesystemen), is dit weer niet handig op programmeerniveau. Het compromis zou een taal zijn, die geen declaraties heeft en waarin alleen de waarden getypeerd zijn. Programma's in een dergelijke taal hebben het nadeel dat ze moeilijk statisch te analyseren zijn. Dit maakt tevens duidelijk, dat de eisen, die men onder andere wat betreft structurering op programmaniveau stelt, op commandoniveau strijdig zijn (of lijken te zijn) met bedieningsgemak.

- Commandotalen worden meestal incrementeel en zonder voorafgaande vertaling geïnterpreteerd. Dit is op programmaniveau (helaas) niet acceptabel om redenen van efficiëntie. Een alternatief is incrementele compilatie op commandoniveau, maar er blijven twee verschillende taalprocessors nodig, zij het dat zij dezelfde syntactische scan kunnen gebruiken.

4.4. Implementatie

Tot slot kort enkele opmerkingen over implementatie van een dergelijk geïntegreerd taalsysteem:

- Het benodigde grote geheugen kan gerealiseerd worden met behulp van paginering. Geheugentoewijzing en vuilopruiming lijken in eerste instantie centraal te kunnen gebeuren. Hierbij moet echter weer (zie de vorige paragrafen) speciale aandacht geschonken worden aan het minimaliseren van het aantal pagina-uitzonderingen. Ruimte voor

alle objecten moet dynamisch worden toegewezen, zodat elk object later, indien gewenst, een permanente status kan krijgen door het aan een permanente naamlijst te binden. Zie ook [16].

- Bescherming tegen het op hol slaan van wijzers (een bekende kwaal in de von Neumann architectuur) kan verkregen worden door alle objecten zelf-identificerend te maken. "Dangling references" (wijzers, die naar reeds gedealloceerde objecten wijzen) mogen niet voorkomen. Als een object uit een programma geëxporteerd wordt, krijgen alle objecten, waaraan door dat object gerefereerd wordt, eveneens een permanente status. Bescherming op hoger niveau (bijvoorbeeld tegen het modificeren van de objecten van iemand anders) kan verkregen worden door van het SIMULA klasse-concept gebruik te maken, waarbij met toegang tot een object een procedure-aanroep geassocieerd kan worden. Dit geeft tevens het zozeer gewenste krachtige typedefinitie-mechanisme. Klasse-definities zijn, behalve in SIMULA [17], beschikbaar in bijvoorbeeld SMALLTALK [18] en de op het Mathematisch Centrum ontworpen en geïmplementeerde taal SUMMER [19]. Zie ook [20].

- Binding van objecten dient dynamisch te gebeuren, omdat in sommige gevallen de naam van te binden objecten pas tijdens executie van het programma bekend wordt (vergelijk het openen van een file in de oude opzet). Daarnaast is het, als objecten dynamisch gebonden worden, mogelijk modules al gedeeltelijk te testen terwijl de benodigde externe objecten nog niet allemaal bestaan.

5. CONCLUSIE

Stringmanipulatie is vooral een kwestie van geheugenorganisatie. De Fairchild SYMBOL computer heeft twee aparte processors, die zich met geheugenbeheer bezig houden. De overige processors, waaronder de centrale verwerkingseenheid, zien een hoog niveau geheugen, dat uit een onbeperkt aantal strings bestaat, die dynamisch in lengte toe en af kunnen nemen. Deze strings worden op twee manieren benut: voor opslag van tijdens uitvoering van een programma gegenereerde objecten, die het programma niet overleven, en als files voor communicatie tussen de verschillende processors.

Dit schema is voor verregaande generalisatie vatbaar en niet aan de specifieke opzet van het SYMBOL systeem gebonden: objecten binnen en buiten het programma kunnen onder één noemer worden gebracht, mits de gebruikte programmeertaal een voldoend krachtig en flexibel typedefinitiemechanisme bezit en de gebruikte machine een voldoende groot (virtueel) geheugen heeft. Alle objecten in een dergelijk systeem zijn zelf-

identificerend en kunnen met standaard taalconstructies gemanipuleerd worden, zonder dat er expliciete in- en uitvoer-acties aan te pas hoeven te komen (behalve voor in- en uitvoer naar eindstations en andere "harde" apparaten). Daarmee kan tevens het verschil tussen programmeer- en commandotaal vervallen. Eén enkele taal kan beide taken vervullen. Als het geheugen niet al te groot is (bijvoorbeeld enkele tientallen Mbyte) kan geheugenbeheer centraal plaatsvinden. Deze opzet leidt tot een grote vereenvoudiging voor de gebruiker, die niet langer geconfronteerd wordt met twee zeer verschillende talen en geheugenmodellen, maar in plaats daarvan één uniform mechanisme tot zijn beschikking krijgt.

LITERATUUR

[1] Barron, I. & Curnow, R., The Future with Microelectronics, Frances Pinter, 1979.

[2] VAX-11 Architecture Handbook, Digital Equipment Co., 1979.

[3] Rice, R. & Smith, W.R., ´SYMBOL - A major departure from classic software dominated von Neumann computing systems´, AFIPS SJCC, 38 (1971), pp. 575-587.

[4] Smith, W.R., et. al., ´SYMBOL - A large experimental system exploring major hardware replacement of software´, AFIPS SJCC, 38 (1971), pp. 601-616.

[5] Richards Jr., H. & Wright Jr., C., ´Introduction to the SYMBOL 2R programming language´, SIGPLAN Notices, 8 (1973), 11, pp. 27-33.

[6] Anderberg, J.W. & Smith, C.L., ´High-level language translation in SYMBOL 2R´, SIGPLAN Notices, 8 (1973), 11, pp. 11-19.

[7] Laliotis, T.A., ´Implementation aspects of the SYMBOL hardware compiler´, Proc. of the First Annual Symposium on Computer Architecture, 1973, pp. 111-115.

[8] Hutchison, P.C. & Ethington, K., ´Program execution in the SYMBOL 2R computer´, SIGPLAN Notices, 8 (1973), 11, pp. 20-26.

[9] Richards Jr., H. & Zingg, R.J., ´The logical structure of the memory resource in the SYMBOL 2R computer´, SIGPLAN Notices, 8 (1973), 11, pp. 1-10.

[10] Smith, W.R., ´System supervision algorithms for the SYMBOL computer´, IEEE Compcon 72, pp. 21-25.

[11] Zingg, R.J. & Richards Jr., H., ´Operational experience with SYMBOL´, IEEE Compcon 72, pp. 31-32.

[12] Klint, P., ´De symbiose van programmeertaal en computer´, (dit colloquium).

[13] Scheepens, W., ´Toepassingen van associatieve geheugens´, (dit colloquium).

[14] Stork, C., ´Mogelijke realisatie van associatieve geheugens´, (dit colloquium).

[15] Organick, E.I., The MULTICS System, MIT Press, 1972.

[16] Bishop, P.B., Computer Systems with a Very Large Address Space and Garbage Collection, Thesis, MIT Laboratory for Computer Science, MIT/LCS/TR-178, 1977.

[17] Birtwistle, G.M., et. al., SIMULA begin, Auerbach Publications, 1973.

[18] Ingalls, D.H.H., ´The SMALLTALK-76 programming system - design and implementation´, Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM, 1978, pp. 9-16.

[19] Klint, P., ´An overview of the SUMMER programming language´, Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, ACM, 1980, pp. 47-55.

[20] IBM System/38 Technical Developments, IBM General Systems Division, Pub. No. G580-0237, 1978.

TOEPASSINGEN VAN ASSOCIATIEVE GEHEUGENS

W. SCHEEPENS
Technische Hogeschool Eindhoven

Samenvatting.

Na een beknopt overzicht (met literatuur verwijzingen) van de toepassingsgebieden van associatieve geheugens (AM's), worden twee toepassingen, die hun invloed hebben op de systeemarchitectuur en de wijze van programmeren van computers, nader toegelicht. Deze toepassingen betreffen het associatief zijn van een gedeelte van het werkgeheugen of het gehele werkgeheugen. De principes worden geïllustreerd aan de hand van enkele voorbeelden.

1. Inleiding.

Het concept associatief geheugen (AM) bestaat reeds ruim 20 jaar. In deze tijdspanne zijn een groot aantal artikelen verschenen over AM's en mogelijke toepassingen ervan. In de eerste jaren stonden vele implementatieproblemen de toepassingen op grote schaal in de weg. De ontwikkeling van LSI (large scale integration) en VLSI (very large scale integration) bieden echter tegenwoordig een reële basis voor de realisatie van grote AM's, en daardoor ook een reële basis voor de toepassingen van AM's. Hierdoor kunnen AM's zich verheugen in een toenemende belangstelling. De toepassing van AM's in rekenmachines past uitstekend bij de tendens om eenvoud van programmering te verkrijgen door uitbreiding van de hoeveelheid hardware.

Vanwege de inhoud-adresseerbaarheid is "information storage and retrieval" een van de voor de hand liggende toepassingen van AM's. Het is dan ook niet toevallig dat de eerste toepassing van AM's een catalogus-geheugen was [1]. Het probleem bij gebruik van AM's bij information storage and retrieval echter is dat het met de huidige stand van techniek onpraktisch is om een AM te construeren dat groot genoeg is om alle gewenste informatie (database) op te slaan. De toepassing van een klein AM met een groot achtergrondgeheugen, bijvoorbeeld een magnetisch schijvengeheugen, resulteert in een aanzienlijke verspilling van tijd vanwege het laden en legen van het AM. Hierdoor gaat de versnelling vanwege het AM voor een groot deel teniet. Een oplossing hiervoor is gezocht in het verschuiven van de associatieve eigenschappen naar het achtergrondgeheugen [2],[3],[4],[5],[6].

Kleine doch snelle AM's worden toegepast voor interne verwerkingsfuncties in grote conventionele computersystemen vanwege de snelle parallelle verwerkingsmogelijkheden. Een bekend voorbeeld hiervan is het AM als adrestranslatietabel in hiërarchische geheugensystemen, zoals virtuele geheugens [7] en cache geheugens [8]. Andere voorbeelden van interne verwerkingsfunkties zijn interrupt processing [9] en task assignment in multi-processor-systemen.

AM's worden ook toegepast als "funktionele geheugens" [10], [11] om te dienen als microprogrammeerbare universele logische bouwstenen. Een Boolse functie kan geschreven worden als een (logische) som van (logische) produkten van ingangsvariabelen. Elke component van de ingangsvariabele kan de waarde 0,1 of x (don't care) aannemen. Deze funktie kan met een partieel-AM gerealiseerd worden, door de priemtermen in het naamgedeelte en de bijbehorende uitgangswaarden in het inhoudsgedeelte op te slaan. Als bij een gegeven ingangswoord een aantal priemtermen associëren, dan levert het leescircuit de OF-funktie van de geselecteerde uitgangswoorden. Optimalisatie, dat wil zeggen het minimaliseren van het aantal benodigde geheugenbits, kan verkregen worden door handig gebruik te maken van het maskerregister (analoog aan fig 3.7). Optimalisatie kan ook worden

verkregen indien elke geheugencel naast de waarden 0 en 1 ook nog de waarde x (don't care) kan aannemen [11], zodat geen maskerregister vereist is.
Een "read-only" versie van funktionele AM's (zonder ripple-schakeling) wordt gevormd door de "programmable logic arrays", PLA's [12],[13].

Een interessant toepassingsgebied vormen de zogenaamde associatieve rekenmachines: rekenmachines waarvan het gehele of een gedeelte van het werkgeheugen associatief is. Op dit gegeven bestaan twee varianten: een systeem met een associatief datageheugen, maar met een conventioneel direkt adresseerbaar programmageheugen en een systeem met een volledig associatief hoofdgeheugen. De eerste variant, veelal associatieve parallelle processor (APP) genoemd, geeft de mogelijkheid om bewerkingen parallel op een willekeurig aantal geselecteerde woorden te verrichten. Een bekende realisatie is STARAN [14],[15], die in eerste instantie ontworpen is voor "multi-target radar tracking" in luchtverkeerbewaking. De tweede variant, associatieve adresseringsmachine (AAM) genoemd, leidt tot een niet-conventionele methode van "instruction fetching" en datastruktuurbeschrijving. Deze toepassing bevindt zich heden nog in de studiefase [16],[17].
Vanwege de niet-conventionele machine-architectuur en programmeermethode van associatieve rekenmachines, worden de principes van deze systemen nader beschreven in de volgende paragrafen.

2. Associatief datageheugen (APP).

In fig. 2.1 is de architectuur weergegeven van een systeem met een associatief datageheugen. Het datageheugen is uitgevoerd als een volledig-AM met zowel de mogelijkheid de geheugenwoorden parallel als sekwentieel te kunnen beschrijven en lezen.
Terwijl de bewerkingen parallel op de datawoorden worden uitgevoerd is de programma-afwikkeling conventioneel: het systeem bevat een conventioneel direkt adresseerbaar geheugen, waarin zich het programma bevindt, dat bestaat uit instrukties, die het AM besturen en

instrukties die de programmavoortgang bepalen (spronginstrukties e.d.). Het essentiële van deze systeemconfiguratie is dat de variabelen tijdens de bewerkingen in het geheugen blijven staan: de bewerkingen worden in het geheugen uitgevoerd (parallel op alle woorden). In conventionele systemen worden de variabelen één voor één van het geheugen naar een centrale verwerkingseenheid getransporteerd.
Als gevolg van de struktuur van het AM worden de meeste algorithmen woord-parallel, doch bit-serie uitgevoerd. De executietijd van de algorithmen is dan ook een funktie van het aantal bits van de

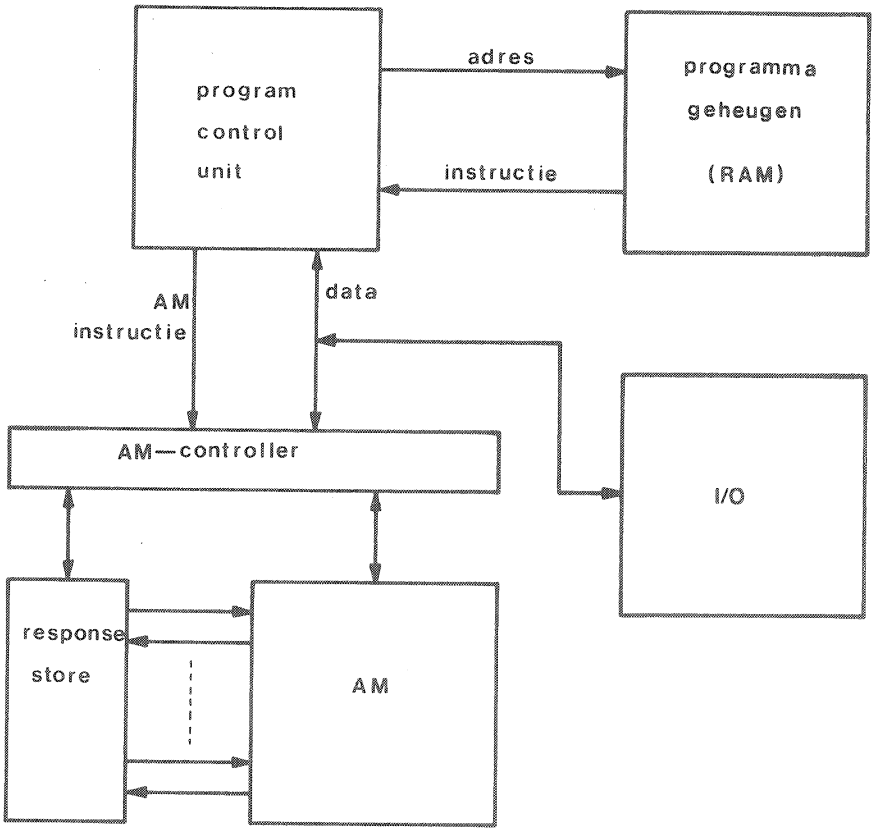


Fig.2.1. architectuur van een machine met een associatief datageheugen

woorden waarop de algorithmen plaatsvinden en niet een funktie van het aantal woorden, zoals in een conventioneel systeem. Hieruit volgt dat de toepassing van APP's voordelig is t.a.v. conventionele systemen in gevallen waar dezelfde algorithmen op meerdere varaiabelen moeten worden uitgevoerd.
De algorithmen voor APP's kunnen verdeeld worden in twee klassen:

- logische algorithmen, die een verzameling variabelen in het AM verdelen in twee of meer subverzamelingen, op grond van hun inhoud.
- arithmetische algorithmen, die bewerkingen uitvoeren op een verzameling variabelen in het AM.

Een andere verdeling van algorithmen is te maken op grond van de behoefte aan terugkoppeling van het AM tijdens de uitvoering van het algorithme. Een voorbeeld van terugkoppeling is het gegeven of er responders zijn op een bepaald zoekargument of niet.
Om de manier te illustreren waarop deze algorithmen werken worden hieronder een tweetal voorbeelden gegeven: een logisch algorithme en een arithmetisch algorithme, beiden zonder terugkoppeling. Bij deze voorbeelden is uitgegaan van een bitparallel-AM zoals behandeld is in [18]. De algorithmen zijn beschreven in een PASCAL-achtige notatie.

Algorithme 1: verdeel de geheugenwoorden in drie klassen: woorden die gelijk zijn, woorden die groter zijn en woorden die kleiner zijn dan een gegeven getal N. Ter vereenvoudiging worden alleen positieve getallen verondersteld, hetgeen niet essentieel is voor het algorithme. De getalwaarde neemt b bits in beslag van elk geheugenwoord W_i (W_i is het i-de geheugenwoord, $W_{i,j}$ is de j-de bit van het i-de woord). Het klassekenmerk wordt met twee bits (x,y) in elk woord aangegeven (zie fig. 2.3). In eerste instantie worden alle woorden op xy=00 gezet, waarmee de onbesliste toestand wordt aangegeven. Vervolgens worden de onbesliste geheugenwoorden bit voor bit met het getal N vergeleken, beginnend bij de meest significante bit (bit b). Bij elke vergelijking zijn drie situaties

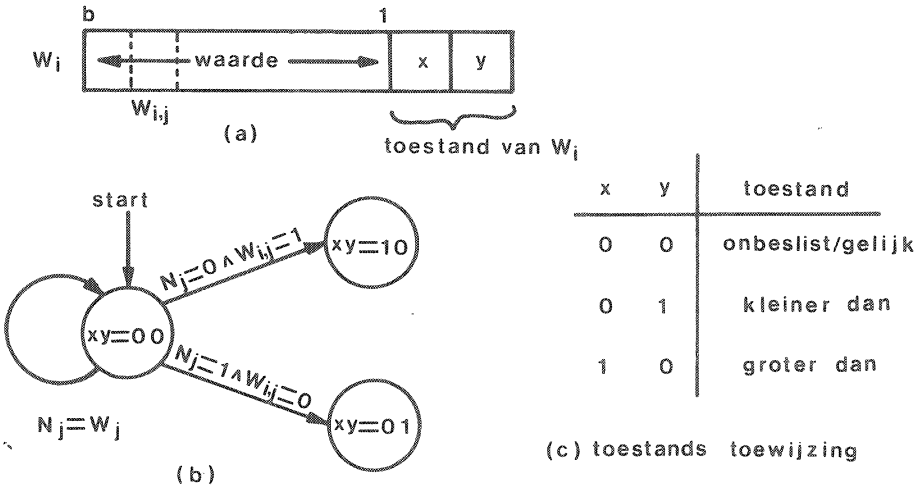


Fig.2.3. bittoewijzing (a) en toestandsdiagram (b) van de geheugenwoorden W_i bij algorithme 1

mogelijk: (1) $N_j = 0$ en $W_{i,j} = 1$, dan is $W_i > N$, zodat xy = 10 (groter dan) wordt gemaakt; (2) $N_j = 1$ en $W_{i,j} = 0$, dan is $W_i < N$, zodat xy = 01 (kleiner dan) wordt gemaakt; (3) $N_j = W_{i,j}$, dan blijft de markering onbeslist (xy = 00). Als alle bits verwerkt zijn, dan zijn de geheugenwoorden met xy = 00 gelijk aan N. Het toestandsverloop van de geheugenwoorden is in fig. 2.3.b in een toestandsdiagram weergegeven.

De toestand van elk geheugenwoord kan bij elke vergelijking van de j-de bit (met j lopend van b naar 1) veranderen. Als verondersteld wordt dat het getal N in register R1 van de program control unit staat, dan kan het algorithme als volgt beschreven worden:

```
set;*) write(xy=00);**)
for j=b downto 1 do begin
    if R1[j]=0
        then begin
            set; compare(W[j]=1,xy=00);***)
            write(xy=10)
            end
        else begin
            set; compare(W[j]=0,xy=00);
            write(xy=01)
            end
    end;
```

*) Met de "set"-instruktie worden alle tag flip-flops op 1 gezet, zodat alle geheugenwoorden worden geselecteerd.

**) Met de notatie "write(xy=00)" wordt aangegeven: comparandreg:=0...000; maskerreg:=1...100; write , zodat van alle geselecteerde woorden de laatste twee, niet gemaskeerde, bits (x en y) beschreven worden met de overeenkomstige bits van het comparand register (00).

***) De notatie: "compare(W[j]=1,xy=00)" betekent: comparandreg:=0..010..000; maskerreg:=1..101..100; compare, hetgeen inhoudt dat de tag flip-flop wordt gereset van geheugenwoorden waarvoor niet geldt dat de j-de bit = 1 én xy = 00 (de andere bits zijn gemaskeerd).

Na afloop van het algorithme kunnen ook de selectiekriteria "kleiner of gelijk" en "groter of gelijk" toegepast worden door:

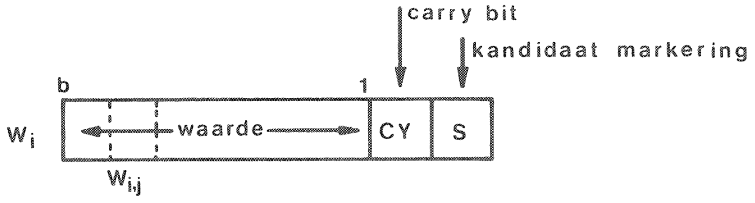
```
    compare(x=0)
```

resp.

```
    compare(y=0)
```

toe te voegen.

Uit de algorithme-beschrijving volgt dat de uitvoeringsduur van het algorithme onafhankelijk is van het aantal geheugenwoorden, doch een funktie is van b: de executie duurt 2+3b AM-geheugentoegangstijden (waarbij voor de set-instruktie ook 1 toegangstijd is genomen).



(a)

voor sommatie		na sommatie		
$W_{i,j}$	CY_i	$W_{i,j}$	CY_i	
0	0	0	0	← geen verandering
0	1	1	0	
1	0	1	0	← geen verandering
1	1	0	1	

(b) bitgewijze sommatie

Fig.2.4. bittoewijzing(a) en toestands tabel(b) van geheugenwoord W_i bij algorithme 2

Algorithme 2: tel 1 op bij een geselecteerd aantal geheugenwoorden: stel dat het sommatie-kandidaatschap van de geheugenwoorden door bit s wordt aangegeven (s=1). Tijdens de uitvoering van het algorithme is er behoefte aan een carry bit cy, zodat elk geheugenwoord W_i behalve de getalwaarde twee bitplaatsen voor s en cy bevat (zie fig.2.4.a). De optelling wordt voor alle woorden met s=1 parallel, bit voor bit uitgevoerd, waarbij cy dient voor de tussentijdse carry-overdracht.

Bij de verwerking van bit j bestaan vier mogelijkheden voor de combinatie $W_{i,j}\,cy_i$ (zie ook fig. 2.4.b): als $cy_i = 0$, dan verandert er niets, maar als $cy_i = 1$ en $W_{i,j} = 0$ dan moet $W_{i,j} = 1$ worden en $cy_i = 0$ en als $cy_i = 1$ en $W_{i,j} = 1$, dan moet $W_{i,j} = 0$ worden en $cy_i = 1$ om te voldoen aan de sommatieregels voor binaire getallen. Elk geheugenwoord W_i werkt dus als een opteller met cy_i als serie-carry. Het algorithme is hieronder weergegeven.

```
set; compare(s=1);
write(cy=1);
for j=1 to b do begin
      set; compare(s=1,cy=1,W[j]=0);
      write(cy=0,W[j]=1);
      set; compare(s=1,cy=1,W[j]=1);
      write(cy=1,W[j]=0)
      end;
```

In de eerste twee regels wordt de carry-bit van alle kandidaatwoorden (met s=1) op 1 gezet. Vervolgens wordt bij alle kandidaatwoorden met cy=1 de carry opgeteld bij bit j en de nieuwe carry geschreven. De volgorde van de twee compare-instrukties met bijbehorende write-instrukties is essentieel, omdat bij verwisseling kandidaten van de dan eerste groep ten onrechte worden meegenomen met de kandidaten van de dan tweede groep.
De uitvoering van dit algorithme neemt 3 + 6b toegangstijden van het AM in beslag.

Bovenstaande algorithmen zijn twee voorbeelden uit een groot aantal algorithmen [20].
Om andere algorithmen te noemen:

(1) logische algorithmen zonder terugkoppeling:
- verdeel een verzameling kandidaatwoorden op grond van gelijkheid/ongelijkheid met een gegeven waarde,
- het wel of niet tussen opgegeven grenzen liggen;

(2) logische algorithmen met terugkoppeling: allerlei variaties op het vinden van het maximum of minimum van een verzameling kandidaatwoorden, zoals:
- vindt het maximum,
- vindt de eerstvolgende kleiner of groter dan de gegeven waarde;

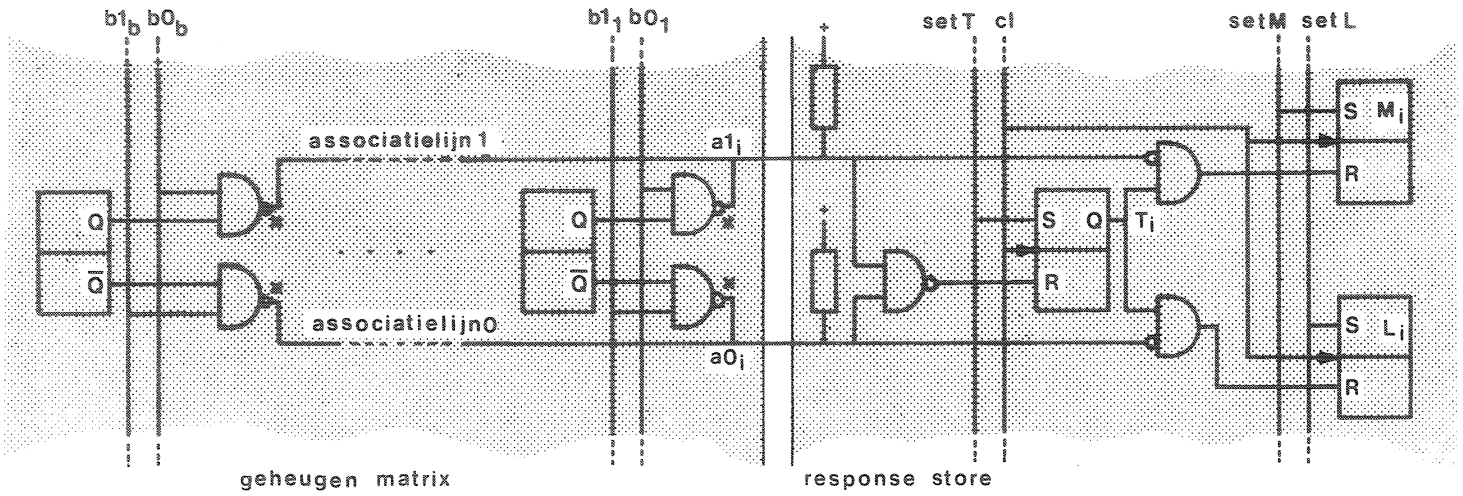
(3) arithmetische algorithmen zonder terugkoppeling:
-inverteren van kandidaatwoorden,
-optellen (aftrekken) met een gegeven constante,
-optellen (aftrekken) van velden van de geheugenwoorden,
-vermenigvuldigen (delen) met een constante,
-vermenigvuldiging (deling) van velden;

(4) arithmetische algorithmen met terugkoppeling:
-berekening van gemiddelde of mediaan van een verzameling kandidaatwoorden of
-het bepalen van het woord van een verzameling woorden met de kleinste Hamming-afstand tot het gegeven woord.

Al deze algorithmen worden uitgevoerd met behulp van de volgende basisbewerkingen:

- vergelijk de geheugenwoorden met de inhoud van het zoekregister, waarbij gelijkheid in het response-store wordt gemarkeerd. De associërende woorden noemt men responders.
- beschrijf de gemarkeerde geheugenwoorden met de inhoud van het zoekregister.
- voor de algorithmen met terugkoppeling: geef aan of er wel of geen responders zijn.
- het lezen van geselecteerde geheugenwoorden om de resultaten van de algorithmen te verkrijgen.

Bij de meeste algorithmen is het mogelijk om de executietijd aanzienlijk te verminderen door toevoeging van extra hardware aan het response store. Om dit te illustreren wordt een mogelijke versnelling van algorithme 1 nader beschreven.
De benodigde hardware-uitbreiding is weergeven in fig. 2.5.a. (alleen de associatie-hardware is getekend).



ⓐ

m_j	c_j	$b1_j$	$b0_j$	$W_{i,j}$	$\geq$ $a1_i$	$\leq$ $a0_i$	$=$ $a1_i \cdot a0_i$
0	0	0	1	0	1	1	1
0	0	0	1	1	1	0	0
0	1	1	0	0	0	1	0
0	1	1	0	1	1	1	1
1	x	0	0	x	1	1	1

ⓑ

Waarheidstabel associatielijnen van bit j (andere bits gemaskeerd)
x=don't care (0 of 1)

Fig.2.5. speciale hardware om algorithme1 te versnellen

In deze schakeling is de associatielijn gesplitst in twee lijnen $a1_i$ en $a2_i$. De NAND-poort en de T_i flip-flop vormen de originele schakeling van [18]. De flipflops M_i en L_i dienen om aan te geven dat W_i groter of kleiner is dan de gegeven waarde N in het zoekregister C. Bij aanvang van het algorithme worden alle flipflops T_i, M_i en L_i op 1 gezet door een "set(T,M,L)"-instruktie. Vervolgens worden alle bits behalve bit j (met j lopend van b tot en met 1) gemaskeerd. Voor dat geval geldt de waarheidstabel van fig. 2.5.b; $a1_i$ geeft aan dat $W_{i,j} \geq C_j$; $a2_i$ geeft aan dat $W_{i,j} \leq C_j$ en $a1_i . a2_i$ geeft aan dat dat $W_{i,j} = C_j$.
Bij selectie van bit j wordt $T_i = 0$ bij alle geheugenwoorden waarvoor $W_{i,j} \neq C_j$. Door de vertraging van de T_i-flipflop wordt ook $M_i = 0$ of $L_i = 0$ afhankelijk van het "type" van de ongelijkheid: $C_j = 1$ and $W_{i,j} = 0$ ==> $M_i = 0$; $C_j = 0$ and $W_{i,j} = 1$ ==> $L_i = 0$. Deze verandering van M_i of L_i kan alleen plaatsvinden als T_i nog 1 was vóór de vergelijking. Nadat alle bits vergeleken zijn geldt voor woorden gelijk aan C dat $L_i T_i M_i = 010$, voor woorden groter dan C dat $L_i T_i M_i = 001$ en voor woorden kleiner dan C dat $L_i T_i M_i = 100$.
Selectie van geheugenwoorden kan plaatsvinden op een logische combinatie van T, L en M. De executietijd van de drieverdeling met behulp van de speciale hardware neemt 1+b toegangstijden van het AM in beslag, een versnelling van ca. 3 op algorithme 1!

In de praktijk zal een compromis gevonden moeten worden tussen een zo kort mogelijke executietijd en minimale kosten voor de extra hardware. Bij een APP als STARAN [14],[15] is nogal wat extra hardware aangebracht.

3. Associatief werkgeheugen (AAM).

Gegeven een associatief datageheugen, ligt het voor de hand, dat men zich af gaat vragen wat het effect zal zijn indien ook het programma in een AM is opgeslagen. In deze paragraaf worden enkele principes besproken van het instruktie ophaalmechanisme uit een AM en de invloed van het AM op de programmeertechniek.

Bij dit type machine wordt het conventionele Von Neumann concept gehandhaafd: de instrukties worden één voor één uit het werkgeheugen opgehaald en uitgevoerd met behulp van een conventionele arithmetische- en logische eenheid (ALU), een registerfile, I/O processoren, enz. De bij de instrukties behorende operanden worden uit hetzelfde werkgeheugen gelezen. De associatieve eigenschappen van het werkgeheugen worden gebruikt voor de instruktie-oproep en de variabele-adressering, zodat volstaan kan worden met een partieel-AM [18]: de instrukties en variabelen worden in het inhoudsgedeelte opgeslagen, terwijl het naamgedeelte fungeert als "adres" van het inhoudsgedeelte. De variabelen worden direkt door hun naam geadresseerd. Door de struktuur in de naam van de variabele aan te geven is het adresseren van een gestruktureerde variabele even simpel als van een ongestruktureerde variabele [16]. Dit type machine wordt associatieve adresseringsmachine (AAM) genoemd.

Omdat alleen het instruktie-ophaal-mechanisme aan de orde is, wordt bij het verdere betoog ter vereenvoudiging een apart datageheugen verondersteld.
Een keten van opeenvolgende instrukties, waarin geen sprongingang voorkomt, in het bijzonder een lineaire programmatak, kan in de verwerkingsvolgorde onder één naam worden opgeslagen. Bij associatie op deze naam worden alle instrukties van de keten in het response store gemarkeerd. Aangezien het AM voorzien is van een ripple-schakeling [18], kunnen de gemarkeerde instrukties één voor één in de juiste volgorde (de opgeslagen volgorde) uit het geheugen worden gehaald. Dit mechanisme maakt dus een programmateller met zijn increment-mechanisme overbodig. De programma-toestand wordt weergegeven door het instruktiezoekregister, bestaande uit een comparand- en maskergedeelte. Tijdens het aflopen van de instruktieketen kan de naam van de volgende keten, de volgnaam, samengesteld worden uit de toestand van een aantal conditiebronnen, zoals bijvoorbeeld de overflow-conditie van de ALU of de informatie of er woorden in het response-store gemarkeerd zijn. Na afloop van de keten moet een associatie op de volgnaam worden uitgevoerd om de volgende keten te markeren. Voor het samenstellen van de volg-

naam zijn conditietoewijzingsinstrukties nodig, die bepalen welke conditiebronnen welke zoekregisterbits met welke toewijzingsmodus beschrijven [17]. In het resterende deel van deze paragraaf zal worden getoond dat elke programmastruktuur uit te voeren is met een associatief programmageheugen en wordt het principe aangegeven van een systematische en dus automatiseerbare programmeermethode.

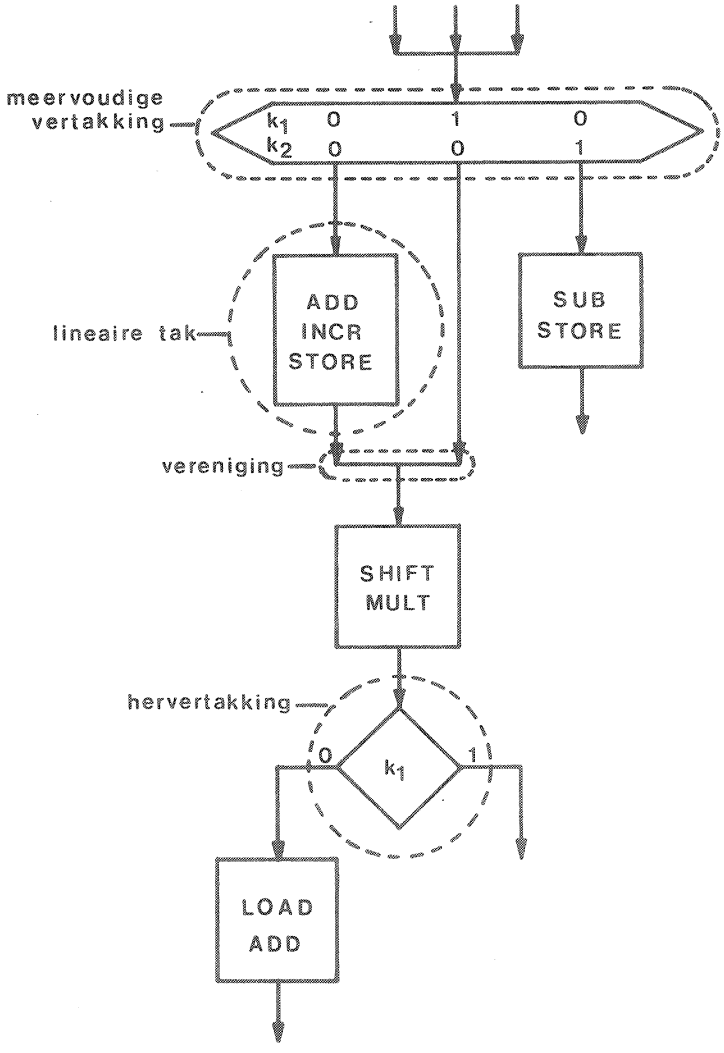


Fig.3.1. programmastruktuur-elementen

Elke willekeurige programmastruktuur kan worden opgebouwd met de volgende drie basiselementen.

- Een lineaire tak, een keten van opeenvolgende instrukties zonder inwendige vertakkingen of verenigingen.
- Meervoudige vertakking, een conditie-gestuurde programmavertakking in een aantal lineaire takken.
- Vereniging, het samenkomen van een aantal lineaire takken.

