EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Architecturen voor parallel processing

Vrijsen, A.J.M.

Award date:
1988

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

EB 166

TECHNISCHE UNIVERSITEIT EINDHOVEN
Faculteit der Elektrotechniek

ARCHITECTUREN VOOR PARALLEL PROCESSING

door: Ing. A.J.M. Vrijsen

afstudeerverslag

Vakgroep: Digitale Systemen (EB)
Afstudeerhoogleraar: Prof.ir. M.P.J. Stevens
Begeleider: Ing. P.H.A. van der Putten
Periode: maart - december 1988

SAMENVATTING

Er zijn twee methoden waarmee de performance (snelheid) van een digitaal systeem vergroot kan worden, namelijk het sneller maken van de onderdelen van het systeem, en het er voor zorgen dat de onderdelen gelijktijdig kunnen werken in plaats van na elkaar (parallel processing).

Dit verslag beschrijft de resultaten van een afstudeeronderzoek naar de architecturen die gebruikt worden voor parallel processing. Hierbij is aandacht besteed aan de besturingsfunkties van het systeem. Het doel van de studie is het verkrijgen van operationele kennis t.b.v. het ontwerpen van ASIC's.

Voor de classificatie van de diverse architecturen en technieken is getracht aansluiting te vinden bij de methode van Flynn (SISD,SIMD,MISD en MIMD).

De onderzochte architecturen zijn:

- Uniprocessor;
- Multiprocessor;
- Dataflow processor;
- Systolisch array;
- Wavefront array.

Er wordt een gedetailleerde beschrijving gegeven van pipelining, een techniek die op de een of andere manier in bijna alle architecturen voor parallel processing gebruikt wordt.

Bij het ontwerpen van een (parallel) digitaal systeem geldt dat een zorgvuldige afweging gemaakt moet worden van de voor- en nadelen van de mogelijke architecturen. Een aantal aspecten die een rol spelen bij de afwegingen worden behandeld, zoals:

- haalbare performance;
- omvang van de hardware;
- algemene toepasbaarheid.

INHOUDSOPGAVE

1. INLEIDING 4

2. CLASSIFICATIE PARALLEL PROCESSING 6

3. UNIPROCESSOREN 9
   3.1. Inleiding 9
   3.2. Externe busstructuren 9
   3.3. Interne busstructuren 11

4. MULTIPROCESSOREN 15
   4.1. Inleiding 15
   4.2. Time-shared / gemeenschappelijke bus systemen . 17
   4.3. Crossbar switch systemen 19
   4.4. Gedistribueerde systemen 20

5. PIPELINE PROCESSING 24
   5.1. Inleiding 24
   5.2. Arithmetische pipelines 30
   5.3. Instructie pipelining 36
   5.4. Reduced Instruction Set Computers 41

6. DATAFLOW COMPUTERS 43
   6.1. Inleiding 43
   6.2. Dataflow programma's 44
   6.3. Pipelining in dataflow programma's 47
   6.4. Dataflow architecturen 50
   6.5. Dataflow multiprocessoren 52
   6.6. Token labelling 55

7. SYSTOLISCHE ARRAYS 58
   7.1. Inleiding 58
   7.2. Semi- en zuiver-systolische arrays 59
   7.3. Eigenschappen van systolische arrays 62
   7.4. De besturing van systolische arrays 63

8. WAVEFRONT ARRAYS 68

9. VECTOR PROCESSING 73

10. VOORBEELD VAN EEN COMBINATIE VAN ARCHITECTUREN . . . 76

11. EEN OVERZICHT VAN ARCHITECTUREN 80
    11.1. Inleiding 80
    11.2. Array processoren 80
    11.3. Multiprocessoren 81
    11.4. Dataflow computers 81
    11.5. Systolische arrays en wavefront arrays 82
    11.6. Tradeoffs 84
    11.7. Van specificatie naar architectuur 86

12. CONCLUSIES 89

SLOTWOORD 90

LITERATUURLIJST 91

FIGUREN

1. Classificatie volgens Flynn 6
2. Uniprocessor 9
3. Gemodificeerde Harvard architectuur 10
4. Datapad met enkele busstructuur 11
5. Datapad met dubbele busstructuur 12
6. Datapad met drievoudige busstructuur 12
7. Multiprocessor systeem 15
8. Multiprocessor met identieke processoren 16
9. Multiprocessor systeem met gemeenschappelijke bus . . . 17
10. Multiprocessor systeem met multiple bus 18
11. Multiprocessor systeem met crossbar switch 19
12. Gedistribueerd systeem met volledige interconnecties . . 20
13. Gedistribueerd systeem met n-dimensionale hypercube structuur 21
14. Gedistribueerd systeem met mesh structuur 22
15. Gedistribueerd systeem met cluster structuur 22
16. Pipeline principe 24
17. Executie van 5 taken op de pipeline van fig.16 24
18. Snelheidswinst als functie van het aantal taken, met het aantal aktiviteiten als parameter 26
19. Snelheidswinst voor n aktiviteiten bij een bottleneck . 27
20. Latency en throughput als functie van het aantal segmenten 29
21. Pipelined floating point add/subtract unit 31
22. Carry-save adder 33
23. Pipelined Wallace tree multiplier 34
24. Gemodificeerde Wallace tree 35
25. Pipelined executie van instructies 36
26. Memory access conflicten bij pipelining 37
27. Vermindering van conflicten door Harvard architectuur . 38
28. Drie segmenten van een instructie pipeline 39
29. Data dependency bij pipelining 39
30. Branch dependency bij pipelining 40
31. Vertraagde spronginstructie 41
32. Vertraagde spronginstructie (4) met NOP-instructie (*) . 41
33. Control flow versus data flow 44

34. Dataflow graph van het programma uit fig.33 45
35. Representatie met activity templates 46
36. Activity templates met acknowledgements 48
37. Dataflow graph van een programmalus 48
38. Implementatie van de lus van fig.37 met templates . . . 49
39. Een dataflow processing element 51
40. Een dataflow multiprocessor 53
41. Aansluiting van een processing element op het netwerk . 53
42. Besturing door de host computer 54
43. Een programmalus met token labelling 55
44. Een processing element voor token labelling 56
45. Conventioneel computersysteem versus systolisch array . 58
46. Systolisch array met broadcasted inputs 60
47. Systolisch array met accumulatoren 60
48. Systolisch array met fan-in 61
49. Zuiver-systolisch array 62
50. Implementatie van een systolisch array 63
51. Architectuur van een systolische processor 64
52. Wavefront array voor matrix vermenigvuldiging 70
53. Wet van Amdahl (snelheidswinst als functie
van de niet-vectoriseerbare fractie) 75
54. Computational Element (CE) van de Alliant FX/8 77
55. FX/8 Multiprocessor of Computational Complex 78
56. Array processor 80
57. Tradeoff van kosten tegen performance 86
58. Van specificatie naar architectuur 87

1. INLEIDING

Er is een steeds groeiende behoefte aan snellere digitale systemen, speciaal in die toepassingen waar "real-time" bewerkingen nodig zijn. Traditionele computers gebaseerd op het Von Neumann concept[1] kunnen niet langer aan die behoefte voldoen.

Conventionele methoden om een krachtiger digitaal systeem te maken, trachten de snelheid op te voeren door de componenten van het systeem sneller te maken. De verwachtingen voor de toekomst zijn dat deze methode spoedig een limiet zal bereiken. Dat er een fundamentele limiet bestaat is duidelijk: geen enkel signaal kan zich met een snelheid groter dan de lichtsnelheid verplaatsen.

Naast het sneller maken van de componenten van het systeem is het echter ook mogelijk de afzonderlijke componenten parallel (gelijktijdig) te laten werken in plaats van sequentieel, of meerdere systemen parallel te laten werken. In dat geval spreekt men van "parallel processing". VLSI[2] maakt de implementatie van parallelle systemen mogelijk die tot voor kort niet te realiseren waren.

De snelheid (throughput) van een digitaal systeem kan gedefinieerd worden als het aantal resultaten dat het systeem per tijdseenheid produceert. Naast de throughput zijn er echter nog meer factoren die de kwaliteit (performance) bepalen, zoals:

- betrouwbaarheid (reliability);
- beschikbaarheid (availability);
- flexibiliteit (het geschikt te maken zijn voor meerdere toepassingen).

In dit verslag zullen we met performance echter hoofdzakelijk throughput bedoelen. Bovendien is het van belang of het systeem

---

[1] Seriële computer met één enkele processor

[2] Very Large Scale Integration

eenvoudig uit te breiden is, in het geval dat er een grotere throughput nodig is.

Afhankelijk van de toepassing kan het gewenst zijn dat een systeem:

- een grote throughput heeft;
- goedkoop is;
- een gunstige prijs/prestatie heeft.

Een conventioneel digitaal systeem bestaat uit een datapad en een besturing (control unit). De vraag is nu hoe, uitgaande van deze "bouwblokken", een architectuur te ontwerpen is die efficiënt gebruik maakt van parallel processing. In het bijzonder is van belang welke architectuur het best geschikt is voor een bepaalde applicatie.

Om deze vragen te kunnen beantwoorden (hetgeen de doelstelling is van dit onderzoek) is het nodig te weten wat de besturingsfunkties zijn van een architectuur en waar de bottlenecks zitten bij die bepaalde toepassing.

2. CLASSIFICATIE PARALLEL PROCESSING

Er zijn in de literatuur diverse classificatieschema's te vinden, die gebruikt kunnen worden om architecturen en/of technieken in groepen onder te verdelen, zie bijv. [BAS87][3]. Wij zullen hier de methode van Flynn [RAF88 pp.306,307] als uitgangspunt gebruiken. Deze methode is weliswaar niet zo eenduidig als vele andere, maar die andere methoden hebben het nadeel dat ze eigenlijk meer individuele machines classificeren dan algemene principes. Het aantal klassen is vaak groter dan het aantal algemene principes. Daarbij komt nog dat de methode van Flynn algemeen bekend is en vaak wordt gebruikt.

Een processor executeert instructies door manipulaties uit te voeren op de in het geheugen opgeslagen data. De instructies zijn óók in het geheugen opgeslagen, en de verplaatsing van instructies is altijd van geheugen naar processor (de situatie dat bij het compileren van een programma instructies in het geheugen worden opgeslagen laten we buiten beschouwing). De verplaatsing van data is bidirectioneel. De classificatie van Flynn gaat uit van het aantal transfers dat gelijktijdig kan plaats vinden tussen het geheugen en de processor(en), zie fig.1.

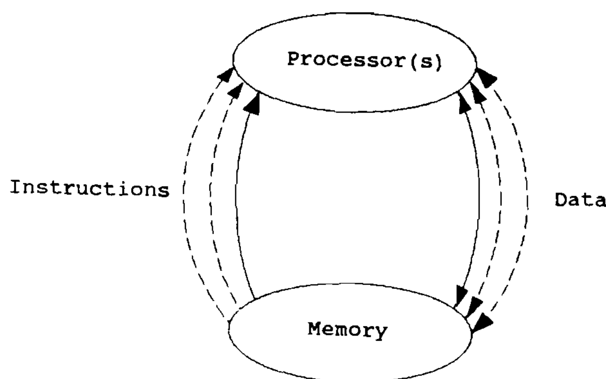

Fig.1. Classificatie volgens Flynn.

[3]Zie literatuurlijst op bladzijde 91.

Als er slechts één stroom van instructies mogelijk is van geheugen naar de processor(en), dan wordt dat weergegeven met SI (Single Instruction stream). In het geval dat er meerdere stromen van instructies zijn wordt dat MI (Multiple Instruction stream). Op dezelfde wijze noteert men voor de verplaatsing van de data SD of MD (Single Data stream respectievelijk Multiple Data stream). Gecombineerd levert deze notatie vier klassen op: SISD, MISD, SIMD en MIMD. De volgende tabel laat zien in welke klassen de architecturen vallen die in dit verslag beschreven worden. Tevens is de classificatie weergegeven van associatieve processoren [YAU77] die niet behandeld zullen worden.

SISD	uniprocessor
MISD	pipeline processing
SIMD	array processor systolic array associative processor
MIMD	multiprocessor dataflow wavefront array

Hoewel er bij de uniprocessor (SISD) niet echt sprake is van parallel processing zullen we later zien dat er toch gebruik gemaakt kan worden van parallellisme in de architectuur. Van de categorie MISD bestaan eigenlijk geen implementaties, maar het begrip pipeline processing komt dicht in de buurt. Bij pipelining worden namelijk gelijktijdig meerdere operaties uitgevoerd op één enkele stroom van objecten. Als de uit te voeren operaties afhankelijk zouden zijn van instructies die parallel uit het geheugen gelezen worden, dan zouden we een

zuiver MISD-systeem hebben. We classificeren pipelining toch bij MISD omdat deze techniek zeer belangrijk is voor parallel processing.

Bij SIMD-systemen is het mogelijk gelijktijdig meerdere data elementen te manipuleren met slechts één instructie. Op alle data elementen wordt dezelfde operatie verricht. Deze architecturen kenmerken zich dan ook doordat er wel meerdere datapaden zijn, maar slechts één (globale) besturing.

Bij MIMD-systemen zijn er wel meerdere (lokale) besturingen, zodat de processoren onafhankelijk van elkaar kunnen werken. Een gemeenschappelijk kenmerk van MIMD-systemen is dat ze "data driven" zijn. Dit wil zeggen dat het tijdstip waarop operaties verricht worden, bepaald wordt door de data zelf.

Systolische arrays en wavefront arrays hebben het pipeline principe gemeen. De verplaatsing van data naar en van de datapaden gebeurt doordat de data elementen worden doorgegeven zoals dat op een lopende band gebeurt.

3. UNIPROCESSOREN

3.1. Inleiding

Een uniprocessor systeem [TAN84 p.20] (of ook wel "single CPU[4] serial computer", zie fig.2) bestaat uit één enkele processor, geheugen, I/O en bussen: address bus, data bus en control bus. De performance die zo'n systeem kan leveren, wordt onder andere bepaald door de structuur van de interne en externe bussen. In dit hoofdstuk zullen we nagaan hoe het gebruik van parallellisme in de busstructuren de performance kan verbeteren.

3.2. Externe busstructuren

Het transport van instructies van het geheugen naar de processor, en de uitwisseling van gegevens tussen processor en geheugen gebeurt via dezelfde bussen. Het voordeel hiervan is dat een eenvoudig en compact geheel ontstaat. In veel gevallen echter zal de bandbreedte[5] van de bussen een beperkende factor

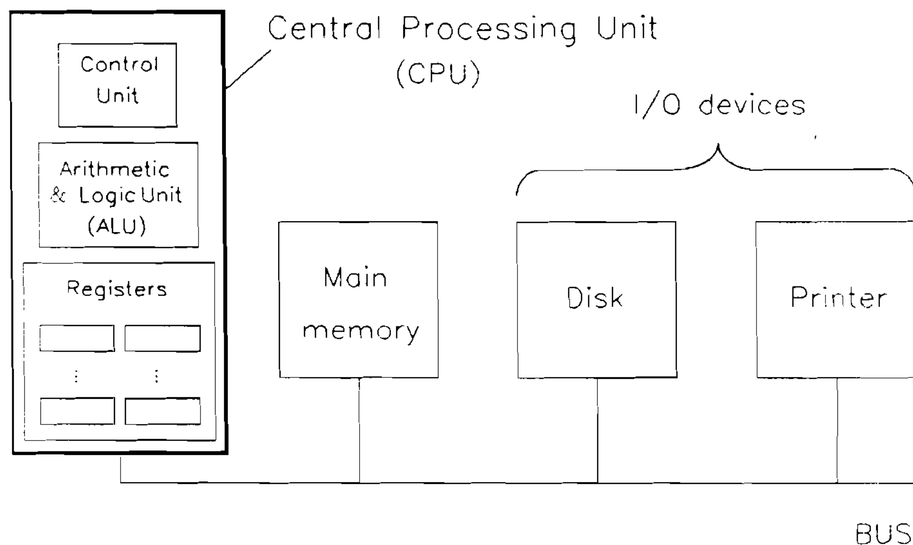

Fig.2. Uniprocessor.

---

[4]Central Processing Unit

[5]De hoeveelheid informatie die per tijdseenheid uitgewisseld kan worden, uitgedrukt in bit/s of byte/s.

zijn in de maximaal te halen performance. Daarom worden in moderne processoren vaak de adres- en databus dubbel uitgevoerd: een adresbus en een (unidirectionele) instructiebus voor de aanvoer van instructies, en een adres- en databus voor de aan- en afvoer van gegevens.

Het nadeel van deze "Harvard" architectuur is het grote aantal pennen dat de processor heeft. Een middenweg is mogelijk door alleen de databus te splitsen in een databus en een instructiebus zoals bij de Am29000 (fig.3) [JOH87, AMD87]. Om te voorkomen dat de gemeenschappelijke adresbus een bottleneck gaat vormen, worden een tweetal technieken gebruikt.

Bij "pipelined access" wordt het geheugenadres extern in een latch onthouden. De processor kan daarna de adresbus gebruiken om al vast een nieuw adres naar buiten te sturen. Het voordeel hiervan is dat de accesstijden van het instructiegeheugen en het datageheugen elkaar gedeeltelijk kunnen overlappen. Ook in het geval dat het (instructie- of data-) geheugen onderverdeeld is in modules met ieder een eigen adres latch, kan er overlapping van accesstijden gebruikt worden.