Deze drie elementen zijn weergegeven in de programmastruktuur van fig. 3.1.
De struktuur van het AM, dat gebruikt wordt om de programma-elementen te implementeren is weergegeven in fig. 3.2. Het response-store

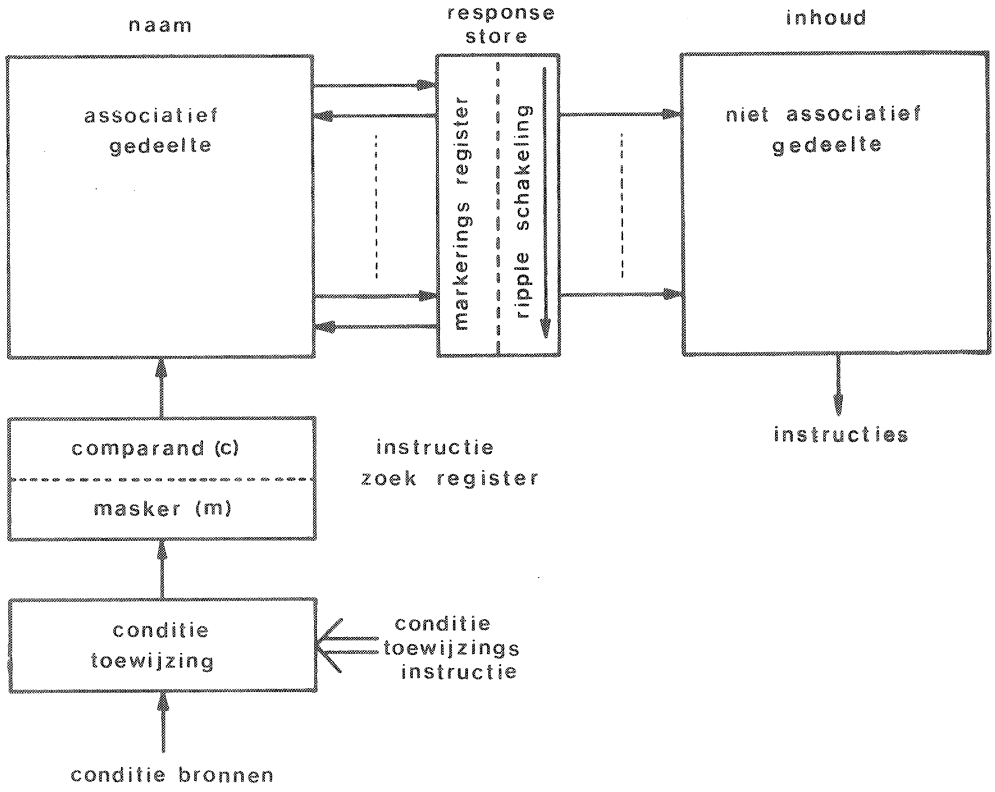


Fig.3.2. structuur van een associatief werkgeheugen met betrekking tot het ophalen van instructies

bevat tenminste één rij tag-flipflops (markeringsregister) om een keten van instrukties te kunnen markeren. Met de ripple-schakeling worden gemarkeerde instrukties van boven naar beneden geselecteerd. De naam van de te selecteren instruktieketen wordt samengesteld in het instruktiezoekregister. De samenstelling van deze naam geschiedt onder besturing van een conditietoewijzingsinstruktie uit de toestand van de conditiebronnen.
De conditietoewijzingsinstruktie bepaalt de conditiebron, de bits van het zoekregister die beschreven worden en de toewijzingsmodus. De modus geeft de manier aan waarop de logische conditiebronwaarde, die true (1), false (0) of don't care (x) kan zijn, wordt afgebeeld op de comparand en het masker van het zoekregister.

Implementatie van de lineaire tak.
In fig. 3.3.a is het stroomdiagram van een lineaire tak weergegeven.

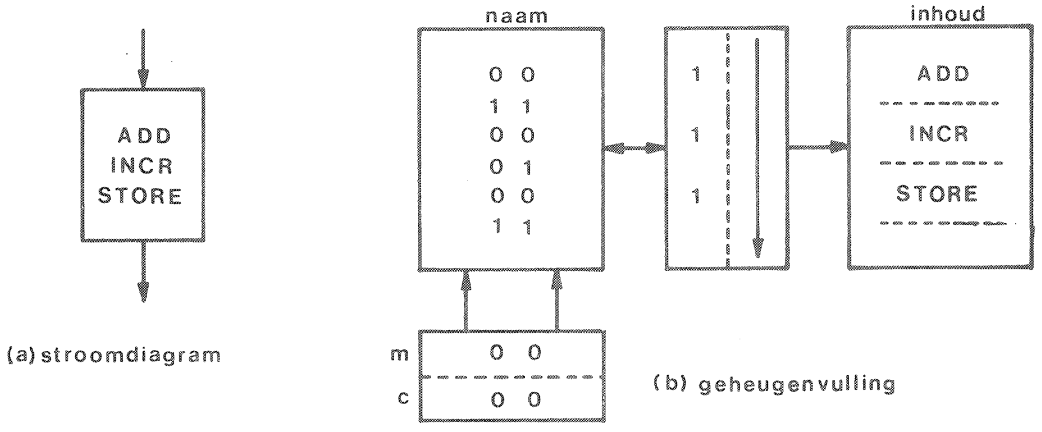


Fig.3.3. implementatie lineaire tak

In fig. 3.3.b is de geheugenvulling weergegeven. De instrukties van de lineaire tak zijn onder dezelfde naam (00) opgeslagen. De plaats van de instrukties is niet belangrijk, maar wel de volgorde van

opslag. In fig. 3.3.b wordt de naam 00 aangeboden (niet gemaskeerd) en is de markering van de instrukties van de associërende lineaire tak aangegeven. Door de ripple-schakeling worden de instrukties van boven naar beneden gelezen.

Implementatie van de vertakking.

Het stroomdiagram van een vertakking is weergegeven in fig. 3.4.a. De struktuur van het stroomdiagram kan eenvoudig worden weergegeven met behulp van een decision-table (fig. 3.4.b) [19]. De tabel geeft weer welke instrukties onder gegeven condities moeten worden uitgevoerd. De instrukties van een lineaire tak worden gezamenlijk een aktie genoemd. De namen van de lineaire takken na de vertakking worden gelijk genomen aan de overeenkomende bitcodering van conditiebronnen: zo heeft b.v. de lineaire tak A1 de naam 00 gekregen. De volgorde van opslag van de akties is niet belangrijk. De geheugenvulling voor de vertakking is weergegeven in fig. 3.4.c. Als voorbeeld is de conditie 00 aangeboden. Hiermee associëren de instrukties van tak A1, die daardoor in het response store gemarkeerd worden. Omdat de namen van de takken gelijk zijn aan de conditiebitcombinaties is de conditietoewijzing simpel: geen masker en $c_1c_2 = k_1k_2$.

Implementatie van de vereniging.

Een stroomdiagram met een vereniging is weergegeven in fig. 3.5.a. Het stroomdiagram bevat tevens een hervertakking op grond van dezelfde conditie als de eerste vertakking om te demonstreren dat een hervertakking geen extra instrukties nodig heeft.

Fig. 3.5.b geeft de bijbehorende decision-table weer. Het probleem is nu dat de tak A3 zowel doorlopen wordt bij conditie 00 als 10. De naam van A3 en de conditietoewijzing wordt zodanig gekozen, dat A3 mee-associeert met zowel de naam 00 als 10. Voor het gegeven stroomdiagram kan dit door A3 de naam 10 te geven en de conditietoewijzing zodanig te maken dat bij $k_1 = 0$ de eerste bit gemaskeerd ($m_1 = 1$) wordt. Zie hiervoor fig. 3.5.c. Als voorbeeld is de conditie 00 aangeboden. Door de toewijzingsinstruktie resulteert dit in

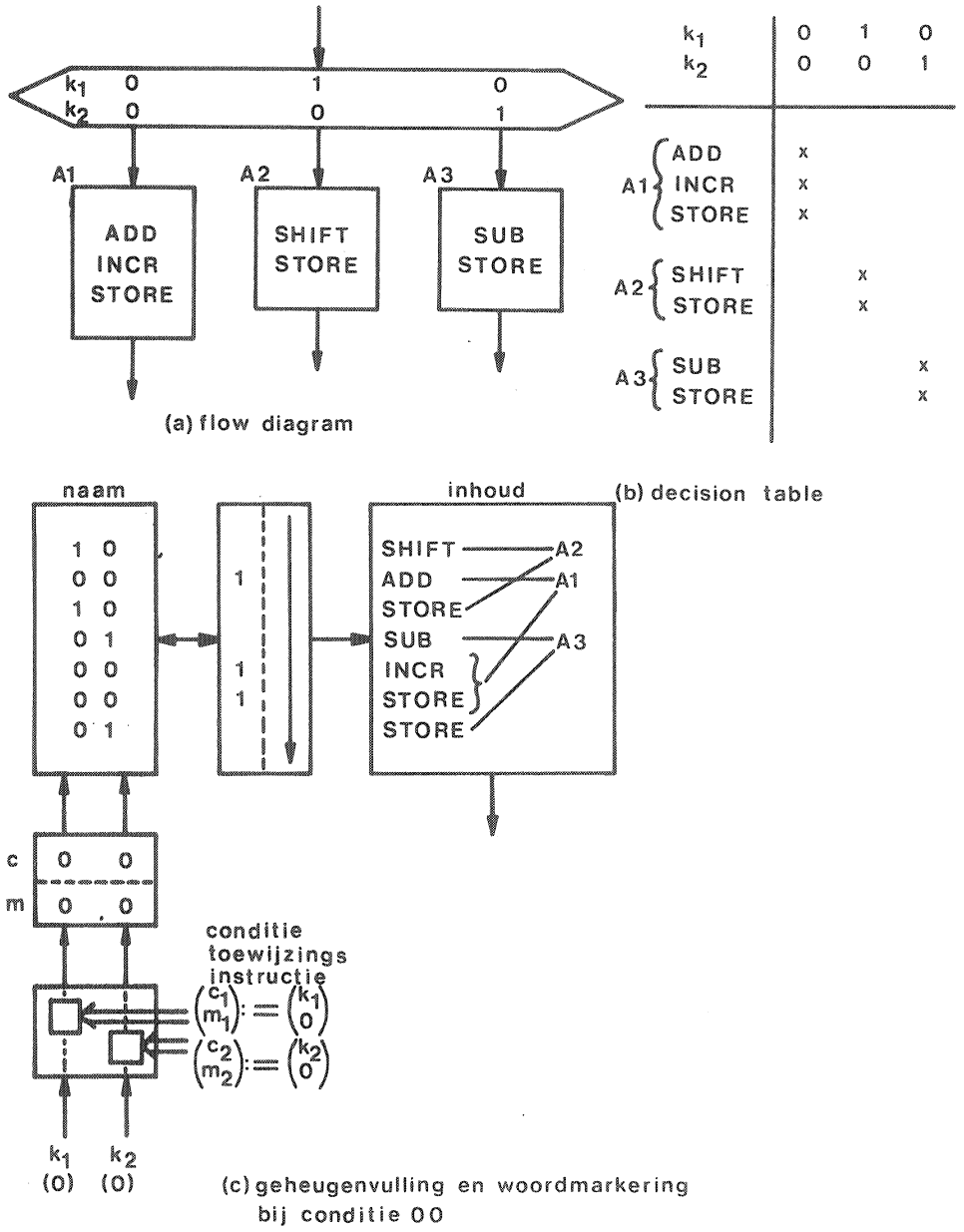


Fig.3.4. implementatie van vertakking

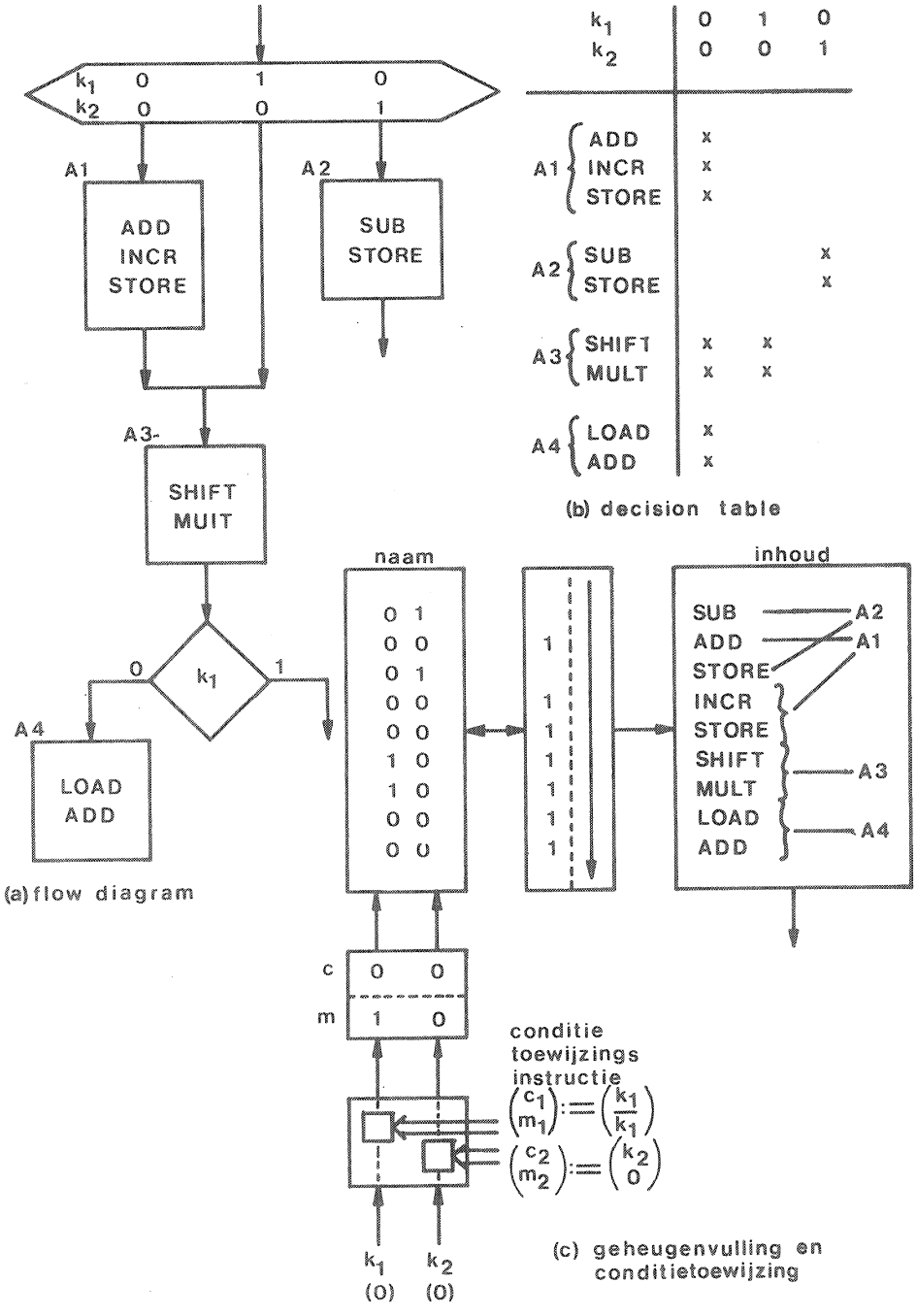


Fig.3.5. implementatie van vereniging (met hervertakking)

00 in het comparand register en een masker 10 (bit 1 gemaskeerd), zodat alle instruktienamen, met op de tweede plaats een 0, associeren. Om de instrukties in de juiste volgorde te kunnen lezen met de ripple schakeling is het noodzakelijk dat de instrukties volgend op de vereniging worden opgeslagen beneden de instrukties van de takken boven de vereniging. In fig. 3.5.c betekent dit: A1 voor A3 en A4. Fig. 3.5 laat zien dat door in het begin één keer te associëren op de optredende conditie het hele pad door het programma wordt gemarkeerd. Verder demonstreert fig. 3.5 dat een lege tak geen instruktie nodig heeft. Dit geldt eveneens voor een hervertakking.

Een systematische programmeermethode.

Uit de implementeerbaarheid van de drie programma-elementen volgt dat elke programma-struktuur gerealiseerd kan worden met de gegeven AM-struktuur. In het volgende wordt aan de hand van een voorbeeld getoond dat dit op een systematische wijze kan geschieden.

Elk programma kan opgebouwd worden gedacht uit een aantal zogenaamde komplexen, die beginnen met een ingangsvereniging en een ingangsvertakking. Daarachter volgen lineaire takken, verenigingen en hervertakkingen [19]. De verdeling in komplexen hangt samen met de probleem-georiënteerde koppeling van de condities. De associatie bij de ingang van het komplex markeert een komplexpad, dat zich tot aan een der uitgangen uitstrekt. Bij het verwerken van de gemarkeerde instrukties wordt de volgnaam samengesteld door conditietoewijzingsinstrukties in het pad. Een associatie aan het eind van het pad markeert automatisch het nieuwe pad in het volgkomplex. Per komplex wordt een decision-table opgesteld. Om de instruktienamen uniek te houden worden de instruktienamen uitgebreid met een komplexnaam. Bij het wisselen van komplex moet een verwijzingsinstructie naar het volgkomplex gegeven worden, d.w.z. de nieuwe komplexnaam wordt in het zoekregister geplaatst.

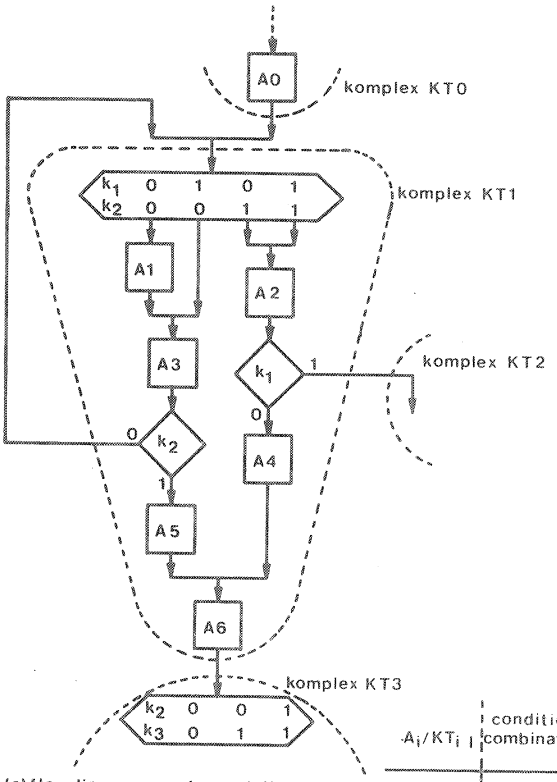
De methode van werken wordt uitgelegd aan de hand van het programma van fig. 3.6.a. In deze figuur is een verdeling in komplexen aangegeven. Aan de hand van het flowdiagram kan men voor elk komplex een decision-table opzetten (zie fig. 3.6.b voor komplex

KT1). De wisseling van komplex is afhankelijk van de padkeuze en dus van de condities, zodat per komplex een komplexwisselingtabel (complex-table) opgesteld kan worden (fig. 3.6.c).
Evenals bij de vorige voorbeelden worden de namen van de takken afgeleid van de condities. Dit kan als volgt op een systematische wijze geschieden. De decision-table en complex-table kunnen beschouwd worden als een waarheidstabel: het optreden van de acties als logische funktie van de condities. B.v. aktie A3 wordt uitgevoerd als $\bar{k}_1\bar{k}_2 + k_1\bar{k}_2 = 1$. Dit is te vereenvoudigen volgens de Boole algebra tot $\bar{k}_1$ en te noteren als x0 (met x = don't care). Nu wordt x0 als naam voor A3 gekozen. Het is echter niet altijd mogelijk om de functie te vereenvoudigen tot één term, b.v. A6: $k_1\bar{k}_2 + \bar{k}_1 k_2$, zodat de instrukties van A6 onder twee namen moeten worden opgeslagen: 10 en 01. Op deze wijze ontstaan de eerste 3 kolommen van de naamtabel van fig. 3.6.d. Vervolgens moeten de logische namen afgebeeld worden op de bits van het AM-naamgedeelte. Deze afbeelding geschiedt componentsgewijze, dat wil zeggen per kolom van de naamtabel. Aangezien de logische naamcomponenten driewaardig zijn (0,1,x), zijn maximaal twee AM-naambits nodig per conditiebit: $k_i \rightarrow n_j n_k$, b.v.: 0-->00, 1-->11 en x-->01. Dit houdt in dat ook de aangeboden conditie overeenkomstig moet worden afgebeeld. Immers $k_i = 0$ moet associëren met $n_j n_k = 00$ (0) en $n_j n_k = 01$ (x), terwijl $k_i = 1$ moet associëren met $n_j n_k = 11$ (1) en $n_j n_k = 01$ (x). Dit wordt bereikt met de toewijzing:

$$\begin{pmatrix} c_j & c_k \\ m_j & m_k \end{pmatrix} := \begin{pmatrix} k_i & k_i \\ k_i & \bar{k}_i \end{pmatrix} .$$

waarbij $\begin{pmatrix} c_j \\ m_j \end{pmatrix}$ het j-de bitpaar van het zoekregister voorstelt.

Het kan echter voorkomen, dat in een bepaalde kolom van de logische naamlijst niet alle drie de waarden 0, 1 en x voorkomen, maar alleen 0 en 1, of 0 en x, of 1 en x.



(a) flowdiagram met verdeling in komplexen

k_1	0	1	0	1
k_2	0	0	1	1
A1	x			
A2			x	x
A3	x	x		
A4			x	
A5		x		
A6		x	x	

(b) decision table

k_1	0	1	0	1
k_2	0	0	1	1
KT1	x			
KT2				x
KT3		x	x	

(c) complex table

A_i/KT_i	conditie combinatie		logische naam		AM-naam n_{11}	n_{12}	n_2
A1	$\overline{k_1}$	$\overline{k_2}$	0	0	0	0	0
A2	x k_1	k_2	X	1	0	1	1
A3	x k_1	$\overline{k_2}$	X	0	0	1	0
A4	$\overline{k_1}$	k_2	0	1	0	0	1
A5	k_1	$\overline{k_2}$	1	0	1	1	0
A6/KT3	k_1	$\overline{k_2}$	1	0	1	1	0
	$\overline{k_1}$	k_2	0	1	0	0	1
KT1	$\overline{k_1}$	$\overline{k_2}$	0	0	0	0	0
KT2	k_1	k_2	1	1	1	1	1

(d) instructienaamtoewijzing/ conditietoewijzing

mode I : $\begin{pmatrix} c_2 \\ m_2 \end{pmatrix} := \begin{pmatrix} k_2 \\ 0 \end{pmatrix}$

mode IV : $\begin{pmatrix} c_{11} c_{12} \\ m_{11} m_{12} \end{pmatrix} := \begin{pmatrix} k_1 k_1 \\ k_1 \overline{k_1} \end{pmatrix}$

Fig.3.6. een systematische programmeermethode

In deze gevallen kan de naamcomponent worden afgebeeld op één AM-naambit. Bijvoorbeeld bij de combinatie 0,x wordt gekozen voor de afbeelding k_i--> n_j : 0-->0, x-->1. Een aangeboden conditie k_i= 0 moet associëren op n_j= 0 (0) en n_j= 1 (x), terwijl k_i= 1 alleen moet associëren op n_j= 1 (x). Dit resulteert in de conditietoewijzing:

$$\begin{pmatrix} c_j \\ m_j \end{pmatrix} := \begin{pmatrix} k_i \\ \overline{k}_i \end{pmatrix}$$

De naamafbeelding en de conditietoewijzingsmodi van de vier genoemde gevallen zijn weergegeven in fig. 3.7 [17].

toewijzings modus	naam toewijzing logisch	naam toewijzing geheugen	zoek argument toewijzing
I	0 $(\overline{k_i})$	0	$\begin{pmatrix} c_j \\ m_j \end{pmatrix} := \begin{pmatrix} k_i \\ 0 \end{pmatrix}$
	1 (k_i)	1	
II	0 $(\overline{k_i})$	0	$\begin{pmatrix} c_j \\ m_j \end{pmatrix} := \begin{pmatrix} k_i \\ k_i \end{pmatrix}$
	x $(\overset{x}{k_i})$	1	
III	x $(\overset{x}{k_i})$	0	$\begin{pmatrix} c_j \\ m_j \end{pmatrix} := \begin{pmatrix} k_i \\ k_i \end{pmatrix}$
	1 (k_i)	1	
IV	0 $(\overline{k_i})$	00	$\begin{pmatrix} c_j & c_k \\ m_j & m_k \end{pmatrix} := \begin{pmatrix} k_i & k_i \\ k_i & k_i \end{pmatrix}$
	x $(\overset{x}{k_i})$	01	
	1 (k_i)	11	

Fig.3.7. instructienaam en conditiebittoewijzingsmodi
(x geeft don't care aan: 0 of 1)

Deze tabel is gebruikt om de naamtabel van fig. 3.6.d kompleet te maken: in de eerste kolom van de naamtabel komen 0, 1 en x voor, zodat modus IV gebruikt is; in de tweede kolom komen alleen 0 en 1 voor, hetgeen leidt tot de keuze van modus I.
De resulterende geheugenvulling is weergegeven in fig. 3.8. De instruktienamen bestaan uit twee delen: een komplexnaam (KT) en een conditienaam (X). De volgorde van opslag van instrukties is alleen van belang bij: A1 voor A3, A3 voor A5, A2 voor A5 en A4 voor A6. Om te voorkomen dat de instrukties van A6 en de verwijzing naar KT3 tweemaal moeten worden opgeslagen is de volgende methode toegepast:

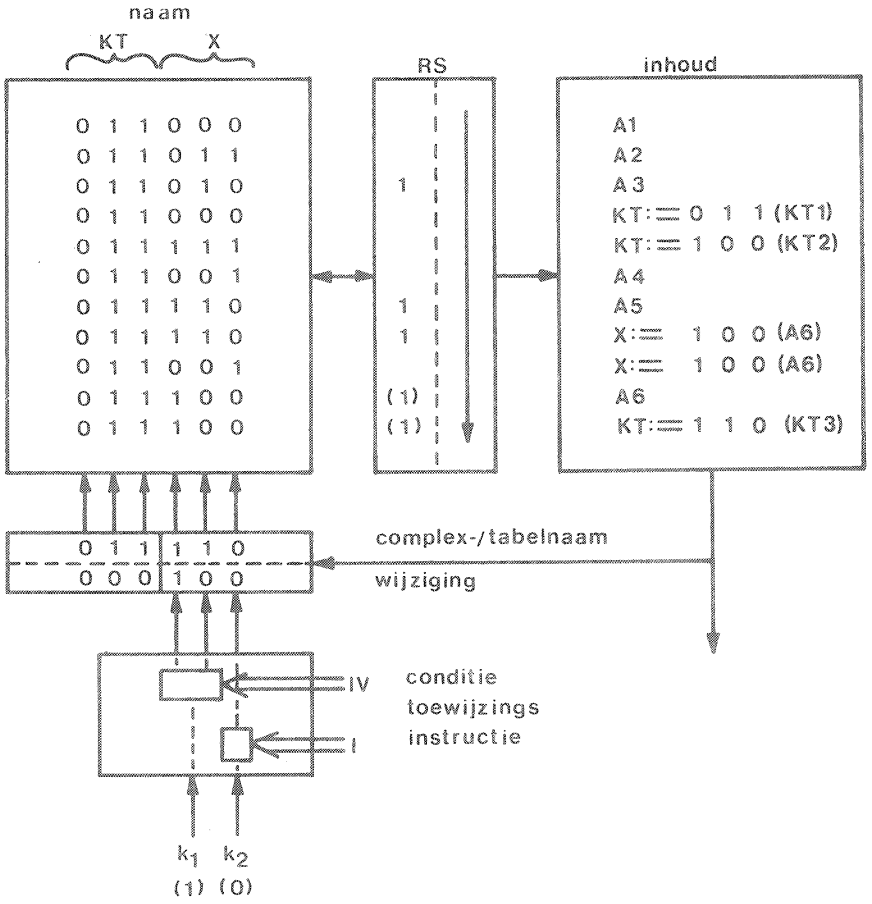


Fig.3.8.geheugenvulling met programma van Fig.3.6.

de instrukties van A6 en de verwijzing naar KT3 worden opgeslagen onder een nog vrije naam (b.v.: 100) en achter de twee namen voor A6 wordt een verwijzingsinstruktie (X:= 100) geplaatst. Samen met deze verwijzing moet een nieuwe associatie plaatsvinden om de instrukties van A6 te markeren, echter zonder dat de oude markeringen verdwijnen. Er moet dus een vereniging van associaties kunnen plaatsvinden. Een verwijzing naar een volgcomplex (wat ook hetzelfde komplex kan zijn) vindt plaats door de verwijzingsinstrukties KT:= KT_i.
Als voorbeeld zijn in fig. 3.8 de condities 10 aangeboden en de door de associatie ontstane markeringen aangegeven. De tussen haakjes geplaatste markeringen ontstaan door de verenigingsassociatie volgend op de verwijzing X:= 100.
In deze paragraaf is getoond dat een associatief programmmageheugen leidt tot een bruikbaar rekenmachineconcept, dat het mogelijk maakt om de gebruikersvriendelijke decision-table-programmeringstechniek op een efficiënte wijze te implementeren.

4. Conclusie.

Getoond is dat het associatieve geheugen een zeer bruikbare bouwsteen vormt voor computersystemen. In paragraaf 2 is een toepassing besproken waar het impliciete parallellisme van het AM tot het uiterste wordt benut. Hierbij is de verwerkingssnelheid te verhogen door toevoeging van extra hardware aan het response-store. In paragraaf 3 is getoond dat een associatief geheugen ook toepasbaar is als programma-geheugen. Het parallellisme wordt hier vooral benut bij het uitvoeren van meervoudige vertakkingen en het adresseren van gestruktureerde variabelen.

Literatuur.

[1] A cryotron catalog memory system
A. Slade en H.O. McMahon, proc. 1956 EJCC, p. 115-120

[2] Retrieval operations and data representations in a context-addressed disc-system
S.Y.W. Su, G.P. Copeland en G.J. Lipovski
ACM SIGPLAN Notices, vol. 10, nr.1, jan 1975

[3] RAP - An associative processor for database management
E.A. Ozkarahan, S.A. Schuster en K.C. Smith
AFIPS conference proceedings 1975, p. 379-387

[4] The design of a rotating associative memory for relational database applications
C.S. Lin, D.C.P. Smith en J.M. Smith
ACM trans. on databases, vol. 1, nr. 1, maart 1976, p. 53-65

[5] The architecture of CASSM: A cellular system for non-numeric processing
G.P. Copeland, G.J. Lipovski, S.Y.W. Su
proc. 1th Annual Symp.on Comp.Architecture, dec. 1973, Florida

[6] Data base machines are coming, D.K. Hsiao
Cellular logic devices: concepts and applications, S.Y.W. Su
Relational database machines, D.C.P. Smith, J.M. Smith
Database machines with large content addressable blocks and structural information processors, D.S. Kerr,
Computer, maart 1979, p. 7-79

[7] Virtual Memory, P.J. Denning
Computing Surveys, vol. 2, nr. 3, sept. 1970, p. 153-188

[8] Digital system implementation
G.A. Blaauw, Prentice Hall 1976, p. 265-274

[9] Interrupt processing with queued content-addressable memories
J.D. Erwin, E.D. Jensen
Fall joint computer conf. 1970, p. 621-627

[10] **Functional memory as a general purpose systems technology**
M. Flinders, P.L. Gardner, R.J. Llewelyn, J.F. Minschull
IEEE proc. of the comp. conf. 1970, Washington D.C. p. 314-324

[11] **Functional memory and its microprogramming implications**
P.L. Gardner
IEEE transactions on computers, vol. C-20, nr. 7, juli 1971

[12] **Signetics Field Programmable Arrays Handbook,** febr. 1976

[13] **PAL, programmable array logic handbook, MMI, 1978**

[14] **A production implementation of an associative array processor STARAN,** J.A. Rudolph
AFIPS conf. proc. 1972, vol. 41, part I, p. 229-241

[15] **STARAN/RADCAP hardware architecture,** K.E. Batcher
1973 Sagamore comp. conf. on parallel processing, p. 147-152

[16] **Ein Beitrag zur Hardware Unterstützung der Speicher - und namens Verwaltung durch Assoziativspeicher**
I. Karlowsky, dissertatie 1976, Universiteit van Braunschweig

[17] **Programmiertechnik und Ablaufsteuerung in einem assoziativen Programmspeicher**
W. Lawrenz, dissertatie 1976, Universitet van Braunschweig

[18] **Realisatie van associatieve geheugens**
C.F.J.M. Stork, MC-colloquium 1980

[19] **Procedural decision tables and their implementation**
R. Thurner, K. Bauknecht
Intern. Comp. Symp. 1973, p. 259-263

[20] **Content addressable parallel processors**
C.C. Foster, Van Nostrand Reinhold Company, 1976

REALISATIE VAN ASSOCIATIEVE GEHEUGENS

C. STORK
Technische Hogeschool Eindhoven

Samenvatting.

Dit artikel behandelt het principe van associatieve geheugens. Kennis over dit principe is onontbeerlijk om inzicht in mogelijke toepassingen van associatieve geheugens te verkrijgen.
Na een meer theoretische opbouw waarbij de diverse eigenschappen van associatieve geheugens aan het licht komen, worden een tweetal reeds gerealiseerde circuits besproken.

1. Begripsomschrijving.

In het algemeen geldt dat geheugens gebruikt worden voor het opslaan van informatie. Deze informatie behoort bij variabelen. Bij het meest verbreide geheugentype, het zogenaamde Random Access Memory (RAM) wordt de koppeling tussen variabelen en hun waarde (inhoud) via een direkt adresseringsmechanisme tot stand gebracht; iedere geheugenplaats heeft zijn eigen specifieke adres (fig. 1.1). De koppeling tussen een variabele en het bijbehorende adres kan b.v. door middel van een programma (assembler, compiler, interpreter enz.) tot stand gebracht worden.
Er moet dus een soort boekhouding worden bijgehouden m.b.t. de plaats waar informatie te vinden is.
Het kan ook voorkomen dat men op zoek is naar variabelen op grond van de inhoud zelf (data bestanden). In dat geval zullen er, voordat de juiste variabele gevonden is, een aantal vergeefse zoekpogingen geweest zijn. Denk hierbij aan lineair-, toevallig- of

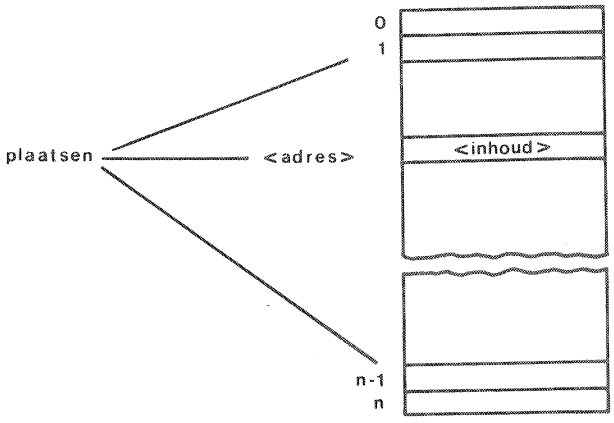


fig. 1.1

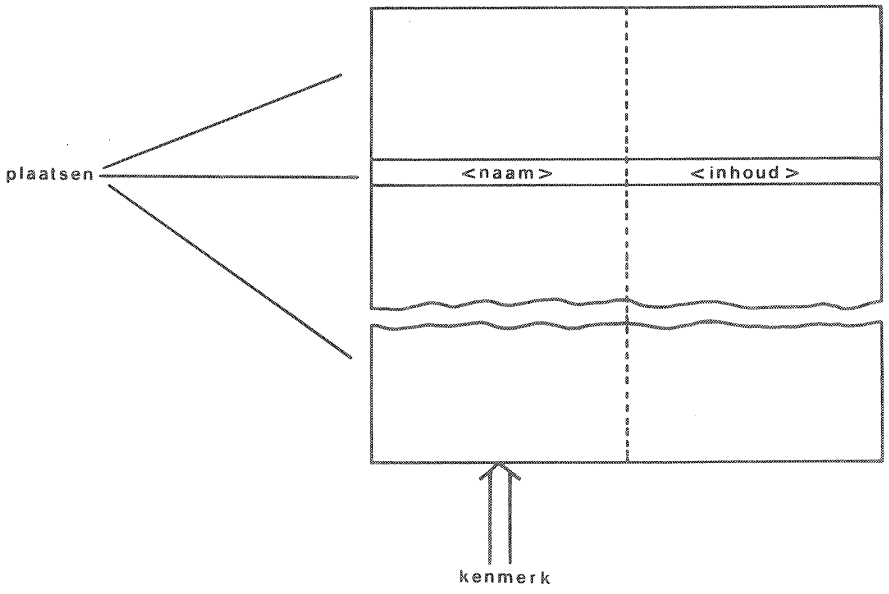


fig. 1.2

gesorteerd zoeken ofwel zoeken d.m.v. een algorithme zoals m.b.v. een hash funktie.
In fig. 1.2 is symbolisch een associatief geheugen (AM) getekend waarmee het onderhavige probleem in hardware is opgelost. Het adresseringsmechanisme ontbreekt. De waarde van een variabele wordt nu bepaald m.b.v. een op die plaats vermelde naam van de variabele. Daartoe wordt een kenmerk aangeboden. Dit kenmerk fungeert als een sleutel (key). Onder associëren verstaan we nu het koppelen van het aangeboden kenmerk aan één of meerdere namen.
Een AM is een inhoudsadresseerbaar geheugen. In de literatuur wordt dikwijls de naam 'Content Adressable Memory' (CAM) aangetroffen. Opgeslagen variabelen zijn dus niet plaatsgebonden.
Het is dan ook onbelangrijk waar bepaalde variabelen zijn ondergebracht.
Het aantal aanwezige kenmerken behoeft daarom niet gelijk te zijn aan het aantal mogelijke kenmerken.
Informatie wordt gevonden op grond van de bij die informatie behorende (deel)naam. Informatie bestaat dus uit een naamgedeelte en een inhoudsgedeelte.
Gezien de wens om op deelnamen te kunnen zoeken is het kenmerk opgebouwd uit twee gedeelten:

a) het comparand gedeelte. Hierin bevindt zich dat gedeelte van de sleutel waarop een of meerdere namen moeten associëren.
b) het maskergedeelte. Hierin staat aangegeven welk gedeelte van de sleutel niet mee doet met het associatieproces.

In fig. 1.3 is een associatief geheugen getekend, opgedeeld in een aantal funktionele blokken. De funkties van deze blokken zullen hieronder globaal worden beschreven. In het volgende hoofdstuk zal op de hardware realisatie van deze funkties worden ingegaan.

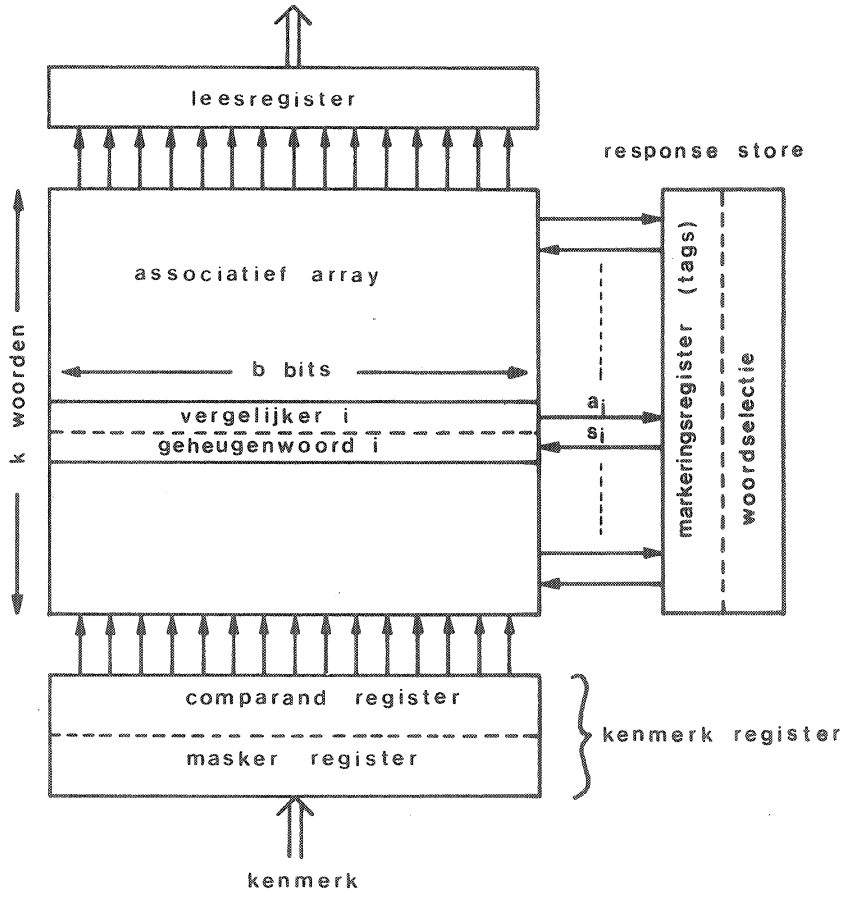


fig. 1.3 Blokschema associatief geheugen.

- associatief array.
  Dit array bevat behalve de geheugenwoorden van b bits een even groot aantal vergelijkschakelingen. Het aangeboden kenmerk kan hierdoor gelijktijdig (parallel) vergeleken worden met alle opgeslagen woorden. Meerdere namen kunnen op hetzelfde kenmerk associëren. Dit wordt ook wel "multiple match" genoemd.

- kenmerkregister.
  dit register bestaat eigenlijk uit twee registers, namelijk een comparandregister en een maskerregister.

- leesregister.
  Dit register bevat de laatst gelezen data.

- markeringsregister (response-store).
  Het response-store bevat een markeringsregister (tag flip-flop's) waarmee bepaalde woorden kunnen gemarkeerd.
  Op gemarkeerde woorden kunnen parallel schrijf- en lees-operaties worden uitgevoerd.

2. Een realisatie.

De opbouw van een associatief geheugen zoals hieronder beschreven is, is bedoeld als uitgangspunt om inzicht te verschaffen in de werking van het geheel.
Achtereenvolgens komen de blokken van fig. 1.3 ter sprake.

2.1. Kenmerkregister.

Zoals gebleken is, is de behoefte aan een associatie over de gehele comparand niet altijd aanwezig. Vandaar de invoering van een maskerregister. In fig. 2.1 is een bitslice j, met j=1..b, van het kenmerkregister afgebeeld.
Voor het aanbieden van de 3 toestanden "0", "1" en "x" (don't care) zijn 2 bitdraden (bl_j en $b0_j$) nodig (zie tabel 2.1).

m_j	c_j	bl_j	$b0_j$	logisch
0	0	0	1	0
0	1	1	0	1
1	0	0	0	x
1	1	0	0	x

tabel 2.1.: conversietabel kenmerkregister.

Uit de tabel volgt:

$$bl_j = c_j \bar{m}_j$$

$$b0_j = \bar{c}_j \bar{m}_j$$

De comparand en het masker worden in de registers gelezen onder besturing van de "strobe-signalen" stb_c en stb_m.

2.2. Het associatief array.

Associatieve geheugens behoren tot de klasse van array logica, d.w.z. dat de elementen rij- en kolom-gewijs worden aangestuurd of uitgelezen. In fig. 2.2 is het principeschema van een bitcel uit het associatieve array getekend. De bitcel bevat een flipflop $W_{i,j}$ voor het onthouden van een "0" of een "1".
De stand van de bitlijnen wordt vergeleken met de stand van de flip-flop. De associatielijn a_i geeft aan of het geheugenwoord w_i associeert ($a_i = 1$) met het aangeboden kenmerk:

$$a_i = \bigwedge_{j=1}^{k} \{(W_{i,j} \equiv c_j) + m_j\} = \bigwedge_{j=1}^{k} (\overline{W_{i,j} \cdot b0_j} \cdot \overline{\overline{W_{i,j}} \cdot bl_j}) \qquad (2.1)$$

$a_i = 1$ (associatie) treedt dus op als voor alle bits $W_{i,j}$ geldt dat of de bit associeert met de overeenkomstige bit van de comparand ($W_{i,j} \equiv c_j$), of de bit is gemaskeerd ($m_j = 1$).
Dit proces geschiedt parallel voor alle woorden W_i, met i=1..k, uit het associatieve array. Tabel 2.2 toont de associatie van bit j. De overige bits worden gemaskeerd verondersteld ($m_x = 1$, met $x \neq j$).

m_j	c_j	bl_j	$b0_j$	$W_{i,j}$	a_i	
0	0	0	1	0	1	<-- associëren
0	0	0	1	1	0	
0	1	1	0	0	0	
0	1	1	0	1	1	<-- associëren
1	x	0	0	x	1	<-- maskeren

tabel 2.2.: waarheidstabel voor de vergelijkschakeling voor bit j.

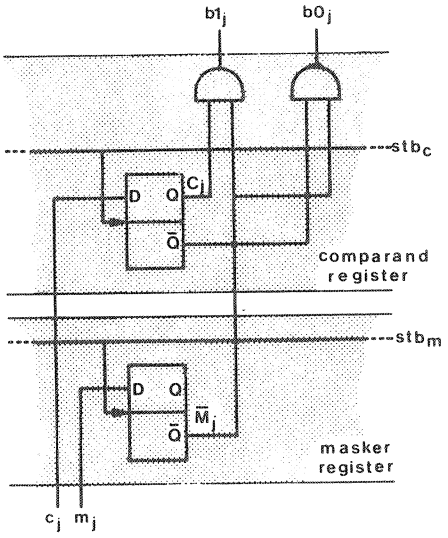


fig. 2.1. Bitslice j van het kenmerkregister.

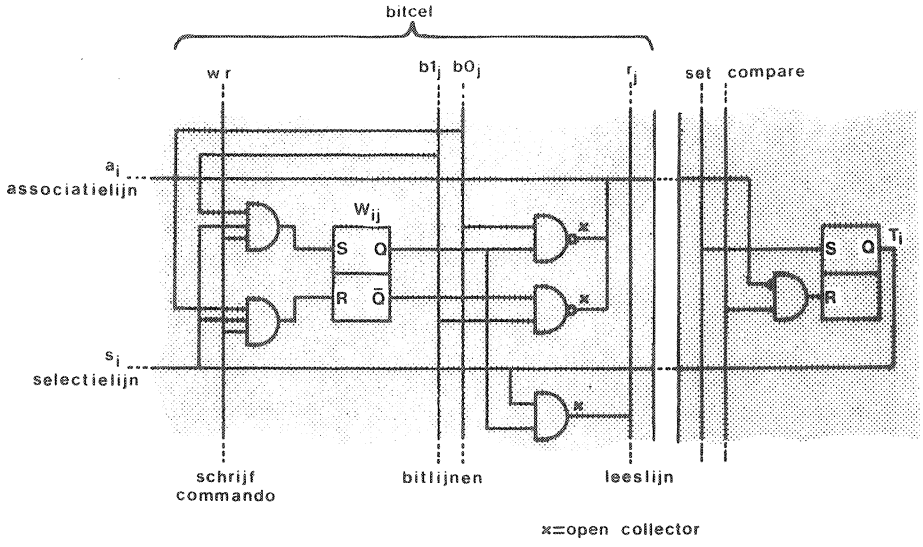


fig. 2.2. Bitcel j van woord i + gedeelte response store.

Voor het schrijven en/of het lezen wordt gebruik gemaakt van de selectielijn (s_i). Indien s_i=1 en een schrijfpuls verschijnt op de schrijflijn (wr), dan wordt de stand van de bitlijnen door de flip-flop overgenomen.
Indien een don't care wordt aangeboden (m_j=1, ofwel $b0_j$= $b1_j$=0) zal de inhoud van de flip-flop ongewijzigd blijven, zodat het mogelijk is om slechts delen van geheugenwoorden te beschrijven.
De lijnen a_i en s_i zijn verbonden met het response store.
De associatielijn a_i kan tag-flip-flop T_i resetten. De selectielijn s_i kan alleen actief worden indien T_i=1 (zie par. 2.3).
Door het aansturen van meerdere selectielijnen is parallel schrijven (modificeren) mogelijk.
Tijdens het lezen wordt de inhoud van het geselecteerde woord op leeslijn (r_j) geplaatst. Worden meerdere woorden geselecteerd, dan zullen de leeslijnen de (wired)-AND-funktie bevatten van de overeenkomstige bits van de geselecteerde woorden. Voor de aansturing van de leeslijnen en de associatielijnen worden poorten van het open-collector type gebruikt.

2.3. Response store.

2.3.1. Tijdens het vergelijken.

Het response store bevat een markeringsregister (tag-flip-flop's) en wel voor ieder woord tenminste één bit (zie figuur 2.2).
Met behulp van een set-operatie worden alle tag-flip-flop's T_i op "1" gezet. De woorden welke niet associëren zullen tijdens compare =1 middels hun associatielijn a_i de betreffende tag-flip-flop terug op "0" zetten. M.a.w. na een compare-operatie geeft het gezet zijn van T_i aan dat er een associatie heeft plaats gehad. Indien vervolgens weer een vergelijkingsslag wordt uitgevoerd, dan kunnen alleen die tag-flip-flop's, die nog "1" zijn, op "0" worden gezet.
Op deze wijze ontstaat dan een AND-funktie over de achtereenvolgende associaties. Voor de funktie $\delta(T_i)$ die de toestandsovergang van T_i beschrijft geldt:

$$\delta(T_i) = set + T_i.\overline{\overline{a_i}}.compare \qquad (2.2)$$

2.3.2. Tijdens het schrijven.

Bij het schrijven worden met het schrijfcommando alleen die woorden beschreven waarvan de selectie-lijn hoog is.

De inhoud van dat gedeelte van het comparand register dat niet gemaskeerd is, wordt door de betreffende bits van de geselecteerde woorden overgenomen.

De werking is als volgt. In fig. 2.2 is de selectie-lijn verbonden met de uitgang van de tag-flip-flop:

$$s_i = T_i \qquad (2.3)$$

Door het geven van een write puls worden alle geselecteerde woorden gelijktijdig beschreven; een zogenaamde "multi-write"-opdracht.

Na het geven van een schrijfopdracht geldt in het algemeen voor de toestandsovergang $\delta(W_{i,j})$:

$$\delta(W_{i,j}) = W_{i,j} \cdot (m_j + \bar{s}_i) + \bar{m}_j \cdot s_i \cdot c_j \qquad (2.4)$$

Voor het beschrijven van één van de geselecteerde woorden kan gebruik gemaakt worden van de ripple schakeling (par. 2.3.4.).

2.3.3. Tijdens het lezen.

Voor het lezen van het associatieve geheugen is per bit een lijn aangebracht (r_j in fig. 2.2). Deze lijn wordt gestuurd vanuit een open collector poort. Hierdoor is het mogelijk om van alle geselecteerde woorden de AND-funktie per bit in het leesregister (fig. 2.3) te laden.

Voor de bitwaarde r_j van het leesregister geldt dan:

$$r_j = \bigwedge_{i=1}^{k} s_i \cdot W_{i,j} \qquad (2.5)$$

Om de inhoud van één woord te lezen zal dat ene woord alleen geselecteerd moeten worden of zal, in geval van een multiple match, van de ripple-schakeling (zie volgende paragraaf) gebruik gemaakt moeten worden.

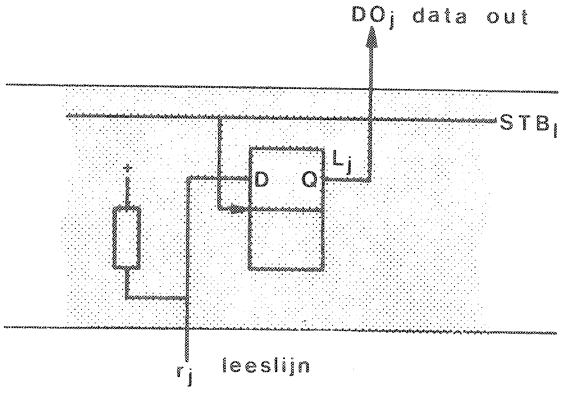


fig.2.3. Bitslice van het leesregister.

2.3.4. Ripple-schakeling.

In het geval dat sequentieel een lees- of schrijfactie ondernomen moet worden, kan gebruik gemaakt worden van de ripple-schakeling van fig. 2.4. Indien par/$\overline{ser}$ =1, dan is de schakeling gelijk aan die van fig. 2.2 en geldt verg. (2.3) voor de woordselectie. Na het "0" maken van par/$\overline{ser}$ blijft slechts de selectie actief van dat woord dat als eerste, van boven naar onder gezien, geselecteerd is. De overige selecties worden tijdelijk onderdrukt. Met de lijn par/$\overline{ser}$ wordt dus kenbaar gemaakt of de selectie parallel (par/$\overline{ser}$=1) dan wel serieel is (par/$\overline{ser}$=0).
Verg. (2.3) wordt dus uitgebreid tot:

$$s_i = (par/\overline{ser} + n_i).T_i \qquad (2.6)$$

waarin n_i aangeeft dat geen van de voorgaande tag-flip-flops op "1" gezet is (n_i=1). Is n_i=0, dan zijn een of meer voorgaande flip-flop's gezet. Voor n_i geldt dan de volgende recurrente betrekking:

$$n_i = \overline{T_{i-1}}.n_{i-1} \qquad \text{met } n_0=1 \qquad (2.7)$$

Indien par/$\overline{ser}$ =0 dan wordt het eerste woord met T_i= 1 geselecteerd. Van de overige geselecteerde woorden T_{i+x} worden de selecties onderdrukt, immers $n_{i+1} = \overline{T_i}.n_i=0$ daar $\overline{T_i}=0$.

Door het resetten van T_i wordt het mogelijk een volgend woord dat associeert te selecteren. Een en ander gebeurt met het signaal "next". Een puls op deze lijn zal de eerste tag-flip-flop die een "1" bevat op "0" zetten. Om te voorkomen dat vervolgens ook andere flip-flop's worden gereset, moeten de flip-flop's van het master-slave principe zijn, of moet de ripple opdracht synchroon (d.w.z. geklokt) worden uitgevoerd.

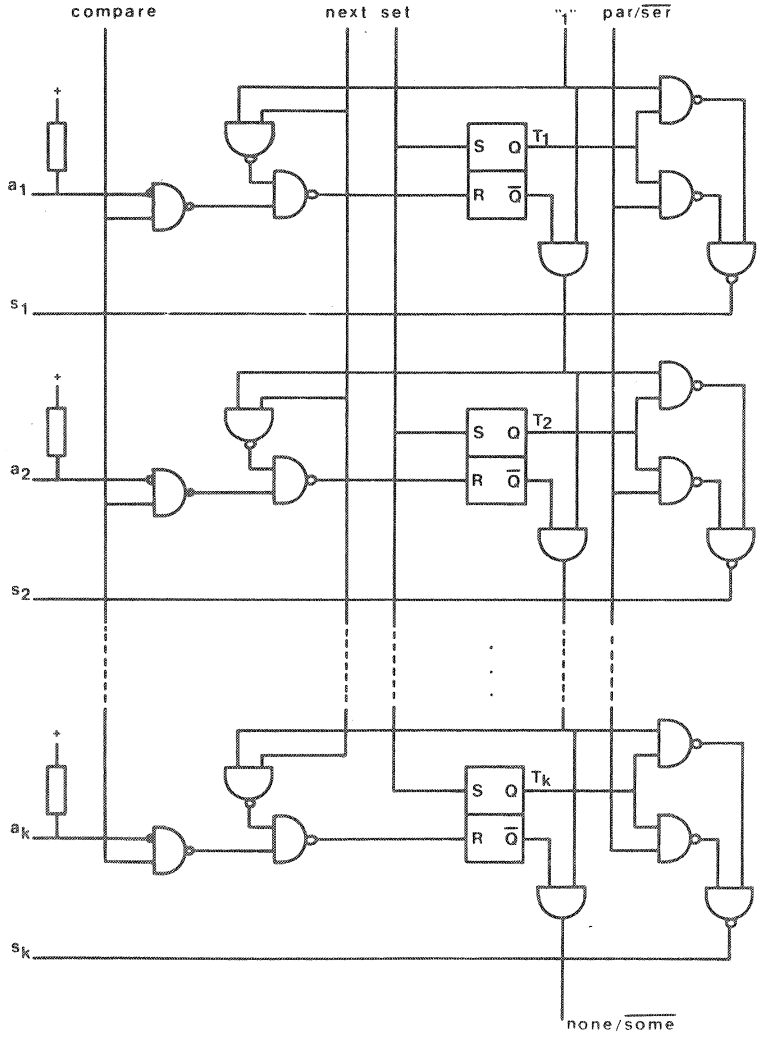


Fig.2.4. Response store.

Vgl. 2.2 kan nu worden uitgebreid tot:

$$\delta(T_i) = set + T_i \cdot (\overline{\overline{a_i} compare + next \cdot n_{i-1}}) \qquad (2.8)$$

De ripple-schakeling geeft ook de mogelijkheid om nieuwe woorden aan het AM toe te voegen (insert). Het toevoegen is op te vatten als het beschrijven van een geselecteerde lege plaats. De selectie van de lege plaats gebeurt door het aanbieden van het "leeg"-kenmerk (een gereserveerde bitcombinatie) en het vervolgens "0" maken van par/$\overline{ser}$, zodat het eerste lege woord geselecteerd wordt. Tenslotte volgt een schrijfopdracht waardoor de informatie uit het kenmerkregister op de geselecteerde plaats geschreven wordt.

3. Variaties op het principe.

3.1 Partieel associatief geheugen.

Aangezien de bitkosten voor een associatief geheugen (AM) hoger zullen zijn dan de bitkosten voor een random access geheugen (RAM) ligt het voor de hand het associatieve array alleen maar voor het naamdeel te gebruiken. Voor het inhoudsdeel wordt dan een RAM gebruikt (fig. 3.1). De selectie van de woorden via het response store moet dan zowel het naamgedeelte in het AM als het inhoudgedeelte in het RAM betreffen. Een volgende variatie lijkt uit kostenoogpunt reëel:

Het naamregister ontbreekt (besparing van het leesmechanisme). Indien nodig kan een kopie van het naamgedeelte in het (goedkopere) RAM worden ondergebracht.

Een tweede mogelijkheid is de inhoud van de naam te lezen m.b.v. een algorithme. Vullen we namelijk in het comparandregister de gemaskeerde plaatsen met b.v. "0", dan kunnen we door het masker, bit voor bit, weg te halen, nagaan of het woord wel of niet blijft associëren. Vervalt de associatie, dan moet het betreffende comparand-bit worden geïnverteerd. Het aantal compare-instructies is dus gelijk aan het aantal ontbrekende bits van het naamdeel.

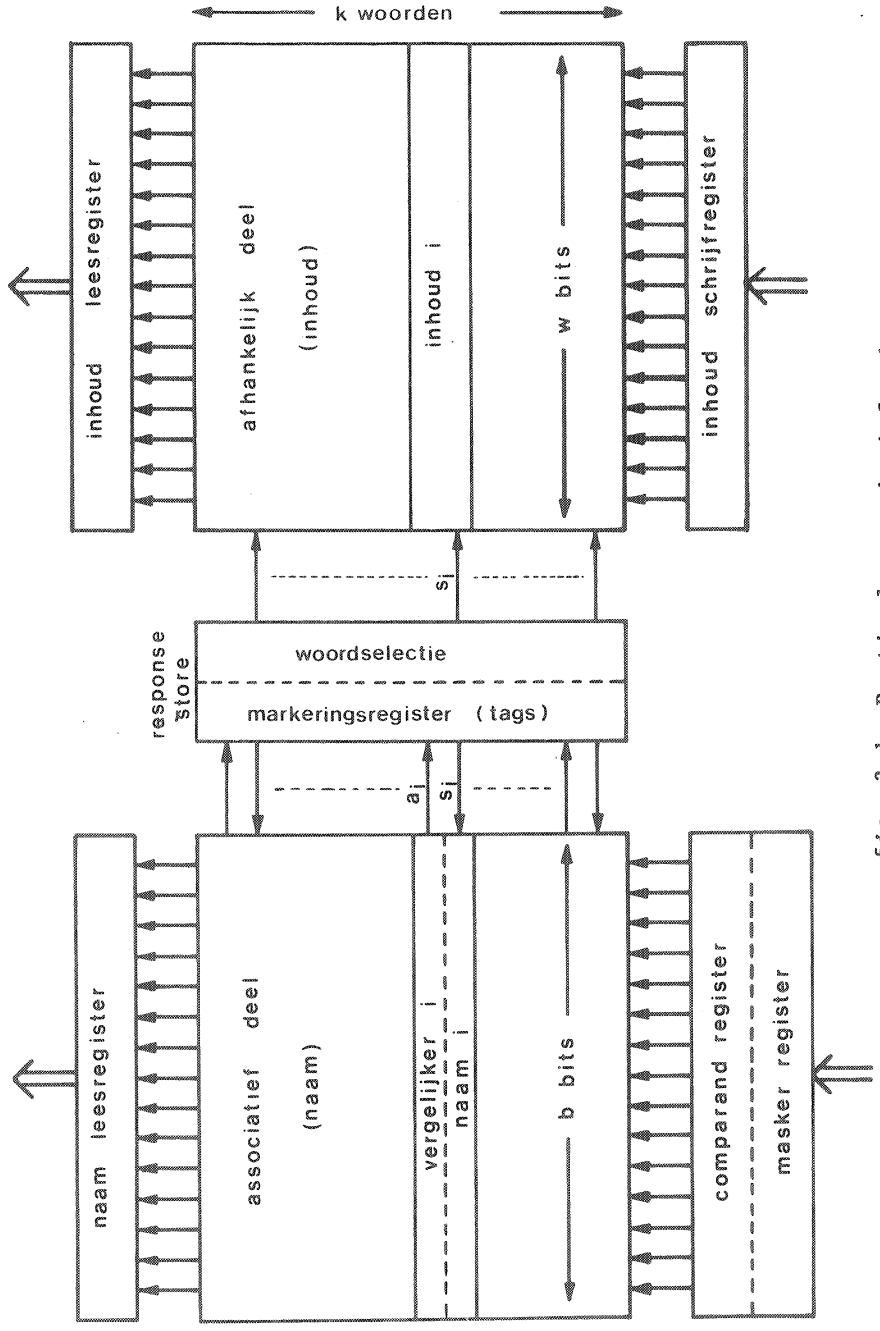


fig. 3.1. Partieel associatief geheugen.

Het probleem is dat alle tot dusver commercieel verkrijgbare RAM's zijn uitgevoerd met een decoderings-mechanisme voor het adres. Gevolg hiervan is dat de selectie-lijn (s_i) gecodeerd dient te worden m.b.v. een prioriteits encoder. Dit betekent dat per keer slechts <u>één</u> plaats geselecteerd kan worden. Hierdoor vervalt dus de mogelijkheid van parallelle arithmetische en logische bewerkingen op het inhoudsdeel.

3.2. <u>Het bitseriële associatieve geheugen.</u>

Bij het bitseriële associatieve geheugen wordt een belangrijke besparing aan hardware verkregen. De vergelijkschakeling per bit is vervangen door een vergelijkschakeling per woord. Dit gaat natuurlijk wel gepaard met verlies aan snelheid. De werking is als volgt (fig. 3.2): de geheugenmatrix is voor te stellen als een aantal één-bit RAM's "op zijn kant". Voor ieder woord is een RAM aanwezig. M.b.v. een teller en decoder wordt in iedere RAM na iedere telpuls de volgende bit aangewezen. Dit geldt ook voor het masker- en comparandregister. De bits van W_i en c worden in tijdmultiplex aan de vergelijkers aangeboden. De betreffende tag-flip-flop wordt gereset indien een ongelijkheid optreedt. Na b telpulsen is het gehele array vergeleken met het kenmerk en geven de tag-flip-flop's de associaties weer.
Tijdwinst is te behalen door gemaskeerde bits niet ter vergelijking aan te bieden.

Dit mechanisme leent zich ook goed voor het realiseren van associatieve achtergrondgeheugens. Veronderstel daartoe de 1-bit RAM's vervangen door b.v. een disk unit met een aantal vaste koppen. Als iedere kop voorzien is van de vergelijkschakeling gaat bovenstaand verhaal ook hier in sterke mate op.

3.3. <u>Andere vergelijkingskriteria.</u>

Door het uitbreiden van de functies van de vergelijkschakeling of door het uitbreiden van het aantal vergelijkschakelingen zal het

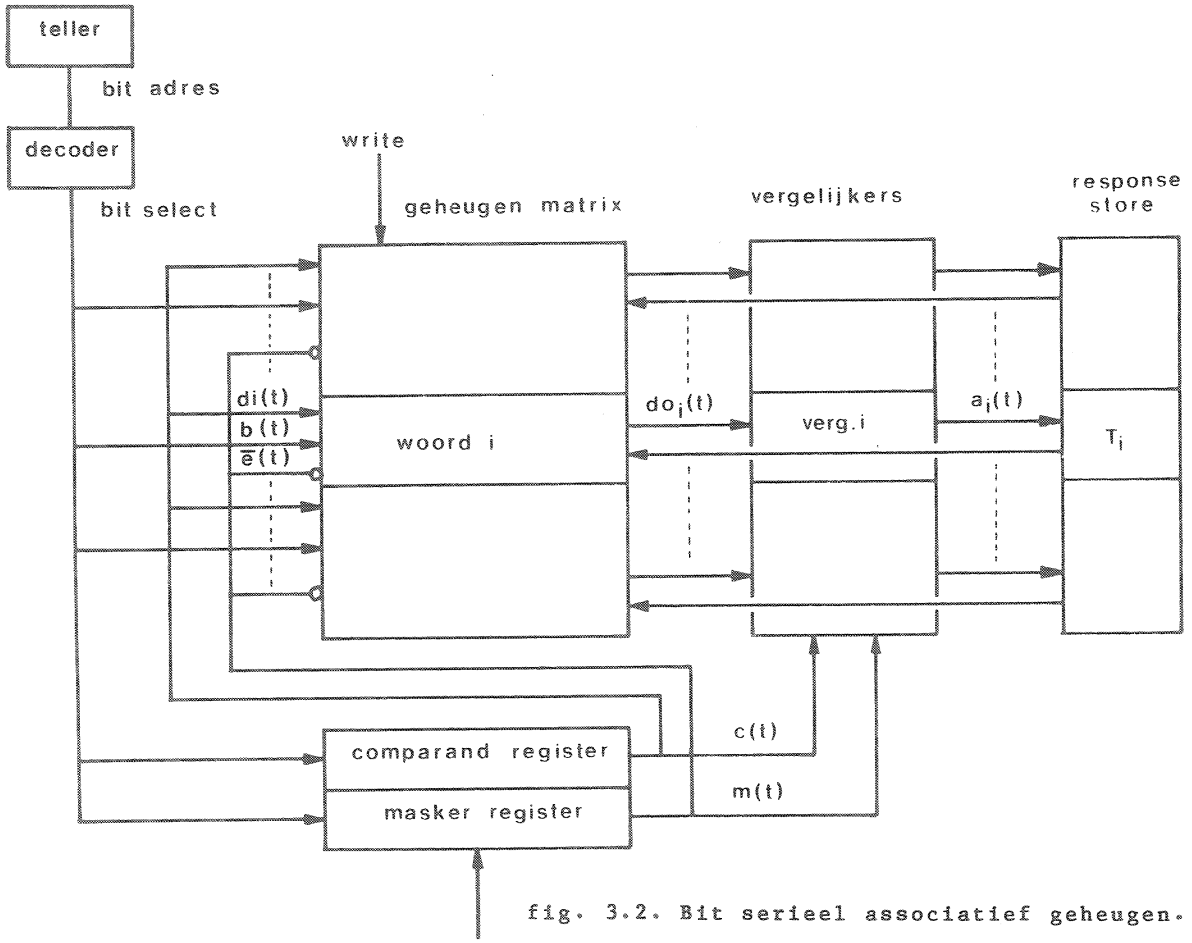


fig. 3.2. Bit serieel associatief geheugen.

mogelijk zijn meerdere vergelijkingskriteria uit te voeren, zoals ongelijkheid, groter dan, kleiner dan, enz.

4. Praktische realisatie.

Binnen de vakgroep Digitale Systemen van de Technische Hogeschool Eindhoven wordt momenteel gewerkt aan een realisatie van een associatief geheugen. Het ontwerp is gebaseerd op een tweetal geïntegreerde circuits, die in 1974 en 1975 ontworpen zijn binnen de vakgroep en onlangs op grotere schaal voorhanden zijn gekomen. De gebruikte technologie is I^2L (Integrated Injection Logic). Het ene IC, genaamd ASS-4, bevat een array van 64 bitcellen en is georganiseerd als 8 woorden van elk 8 bits lang. De array kan, logisch gezien, in horizontale zin (woordlengte) en in verticale zin (aantal woorden) onbeperkt worden uitgebreid (zie fig. 4.1). Het andere IC, genaamd ASS-3, bevat de besturing voor 8 woorden en wordt d.m.v. een ripple-circuit onderling verticaal verbonden. Met de mogelijkheid voor het aanbrengen van look-ahead circuits om de doorlooptijd van het ripple-circuit te verminderen is rekening gehouden.
Indien er behoefte bestaat tot het onthouden van associaties, zal een response store (zie hoofdstuk 2) moeten worden toegevoegd. Indertijd is bij het ontwerp ook rekening gehouden met een opzet zoals onder 3.1 is besproken, namelijk het partieel-associatief geheugen zonder leesschakeling voor het naamgedeelte. Dit betekent dat de array niet rechtstreeks gelezen kan worden. Dit geeft niet alleen een besparing aan chip-oppervlakte, maar wat belangrijker is, het reduceert het aantal aansluitpennen met 8. De bitlijnen, associatielijnen en selectielijnen leiden tot een aantal aansluitingen van 8x2 +8 +8 = 32. De logische opbouw van het associatieve array ASS-4 is weergegeven in fig. 4.2.

4.1 De bitcel.

In fig. 4.4 is de bitcel getekend, opgebouwd uit I^2L gates. Bij een I^2L transistor moet aan iedere basis een stroombron verondersteld worden. Is de basis niet aangesloten, dan zal alle

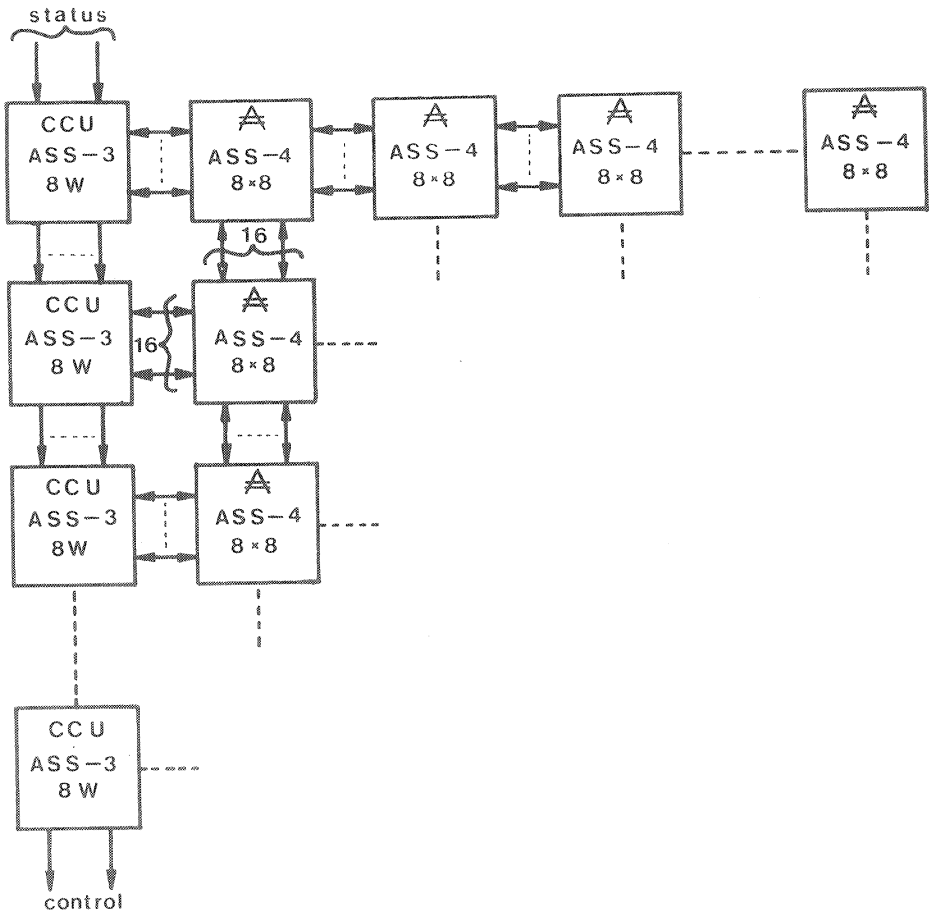


fig.4.1. Associatief geheugen m.b.v. ASS-3 (CCU) en ASS-4(≙).

stroom via de basis-emitter overgang lopen en de transistor in zijn verzadiging drukken, zodat de collector laag is ("0").
Wordt echter de basis omlaag getrokken, b.v door de collector van een andere transistor, dan loopt de stroom niet meer door de basis-emitter-overgang en zal het potentiaal van de collector door andere circuits worden bepaald; de collector is in dat geval te beschouwen als niet aangesloten.
In fig. 4.3 is een NOR poort gerealiseerd met 2 I^2L-transistors; hieruit moge de werking van zo'n transistor blijken.
Een I^2L-transistor kan meerdere collectoren bevatten.

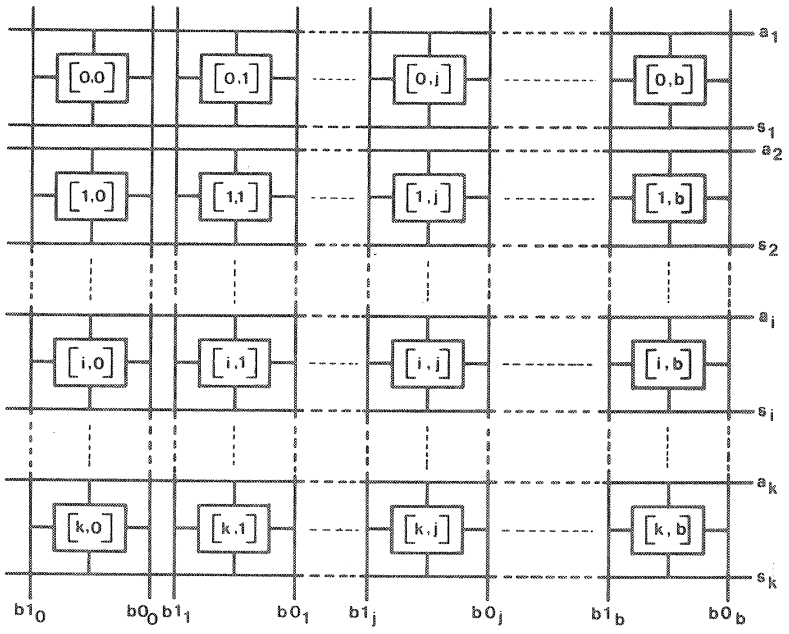


fig. 4.2. Organisatie van het associatief array m.b.v. ASS-4.

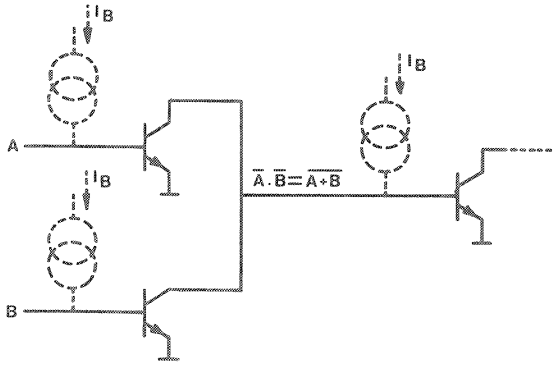


fig. 4.3. NOR-poort met I^2L-transistoren.

De bitcel (fig. 4.4) bevat een flip-flop (transistoren e en d) voor het onthouden van een logische "0" of een logische "1". Komt de stand van de flip-flop overeen met de stand van de bitlijnen, dan wordt gesproken van een bit-associatie of "match". Worden beide bitlijnen "1" gemaakt, dan doet de stand van de flip-flop er niet meer toe: we noemen dit een "don't care". In het geval dat beide bitlijnen "0" zijn, zal er nooit een associatie plaatsvinden: we noemen dit een "mismatch". M.b.v. het activeren van de selectielijn s_i wordt de stand van de bitlijnen door de flip-flop overgenomen: het schrijven van het kenmerk. Het uitlezen van de inhoud van een bitcel is niet direct mogelijk i.v.m. het ontbreken van een leescircuit.
M.b.v. de waarheidstabel in fig. 4.5 is de werking van de bitcel eenvoudig na te gaan.

4.2. De besturing. (C.C.U.)

In fig. 4.1 is te zien hoe met het besturings-IC (ASS-3) en met het 64-bits CAM-IC (ASS-4) een associatief geheugen is samengesteld. Dit besturings-IC bevat behalve een ripple automaat met adrescodering een mechanisme voor het bijhouden van de lege plaatsen. Het IC bevat twee flip-flops per woord (fig. 4.6).
Een flip-flop (F_i) dient voor het aangeven of een woord al of niet beschreven is, en een flip-flop (M_i) dient voor het onderdrukken van een associatie in het geval van een multiple match. Het C.C.U.-IC bevat de besturing van 8 woorden en werkt als volgt: als alle voorgaande woorden beschreven zijn ($\overline{full_{i-1}}=0$), als het betreffende woord nog niet beschreven is ($F_i=0$) en als een toevoegopdracht ("INSERT") wordt gegeven, dan resulteert dit in het op "1" zetten van F_i. Tevens wordt $\overline{full_i}$ "0" gemaakt als signalering aan een volgend woord, dat in aanmerking komt voor een schrijfopdracht. Het op "1" zetten van F_i kan ook bewerkstelligd worden middels een adresseringsmechanisme.
Het wissen van een woord geschiedt d.m.v. datzelfde adresseringsmechanisme of door een "clear match"-commando, waardoor gelijktijdig alle woorden, die associëren, worden gewist.

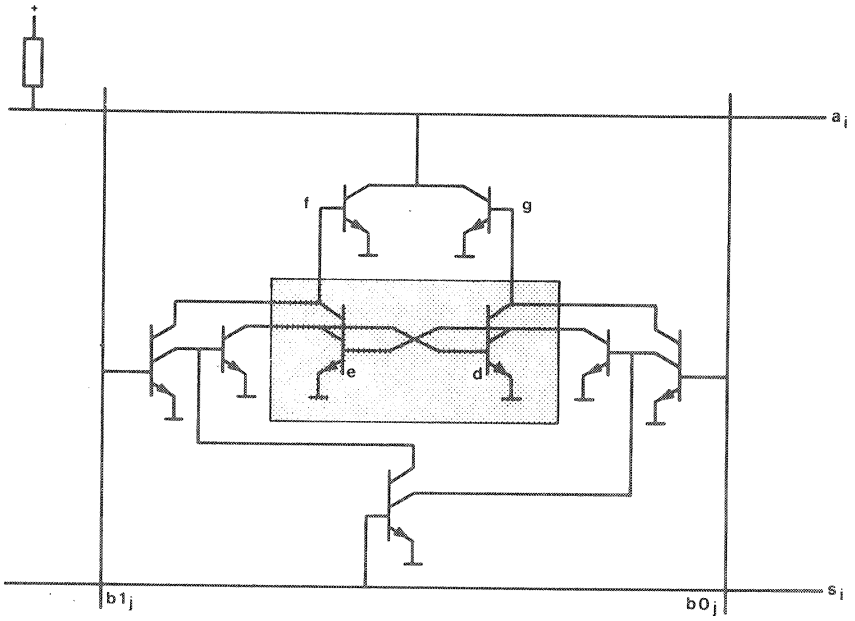


fig. 4.4. Geheugencel in I^2L.