Bij "burst-mode access" kunnen meerdere geheugenreferenties op opeenvolgende lokaties plaatsvinden zonder dat de processor

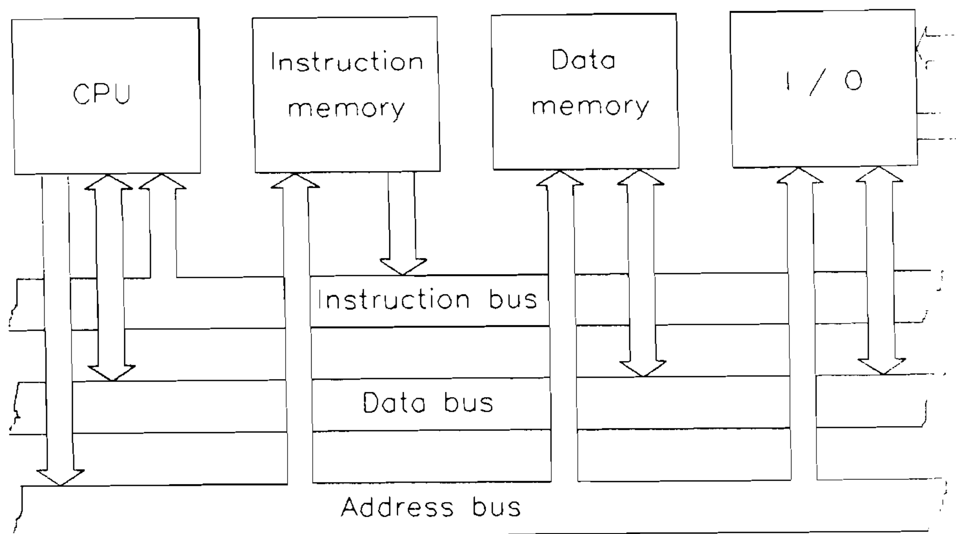

Fig.3. Gemodificeerde Harvard architectuur.

elke keer weer een adres moet genereren. Hierbij maakt men gebruik van het lokaliteitsprincipe: als op een gegeven moment de instructie op adres N geëxecuteerd wordt, is het waarschijnlijk dat de volgende instructie op adres N+1 staat. Ook voor het datageheugen kan dit principe gebruikt worden, denk bijvoorbeeld aan de situatie dat de inhouden van de registers bewaard moeten worden in een geheugenblok, of dat in een geheugenblok gezocht moet worden naar een bepaald karakter. Zodra een burst-mode access is opgezet, kan de adresbus weer voor andere adressen gebruikt worden. Om deze techniek te kunnen implementeren moeten de geheugens voorzien worden van tellers, of er moeten speciale geheugens gebruikt worden die automatisch toegang verlenen tot een aantal opeenvolgende geheugenplaatsen.

3.3. Interne busstructuren

De processor van fig.2 is op zijn beurt weer opgebouwd uit een besturingseenheid ("control unit") en een datapad. Het datapad kan bijvoorbeeld een ALU[6] en enkele registers bevatten, alsmede een bus om de onderdelen met elkaar te kunnen verbinden. Fig.4 geeft weer hoe een gedeelte van het datapad er uit zou kunnen zien. Om een operatie van de vorm R[k]:=R[i] op R[j] uit te kunnen voeren zijn de volgende stappen nodig:

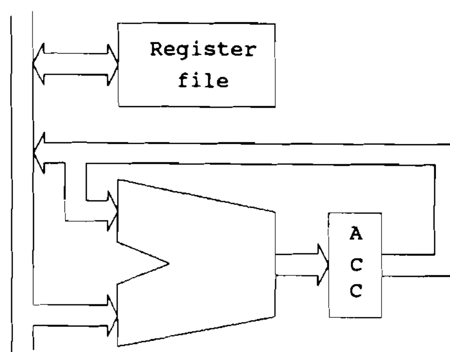

Fig.4. Datapad met enkele busstructuur.

[6] Arithmetic & Logical Unit

1. ACC:=R[i]
2. ACC:=ACC op R[j]
3. R[k]:=ACC

Als we aannemen dat elke stap 1 klokcyclus duurt, dan zijn er dus 3 klokcycli nodig voor de operatie. Een versnelling is mogelijk door meerdere bussen te gebruiken in plaats van één enkele. Fig.5 laat zien hoe de structuur wordt voor twee bussen. Een operatie duurt nu 2 klokcycli:

1. T:=R[i]
2. R[k]:=T op R[j]

De registers zijn nu dual ported, en de accumulator is vervangen door een register T dat zorgt voor de tijdelijke opslag van één van de operanden. Tenslotte is het met drie bussen mogelijk de operatie in één klokcyclus uit te voeren (zie fig.6):

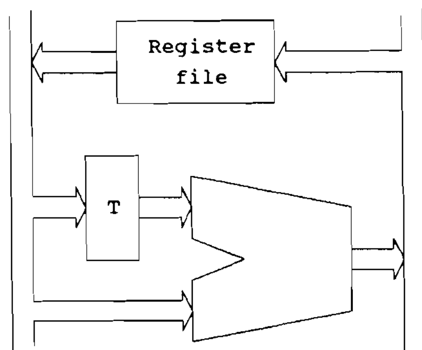

Fig.5. Datapad met dubbele busstructuur.

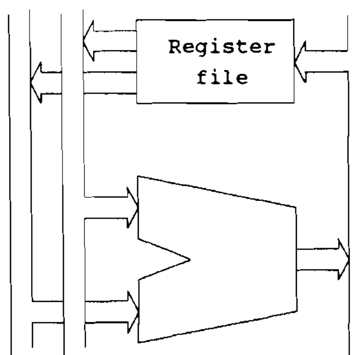

Fig.6. Datapad met drievoudige busstructuur.

1. R[k]:=R[i] op R[j]

Het blijkt dus mogelijk te zijn een hogere performance te halen door multi ported registers en meer bussen te gebruiken. Dat dit niet zonder nadelen is blijkt uit het feit dat een groot gedeelte van de oppervlakte van een IC nodig is voor verbindingen. Er zal dus een tradeoff moeten plaats vinden tussen performance en complexiteit.

Ogenschijnlijk is het datapad van fig.6 drie keer zo snel als dat van fig.4, er is immers maar één klokcyclus nodig terwijl er eerst drie nodig waren. Dat de snelheidswinst kleiner dan drie kan zijn blijkt uit de volgende beschouwing. De verschillende aktiviteiten die een rol spelen in de timing van de getoonde datapaden zijn:

1. het ophalen van een operand uit de register file;
2. het uitvoeren van de ALU funktie;
3. het wegschrijven van het resultaat.

We nemen aan dat voor aktiviteit i een tijd t_i nodig is. De minimale periodetijd van de klok in fig.4 is dan gelijk aan het maximum van t_1+t_2 en t_3. Voor het datapad van fig.6 is de minimale periodetijd gelijk aan $t_1+t_2+t_3$. Als er geen andere faktoren zijn die er voor zorgen dat de frequentie van de klok lager wordt dan de genoemde minima (zoals de interface met de buitenwereld), dan zal de snelheidswinst van fig.6 t.o.v. fig.4 gelijk zijn aan:

$$\frac{3 * \max (t_1+t_2,\ t_3)}{t_1 + t_2 + t_3}$$

In het speciale geval dat $t_1=t_2=t_3$ wordt de snelheidswinst gelijk aan twee. Het is dus best mogelijk dat de snelheidswinst (aanzienlijk) kleiner is dan drie. Ook voor andere vormen van parallellisme is er vaak sprake van een vergelijkbare situatie: door het verhogen van de complexiteit kan de performance ver-

groot worden, maar in het algemeen is er geen sprake van een lineair verband.

De hulpregisters ACC en T zijn in de 3-bus architectuur komen te vervallen. In het hoofdstuk over pipelining zullen we zien dat er mogelijk toch hulpregisters gebruikt worden om de performance te vergroten.

4. MULTIPROCESSOREN

4.1. Inleiding

Onder multiprocessor systemen (fig.7) verstaan we systemen, bestaande uit meer dan één processor, gemeenschappelijk geheugen, gemeenschappelijke I/O kanalen en één enkel geïntegreerd operating system. Het operating system moet interactie mogelijk maken tussen de processoren en hun programma's op het niveau van complete jobs, tasks, job steps, data sets tot data elementen. In deze definitie wordt dus niet de mogelijkheid uitgesloten dat de processoren lokale geheugens hebben naast het gemeenschappelijke geheugen.

Een computer die bestaat uit één CPU en één of meer I/O processoren noemen we een multiprocessor systeem als de mogelijkheden van de I/O processor(en) vergelijkbaar zijn met de mogelijkheden van de CPU.

Een multiprocessor systeem bestaat vaak uit identieke processoren (fig.8). Deze werken dan samen een job queue af, en maken daarbij gebruik van dezelfde routines, werkgebieden en peripherals. Voor de besturing van een multiprocessor systeem (het

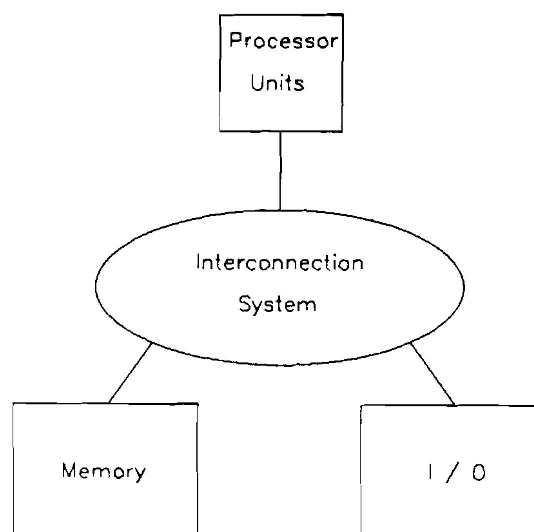

Fig.7. Multiprocessor systeem.

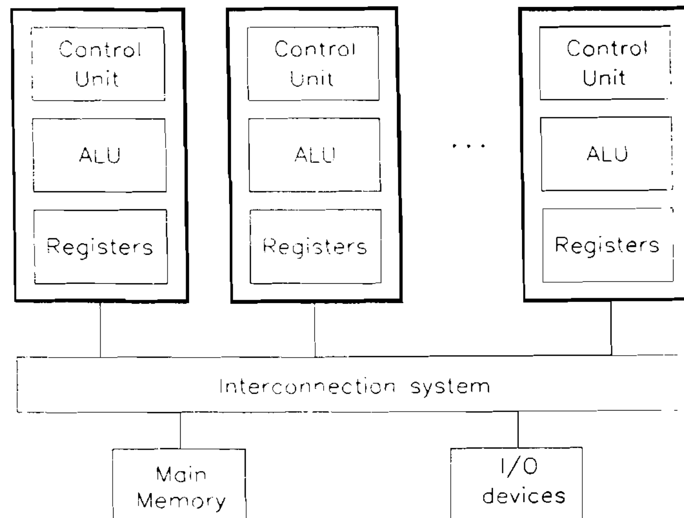

Fig.8. Multiprocessor met identieke processoren.

"operating system") zijn er een aantal mogelijkheden. Ten eerste zou één van de processoren belast kunnen worden met deze taak. Een tweede mogelijkheid is dat één van de processoren (de "master") alleen de supervisie heeft over het geheel, en taken van het operating system kan delegeren naar de "slaves".

Een derde mogelijkheid wordt duidelijk als we bedenken dat het operating system niets anders is dan een aantal procedures, I/O drivers en semaforen. Het is dus mogelijk dat de diverse processoren gezamenlijk op gelijkwaardige basis het operating system beheren. Moeilijkheden ontstaan hier doordat routines beschermd moeten worden, of re-entrant moeten zijn. De communicatie tussen de processoren zal in elk geval hoofdzakelijk via het gemeenschappelijk geheugen verlopen, hoewel rechtstreekse verbindingen altijd nodig zullen blijven (denk bijvoorbeeld aan interrupts).

Het belangrijkste in de classificatie van multiprocessor systemen is de topologie en werking van het verbindingsnetwerk tussen de verschillende functionele eenheden (processoren, geheugen, I/O devices). Hier bekijken we drie verschillende systeem organisaties:

- time-shared of gemeenschappelijke bus;
- crossbar switch matrix;
- gedistribueerde systemen.

4.2. Time-shared / gemeenschappelijke bus systemen

Het eenvoudigste interconnectie systeem voor meerdere processoren is een gemeenschappelijk communicatie pad dat alle functionele eenheden verbindt (fig.9). Het verbindingsnetwerk bestaat in dit geval uit drie bussen (adres, data en besturing), die we hier zullen beschouwen als één enkele bus. Aangezien er maar één processor gelijktijdig gebruik kan maken van de bus, is er een scheidsrechter ("arbitrator") nodig die bepaald welke processor toegang tot de bus heeft. Het systeem is wat de hardware betreft eenvoudig uit te breiden door extra functionele eenheden aan te sluiten. Wat de software betreft zijn uitbreidingen minder eenvoudig te realiseren omdat de verschillende eenheden moeten weten welke andere eenheden in het systeem aanwezig zijn en hoe die geadresseerd moeten worden.

Door de eenvoud van dit type interconnectiesysteem zijn de voordelen lage kosten en een kleine kans op storingen. Een nadeel is echter dat een defect in één van de bus interfaces een totale system failure kan veroorzaken omdat de bus een kritieke component is.

Het feit dat er slechts één pad is voor alle transfer operaties veroorzaakt nog een nadeel. De bus transfer rate binnen het

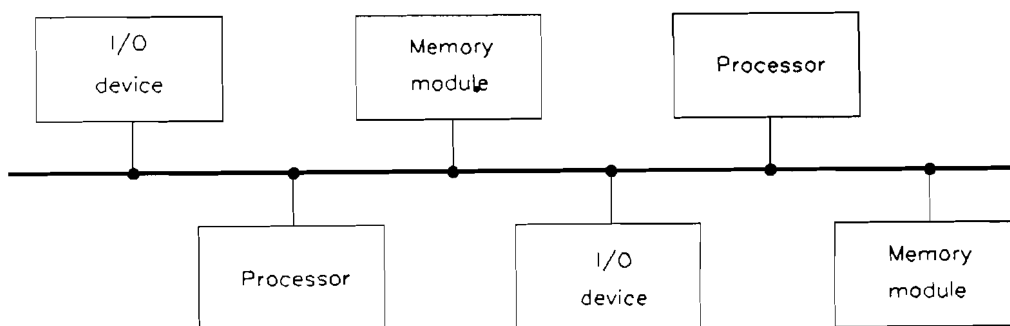

Fig.9. Multiprocessor systeem met gemeenschappelijk bus.

systeem, en daarmee de totale systeem capaciteit, wordt beperkt door de bandbreedte van dit enkele pad. Uitbreiden van het systeem door het toevoegen van functionele eenheden kan een negatieve invloed hebben op de totale throughput doordat de bus een bottleneck gaat vormen. Dit maakt dat de maximaal bereikbare systeem efficiëntie (gebaseerd op het gelijktijdig gebruik van beschikbare eenheden) van de time-shared bus laag is. Voor kleine systemen echter, kan deze organisatie goede resultaten opleveren.

Er zijn twee methoden om te voorkomen dat de bus een bottleneck vormt: het beperken van de communicatie tussen de processoren, en het vergroten van de bandbreedte van de bus.

De communicatie die nodig is tussen de processoren kan beperkt worden indien de processen die verwerkt worden, weinig of geen interactie hebben. Hiervoor is het nodig dat de processoren lokale geheugens hebben, maar bovendien moeten de applicaties te splitsen zijn in grote onafhankelijke processen die gelijktijdig uitgevoerd kunnen worden. Men spreekt in dit geval van parallellisme met een grove korrel ("coarse grain").

De tweede oplossing, het vergroten van de bandbreedte, is mogelijk door gebruik te maken van meerdere bussen zodat ook meerdere gelijktijdige transfers mogelijk worden (fig.10). De complexiteit van het systeem wordt hierdoor echter groter omdat

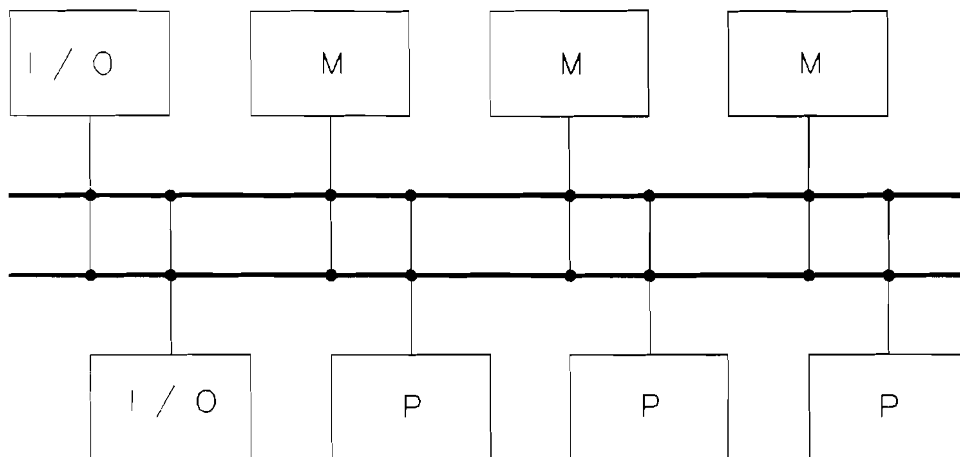

Fig.10. Multiprocessor systeem met multiple bus.

een processor die een geheugenmodule wil gebruiken, nu een keuze moet maken uit de diverse bussen. Bovendien moet er nu voor iedere bus een arbitrator zijn die bij gelijktijdige aanvragen bepaald welke processor toegang tot die bus krijgt.

4.3. Crossbar switch systemen

Als het aantal bussen in een time-shared bus systeem steeds verder toeneemt, dan ontstaat een punt waarop een afzonderlijk pad beschikbaar is voor elke processor of voor elke geheugenmodule. Het verbindingsnetwerk dat dan ontstaat heet een (non-blocking) crossbar. In fig.11 is een systeem weergegeven waarbij er van uit gegaan is dat er meer processoren zijn dan geheugenmodules en elke geheugenmodule een eigen bus heeft. Het maximale aantal transfers dat per tijdseenheid kan plaatsvinden wordt nu beperkt door het aantal geheugenmodules en de bandbreedtes van de bussen.