$b1_j$	$b0_j$	d	e	f	g	a_i	
0	0	0	1	0	1	0	
0	1	0	1	0	0	1	← associeren
1	1	0	1	0	0	1	← maskeren
1	0	0	1	0	1	0	
0	0	1	0	1	0	0	
0	1	1	0	1	0	0	
1	1	1	0	0	0	1	← maskeren
1	0	1	0	0	0	1	← associeren

fig. 4.5. Waarheidstabel.

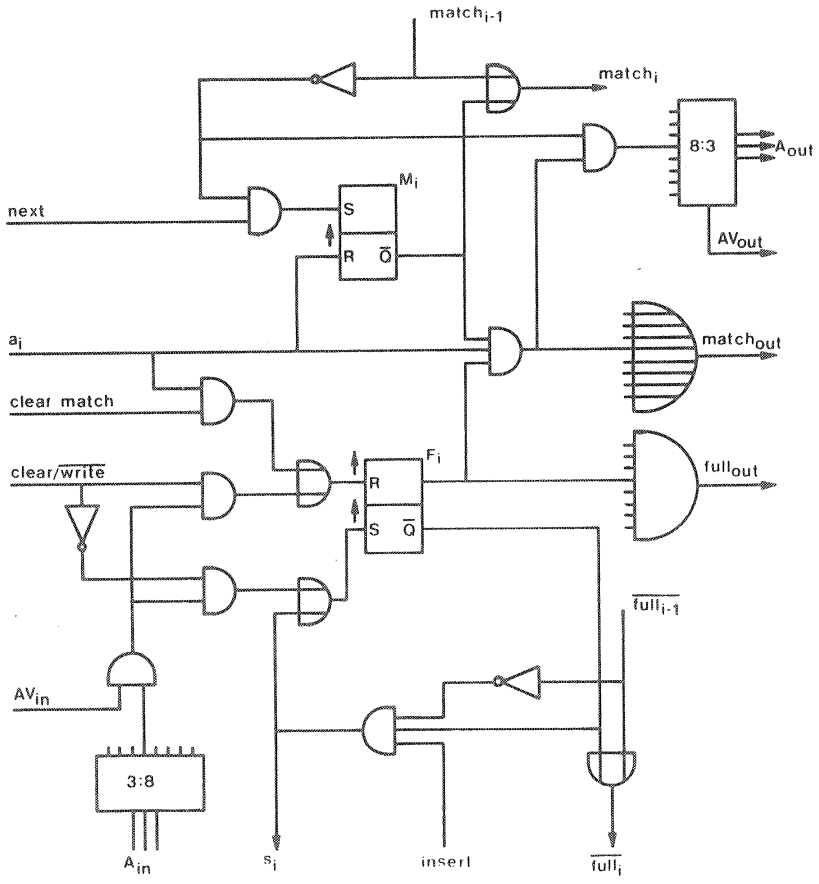


fig.4.6. besturing van één woord.

Indien een woord associeert (a_i=1) en indien dat woord beschreven is (F_i=1) zal dit resulteren in een "match-out" signaal. Als bovendien het i-de woord tevens het enige of, in geval van multiple match, het eerste woord is dat associeert ($match_{i-1}$=0) wordt via een encoder het adres van woord i naar buiten gestuurd (A_{out} & AV_{out}). Dit adres kan dan b.v. gebruikt worden om een RAM te adresseren. M.b.v. het "next"-signaal kan deze associatie worden onderdrukt, immers flip-flop M_i wordt dan op "1" gezet (Q:="0"), waardoor bovendien $match_i$ "0" wordt en waardoor aan een volgend woord dat associeert de gelegenheid wordt gegeven zijn adres naar buiten te sturen. Met behulp van dit "next"-commando kunnen alle associaties sekwentieel worden opgeroepen. Door het tijdelijk weghalen van het kenmerk vervalt de associatie (a_i=0), waardoor alle M_i weer terug op "0" gezet worden ("first"-commando): het adres van het eerste woord dat associeert verschijnt dan weer.

Literatuur.

[1] **Content addressable parallel processors**
C.C. Foster, Van Nostrand Reinhold Company, 1976

[2] **Het realiseren v/e associatief geheugen m.b.v. injectie logica**
R.G.J. Dijkstra
Afstudeerverslag T.H. Eindhoven, vakgroep Digitale Systemen
1974

[3] **Toepassingen van associatieve geheugens**
W.W.M. Scheepens
MC Colloquium, maart 1980

DATAFLOW COMPUTERS

A.H. VEEN
Mathematisch Centrum

1. IN HET BEGIN

In den beginne was er een machine, die bestond uit een geheugen en een Centrale Verwerkings-Eenheid (CVE). De werking was simpel. Het geheugen was passief; daar lagen instrukties en gegevens opgeslagen. De CVE was aktief: hij doorliep een oneindige cyclus. Iedere stap van de cyclus bestond uit het ophalen van een instruktie uit het geheugen en het uitvoeren van de bewerkingen, die door de instruktie werden gespecificeerd. De programma-wijzer bepaalde welke instruktie werd opgehaald. Deze werd na iedere instruktie met één opgehoogd. Er waren ook instrukties, die de programma-wijzer een nieuwe waarde konden geven. Een programma bevatte twee soorten instrukties: instrukties die de programma-wijzer wijzigden (control flow) en instrukties die gegevens in het geheugen wijzigden (data flow).

Dit koncept, dat later naar von Neumann werd genoemd, was eenvoudig en overzichtelijk en geheel toegesneden op de stand van de technologie van die dagen: een eenvoudige CVE kon nog maar net gemaakt worden. In de drie decennia die op de eerste computers volgden, veranderde de situatie drastisch. Complexe hardware werd steeds goedkoper, waardoor, bij het ontwerpen van een CVE, snelheid en kracht belangrjker werden dan eenvoud. Nieuwe toepassingsgebieden dienden zich aan en flexibiliteit werd belangrijk. Programma's werden steeds groter en onoverzichtelijker. Om de koppeling van randapparatuur (en later multi-programmering) te vergemakkelijken deed de interrupt zijn intrede, waardoor de control flow geheel ondoorzichtig werd. Op den duur bleef van de eenvoud van de oorspronkelijke machine weinig meer over.

Zo'n tien jaar geleden deed een nieuw, simpel koncept zijn intrede: dataflow. Een dataflow-programma bestaat alleen uit dataflow-instrukties. Zo'n instruktie specificeert een bewerking op een aantal operanden en bevat de adressen van de instrukties, die de geproduceerde resultaten weer als operand gebruiken. Control flow instrukties zijn er niet meer. De volgorde van uitvoering wordt geheel bepaald door de beschikbaarheid van de operanden. Een instruktie wordt uitgevoerd als

alle benodigde operanden beschikbaar zijn. Het enige effekt van het uitvoeren van een instruktie is de konsumptie van de operanden en de produktie van de resultaat-gegevens. De instrukties zijn funktioneel, dat wil zeggen dat de waarden van de resultaat-gegevens alleen afhangen van de waarden van de operanden. Er zijn geen "onzichtbare" neveneffecten. Een dataflow-computer is een machine, die speciaal gebouwd is voor het efficiënt uitvoeren van dataflow-programma's. Hij heeft geen expliciete programma-wijzer, maar laat zich leiden door de onderlinge afhankelijkheid van gegevens in het programma.

Dit koncept heeft zijn wortels in de zogenaamde dataflow-analyse, een methode voor het analyseren van programma's voor optimaliseringsdoeleinden en semantische verifikatie. Zie [6,13,16] voor vroege referenties.

In het volgende hoofdstuk zal ik nader ingaan op de ontwikkelingen, die tot dataflow-machines en -talen hebben geleid. In hoofdstuk 3 worden de principes van een dataflow-machinetaal behandeld aan de hand van een eenvoudig voorbeeld. In het daaropvolgende hoofdstuk wordt nader ingegaan op een aspekt van de machinetaal, dat bepalend is voor de graad van parallellisme. Hoofdstuk 5 presenteert het ontwerp van een realistische dataflow-machine. De laatste hoofdstukken zijn gewijd aan hogere programmeertalen voor dataflow-computers.

2. MOTIVATIE

Zoals al eerder in dit colloquium [15] is opgemerkt, zijn nieuwe ontwikkelingen op het grensgebied tussen machine-architektuur en hogere programmeertalen veelal het gevolg van impulsen uit zowel de hardware- als de software-hoek. Dataflow vormt hierop geen uitzondering.

2.1. Hardware

De drijvende kracht vanuit de hardware is in dit geval het verlangen om een computer te bouwen, die zijn snelheid haalt uit de parallelle executie van onafhankelijke delen van een programma. Het verband tussen prijs en snelheid van een processor is verre van lineair. De koper van een microprocessor krijgt relatief gezien veel meer rekenkracht voor zijn geld dan de koper van een supersnelle maxiprocessor. Met andere woorden: hoe duurder de processor, hoe ongunstiger de prijs/prestatie-verhouding. Het verschil loopt op tot één of twee ordes van grootte. Het is voor de hand liggend om te zoeken naar een architektuur, waarbij een groot aantal microprocessoren, of speciaal ontworpen LSI-komponenten van een soortgelijke complexiteit, met elkaar verbonden zijn door een eenvoudige kommunikatie-struktuur. Als we een organisatie kunnen vinden, die

garandeert dat op ieder moment een redelijk deel van de processoren bezig is met relevante berekeningen, dan hebben we een machine, die veel sneller is dan een konventionele machine van gelijke prijs.

Voor sommige toepassingen is zoiets vrij gemakkelijk te realiseren. Er zijn relatief goedkope machines op de markt, die een enorme snelheid behalen door gebruik te maken van specifieke eigenschappen van algorithmen in een klein toepassingsgebied. Voorbeelden hiervan zijn Fast Fourier Transform machines of array-processoren, die speciaal zijn ontworpen voor een aantal standaard bewerkingen op vectoren. Deze voorbeelden vallen in de zogenaamde Single Instruction Multiple Data stream (SIMD) klasse van parallelle processoren: dezelfde bewerking wordt uitgevoerd op een aantal verschillende gegevens-stromen. Doordat er maar één instruktie-stroom is hebben we maar één programma-wijzer nodig. Het aantal toepassingen, dat zich voor implementatie op een SIMD-computer leent, is helaas beperkt. Een algemeen bruikbare parallelle processor zal meerdere onafhankelijke instruktie-stromen en dus geen centrale programmawijzer hebben.

Om een parallelle processor met een ruim toepassingsgebied te kunnen realiseren, is het wachten echter op een algemene methode om een tijdrovende berekening in delen te splitsen, die aan onafhankelijke processoren uitgedeeld kunnen worden. Er bestaat al lang behoefte aan zo'n methode, maar aan die behoefte bleek veel moeilijker te voldoen dan aanvankelijk werd verondersteld. Het probleem is dat parallellisme en het von Neumann koncept fundamenteel met elkaar in strijd zijn. Het dataflow koncept is een veelbelovend alternatief.

2.2. Software

In de software-hoek zijn er ontwikkelingen, die in dezelfde richting wijzen. Met de "software crisis" hebben we al zo lang moeten leven, dat we het nauwelijks nog een crisis kunnen noemen. De problemen zijn er sindsdien nauwelijks kleiner op geworden. Vooral grote software-projekten met een lange levensduur zitten in de problemen. De programma's zijn onoverzichtelijk en slecht gestruktureerd, het gedrag is soms onvoorspelbaar, fouten zijn moeilijk op te sporen. Eén van de oplossingen (gestruktureerd programmeren) is een ontwerp- en programmeer-discipline waarmee de meest voorkomende fouten voorkomen kunnen worden. Een programma moet bestaan uit modulen, die zo veel mogelijk onafhankelijk zijn en die elkaar expliciet informatie doorgeven. Het effekt van het uitvoeren van een moduul moet zoveel mogelijk zichtbaar zijn. Onzichtbare neveneffekten zijn gevaarlijk. Om deze redenen zijn onder andere goto's en globale variabelen naar de strafbank verwezen. Gezocht wordt naar talen die het toepassen van deze principes gemakkelijker maken en het

zondigen ertegen bemoeilijken. In een dataflow-taal zijn een aantal van de gesignaleerde problemen uitgebannen. Omdat er geen programma-wijzer is, bestaan er ook geen goto´s. Globale variabelen zijn verdwenen, omdat procedure´s geen neveneffekten mogen hebben. Of in zo´n taal ook nog geprogrammeerd kan worden, valt natuurlijk nog te bezien.

3. MACHINETAAL

We zagen in het eerste hoofdstuk al dat een dataflow-instruktie de adressen bevat van zijn opvolgers, dat wil zeggen van de instrukties die het geproduceerde resultaat als operand gebruiken. We kunnen een dataflow-programma daarom zien als een gerichte graaf: de knopen zijn de instrukties en de kanten zijn éénrichtings-datapaden. Iedere instruktie heeft één of meer invoerpaden en één of meer uitvoerpaden. De gegevens, die over de paden lopen, worden tokens genoemd. De uitvoering van een programma wordt geheel door de Laad-regel en de Executie-regel bepaald. Een instruktie heet geladen als, op alle benodigde invoerpaden een token aanwezig is (Laad-regel). Als een instruktie geladen is dan gaat hij, op een niet gespecificeerd moment, over tot executie, waarbij alle tokens op de invoerpaden verdwijnen en na verloop van tijd een token op alle uitvoerpaden verschijnt (Executie-regel). De waarde van de uitvoertokens is een funktie van die van de invoertokens. De funktie wordt bepaald door het soort instruktie (aangegeven door de zogenaamde opkode).

In dit hoofdstuk worden de principes van een dataflow-machine behandeld. Voor dat doel wordt, bij wijze van voorbeeld, een vereenvoudigde machine met een bijbehorende machinetaal geïntroduceerd. Echte dataflow-machines verschillen aanzienlijk hiervan en van elkaar, zoals we in de volgende hoofdstukken nog zullen zien.

3.1. Een Eenvoudige Graaf

We nemen als voorbeeld de berekening van de wortels van een vierkantsvergelijking

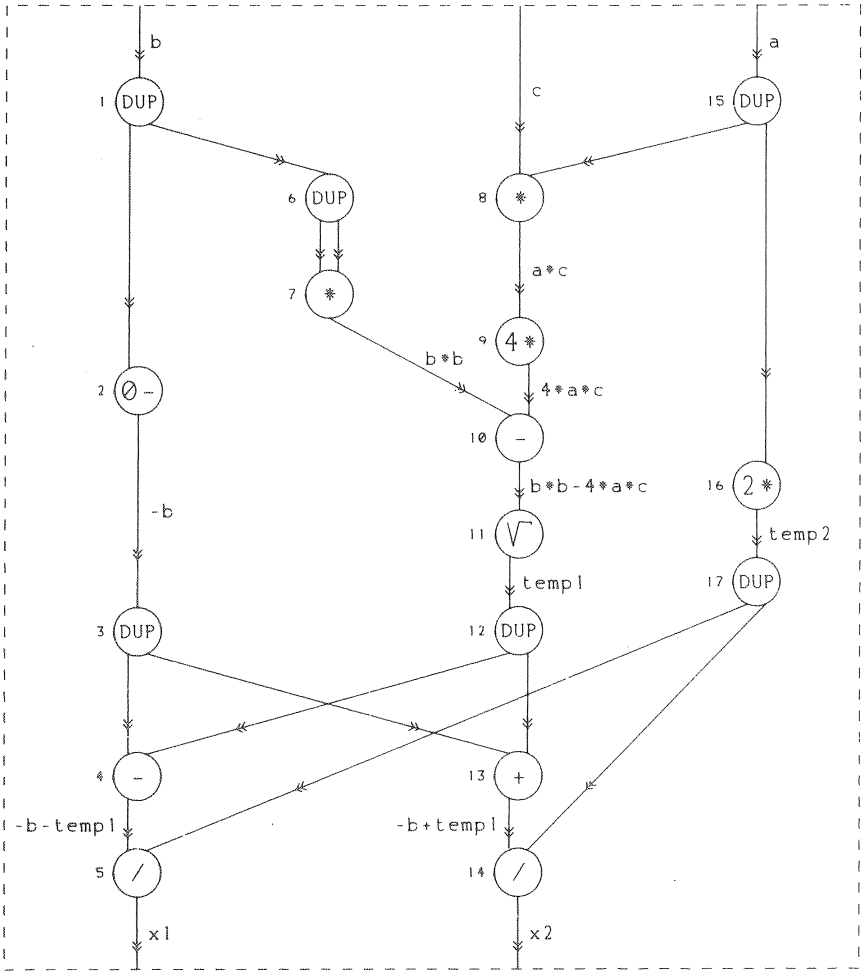
$$ax^2 + bx + c = 0$$

$$\langle x1, x2 \rangle = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In konventionele vorm:

```
temp1 := wortel( b*b - 4*a*c);
temp2 := 2*a;
x1    := (-b + temp1) / temp2;
x2    := (-b - temp1) / temp2;
```

De dataflow-graaf ziet er als volgt uit:



Ik heb hier aangenomen dat alle rekenkundige operaties slechts één uitvoerkanaal hebben. Er zijn daarom speciale dupliceer operaties (DUP) nodig, die een duplikaat van het invoertoken op de beide uitvoerkanalen plaatsen. De benodigde konstanten zijn in de instruktiekode opgenomen. De "4*" instruktie heeft dus maar één invoertoken nodig. Bij de paden zijn namen vermeld om vergelijking met het voorafgaande en het volgende te vergemakkelijken. Deze namen horen niet bij de dataflow-graaf.

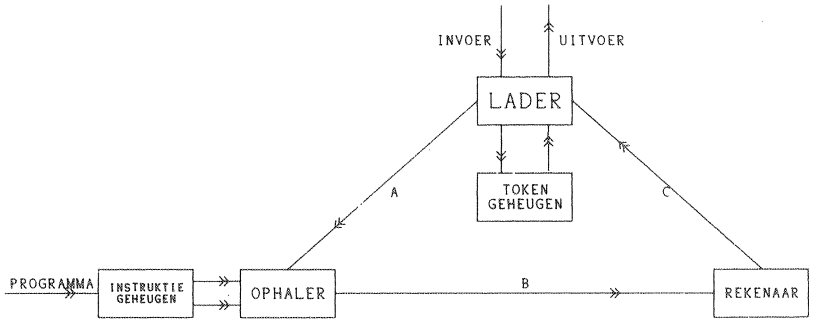
De machinekode voor deze graaf zou er als volgt uit kunnen zien:

ADRES	INSTRUKTIE			
	OPKODE	KONSTANTE	OPVOLGERS	
1	DUP		2/1	6/0
2	MIN	0/0	3/0	
3	DUP		4/0	13/0
4	MIN		5/0	
5	DEEL		UITVOER	
6	DUP		7/0	7/1
7	MAAL		10/0	
8	MAAL		9/1	
9	MAAL	4/0	10/1	
10	MIN		11/0	
11	WORTEL		12/0	
12	DUP		4/1	13/1
13	PLUS		14/0	
14	DEEL		UITVOER	
15	DUP		8/1	16/1
16	MAAL	2/0	17/0	
17	DUP		5/1	14/1

De kodering van de opvolgers is aangegeven als opvolgeradres/invoerkanaal. De invoerkanalen hebben nummers 0 en 1. Als de instruktie een konstante bevat, komt deze in de plaats van een invoerkanaal. Dit is aangegeven in de kolom "konstante" met waarde/invoerkanaal. Het is interessant om te zien dat de volgorde van de instrukties niet van belang is. We kunnen de instrukties willekeurig permuteren (mits we natuurlijk de opvolger-adressen mee muteren) zonder de betekenis van het programma te wijzigen. De uitvoering van het programma wordt geheel door de Laad- en de Executie-regel bepaald.

3.2. Een Vereenvoudigde Machine

In de volgende figuur is schematisch een hypothetische dataflow-machine weergegeven.



Het programma in machinekode wordt in het instruktiegeheugen geladen. Via het invoerkanaal worden gegevens ingebracht in de vorm van token-pakketten. Zo'n pakket bestaat uit een waarde en zijn bestemming. De Lader accepteert zo'n pakket en kijkt of het invoertoken de geadresseerde instruktie in de Laad-toestand brengt. Zo nee, dan wordt het token opge-borgen, zo ja, dan worden het instruktie-adres en de bijbehorende token(s) in de vorm van een invoerpakket naar de Ophaler gestuurd. Deze voegt de invoerwaarden samen met een kopie van de instruktie tot een kommando-pakket. De Rekenaar voert de instruktie uit, die door de opkode wordt gespecificeerd, gebruik makend van de invoerwaarden. Met ieder opvolger-adres wordt een kopie van de uitgerekende waarde gekombineerd tot een tokenpakket en doorgestuurd naar de Lader. Als de Lader de bestemming UITVOER tegenkomt dan wordt het token naar het uitvoerkanaal gestuurd.

Over de verbindingskanalen lopen dus drie soorten pakketten:

A invoerpakket <adres, waarde(n)>
B kommandopakket <opkode, waarde(n), adres(sen)>
C tokenpakket <adres, waarde>

Stel dat we in ons voorbeeld de wortels willen uitrekenen voor de waarden 2.5, 6 en 2 voor respektievelijk a, b en c. De invoer zou bestaan uit de volgende tokenpakketten:

<15/0, 2.5 >
< 1/0, 6.0 >
< 8/0, 2.0 >

Omdat instrukties 1 en 15 maar één operand nodig hebben, zijn zij nu geladen. De Lader stuurt de volgende invoerpakketten naar de Ophaler:

< 15, 2.5 >
< 1, 6.0 >

Instruktie 8 heeft twee operanden nodig en is dus nog niet geladen. Het desbetreffende token wordt in het token-geheugen opgeborgen. De Ophaler stuurt op zijn beurt de volgende kommando-pakketten de baan op:

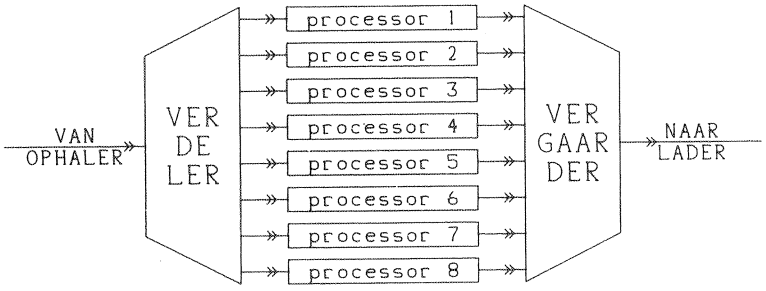
< DUP, 2.5, 8/1, 16/1 >
< DUP, 6.0, 2/1, 6/0 >

De Lader ontvangt vervolgens van de Rekenaar:

< 8/1, 2.5 >
< 16/1, 2.5 >
< 2/1, 6.0 >
< 6/0, 6.0 >

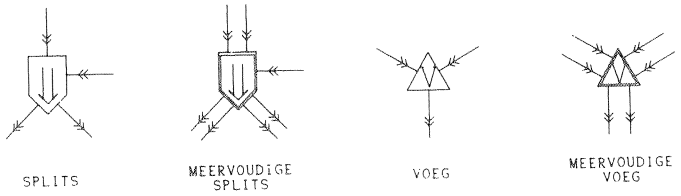
Nu zijn de instrukties 2, 6, 8 en 16 geladen. Deze worden uitgevoerd en brengen op hun beurt weer nieuwe instrukties in de Laad-toestand. Uiteindelijk zal de executie van instrukties 5 en 14 tokenpakketten met bestemming UITVOER opleveren. De hier geschetste executie-volgorde van de instrukties is slechts één van de mogelijkheden. Omdat de instrukties echter funktioneel zijn hebben veranderingen in de volgorde geen invloed op het resultaat. Daarom bevat ieder pakket dat over één van de kanalen A, B of C loopt alle informatie, die nodig is voor een bepaalde aktie. De pakketten hebben geen onderlinge verwijzingen of andere afhankelijkheden. In een korrekt dataflow-programma is de volgorde, waarin de pakketten worden afgehandeld onbelangrijk. De kanalen kunnen daarom van buffers worden voorzien en de drie modulen kunnen asynchroon opereren. Dit heeft de belangrijkste konsekwentie voor de Rekenaar. Omdat alle dataflow-instrukties funktioneel zijn, zijn de geproduceerde tokenpakketten volledig bepaald door het bijbehorende kommandopakket. De Rekenaar kan daarom zonder problemen gerealiseerd worden door middel van een aantal parallelle processoren, die ieder een kommandopakket accepteren, de berekening uitvoeren en een tokenpakket produceren (zie onderstaande

figuur). De Verdeler en de Vergaarder zijn simpele schakelkomponenten die pakketten naar de eerste vrije processor sturen, respektievelijk pakketten van alle processoren accepteren. Omdat de Ophaler en de Lader over een geheugen beschikken, waarvan bij de behandeling van een pakket gebruik wordt gemaakt, is het veel moeilijker om deze modulen ook parallel uit te voeren. Deze vormen dan ook potentiële bottlenecks.



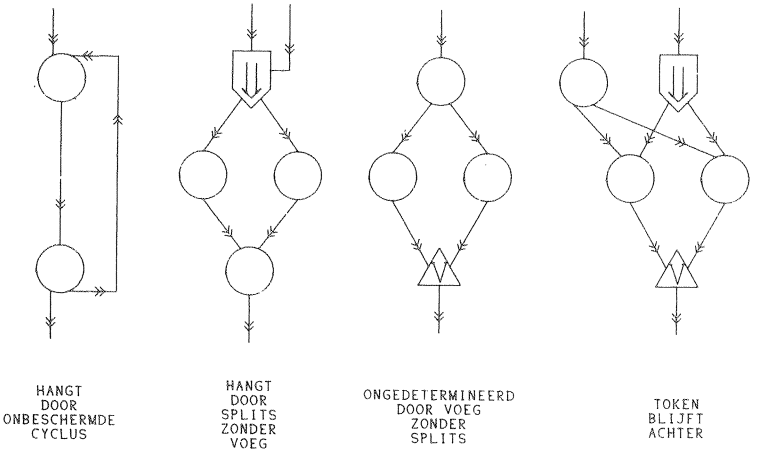
3.3. Overige Instrukties

Met de soort instrukties die we tot nu toe hebben gezien kunnen we alleen zeer primitieve grafen maken, overeenkomend met programma's in een konventionele taal, waarin alleen maar simpele variabelen, toekenning en arithmetische operaties voorkomen. Ieder datapad in de graaf komt overeen met een simpele variabele. Het plaatsen van een token op een uitvoerpad komt overeen met de konventionele toekennings-operatie. Voor interessante programma's, met konditionele expressies en iteraties, hebben we allereerst het ekwivalent van de konditionele sprong nodig. In de volgende figuur zien we de SPLITS operator. Deze stuurt een kopie van zijn invoertoken naar slechts één van zijn uitvoerkanalen. De splitsrichting wordt bepaald door de waarde van het logische token op het rechter invoerkanaal. De executie-regel van SPLITS wijkt af van de gangbare, omdat slechts op één van de twee uitvoerpaden een token verschijnt. De meervoudige SPLITS wordt gebruikt als meerdere datapaden door dezelfde test worden gesplitst. Om de SPLITS te sturen hebben we nu ook operatoren nodig, die logische waarden produceren (relationele operaties) en manipuleren (logische operaties).

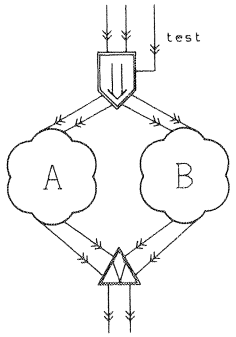


Het komplement van de SPLITS is de VOEG operatie. Deze accepteert een token van zijn linker of rechter invoerpad en plaatst een duplikaat op het uitvoerkanaal. VOEG heeft een afwijkende Laad-regel: de instruktie is geladen zodra alle invoerpaden aan de linkerkant of alle invoerpaden aan de rechterkant een token bevatten. Als zowel tokens op linker als rechter paden binnenkomen, dan is de volgorde van de uitvoertokens niet gedetermineerd.

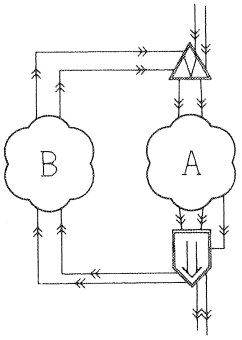
Een procedure P is een graaf die wordt gemarkeerd door de twee instrukties INGANG P en UITGANG. Deze procedure wordt aangeroepen door de AANROEP P instruktie. Als deze instruktie tot executie overgaat, wordt een kopie gemaakt van de met P korresponderende graaf, de invoertokens worden op de invoerpaden van INGANG P geplaatst en de uitvoerpaden van de UITGANG instruktie worden verbonden met de uitvoerpaden van de AANROEP P instruktie. Omdat bij iedere aanroep een nieuwe kopie wordt gemaakt, kunnen de verschillende aanroepen van procedures geen invloed op elkaar hebben. Recursie krijgen we er zodoende gratis bij.



Omdat SPLITS en VOEG een afwijkende Laad-, respektievelijk Executie-regel hebben, moeten we oppassen hoe we deze operatoren in een programma verwerken. Een willekeurige cyclus in een gerichte graaf geeft ook problemen. Bovenstaande figuur laat een aantal narigheden zien. Een veilige manier om SPLITS, VOEG of een cyclus te gebruiken, zien we in de volgende figuur, waar de effekten van het afwijkende gedrag van SPLITS en VOEG elkaar opheffen:

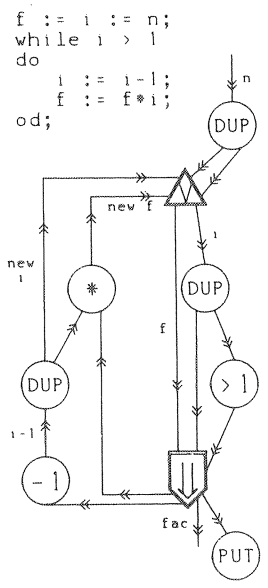


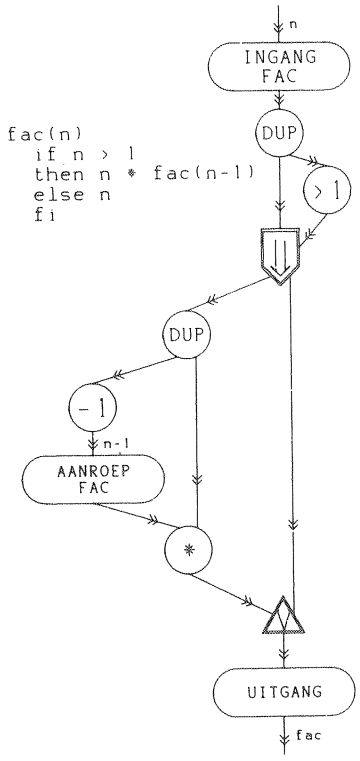
if test then A else B fi



while A do B od

Hieronder zien we als voorbeeld van iteratie de graaf om n! uit te rekenen met daar naast dezelfde berekening in recursieve vorm:





4. KRINGLOOP EN KLEUREN

In het laatste hoofdstuk zagen we twee manieren om een stuk programma meer dan eens te gebruiken: iteratie en recursie. Iteratie en recursie zijn voorbeelden van een kringloop. Een programma zonder kringloop is moeilijk denkbaar: de programmeur zou zijn tijd verdoen met het schrijven van instrukties die slechts één keer worden uitgevoerd. In een dataflow-machine vereist kringloop speciale voorzieningen. Worden er namelijk tokens geïnjecteerd in een subgraaf waar nog tokens in achter gebleven zijn van een vorige ronde van de kringloop, dan zouden de tokens behorende bij verschillende rondes met elkaar verward kunnen worden, met alle gevolgen van dien. De machine uit het vorige hoofdstuk gebruikt twee verschillende voorzieningen. Bij iteratie vormen de meervoudige SPLITS- en de meervoudige VOEG-instruktie een sluis, die ervoor zorgt dat tokens van een nieuwe ronde pas doorgelaten worden als de vorige geheel beëindigd is. Bij recursie wordt een verse kopie van de gehele graaf gemaakt. In het algemeen bestaan er drie methoden om dergelijke interferenties van tokens te vermijden:

SLUIS De subgraaf wordt omsloten door twee poorten en tokens voor een nieuwe ronde worden slechts toegelaten als alle tokens van een vorige ronde zijn verdwenen [23].

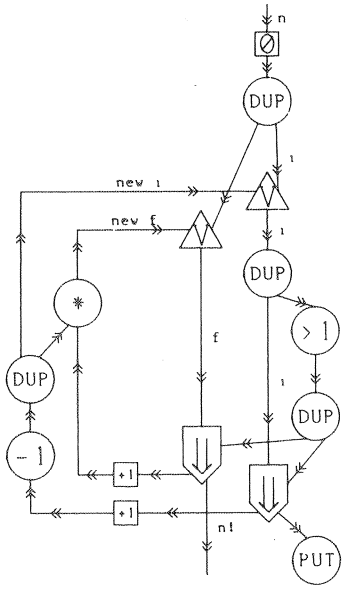
KOPIE Aan het begin van een nieuwe ronde wordt de subgraaf gekopiëerd [20].

KLEUR Iedere ronde krijgt een unieke identifikatie (kleur) en alle tokens die bij één ronde horen worden gemarkeerd met deze kleur. De Laad-regel wordt gewijzigd. Een instruktie is nu geladen als op alle invoerpaden een token met dezelfde kleur aanwezig is [2,7].

De SLUIS oplossing heeft een drastische beperking van het parallellisme tot gevolg: iedere ronde moet wachten op de voltooiing van de vorige. Bij KOPIE en KLEUR kunnen wel verschillende ronden tegelijkertijd aktief zijn. Deze twee laatste oplossingen bieden daarom de beste mogelijkheden om een dataflow-machine optimaal te benutten. KOPIE echter vereist een enorm instruktiegeheugen, omdat voor iedere ronde de volledige subgraaf moet worden gekopiëerd, inclusief de instrukties die niet worden gebruikt. Denk eens aan een grote recursieve procedure met veel konditionele instrukties. Een instruktie zou eigenlijk pas gekopiëerd moeten worden op het moment dat er invoertokens voor arriveren. De administratie die hiervoor nodig is, komt overeen met die voor het KLEUR mechanisme. We kunnen het KLEUR-mechanisme ook zien als een efficiënte implementatie van KOPIE. Bij KOPIE loopt maar één token over een

datapad. Nadat het token gepasseerd is verdwijnt het datapad. Bij KLEUR kunnen over een datapad meerdere tokens lopen, maar nooit meerdere van dezelfde kleur.

Ter illustratie een voorbeeld van de iteratie voor n! met kleuren geïmplementeerd:



De vierkante hokjes zijn operatoren die alleen de kleur van een token wijzigen. De twee operatoren in dit voorbeeld zetten de kleur op 0 of hogen hem op. De meervoudige SPLITS en VOEG zijn door hun enkelvoudige broertjes vervangen, want we hebben geen SLUIS meer nodig. Als nu de vermenigvuldig-operatie veel langer duurt dan alle andere operaties, dan kan de rij tokens op datapad i (met waarden n-1,n-2,...,1) al geproduceerd zijn voordat de eerste vermenigvuldiging is voltooid. Al deze tokens i hebben als bestemming de vermenigvuldig instruktie, maar ze hebben allemaal een verschillende kleur. Pas als weer een f beschikbaar komt, wordt een token i met de juiste kleur opgezocht en naar de vermenigvuldig-instruktie gestuurd. In dit voorbeeld wordt het dure gedeelte van de berekening dus weer sequentieel uitgevoerd, maar dit is te wijten aan het inherent sequentiële karakter van dit algorithme. In

het algorithme:

```
fac(n)
   f(1,n)

f(min,max)
   if (min = max)
      min
   else
      f(min, ((min+max)/2)) * f(((min+max)/2)+1, max)
```

verdeelt f het aantal vermenigvuldigingen steeds in tweeën. Hierdoor kunnen maximaal n ronden tegelijkertijd aktief zijn. Dit illustreert het betreurenswaardige feit, dat het maximaal benutten van de capaciteit van een parallelle machine niet alleen een geëigende taal vereist, maar in vele gevallen ook geëigende algorithmen. In het algemeen liggen in het parallel uitvoeren van kringloopstappen de grootste mogelijkheden voor het benutten van parallellisme.

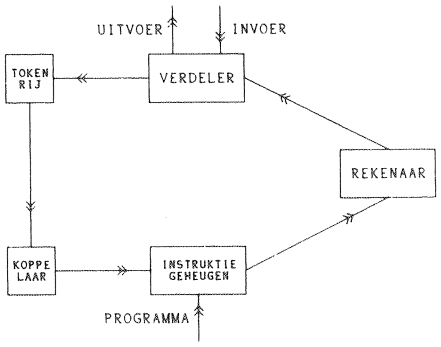
Het kleurmechanisme, zoals in dit voorbeeld is geschetst, is een vergaande vereenvoudiging. Als we rekening willen houden met geneste iteraties, meervoudige recursie en alle kombinaties hiervan dan hebben we een veel complexer schema nodig. In het volgende hoofdstuk zien we daar een voorbeeld van. Door het kleurmechanisme niet geheel onzichtbaar te maken, maar in de machinetaal primitieven op te nemen, die kleuren van tokens kunnen manipuleren, opent zich plotseling een scala van interessante mogelijkheden, waar ik in hoofdstuk 6 nog op terug zal komen.

5. DE MANCHESTER DATAFLOW MACHINE

In de afgelopen vijf jaar zijn er een aantal ontwerpen van dataflow-machines gepubliceerd en zeker twee daarvan zijn inmiddels gerealiseerd. Eén daarvan zal ik hier uitvoerig beschrijven om een indruk te geven van de complexiteit van een realistische machine. Voor de overige machines zie [21,23,5,2].

Zeker vier onderzoeksgroepen in Engeland houden zich op dit moment met dataflow bezig. Op Manchester University wordt een prototype gekonstrueerd, dat in 1981 operationeel moet zijn. Het is de enige machine waarvan een gedetailleerd ontwerp is gepubliceerd [24,10]. De machine werkt met kleuren om een zo groot mogelijk parallellisme uit een kringloop te halen. Iedere nieuwe ronde van een kringloop krijgt een unieke kleur toegewezen, die hem onderscheidt niet alleen van de andere ronden van de kringloop maar ook van andere kringlopen. Arvind & Gostelow [2] hebben een schema voorgesteld, waarbij een instruktie uit de

kleur van een invoertoken een nieuwe unieke kleur kan berekenen zonder van globale informatie gebruik te maken. Helaas groeit volgens dit schema het aantal bits dat nodig is om de kleur te koderen lineair met de recursiediepte. In Manchester is een variatie bedacht, die wel gebruik maakt van globale informatie maar veel minder geheugen gebruikt [11]. De kleur bestaat uit drie velden van elk twaalf bits, één voor een index in een datastruktuur (array van vaste lengte), één voor de teller in een iteratie en één voor een unieke procedure-kleur. De eerste twee kunnen door de instruktie lokaal worden berekend, voor de laatste is een instantie in de machine nodig, die unieke procedure-kleuren uitgeeft. Een iteratie krijgt ook een eigen procedure-kleur om geneste iteraties mogelijk te maken.



Bovenstaande figuur geeft de architektuur van de machine weer. De invoergegevens komen binnen in de Verdeler als een tokenpakket (95 bits), bestaande uit kleur (36), bestemming (22) en waarde (37). De Tokenrij is een FIFO buffer met plaats voor 16K van deze pakketten. De Koppelaar is het interessantste gedeelte van deze machine. Hij houdt de Laad-regel bij en moet dus voor ieder binnenkomend token bepalen of de bestemmingsinstruktie nu geladen is en in dat geval de eventueel bijbehorende invoertokens bijvoegen. Een vereenvoudiging is, dat instrukties met meer dan twee invoerpaden al op hoger niveau zijn gesplitst in eenvoudiger instrukties, zodat in de machinetaal alleen operaties met één of twee invoerpaden bestaan. Het aantal invoerpaden van een instruktie wordt aangegeven door een bit in het bestemmingsveld. "Alleenstaande" tokens worden direkt doorgestuurd. Voor de overige tokens wordt gekeken of de partner al aanwezig is. Zo ja, dan wordt deze opgehaald en samen met de andere doorgestuurd naar het instruktie-geheugen. Als de partner nog niet aanwezig is, dan wordt het token opgeborgen. Voor dit opzoeken simuleert de Koppelaar een associatief geheugen door middel van hardware hashtabellen. De zoek-sleutel wordt gevormd door 54 bits van de bestemming en

de kleur. De Koppelaar kan 16K tokens herbergen. Het instruktie-geheugen is gesegmenteerd en biedt plaats aan 8-16K instrukties (68 of 34 bits). De Rekenaar bestaat uit twee trappen. De eerste trap voert de instrukties uit die globale informatie gebruiken ("Genereer Procedure Kleur"). De tweede trap is opgebouwd uit 15 elementen. Ieder element is een gemicroprogrammeerde bit-slice processor.

De machine is niet vrij van deadlock. De Koppelaar kan namelijk vol raken. Iedere instruktie konsumeert één of twee tokens en produceert één of twee tokens. Instrukties kunnen daarom in drie klassen worden verdeeld: expanderend, inkrimpend of neutraal, al naar gelang ze het aantal in omloop zijnde tokens vermeerderen, verminderen of gelijk laten. Wanneer er nu tijdens een bepaalde fase van de uitvoering van een programma veel expanderende instrukties worden uitgevoerd (bijvoorbeeld door het simultaan starten van vele ronden van een uitgebreide kringloop), dan groeit het aantal tokens in het systeem en de Koppelaar en de Tokenrij raken steeds voller. Als deze beiden tot hun limiet zijn gevuld, zal de Rekenaar zijn tokens niet meer kwijt kunnen en komt de machine abrupt tot stilstand. Dit zal niet snel gebeuren, omdat naast de 32K tokens die in de Tokenrij en de Koppelaar terecht kunnen, ook nog tokens naar een overstroom-bak kunnen afvloeien. Hoe groot deze bak is, wordt niet vermeld, maar het is wel duidelijk, dat de snelheid van de Koppelaar evenredig daalt met het aantal overgestroomde tokens. De volgende modifikatie op het ontwerp zou een aanzienlijke verlichting kunnen brengen. In ieder token-pakket wordt een extra bit opgenomen om aan te kunnen geven of de opvolger-instruktie expanderend, inkrimpend of neutraal is. De Tokenrij wordt gewijzigd, zodat hij de tokens nu niet meer in de oorspronkelijke volgorde aan de Koppelaar doorgeeft, maar alle expanderende of inkrimpende of neutrale tokens vóór laat gaan. De keuze wordt bepaald aan de hand van signalen, die de aktiviteit van de andere delen van de machine aangeven. Bij een onderbezette Rekenaar krijgen expanderende tokens voorrang, terwijl als de overstroombak tokens bevat bij voorkeur inkrimpende tokens worden doorgestuurd.

Als de machine optimaal werkt dan wordt iedere 200-300 nsec een instruktie uitgevoerd. Het geheugen, dat in de diverse onderdelen wordt gebruikt (zo'n 460 Kbyte) heeft een snelheid van 100 nsec. De cyclustijd van het microprogramma is 200 nsec. Het zijn dus snelle komponenten die worden gebruikt (bipolair). De maximum snelheid komt toch niet boven de vijf miljoen instrukties per sekonde (5 MIPS) uit. Een DEC VAX-11 heeft bijvoorbeeld een snelheid van ongeveer 1 MIPS. Hierbij moeten we nog bedenken dat, onder andere vanwege de kleuren-administratie, het aantal machine-instrukties in een dataflow-programma groter is dan in een ekwivalent konventioneel programma. Het probleem is, dat belangrijke

onderdelen en vooral de Koppelaar niet eenvoudig parallel zijn uit te voeren.

Het hier beschreven prototype is echter slechts een deel van de uiteindelijk te bouwen machine. Deze gelaagde machine zal bestaan uit een aantal ringen, die ieder vrijwel hetzelfde zijn als het prototype zoals dat nu wordt gebouwd. Alleen de Verdeler is anders. De Verdelers van alle ringen tesamen vormen een schakelnetwerk, waardoor tokens tussen de verschillende ringen kunnen worden uitgewisseld. De route van de tokens door het netwerk wordt bepaald door het bestemmingsveld. Het verkeer door het netwerk, en dus de belasting van dit centrale onderdeel van de machine, wordt dan bepaald door de verdeling van de instrukties over de afzonderlijke ringen. Nog veel onderzoek is nodig om tot een goede strategie voor deze verdeling te komen. Er is overigens weinig bekend over deze gelaagde machine.

6. FUNKTIONALITEIT EN STROMEN

Voor ik de mogelijkheden voor hogere programmeertalen voor dataflow computers behandel, wil ik eerst wat dieper ingaan op een centraal begrip in deze kontext: funktionaliteit. Alle operatoren in de machinetaal in hoofdstuk 3 vertonen een funktioneel gedrag, dat wil zeggen dat de waarden van de uitvoertokens geheel bepaald worden door de waarden van de invoertokens. Dit geldt ook voor procedure-aanroepen: de uitvoertokens van een AANROEP instruktie zijn geheel bepaald door de waarden van de invoertokens. Alle procedures zijn dus funkties en alle informatie die een procedure nodig heeft moet in de invoertokens besloten liggen. De procedure heeft geen toegang tot globale informatie en hij heeft ook geen intern geheugen waardoor informatie van één aanroep aan een andere aanroep kan worden doorgegeven. Dit is tegelijkertijd de kracht en de zwakte van dataflow-talen. Omdat aanroepen elkaar niet ondergronds kunnen beïnvloeden is de onderlinge afhankelijkheid van procedure-aanroepen direkt uit de dataflow-graaf af te lezen: De Laad-regel en de Executieregel zijn voldoende om een deterministisch gedrag te verzekeren. De strikte eis van funktionaliteit vormt echter ook een grote handicap, want niet-funktionele procedures zijn bij het programmeren vaak erg handig en soms zelfs onmisbaar.

Een voorbeeld van het eerste is een procedure NAMEN in een vertaler die de namentabel beheert. Verspreid door de vertaler komen twee soorten aanroepen voor: NAMEN(SCHRIJF, NAAM) waardoor met naam een adres wordt geassocieerd, dat in de tabel wordt opgeslagen en NAMEN(LEES, NAAM) die als waarde het met NAAM geassocieerde adres oplevert. De SCHRIJF aanroep heeft een neveneffekt waardoor de LEES aanroep niet meer funktioneel is, maar van voorgaande aanroepen van NAMEN afhangt. De in dataflow-talen

meest gebruikelijke oplossing is om de waarde van het interne geheugen (de tabel) als extra in- en uitvoerwaarden mee te geven. De aanroepen zijn dan van de vorm:

```
<dummy,TABEL52> := NAMEN(SCHRIJF, NAAM, TABEL51);
<adres,dummy  > := NAMEN(LEES   , NAAM, TABEL52);
```

Door iedere SCHRIJF opdracht wordt dus een nieuwe tabel gecreëerd. Voor een complex programma als een vertaler wordt dit al gauw ondoenlijk, omdat de tabel bij iedere wijziging gekopieerd moet worden. De programma's worden onoverzichtelijk, doordat de programmeur voor iedere wijziging een nieuwe naam voor de tabel moet bedenken. Het druist ook in tegen modulariteits-principes: we willen juist het beheer over de namen-tabel aan een aparte procedure overlaten en we willen dus zeker geen interne informatie exporteren.

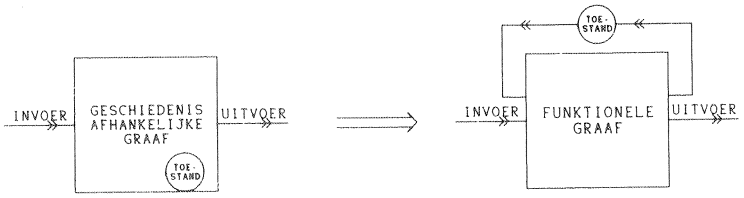
Hetzelfde probleem komen we op grotere schaal tegen bij het kommuniceren met een centraal gegevensbestand. De methode van het doorgeven van de oude en nieuwe waarde van het intern geheugen bij iedere aanroep wordt hier geheel onbruikbaar. Het is immers juist één van de voordelen van een centrale bestandsbeheerder dat de programma's, die van het bestand gebruik maken, niet van elkaar op de hoogte hoeven te zijn.

We kunnen deze kommunikatie met een gegevensbestand zien als een vorm van in- en uitvoer, waarbij in het algemeen het probleem zich voordoet, dat de wereld buiten het dataflow-programma niet funktioneel is en de kommunikatie ermee zorgvuldig moet worden gesynchroniseerd. Een goede oplossing voor het in- en uitvoerprobleem kan daarom model staan voor een algemene oplossing voor komponenten in een dataflow-programma met een niet-funktioneel gedrag.

Laten we eerst twee oorzaken van niet-funktioneel gedrag onderscheiden: niet-determinisme en geschiedenis-afhankelijkheid. Een teksteditor, die gebruikt wordt om een nieuwe tekst te creëren (het modificeren van een bestaande tekst laten we hier even buiten beschouwing), vertoont een niet-funktioneel gedrag: De invoer opdracht "TYP LAATSTE REGEL" leidt niet altijd tot dezelfde uitvoer, maar is afhankelijk van voorgaande invoer. Een goede editor is echter wel konsistent: als we later dezelfde reeks opdrachten invoeren, verwachten we hetzelfde resultaat. Op dit niveau is de editor dus wel funktioneel. Meerdere gebruikers kunnen er dan ook simultaan mee werken als de verschillende reeksen opdrachten maar uit elkaar worden gehouden. Zo'n programma noemen we deterministisch en geschiedenis-afhankelijk. Een echte random-generator is ook niet funktioneel: "TYP RANDOM-GETAL" geeft meestal niet dezelfde uitvoer. De uitvoer wordt echter niet door voorafgaande invoer bepaald,

maar door andere, buiten ons model vallende, faktoren. Zo´n programma heet niet-deterministisch.

Terug naar de dataflow-graaf. Van geschiedenis-afhankelijke grafen willen we kunnen zeggen op welk niveau zo´n graaf funktioneel is. Dit bepaalt namelijk welke tokens onafhankelijk van elkaar en dus parallel kunnen worden verwerkt (vergelijk de opdrachten van verschillende gebruikers van een editor) en welke op elkaar zullen moeten wachten. Deze afhankelijkheid kunnen we beschrijven met behulp van zogenaamde stromen. Een stroom is een geordende reeks tokens, die over hetzelfde datapad loopt. Een deterministische geschiedenis-afhankelijke graaf heeft een intern geheugen, d.w.z. de graaf bevat een cyclus met een token dat niet bij iedere aanroep dezelfde initiële waarde heeft. Het gedrag is ekwivalent met dat van een funktionele graaf waaraan we de waarde van het geheugen (de toestand) als extra in- en uitvoer meegeven (externe cyclus).



De toestand (het rondcirkelende token) wordt geïnitialiseerd door een speciaal invoer-token, dat het Stroom-Scheidings-Token (SST) wordt genoemd. Een stroom wordt omsloten door SST´s. Een deterministische graaf is funktioneel op het niveau van een stroom: bij dezelfde invoerstroom hoort altijd dezelfde uitvoer. Tokens die tot verschillende stromen behoren, kunnen onafhankelijk van elkaar (en dus parallel) behandeld worden. We moeten dan wel zorgen, dat we met iedere stroom een apart intern geheugen associeren. Om het parallellisme niet onnodig te beperken willen we ook nog graag, dat de tokens, die tot één stroom behoren, alleen dan sequentieel worden verwerkt als het strikt noodzakelijk is (dat wil zeggen binnen een geschiedenis-afhankelijke graaf). Onafhankelijke bewerkingen op de individuele tokens moeten parallel worden uitgevoerd.

Een uitgebreid KLEUR-mechanisme kan aan deze tegenstrijdige eisen voldoen. Zoals we in hoofdstuk 4 zagen, zorgen de kleuren voor maximaal parallellisme bij kringlopen. Door nu in de machinetaal operaties op kleuren toe te voegen kan de scheidslijn tussen parallel en sequentieel worden beïnvloed. Introduktie van een token met een nieuwe kleur in een

graaf is ekwivalent met het kopiëren van die graaf. Twee tokens met verschillende kleur kunnen immers niet met elkaar gekoppeld worden, dus kunnen ze elkaar ook niet beïnvloeden. Dit geldt natuurlijk alleen als de graaf geen veranderingen in de kleur aanbrengt. Als de tokens van een stroom van de geschiedenis-afhankelijkheid van een graaf gebruik moeten maken, dan zullen ze allemaal dezelfde instantie van die graaf moeten doorlopen. Met andere woorden: ze moeten allemaal dezelfde kleur krijgen, maar ze moeten toch onderscheiden worden van de tokens van een andere stroom. Dit wordt op de volgende manier bewerkstelligd. Een deel van de kleur heeft de funktie van rangnummer-veld. Opeenvolgende tokens van een stroom hebben opeenvolgende rangnummers en overigens dezelfde kleur. De geschiedenis-afhankelijke graaf wordt omsloten door een stroomsluis bestaande uit een STROOM-IN en een STROOM-UIT instruktie. De STROOM-IN instruktie heeft tot taak de binnenkomende tokens in volgorde te accepteren en ze, ontdaan van het rangnummerveld, aan de graaf door te geven. De tokens van een stroom hebben dus binnen de graaf een identieke kleur. De STROOM-UIT instruktie accepteert de resultaat-tokens uit de graaf en vult het rangnummer-veld weer in. De STROOM-IN en STROOM-UIT instrukties zijn gesynchroniseerd, dat wil zeggen STROOM-IN levert het volgende token pas af als STROOM-UIT het resultaat van het voorgaande token heeft verwerkt. Hierdoor is deterministisch gedrag verzekerd.

7. HOGERE PROGRAMMEERTALEN

De machinetaal uit hoofdstuk 3 is natuurlijk niet erg geschikt om in te programmeren. De instrukties zijn van een veel te laag niveau en de mogelijkheden voor foute konstrukties zijn legio (zie 3.3). Hetzelfde geldt voor de kleuren manipulatie mechanismen, die in het vorige hoofdstuk zijn beschreven. We hebben een hogere programmeertaal nodig, waarin het "goed" programmeren is en die vrij eenvoudig naar een dataflow-machinetaal te vertalen is. Uit het oogpunt van kontinuïteit en gebruikersgemak zou het de voorkeur verdienen voor deze taal een bestaande hogere programmeertaal te kiezen en een vertaler te schrijven die programma's in die taal geschreven, accepteert en machine-kode voor een dataflow-computer aflevert. Daarvoor moet de vertaler echter (statisch) de volledige data-afhankelijkheid van de statements in een programma vaststellen. Voor de meeste talen (speciaal die met konditionele sprongen) is dit een ingewikkelde procedure en noodzakelijkerwijze zal de vertaler hier en daar een benadering moeten maken. Om determinisme te verzekeren zal dit altijd een pessimistische schatting moeten zijn en het parallellisme in het vertaalde programma zal dan ook verre van optimaal zijn.

Vandaar dat in eerste instantie een aantal nieuwe talen zijn

ontstaan die speciaal voor dataflow-machines zijn gemaakt. Behalve de problemen, die we al van het ontwerpen van konventionele talen kennen, dienen zich, bij het ontwerpen van deze dataflow-talen, een aantal nieuwe problemen aan:

- Door de afwezigheid van een centraal geheugen heeft de vertaalde kode geen toegang tot globale informatie.

- Alle dataflow-instrukties zijn funktioneel, de buitenwereld (achtergrond-geheugen, de gebruiker van een interaktief programma) is dat niet. Alle operaties op deze buitenwereld (in- en uitvoer) moeten daarom zorgvuldig worden gesynchroniseerd.

- Om de graad van parallellisme zo hoog mogelijk te houden, moet de taal de juiste konstrukties bevatten. Als een (abstrakt) algorithme wordt omgezet in een programma in deze taal dan mag er geen parallellisme verloren gaan.

- We zijn gewend om bij het programmeren sequentieel te denken. De dataflow-taal moet liefst zo zijn, dat het (mentaal) sequentieel interpreteren van een programma de betekenis niet aantast.

Zeker vijf hogere programmeertalen zijn in de afgelopen jaren speciaal voor dataflow-machines ontworpen. Drie daarvan zal ik in dit hoofdstuk behandelen. Zie voor de overige talen [25,22]. Al deze talen hebben een aantal belangrijke eigenschappen gemeen:

- Het zijn zogenaamde Talen met Eenmalige Toekenning (TET´s): aan een variabele kan maar één maal in een programma een waarde worden toegekend. Van variabel en toekennen is daarom nauwelijks nog sprake en men gebruikt dan ook wel de woorden waarde en definitie. De reden voor de eenmalige toekenning is eenvoudig: in een dataflow-machine beschikken we niet over geheugenplaatsen waar naar believen waarden in geschreven of uit gelezen kunnen worden. We hebben alleen maar te maken met datapaden waar, afgezien van kringloop, slechts één waarde over loopt. De paden kunnen we namen geven. Toekenning (definitie) korrespondeert dan met het plaatsen van een token op een uitvoerpad en referentie korrespondeert met het konsumeren van een token van het invoerpad. Meerdere referenties aan dezelfde naam leiden tot dupliceer (DUP) knopen.

- Om statisch op eenvoudige wijze te kunnen bepalen of aan de regel van eenmalige toekenning is voldaan zijn konditionele statements niet toegestaan. Bij konditionele expressies moet de else tak altijd aanwezig zijn. De definities van meer dan één waarde kunnen

door dezelfde konditie worden gestuurd als in:

<min,max> := if a > b then <b,a> else <a,b> fi

Dit heeft het nadeel dat een waarde en zijn definiërende expressie ver uit elkaar kunnen komen te staan (om van het komma's tellen nog maar niet te spreken).

- Omdat procedures funktioneel zijn hebben zij geen inwendig geheugen, dat gebruikt kan worden om informatie van een aanroep naar een andere door te geven. Evenmin zijn er globale variabelen. De invoerwaarden en de resultaten zijn strikt gescheiden. Alle parameters zijn call-by-value.

- Een programma bestaat uit een reeks definities (toekenningen), waardoor een naam aan de waarde van een expressie wordt gebonden. Expressies worden, voor zover mogelijk, gelijktijdig geëvalueerd. Alleen als in een expressie aan een naam wordt gerefereerd, wordt zonodig gewacht tot de bijbehorende expressie is geëvalueerd. Wat parallelle executie betreft, is hier dus de situatie precies omgekeerd in vergelijking met konventionele talen. Een konventioneel programma wordt geheel sequentieel uitgevoerd, tenzij parallelle executie door speciale primitieven wordt aangegeven. Bij dataflow-talen wordt juist alles parallel uitgevoerd tenzij speciale omstandigheden dat verhinderen.

Ik zal de twee voorbeelden uit hoofdstuk 3 in de drie talen weergeven om een idee te geven hoe een programma er uitziet.

7.1. LAPSE

Voor de in hoofdstuk 5 beschreven machine is onlangs op de Manchester University een hogere programmeertaal ontworpen, die de naam LAPSE heeft gekregen [8]. De syntax is op PASCAL gebaseerd.

Het stuk programma, dat de wortels van een vierkantsvergelijking oplost, ziet er, in LAPSE, als volgt uit:

```
decl temp1, temp2: real;
 ...
x1 := (-b + temp1) / temp2;
x2 := (-b - temp1) / temp2;
temp1 := wortel( b*b - 4*a*c );
temp2 := 2*a;
```

Dit ziet er dus vrijwel hetzelfde uit als in normaal PASCAL, alleen is de volgorde van de statements niet van belang: x1 en x2 worden pas uitgerekend als temp1 en temp2 zijn berekend.

De elementaire datatypen zijn boolean, integer en real. De toegestane datastrukturen zijn (net als in PASCAL) record en array (met vaste grenzen).

Alle funkties zijn op hetzelfde niveau gedeklareerd (geen geneste procedure-deklaraties). Funkties mogen elkaar willekeurig aanroepen en recursie is zonder beperking toegestaan. Het kleur-mechanisme zorgt ervoor dat parameters onafhankelijk van elkaar worden doorgegeven, zodat de uitvoering van een deel van een funktie al kan beginnen, voordat alle parameters beschikbaar zijn (asynchrone parameter konsumptie). Ook de resultaten worden asynchroon opgeleverd.

Het is typerend voor dataflow, dat iteratie meer problemen oplevert dan recursie. De iteratie-konstruktie heeft dan ook de vorm van een funktie gekregen en is geïmplementeerd als een efficiënte vorm van staartrecursie :

```
iteration fit(i, f: integer);
repeat
    i := i - 1;
    f := f * i;
until i < 1 end

function fac(n: integer): integer;
decl dummy: integer;
begin
    [fac, dummy] := fit(n, n);
end
```

Een iteratie heeft dus een naam en de aanroep intialiseert de iteratievariabelen. Het nul maal doorlopen van een iteratie is onmogelijk.

Aan elementen van een array kan op de normale wijze worden gerefereerd (als in B := AR[i]). Omdat een dataflow-machine geen adresseerbaar gegevens-geheugen heeft, wordt de array-referentie gesimuleerd door een kopie van het gehele array AR te maken en vervolgens alle elementen behalve i weg te gooien. Aan een individueel element kan niet worden toegekend. Als dit was toegestaan, zou statisch niet meer bepaald kunnen worden of aan de regel van Eenmalige Toekenning is voldaan. De waarden van een array worden altijd gelijktijdig toegekend in een forall konstruktie. Bijvoorbeeld

```
forall i in [1..10] do AB[i] := A[i] + B[i] od
```

creëert een nieuw array AB met grenzen 1 en 10. Alle doorgangen van een forall worden parallel geëvalueerd. Om redenen van implementatie kan binnen een forall geen funktie-aanroep voorkomen.

Voor de in- en uitvoer bestaan zeer eenvoudige lees- en schrijfoperaties. Deze kunnen niet binnen een iteratie of een funktie worden gebruikt. De programmeur moet zelf voor de synchronisatie zorgen, door middel van extra parameters. In

```
[s2,B] := read(file, s1);
[s1,A] := read(file, true );
```

wordt eerst A en dan B ingelezen. De dummy-variabele s1 zorgt voor de synchronisatie. We zien hier overigens hetzelfde probleem en dezelfde oplossing als bij de procedure NAMEN uit hoofdstuk 6.

7.2. Value-Oriented Algorithmic Language

Op het MIT Laboratory for Computer Science is de dataflow-taal VAL ontwikkeld [1]. De syntax is gebaseerd op CLU [18,17].

De wortels van de vierkantsvergelijking worden opgeleverd door de volgende expressie:

```
let  temp1, temp2: real ;
     temp1 := wortel( b*b - 4*a*c );
     temp2 := 2*a;
in
     (-b + temp1) / temp2,
     (-b - temp1) / temp2
endlet
```

Deze let konstruktie introduceert een nieuw blok, waarbinnen namen kunnen worden ge(her)definiëerd. De let konstruktie is zelf ook een expressie die dezelfde waarde(n) heeft als de expressie(s) tussen in en endlet. Deze konstruktie kan dus genest worden.

De elementaire datatypen zijn boolean, integer, real en character. Deze kunnen gekombineerd worden tot records, arrays (met vaste grenzen) en unions.

Funkties kunnen genest gedeklareerd worden. Anders dan bij een let konstruktie zijn de aktuele parameters de enige waarden waar een funktie toegang toe heeft. Om onduidelijke redenen is recursie niet toegestaan.

De iteratie konstruktie ziet er als volgt uit:

```
for fac, f: integer := n, n;
do
      if f > 1
      then iter f := f-1; fac := fac * n; enditer
      else fac
      endif
endfor
```

In de eerste regel worden de iteratie variabelen gedeklareerd en tegelijkertijd geïnitialiseerd. Als de iter-enditer tak is uitgevoerd wordt het do-endfor gedeelte weer gestart.

Aan elementen van arrays kan op normale wijze worden gerefereerd. Toekenning aan een arrayelement kan alleen plaatsvinden door een nieuw array te creëren. Bijvoorbeeld

```
B := A[5 : k]
```

maakt een nieuw array B, dat gelijk is aan A met het vijfde element vervangen door k. Arrays kunnen ook worden gemaakt met de forall konstruktie als in

```
AB := forall i in [1,10] do
      construct A[i] * B[i]
      endall
```

waar een array AB met grenzen 1 en 10 wordt gecreëerd. In

```
sum := forall i in [1,10] do
       eval plus A[i] * B[i]
       endall
```

wordt het inprodukt van arrays A en B als waarde opgeleverd. In plaats van plus kunnen ook andere operaties gebruikt, mits zij zowel associatief als kommutatief zijn. De bedoeling van deze konstruktie is om aan te geven dat iteratie-ronden onafhankelijk zijn en dus parallel kunnen worden uitgevoerd. De eval konstruktie kan de berekende resultaten asynchroon verwerken (dankzij de speciale eigenschappen van de operator).

Als tijdens de executie van een dataflow-programma een fout wordt gekonstateerd, kan niet eenvoudigweg de executie worden onderbroken en een foutboodschap worden verstuurd. Andere berekeningen kunnen immers nog in volle gang zijn. In VAL is een volledige kollektie foutwaarden gedefiniëerd, alsook het gedrag van alle operatoren die zo'n waarde als

invoer krijgen. De foutwaarde zal uiteindelijk in de uitvoer verschijnen. Er bestaat een voorstel om aan een foutwaarde een spoor van alle bezochte knopen te hangen, zodat de oorsprong van de fout gemakkelijker is op te sporen [19].

Er zijn geen in- en uitvoer-primitieven gedefiniëerd. Dit wordt wel als een noodzakelijke maar gecompliceerde uitbreiding aangekondigd.

7.3. Irvine Dataflow

De dataflow-groep op de University of California in Irvine heeft voor de machine die hen voor ogen staat de taal ID ontworpen [3]. Er bestaat een vertaler voor, die machinetaal aflevert die vooralsnog wordt gebruikt als invoer voor een simulator. Er zijn ook uitgebreide simulatie statistieken gepubliceerd [9].

De wortels van de vierkantsvergelijking worden opgeleverd door:

```
( temp1 <- wortel(b↑2 - 4*a*c);
  temp2 <- 2*a;
  return (-b + temp1) / temp2, (-b - temp1) / temp2 )
```

Deze konstruktie (omsloten door haakjes) heet een blok expressie. Blokken mogen worden genest. Binnen een blok betekent een toekenning een (her)definitie van de naam in de linkerkant. ID kent geen deklaraties.

Tot de elementaire datatypen behoren integer, real, boolean, string en error. Procedure is ook een datatype met als belangrijkste operatie AANROEP. De tekst van een procedure wordt ingevoerd als konstante van dit type. Hierdoor kan overal op de plaats van een expressie ook een procedure definitie staan en kunnen procedures als parameters worden doorgegeven. Een proceduretekst kan een naam hebben om recursie mogelijk te maken. Alle invoerwaarden van een procedure moeten beschikbaar zijn voordat de funktie tot executie over kan gaan (synchrone parameter konsumptie). In ID zijn nieuwe operator-definities toegestaan en de taal heeft ook een abstrakt data type mechanisme, vergelijkbaar met de klassen uit SIMULA [4].

De iteratie voor n! wordt in ID een blokexpressie

```
(  initial fac <- n; f<- n;
   while f > 1
   do new fac <- fac*n;
      new f <- f-1;
   return fac )
```

Alle iteratie-variabelen (variabelen waarmee informatie van één ronde naar de volgende kan worden doorgegeven) moeten in het initial gedeelte worden geïnitialiseerd. Binnen de iteratie is de waarde van de vorige ronde beschikbaar (fac) alsook de nieuwe waarde (new fac).

Ook bestaat er de

for i from 1 to n do

konstruktie.

ID kent een datastruktuur stroom. De elementaire operaties die hierop zijn gedefinieerd zijn dusdanig, dat de elementen van een stroom alleen opeenvolgend kunnen worden verwerkt. Voor stromen is een uitzondering op de regel van synchrone parameter konsumptie gemaakt: de elementen van een stroom worden tijdens executie van een funktie gekonsumeerd. Hierdoor kunnen op eenvoudige wijze modules worden gecreëerd die met elkaar kommuniceren door het sturen van boodschappen (vergelijk de Communicating Sequential Processes van Hoare [12]).

Een interessante konstruktie in ID is verder de beheerder (resource manager). Dit is een speciaal soort procedure, die apart wordt gecreëerd en gebruikt. Anders dus dan bij de normale procedure, waar immers iedere aanroep tot een aparte instantie van de procedure leidt. Ieder gebruik van een beheerder richt zich op dezelfde instantie. De ingang van een beheerder is een niet-deterministische voeg-operator, die de tokens, die van verschillende paden binnenkomen, tot één stroom samenvoegt. De volgorde van de tokens in deze stroom is niet gedetermineerd. Het in- en uitvoer probleem is hiermee elegant op te lossen. Een gegevensbestandbeheerder en een beheerder voor randapparatuur is gemakkelijk te realiseren.

8. EEN HYBRIDE TAAL

De TET's (talen voor eenmalige toekenning) uit het vorige hoofdstuk zijn alle drie speciaal voor een dataflow-machine ontworpen. De afstand tot konventionele talen is aanzienlijk, hoewel de gelijkenis in syntax soms anders doet vermoeden. Een programmeur, die gewend is aan het programmeren in een konventionele taal, zal zich een heel andere programmeer-stijl eigen moeten maken. Dit is een voordeel voor zover het gaat om het afleren van slechte gewoonten. Maar niet alle mogelijkheden die konventionele talen bieden, maar TET's missen, zijn verderfelijk. In dit hoofdstuk beschrijf ik de principe's van een dataflow-taal, die zo dicht mogelijk aansluit bij een konventionele taal (zie ook [26]). De bedoeling hiervan is, onder andere, om te onderscheiden welke van de

eigenschappen van dataflow-talen essentiëel zijn en welke niet. Het ideaal is, dat programma's normaal in een (nette) konventionele taal geschreven kunnen worden, die dan door een vertaler dermate slim naar dataflow-machinekode vertaald worden, dat de graad van parallellisme even hoog is als wanneer een TET was gebruikt. In de rest van dit hoofdstuk zal ik laten zien, dat een taal die sterk op een konventionele taal lijkt, ook geschikt is als brontaal voor een dataflow-machine. Dat zal ik doen, door bij iedere konstruktie in de taal, die tot problemen kan leiden aan te geven hoe een vertaler een programma in zo'n taal om kan zetten naar een ekwivalent programma in een TET.

Ik ga, bij wijze van voorbeeld, uit van een op het Mathematisch Centrum ontworpen en geïmplementeerde taal SUMMER [14]. De hybride taal zou, evenals SUMMER, in het bijzonder geschikt moeten zijn voor het schrijven van string-verwerkende programma's, zoals vertalers, tekstverwerkers en editor's. Anders dan bij numerieke toepassingen, waar dataflow-machines en -talen grotendeels op zijn gericht, lijkt op dit toepassingsgebied geschiedenis-afhankelijkheid van procedures een grote rol te spelen en tegelijkertijd stroom een voor de hand liggend data-type te zijn. SUMMER kent geen goto's en geen pointers, maar wel globale variabelen. De taal heeft ook abstrakte data-typen, waardoor klasseobjekten kunnen worden gecreëerd.

We stellen allereerst vast, dat het verbod op konditionele statements overbodig is. Ieder konditioneel statement kan via een eenvoudig algorithme worden omgezet in een toekenning met een konditionele expressie aan de rechterkant. Aan de linkerkant van de nieuwe toekenning komt dan een reeks variabelen te staan, die de vereniging vormt van de variabelen, waaraan in de twee takken van het oorspronkelijke statement wordt toegekend. In de takken van de konditionele expressie komen de expressies van de oorspronkelijke toekenningen te staan of de variabelen zelf. Bijvoorbeeld:

```
if k > 0
then
     a := k*3;
     b := 0;
else
     b := 1;
     c := k*3;
fi
```

wordt

```
<a,b,c> :=
if k > 0
then <k*3,0,c>
else <a,1,k*3>
fi
```

De regel van Eenmalige Toekenning is ook overbodig. Bij het vertalen wordt voor iedere variabele bij een toekenning een nieuwe unieke naam gegenereerd. Referenties aan deze variabele in latere statements worden ook door deze nieuwe naam vervangen. Iedere keer dat aan een variabele opnieuw wordt toegekend, wordt weer een nieuwe naam gegenereerd. Als aan een variabele wordt gerefereerd waar nog geen unieke naam voor is gegenereerd dan is het programma inkorrekt.

```
open := 16;
brug := 83.5 + open;
open := brug * open;
verlies := open / 2;
```

wordt

```
var1 := 16;
var2 := 83.5 + var1;
var3 := var2 * var1;
var4 := var3 / 2;
```

Iteraties geven niet meer problemen dan in de gebruikelijke dataflow-talen. Alleen iteratie-variabelen vereisen een speciale behandeling. Dit zijn variabelen waaraan, binnen een iteratie-blok, eerst wordt gerefereerd (oude waarde) en later wordt toegekend (nieuwe waarde). Voor iedere variabele wordt de laatst gegenereerde nieuwe waarde op de gebruikelijke wijze (via KLEUR of SLUIS-mechanisme) als oude waarde de volgende ronde ingestuurd.

De grootste problemen komen we tegen bij kommunikatie met niet-funktionele elementen. In een TET blijft dat beperkt tot kommunikatie met de wereld buiten de dataflow-machine (in- en uitvoer), maar in een konventionele taal doen dit soort problemen zich bovendien voor bij bijvoorbeeld toekenning aan een array-element of aanroepen aan procedures die toegang hebben tot globale informatie. Welke vrijheid we in onze taal toelaten wordt bepaald door een afweging van drie faktoren: het gemak van de programmeur, de eenvoud van de vertaler en de graad van parallellisme in de gegenereerde machine-kode.

We nemen als voorbeeld weer de namentabel uit hoofdstuk 6. Een

stukje SUMMER programma, waarin van een namentabel gebruik wordt gemaakt, ziet er als volgt uit:

```
namen[var1] := nieuw_adres;
adres2      := namen[var2];
namen[var3] := volgend_adres;
```

De waarde tussen [en] heet de sleutel. De vertaler kan voor de synchronisatie van deze toegangen tot de tabel zorgen door voor iedere toegang zogenaamde synchronisatie-signalen te genereren. Bovenstaand programma wordt dan omgezet in:

```
s5          := namen.schrijf( var1, nieuw_adres, s4);
<s6,adres2> := namen.lees   ( var2, s5);
s7          := namen.schrijf( var3, volgend_adres, s6);
```

Hier zorgen dus de signalen s5 en s6 ervoor dat de toegang tot de tabel volledig wordt gesequentialiseerd. Dit is hetzelfde procedé dat de programmeur in LAPSE moet volgen om de in- en uitvoer te synchroniseren.

Als we het gemak van de programmeur voorop stellen dan laten we toe dat de tabel als globaal wordt gedeklareerd, zodat iedere procedure er toegang toe heeft. In elke procedure, die van de tabel gebruik maakt wordt dan een extra signaal toegevoegd aan zowel de parameterlijst als de resultatenlijst. Dit heeft tot gevolg dat de signalen alle toegangen tot de tabel aaneenrijgen tot één lineaire keten. We kunnen alle globale variabelen in één grote tabel onderbrengen en daar hetzelfde sequentialisatie-algorithme voor gebruiken. Dit heeft als konsekwentie dat alle gebruik van globale variabelen in het gehele programma volledig wordt gesequentialiseerd. In een programma waar vrijelijk van globale variabelen gebruik wordt gemaakt blijft op die manier weinig parallellisme over.

Als behoud van parallellisme belangrijker is dan eenvoud van de vertaler, kunnen we beter voor iedere globale variabele een aparte signalenketen maken. Dan wordt aan de parameter- en aan de resulaten-lijst van iedere procedure een hele rij signalen toegevoegd; één signaal voor iedere globale variabele, die in de procedure of in een (direkt of indirekt) aangeroepen procedure wordt gebruikt. Als de parameters van procedures asynchroon worden gekonsumeerd, wordt door deze methode een maximaal parallellisme verkregen ten koste van een extra belasting van vertaler en machine.

Een andere benadering is om de vrijheid van de programmeur enigszins te beperken. In de taal wordt een speciale konstruktie opgenomen, die

een objekt creëert en tegelijkertijd als globale variabele beschikbaar stelt. Alle statements die (dynamisch) binnen deze konstruktie vallen, hebben toegang tot dit objekt. Bovenstaand voorbeeld wordt dan:

```
install (tabel)
     schrijf(var1, nieuw_adres);
     adres2 := lees(var2);
     schrijf(var3, volgend_adres);
endinstall
```

Deze konstruktie kan worden genest. Belangrijk is dat bij install de tabel wordt gecreëerd en bij endinstall weer wordt vernietigd. Dit heeft als voordeel, dat statements die binnen afzonderlijke installs vallen, niet met elkaar hoeven te worden gesynchroniseerd. Dit vermindert de complexiteit van de vertaler en verhoogt de graad van parallellisme in de machinekode.

Een interessante mogelijkheid doet zich hierbij voor. We kunnen speciale "tabellen met eenmalige toekenning" definiëren. Dit zijn tabellen waaraan nooit twee schrijf-opdrachten met dezelfde sleutel mogen worden gegeven. Lees- en schrijf-opdrachten aan zo'n tabel hoeven dan niet meer expliciet met elkaar te worden gesynchroniseerd. Als zo'n tabel een lees-opdracht ontvangt met een sleutel waarvoor nog geen schrijf-opdracht is ontvangen, dan wordt het resultaat pas teruggestuurd als die schrijf-opdracht binnenkomt. Bij vernietiging van het objekt door de endinstall krijgt iedere nog wachtende lees-opdracht de waarde "ongedefinieerd". Op deze manier worden multi-pass vertalers overbodig: de tabel zorgt zelf voor de juiste volgorde van afhandeling.