De benodigde hardware in de kruispunten is aanzienlijk. Niet alleen moet elk kruispunt parallelle transmissies kunnen scha-

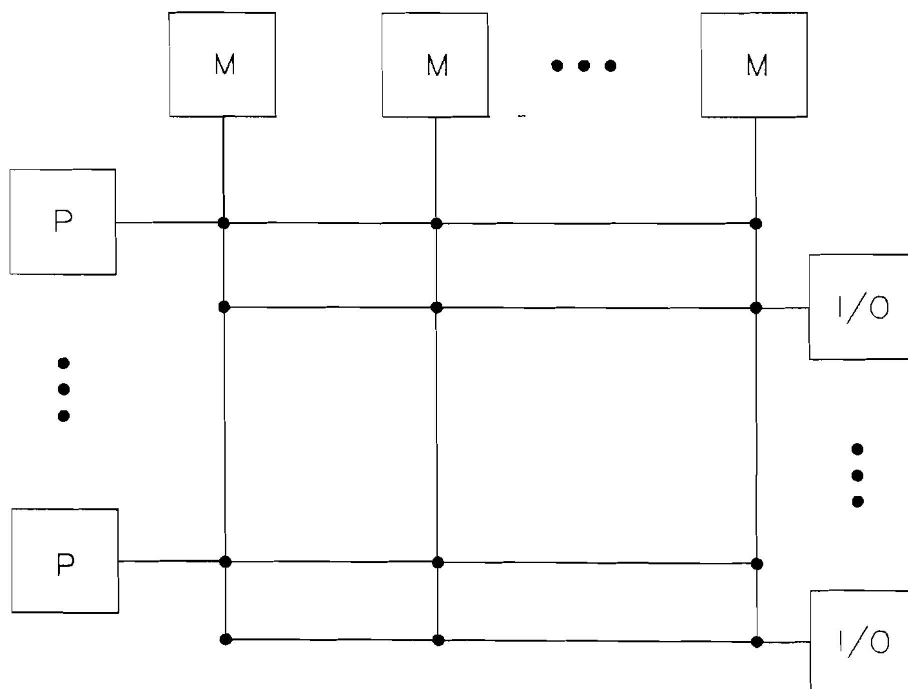

Fig.11. Multiprocessor systeem met crossbar switch.

kelen, maar ook moet er logica zijn voor het oplossen van meerdere aanvragen voor toegang tot dezelfde geheugenmodule gedurende een geheugencyclus. Deze aanvragen worden gewoonlijk afgehandeld op een van te voren vastgestelde prioriteitsbasis, bijv. I/O heeft de hoogste prioriteit in alle conflictsituaties, of processor 2 heeft altijd de hoogste prioriteit voor toegang tot geheugenmodule 2, enz. Aangezien alle data transfers en bus arbitrage taken verzorgd worden door de schakelmatrix, kunnen de functionele eenheden eenvoudig en goedkoop zijn.

Andere methoden om de processoren te verbinden met geheugenmodules, maken gebruik van "multiport memory" [ENS77 pp.114-116] of "multistage switches" [OBE88 p.280] (bijvoorbeeld vlindernetwerken [FOX87 p.50]).

4.4. Gedistribueerde systemen

Bij de tot nog toe genoemde systeemorganisaties lag het accent op het geheugen als medium voor de communicatie tussen processoren. Het is echter ook mogelijk de communicatie rechtstreeks van processor tot processor te laten plaatsvinden. In dat geval spreekt men van gedistribueerde systemen [FOX87 p.51].

Als het aantal processoren klein is, kan een netwerk opgebouwd worden door elke processor te verbinden met elke andere processor, zie fig.12. Bij n processoren bedraagt het aantal verbindingen dan (n-1) + (n-2) + ... + 1 = n(n-1)/2, terwijl iedere

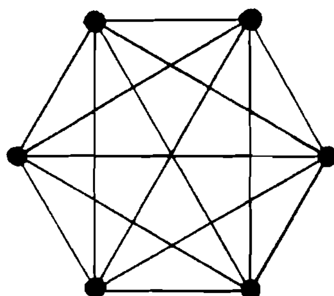

Fig.12. Gedistribueerd systeem met volledige interconnecties.

processor n-1 aansluitingen moet hebben. Voor grotere aantallen processoren is deze methode dus niet bruikbaar. In dat geval is het bijvoorbeeld mogelijk de processoren onderling te verbinden volgens een "hypercube" structuur zoals weergegeven in fig.13.

De hypercube structuur heeft het voordeel dat het aantal verbindingen en het aantal aansluitingen per processor aanzienlijk lager is dan bij toepassing van volledige interconnectie. De communicatie tussen twee processoren verloopt nu echter via een aantal tussenliggende processoren, en dit heeft de volgende nadelen:

- het aantal transfers dat gelijktijdig kan plaatsvinden is beperkt;
- er is een zekere mate van overhead nodig voor de routing van de communicatie;
- er is meer tijd nodig voor een transfer van informatie.

Een n-dimensionale hypercube bevat 2^n processoren die ieder n aansluitingen hebben, zodat het totale aantal verbindingen gelijk is aan $n.2^{n-1}$. De grootste "afstand" tussen twee processoren is bij een n-dimensionale hypercube gelijk aan n.

Het aantal verbindingen en het aantal aansluitingen per processor is nog verder te verminderen door gebruik te maken van een "mesh" netwerk zoals weergegeven in fig.14. Een n x n mesh bevat n^2 processoren die elk 4 aansluitingen hebben. Het totale

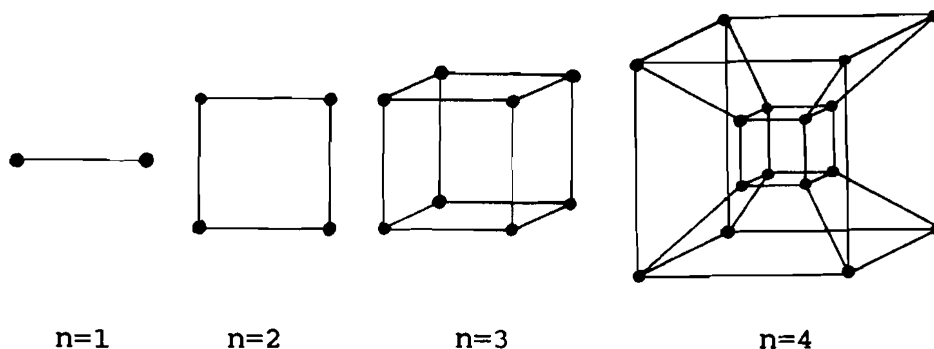

Fig.13. Gedistribueerd systeem met n-dimensionale hypercube structuur.

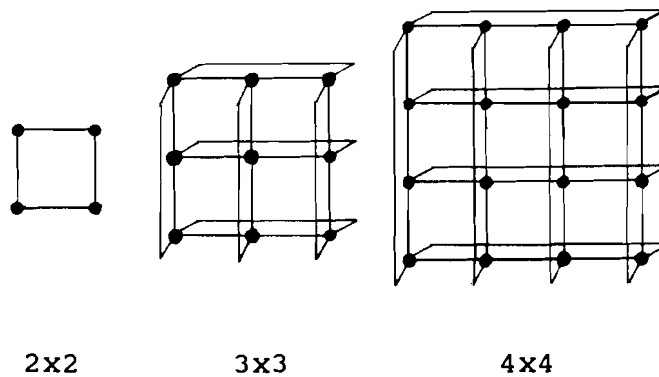

Fig.14. Gedistribueerd systeem met mesh structuur.

aantal verbindingen komt daarmee op $2n^2$. Het nadeel is dat voor grote aantallen processoren de maximale afstand tussen twee processoren groot is: n (als n even is) of n-1 (als n oneven is). Bij kleine aantallen processoren heeft de mesh structuur alleen maar voordelen t.o.v. de hypercube: voor n=2 en n=4 is de n x n mesh identiek met de n-dimensionale hypercube, terwijl de 3x3 en 5x5 mesh kortere afstanden hebben dan hypercubes met even veel processoren (resp. 4- en 5-dimensionaal).

Ten slotte geven we nog een voorbeeld van een structuur waarbij gebruik gemaakt wordt van clusters van processoren. In fig.15. is een cluster weergegeven van 2x2 processoren. Met 4 van deze clusters is nu een 4x4 cluster te maken, en met 4 van deze

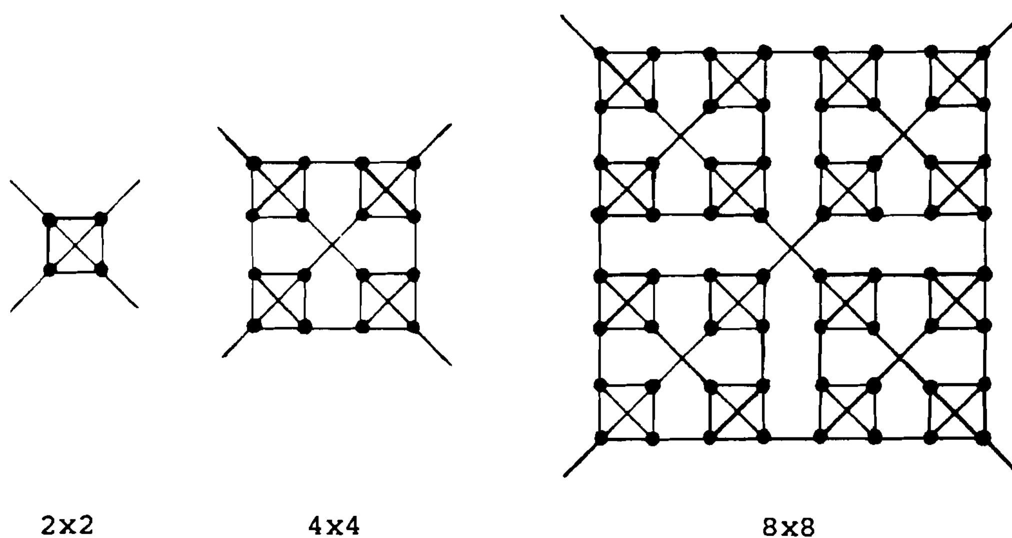

Fig.15. Gedistribueerd systeem met cluster structuur.

"superclusters" is weer een netwerk op te bouwen met 8x8 processoren, enzovoort. Een n x n cluster (waarbij n een macht van 2 is) bestaat uit n^2 processoren met ieder 4 aansluitingen, terwijl de grootste afstand n-1, en het aantal verbindingen $2(n^2-1)$ is.

structuur	aansl./proc.	verbindingen	max.afstand
voll.interconn.	N-1	N(N-1)/2	1
hypercube	log N	(N/2).log N	log N
mesh	4	2N	$\sqrt{N}$ of $\sqrt{N}-1$
cluster	4	2(N-1)	$\sqrt{N}-1$

Bovenstaande tabel geeft een overzicht van de getoonde voorbeelden, waarbij het aantal processoren genormeerd is tot N. Bij de laatste drie structuren is er bespaard op het aantal verbindingen en aansluitingen. Dit heeft tot gevolg dat de hoeveelheid informatie die gelijktijdig uitgewisseld kan worden beperkt is, zodat dit een bottleneck kan gaan vormen. Het is dan ook noodzakelijk de globale communicatie te beperken door processen zodanig over de diverse processoren te verdelen dat de meeste communicatie lokaal (bijvoorbeeld tussen twee of vier processoren) kan plaats vinden.

De structuur van fig.15 leent zich hier goed voor, omdat processen die veel met elkaar communiceren zijn te lokaliseren in een cluster van processoren. De structuur van fig.15 heeft nog een voordeel. Er zijn op de vier hoekpunten van het netwerk verbindingen die nog niet zijn benut. Deze zouden eventueel te gebruiken zijn voor de aansluiting van het netwerk op het gemeenschappelijke geheugen en de gemeenschappelijke I/O devices.

5. PIPELINE PROCESSING

5.1. Inleiding

Stel dat taken bestaan uit n aktiviteiten $A_1, A_2, \ldots, A_n$ die in deze volgorde uitgevoerd moeten worden. Neem verder aan dat we voor elke aktiviteit een stuk combinatorische logica (een segment) kunnen maken dat die bepaalde aktiviteit realiseert. Om één enkele taak uit te voeren in een zo kort mogelijke tijd kan men deze segmenten eenvoudig in serie schakelen.

Indien er echter veel van zulke taken uitgevoerd moeten worden heeft het voordeel een pipeline als in fig.16 te gebruiken. Hierin zijn registers of latches geplaatst tussen twee opeenvolgende segmenten. Dit heeft tot gevolg dat het resultaat van een aktiviteit uitgevoerd door een segment i, in de volgende klokperiode dient als invoer voor het segment i+1.

Fig.17 geeft de executie weer van m(=5) taken A,B,C,D en E met de configuratie van fig.16, waarbij het aantal aktiviteiten

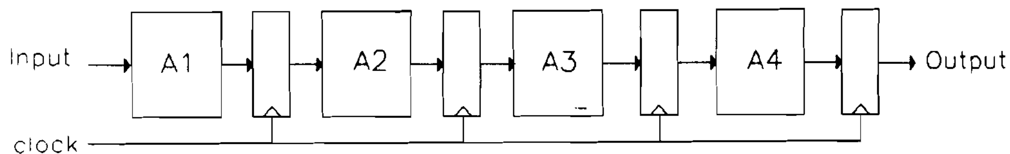

Fig.16. Pipeline principe.

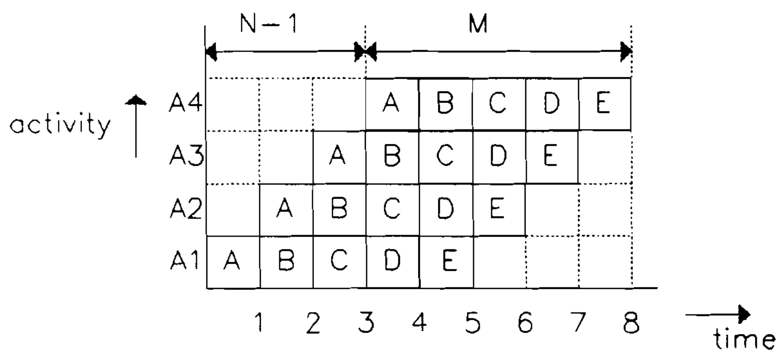

Fig.17. Executie van 5 taken op de pipeline van fig.16.

n(=4) is. De taak A is pas klaar na 4 klokperiodes, maar bij iedere volgende klokpuls is weer een volgende taak gereed. Een pipeline is te vergelijken met een lopende band: Er worden gelijktijdig meerdere aktiviteiten verricht, maar op verschillende objecten. Zonder pipelining zou er op elk tijdstip slechts één segment aktief zijn, zodat het langer duurt voordat alle taken verricht zijn.

Nemen we aan dat het segment dat aktiviteit A_i voor zijn rekening neemt, een tijd T_i nodig heeft, en dat de vertragingstijd + set-up tijd van een register δ bedraagt. De minimale periodetijd van de pipeline klok is dan:

$$T = \delta + \max(T_1, T_2, \ldots, T_n).$$

De aktiviteit die het langste duurt bepaalt dus de maximale klokfrequentie. Uit fig.17 volgt dat de tijd die nodig om alle m taken uit te voeren $(m+n-1)T$ bedraagt. Een schakeling die geen gebruik maakt van pipelining, maar waarbij alle segmenten direkt met elkaar zijn doorverbonden, heeft voor het verrichten van alle m taken een tijd $m(T_1+T_2+\ldots+T_n)$ nodig. De snelheidswinst (of "speedup") S(n) verkregen door een n-segments pipeline is dus:

$$S(n) = \frac{m(T_1 + T_2 + \ldots + T_n)}{(m+n-1)[\delta+\max(T_1, T_2, \ldots, T_n)]}$$

In het ideale geval is $T_1 \sim T_2 \sim \ldots \sim T_n >> \delta$ en voor de snelheidswinst geldt dan bij benadering:

$$S(n) \sim \frac{m.n.T}{(m+n-1).T} = \frac{m.n}{m+n-1} \sim n \quad (m>>n).$$

Dit wil zeggen dat indien de verschillende aktiviteiten ongeveer even lang duren en het aantal taken is zeer groot, dat dan de snelheidswinst vrijwel gelijk wordt aan het aantal segmenten

van de pipeline. Er zijn echter een tweetal factoren die de snelheidswinst nadelig beïnvloeden.

Fig.18 laat voor pipelines met 2, 5 en 9 segmenten zien hoe de snelheidswinst afhangt van m, het aantal taken. Het is duidelijk dat het "vullen van de pipeline" ten koste gaat van de snelheidswinst (bij weinig taken is S ongeveer evenredig met het aantal taken). De tweede oorzaak ligt in het feit dat de diverse aktiviteiten niet allemaal even groot hoeven te zijn. In het minst gunstige geval is er een aktiviteit A_i die veel langer duurt dan alle overige aktiviteiten tezamen. Dan geldt bij benadering:

$$S(n) \sim \frac{m.T_i}{(m+n-1).(\delta+T_i)} \sim \frac{m}{m+n-1}$$

Door de bottleneck die A_i vormt wordt in dit geval S kleiner dan 1, hetgeen een snelheidsverlies betekend. Dat de invloed van de bottleneck A_i afhankelijk is van de lengte van de pipeline blijkt uit het volgende voorbeeld: Stel dat n-1 aktiviteiten even lang duren, namelijk een tijd T_0, terwijl voor A_i een tijd $k.T_0$ nodig is ($k \geq 1$). In dat geval is de snelheidswinst:

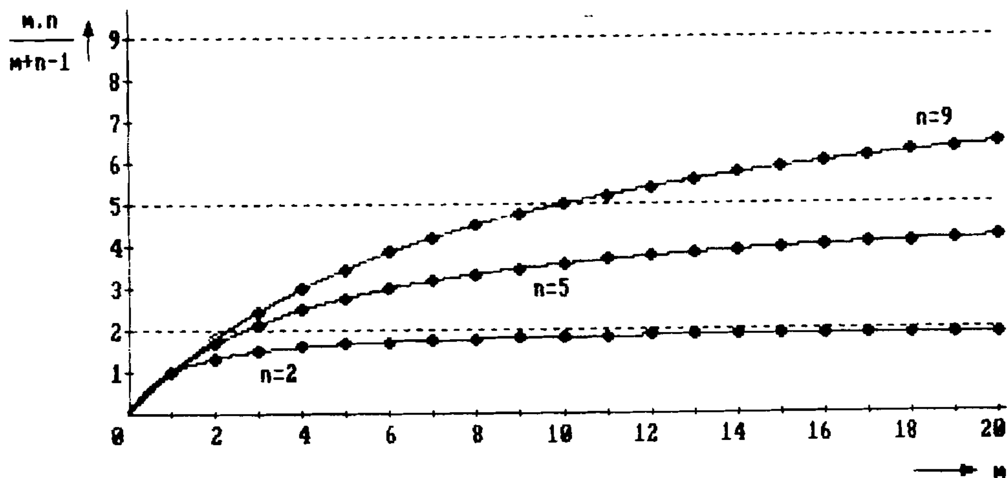

Fig.18. Snelheidswinst als functie van het aantal taken, met het aantal aktiviteiten als parameter.

$$S(n) = \frac{m[(n-1)T_0+kT_0]}{(m+n-1)(\bar{o}+kT_0)} = \frac{m(k+n-1)T_0}{(m+n-1)(\bar{o}+kT_0)} \sim \frac{m(k+n-1)}{(m+n-1)k} \sim \frac{k+n-1}{k}$$

Fig.19 geeft het verband tussen S(n) en k voor een aantal waarden van n. Het is duidelijk dat S(n) snel afneemt als de waarde van k groter dan 1 wordt. Slechts in het geval dat het aantal korte aktiviteiten groot is, kan er nog sprake zijn van een behoorlijke snelheidswinst.