9. KONKLUSIES

Dataflow-machines zijn een veelbelovend alternatief voor snelle computers die volgens het von Neumann principe werken. In de machinekode van dataflow-computers is bij iedere instruktie alle noodzakelijke informatie over de data-afhankelijkheid voor handen. In zulke machines kunnen daarom vele processoren parallel instrukties van één programma uitvoeren zonder dat hiervoor, tijdens executie, een ingewikkelde analyse nodig is. Er zijn een aantal hogere programmeertalen ontwikkeld, die speciaal als brontaal voor dergelijke machines zijn bedoeld. In het algemeen leggen deze talen de programmeur een aantal ongewenste beperkingen op. Een enigszins gewijzigde versie van een konventionele taal lijkt echter ook geschikt om als brontaal voor een dataflow-machine te dienen.

LITERATUUR

[1] Ackerman, W. and Jack B. Dennis, "VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual," Technical Report 218, MIT/Laboratory for Computer Science (June 1979).

[2] Arvind, and Kim P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Information Processing 77, pp.849-853, North Holland (1977).

[3] Arvind, Kim P. Gostelow, and Wil Plouffe , "An Asynchronous Programming Language and Computing Machine," Technical Report #114a, University of California, Irvine, Information and Computer Science Dept (Dec 1978).

[4] Birtwistle, G. M. and others, SIMULA begin, Auerbach Publications, Philadelphia (1973).

[5] Davis, A. L., "A Data Flow Evaluation Sytem Based on the Concept of Recursive Locality," Proceedings National Computing Conference, pp.1079-1086, AFIP (Jun 1979).

[6] Dennis, J. B., J. B. Fosseen, and J. P. Linderman, "Data Flow Schemas," International Symposium on Theoretical Programming Vol. 5, pp.187-216 (1972).

[7] Dennis, J. B., "First Version of a Data Flow Procedure Language," Programming Symposium Vol. 19, pp.362-376, Springer (1974).

[8] Glauert, J. R. W., "A Single Assignment Language for Data Flow Computing," Dissertation, Dept. of Computer Science - Victoria University of Manchester (Jan 1978).

[9] Gostelow, Kim P. and Robert E. Thomas, "Performance of a Data Flow Computer," Technical Report #127a, University of California, Irvine, Information and Computer Science Dept (Oct 1979).

[10] Gurd, John and Ian Watson, "A Data Driven System for High Speed Parallel Computing," Report, Dept. of Computer Science - Victoria University of Manchester.

[11] Gurd, John, Ian Watson, and John Glauert, "A Multilayered Data Flow Architecture," Internal Report, Dept. of Computer Science - Victoria University of Manchester (July 1978). Second issue

[12] Hoare, C. A. R., "Communicating Sequential Processes," CACM Vol. 21(8), pp.666-677 (Aug 1978).

[13] Karp, R. M. and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," SIAM Journal of Applied Mathematics Vol. 14 (Nov 1966).

[14] Klint, Paul, "An Overview of the SUMMER Programming Language," Conference Record of the Seventh Annual ACM Symposium of Programming Languages, pp.47-55, ACM (Jan 1980).

[15] Klint, Paul, "De Symbiose van Programmeertaal en Computer," in deze syllabus, Mathematisch Centrum (Feb 1980).

[16] Kosinski, P. R., "A Data Flow Language for Operating Systems Programming," Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting SIGPLAN Notices Vol. 8(9), pp.89-94 (Sep 1973).

[17] Liskov, B. H. and others, "Abstraction Mechanisms in CLU," CACM Vol. 20(8), pp.564-576 (Aug 1977).

[18] Liskov, B. H. and others, "CLU Reference Manual," Computation Structures Group Memo 161, MIT/Laboratory for Computer Science, Cambridge, Mass. (Jul 1978).

[19] McGraw, James R., "Data Flow Computing - The VAL Language," Computation Structures Group Memo 188, MIT/Laboratory for Computer Science (Jan 1980).

[20] Miranker, Glen Seth, "Implementation of Procedures on a Class of Data Flow Processors," Proceedings of the 1977 International Conference on Parallel Processing, pp.77-86.

[21] Misunas, D. P., "A Computer Architecture for Data Flow Computation," Technical Memorandum 100, MIT/Laboratory for Computer Science (1978).

[22] Plas, A. and others, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment," Proceedings of the 1976 International Conference on Parallel Processing, pp.293-302, IEEE (Aug 1976).

[23] Rumbaugh, J., "A Data Flow Multiprocessor," IEEE Trans. Comp. Vol. C-26(2), pp.138-146 (Feb 1977).

[24] Watson, Ian and John Gurd, "A Prototype Data Flow Computer with Token Labelling," Proceedings National Computing Conference Vol. 48, pp.623-628, AFIPS (Jun 1979).

[25] Weng, K. S., "Stream-Oriented Computation in Recursive Data Flow Schemas," Technical Memorandum 68, MIT/Project Mac (1975).

[26] Whitelock, P. J., "A Conventional Language for Data Flow Computing," dissertation, Dept. of Computer Science - Victoria University of Manchester (Oct 1978).

LISP MACHINES

B. WIELINGA
Universiteit van Amsterdam

1. INLEIDING

LISP heeft zich sinds het baanbrekend werk van McCarthy (1960) ontwikkeld tot een van de belangrijkste programmeertalen voor symbolische informatieverwerking. Met name voor het onderzoek op het gebied van de Artificiële Intelligentie (AI) is LISP een sine qua non geworden. Er zijn verschillende redenen aan te wijzen waarom LISP zo populair is in AI kringen. Ten eerste is LISP lange tijd de enige programmeertaal geweest waarin op eenvoudige manier symbolische informatie verwerkt kon worden. Een tweede eigenschap van LISP die bijgedragen heeft tot de grote populariteit is de uniforme representatie van data en programma's in de vorm van lijststrukturen. In combinatie met het feit dat LISP systemen georganiseerd zijn rond een interpretator, en dus interaktief werken met LISP mogelijk maken, levert deze uniforme informatie representatie mogelijkheden om LISP zelf als taal te gebruiken voor editing, debugging, (semi-) automatische programmakorrektie etc.

Hoewel LISP meestal beschouwd wordt als een hogere programmeertaal zou LISP ook gezien kunnen worden als een machinetaal voor een lijstverwerkende machine. De LISP datastrukturen zijn primitief, maar wel machtig genoeg om zeer komplexe strukturen te konstrueren. LISP kent nauwelijks konstrukties die gestruktureerd programmeren mogelijk maken (zoals besturingsstrukturen als WHILE, UNTIL, blokstruktuur, typing) en is ook als zodanig eerder te beschouwen als een machinetaal dan als een (moderne) hogere programmeertaal. Het feit dat LISP in bepaalde opzichten een primitieve taal is heeft echter een voordeel dat zeker bijgedragen heeft tot het sukses van LISP in de AI gemeenschap: het is zeer eenvoudig om LISP uit te breiden met faciliteiten die een specifieke toepassing ondersteunen. Zo is het bijvoorbeeld mogelijk om LISP uit te rusten met een "patroon-gericht invokatie mechanisme", waardoor procedures niet bij naam worden aangeroepen, maar op basis van een patroon (lijststruktuur) dat een globale specificatie van hun werking bevat. Op deze wijze ontstaan nieuwe programmeertalen (zoals PLANNER, CONNIVER, KRL) die het LISP-systeem als basis hebben, maar die mogelijkheden hebben

die in weinig andere programmeertalen terug te vinden zijn. Het gebruik van deze "embedded languages" (Sandewall, 1978) is zeer algemeen in de LISP gebruikers gemeenschap en is een belangrijk hulpmiddel bij de konstruktie van complexe AI programma's.

Een van de belangrijkste technische problemen waarmee de AI gemeenschap tegenwoordig gekonfronteerd wordt is de toenemende complexiteit en grootte van de AI programma's. Het gevolg hiervan is dat de adresseerruimte van de huidige computers (bv. 256k, 36 bits in het geval van de in AI onderzoek veel gebruikte DEC-10) niet meer toereikend is. Een middelgroot AI programma zoals MYCIN (Davis et. al. 1977) omvat al snel 200k (36 bits) geheugenwoorden, terwijl enkele programma's (SOPHIE, Brown et. al. 1974) meer dan 500k woorden in gebruik nemen, met als gevolg dat partitie van het programma in verschillende segmenten nodig is, met alle problemen van dien. Verwacht mag worden dat in de toekomst de behoefte aan een grote adresseerruimte nog aanmerkelijk zal toenemen wannneer AI programma's grotere kennisbestanden zullen gaan bevatten en meer en meer zullen worden uitgerust met natuurlijke taal in- en uitvoer. Een bijkomend probleem is dat de normale time-sharing omgeving die op de meeste computersystemen geboden wordt, niet is afgestemd op het interaktief verwerken van programma's met een dergelijk (virtueel) geheugengebruik.

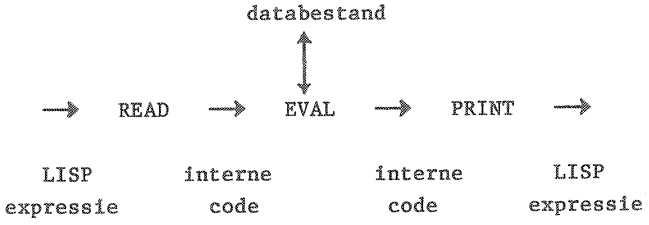
Het gevolg van een en ander is dat binnen de AI gemeenschap verscheidene projekten gaande zijn die gericht zijn op het ontwikkelen van hard- en software die de groeiende behoeften van het AI onderzoek t.a.v. LISP kunnen vervullen. Enerzijds worden technologische hulpmiddelen ontwikkeld die de LISP gebruiker in staat stellen zeer complexe programma's te ontwikkelen, te overzien, te testen en het gedrag ervan te bestuderen. Anderzijds worden computersystemen ontwikkeld die zeer grote LISP programma's efficiënt kunnen verwerken. In paragraaf 2 worden een aantal aspekten van LISP systemen besproken in het licht van een steeds toenemende integratie van programmeertaal en computersysteem. In paragraaf 3 zullen een aantal ontwikkelingen op het gebied van hardware en microprogrammering aan de orde komen. Enkele ontwikkelingen op het gebied van multi-processor systemen die georienteerd zijn op het verwerken van symbolische informatie komen in paragraaf 4 ter sprake. Opgemerkt dient nog te worden dat het begrip "computer-architektuur", in de kontext van dit hoofdstuk vrij ruim geïnterpreteerd moet worden: zowel de organisatie van software systemen als de (gedeeltelijke) realisatie daarvan in hardware, zullen in relatie tot de programmeertaal LISP en afgeleide talen in de discussie betrokken worden.

2. ARCHITECTUUR VAN HET LISP SYSTEEM

2.1. De struktuur van een typisch LISP systeem

Vrijwel alle LISP systemen zijn interpretatief: een LISP expressie wordt ingelezen (omgezet in een interne representatie), vervolgens geEVALueerd, en het resultaat van de evaluatie (een LISP objekt) wordt weer omgezet naar een externe representatie. Deze cyclus (de READ-EVAL-PRINT loop) is de basis van alle interaktie tussen gebruiker en het LISP systeem. Hoewel de meeste LISP systemen beschikken over compilers, is het ongebruikelijk om LISP programma's te compileren voordat ze volledig getest zijn.

Objekten die tijdens evaluatie van een LISP expressie worden gekreëerd (lijststrukturen, procedures) kunnen worden opgeslagen in een databestand dat toegankelijk is via cellen waaraan een unieke naam is toegekend (atomen). LISP programma's worden incrementeel ingevoerd als een verzameling losse funkties die aan het databestand worden toegevoegd. Uitvoering van een programma geschiedt eenvoudigweg door aanroep van een funktie op top-niveau. De architektuur van het LISP systeem is dus een direkt uitvoerende, althans gezien vanuit de virtuele LISP machine (vgl. Klint, 1980) en is schematisch weergegeven in figuur 1.



Figuur 1. Basis architektuur van een LISP systeem.

Aangezien de interne code die door de interpretator uitgevoerd wordt altijd omgezet kan worden naar voor de gebruiker begrijpelijke LISP expressies is het mogelijk de gedragingen van de interpretator op de voet te volgen en dus LISP funkties interaktief te debuggen. Het wijzigen van gedefinieerde strukturen of funkties kan met standaard LISP operaties geschieden. Editors, pretty-printers en debugging pakketten kunnen dan ook makkelijk in LISP zelf geschreven worden.

Veel komponenten van het LISP systeem zijn "programmeerbaar" (Sandewall, 1978), d.w.z. een gebruiker kan procedures definiëren, die onder bepaalde kondities door onderdelen van het systeem -zoals parser, interpretator, editor of (pretty-) printprocedures,- worden uitgevoerd. Uitzonderingskondities, zoals ongedefinieerde variabelen of procedures, illegale operaties op datastrukturen, kunnen door de gebruiker onderschept worden en via speciale procedures afgehandeld worden. Deze faciliteiten maken de konstruktie van interpretatoren voor de eerder genoemde "embedded languages" mogelijk.

De architektuur van het LISP systeem - met name de interaktieve interpretator en de uniforme representatie van data en programma's in een dynamisch databestand - maakt dat een LISP gebruiker vrijwel alle operaties die hij een computersysteem wil laten uitvoeren in LISP kan formuleren en gemakkelijk het conceptueel niveau waarop het systeem toegesproken wordt kan verhogen, waardoor de complexiteit van de kommunikatie met het systeem als geheel aanmerkelijk gereduceerd wordt.

2.2. Interlisp als LISP machine

De integratie tussen programmeertaal en de virtuele machine die executie van programma's in die taal ondersteunt is in sommige LISP systemen dermate ver doorgevoerd, dat we eerder kunnen spreken van een virtuele LISP machine waarmee de gebruiker kommuniceert, dan van een programmeertaal waarin de gebruiker programma's schrijft. Een van de systemen waarin deze integratie het verst is doorgevoerd is Interlisp (Teitelman, 1975). De Interlisp gebruiker kommuniceert gedurende een interaktieve sessie uitsluitend met het Interlisp systeem. Zowel de bibliotheek van standaard procedures als het gebruikersprogramma en de daarbij behorende data zijn permanent onderdeel van het systeem en worden gemanipuleerd door een editor, pretty-printer, interpretator en compiler die permanent in het systeem aanwezig zijn. Na een sessie kan het gehele systeem "gered" worden en in een volgende sessie kan de interaktie voorgezet worden. Opslag van informatie in files is slechts nodig als "back-up" en wordt soms gebruikt om volledige "garbage collectie" mogelijk te maken. Het Interlisp systeem wordt wel een volledig "residentieel" systeem genoemd (Sandewall, 1978): alle informatie en programmatuur waarover de gebruiker moet beschikken is onderdeel van het systeem zelf. Deze opzet opent de mogelijkheid om de gebruiker tijdens interaktie met het systeem van verschillende soorten informatie te voorzien, en om het LISP systeem zelf van verschillende informatiebronnen gebruik te laten maken en aldus de gebruiker bij te staan gewenste operaties zo optimaal mogelijk uit te voeren en eventueel fouten te korrigeren. In Interlisp wordt deze funktie vervult door een "Programmer's Assistant" (Teitelman, 1972, 1977) (afgekort P.A.), die de

verschillende taken die een gebruiker wil uitvoeren koördineert. De gebruiker kommuniceert dus niet meer rechtstreeks met de LISP interpretator maar met de P.A. Deze kommunikatie verloopt via een geavanceerd beeldscherm, waarop verschillende soorten informatie tegelijkertijd getoond kunnen worden in gescheiden "windows". Zo kan er naast een window dat een transcript bevat van de interaktie met de LISP interpretator, een window zijn dat een ("gepretty-printe") tekst van een in ontwikkeling zijnde procedure toont, een window dat een "back-trace" geeft van een evaluatie van een LISP expressie, en verscheidene windows met besturingsinformatie voor de P.A. zelf, de editor etc. Met behulp van een "muis" kan de gebruiker een cursor over het scherm bewegen en door elementen van een menu aan te wijzen de P.A. besturen. Naast de administratie van de windows op het beeldscherm, verzorgt de P.A. een aantal ondersteunende aktiviteiten, zoals (semi-) automatische korrektie van spellingsfouten ("DWIM", Do What I Mean, Teitelman, 1972), uitvoerige explicatie van foutmeldingen van het LISP systeem, waarbij gebruik gemaakt wordt van alle informatie die het systeem bezit over de context waarin de fout gedetekteerd is, en een "historie faciliteit", die de gebruiker de mogelijkheid geeft een historie van zijn interaktie met het systeem te tonen, geselekteerde kommando´s te herhalen of te niet te doen.

Een belangrijke eigenschap van de Interlisp P.A. is dat de verschillende taken zoals editing, debugging, gelijktijdig kunnen verlopen, in die zin dat de P.A. verschillende kontexten onderhoudt voor de verschillende processen, waartussen de gebruiker met een handgreep kan omschakelen. Zo kan bijvoorbeeld nadat een fout in een komplex programma is gekonstateerd het proces van wijzigen van het programma en het testen van de nieuwe versie afwisselend geschieden zonder dat de oorspronkelijke berekening steeds geheel opnieuw hoeft te gebeuren. Het executieproces wordt slechts onderbroken om later weer gekontinueerd te worden.

Het zal duidelijk zijn dat een geavanceerd systeem als de Interlisp P.A. een aanzienlijke informatieverwerkende capaciteit vereist. Naast het grote geheugengebruik dat gepaard gaat met het onderhouden van kontexten voor meerdere processen (mogelijk gemaakt door de "spaghetti-stack" faciliteit in Interlisp), vragen de diverse aktiviteiten van de P.A. vaak meer processortijd dan de executie van het gebruikersprogramma zelf. Het gevolg is dat traditionele multi-user systemen niet meer adekwaat zijn: een DEC-10 kan slechts enkele Interlisp gebruikers ondersteunen, waarbij dan nog vaak ernstige problemen optreden m.b.t. de adresseerruimte. De oplossing voor deze problemen wordt enerzijds gezocht in het gebruik van "personal computers", die geschikt zijn voor het uitvoeren van LISP programma´s (zie paragraaf 3), en anderzijds in multi-processor systemen, waarin de verschillende aktiviteiten van de

P.A. gedistribueerd zijn over een aantal processoren. Problemen hierbij zijn echter de complexe interaktie tussen de verschillende komponenenten van het P.A. systeem, en het feit dat van een gemeenschappelijk databestand gebruik gemaakt wordt (zie verder paragraaf 4).

3. LISP MACHINES

3.1. LISP en de von Neumann machine

Hoewel de gebruiker van een LISP systeem gekonfronteerd wordt met een virtuele machine die min of meer direkt expressies geformuleerd in LISP uitvoert, is er een grote afstand tussen deze virtuele machine en de physieke machines waarop LISP systemen zijn geïmplementeerd. Een belangrijk verschil tussen de architektuur van de computers gebaseerd op de von Neumann machine en de architektuur van een LISP systeem ligt in de struktuur van het geheugen. Terwijl de von Neumann machine een geheugen heeft van sequentieel genummerde cellen, is de struktuur van het LISP geheugen niet sequentieel maar is opgebouwd uit boomstrukturen. In de meeste implementaties van LISP worden deze boomstrukturen opgebouwd uit cellen die uit twee elementen (de CAR en CDR) bestaan, en die een vertakking in de boom representeren. Zo kan de lijst:

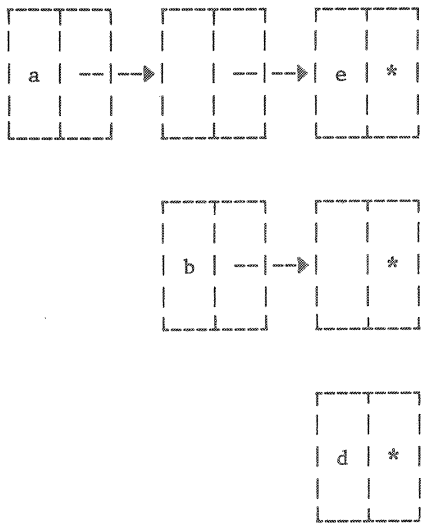
(a (b (d)) e)

gerepresenteerd worden door de verzameling cellen in Figuur 2. Letters in de cellen representeren wijzers naar de "bladen" van de boom: atomen. Het symbool "*" verwijst naar de lege lijst: NIL. Verder bevatten de cellen verwijzingen naar andere cellen. In deze representatie wordt dus geen gebruik gemaakt van de "natuurlijke" opeenvolging van geheugencellen in de von Neumann machine, zoals dat bijvoorbeeld in de representatie van een Algol array wel het geval kan zijn. De reden voor de keuze van een representatie van dit type, die niet optimaal is, is gelegen in het feit dat delen van boomstrukturen vervangen kunnen worden door nieuwe strukturen die niet noodzakelijkerwijze het zelfde aantal cellen in beslag nemen.

Een tweede probleem met verwerking van LISP op een computer van het von Neumann type, is dat cellen dynamisch gekreëerd kunnen worden en dat niet voorspelbaar is hoeveel cellen gedurende een berekening nodig zijn. Dit vereist een dynamisch geheugenallocatie mechanisme aangevuld met een "garbage collector" die er voor zorgt dat cellen die niet meer gebruikt worden (m.a.w. er zijn geen verwijzingen meer naar een cel) vrijgegeven worden voor hernieuwde allocatie. Garbage collectie in grote adresruimten is een zeer tijdrovende zaak, aangezien het gehele geheugen meerdere

malen doorzocht moet worden om de vrije cellen te vinden.

Een derde probleem met de architektuur van de von Neumann machine is er een dat LISP deelt met andere programmeertalen waarin gegeneraliseerde besturingsstrukturen voorkomen (bv. Simula-67). De meeste LISP systemen hebben tegenwoordig faciliteiten om meerdere processen tegelijkertijd aktief te laten zijn. Implementatie hiervan op stapelgeorienteerde computers is echter een moeizame en inefficiënte zaak: er zijn mechanismen nodig voor het onderhouden van een gefragmenteerde stapel, voor het wisselen van kontext en eventueel voor het uitvoeren van garbage collectie op de stapel. De architektuur van de meeste tegenwoordige computers laat een efficiënte implementatie van deze mechanismen niet toe.



Figuur 2. Representatie van een LISP lijst.

De grote hoeveelheid van niet sequentieel gelocaliseerde geheugencellen die in normale LISP programma's geadresseerd worden, geeft nog additionele inefficiënties in systemen die gebruik maken van paging. Ook hardware optimalisaties zoals snelle arithmetiek, vektoroperaties en pipelining geven bij executie van LISP programma's slechts in bepaalde gevallen vergroting van de efficiëntie.

Kortom, de architektuur van de von Neuman machine is zeker niet de meest geschikte voor het uitvoeren van LISP programma's. In de volgende paragrafen worden een aantal alternatieven besproken die wellicht meer geschikt zijn voor LISP.

3.2. Niet specifieke LISP machines

Onder niet specifieke LISP machines worden computers verstaan die gebaseerd zijn op een traditionele (von Neumann) processor, maar die door hardware modificaties geschikter zijn gemaakt voor de executie van LISP programma´s dan de standaard beschikbare hardware. In ca. 5 verschillende onderzoekcentra zijn dergelijke machines in ontwikkeling: MIT, Xerox PARC, BBN, en twee universiteiten in Japan. We zullen ons hier voornamelijk beperken tot het bespreken van een aantal aspekten van de MIT LISP machine en de Xerox PARC "Alto".

Basis van de MIT LISP Machine (Greenblatt, 1979) is een microgeprogrammeerde processor, "CONS" (Steele, 1980), die voorzien is van een 32-bits datakanaal en een 24-bits adreskanaal. Het micro-code geheugen van 16K, 48-bits bevat een groot aantal micro-gecodeerde onderdelen van het LISP systeem. De processor is uitgerust met specifieke hardware voor het manipuleren van bit-velden in registers en een stapel-buffer die fungeert als een cache-geheugen voor de stapel van de LISP interpretator. De programmering van de MIT LISP Machine geschiedt geheel in LISP (en uiteraard in micro-code). LISP fungeert hier dus als de systeem implementatie taal. Het systeem is voorzien van een virtueel geheugen (paging is geheel door microcode gerealiseerd), is "single-user" en kommuniceert met de gebruiker via een beeldscherm met hoge resolutie, dat het zichtbaar maken van komplexe informatie mogelijk maakt (vgl. de Interlisp P.A).

De representatie van data in de LISP machine is anders dan in de meeste LISP implementaties. Een LISP objekt wordt gerepresenteerd door een 24-bits adres en een 5-bits datatype. Het datatype kan zijn atoom, getal, lijstcel, string of array. Bij de representatie van lijststrukturen wordt gebruik gemaakt van een techniek die een meer optimaal gebruik van de sequentiele geheugenorganisatie mogelijk maakt: CDR-coding. Een aantal bits in elk geheugenwoord zijn gereserveerd om aan te geven waar de CDR van de betreffende cel zich bevindt. De bits geven aan of de CDR gevonden kan worden in de volgende geheugencel, ofwel de CDR verwijst naar de lege lijst (NIL), of dat de CDR indirekt via de volgende geheugencel gevonden moet worden. In de praktijk blijkt dat een groot deel van de lijststrukturen die door LISP programma´s gemanipuleerd wordt lineair is en met NIL wordt afgesloten. In deze gevallen geeft CDR-coding een aanzienlijke geheugenbesparing. De in de vorige paragraaf genoemde problemen die ontstaan door vervanging van delen van lijststrukturen door andere met een verschillende lengte, worden opgelost door introduktie van "onzichtbare" verwijzingen naar cellen elders in het geheugen. Deze onzichtbare wijzers zijn een vorm van indirekte adressering. Speciale micro-programmering zorgt ervoor dat deze verwijzingen niet al te veel extra processortijd in beslag nemen. Tijdens garbage collectie kunnen

deze ontzichtbare verwijzingen verwijderd worden. Een aantal andere mechanismen kunnen ook efficiënt geïmplementeerd worden door gebruik te maken van onzichtbare verwijzingen. Bijvoorbeeld het opzoeken van globale variabelen in closures kan zeer versneld worden door de interne waarde van een gesloten variabele via een onzichtbare verwijzing te verbinden aan de normale cel waarin de waarde van de variabele te vinden is (de LISP machine gebruikt "shallow binding").

De garbage collector is speciaal voor de MIT LISP machine ontwikkeld (Baker, 1978). Deze garbage collector is incrementeel, d.w.z. telkens wanneer een objekt gekreëerd wordt, is de garbage collector even actief. Dit is belangrijk aangezien een niet incrementele garbage collector zeker enige minuten nodig zou hebben om de volledige adresruimte door te werken. Het principe van de garbage collector is eenvoudig: de adresruimte van de machine wordt in twee helften verdeeld. Cellen die aktief zijn worden voortdurend naar de andere helft gekopieerd en vervangen door een onzichtbare verwijzing. Wanneer een geheugenhelft geen aktieve cellen meer bevat, verwisselen de geheugenhelften van rol. Dit principe maakt ook dichte pakking van de cellen mogelijk, die van belang is i.v.m. paging.

De macroinstrukties van de LISP machine zijn 16 bits lang en opereren op een stapel georienteerde machine. De stapel is verdeeld in segmenten, waarbij een segment de informatie bevat die hoort bij de invokatie van een funktie. Elk segment bevat de argumenten waarmee de funktie is aangeroepen, de waarden van de lokale variabelen, een lokale stapel voor tussenresultaten, en verwijzingen naar andere segmenten. De instrukties van de LISP machine kunnen de elementen uit een stapelsegment en een aantal systeemkonstanten (o.a. NIL) direkt adresseren. Adressering van niet lokale variabelen en funkties geschied via onzichtbare verwijzingen in een tabel die specifiek is voor elke gecompileerde funktie. Deze techniek is gebaseerd op de observatie dat in de praktijk LISP funkties slechts een beperkt aantal globale variabelen adresseren en een beperkt aantal funkties aanroepen (Deutsch, 1973). De macroinstrukties implementeren een aantal min of meer standaard operaties zoals datatransmissie (optioneel gekombineerd met primitieve operaties op lijststrukturen zoals CAR, CDR etc), operaties op de verschillende typen LISP objekten (arithmetische operaties, operaties op lijsten, vergelijkingsopdrachten), konditionele sprongopdrachten en instrukties voor het aanroepen van funkties. Naast deze opdrachten zijn er 512 instrukties voor veel gebruikte LISP funkties die micro-gecodeerd zijn.

Hoewel de stapelsegmenten voldoende informatie bevatten om implementatie van een "spaghetti-stack" mechanisme (de stapel kan gedeeltelijk behouden blijven, waardoor een stapel met een boomstruktuur ontstaat;

Bobrow, 1973) mogelijk te maken, is in de MIT LISP machine gekozen voor een eenvoudiger mechanisme om multi-processing te implementeren: de stapel-groep. Een stapel-groep is een LISP objekt waarin de toestand van een proces (een verzameling stapel segmenten) behouden kan blijven om later weer geaktiveerd te worden. In AI programma's veel gebruikte besturingsregimes zoals co-routines en generatoren (processen die meer dan een waarde kunnen afgeven) kunnen via het stapel-groep mechanisme efficiënt geïmplementeerd worden. Het voornaamste verschil tussen het spaghetti-stack mechanisme en het gebruik van de stapelgroep zit in het feit dat de gebruiker stapel-groepen expliciet moet kreëren en manipuleren, terwijl in het geval van het spaghetti-stack mechanisme deze administratie door de interpretator gedaan wordt.

De verwerkingskapaciteit van de huidige versie van de MIT LISP machine is ca. 3 maal groter dan die van een DEC-10 (KA, stand alone), de prijs is echter een factor 10 lager.

De Xerox LISP machine (de Alto), is in veel opzichten vergelijkbaar met de MIT machine. Een belangrijk verschil ligt in de vorm van de datarepresentatie. De Alto representeert het datatype van een LISP objekt niet expliciet, maar groepeert objekten van eenzelfde type in pagina's, waarvan het datatype in een tabel gegeven is. Het gevolg is dat meer bits beschikbaar zijn voor andere doeleinden, met name voor de CDR-codering van lijstcellen. In de Alto bestaat een lijstcel uit een 24 bits CAR veld, en een 8 bits CDR veld, dat een relatief adres op dezelfde pagina aangeeft. Valt de CDR buiten de pagina dan wordt gebruik gemaakt van onzichtbare verwijzingen op een analoge wijze als bij de MIT LISP machine. Afhankelijk van de linearizering van de lijsten in het systeem is de codering van de Alto efficiënter in ruimtegebruik (Bobrow, 1979). Andere verschillen tussen de MIT en Xerox machines zijn: het gebruik van spaghetti-stacks (Xerox) en een kompaktere representatie van programma's die dichter staat bij de representatie als lijststruktuur.

3.3. Specifieke LISP machines: LISP chips

Naast de ontwikkelingen op het gebied van niet-specifieke LISP machines, zijn een tweetal chips ontworpen voor het verwerken van LISP (Steele, 1979; Holloway, 1980). Beide chips interpreteren een dialekt van LISP, SCHEME. Het belangrijkste verschil tussen SCHEME en LISP is dat in SCHEME statische binding van variabelen wordt toegepast, in tegenstelling tot de dynamische binding in LISP. SCHEME-79 (de meest geavanceerde versie van de LISP chip) is een 32 bits machine met een 24 bits adreskanaal. Alle informatie die door de chip wordt gemanipuleerd is opgebouwd uit LISP cellen, bestaande uit een 24 bits adres, een 7 bits datatype veld en

1 bit ten behoeve van de geheugenallocator en de garbage collector. LISP programma´s worden gerepresenteerd in S-code, de machinecode van de chip, die bestaat uit lijststrukturen waarin de datatype velden van elke cel aangeven hoe een lijst moet worden geïnterpreteerd. Zo zijn er datatypen voor procedures, lokale en globale variabelen, funktie aanroepen (postfix code). Het datatype van een cel kan dus gezien worden als een instruktiecode. De machine code van de chip staat dus zeer dicht bij de oppervlakte representatie van LISP, terwijl de uniformiteit van data en programma´s optimaal is.

Evaluatie van een lijststruktuur geschiedt in stappen waarin steeds een LISP cel geëvalueerd wordt. De akties die de machine onderneemt worden geheel bepaald door het datatype van de cel. Onder normale omstandigheden wordt de volgende te evalueren cel gevonden in de CDR van de juist geevalueerde cel. Bij het evalueren van subexpressies wordt gebruik gemaakt van een interne stapel, die gerepresenteerd is als een lijst. De evaluator zelf is in de chip geïmplementeerd als een konventionele eindige toestand machine, voorzien van een geheugen allocatie mechanisme en een garbage collector.

Opmerkelijk is wellicht de methodologie die gehanteerd is bij het ontwerpen van de chip. De komplexiteit van het VLSI ontwerp wordt gereduceerd door verschillende specifikatie niveau´s te hanteren. In de eerste fase van het ontwerp worden de algorithmen die de machine moet implementeren geschreven in LISP (uitgebreid met primitieve operaties op machinestrukturen). De in LISP geschreven interpretator wordt gecompileerd naar een relatief conventionele machinecode, waarin met name de gestruktureerde expressies gelinearizeerd zijn. In een tweede fase van het compilatieproces wordt de microcode geconverteerd naar specificaties voor de geprogrammeerde logica array´s (PLA´s). De derde fase genereert de architektuur van de chip. Hierbij wordt gebruik gemaakt van een op LISP gebaseerde layout taal, waarin de layout van verschillende komponenten gedefinieerd kan worden. De definitieve layout van de chip wordt in interaktie met de ontwerper ontwikkeld. Een laatste programma zet de symbolische representatie van de layout om in een tekening van de chip, die geschikt is als invoer in het eigenlijke fabrikageproces.

De prestaties van de LISP chip zijn vergelijkbaar met die van geïnterpreteerd LISP (op een DEC-10), waarbij opgemerkt moet worden dat de chip ca. 80% van de tijd bezig is met geheugenalloactie en garbage collectie, een percentage dat bij gebruik van een conventionele stapel drastisch verlaagd zou kunnen worden.

4. MULTI-PROCESSOR SYSTEMEN

Naast het ontwikkelen van specifieke LISP machines, is het gebruik van multi-processor systemen een mogelijke oplossing voor het probleem van de steeds groeiende behoefte aan adresruimte en verwerkingskracht. Bij het ontwerpen van dergelijke systemen is een cruciale vraag op welk niveau de werklast gedistribueerd moet worden. Een typisch AI programma kan opgebouwd gedacht worden uit een aantal lagen (vgl. Winston, 1975):

1) hardware
2) machinecode
3) LISP
4) AI taal, kennisrepresentatie systeem
5) AI programma

Het distribueren op de laagste twee niveau's heeft tot gevolg dat een LISP gebruiker geen weet heeft van het feit dat zijn programma door meerdere processoren wordt uitgevoerd. Afgezien van de mogelijkheid om geheugenallocatie en garbage collectie parallel te laten verlopen aan het evaluatie proces, biedt parallelle verwerking van LISP vooralsnog weinig perspektieven, voornamelijk vanwege zijeffekten. Beperkte ervaring met het PLEIADES I systeem (Coghill 1979), dat bestaat uit meerdere processoren en een gedistribueerde LISP interpretator lijkt aan te tonen dat grotere winst behaald kan worden wanneer de gebruiker zelf de werklast verdeelt over meerdere processen, dan met een gedistribueerde interpretator. Overigens kunnen processen die de interaktie met de gebruiker ondersteunen (vgl. de Programmer's Assistant) wel gedistribueerd worden over een aantal processoren, mits de verschillende processen voldoende toegang hebben tot elkaars databestanden. Een aantal processoren gegroepeerd rond een gemeenschappelijk geheugen zou voor deze specifieke toepassing een oplossing kunnen zijn.

Distributie op het niveau van een AI taal of representatie systeem (zoals PLANNER, KRL) biedt wellicht meer perspektieven dan distributie op lagere niveau's. De meeste AI talen bieden mogelijkheden voor het onderhouden van meerdere aktieve processen (co-routines, generatoren, "frames" die een procedurele component hebben, vergelijkbaar met Simula classes). Een generator in CONNIVER (McDermott, 1972) bijvoorbeeld is een proces dat geaktiveerd wordt op grond van een vraag aan het databestand, en dat successieve antwoorden genereert door nieuwe vragen aan het databestand te stellen en door inferenties te maken. Distributie van dergelijke processen over meerdere processoren heeft dan het effekt van een parallel zoekproces door het databestand, zonder dat de gebruiker van de AI taal expliciet parallelle berekeningen hoeft te specificeren.

Een ander type systeem dat zich goed leent voor impliciet parallellisme is het produktiesysteem. In veel AI programma's wordt kennis gerepresenteerd in de vorm van regels, die bestaan uit een conditie en een aktie deel. Data wordt geschreven in en gelezen uit een werkgeheugen. In elke interpretatiestap worden de kondities van alle produkties geëvalueerd in de kontext van de inhoud van het werkgeheugen, en worden die akties uitgevoerd die korresponderen met kondities waaraan voldaan is. Produktiesystemen hebben dus van nature een parallel karakter. Probleem is wel dat verschillende produkties die tegelijkertijd geaktiveerd worden tegenstrijdige informatie aan het werkgeheugen kunnen toevoegen. Verschillende strategieën voor het oplossen van dergelijke konflikten worden gehanteerd, maar deze zijn i.h.a. niet kompatibel met parallellisme. Een ander probleem met distributie op het niveau van de AI taal of het representatiesysteem, is de grote bandbreedte die nodig is voor de kommunikatie tussen de processoren en een databestand of werkgeheugen. Een netwerk van zwak gekoppelde processoren kan hier niet volstaan, waardoor de hardware organisatie komplex wordt en synchronisatieproblemen van allerlei aard kunnen ontstaan.

Distributie op het niveau van het AI programma zelf is mogelijk als de taak die moet worden uitgevoerd kan worden opgesplitst in verschillende subtaken die slechts een beperkte interaktie hebben en dus grotendeels parallel kunnen worden uitgevoerd. Er bestaat een tendens in het huidige AI onderzoek om grote AI programma's te struktureren in verschillende processen die ieder op zich expert zijn op een bepaald subdomein. De achtergrond hiervan is niet het gebruik van meerdere processoren, maar een methodologie die de komplexiteit van grote AI programma's hanteerbaar houdt. Kommunikatie tussen de verschillende expert processen kan geschieden door het zenden van boodschappen of via een gemeenschappelijk databestand. Zo kan bijvoorbeeld een programma dat natuurlijke taal verwerkt bestaan uit een komponent die woorden in een lexikon opzoekt, een syntaktische komponent, een semantische komponent die geaktiveerde kennisstrukturen aan elkaar relateerd en een pragmatische komponent die op basis van kennis van de wereld inferenties maakt om stukken verwerkte taal met elkaar te verbinden tot een koherent geheel. Het HEARSAY-II spraakverwerkings systeem (Lesser et. al. 1975) werkt volgens dit principe.

Een nog sterker gedecentraliseerd model is dat van het "contract-net" (Smith, 1979). Het contract-net bestaat uit een aantal knooppunten, waarin zich processoren bevinden die bepaalde taken kunnen uitvoeren. Distributie van taken geschiedt op basis van een onderhandeling tussen een aantal knooppunten: een knoop die een taak wil laten uitvoeren (de "manager") en een of meer knopen die menen de taak te kunnen uitvoeren

(de "aannemers"). De aannemers doen ieder een "bod", dat aangeeft hoe kapabel zij zichzelf achten om de taak uit te voeren. De beslissing welke knoop uiteindelijk de taak zal uitvoeren wordt door de manager genomen, op grond van het hoogste bod. Het contract-net lijkt een geschikte architektuur voor systemen die problemen moeten oplossen die opgesplitst kunnen worden in taken die een relatief grote verwerkingstijd vragen vergeleken met de tijd die nodig is in de onderhandelingsfase.