Er zijn drie begrippen die vaak gebruikt worden bij pipelining, te weten [DES84 p.77]:

- efficiency (E);
- latency (L);
- throughput (U).

De pipeline efficiency is gedefinieerd als de verhouding tussen de werkelijke snelheidswinst, en de maximaal haalbare snelheidswinst (bij oneindig veel taken, en aktiviteiten die even lang duren): E=S/n. De efficiency nadert dus tot 1 naarmate S dichter bij n komt. Het volgende voorbeeld dient om de begrippen latency en throughput toe te lichten:

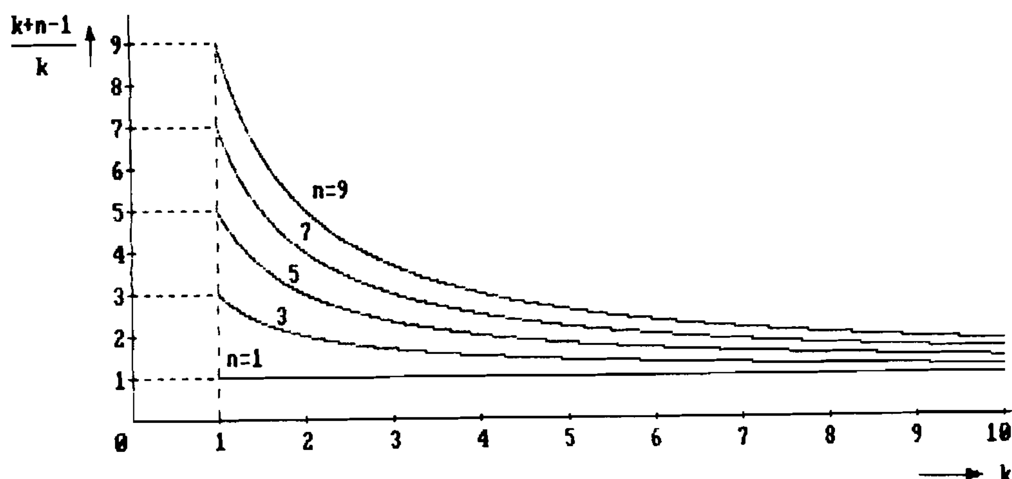

Fig.19. Snelheidswinst voor n aktiviteiten bij een bottleneck.

Tot nog toe zijn we er van uit gegaan dat het aantal segmenten van een pipeline gelijk is aan het aantal aktiviteiten (n) waaruit de taak bestaat. Nu gaan we uit van een pipeline met minder segmenten namelijk s, waarbij n deelbaar is door s. Het aantal aktiviteiten per segment is dus n/s. Nemen we verder aan dat alle aktiviteiten een tijd t duren. De periodetijd van de pipeline klok wordt dan:

$$T = \tilde{o} + \frac{n.t}{s}$$

Deze aannames komen aardig overeen met de situatie dat we elk poortniveau als een aktiviteit beschouwen. De vraag is dan hoeveel poortniveaus gecombineerd moeten worden tot één segment. In het ene uiterste geval (s=n) wordt na elk poortniveau een register geplaatst, zodat ieder segment alleen 1-laags logica bevat. In het andere uiterste geval (s=1) bestaat de schakeling uit slechts één segment met n-laags logica.

Onder latency (of ook wel pipeline delay) verstaat men de tijd die verstrijkt voordat de eerste taak helemaal verwerkt is:

$$L = s.T = s.\tilde{o} + n.t$$

De throughput is het aantal taken dat per tijdseenheid wordt uitgevoerd. Aangezien er bij iedere klokpuls een taak gereed komt (bij een volle pipeline) is de throughput gelijk aan de reciproke van de periodetijd van de klok:

$$U = \frac{1}{T} = \frac{s}{s.\tilde{o} + n.t}$$

Fig.20 geeft de latency en throughput weer als functie van het aantal segmenten, waarbij n=12 en õ=2t gekozen is. Het blijkt dat s zo groot mogelijk moet zijn om de throughput groot te maken. Nadeel is echter dat daardoor ook de latency en de nodige hardware (in de vorm van extra registers) toeneemt. Er

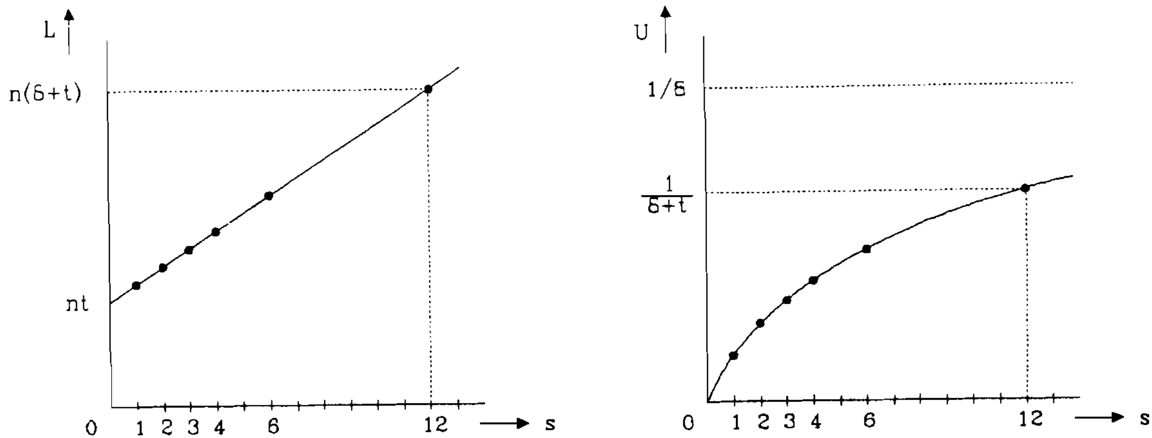

Fig.20. Latency en throughput als functie van het aantal segmenten.

zal dus een tradeoff moeten plaats vinden. Men zal bij grote m (het aantal achter elkaar uit te voeren taken) eerder kiezen voor een grote throughput, en bij kleine m voor een kleine latency en weinig hardware. Als m een vast getal is, kan er een optimale waarde van s bepaald worden waarvoor de tijd die nodig is om alle m taken uit te voeren minimaal is:

$$(m+s-1).(\delta + \frac{n.t}{s}) \text{ is minimaal als } s^2 = \frac{(m-1).n.t}{\delta}.$$

In de hier boven geschetste situatie dat er een aktiviteit is die veel langer duurt dan alle overige, zou men de throughput kunnen vergroten door A_i te splitsen in meerdere kleinere aktiviteiten. Als dat niet mogelijk is, dan kunnen wellicht meerdere van de overige aktiviteiten gecombineerd worden in één of meer grotere segmenten. Hierdoor wordt de throughput weliswaar niet groter, maar de latency wordt wel kleiner, terwijl er minder registers nodig zijn.

Het pipeline concept wordt in de praktijk voor twee taken gebruikt: arithmetische operaties en instructie executie. Hieronder volgen enkele voorbeelden van deze toepassingen.

5.2. Arithmetische pipelines

De ALU in een processor zorgt voor de executie van de arithmetische en logische operaties. De logische operaties zijn veel gemakkelijker te implementeren dan de arithmetische, omdat we bij de laatste te maken kunnen krijgen met floating point getallen en carry propagatie tussen de afzonderlijke bits van de getallen. De klokfrequentie moet laag genoeg zijn om de arithmetische operaties in één periode van de klok te kunnen uitvoeren. Als we één schakeling gebruiken voor beide operaties, dan wordt voor de logische operaties meer tijd gereserveerd dan nodig is. Om deze reden kan de ALU gesplitst worden in een logische unit, en één of meer arithmetische units. In de arithmetische units kan dan eventueel gebruik gemaakt worden van het pipeline concept.

Als eerste voorbeeld beschouwen we de optelling van twee floating point getallen. Vóór de eigenlijke optelling moeten de exponenten van de twee getallen aan elkaar gelijk gemaakt worden. Dit gebeurt door de twee exponenten van elkaar af te trekken, en de mantisse van één van de getallen te verschuiven over het aantal bits dat het resultaat van de aftrekking is. Na de eigenlijke optelling (of aftrekking) van de mantissen moet het resultaat genormaliseerd worden tot een standaard formaat. Dit betekent verschuiven van de mantisse en aanpassen van de exponent.

Fig.21 geeft weer hoe een add/subtract unit voor floating point getallen opgebouwd kan worden als een pipeline. In het eerste segment wordt het verschil bepaald tussen de exponenten van de twee getallen. Het verschuiven van de mantisse in segment 2 gebeurt met een barrel shifter. In segment 4 worden de mantisse en de exponent aangepast. Deze twee aktiviteiten kunnen gelijktijdig plaats vinden. De diverse optellers die in de pipeline voorkomen, worden gerealiseerd met carry look-ahead adders, waarbij een segment met een opteller eventueel gesplitst kan worden in meerdere segmenten indien de optelling te lang duurt.

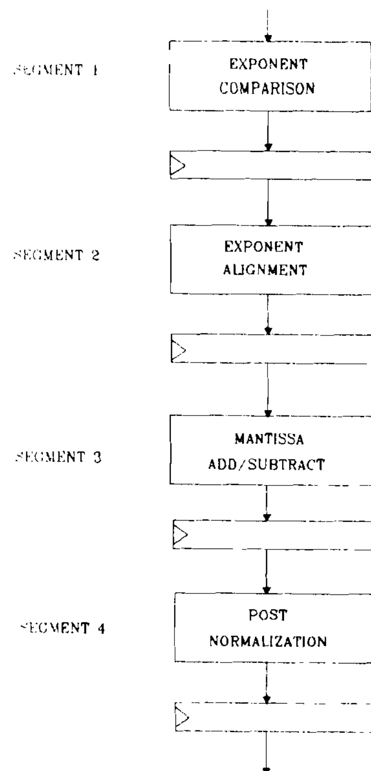

Fig.21. Pipelined floating point add/subtract unit.

De schakeling van fig.21 is zeer geschikt voor het optellen of aftrekken van vectoren, omdat daarbij steeds meerdere operanden beschikbaar zijn. Vandaar dat dit soort schakelingen "vector units" genoemd worden. Voor scalaire grootheden (m=1 in fig.18) levert het pipeline concept geen enkele snelheidswinst op, of zorgt zelfs voor een verslechtering als er verschillende tijden nodig zijn voor de diverse aktiviteiten. Vaak wordt toch de vector unit gebruikt voor de scalaire operaties om geen aparte scalaire unit te hoeven bouwen.

Voor het tweede voorbeeld nemen we de vermenigvuldiging van twee n-bit getallen:

$$A = a_0.2^0 + a_1.2^1 + \ldots + a_{n-1}.2^{n-1};$$
$$B = b_0.2^0 + b_1.2^1 + \ldots + b_{n-1}.2^{n-1}.$$

Het 2n-bit resultaat A*B is als volgt te schrijven:

$$A * B = A * (b_0.2^0 + b_1.2^1 + \ldots + b_{n-1}.2^{n-1})$$

$$= P_0.2^0 + P_1.2^1 + \ldots + P_{n-1}.2^{n-1}$$

Hierin is $P_i = A.b_i$ een n-bit getal dat gelijk is aan A (als b_i=1), of dat gelijk is aan 0 (als b_i=0). Deze n tussenprodukten moeten nu verschoven en opgeteld worden om het resultaat A*B te krijgen. Het optellen kan problemen geven als hiervoor n ripple-carry adders (RCA's) of carry look-ahead adders (CLA's) gebruikt worden omdat de vermenigvuldiging dan erg traag en/of erg complex wordt. Er is echter een techniek waarmee de optelling van drie getallen gereduceerd kan worden tot een optelling van twee getallen. Nemen we als voorbeeld de volgende optelling van decimale getallen:

```
  9 6 3 0
  8 5 2 9
  7 4 1 8
---------
2 5 5 7 7
```

Hierbij vindt steeds een carry-ripple plaats van de ene cijferpositie naar de volgende. Dit heeft tot gevolg dat de verschillende digits van het resultaat van de optelling alleen in een strikt sequentiële volgorde berekend kunnen worden. Door nu het optellen van de carry's uit te stellen krijgen we de volgende situatie:

```
  9 6 3 0
  8 5 2 9
  7 4 1 8
---------
  4 5 6 7
2 1 0 1
---------
2 5 5 7 7
```

Op deze manier kan tegelijkertijd van elke cijferpositie een som-digit en een carry-digit berekend worden. Daarna moeten de som-digits en de verschoven carry-digits nog op de gewone manier opgeteld worden of, indien nog meer getallen opgeteld

moeten worden, samen met het vierde getal op dezelfde manier opgeteld worden.

De schakeling waarmee drie binaire getallen van m bit gereduceerd worden tot twee getallen heet een carry-save adder (CSA), en is weergegeven in fig.22. De schakeling bestaat net als een RCA uit m full adders (FA's) hetgeen een eenvoudige schakeling geeft, maar hier zijn de FA's niet onderling doorverbonden hetgeen bovendien een snelle schakeling geeft. Met behulp van CSA's en het pipeline principe is nu een snelle vermenigvuldiger te construeren, waarvan in fig.23 een voorbeeld is gegeven voor 8-bit getallen.

In deze "Wallace tree multiplier" [RAF88 p.318] bestaat het eerste segment (de "partial product generator") uit 64 AND-poorten. Hiermee wordt het getal A acht keer gecopieerd of gemaskeerd zodat de getallen $A.b_0$, $A.b_1$, ..., $A.b_{n-1}$ ontstaan. In het tweede segment vinden we twee CSA's terug waarin $A.b_2.2^0$, $A.b_3.2^1$, $A.b_4.2^2$ en $A.b_5.2^0$, $A.b_6.2^1$, $A.b_7.2^2$ gereduceerd worden. De resultaten hiervan zijn twee som-getallen en twee carry-getallen van 10 bit. Als we deze S_0,C_0 en S_1,C_1 noemen, dan vinden in het volgende segment de reducties plaats van $A.b_0.2^0$, $A.b_1.2^1$, $S_0.2^2$ en $C_0.2^0$, $S_1.2^1$, $C_1.2^2$.

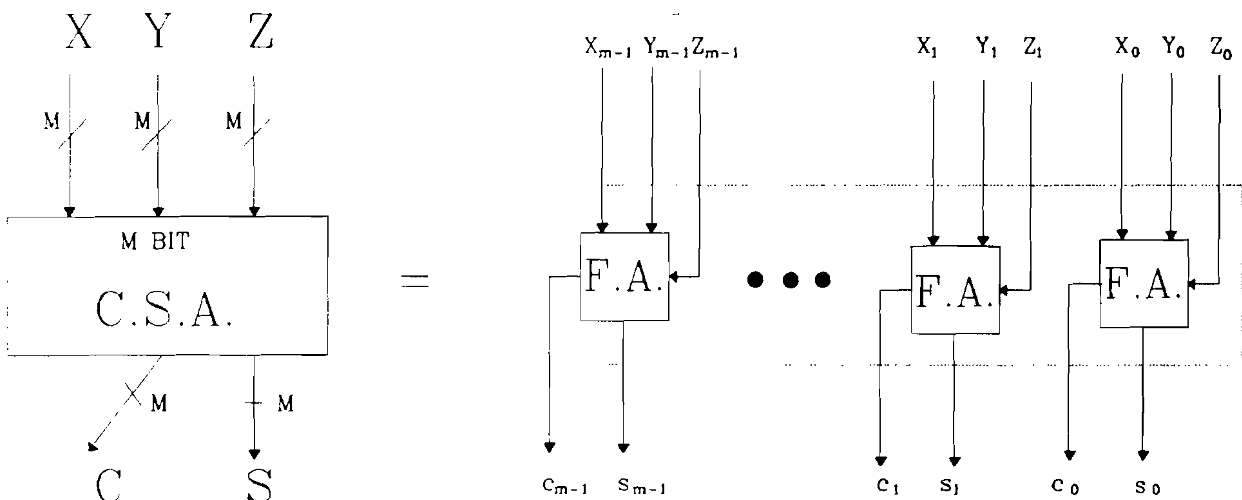

Fig.22. Carry-save adder.

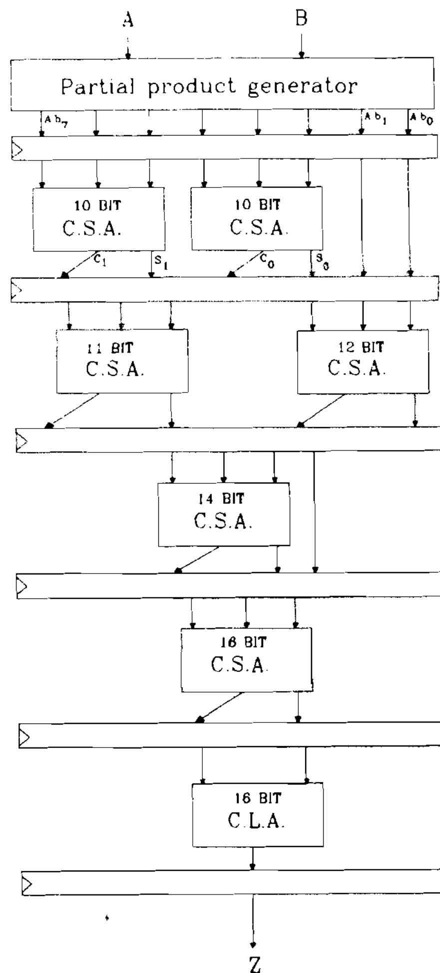

Fig.23. Pipelined Wallace tree multiplier.

Na de laatste CSA in het vijfde segment resteren twee 16-bit getallen die nog op de normale manier opgeteld moeten worden. Voor deze aktiviteit is in de figuur een CLA getekend. Een 16-bit CLA wordt gewoonlijk gerealiseerd met 6-laags logica om de schakeling niet al te complex te maken. De partial product generator en de CSA's in de andere segmenten kunnen eenvoudig gemaakt worden met 2-laags logica. Hier ontstaat dus de situatie dat één bepaald segment van een 6-segments pipeline 3 keer zo veel tijd nodig heeft als de overige 5 segmenten (zie fig.19 met n=6 en k=3). De CLA vormt dus een bottleneck die de snelheid van de totale vermenigvuldiger beperkt.

Er zijn een aantal methoden om dit nadeel te beperken:

- Aangezien de CLA de maximaal te halen snelheid bepaalt is het mogelijk een aantal registers te besparen door meerdere

segmenten met CSA's uit fig.23 te combineren tot één segment. Dit heeft dus tot gevolg dat de latency kleiner wordt.

- Als we één (of meer) extra register(s) gebruiken, kunnen we de CLA splitsen in twee (of meer) segmenten. Hierdoor wordt het mogelijk een hogere klokfrequentie te gebruiken, zodat de throughput groter wordt.
- Fig.24 geeft een combinatie van de twee bovenstaande mogelijkheden. Hierbij zijn het tweede en derde segment, en het vierde en vijfde segment gecombineerd. Bovendien is de CLA nu vervangen door twee segmenten, met in elk segment twee (gewone) 4-bit optellers in serie. De segmenten 2,3,4 en 5 bestaan nu allemaal uit 4-laags logica.

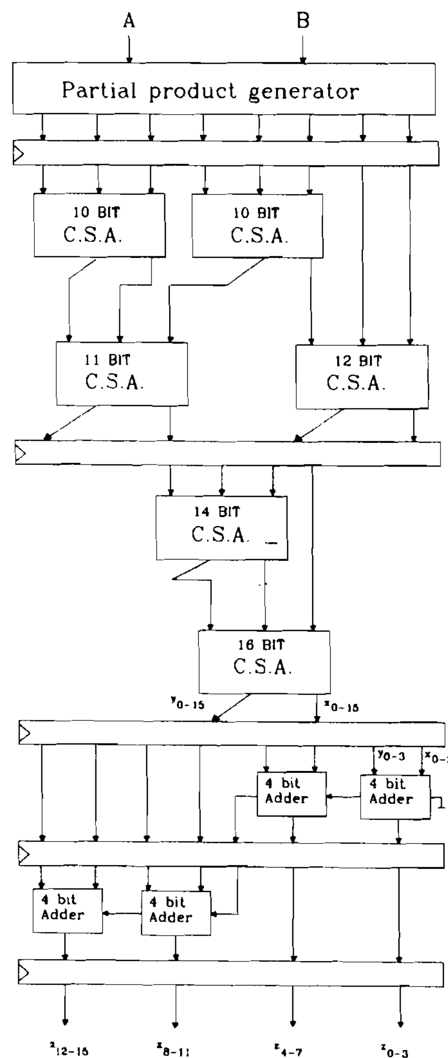

Fig.24. Gemodificeerde Wallace tree.

5.3. Instructie pipelining

In het geval dat de verschillende uit te voeren taken instructies zijn die geëxecuteerd moeten worden, bestaat een taak bijvoorbeeld uit de volgende vijf aktiviteiten:

FI (Fetch Instruction): haal de volgende instructie op uit het geheugen
DI (Decode Instruction): decodeer de instructie
FO (Fetch Operand): haal de operand op uit het geheugen
EO (Execute Operation): voer de gespecificeerde operatie uit
SR (Store Result): sla het resultaat op in het geheugen

In dit voorbeeld van een 2-adres machine bepalen de adressen de plaatsen in het geheugen van de eerste operand en de bestemming. De tweede operand is de inhoud van een register.

Op dezelfde manier als bij de arithmetische operaties kunnen we hier pipelining toepassen door registers te plaatsen tussen de segmenten die de verschillende aktiviteiten uitvoeren. Fig.25 laat zien hoe de verwerking van een aantal instructies zou kunnen verlopen in een 5-segments pipeline. In de eerste klokcyclus wordt de eerste instructie uit het geheugen opgehaald. Gedurende de tweede klokcyclus kan de tweede instructie uit het geheugen worden opgehaald, terwijl de eerste instructie wordt gedecodeerd, enzovoort. Zonder pipelining zou er elke vijf klokperiodes een instructie verwerkt worden. Nu kan er elke klokperiode een instructie voltooid worden (als de pipeline eenmaal gevuld is), zodat de throughput (het aantal instructies

SR					1	2	3	4	5	6	7	8	9	10
EO				1	2	3	4	5	6	7	8	9	10	11
FO			1	2	3	4	5	6	7	8	9	10	11	12
DI		1	2	3	4	5	6	7	8	9	10	11	12	12
FI	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Fig.25. Pipelined executie van instructies.

per seconde) vijf keer zo groot wordt. Vergeleken met arithmetische pipelines zijn er echter nog een aantal extra factoren die de pipeline efficiency nadelig kunnen beïnvloeden [STA86, BRI86].

Een probleem treedt namelijk op bij de derde klokcyclus. Terwijl de operand van de eerste instructie uit het geheugen wordt opgehaald, zou ook de derde instructie moeten worden opgehaald. Indien er per klokperiode maar één geheugen access mogelijk is (d.w.z. dat een geheugencyclus even lang duurt als een machine cyclus), dan moet het ophalen van de derde instructie worden uitgesteld. Omdat in de vierde, vijfde en zesde klokcycli ook geheugen referenties plaats vinden, gebeurt de fetch van de derde instructie pas in de zevende periode van de klok.

Zoals uit fig.26 blijkt worden er maar twee instructies per zes klokperiodes uitgevoerd, zodat de snelheidswinst slechts S=(2/6)/(1/5)=5/3 is, en de pipeline efficiency E=(5/3)/5=1/3. Door de veelheid aan geheugenreferenties en de beperkte toegankelijkheid van het geheugen, is het niet erg zinvol in deze situatie pipelining toe te passen. Methoden die hiervoor een oplossing geven moeten er voor zorgen dat de geheugentoegang niet de bottleneck vormt. In de praktijk worden één of meer van de volgende maatregelen genomen:

- Toepassen van memory interleaving. Door het geheugen te verdelen in modules die gelijktijdig toegankelijk zijn, is het mogelijk meerdere geheugenreferenties per machinecyclus te laten plaats vinden.

SR					1	2					3	4		
EO				1	2					3	4			
FO			1	2					3	4				
DI		1	2					3	4					5
FI	1	2					3	4					5	6

Fig.26. Memory access conflicten bij pipelining.

- Toepassen van Harvard architectuur. Als er aparte bussen zijn voor instructies en gegevens, dan wordt de pipeline iets efficiënter. Uit fig.27 volgt dat S=(2/4)/(1/5)=5/2 en E=(5/2)/5=1/2.
- Meer registers in de processor. In het hierna volgende gaan we er van uit dat de processor zo veel registers heeft dat het aantal geheugenreferenties voor datawoorden verwaarloosbaar is. Tevens moet de architectuur van de processor zodanig zijn dat er gelijktijdig van een register gelezen, en naar een register geschreven kan worden. Hiervoor is bijvoorbeeld de interne busstructuur van fig.6 te gebruiken. Om hierin pipelining mogelijk te maken, worden er dan extra registers geplaatst aan de ingangen en de uitgangen van de ALU. Het resultaat (dat de segmenten FO, EO en SR van de pipeline vormt) is weergegeven in fig.28.

Een andere complicatie die kan optreden bij instructie pipelining is data dependency. Nemen we als voorbeeld dat het resultaat van de operatie in instructie nummer 3 gebruikt moet worden in instructie nummer 4. Het ophalen van de operand in instructie 4 kan dan pas plaats vinden nadat instructie 3 het resultaat heeft opgeslagen, zie fig.29. Dit nadeel is te verminderen indien instructies 2 en 3, en/of instructies 4 en 5 van plaats verwisseld mogen worden, zonder dat dit invloed heeft op de werking. Dit vraagt dus meer inspanning van de machinetaal programmeur, of extra investeringen in de compiler. In het laatste geval spreekt men van een "optimizing" compiler.

Er zal echter niet in alle gevallen een verwisseling van instructies mogelijk zijn, zodat in de hardware logica aanwezig

SR					1	2			3	4			5	
EO				1	2			3	4			5	6	
FO			1	2			3	4			5	6		7
DI		1	2	3			4	5			6	7		8
FI	1	2	3	4			5	6			7	8		9

Fig.27. Vermindering van conflicten door Harvard architectuur.

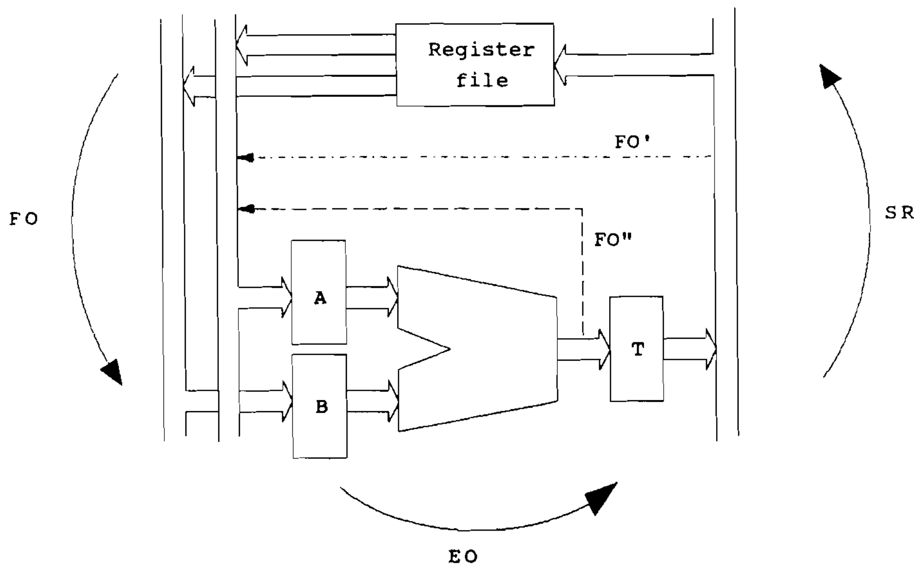

Fig.28. Drie segmenten van een instructie pipeline.

SR					1	2	3			4	5	6	7	8
EO				1	2	3			4	5	6	7	8	9
FO			1	2	3			4	5	6	7	8	9	10
DI		1	2	3	4			5	6	7	8	9	10	11
FI	1	2	3	4	5			6	7	8	9	10	11	12

Fig.29. Data dependency bij pipelining.

moet zijn dat data dependencies kan detecteren. Als bovendien in het datapad extra bussen geplaatst zijn, die gebruikt kunnen worden bij afhankelijkheden, dan is het nadeel altijd geheel of gedeeltelijk te vermijden. Door een bus, aangegeven met de punt-streep lijn FO' in fig.28, toe te voegen kan de inhoud van het pipeline register T gelijktijdig in de register file en in het pipeline register A geschreven worden. Met de gestreepte lijn FO" wordt het mogelijk het resultaat van de ALU-operatie gelijktijdig in T en in A te schrijven.

Een laatste moeilijkheid wordt gevormd door de spronginstructies. In het voorbeeld van fig.30 maakt instructie 4 een sprong naar instructie 18. Het ophalen van instructie 18 kan pas plaats vinden nadat de nieuwe waarde van de programmateller is

SR					1	2	3	4					18	19
EO				1	2	3	4					18	19	20
FO			1	2	3	4					18	19	20	21
DI		1	2	3	4					18	19	20	21	22
FI	1	2	3	4					18	19	20	21	22	23

Fig.30. Branch dependency bij pipelining.

opgeslagen. De reeds opgehaalde, en eventueel gedeeltelijk verwerkte, instructies 5,6,... zijn dan voor niets geweest.

In dit voorbeeld is uit gegaan van een relatieve sprong ten opzichte van de programmateller. In het segment "execute operation" van instructie 4 wordt de nieuwe waarde van de programmateller berekend, en in het segment "store result" wordt de waarde opgeslagen. In het geval dat er geen relatieve sprongen voorkomen, is het eventueel mogelijk in een vroeger stadium van de pipeline de instructie op het sprongadres op te halen. Dit gaat echter niet zonder meer bij conditionele sprongen. In dat geval gebruikt men een strategie die "target prefetch" genoemd wordt. Hierbij worden op het moment dat de conditionele sprong herkend is, zowel de volgende instructie opgehaald als de instructie op het sprongadres. Deze laatste wordt dan bewaard in een aparte buffer.

Een andere methode waarmee de nadelen van spronginstructies worden beperkt is het "delayed branch" concept. Hierbij wordt een sprong pas uitgevoerd ná de volgende instructie. In het voorbeeld van fig.30 moet de (optimizing) compiler er dus voor zorgen dat instructies 3 en 4 van plaats verwisseld worden. Het resultaat is weergegeven in fig.31. In het geval dat het een conditionele sprong betreft, en de onmiddellijk voorafgaande instructie (3) heeft invloed op de conditie, dan mogen de twee instructies niet van plaats verwisseld worden. Als ook geen van de voorgaande instructies (2 of 1) na de spronginstructie geplaatst kan worden met behoud van de werking, dan moet de compiler een NOP-instructie (*) invoegen, zie fig.32. De erva-

SR					1	2	4	3				18	19	20
EO				1	2	4	3				18	19	20	21
FO			1	2	4	3				18	19	20	21	22
DI		1	2	4	3				18	19	20	21	22	23
FI	1	2	4	3				18	19	20	21	22	23	24

Fig.31. Vertraagde spronginstructie.

SR					1	2	3	4	*				18	19
EO				1	2	3	4	*				18	19	20
FO			1	2	3	4	*				18	19	20	21
DI		1	2	3	4	*				18	19	20	21	22
FI	1	2	3	4	*				18	19	20	21	22	23

Fig.32. Vertraagde spronginstructie (4) met NOP-instructie (*).

ring leert echter dat de meerderheid van conditionele sprongen op deze wijze geoptimaliseerd kan worden [STA86 p.78].

5.4. Reduced Instruction Set Computers

In de vorige paragraaf zijn we er van uit gegaan dat de executie van elke instructie opgedeeld kan worden in de vijf genoemde aktiviteiten. Dit zal niet het geval zijn indien er meerdere instructieformaten mogelijk zijn (denk bijvoorbeeld aan de diverse adresseringsmethoden). In het algemeen geldt dat voor een eenvoudige en efficiënte instructie pipeline, een regelmatig instructieformaat nodig is met weinig verschillende adresseringsmethoden.

In RISC[7] processoren [STA86, LEO88] wordt zo'n "gestroomlijnd" instructieformaat gebruikt, vaak samen met enkele van de diverse technieken die genoemd zijn om de efficiëntie van de instructie pipeline te verbeteren:

---

[7]Reduced Instruction Set Computer

- Harvard architectuur;
- grote register file;
- interne structuur met 3 bussen;
- extra bussen voor het oplossen van data dependencies;
- target prefetch;
- delayed branch;
- optimizing compiler.

Eén van de doelstellingen van het RISC-concept is dan ook het verminderen van het aantal klokcycli dat nodig is voor één instructie.

Een tweede doelstelling is de frequentie van de klok zo hoog mogelijk maken. Dit wordt bereikt door de omvang van de instructieset sterk te beperken, zodat de besturing eenvoudiger kan worden. In veel gevallen is de besturing dan ook zo eenvoudig dat er geen microprogrammering wordt gebruikt, maar dat de besturing "hardwired" is.

De filosofie die ten grondslag ligt aan deze aanpak, is dat veel complexe instructies van een CISC[8] processor heel weinig gebruikt worden. Door nu de instructieset zodanig samen te stellen dat de complexe instructies opgebouwd kunnen worden uit meerdere eenvoudige, wordt het mogelijk de veel gebruikte instructies sneller te verwerken. Bovendien zou de ontwerptijd van een eenvoudigere processor korter moeten zijn, zodat bespaard wordt op ontwikkelingskosten [TRE84 pp.6,7].

---

[8]Complex Instruction Set Computer

6. DATAFLOW COMPUTERS

6.1. Inleiding

De efficiëntie van een parallelle computer wordt door veel factoren beïnvloed. Een groot probleem wordt gevormd door de conflicten die op kunnen treden wanneer een resource door meerdere processoren wordt gebruikt (gemeenschappelijk geheugen of een communicatiekanaal). Het reduceren van deze conflicten vergt een zekere mate van overhead: processing die niet nodig zou zijn als er geen sprake was van parallellisme. Een doelstelling bij het ontwerpen of toepassen van een parallelle computer is dan ook dat deze overhead niet te groot wordt. Verder is het gewenst dat de performance van een machine uitgebreid kan worden door meer processoren (of processing elements) te gebruiken. In het ideale geval is de performance evenredig met het aantal processing elements.

Eén van de strategieën die gebruikt wordt om de overhead te beperken is het minimaliseren van de communicatie tussen berekeningen. Dit gebeurt door de taak te verdelen in grote processen die hoofdzakelijk werken op een privé verzameling gegevens. Men spreekt ook wel van parallelle computers met een grote korrel (coarse grain).