5. KONKLUSIES

Drie ontwikkelingen zijn in het voorgaande naar voren gekomen die de relatie tussen LISP als hogere programmeertaal en de architektuur van computersystemen betreffen. Ten eerste is er de toenemende integratie van LISP zelf en het computersysteem in de vorm van een geavanceerd interaktief systeem, dat geheel georiënteerd is op het werken met LISP. Verdere ontwikkelingen in deze richting zijn te verwachten op het gebied van intelligente ondersteuning van de programmeeraktiviteit door het computersysteem. De Interlisp Programmer's Assistant bezit enige kennis van LISP en het LISP systeem op syntaktisch niveau. Systemen die kennis bezitten van de semantiek van LISP zijn in ontwikkeling (de "Programmer's Apprentice", Rich, 1976). In dergelijke systemen zal de interaktie met de gebruiker meer en meer de vorm aannemen van een dialoog waarin de gebruiker specificeert wat er gedaan moet worden, en het computersysteem beslist hoe het gedaan moet worden. Er is niet alleen sprake van een symbiose van programmeertaal en computersysteem, maar ook van een integratie van de programmeeraktiviteit en de executie van het programma.

De tweede ontwikkeling richt zich op de konstruktie van hardware- en software systemen die een adekwate ondersteuning van de steeds groter wordende eisen van de LISP gebruiker mogelijk maken, in de vorm van "personal computers". Hoewel deze computersystemen in sommige opzichten specifiek voor LISP ontworpen zijn, blijft de basis architektuur waarop ze gebaseerd zijn die van de von Neumann machine. Dit is verwonderlijk gezien de grote afstand die bestaat tussen de virtuele LISP machine en de von Neumann architektuur, maar geeft wellicht aan hoe moeilijk het is om het denkkader van de von Neumann machine te verlaten. Ontwikkelingen op het gebied van machines die op boomstrukturen opereren ("tree-machines") en introduktie van associatieve geheugens zijn hier mogelijk richtingen van verder onderzoek.

De derde ontwikkeling die met name in recent AI onderzoek gesignaleerd kan worden, is die naar gedistribueerde verwerking door een aantal zwak gekoppelde processen die ieder een specifieke taak uitvoeren. Hier ligt niet de nadruk zo zeer op de interaktie tussen computerarchitektuur en programmeertaal maar eerder op de relatie tussen de struktuur

van het computersysteem en de strukturele opbouw van het verwerkingsproces. Het is zeer wel mogelijk dat op den duur deze ontwikkelingen zullen leiden tot een geheel nieuwe programmeermethodologie en tot programmeertalen die een dergelijke methodologie ondersteunen.

REFERENTIES

Baker, H.G. (1978) List processing in real time on a serial computer Comm. ACM, 21(1978)280.

Bobrow, D.G. & Clark, D.W. (1979) Compact encodings of list structure ACM TOPLAS 1(1979) pp266-286.

Bobrow, D.G. & Wegbreit, B. (1973) A model and stack implementation of multiple environments Comm. ACM 18 (1973),pp 591-603.

Brown, J.S., Burton, R.R., Bell, A.G. & Bobrow, R.J. (1974) SOPHIE: A Sophisticated Instructional Environment Lowry Air Force Base, Colorado: Technical Training Division, AFHRL-TR-74-93, Dec. 1974.

Coghill, G.G. & Hanna, F.K. (1979) PLEIADES Internal report 3: LISP interpreter Internal report, Electronics Lab. Univ. of Kent at Canterbury, 1979.

Davis, R., Buchanan, B. & Shortliffe, E. (1977) Production Rules as a Representation for a Knowledge-Based Consultation Program Artificial Intelligence, vol 8, 1977.

Deutsch, L.P. (1973) A LISP machine with very compact programs Proc. 3rd IJCAI, Stanford, Calif, 1973, pp697-703.

Greenblatt, R. Bawden, A, Holloway, J. Knight, T. Moon, D. & Wienreb, D. (1977) LISP machine progress report MIT AI-memo 444, 1977.

Holloway,J. Steele, G.L. Sussman, G.J. & Bell, A. (1980) The SCHEME-79 chip MIT AI-memo 559.

Klint, P. (1980) De symbiose van programmeertaal en computer (dit colloquium).

Landin, P.J. (1964) The mechanical evaluation of expressions Comp. Journal 6(1964),308.

Lesser, V.R., Fennel, R.D., Erman, L.D. & Reddy, D.R. (1975) Organisation of the HEARSAY II speech understanding system. IEEE Trans. on Acoustics, Speech and Signal Processing, 1975, ASSP-23, 11-23.

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine Comm. ACM, 1960.

McDermott, D.V. & Sussmann, G.J. (1972) CONNIVER reference manual MIT AI-Memo 259, 1972.

Rich, C. & Shrobe, H.E. (1976) Initial report on a LISP Programmer's Apprentice MIT AI-TR-354, 1976.

Sandewall, E. (1978) Programming in the interactive environment: the LISP experience. Comp. surveys 10 (1978), 73.

Smith, R.G. (1979) A framework for distributed problemsolving Proc. 6th IJCAI, Tokyo, 1979.

Steel, G.J. (1980) CONS MIT AI-memo, nog niet gepubliceerd.

Steele, G.L. & Sussman, G.J. (1979) Design of LISP-based processors. MIT AI-Memo 514, 1979.

Teitelman, W. (1972) Automated programming - the programmer's assistant Proc. Fall Joint Comp. Conf. 1972.

Teitelman, W. (1974) Interlisp reference manual Xerox Palo Alto Research Center, Palo Alto, 1974.

Teitelman, W. (1977) A display oriented programmer's assistant Proc. 5th Int. Joint Conf. on Artificial Intelligence, 1977.

Winston, P. (1975) Artificial Intelligence Addison-Wesley, 1975.

HOGERE PROGRAMMEERTALEN GEZIEN VANUIT DE MACHINETAAL

G.A. BLAAUW
Technische Hogeschool Twente

0. INLEIDING

Voor sommige onderwerpen in de computer-architektuur is het aantal publikaties omgekeerd evenredig aan het aantal geproduceerde machines. Daar dit verschijnsel ook geldt voor het onderwerp van dit colloquium, is het nuttig na te gaan wat hiervan de oorzaak is. Naar ik meen is deze oorzaak ten dele een begripsverwarring en ten dele inhoudelijk.

Om begripsverwarring te voorkomen behandelen wij eerst de termen hogere programmeertalen en computer-architektuur. Ten tweede zal de von Neumann computer besproken worden. Wij doen dit omdat deze computer vaak genoemd wordt in kontrast tot computers die voor hogere programmeertalen meer geschikt zouden zijn; tevens schetsen wij kort de ontwikkeling van de computer-architektuur. Ten slotte beschouwen wij het begrip machinetaal.

De inhoudelijke overwegingen laten zich in twee delen splitsen. In ´de omhoogstrevende uitrusting´ stellen wij dat de drang naar een hogere machinetaal van nature vanuit de computerapparatuur komt. In ´de neerdrukkende programmatuur´ stellen wij dat de hogere talen en hun compilers in tegengestelde richting werken [1].

1. TERMEN

1.1. Hogere programmeertalen

De aankondiging van dit colloquium*) toont Basic, Fortran, en Algol 68 in een illustratie. Dit is een gelukkige suggestie. Basic en Fortran behoren tot de meest gebruikte programmeertalen. Algol 68 wordt, in verhouding, heel weinig gebruikt, maar is wel algemeen bekend als een geavanceerde hogere programmeertaal. Deze suggestie kan nog wat worden

*) Blaauw verwijst hier naar de figuur op de omslag van de colloquium-folder. Deze is om reproductietechnische redenen niet opgenomen (red.).

uitgebreid door te stellen dat een typisch computercentrum een groot aanbod van interactief werk in Basic en batchwerk in Fortran te verwerken krijgt en daarnaast ook nog zo nu en dan Algol 68 moet kunnen verwerken. Ik neem aan dat wij het over zulk een omgeving zullen hebben.

Deze aanname is voor ons onderwerp belangrijk. Het betekent dat een computer veel gebruikte oude en lelijke talen moet kunnen verwerken zowel als weinig gebruikte nieuwe en mooie talen. Het betekent ook dat een veelheid van talen door elkaar verwerkt moet kunnen worden; in de praktijk natuurlijk veel meer talen dan de drie genoemde voorbeelden.

Dat deze aanname niet triviaal is kan geïllustreerd worden uit het recente boek van Myers [2] dat het onderwerp van ons colloquium behandelt. Van de vier voorbeeld-machines uit dat boek vallen er drie meteen af. De SYMBOL van Rice aanvaardt alleen programma's in zijn eigen taal; Wortman's machine is beperkt tot een incompatibele subset van PL/I; Myers' eigen machine, SWARD, is meer algemeen, maar kan onmogelijk voor Fortran, Cobol of PL/I gebruikt worden. De vierde machine is de Burroughs B1700 waar we later op terug komen.

1.2. Computer-architektuur

De architektuur van een computer is het functionele gedrag van de computer ten aanzien van zijn onmiddellijke gebruiker. Deze onmiddellijke gebruiker is gewoonlijk een systeemprogrammeur die in de machinetaal programmeert.

Deze definitie onderscheidt de computer-architektuur van andere gebieden van computerapparatuur-ontwerp en wel:

- de logische organisatie van de interne databehandeling en besturing, die we de uitrusting noemen, en
- de fysische belichaming van de uitrusting met componenten, de realisatie.

Deze driedeling is in het kader van het Mathematisch Centrum reeds tien jaar geleden aan de orde geweest [3]. Voor ons onderwerp van vandaag is vooral het kontrast tussen de eerste twee, de architektuur en de uitrusting, van belang.

De term computer-architektuur is inmiddels bijna twintig jaar oud [4]. Hij is in de loop der jaren verscherpt tot bovengenoemd begrip [5] en is de laatste tijd door diverse auteurs gehanteerd [6,7,2]. Andere auteurs [8,9] gebruiken deze term echter voor een combinatie van architektuur en uitrusting. Het bezwaar van zulk een combinatie is dat we nu

niet meer over elk gebied afzonderlijk kunnen spreken. Deze gebieden zijn namelijk sterk verschillend in aard en ontwerpmethodiek en de bewustwording van dit verschil is een belangrijke stap geweest in de ontwikkeling van de computer. Ook de aankondiging van dit colloquium suggereert in tenminste één van de gebezigde diagrammen dat de uitrusting deel is van ons onderwerp. Het is dus niet overbodig tevoren te vermelden wat we met onze termen bedoelen.

Waar de term architektuur verschillend gebruikt wordt is dit met de term computer minder het geval. Wel is het nodig te vermelden welke soort computer we in gedachten hebben. In dit hoofstuk wil ik spreken over computers die commercieel, dat wil zeggen concurrerend, geproduceerd kunnen worden. Alleen van zulke computers kan de normale gebruiker van hogere talen met profijt gebruik maken. Ook deze aanname is niet triviaal. Twee van Myers´ eerder genoemde ontwerpen zijn papieren voorstellen; van de derde, SYMBOL, bestaat slechts één exemplaar. Deze drie ontwerpen hebben dus niet tot concurrerende produktiemodellen geleid. Hoe interessant deze ontwerpen ook zijn, zij zijn in ieder geval op dit moment niet het antwoord op de wens naar apparatuur die beter voor hogere talen geschikt is. De Burroughs B1700 is daarentegen wel een produktiemodel en komt daarmee als kandidaat in aanmerking.

2. VON NEUMANN ARCHITEKTUUR

Het is de laatste jaren de gewoonte om een klassieke computer architektuur als ´von Neumann architektuur´ aan te duiden [10]. Nu is het verheugend dat von Neumann´s aanzienlijke bijdrage aan de ontwikkeling van de computer hiermee erkend wordt. Maar het is spijtig dat er vaak een negatieve connotatie aan dit gebruik gekoppeld is: zo van ´dat willen we in ieder geval niet´. Bovendien is het gewoonlijk onduidelijk, of van geval tot geval verschillend, wat met de term bedoeld wordt.

Von Neumann´s ontwerp dateert van een voorstel uit 1946 [11] dat in Princeton als machine voltooid werd, maar voordien reeds een groot aantal ontwerpen beïnvloed heeft, zoals de EDSAC, Manchester Mark I (nu MU1 genoemd), WHIRLWIND, MANIAC, ORDVAC, ILLIAC, JOHNNIAC en IBM 701. Wij beschouwen hieronder de hoofdpunten van dit ontwerp en zullen daarbij nagaan in hoeverre deze nog kenmerkend zijn voor de huidige computer.

2.1. Gemeenschappelijk geheugen

Het gebruik van het hoofdgeheugen voor zowel instrukties als data is een opmerkelijke stap in de ontwikkeling van de computer. Babbage´s Difference Engine (1833) en Analytical Engine (1867), Zuse´s Z3 (1941) en Aiken´s Mark I (1944) gebruikten gespecialiseerde input-apparatuur voor

instrukties, zijnde Jacquard kaarten, film en brede papierband. In principe was dit een ROM (ready-only memory) met sequentiële toegang. Eckert en Mauchly's ENIAC (1946) gebruikte een ROM met aselekte (random-access) toegang in de vorm van een stekerbord (plug-board) en schakelaars. Een ander voorbeeld van zulk een gespecialiseerd aselekt ROM-geheugen was de trommel van de Harvard Mark IV (1952). Het programma werd als een aparte aktie op de trommel geschreven, niet ongelijk aan een huidige PROM (programmable read-only memory).

Het gemeenschappelijk gebruik van een aselekt (RAM) lees- and schrijfgeheugen voor instrukties en data is door Eckert en Mauchly in 1944 voorgesteld voor hun EDVAC (1952); het is het eerst gebruikt in hun BINAC, in Wilkes' EDSAC en in Kilburn's MU1 in 1949, von Neumann nam deel in het ontwerp van EDVAC en publiceerde het idee van een gemeenschappelijk geheugen in 1945. Metropolis en Worlton [12] hebben zorgvuldig aangetoond, dat von Neumann aanzienlijk geholpen heeft in de ontwikkeling van dit idee maar het niet uitgevonden heeft.

Het gebruik van een gemeenschappelijk geheugen voor instrukties en data was oorspronkelijk aantrekkelijk om instrukties te modificeren, maar dat werd al spoedig van voordeel tot nadeel veranderd. Wel is het nog steeds aantrekkelijk de geheugenruimte naar believen in te kunnen delen. In veel microprocessoren wordt tegenwoordig een aparte PROM of ROM voor instrukties toegepast.

2.2 Binair grondtal

Het von Neumann ontwerp was geheel binair, zoals Zuse al eerder gebruikt had. Een tijd lang zijn er een binaire en een decimale traditie naast elkaar geweest. De huidige computer bevat veelal beide, waarbij de adressering binair is en de decimale mogelijkheden vaak beperkt zijn.

2.3. Registers

Von Neumann's ontwerp gebruikte enkel-adres (one-address) instrukties en een geïmpliceerde accumulator. Zulk een accumulator kan als een werkgeheugen opgevat worden, waarbij van belang is hoe groot dit werkgeheugen is en hoe het geadresseerd wordt.

De IBM 705 (1956) verving als een van de eersten het enkelvoudige werkgeheugen door een 16-tal registers die elk een tussentijds resultaat konden bevatten; een traditie die nog in veel huidige ontwerpen zoals IBM System/360, CDC 6600, DEC PDP11 en INTEL 8080 te vinden is.

2.4. Adrestype

Een veelvoud van registers vereist register-adressering. Deze adressering was oorspronkelijk steeds direct, maar de English Electric KDF9 (1962) en Burroughs B5500 (1965) hebben hiervoor de LIFO (last in first out) implicatie gebruikt.

Naast de één-adres instruktie van von Neumann zijn vanaf het begin de twee-adres, zoals in de ERA 1101 (1953), en de drie-adres instruktie, zoals in de SEAC (1951) gebruikt. Deze alternatieven zijn nog steeds in computers zoals de IBM System/360 en de CDC 6600 terug te vinden.

2.5. Indicering

Het ontbreken van adres-indicering in von Neumann´s ontwerp was zulk een tekort dat het vrijwel onmiddellijk achterhaald werd. Kilburn´s MU1, een van de eerste werkende von Neumann ontwerpen, introduceerde indicering in de vorm van een zogenaamde B-line; later bevatte deze machine reeds 8 indexregisters. Dubbele indicering was in Kilburn´s ATLAS (1962) systematisch aanwezig voor 127 registers. Zestienvoudige indicering als aparte aktie in de IBM Stretch (1961).

2.6. Bewerkingsrepertoire en datatype

Oude computers hadden een rijker aritmetisch bewerkingsrepertoire dan het von Neumann ontwerp. De Mark I leek wat dat betreft op een moderne handrekenmachine.

Von Neumann beperkte zich tot de elementaire rekenbewerkingen en tot het fixed-point datatype. Eerdere ontwerpers, zoals Zuse (1941) en Stibitz (1947) hadden al floating-point gebruikt. De IBM 704 (1955) en andere opvolgers van de eerste groep von Neumann machines hervatten het gebruik van dit datatype.

De beperking van de aritmetische bewerkingen tot optellen, aftrekken, vermenigvuldigen en delen illustreert het quid pro quo principe. De beslissingen van de computer-architektuur betreffen niet het al of niet beschikbaar stellen van een funktie, maar het afwegen van de wenselijkheid van een funktie ten opzichte van een andere. Vandaar dat het operatie-repertoire moeilijk uit te breiden is.

2.7. Bewerkingsspecificatie

De vaste-lengte-opdrachtcode van de eerste machines is al spoedig verlaten. Een extreem voorbeeld is de variabele-lengte-code van de IBM Stretch (1961). Ook de volledige codering met een enkel getal is vervangen door een combinatie van codes en modificatiebits. Een uiterste geval is hier de Zebra van van der Poel (1959) die vrijwel alleen modificatiebits kende.

De volledige gecodeerde specificatie komt overeen met de vertikale microcode; volledig gedecodeerde specificatie in termen van modificatiebits correspondeert met de horizontale microcode; de tussenoplossing lijkt op de gebruikelijke quasi-horizontale microcode. Hieruit volgt dat het type opdrachtcode niet bepalend is voor het type computer; we spreken over mini- en microcomputers vanwege de omvang van de realisatie.

2.8. Vaste-lengte-operands

Het gebruik van vaste-lengte-operands in de von Neumann machine is nooit de enige oplossing geweest. Tabelleermachines kenden gebrekkige variabele lengte. Vanaf de IBM 702 (1954) is er een groot aantal ontwerpen met variabele-lengte-operands geweest. Het uiterste van variabiliteit is de bitresolutie van de IBM Stretch die ook in de B1700 gevonden wordt.

2.9. Datastrukturen

Het von Neumann-ontwerp kende vektoren van logische elementen als datatype. Vektoren van getallen zijn in principe weinig anders. Praktisch heeft het echter tot de laatste tien jaar geduurd voordat numerieke vektoren als een eenheid bewerkt konden worden. De CDC STAR-100 (1973) en Cray 1 (1975) zijn prominente voorbeelden, hoewel niet de eerste. Een zelfde opmerking kan gemaakt worden voor karakterstrings. De IBM Harvest (1961) is een vroeg voorbeeld van een ambitieus special-purpose ontwerp voor de behandeling van karakterstrings.

2.10. Direkte geheugenadressering

De direkte adressering van het hoofdgeheugen is vanaf het begin gebruikt en geen von Neumann karakteristiek. Wel was het door hem voorgestelde adres van 12 bits veel te kort; Aiken gebruikte reeds 10.000 instruktielocaties. Concurrenten van direkte adressering zijn er wel geweest. Sequentiële geheugens kwamen bij tabelleermachines voor en bij de theoretische Turing-machine. De eeuwige kandidaat is echter de associatieve adressering. Deze vorm van adressering is ideaal voor trommelgeheugens en is zo ook in de Harvard Mark III (1948) in de uitrusting

- niet in de architektuur - gebruikt. Evenzo speelt associatieve adressering een belangrijke rol bij de uitrusting van adresvertalings-architekturen. Het principe is dus al lang bekend. De beperkte toepassing in de architektuur is een gevolg van quid pro quo; het is aantoonbaar kostbaarder dan direkte adressering.

De royale literatuur over associatieve adressering is wellicht een gevolg van de ´next technology fallacy´. Hierbij stelt men dat in een toekomstige technologie, zoals cryogeen, een gegeven funktie goedkoper zal zijn; maar in die technologie kunnen de concurrerende funkties vaak ook goedkoper zijn en het dus weer winnen.

2.11 Adresvertaling

De von Neumann-machine kende een logisch adres dat een één-op-één relatie had met het fysisch adres. Deze direkte relatie is in de zestiger jaren doorbroken door paginering op de ATLAS (1962) en segmentering op de Burroughs B5500 (1965).

2.12. Gelijktijdigheid

Wij onderscheiden parallelliteit in datastrukturen, zoals vektoren, van gelijktijdigheid van bewerkingen en procedures. Deze twee wezenlijk verschillende zaken worden vaak onder één noemer gebracht [13]; zij behoren echter apart behandeld te worden. De datastrukturen hebben wij onder 2.9 al genoemd.

Gelijktijdigheid van bewerkingen is reeds zeer oud. De Harvard Mark IV kon gelijktijdig optellen, vermenigvuldigen en delen. De programmeur kon hier gebruik van maken. Het is echter veel beter de programmeur niet met deze uitrustingsproblemen lastig te vallen. In de meeste snelle computers is dit type gelijktijdigheid nu volledig in de uitrusting verborgen.

Gelijktijdigheid van procedures ontstaat wanneer er meer dan één instruktiestroom gelijktijdig uitgevoerd wordt. Deze gelijktijdigheid behoort tot de architektuur. Het is mogelijk dat de uitrusting deze gelijktijdigheid alleen nabootst. Dit was onder andere het geval bij de TX2 (1959), die gelijktijdigheid van een hoofdproces en een aantal periphere processen kende.

Ten behoeve van de gelijktijdigheid is in de vijftiger jaren de interrupt ingevoerd; in 1963 heeft Jules Dirac de Test and Set voorgesteld, waarmee een semafoor geïmplementeerd kan worden.

Blijft de vraag naar de wenselijkheid van gelijktijdigheid. Is het snelheid, betrouwbaarheid, of de lage kosten van de hardware, dan is dit een uitrustingsmotivatie. Echte architektuur-redenen voor gelijktijdigheid, zoals gelijktijdigheid van in- en uitvoer, zijn zeldzamer dan uitrustingsredenen. De programmatuur bewijst zichzelf een slechte dienst als zij de taak van de apparatuur overneemt door gelijktijdigheid te forceren waar deze er van nature niet is. Dat leidt onherroepelijk tot complicatie. De drastische prijsverlaging van de apparatuur wordt wel als motivatie voor zulk een aktie gezien; zij zou in tegendeel juist aanleiding voor eenvoudige programmatuur moeten zijn.

2.13. Conclusie

Deze oplossing van architektuur-elementen zou kunnen worden voortgezet met soortgelijke opmerkingen over beslissingen, geheugenbescherming, etc. We kunnen echter zo al enige conclusies trekken.

(a) In enge zin is de von Neumann computer een baanbrekend ontwerp voor de zogenaamde wetenschappelijke computers geweest. Hier zijn echter vrijwel onmiddellijk drastische verbeteringen in gemaakt. Daarnaast is er steeds de administratieve computer geweest, waarvan het ontwerp in de zestiger jaren met dat van de wetenschappelijke computers tot de zogenaamde general-purpose computer is verenigd.

(b) In ruime zin is de von Neumann computer de ´stored program´ computer, zoals die nog steeds gebruikt wordt. Ook hier kan op een voortdurende stroom van verbeteringen gewezen worden, die zich ongetwijfeld nog zal voortzetten.

(c) Behalve verbeteringen zijn er ook tal van experimenten en voorstellen geweest die verworpen zijn. Een van de belangrijkste motieven voor dit verwerpen is het quid pro quo principe.

(d) Voor de ´semantische kloof´ [14] noemt Myers [2] voorzieningen voor array structures, strings, procedures, blocks, on-units en datarepresentatie. Op deze gebieden is echter wel degelijk werk verzet. Voor zover genoemde programmeerconcepten niet direkt in de machinetaal aanwezig zijn zullen we naar een beter argument moeten zoeken dan dat van een starre verouderde machine-architektuur. In sekties 4 en 5 noemen wij enige inhoudelijke redenen; eerst bezien wij echter in sektie 3 het begrip machinetaal.

3. MACHINETAAL

De titel van deze voordracht bevat de term machinetaal. Ook deze term kan een bron van begripsverwarring zijn. Voordat we een definiërend kenmerk geven volgt hier eerst een voorbeeld dat de noodzaak van begripsbepaling illustreert.

3.1. De eentalige machine

Al enige jaren worden er commercieel APL en Basic machines geproduceerd; ook zijn er nu Pascal machines beschikbaar. De commerciële produktie betekent dat aan onze terminologie van ´computer´ is voldaan. Ook zijn deze talen zeker ´hoog´. We laten de eis van meertaligheid voorlopig even rusten. Is nu voor deze machines de ´semantische kloof´ gesloten? Het antwoord hangt af van het begrip machinetaal.

3.2. Kenmerk van de machinetaal

Het ontwerp van een computer-architektuur is het ontwerpen van de volledige programmeertaal voor de computer. Deze programmeertaal wordt machinetaal genoemd. Het wezenlijke verschil tussen deze machinetaal en andere programmeertalen is dat de uitingen in de machinetaal kostbaar zijn. Efficiëntie van uitdrukking staat bij de machinetaal centraal. Bij hogere talen is dit duidelijk niet het geval, lange namen worden terwille van de begrijpbaarheid gebruikt. Ook bij de lagere microprogrammering is dit niet het geval; het lage niveau van de apparatuur leidt locaal tot een breedvoerige specificatie. Voor de machinetaal wordt echter elke bewerking en elke variabele uiteindelijk gerealiseerd in ruimte en tijd. Dit komt in het bijzonder tot uiting in het verkeer tussen het geheugen en de processor. Bits die onnodig aan dit verkeer deelnemen reduceren de prestatie/kosten-verhouding van de computer, onafhankelijk van de gebruikte technologie. Vandaar dat voor de machinetaal de uitingen efficiënt moeten zijn.

Uiteraard geven technologische ontwikkelingen verschuivingen in kosten en prestatie. Draden en pennen kosten in large scale integration meer dan logische componenten. Het verwijderen van overtolligheid in de architektuur helpt echter altijd de efficiëntie te verbeteren of er nu pennen of nands gespaard worden.

3.3. Microprogrammering

Microprogrammering is een uitrustingstechniek. Als deel van de uitrusting zal de microcode van model tot model radikaal verschillen. Maar ook binnen eenzelfde model kan er verschil zijn van machine tot machine,

terwijl zelfs voor een individuele machine in de loop der tijd wijzigingen mogelijk zijn. Als deel van de uitrusting is microcodering gewoonlijk gespecialiseerd in toepassing, complex in gebruik en primitief in funktie.

3.4. Beschikbare microcode

Wordt de microcode toch door een fabrikant aan de klant beschikbaar gesteld dan wordt dit een deel van de architektuur en dus machinetaal. De vrijheid om te wijzigen is daarmee verloren gegaan. De DEC PDP8 en PDP11/60 zijn voorbeelden hiervan. Het feit dat de codering horizontaal is, en de funktie primitief, is niet essentieel, al zal het wel het gebruik beinvloeden.

Op grond van onze definitie van machinetaal zijn de S-talen van de B1700 tussentalen en geen machinetaal. De taal waarin de S-talen zijn geïmplementeerd is machinetaal en geen microcode. Een vergelijking met andere machinetalen bevestigt dit, evenals het feit dat de architektuur een zestal uitrustingen kent. Myers´ vierde voorbeeld blijkt dus in feite een von Neumann-architektuur te hebben.

3.5. Verborgen machinetaal

Voor de APL en Pascal machines kunnen we nu stellen dat de machinetaal wel aanwezig maar niet beschikbaar is. Het feit dat er slechts een hoge taal is heeft op de aanwezigheid van een machinetaal geen invloed. Een van de APL machines kan juist wegens de onderliggende machinetaal eenvoudig tot Basic machine omgezet worden.

4. DE OMHOOGSTREVENDE UITRUSTING

Wij beschouwen nu de argumenten die leiden tot een machinetaal van hoger niveau. Wij beginnen daartoe met een architektuur van laag niveau.

4.1. Lage architektuur

Van der Poel´s Zebra (1959) [15] was de eerste computer waarvan de architektuur opzettelijk laag was gehouden. Elke instruktie van de Zebra vereiste een enkele datapad-aktie. Zulk een architektuur zullen wij een lage architektuur noemen. Deze komt veel voor bij de eerste generatie microprocessoren.

De lage architektuur die ontstaat door uit een hogere architektuur bewerkingen zoals vermenigvuldiging en deling te verwijderen zal geen goede prestatie hebben. Het vermenigvuldigen met een lus van optellingen

en verschuivingen duurt veel langer dan het datapad van zulk een machine toelaat. Vandaar dat de Zebra aan de onderdelen van grotere bewerkingen speciale aandacht schonk en in essentie een multiply-step en divide-step bevatte.

4.2. Instruktiebit-besparing

Naast de juiste datapad-akties moet ook het aantal instrukties dat uit het geheugen opgehaald wordt gereduceerd worden om overtollig bit-verkeer te elimineren. De uitrusting kan hiertoe diverse methoden gebruiken, zoals het bufferen van een 'inner loop'.

De klassieke methode om bit-efficiëntie te verkrijgen is echter de architektuur zo hoog te maken als de programmatuur toelaat. Vermenigvuldiging is daarbij beter dan een lus met multiply-steps, een drijvende-komma-optelling beter dan een subroutine en een Fast Fourier transformatie-instruktie beter dan een overeenkomstige procedure. Dit is ook duidelijk de tendens in mini- en microcomputer-architektuur.

4.3. De uitrusting kan het beter

Deze opwaartse druk wordt versterkt door secondaire presentatieverbeteringen die de uitrusting kan bewerkstellingen. Het datapad van de uitrusting is in wezen gelijktijdig. Daarom kan voor een drijvende-komma-resultaat de exponent gelijktijdig met de coëfficient berekend worden; bij een vektoroptelling kunnen verschillende elementen op het zelfde moment verwerkt worden. De ontwerper van de uitrusting heeft een arsenaal van methoden zoals pipeline, look-ahead, look-aside, guess and correct en specialcasing, om de prestatie te verbeteren, waarbij hij de beschikbare apparatuur zo goed mogelijk benut. Daarom kan men veilig stellen dat de uitrusting het altijd beter kan doen dan een programma.

5. DE NEERDRUKKENDE PROGRAMMATUUR

De opwaartse druk vanuit de uitrusting wordt op verschillende wijzen vanuit de programmatuur tot evenwicht gebracht.

5.1. Compilatie versus interpretatie

De aard van hogere programmeertalen is een van de voornaamste redenen voor het huidige machinetaalniveau. Deze talen vereisen veelal het omzetten van een potentieel lange mnemonische naam tot een compacte symbolische naam tijdens compilatie en vervolgens een omzetting van deze

compacte naam tot een eveneens compact adres tijdens het laden. De machinetaal staat ideaal juist tussen de eenmalige compilatie (early binding) en de herhaalde interpretatie (late binding).

5.2. Som versus product

Er zijn vele hoge talen; een lijst van Sammet uit 1969 vermeldde er reeds 120. Elk van deze heeft zijn gebruikers; voor de meeste is er een stel toepassingen voor welke deze taal het meest geeigend wordt geacht.

Er zijn ook vele computeruitrustingen die op een gegeven moment voor diverse investeringen de beste prestatie leveren.

Een direkte uitrusting voor elke taal en elke investering is het produkt van een aantal talen en het aantal gewenste prestaties. Een machinetaal als intermediair vereist in vergelijking slechts de som van het aantal talen en investeringen.

5.3. Ontkoppeling

De definitie van een hoge taal wijzigt zich met de tijd. Evenzo verandert de realisatie en daarmee de uitrusting van een architektuur. Een stabiele architektuur ontkoppelt deze veranderingen, zodat ze gemakkelijker gerealiseerd kunnen worden.

5.4. Grootste gemene deler

Een studie van taalconcepten, bijvoorbeeld beslissingsstrukturen, toont dat deze vaak in details verschillen. Een direkte absorptie van zulke taalconcepten in de architektuur maakt deze architektuur daarmee ongeschikt voor andere talen. De geschiedenis van de computerarchitektuur toont een hele reeks slachtoffers van zulke goedbedoelde funkties. Het resultaat is dat primitievere funkties als deel van de architektuur beter toepasbaar zijn dan totale funkties. Dit is ook de reden dat Myers' SWARD machine geen Fortran kan verwerken.

5.5. De vertaler weet het beter

Het interpretatieproces wordt door de machinetaal in twee delen gesplitst. De programmatuur neemt een deel voor zijn rekening, bijvoorbeeld van de Pascal P-code naar de machinetaal. De microcode neemt een tweede traject voor zijn rekening. Wij stelden dat de uitrusting beter kan optimaliseren als er meer gedaan kan worden. Maar de programmatuur is beter in staat te weten wat er gedaan moet worden. Op grond daarvan kan een geavanceerde compiler door kennis winst behalen op de uitrusting.

Een lage architektuur biedt dan een groot interpretatie-traject dat gunstiger is dan twee kleine trajecten.

5.6. Mogelijkheden voor hoge talen

De argumenten voor een lager taalniveau van de architektuur suggereren ook de toepassing van een hoog niveau. Indien wij ons kunnen beperken tot één enkele taal die tevens stabiel geacht kan worden en bij voorkeur sterk interpretatief is, dan is dit een gunstige kandidaat. Sommige implementaties van APL maken gebruik van zulk een hoge-orde-architektuur. Abrams [16] heeft echter aangetoond dat zelfs voor APL het beste machinetaalniveau lager ligt dan APL zelf.

6. CONCLUSIE

Het verband tussen hoge talen en computer-architektuur wordt vaak vertroebeld doordat het onduidelijk is wat met architektuur en met machinetaal bedoeld wordt. Wij hebben gepoogd definities te geven die zinvol en toepasbaar zijn. Vanuit deze definities blijkt dat er diverse krachten in het spel zijn die het niveau van de machinetaal naar boven dan wel naar beneden trachten te verplaatsen. Dit bevestigt de historische observatie dat de computer-architektuur een graduele ontwikkeling doormaakt waarin nog steeds zinvolle verbeteringen te behalen zijn.

REFERENTIES

[1] Blaauw, G.A. & F.P. Brooks, Computer Architecture. In voorbereiding.

[2] Myers, G.J., Advances in Computer Architecture. Wiley, New York, 1978.

[3] Blaauw, G.A., Het Gebruik van APL bij het Ontwerpen van Computers. MC-25 Informatica Symposium. Mathematisch Centrum, Amsterdam, 1971.

[4] Buchholz, W., Planning a Computer System. McGraw-Hill, New York, 1962.

[5] Amdahl, G.M., G.A. Blaauw & F.P. Brooks, Architecture of the IBM System/360. IBM Journal of Research and Development, Vol. 8, No. 2, pp. 87-101, 1964.

[6] Fuller, S.H., H.S. Stone & W.E. Burr, Initial Selection and Screening of the CFA Candidate Computer, Architectures, AFIPS, pp. 139-146, 1977.

[7] Bell, G.G., J.C. Mudge & J.E. McNamarra, Computer Engineering. Digital Press, 1978.

[8] Mano, M.M., Computer Architecture. Prentice-Hall, Englewood Cliffs, 1976.

[9] Foster, G.C., Computer Architecture. Van Nostrand, New York, 1976.

[10] Chu, Y., High-Level Language Computer Architecture. Academic Press, New York, 1975.

[11] Burks, A.W., H.H. Goldstine & J. von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. Report to U.S. Army Ordnance Department, 1946.

[12] Metropolis, N. & J. Worlton, A Trilogy of Errors in the History of Computing. First USA-JAPAN Computer Conference, pp. 683-691, 1972.

[13] Flynn, M.J., Very High-Speed Computing Systems. Proceedings of the IEEE, Vol. 54, No. 12, pp. 1901-1909, 1976.

[14] Goglardi, U.O., Report on Software Related Advances in Computer Hardware. Proceedings of a Symposium on the High Cost of Software. S.R.I. Menlo Park, California, 1973.

[15] Poel, W.L. van der, The Logical Principles of some Simple Computers. Excelsior, 's Gravenhage, 1956.

[16] Abrams, P.S., An APL Machine. Stanford University, SLAC-114, 1970.

ONTWERPREGELS VOOR VOOR MACHINE-ARCHITECTUREN

R. HOOGERWOORD
Technische Hogeschool Eindhoven

1. INLEIDING

Het doel van een rekenautomaat is het uitvoeren van programma's. Een programma is, in eerste instantie, een formele beschrijving van een relatie op twee verzamelingen: een verzameling parameters of invoerwaarden en een verzameling uitkomsten of uitvoerwaarden. Wanneer de relatie zó is dat iedere aan een invoerwaarde gerelateerde uitvoerwaarde een oplossing representeert van een probleem waarvan die invoerwaarde een parameter is, dan zeggen we ook dat het programma een oplossing van dat probleem beschrijft. Een tweede interpretatie van programma's is dat een programma beschrijft welke berekeningen verricht moeten worden om, gegeven een invoerwaarde, een uitvoerwaarde te verkrijgen zodanig dat het paar: (invoer, uitvoer) aan de door middel van het programma beschreven relatie voldoet. Een *rekenautomaat* is nu een machine die, gegeven een programma en een invoerwaarde, in staat is de door het programma voorgeschreven berekeningen uit te voeren. Op deze manier kunnen we rekenautomaten gebruiken om oplossingen voor problemen effectief te berekenen.

Omdat zowel programma's als rekenautomaten zeer complex samengesteld kunnen zijn, is een zeer voorzichtige benadering bij het ontwerpen ervan noodzakelijk, willen wij in staat zijn onze taak tot een succesvol einde te brengen. Een van de belangrijkste hulpmiddelen die ons ter beschikking staan bij het oplossen van ingewikkelde problemen is het beginsel van *separation of concerns* . Omdat nog te vaak tegen dit principe gezondigd wordt en om de noodzaak van consequente toepassing ervan te benadrukken zullen we hier in § 2 dieper op in te gaan.

Onder de *architectuur* van een machine verstaan we die eigenschappen van de machine die relevant zijn voor het gebruik ervan, ofwel: de *functionele* beschrijving van de machine. Hiernaast staat de beschrijving van de *realisatie* van de machine, die beschrijft hoe de functies van de machine tot stand komen. Deze tweedeling in de beschrijving is essentiëel: wanneer de gebruiker kennis van de realisatie nodig heeft om de machine te kunnen gebruiken wordt hij belast met onnodige details; het beginsel van separation of concerns is dan niet juist toegepast, terwijl dit wel mogelijk was. Verder blijkt dat het antwoord op de vraag wat tot de functionele aspecten en wat tot de realisatie van de machine behoort, afhangt van de mate van detail waarmee men de machine beschouwt: de realisatie vindt meestal plaats door de machine op te bouwen uit kleinere componenten, waarvan weer de functionele aspecten en de realisatie van elkaar gescheiden kunnen worden. Of men hierbij "top down" of "bottom up" te werk gaat lijkt nauwelijks van belang. Wel is essentiëel dat bij het aantonen van de correctheid van het geheel alleen de functionele aspecten van de delen gebruikt worden.