Een tweede methode bestaat uit het snel en eenvoudig maken van de communicatie tussen berekeningen. Dit gebeurt dan door speciale hardware voorzieningen, en door het programma in een speciaal formaat te coderen. Omdat de communicatie snel is kunnen de processen zeer klein gemaakt worden, vergelijkbaar met de grootte van één enkele instructie in een conventionele computer. Men spreekt dan van kleine korrel (fine grain) computers. Deze strategie is geschikt voor toepassingen waarin veel parallellisme voorkomt, maar waarbij de structuur onregelmatig is, d.w.z. niet a priori vastgelegd.

6.2. Dataflow programma's

Een programma voor een conventionele computer (control flow programma) bevat twee soorten verwijzingen, zie fig.33. De getrokken pijlen refereren naar gegevens in het data geheugen, terwijl de gestippelde pijlen refereren naar instructies in het programma geheugen. De verwijzing naar instructies kan impliciet zijn (de volgende instructie in het geheugen), of expliciet (een adres in de instructie geeft aan waar de volgende uit te voeren instructie in het geheugen staat). Bij parallel processing vormt de coördinatie van de twee soorten verwijzingen het grootste probleem. In de figuur kunnen de vermenigvuldiging en de aftrekking alleen gelijktijdig plaats vinden indien de variabele a voor beide operaties toegankelijk is.

In een dataflow programma wordt elke instructie beschouwd als een afzonderlijk proces, en de verwijzing naar andere instructies gebeurt door de data zelf. Een instructie die een waarde kan produceren, bevat pointers naar alle instructies die de

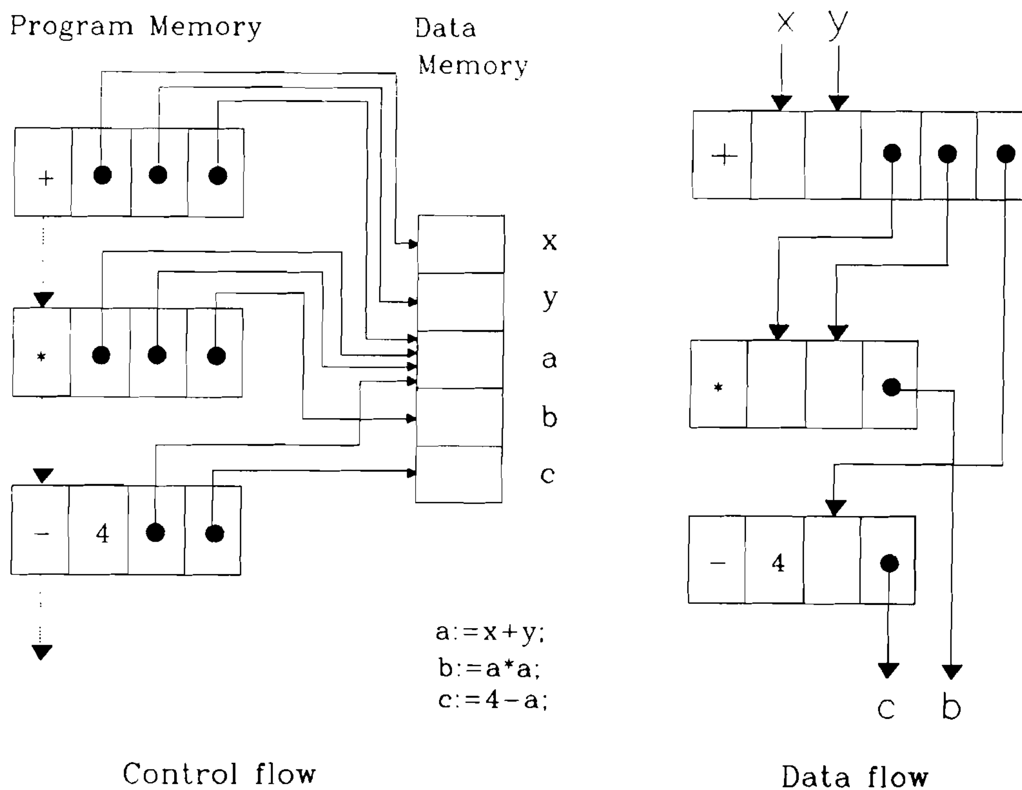

Fig.33. Control flow versus data flow.

waarde consumeren. De uitvoering van instructies is "data-driven", d.w.z. dat een instructie klaar is voor executie als de bijbehorende operanden gearriveerd zijn. De pijlen in de figuur geven direkt de afhankelijkheden van gegevens weer: de vermenigvuldiging en de aftrekking kunnen gelijktijdig plaats vinden zodra de optelling voltooid is.

Een dataflow programma wordt gewoonlijk weergegeven in een "dataflow graph". Fig.34 geeft de dataflow graph weer van het programma van fig.33. Hierin komen knooppunten voor die "actors" genoemd worden. De actors in deze figuur zijn allemaal operatoren. De actors worden onderling doorverbonden door takken waarop "tokens" kunnen staan. Een instructie komt overeen met een actor en zijn uitgaande takken (en eventueel de bijbehorende constanten).

Tokens worden geplaatst en verwijderd van de takken volgens "firing rules". Een actor is enabled als er op elke ingangstak een token staat, en er geen tokens aanwezig zijn op de uitgangstakken. Een actor die enabled is kan afgevuurd worden, hetgeen in het geval van een operator betekent: verwijder een token van elke ingangstak, pas de gespecificeerde functie toe op de waarden die met deze tokens geassocieerd zijn, en plaats tokens met de waarde van het resultaat op elke uitgangstak.

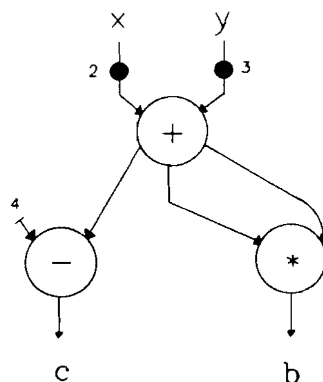

Fig.34. Dataflow graph van het programma uit fig.33.

In fig.34 is de plus-operator enabled. Als deze afvuurt worden de twee tokens op de ingangstakken verwijderd, en een token met het resultaat van de optelling (5) geplaatst op de drie uitgangstakken. Hierna zijn de twee onderste actors enabled (de ingang met de constante bij de min-operator vormt geen tak, en hoeft dus ook geen token te bevatten). Omdat er geen takken zijn tussen deze twee actors kunnen ze in een willekeurige volgorde afvuren, en zoals we later zullen zien kunnen ze ook gelijktijdig afvuren mits er voldoende processing elements beschikbaar zijn. Actors die meer tokens produceren dan dat ze consumeren, vergroten de hoeveelheid parallellisme van het dataflow programma.

De representatie van een programma in het geheugen van een dataflow computer maakt gebruik van "activity templates" [DEN79]. Een activity template correspondeert met één of meer actors van een dataflow programma, zie fig.35. De verschillende velden van een template die op opeenvolgende lokaties in het geheugen worden opgeslagen zijn achtereenvolgens:

- een opcode die de uit te voeren operatie specificeert;
- nul of meer velden die een constante bevatten;
- één of meer ontvangers die gevuld kunnen worden met operand waarden;
- één of meer bestemmingen die specificeren wat met het resultaat van de operatie moet gebeuren.

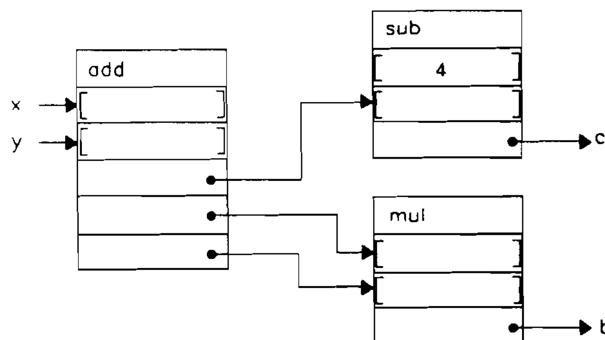

Fig.35. Representatie met activity templates.

Een "instructie" in een dataflow programma bestaat uit dat gedeelte van een activity template dat constant is:

<instruction> ::= <opcode> [<constants>] <destinations>.

Een bestemmingsveld bevat het adres van een activity template, en een integerwaarde "input slot" die aangeeft welke ontvanger van de template de bestemming is:

<destination> ::= <address> <input slot>.

6.3. Pipelining in dataflow programma's

Met de representatie van een dataflow graph door templates zoals hierboven beschreven is, kan het tweede gedeelte van de firing rule, dat een actor niet kan afvuren als een uitgangstak tokens bevat, niet zonder meer geïmplementeerd worden. Toch is dit gewenst omdat daarmee pipelining mogelijk wordt. Stel dat de uitgangstakken van de operatoren voor de aftrekking en de vermenigvuldiging in fig.34 nog tokens bevatten die gebruikt moeten worden door actors die de variabelen b en c nodig hebben. De plus-operator zou dan opnieuw kunnen afvuren indien er tokens voor x en y aanwezig zijn. Na het afvuren van de plus-operator moet dan verhinderd worden dat de twee andere actors ook afvuren, omdat daarmee de vorige waarden van b en c verloren zouden gaan.

Dit wordt bereikt door gebruik te maken van bevestigingssignalen (acknowledgements) die op dezelfde manier gegenereerd worden als het resultaat van een operatie. Fig.36 laat zien hoe fig.35 uitgebreid kan worden met deze signalen. Naast de opcode van een template staat nu hoeveel bevestigingen ontvangen moeten zijn voordat de betreffende actor opnieuw kan afvuren. Verder wordt de template uitgebreid met net zo veel velden als er bevestigingen verzonden moeten worden. In een acknowledge veld staat een bestemming om aan te geven naar welk template de bevestiging gestuurd moet worden, alsmede een identificatie om aan te geven dat het hier om een acknowledgement en niet om

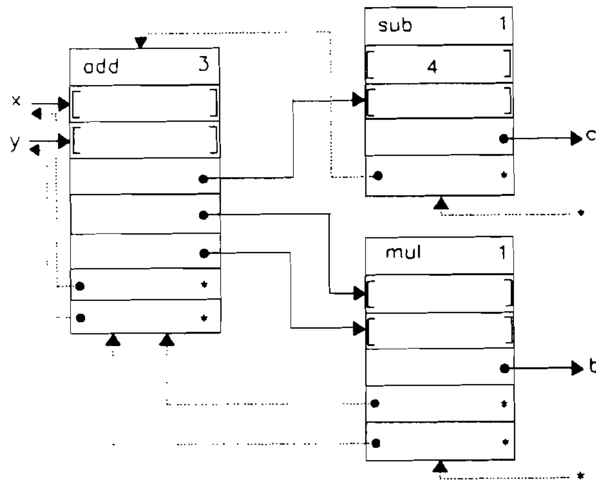

Fig.36. Activity templates met acknowledgements.

een resultaat gaat (in de figuur is dit aangegeven met een "*").

De actors die we tot nu toe gezien hebben zijn operatoren. Om condities en programmalussen te kunnen implementeren zijn echter ook "control actors" nodig. De dataflow graph in fig.37 geeft de berekening van x mod 3 weer, ofwel:

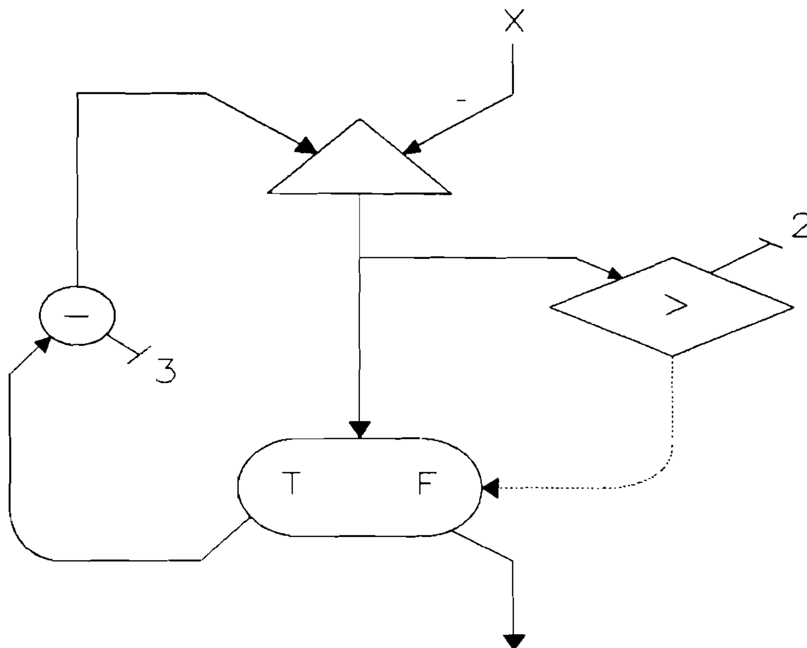

Fig.37. Dataflow graph van een programmalus.

while x>2 do x:=x-3 od.

Boven in de figuur komen we een "merge" actor tegen. Deze heeft een afwijkende firing rule omdat het hier volstaat om af te vuren als op één van de ingangstakken een token aanwezig is. (De merge actor heeft twee uitgangstakken hoewel dat getekend is als één uitgang met een aftakking.) De vergelijker is een operator die een logische waarde (true of false) als resultaat kan geven, vandaar dat de uitgangstak als een stippellijn is weergegeven. Deze tak komt uit op de besturingsingang van een "switch" actor. Als op de besturingsingang een token met de waarde true staat, dan wordt een token op de data-ingang doorgegeven naar de T-uitgang, terwijl bij een false een token op de data-ingang doorgegeven wordt naar de F-uitgang.

De implementatie met behulp van templates is weergegeven in fig.38. Het blijkt dat de merge actor gerealiseerd wordt door bestemmingsvelden van twee verschillende templates naar dezelfde ontvanger te laten wijzen. Dit heeft wel consequenties voor

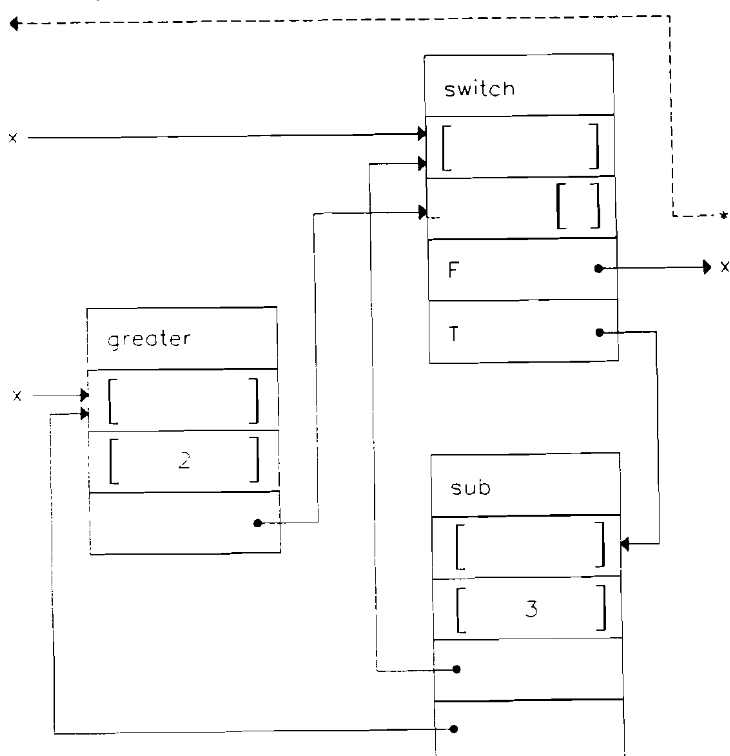

Fig.38. Implementatie van de lus van fig.37 met templates.

de mate van pipelining die mogelijk is. Stel namelijk dat de switch actor afvuurt. Deze moet dan twee acknowledgements geven, waarvan één naar de vergelijker. Om de tweede bevestiging te kunnen geven moet de actor weten of een token ontvangen is van de verschil-operator, of van de x-ingang. Aangezien dit niet bekend is moet een eventuele bevestiging die de x-ingang verwacht, geleverd worden door de actor die aangesloten is op de uitgang (in de figuur aangegeven door de "*" en de gestreepte lijn).

Dit betekent dat de complete lus ongeschikt is gemaakt voor pipelining: de x-ingang mag pas een nieuw token leveren als het resultaat van de lus verwerkt is. In het geval van fig.37 maakt dat niet zo veel uit omdat de lus uit slechts één operator bestaat. Zou de verschil-operator echter vervangen worden door een grotere dataflow graph met vele operaties en geneste lussen, dan maakt dit wél veel uit. Later zullen we zien hoe tokens die behoren bij verschillende iteraties gelijktijdig in een lus toegelaten kunnen worden.

6.4. Dataflow architecturen

Fig.39 geeft de werking weer van een processing element (PE) dat gebruikt zou kunnen worden in een dataflow computer. Het dataflow programma is opgeslagen in de "activity store" als een verzameling van activity templates. Elk template in dat geheugen heeft een uniek adres. De "instruction queue" bevat adressen van templates die gereed zijn voor executie. De "fetch unit" leest nu een adres uit de instruction queue en haalt de bijbehorende template op uit de activity store. Uit de gegevens van een template wordt vervolgens een "operation packet" gevormd dat de volgende vorm heeft:

<operation packet> ::= <opcode> <operands> <destinations>.

De <operands> zijn waarden die ontvangen zijn van andere actors, eventueel aangevuld met constanten. Het operation

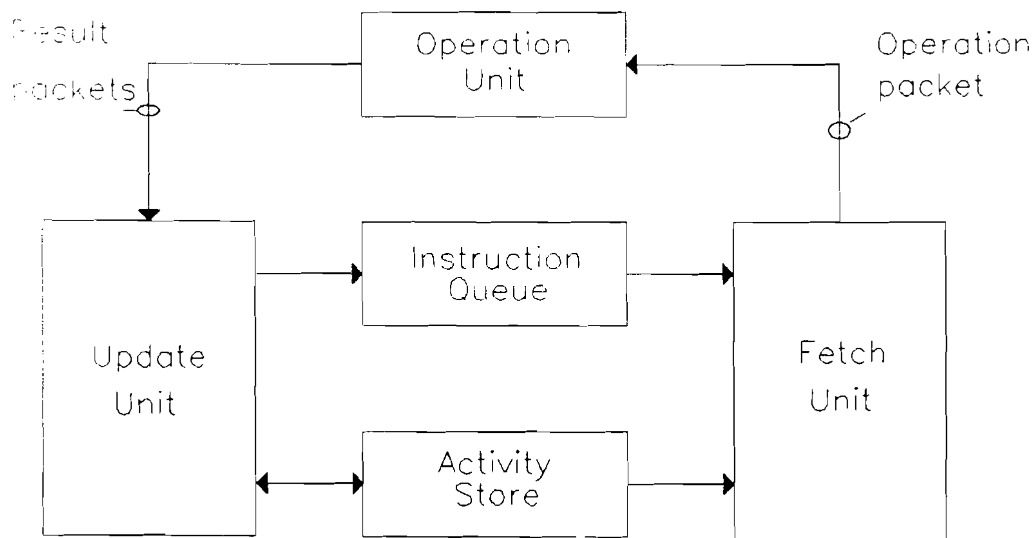

Fig.39. Een dataflow processing element.

packet wordt daarna verzonden naar de "operation unit" die voor elke bestemming een "result packet" genereert:

<result packet> ::= <value> <destination>.

Hierin kan <value> een acknowledgement zijn, het resultaat van een operatie (in geval van een operator), of een van de operanden (in geval van een control actor). Als de "update unit" een result packet ontvangt, dan wordt <value> weggeschreven naar de geadresseerde lokatie in de activity store. Verder gaat de update unit na of nu alle operand- en acknowledgement packets die nodig zijn voor het aktiveren van de bestemmingsinstructie ontvangen zijn. Indien dat het geval is, dan slaat de update unit het adres van de instructie op in de instruction queue.

Gedurende programma executie is het aantal adressen in de instruction queue een maat voor de hoeveelheid parallellisme die aanwezig is het programma. Een probleem treedt op wanneer de instruction queue vol raakt. Als dit het geval is, dan moet de fetch unit het afvuren van een actor uitstellen indien de template meer dan één bestemming heeft. Omdat de volgorde waarin actors die enabled zijn afgevuurd worden niet van belang is, kan dit gedaan worden door het adres van de template achter in de queue te plaatsen. Als de queue net zo veel adressen kan

bevatten als dat er templates in de activity store zijn, dan wordt het probleem helemaal vermeden.

Om het parallellisme dat een dataflow graph biedt inderdaad te benutten, is het mogelijk meerdere operation units te gebruiken. Deze kunnen allemaal identiek zijn zodat de fetch unit een operation packet naar een willekeurige operation unit die vrij is kan sturen. Een andere mogelijkheid is dat er verschillende operation units zijn voor verschillende operaties. In dat geval bepaald de opcode van een template waar het operation packet naar toe gestuurd moet worden.

De communicatie tussen de verschillende units van een PE gebeurt hoofdzakelijk met operation en result packets. Daardoor kunnen de verschillende eenheden gelijktijdig aktief zijn. Terwijl de operation unit een operatie verricht, kan de fetch unit bezig zijn met het samenstellen van een volgend operation packet, en kan de update unit bezig zijn met het verwerken van de resultaten van een vorige operatie. Het geheel wordt daarom ook wel een "circulaire pipeline" genoemd.

6.5. Dataflow multiprocessoren

We hebben tot nog toe twee manieren gezien waarop het parallellisme dat in een programma voorkomt benut kan worden binnen een PE: door de circulaire pipeline, en door meerdere operation units te gebruiken. In een dataflow multiprocessor is nog meer parallellisme mogelijk doordat meerdere PE's worden gebruikt, zie fig.40. De communicatie tussen de PE's gebeurt via een netwerk. Dat zou een vlindernetwerk kunnen zijn, of een boomstructuur kunnen hebben. Het eenvoudigste netwerk dat mogelijk is, heeft de vorm van een ring (PE's achter elkaar geschakeld).

De activity stores van de diverse PE's vormen te samen één grote adresruimte, waarin het adres van een bestemming uniek een template selekteert. Indien een dataflow programma voldoende parallellisme bevat, en zo lang de capaciteit van het communicatie netwerk groot genoeg is, kan de performance van de

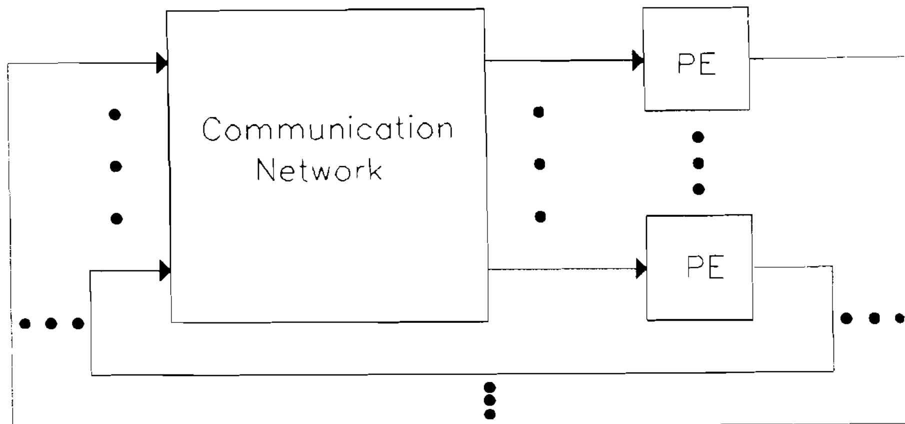

Fig.40. Een dataflow multiprocessor.

multiprocessor uitgebreid worden door PE's toe te voegen. Om te voorkomen dat het netwerk al snel de bottleneck vormt kan er gebruik gemaakt worden van het lokaliteitsprincipe. Hiervoor moet men er bij het programmeren van de multiprocessor voor zorgen dat de templates zodanig over de verschillende PE's gepartitioneerd worden, dat er zo weinig mogelijk communicatie nodig is [BUR82].

Fig.41 laat zien hoe de PE's op het netwerk aangesloten zijn: de verbinding tussen operation unit(s) en update unit is als

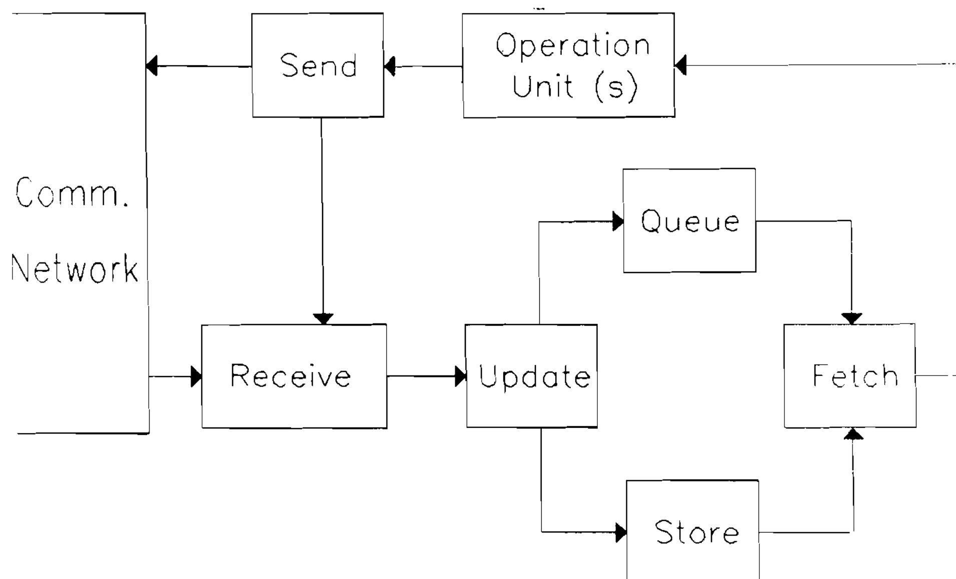

Fig.41. Aansluiting van een processing element op het netwerk.

het ware opengeknipt, en het (pakket geschakelde) netwerk is er tussen geplaatst. Als de "send unit" een result packet ontvangt dat bestemd is voor een lokale template, dan geeft deze het pakketje onmiddellijk door naar de "receive unit" zodat het netwerk niet onnodig belast wordt.

De besturing van een dataflow multiprocessor gebeurt met result packets, zodat er afgezien van het netwerk vrijwel geen verbindingen tussen de PE's nodig zijn. In de opgeslagen activity templates is vastgelegd welke instructies geëxecuteerd moeten worden, en de uitvoering van een instructie heeft tot gevolg dat result packets gemaakt worden die op hun beurt volgende instructies enablen. De vraag is nu hoe de diverse PE's geprogrammeerd worden, en hoe de communicatie met de buitenwereld (invoer van gegevens en uitvoer van resultaten) plaats vindt.

Het zou mogelijk zijn hiervoor elk processing element te voorzien van I/O-faciliteiten, maar dit heeft een negatieve invloed op de specifieke voordelen van een dataflow multiprocessor: de lokale verbindingen en de uitbreidbaarheid. Het ligt meer voor de hand hiervoor het communicatienetwerk te gebruiken zoals weergegeven in fig.42. De host-computer kan invoergegevens en instructies naar de PE's verzenden door er pakketjes van te

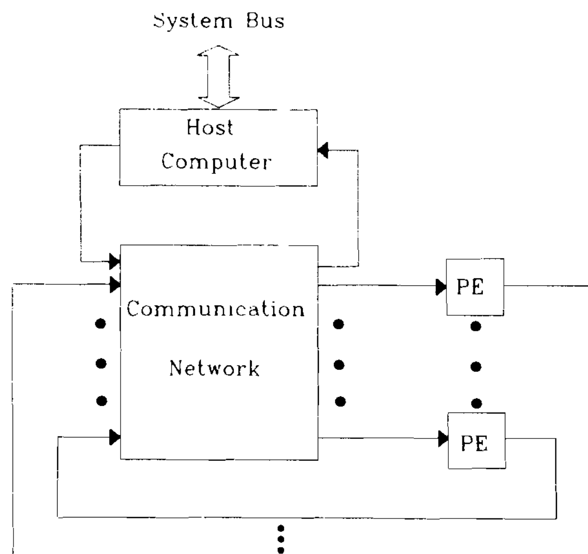

Fig.42. Besturing door de host computer.

maken, en deze het netwerk in te sturen. Een PE kan op dezelfde manier uitvoergegevens naar de host sturen.

6.6. Token labelling

In fig.38 hebben we gezien dat de hoeveelheid parallellisme die mogelijk is in een programmalus, sterk wordt beperkt als men gebruik maakt van het acknowledgement principe. Als we willen dat tokens die behoren bij verschillende iteraties gelijktijdig in een lus aanwezig kunnen zijn, dan moet er een mechanisme zijn waarmee die verschillende iteraties te onderscheiden zijn. Dit is mogelijk door tokens te voorzien van een label (ook wel "tag" of "color" genoemd). Fig.43 geeft als voorbeeld een programmalus waarin de volgende berekening wordt uitgevoerd:

S:=0; <u>for</u> x:=n-1 <u>downto</u> 0 <u>do</u> S:=S+f(x) <u>od</u>.

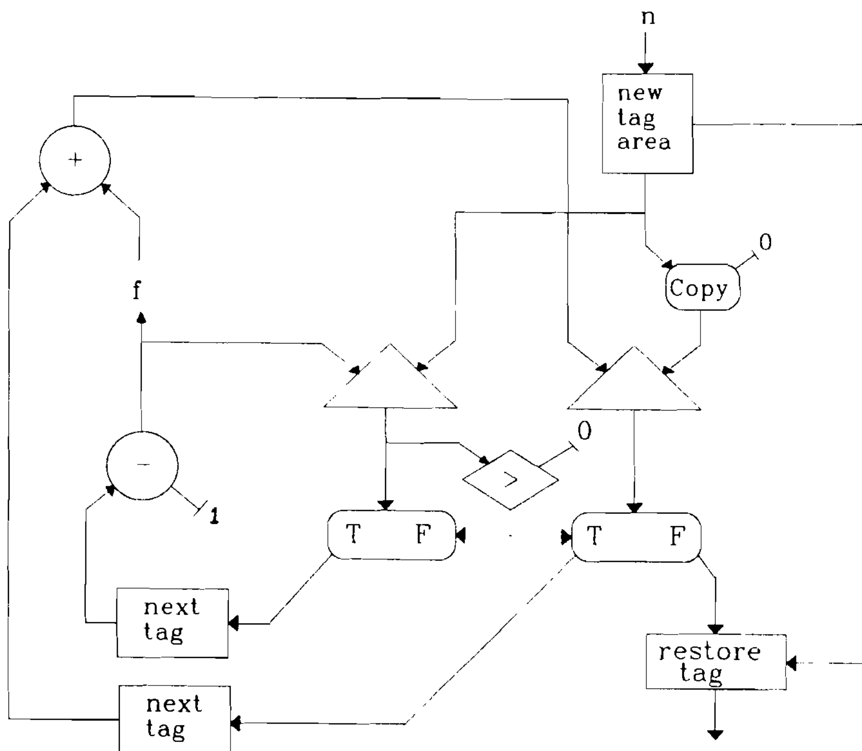

Fig.43. Een programmalus met token labelling.

Aan het begin van de lus wordt door de actor "new tag area" een nieuw label gereserveerd. Het eerste token dat de actor "next tag" bereikt, krijgt het eerste label (bijvoorbeeld het getal nul). Volgende tokens krijgen opvolgende labels (bijvoorbeeld 1,2,...), zodat de labels geordend zijn binnen de lus. De firing rule wordt nu: een actor kan afvuren als op elke ingangstak een token aanwezig is met identieke labels. Het afvuren heeft tot gevolg dat op de uitgangstak een token komt te staan met dezelfde labels als de tokens op de ingangstakken.

Tokens die behoren bij dezelfde iteratie hebben dus dezelfde labels, en actors werken alleen op tokens die bij elkaar horen. Door voor elk niveau een nieuw label te reserveren wordt het mogelijk lussen te nesten. Het nesten van subroutines gebeurt op een vergelijkbare manier.

Dataflow machines die gebruik maken van acknowledging om pipelining mogelijk te maken worden statisch genoemd. Architecturen met token labelling noemt men dynamisch. Fig.44 geeft het blokschema van een PE voor dynamische dataflow machines [VEE88]. Vergeleken met fig.39 zijn de update unit en instruction queue nu vervangen door een "match unit" en een "matching store". Voor elk result packet dat de match unit ontvangt wordt nagegaan of alle benodigde operanden met dezelfde labels ontvangen zijn. Is dit niet het geval, dan wordt het token zolang opgeslagen in de matching store. Als wel alle operanden ontvan-

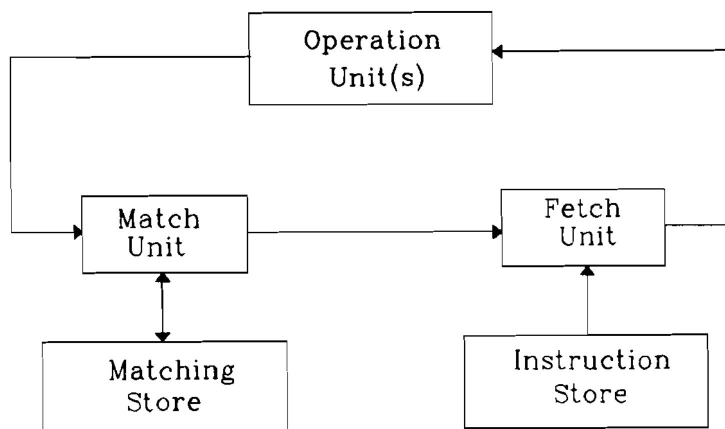

Fig.44. Een processing element voor token labelling.

gen zijn, dan geeft de match unit deze door naar de fetch unit die er een operation packet van maakt.

De match unit moet dus weten hoeveel ingangstakken een geadresseerde actor heeft. Om het geheel echter niet te complex te laten worden, beperkt men meestal het maximaal aantal ingangstakken dat een actor kan hebben tot twee. Ieder token krijgt dan een extra bit toegevoegd dat aangeeft of de geadresseerde actor monadisch of dyadisch is. Result packets voor monadische actors kunnen altijd onmiddellijk doorgegeven worden naar de fetch unit. Alleen voor dyadische actors moet de match unit nagaan of de matching store reeds een pakket bevat met dezelfde bestemming en identieke labels.

7. SYSTOLISCHE ARRAYS

7.1. Inleiding

Een systolisch systeem bestaat uit een verzameling cellen (processing elements) die ieder een (meestal eenvoudige) operatie kunnen uitvoeren. De cellen in een systolisch systeem zijn zodanig met elkaar verbonden dat er een boomstructuur of een één- of twee-dimensionale matrix ontstaat. De informatie in een systolisch systeem "stroomt" tussen cellen op een manier die te vergelijken is met de informatiestroom in een pipeline.

In fig.45 is het basisschema van een één-dimensionaal systolisch array gegeven. Het verschil met een conventioneel computersysteem is dat één enkel processing element (PE) vervangen is door een array van PE's. Data items worden opgehaald uit het geheugen, en kunnen daarna effectief gebruikt worden in elke cel die ze passeren. Hierdoor is het mogelijk een grote performance te behalen zonder dat een grote bandbreedte van het geheugen nodig is.

De communicatie tussen de cellen van een systolisch array wordt gesynchroniseerd door een communicatie klok. Dit betekent dat op van te voren vastgestelde tijden elke cel in de array één of meer data items doorgeeft naar zijn buur of buren, en gelijktijdig data items afkomstig van zijn buur of buren ontvangt. Daarnaast kan het zijn dat voor het rekenwerk in een cel óók een klok nodig is, de reken klok. Er zijn nu drie mogelijkheden:

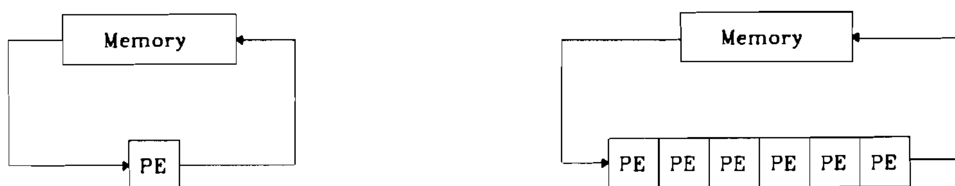

Fig.45. Conventioneel computersysteem versus systolisch array.