Programma's worden genoteerd in wat meestal een - "hogere" - *programmeertaal* genoemd wordt. Machines voor het verwerken van programma's worden vaak gerealiseerd door een machine te maken die alleen veel eenvoudiger genoteerde programma's kan verwerken en deze machine uit te rusten met een *vertaler* die programma's in de programmeertaal omzet in equivalente programma's in wat dan de *machinetaal* genoemd wordt. Dat de vertaler zelf vaak met behulp van dezelfde machine en een speciaal programma gerealiseerd wordt is een kwestie van uitvoering. Op de voordelen van deze aanpak gaan we in § 3 verder in. Voor de gebruiker van de machine is deze wijze van werken uiteraard niet van belang. Voor de realisatie van zo'n machine beschikken we dan over de vrijheid van de keuze van de machinetaal, maar dan ook over de plicht een verantwoorde keuze te maken. Zodra we ons met deze keuze bezighouden is de machinetaal voor ons de functionele beschrijving van de machine; de realisatie ervan zullen we hiervan gescheiden moeten houden. In §§ 4 en 5 zullen we enkele aspecten van het machinetaal ontwerp

toelichten terwijl we in § 6 zullen aantonen dat discussies over "registers" bij het ontwerp van een machinetaal geen rol hoeven te spelen maar pas bij de realisatie ervan relevant worden.

We beperken ons tot de bestudering van de klassieke "Von Neumann" machine in de meest algemene zin: deze bestaat uit twee componenten die meestal *processor* en *store* of *geheugen* genoemd worden. In plaats van de term "processor" zullen we ook de term *interpreter* gebruiken. Het geheugen is hierin een eindige, geordende verzameling identieke componenten die vaak *woorden* genoemd worden. Elke component kan een waarde bevatten en de componenten worden van elkaar onderscheiden door ze te nummeren; de gebruikte nummers heten *adressen* - de term "adres" zullen we uitsluitend reserveren voor het gebruik als nummer van een geheugencomponent, daarmee misverstanden over "logische" en "fysische" of "absolute" en "relatieve" adressen uitsluitend - . In het Von Neumann model is de processor een eindige automaat die een programma(text) in het geheugen interpreteert en bewerkingen op waarden in het geheugen uitvoert. De beperking tot *eindige* automaten als interpreters is, gezien de huidige stand van de techniek, niet strikt meer nodig; gebruik van een stapelautomaat als interpreter is, economisch gezien, heel goed mogelijk, al zullen realisaties ervan altijd eindig zijn. Von Neumann's opvatting dat het een voordeel zou zijn dat de machine zijn eigen programma tijdens uitvoering kan wijzigen is een misvatting gebleken en al lang achterhaald. Wij besteden er hier verder geen aandacht aan.

2. HET BELANG VAN "SEPARATION OF CONCERNS".

Laten we beginnen met toepassing van het principe van separation of concerns bij het beschouwen van machinebeschrijvingen. Als we de gebruikershandleiding of een andere beschrijving van een machine bestuderen, dan blijkt vaak dat een functioneel aspect van de machine uitgelegd wordt in termen van de realisatie. Als dit te wijten is aan de handleiding en een realisatieonafhankelijke be-

schrijving ook mogelijk is dan deugt de handleiding niet: ook van goed ontworpen machines wordt vaak een slechte beschrijving gegeven. Als echter een realisatieonafhankelijke beschrijving niet of slechts ten koste van onevenredig veel moeite te geven is, dan deugt het ontwerp van de machine niet. Het feit dat zowel handleidingen als machines wat dit betreft veel fouten bevatten maakt het beoordelen van machines aan de hand van hun handleidingen er niet gemakkelijker op.

In de inleiding hebben we al gezegd dat het belangrijk is onderscheid te maken tussen functie en realisatie van een machine. Een - oprechte - gebruiker heeft geen behoefte aan kennis van de realisatie en bij een goede machine kan deze kennis alleen leiden tot misbruik. Beter is een machine waarbij kennis van de realisatie nergens toe leidt en zelfs niet tot misbruik aanleiding kan geven: kennis van de realisatie mag de gebruiker niet meer mogelijkheden verschaffen dan hij op grond van de functionele beschrijving al ter beschikking heeft. Anderzijds is een machine die *zonder* kennis van de realisatie niet te gebruiken is ongewenst, om niet te zeggen: onbruikbaar. Dit geldt ook als we de machinetaal als functionele beschrijving nemen, hetgeen we aan het volgende voorbeeld kunnen illustreren:

Bij veel implementaties van ALGOL-achtige talen worden variabelen en procedure-paramters in een soort stapel ondergebracht. Dit is geen echte stapel, alleen het groeien en slinken vindt plaats zoals bij een stapel. Verder wordt voor het uitwerken van expressies vaak een stapel gebruikt. Beide stapels worden, hoewel functioneel verschillend, bij de realisatie vaak gecombineerd tot één stapel: de stapel, gebruikt voor het evalueren van expressies, wordt op de variabelen-stapel geplaatst, hierbij gebruikmakend van de - functionele - eigenschappen van beide stapels, die dit combineren toelaten. Hier is niets op tegen, maar de volgende manier van werken is misbruik van kennis van de realisatie: na evaluatie van een expressie bevindt de waarde zich op de stapel en omdat dit tevens de variabelenstapel is kan deze waarde ook als - locale - variabele geadresseerd worden. Hiervan wordt vaak gebruik gemaakt bij de

overdracht van procedureparameters in plaats van dat de machinetaal van speciale instructies voor parameteroverdracht voorzien is. De correcte werking van dit mechanisme berust nu op de - min of meer toevallige - eigenschappen van de twee gecombineerde stapels; realisaties waarin beide stapels niet gecombineerd zijn zijn nu ook uitgesloten: de combinatie maakt nu in feite deel uit van de architectuur, die hierdoor nodeloos ingewikkeld gemaakt wordt. Door functie en realisatie niet strikt te scheiden wordt het ontwerp van een machine onnodig gecompliceerd.

Een tweede voorbeeld van scheiding van functie en realisatie is de scheiding tussen de namen en waarden van variabelen enerzijds en de administratie ervan anderzijds. De toestandsruimte van een rekenproces wordt gekarakteriseerd met behulp van variabelen. De waarden van de variabelen tezamen vormen op elk moment de toestand van de berekening. In het programma worden de variabelen van elkaar onderscheiden met behulp van namen. Het maakt hierbij niet uit of we het programma gecodeerd denken in een programmeertaal of een machinetaal en wat voor soort namen gebruikt is. Anderzijds worden de waarden van de variabelen op een of andere manier ondergebracht in geheugencomponenten en dus gekenmerkt door adressen. De manier waarop kan van realisatie tot realisatie verschillen, zelfs van moment tot moment tijdens de uitvoering van het programma, en dien *dus* los te staan van de naamgeving in de programmatext. We maken daarom onderscheid tussen de waarden van de variabelen in het geheugen en alle andere informatie die de machine ten behoeve van de uitvoering in het geheugen bijhoudt; deze informatie zullen we de *administratie* noemen. Terwijl de namen en waarden der variabelen een rol spelen bij de functionele beschrijving is de administratie onderdeel van de realisatie. Verschillende realisaties van dezelfde taal kunnen verschillende administraties met zich meebrengen. Om dezelfde reden is het feit dat adressen getallen zijn en waarden van variabelen ook vaak getallen zijn geen aanleiding om adressen en waarden maar door elkaar te gooien en bijvoorbeeld bewerkingen op adressen in de (machine)taal toe te laten. Het feit dat de

interpreter voor de uitvoering van zijn taak vaak adressen moet uitrekenen staat hier los van.

Als laatste voorbeeld willen we onderscheid maken tussen objecten en hun representatie. Dit onderscheid geldt zowel voor programma's als voor waarden van variabelen. In het geheugen van de machine worden zowel programmainstructies als waarden van variabelen meestal voorgesteld door middel van bitrijtjes. Welke bitrijtjes met welke instructies corresponderen is een kwestie van de gekozen representatie: de machinetaal verandert niet als een andere representatie gekozen wordt. Door ontwerp van de taal en keuze van de representatie gescheiden te houden wordt weer vermeden dat de functionele beschrijving nodeloos gecompliceerd wordt en beschikt men bij de realisatie over de vrijheid die representatie te kiezen die het beste past bij de gekozen realisatievorm. Hierbij kunnen dan economische overwegingen - compactheid van de code, snelheid - een rol spelen. Evenzo is het voor de functie van de machine niet van belang of bijvoorbeeld gehele getallen gerepresenteerd worden in het tweetallig of het tientallig stelsel en hoeveel bits voor de representatie gebruikt worden. De "gebruiker" van de machinetaal, dit is: de maker van de vertaler, heeft ook geen belang bij een dergelijke overgespecificeerde beschrijving. Voor het bewijzen van de correctheid van de vertaler is een meer axiomatische definitie van de machinetaal veel geschikter ([1]).Het feit dat machines eindig zijn en dus niet alle gehele getallen representeerbaar zijn is een capaciteitsprobleem. De capaciteit van de machine is niet een aspect van de functie maar van de realisatie: verschillende realisaties kunnen verschillende capaciteiten hebben. Daarom rekenen we de representatie van waarden ook tot de realisatie: we kunnen ons twee verschillende realisaties van dezelfde machine(taal) voorstellen, de één waarin veel bits voor de representatie van getallen gebruikt worden, zodat vrij grote getallen representeerbaar zijn, en de andere waarin weinig bit gebruikt worden. Zolang een berekening in beide machines "past" zijn beide realisaties equivalent; de tweede

realisatie is - waarschijnlijk - sneller en/of goedkoper dan de eerste. We rekenen het dan ook tot de taak van de realisatie een melding te geven wanneer de capaciteit van de machine niet toereikend is voor een bepaalde berekening. Bij veel machines wordt een dergelijke melding gegeven in de vorm van een *fout*melding, maar dit is natuurlijk niet terecht: de gebruiker heeft niet aantoonbaar een fout gemaakt en de melding zou eerder een verontschuldiging moeten zijn: "sorry, deze machine is te klein voor Uw berekening". (afgezien van het feit dat sommige fouten zich als capaciteitsoverschrijdingen manifesteren).

3. HET WAAROM VAN VERTALERS.

Zoals in de inleiding al is opgemerkt worden machines voor het verwerken van programma's in een programmeertaal vaak gerealiseerd met behulp van vertalers die de programma's omzetten in een andere codering, die machinetaal heet. Deze manier van werken lijkt omslachtiger dan het bouwen van een machine die de programma's rechtstreeks, i.e.: zonder vertaling, uitvoert, maar er zijn voordelen aan verbonden, die er alle op neerkomen dat de machinetaal eenvoudiger gemaakt kan worden dan de programmeertaal; hierdoor is een efficiëntere verwerking mogelijk. Deze voordelen zijn:

.1 : De grotere *compactheid* van de in machinetaal gecodeerde programma's laat een efficiënter gebruik van het geheugen toe, wat zowel in de hoeveelheid benodigd geheugen als de snelheid tot uitdrukking komt.

.2 : De manier waarop in de programmeertaal namen voor variabelen gekozen mogen worden is prettig voor de programmeur: de verzameling namen vormt een *open* nomenclatuur, wat wil zeggen dat uit het vóórkomen van twee namen niets volgt over het voorkomen van tussenliggende namen. Dit is lastig bij de *adressering* , dit is het afbeelden van de namen op geheugenadressen. Voor de adressering is een zogenaamde *gesloten* nomenclatuur veel prettiger: een

verzameling namen noemen we gesloten als voor elk tweetal namen eruit ook alle tussenliggende (volgens de in die verzameling geldende ordening) namen in de verzameling zitten. Een rij opeenvolgende gehele getallen vormt bijvoorbeeld een gesloten naamgeving. De voordelen hiervan zijn dat:

.1: zo'n verzameling eenvoudig te karakteriseren is, bijvoorbeeld door middel van het kleinste en het grootste element, en dus eenvoudig te verifiëren is of een mogelijke naam ook in gebruik is;

.2: een compacte representatie van de namen mogelijk is, bijvoorbeeld met behulp van volgnummers;

.3: gesloten verzamelingen namen eenvoudig af te beelden zijn op geheugenadressen: beschouwd als namen vormen de adressen ook een gesloten nomenclatuur, waardoor een efficiënte verwerking mogelijk is.

Door nu in de machinetaal een - min of meer - gesloten naamgeving te gebruiken combineren we de voordelen van beide soorten naamgeving. De vertaler moet dan de namen uit de open naamgeving van de programmeertaal vervangen door namen uit de machinetaal.

.3: De betekenis van taalconstructies wordt altijd gedefinieerd aan de hand van hun opbouw. Om een programma te kunnen uitvoeren moet de interpreter dan ook een syntactische analyse maken van de programmatext. Omdat delen van het programma meestal herhaald uitgevoerd moeten worden, leidt dit tot het eveneens herhaald uitvoeren van de syntactische analyse van die delen. Omdat de syntaxis van programmeertalen vrij ingewikkeld is kan hier aan werk bespaard worden: door de syntactische analyse één keer te doen en en passant het programma in een syntactisch eenvoudigere machinetaal te coderen, kan de hoeveelheid analyseerwerk voor de interpreter aanzienlijk verminderd worden.

.4: Interpreters kunnen eenvoudiger en dus goedkoper zijn naarmate de geinterpreteerde taal een meer uniforme structuur heeft. In programmeertalen komen bijvoorbeeld mengvormen van pre-,

in- en postfix notatie voor, terwijl een meer uniforme notatie, bijvoorbeeld alleen postfix, eenvoudiger te interpreteren is.

4. NAAMGEVING VAN VARIABELEN.

Als elementaire bouwsteen van programma's beschouwen we de assignment statement. We beschouwen hier alleen assignment statements waarin aan één variabele een waarde wordt toegekend. Welke variabele dit is wordt aangegeven door middel van de *naam* van die variabele. Welke waarde wordt toegekend wordt gespecificeerd met behulp van een expressie waarin, onder andere, namen van variabelen voor kunnen komen. Zoals in de vorige paragraaf al aangegeven is laten de meeste programmeertalen de gebruiker een grote vrijheid in de keuze van de gebruikte namen: door in een programma alle gebruikte namen door andere te vervangen ontstaat een nieuw programma dat semantisch equivalent is aan het eerste. Deze vrijheid van keuze hebben we in de machinetaal niet nodig; een uniforme, meer gesloten naamgeving laat bovendien een efficiëntere realisatie toe. In deze paragraaf zullen we ons met de keuze van een dergelijke naamgeving en de realisatie ervan bezighouden. Hierbij veronderstellen we dat declaraties van variabelen genest voorkomen zoals dat in ALGOL-achtige talen het geval is (block structuur). De naamgeving in de machinetaal moet dan zó zijn dat enerzijds de vertaler gemakkelijk namen kan toewijzen en anderzijds de interpreter deze namen gemakkelijk kan afbeelden op geheugenadressen. Omdat in de programmeertaal namen vrijelijk gekozen kunnen worden kan de vertaler geen gebruik maken van de feitelijk gebruike namen: het lijkt wenselijk om programma's die alleen in de gebruikte namen verschillen maar overigens gelijk zijn op dezelfde manier in de machinetaal om te zetten. De enige andere informatie die de vertaler over de variabelen heeft is de plaats van hun voorkomen in de programmatext, in het bijzonder de plaats van hun declaraties. Voorlopig veronderstellen we dat voor de opslag van de waarde van een variabele precies één geheugencomponent nodig is. Bijvoorbeeld:

.1: Nummer de variabelen naar voorkomen van hun declaratie in de text. Alle in het programma voorkomende variabelen krijgen dan verschillende namen. Voor de adressering kunnen we dan de identieke afbeelding gebruiken: de waarde van variabele nr. i brengen we onder in de geheugenlocatie met adres i. Omdat echter niet alle variabelen gelijktijdig "bestaan" is de zo ontstane naamgeving niet gesloten, wat als gevolg heeft dat, bij de gekozen adressering, wat kwistig met het geheugen omgesprongen wordt.

.2: Beter is dan de volgende naamgeving: geef iedere variabele als naam het laagste vrije nummer. Bij afsluiting van een block door de vertaler komen de in dat block gebruikte namen weer vrij. Voor de adressering gebruiken we, als in .1, i, de identieke afbeelding. Op deze manier krijgen we een gesloten naamgeving en een overeenkomstige geheugenbenutting. Wanneer de programmeertaal alleen geneste blocks en geen procedures bevat dan werkt dit mechanisme feilloos. Als de taal procedures bevat met de restrictie dat in procedures alleen aan locale variabelen - en parameters - en niet aan globale variabelen gerefereerd wordt dan kunnen we alle locale variabelen van één procedure benoemen door ze vanaf 0 te nummeren. Iedere procedure heeft dan immers zijn eigen verzameling variabelen, disjunctvan alle andere. Zij nu l het kleinste adres van de, bij aanroep van de procedure, niet in gebruik zijnde geheugenplaatsen, dan levert: adres = l + nummer een efficiënte adressering op. Deze optelling dient, tijdens uitvoering, door de processor gedaan te worden; l is dan een hulpvariabele van de processor. Het is duidelijk dat dit schema een gesloten nomenclatuur oplevert. Het volgende voorbeeld illustreert dat er problemen ontstaan als in procedures wel aan globale variabelen gerefereerd mag worden; de gebruikte notatie wordt geacht voor zich te spreken, de indices aan de namen der variabelen geven de toegewezen nummers aan:

```
var  a0, b1, c2;
procedure p;  var  d3, e4, f5;
   begin .... end p;
procedure q;  var  g3, h4;
   begin .... { * }  p .... end q;

begin .... q .... end.
```

Het voorgestelde mechanisme zou dan moeten leiden tot de volgende situaties:

	de toestands-ruimte bevat de variabelen:	op adressen:
.1: vlak voor de aanroep van p bij { * } :	a,b,c,g,h.	0,1,2,3,4
.2: aan het begin van de body van p :	a,b,c.	0,1,2
.3: in de body van p :	a,b,c,d,e,f	0,1,2,3,4,5
.4: na terugkeer uit p :	a,b,c,g,h	0,1,2,3,4

Na uitvoering van de procedure p bevat de toestandsruimte weer de variabelen g en h met de waarden die zij vóórde aanroep van p hadden. De gegeven adressering werkt nu niet meer correct: geheugenlocaties 3 en 4 worden nu zowel voor de variabelen g en h gebruikt als voor d en e . We kunnen dit conflict oplossen door naar behoefte, dit is: bij de declaratie van d en e , de waarden van g en h te "redden" door ze te copiëren naar een tijdelijke bewaarplaats. De variabelen g en h zijn immers tijdens de uitvoering van p toch niet toegankelijk. Vóór terugkeer uit p moeten de geredde waarden dan teruggebracht worden. Hoewel vanuit complexity-standpunt aanvaardbaar verwerpen we deze manier van werken: het red- en herstelwerk is non-productief, want het draagt niet direct bij tot de voort-

gang van de berekening. Merk nu op dat de naamgeving zelf nog steeds correct is, alleen de adressering levert een probleem op. Door bij dezelfde naamgeving een andere adressering te kiezen krijgen we:

.3: Kies als adres voor een variabele weer het kleinste vrije adres. In het voorbeeld geldt dan in de body van p na aanroep vanuit q :

de variabelen:	bevinden zich op adressen:
a_0, b_1, c_2,	0, 1, 2,
g_3, h_4,	3, 4,
d_3, e_4, f_5	5, 6, 7.

waarbij alleen a, b, c, d, e en f toegankelijk zijn. De verzameling toegankelijke stukjes consecutief geheugen bestaat uit k verschillende stukjes, als k het heersende "lexicografische nivo" is. De interpreter moet nu in staat zijn bij elke naam te bepalen welk adres er op elk moment bijhoort; in het voorbeeld: bij variabele nummer: 3 hoort adres 5 - in de body van p - of adres 3 - in de body van q -. Dit kan gerealiseerd worden met behulp van een tabel die per block-nivo een paar (n,a) bevat, waarin n het nummer en a het adres van de eerste variabele uit dat block is. Door de tabel gesorteerd op te slaan kan de interpreter met behulp van een "binary search" het geheugenstukje identificeren waarin zich een variabele bevindt. Ook kan men denken aan de toepassing van een stukje associatief geheugen. Wat we nu in feite aan het doen zijn is de processor bij elk access aan een variabele laten uitzoeken wat het lexicografische nivo van die variabele is. Dit is echter iets wat de vertaler ook kan doen, en dan eenmalig! Dan krijgen we:

.4: Per referentie aan een variabele noteren we in de code wat zijn nivo is. Dan kunnen we ook volstaan met in ieder block de variabelen vanaf 0 te nummeren. Variabelen uit verschillende blokken worden nu immers van elkaar onderscheiden door hun nivo-aanduiding. De namen der variabelen zijn nu

dus paren getallen, waarin de verzameling nivonummers gesloten is en, per nivo, de verzameling variabelennummers gesloten is. De nomenclatuur als geheel is nu niet gesloten. Toch is nu een efficiënte realisatie mogelijk: van de in .3 genoemde tabel paren hebben we alleen de rechterleden nodig. In deze tabel kan nu direct geindiceerd worden. Traditioneel heet deze tabel: *display* . Verificatie of een paar getallen een geldige naam is is eenvoudig als we bijhouden wat het lexicografisch nivo is en wat, voor elk nivo, het aantal variabelen in dat nivo is.

Zoals we in bovenstaande voorbeelden kunnen zien wordt de keuze van de naamgeving in de machinetaal beïnvloed door de eigenschappen van de programmeertaal enerzijds en, vooral in .4, door eisen van realiseerbaarheid anderzijds. Tevens blijkt dat het streven naar een min of meer gesloten nomenclatuur een middel is om op vrij systematische wijze tot een efficiënte realisatie te komen.

Wanneer voor het bergen van de waarde van een variabele een aantal geheugenelementen nodig is, per variabele mogelijk verschillend maar vast gedurende de gehele "levensduur" van die variabele, dan is de verzameling adressen van variabelen niet meer gesloten, dan wel op te delen in een aantal gesloten stukjes. Als we nu dezelfde adressering als in .4 gebruiken dan is ook de naamgeving per nivo niet meer gesloten: In plaats van de gesloten nomenclatuur: (1,0), (1,1), .., (1,n-1) gebruiken we de open nomenclatuur: $(1,a_0)$, .., $(1,a_{n-1})$, met: $a_0 = 0$ en $a_{i+1} = a_i + l_i$, als l_i het aantal voor de opslag van variabele i benodigde geheugenelementen is. Nadelen hiervan zijn de geringere codeerefficiëntie omdat, vooral als de l_i 's grote waarden aan kunnen nemen, grotere nummers in de namen voorkomen - zonder dat ze meer informatie bevatten - en het feit dat namen niet meer op geldigheid geverifiëerd kunnen worden. We kunnen dit oplossen door de adressen a_i in een tabelletje te zetten,

waardoor de naamgeving gesloten blijft: In plaats van:

```
adres(l,i)= display(l) + i, krijgen we dan:
adres(l,i)= a(display(l) + i).
```

De elementen van a noemen we *descriptors* . Omdat er per variabele een descriptor is, kunnen we er, naast het adres, ook andere administratieve informatie over de variabelen in opnemen, zoals het type of het feit dat de variabele nog niet geinitialiseerd is. De prijs van deze oplossing is een extra geheugenaccess per access van een variabele, de opbrengst een robuuster mechanisme en compactere code.

5. HET INPERKEN VAN COMBINATORISCHE VRIJHEID.

Wanneer we de instructiecodes van bestaande computers bestuderen dan is het eerste wat ons opvalt het gebrek aan structuur in zulke codes. Zo'n instructiecode is immers meestal gedefinieerd als een verzameling *instructies* en een programma wordt dan gedefinieerd als een - willekeurige - *rij* instructies. Dit suggereert dat programma's gemaakt kunnen worden door maar een willekeurig rijtje instructies te nemen; wanneer we echter wat nauwkeuriger kijken dan ontdekken we dat er een heleboel rijen instructies bestaan die niet op een zinnige manier als programma's opgevat kunnen worden; feitelijk blijkt de meerderheid van al deze rijen betekenisloos te zijn. Verder blijkt dat, wanneer we een algoritme ontwikkeld hebben en we proberen dit te coderen als een rij instructies, dit vaak op meer dan één manier gedaan kan worden. We zeggen dan ook dat de meeste instructiecodes de gebruiker te veel *combinatorische vrijheid* bieden: te veel combinaties van instructies zijn ofwel nutteloos ofwel redundant en deze eigenschap maakt het correct coderen van programma's moeilijker dan nodig is.

Vanuit een meer taalkundig standpunt zien we dat de taal van alle instructierijtjes beschreven kan worden met behulp van zeer eenvoudige grammatica's - bijvoorbeeld: rechts lineaire grammatica's - ; bovendien is de betekenis van iedere instructie

gedefinieerd in termen van de toestand van de machine en onafhankelijk van de andere instructies, dat wil zeggen: alle instructies zijn semantisch onafhankelijk van elkaar. Daarom kan een eenvoudige, eindige automaat als interpreter gebruikt worden voor het realiseren van zo'n instructiecode en in de begindagen van de rekenautomaten was deze eenvoud een voorwaarde voor de realiseerbaarheid van een machine. Tegenwoordig kunnen we echter, met technieken als LSI en micro-programmering tot onze beschikking, ingewikkelder interpreters op een economisch verantwoorde manier bouwen. Het beperken van combinatorische vrijheid komt dan neer op het inperken van de verzameling instructierijtjes en de meest voor de hand liggende manier om dit te doen lijkt het gebruiken van "krachtiger" grammatica's zoals contextvrije grammatica's. Operationeel gezien betekent dit dat we in plaats van een eindige automaat een stapelautomaat als interpreter kunnen gebruiken. Door hierbij naar een zo eenvoudig mogelijke taal te streven proberen we onze machine niet al te duur te maken.

Het volgende voorbeeld illustreert het voorgaande zo overtuigend dat we het niet willen achterhouden:

Veronderstel dat we een stapelmachine hebben met in zijn instructierepertoire de volgende subset:

```
< instruction > → const    < value >
< instruction > → var      < variablename >
< instruction > → add
< instruction > → mult
< instruction > → store    < variablename >
```

dan kan de assignment statement: x:= 2 * x + y * z
op de volgende manier gecodeerd worden:

```
const    2
var      x
mult
var      Y
var      z
mult
add
store    x
```

en dit instructierijtje representeert dan een zinnig stukje programma. Er kunnen echter ook een heleboel on-zinnige rijtjes instructies geconstrueerd worden; in feite zo gemakkelijk dat we er geen voorbeelden van zullen geven: de lezer kan dit zelf als hij daarin geinteresseerd is. Wanneer we de gegeven instructies alleen gebruiken voor het coderen van expressies en assignments dan kan de verzameling feitelijk gebruikte instructierijtjes ook beschreven worden met behulp van de volgende regels:

< assignment > → < expression > store < variablename >
< expression > → const < value >
< expression > → var < variablename >
< expression > → < expression > < expression > < operator >
< operator > → add
< operator > → mult

Op deze manier wordt het aantal "toegelaten" instructierijtjes aanzienlijk gereduceerd en wanneer de machine nu iedere assignment interpreteert volgens deze regels zal ieder non-sens rijtje ontdekt worden. Hoe aanzienlijk deze beperking is wordt nog treffender gedemonstreerd wanneer we wat telwerk verrichten. Bijvoorbeeld: wanneer we "const < value >" als één instructie tellen ongeacht de gebruikte waarde, en we doen hetzelfde met "var" en "store" dan is het aantal instructierijtjes ter lengte 8 wanneer we de eerste set regels gebruiken gelijk aan 390625 ; bij gebruik van de tweede set regels wordt dit aantal teruggebracht tot 640 .

Het probleem is nu de machinetaal zo te ontwerpen dat deze enerzijds zo beperkt mogelijk is en dat deze anderzijds zo eenvoudig mogelijk beschreven kan worden. Eenvoud kan bereikt worden door een uniforme structuur te kiezen die op alle aspecten van de taal betrekking heeft: het zijn de uitzonderingen die een taalbeschrijving ingewikkeld maken. Laat ons daarom het volgende voorstel bekijken:

In klassieke machines zijn de instructies meestal samengesteld uit een zogenaamde "operatiecode" en een aantal "operanden" ; dit kunnen we syntactisch als volgt beschrijven:

< instruction > → < operationcode > { < operand > }

Deze regel geldt voor alle instructies - uniformiteit! - , waardoor de realisatie eenvoudiger wordt. De operatiecode vervult hierin twee functies: behalve als aanduiding van de uit te voeren bewerking werkt de operatiecode als een prefix waarvan de interpretatie der operanden afhankelijk is. Deze eigenschap is belangrijk omdat zo'n prefix in feite een context vastlegt waarbuiten operanden betekenisloos zijn. Wanneer we dit schema nu generaliseren door toe te laten dat operanden zelf opgebouwd kunnen zijn uit prefixen en andere operanden verkrijgen we een beschrijving die nog steeds uniform is maar meer structuur verschaft. Iedere prefix definieert dan zowel een uit te voeren bewerking als een context voor zijn operanden, en het is deze laatste eigenschap die het mogelijk maakt de gewenste inperking van de combinatorische vrijheid te verkrijgen. Deze benadering past goed bij de structuur van onze programma's waarin immers hetzelfde geldt: expressies hebben bijvoorbeeld alleen betekenis in de context van assignments en mogelijk wat andere constructies; optellingen en vermenigvuldigingen zijn vervolgens alleen zinvol in de context van expressies, etc. Bijvoorbeeld:

< assignment > → assign < variablename > < expression >
< expression > → const < value >
< expression > → var < variablename >
< expression > → < operator > < expression > < expression >
< operator > → add
< operator > → mult

Deze verzameling regels heeft de genoemde eigenschappen en is "semantische equivalent" met de verzameling regels uit het vorige voorbeeld. De syntactische verschillen met het vorige voorbeeld zijn gering; terwijl in het vorige voorbeeld een mengsel van prefix en postfix code gebruikt werd bevatten deze regels alleen prefix code.

De soort machinetaal die we hier voorstellen zullen we aanduiden met *gegeneraliseerde prefix code* omdat we het klassieke begrip

"instructie" gegeneraliseerd hebben en deze instructies in feite prefix codes zijn. Het gebruik van termen als "instructie" of "instructie code" is nu niet erg zinvol meer vanweg de geintroduceerde recursie. In plaats van over "instructies" kunnen we nu beter over "statements", "expressions" of "operators" spreken, afhankelijk van de mate van detail die we wensen te bespreken. Gegeneraliseerde prefix code kan geinterpreteerd worden met behulp van een zogenaamde stapelautomaat van dezelfde vorm als de zogenaamde "top-down parser" die in vertalers voor programmeertalen gebruikt wordt. De stapel is ook nodig voor het vasthouden van informatie over operatoren die nog niet uitgevoerd kunnen worden omdat de bijbehorende operanden nog geëvalueerd moeten worden; een proces dat nu recursief is geworden.

Hoewel we alleen voorbeelden gegeven hebben over het coderen van assignments en expressies kunnen we ook de "sequencing" op deze manier behandelen. Onder sequencing verstaan we de selectie van de volgende uit te voeren operatie, in het bijzonder wanneer er alternatieven zijn en er een keuze gemaakt moet worden. Klassieke machinetalen bevatten hiertoe speciale instructies die de "besturing" van de machine expliciet "overdragen" aan een ander deel van het programma. Dergelijke instructies zijn nauwelijks op een realisatieonafhankelijke manier te beschrijven en bieden, vanwege het onbeperkte gebruik dat ervan gemaakt kan worden, veel meer combinatorische vrijheid dan wenselijk is. Het gebruik van gegeneraliseerde prefix code biedt hiervoor een oplossing, maar we gaan hier verder niet op in (zie: [2]).

6. DE ROL VAN REGISTERS.

Een *register* is een speciale component die in staat is een waarde te bevatten, hoewel het geen onderdeel is van het geheugen. Tot nu toe hebben we niet over registers gesproken omdat daar geen noodzaak toe was. Dit is niet verwonderlijk want we hebben onze aandacht voornamelijk gericht op de functionele beschrijving van de machine en deze functionele beschrijving staat in nauw verband

met de semantiek van onze programma's, waarin het begrip registers helemaal niet voorkomt. We concluderen daarom dat het geen zin heeft registers te introduceren in de functionele beschrijving van de machine. In deze paragraaf zullen we nagaan wat de rol van registers is bij de realisatie van de machine.

Allereerst merken we op dat enerzijds het geheugen van de machine zodanig is dat van slechts één component tegelijkertijd de waarde geinspecteerd of gewijzigd kan worden en anderzijds voor de realisatie van de meeste operaties gelijktijdige toegang tot meer dan één geheugencomponent nodig is. Dit conflict kan alleen opgelost worden door registers te gebruiken waarin de voor het uitvoeren van een operatie benodigde waarden tijdelijk opgeborgen kunnen worden, dat wil zeggen zolang als nodig is om de operatie te kunnen voltooien. Op deze manier gebruikt werken registers dus als *buffers* tussen het geheugen en de processor. Typerend voor dit soort registergebruik is dat de informatie in dergelijke registers zeer frequent wijzigt en alleen interpreteerbaar is binnen de contex van de op dit moment uitgevoerde operatie: deze registers worden in feite gebruikt als "zeer-locale" variabelen van de interpreter, nodig voor het realiseren van de operaties. Voorbeelden van zulke registers zijn: adres- en dataregisters voor het realiseren van geheugenaccessen, accumulator- en operandregisters voor het realiseren van rekenkundige bewerkingen en registers voor in- en uitvoer operaties. Merk op dat in klassieke machines *sommige* van deze registers anoniem blijven, maar andere niet, zoals expliciet benoembare accumulators. We concluderen dan ook dat bij zulke machines de scheiding tussen functie en realisatie niet consequent is doorgevoerd.

Een tweede toepassing van registers is voor het verkrijgen van een efficiëntere realisatie. Het aantal geheugenaccessen dat de machine in een bepaalde hoeveelheid tijd kan uitvoeren is namelijk begrensd. Wanneer nu de snelheid van de processor zo hoog is dat deze grens bereikt wordt dan wordt de snelheid van de gehele machine bepaald door die van het geheugen. Door nu de machine uit te rusten met wat registers waarin de meest frequent geaccesseerde informatie, of copieën daarvan, wordt opgeslagen kan het aantal geheugenaccessen

aanzienlijk verminderd worden, waardoor de snelheid van de machine vergroot kan worden zonder dat de investering in een sneller geheugen nodig is. De winst in snelheid kan aanzienlijk zijn, ten eerste omdat er meestal zeer frequent geaccesseerde informatie is en ten tweede omdat registers meestal sneller zijn dan en parallel kunnen werken met het geheugen. Het is duidelijk dat dergelijke overwegingen, hoe zinvol en effectief ze ook mogen zijn, voor de functionele beschrijving van de machine niet relevant zijn, maar thuis horen bij de realisatie ervan. Voorbeelden van op deze wijze gebruikte registers zijn: de programmawijzer - meestal ten onrechte programma*teller* genoemd - , registers die de eerste vrije locatie in een stapel aangeven, meer in het algemeen: registers voor administratie van het geheugengebruik en de bovenste paar elementen van de voor het evalueren van expressies gebruikte stapel, wanneer we ontdekt hebben dat het merendeel der expressies erg kort is. Optimaliseringen van het laatste soort dienen echter met grote voorzichtigheid uitgevoerd te worden: de gebruiker van zo'n machine mag niet in de verleiding gebracht worden lange expressies op te splitsen in meerdere korte vanwege de snelheidswinst die dat op zou leveren; kennis van de realisatie beinvloedt dan het gebruik van de machine en leidt tot conflicten die beter vermeden kunnen worden.

7. SLOTOPMERKINGEN.

De vrij strenge scheiding tussen functie en realisatie heeft ertoe geleid dat nogal wat aspecten van het machineontwerp naar het rijk van de realisatie verwezen zijn. De bedoeling hiervan is niet om ons ervan af te maken: het is een gevolg van consequente toepassing van het beginsel van separation of concerns. Het ontwerp van een goede machinetaal, los van allerlei realisatiebeschouweingen, blijkt al moeilijk genoeg te zijn. Discussies in de literatuur, zoals die in de SIGARCH-newsletters ([3]) over het gebruik van registers of stapels, raken dan ook hopeloos verward doordat aspecten van de functie en van de realisatie door elkaar gehaald worden zonder dat men zich daarvan bewust lijkt te zijn

LITERATUUR

[1] : C. Hemerik, "Relaties tussen taaldefinitie en taalimplementatie", in: "Colloquium capita implementatie van programmeertalen", MC syllabus 42, Mathematisch Centrum (1979).

[2] : R. Hoogerwoord, "On the design of instruction codes", afstudeerverslag, Technische Hogeschool Eindhoven (1979).

[3] : J.L. Keedy, "More on the Use of Stacks in the Evaluation of Expressions", in: "Computer Architecture News", Vol. 7, nr. 8, newsletter of the SIGARCH of the ACM (juni 1979, en vorige uitgaven).

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.6b B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, *Leergang Besliskunde, deel 6b: Wiskundige programmering 2*, 1977. ISBN 90 6196 150 5.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium Stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium Stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968.
ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969.
ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970.
ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970.
ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971.
ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 x.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975.
ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra , deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.

MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.

MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 144 0.

* MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHARST, *Efficiency begrippen in de statistiek*, . ISBN 90 6196 149 1.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.

MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, 1977. ISBN 90 6196 138 6.

MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.

MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.

MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.

MCS 37 J.C. van Vliet (red.), *Colloquium Capita Datastructuren*, 1978. ISBN 90 6196 159 9.

MCS 38.1 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 161 0.

MCS 38.2 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 181 5.

MCS 39 O.J. VRIEZE & G.L. WANROOY, *Colloquium Stochastische Spelen*, 1978. ISBN 90 6196 167 X.

MCS 40 J. VAN TIEL, *Convexe Analyse*, 1979. ISBN 90 6196 187 4.

MCS 41 H.J.J. TE RIELE (ED.), *Colloquium Numerical Treatment of Integral Equations*, 1979. ISBN 90 6196 189 0.

MCS 42 J.C. VAN VLIET (RED.), *Colloquium Capita Implementatie van Programmeertalen*, 1980. ISBN 90 6196 191 2.

MCS 43 A.M. COHEN & H.A. WILBRINK, *Eindige groepen (Een inleidende cursus)*, 1980. ISBN 90 6196 203 X

MCS 44 J.G. VERWER (ED.), *Numerical solution of partial differential equations*, 1980. ISBN 90 6196 205 6.

MCS 45 P. KLINT (red.), *Colloquium hogere programmeertalen en computerarchitectuur*, 1980. ISBN 90 6196 206 4.

De met een * gemerkte uitgaven moeten nog verschijnen.