1. De reken klok is dezelfde als de communicatie klok. Dat wil dus zeggen dat alle berekeningen in één klokcyclus uitgevoerd moeten kunnen worden.
2. De communicatie klok wordt afgeleid van de reken klok door middel van frequentiedelers.
3. De communicatie klok wordt binnen een cel met behulp van een PLL[9] omhoog getransformeerd tot de reken klok.

Systolische systemen kunnen toegepast worden voor berekeningen waarbij herhaalde operaties moeten worden uitgevoerd op elk data item. Meestal worden systolische systemen niet als standalone computer gebruikt, maar als uitbreiding voor specifieke taken in een host computer. Voorbeelden van toepassingen zijn signaal- en beeldbewerking, matrix vermenigvuldiging, LU-decompositie en convolutie berekeningen. Ter illustratie wordt hier een voorbeeld gegeven voor de laatstgenoemde toepassing.

Het convolutie probleem luidt als volgt:

Gegeven een reeks gewichten $\{w_1,w_2,\ldots,w_k\}$ en een invoer reeks $\{x_1,x_2,\ldots,x_n\}$.
Bepaal de uitvoer reeks $\{y_1,y_2,\ldots,y_{n+1-k}\}$ gedefinieerd door $y_i = w_1x_i + w_2x_{i+1} + \ldots + w_kx_{i+k-1}$.
(N.B. voor n=k volgt $y = y_1 = w_1x_1 + w_2x_2 + \ldots + w_kx_k$, het inprodukt van de twee gegeven reeksen.)

7.2. Semi- en zuiver-systolische arrays

In de figuren 46,47 en 48 zijn een aantal systolische ontwerpen gegeven voor het oplossen van het convolutie probleem. In alle tekeningen is aangenomen dat k=3.

In fig.46 is een implementatie gegeven waarbij de gewichten w_i in de cellen blijven staan, de resultaten y_i systolisch door de array bewegen en de invoerwaarden x_i gelijktijdig naar alle cellen verzonden worden ("broadcasted inputs"). Aan het begin van de eerste cyclus wordt de waarde van x_1 verzonden naar de

[9]Phase Locked Loop

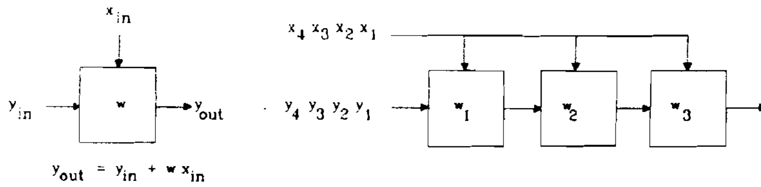

Fig.46. Systolisch array met broadcasted inputs.

verschillende cellen, en wordt y_1, geïnitialiseerd op nul, aangeboden aan de uiterst linkse cel. Gedurende de eerste cyclus wordt het produkt w_1x_1 berekend en opgeteld bij y_1. Het resultaat wordt doorgegeven naar de volgende cel. In de tweede cyclus worden w_1x_2 en w_2x_2 berekend en opgeteld bij respectievelijk y_2 en y_1. Vanaf de derde cyclus komen de uiteindelijke waarden van $y_1, y_2, \ldots$ beschikbaar op de uitgang van de uiterst rechtse cel.

In de implementatie van fig.47 blijven de tussenresultaten y_i in een cel staan (de cellen bevatten nu een vermenigvuldiger en een accumulator), terwijl de gewichten w_i systolisch door de array circuleren. Het eerste gewicht w_1 heeft een "tag bit". Dit bit geeft aan de accumulator de opdracht zijn inhoud naar buiten te voeren, en daarna te resetten (initialiseren op nul). De invoerwaarden x_i worden weer verzonden naar alle cellen. De resultaten worden naar buiten gevoerd via de stippellijn die aangesloten is op alle cellen.

Fig.48 geeft een ontwerp weer waarbij de gewichten net als in fig.46 in de cellen blijven staan. De invoerwaarden bewegen nu systolisch door de array, en de resultaten verschijnen op de uitgang van de opteller. Deze techniek waarbij de uitgangen van

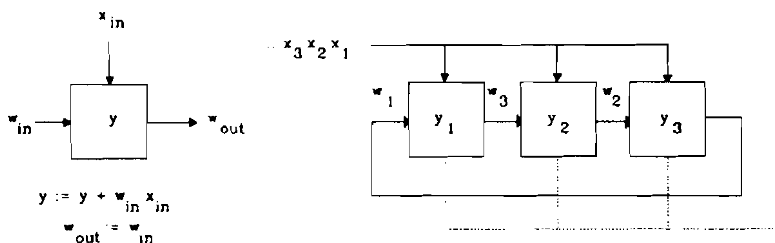

Fig.47. Systolisch array met accumulatoren.

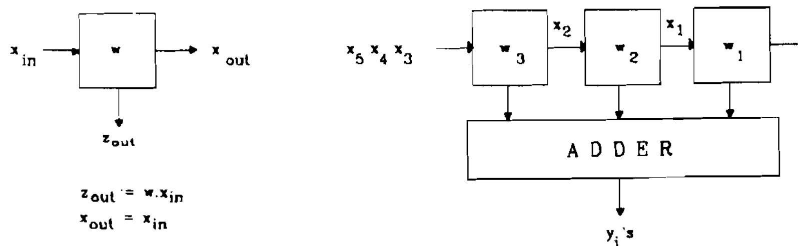

Fig.48. Systolisch array met fan-in.

meerdere cellen samengevoegd worden tot één resultaat (het tegenovergestelde van broadcasting) heet "fan-in". Als k, het aantal cellen, erg groot is, dan kan de vertragingstijd van de opteller (geïmplementeerd als een boom van kleinere optellers) groter worden dan de tijd die de vermenigvuldigers in de cellen nodig hebben. In dat geval heeft het voordeel pipelining toe te passen in de boom van optellers.

De hierboven gegeven voorbeelden zijn zogenaamde "semi-systolische" arrays, omdat broadcasting of fan-in wordt gebruikt. De implementatie hiervan is echter moeilijk. Om een data item te verzenden naar (of te ontvangen van) alle cellen in de array, is een bus of een netwerk met een boomstructuur nodig. Uitbreidingen van de array zijn alleen mogelijk als ook de paden voor de niet-lokale communicatie worden aangepast (i.v.m. zwaardere belasting of bredere boom). Er bestaan echter ook systemen voor convolutieberekeningen zonder globale communicatie. Deze "zuiver-systolische" arrays maken gebruik van het gelijktijdig systolisch bewegen van twee of meer data stromen, al of niet met verschillende snelheden, en al of niet in verschillende richtingen [KUN82].

Fig.49 geeft een voorbeeld van zo'n zuiver-systolisch systeem. Hier zijn de gewichten w_i in de cellen opgeslagen, de invoerwaarden bewegen systolisch van links naar rechts, en de tussenresultaten bewegen systolisch van rechts naar links. Merk op dat opeenvolgende x_i's en y_i's gescheiden worden door twee klokcycli (in de figuur aangegeven door de nullen). Dit heeft tot gevolg dat op elk tijdstip slechts de helft van het aantal cellen effectief wordt gebruikt. Dit nadeel is eventueel te

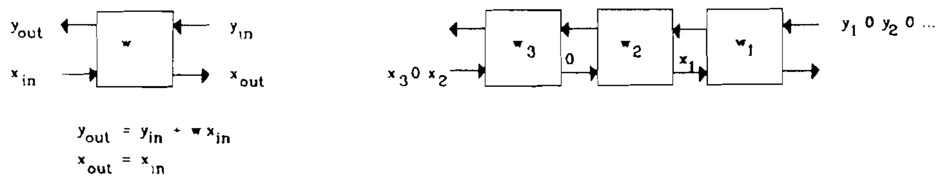

Fig.49. Zuiver-systolisch array.

vermijden door twee onafhankelijke convolutie berekeningen gelijktijdig in dezelfde array uit te voeren ("interleaving").

7.3. Eigenschappen van systolische arrays

Een unieke eigenschap van de systolische aanpak is de modulaire en uitbreidbare structuur (tenminste voor één-dimensionale arrays). Dit heeft tot gevolg dat de kosten en de performance van een systeem proportioneel toenemen met het aantal cellen. Bij grote twee-dimensionale systemen echter kan het synchroniseren van de communicatie tussen alle cellen zeer moeilijk worden.

Verdere voordelen van systolische arrays zijn:

- de eenvoud van het ontwerp (door het gebruik van een groot aantal eenvoudige cellen die allemaal identiek zijn);
- de lokale communicatie (afgezien van de bedrading voor de voedingsspanning en een klok zijn er bij zuiver-systolische arrays geen lange of onregelmatige verbindingen nodig).

Het belangrijkste nadeel van systolische arrays is de specifieke toepassing waarvoor het systeem geschikt is ("special purpose system"). Dit is vooral dàn het geval als de operaties die de processing elements uitvoeren in de hardware zijn vastgelegd. Een eerste verbetering is mogelijk door de processing elements programmeerbaar te maken.

Ten tweede kan men hardware mechanismen (schakelaars) toevoegen waarmee de topologie van het systeem gewijzigd kan worden. Deze zogenaamde "herconfigureerbare" arrays hebben ook het voordeel

dat het in principe mogelijk wordt fout-tolererende systemen te maken. Het nadeel is echter dat de hardware overhead aanzienlijk kan zijn. Bij ontdekking van een fout in één van de cellen, moet om die cel heen een "bypass" gemaakt worden, en eventueel moet een aantal van de overige cellen geheel of gedeeltelijk opnieuw geprogrammeerd worden. Indien de array ontworpen is met redundante cellen, dan kan daarna de array met dezelfde performance als voorheen verder functioneren.

7.4. De besturing van systolische arrays

Fig.50 [ALL85 p.871] geeft een mogelijke realisatie van de systolische array van fig.49. De PE's hierin bestaan uit een vermenigvuldiger, een opteller, latches (of registers) en een geheugen voor de opslag van het gewicht w_i. Iedere cel in de array heeft nu een adres gekregen, en bij de initialisatie van de array moet de vector "cell control" in sequentiële volgorde de gewichten $w_1, w_2, \ldots, w_k$ met de juiste adressen bevatten. Het

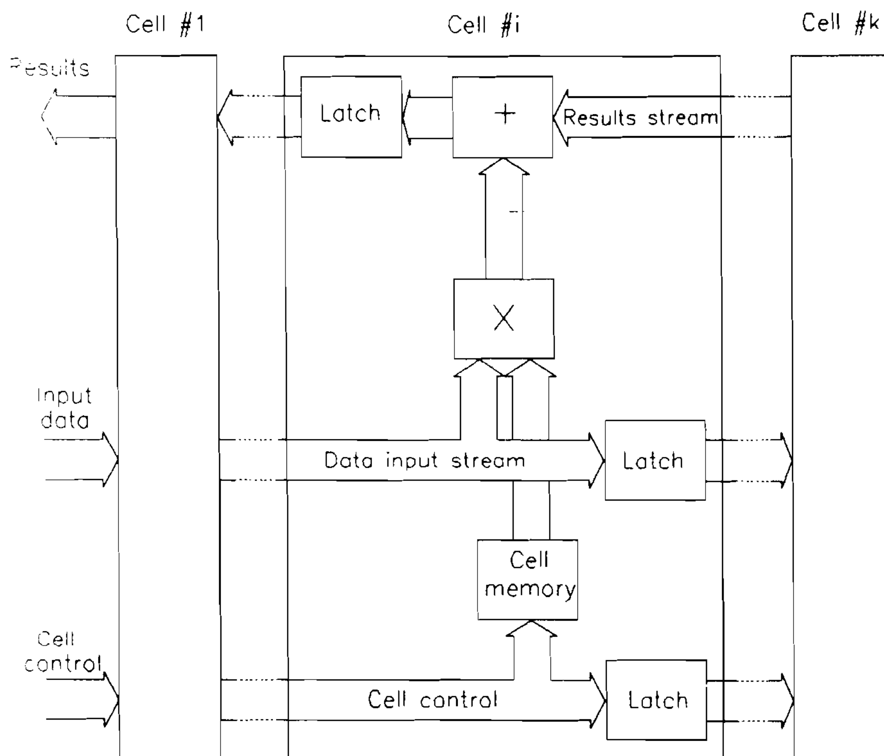

Fig.50. Implementatie van een systolisch array.

initialiseren van de PE's gebeurt hier dus op dezelfde systolische manier als de verplaatsing van invoergegevens en tussenresultaten.

De besturing van zo'n array van PE's zou kunnen gebeuren met een systolische processor zoals weergegeven in fig.51. Het blok "pre-processing" genereert de cell control vector, en eventueel kan hierin de invoer data gebufferd worden. In het blok "post-processing" kan (naast eventuele buffering van de uitvoer) bijvoorbeeld afronding en bepaling van extrema (minima en/of maxima) plaatsvinden. De "command interpreter" ontvangt commando's van een host computer, en bestuurt op zijn beurt de pre- en post-processing units. De interface tenslotte zorgt voor de aansluiting van de systolische processor op de systeembus van de host.

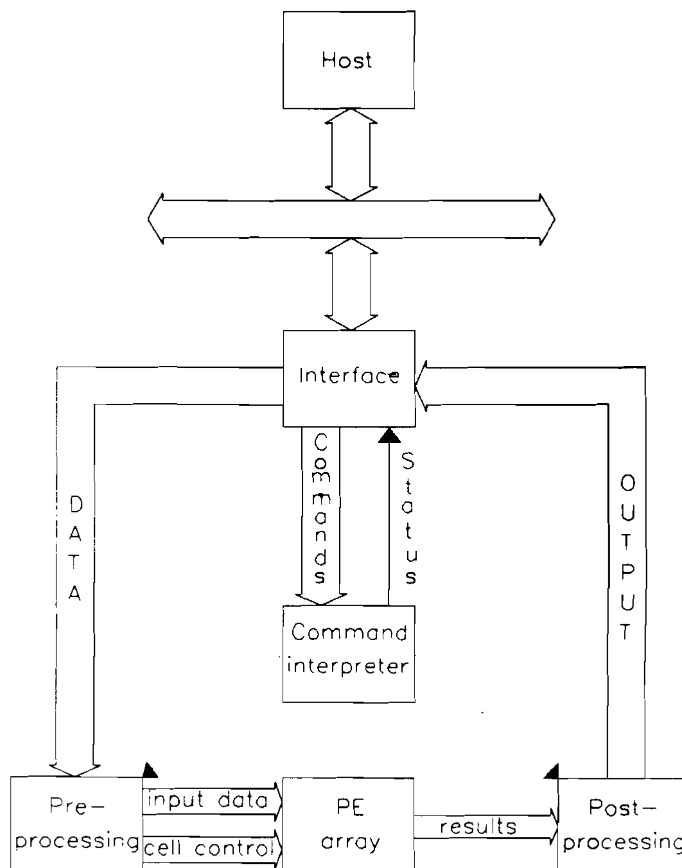

Fig.51. Architectuur van een systolische processor.

De systolische array van fig.50 kan maar voor één doel gebruikt worden, namelijk het berekenen van convolutiesommen. Dit special purpose karakter heeft wel tot gevolg dat het geheel snel en relatief eenvoudig is. Om een meer "general purpose" systeem te krijgen kunnen we de vermenigvuldiger en/of opteller vervangen door een ALU. We gaan er hier vanuit dat de ALU's in de verschillende cellen van de array dan allemaal dezelfde funktie uitvoeren, en dat deze funktie alleen kan veranderen nadat een commando van de host volledig verwerkt is.

In dat geval rijst de vraag welk blok de ALU-funktie moet genereren. De eerste mogelijkheid is de pre-processing unit hiervoor te gebruiken. De cell control vector moet dan een veld bevatten dat de ALU-funktie aangeeft, en de inhoud van dit veld wordt samen met eventuele constanten opgeslagen in het lokale geheugen van de cellen. De constanten kunnen voor elke cel verschillend zijn, maar de ALU-funktie is voor elke cel hetzelfde. Dit geeft dan ook meteen het nadeel van deze methode aan: bij een array bestaande uit k cellen moet dezelfde ALU-funktie k keer opgeslagen worden.

Als tweede mogelijkheid kan men de command interpreter gebruiken voor de ALU-besturing. Deze genereert dan een "array control" vector die direkt aangesloten is op alle PE's ("control broadcasting"). In dit geval is het niet nodig de ALU-funktie in elk PE apart op te slaan, zodat bespaard kan worden op hardware. Een bezwaar tegen deze methode zou kunnen zijn dat control broadcasting het principe van uitsluitend lokale communicatie aantast. Bij het uitbreiden van de array moet de array control vector extra gebufferd worden, hetgeen timing problemen zou kunnen veroorzaken. De timing van de communicatie tussen command interpreter en de array is echter niet zo kritisch als de timing van de communicatie tussen PE's onderling. Dit komt omdat de ALU-funktie alleen kan veranderen nadat een nieuw commando van de host computer is ontvangen, en op dat moment doet het er niet toe welke operaties de PE's uitvoeren.

Om de systolische array nog meer geschikt te maken voor algemene doeleinden, kan men de enkele operatie die een PE uitvoert

vervangen door een algoritme. Hiertoe moet de interne architectuur van de PE's uitgebreid worden met bijvoorbeeld registers, bussen en bidirectionele verbindingen met de naburige cellen.

Voor de besturing van een PE zijn er nu weer twee mogelijkheden. Allereerst zou de command interpreter deze taak kunnen vervullen door op dezelfde manier als bij het aanbieden van de ALU-funktie gebruik te maken van control broadcasting. De PE's bestaan in dat geval alleen uit een datapad. Het aanbieden van de ALU-funktie gebeurde echter volledig statisch zodat de timing geen problemen opleverde. Nu is de besturing van de PE's dynamisch geworden. Om deze reden kunnen de datapaden van de PE's niet bestuurd worden door gebruik te maken van control broadcasting, maar zijn er lokale control units nodig. De processing elements worden vervangen door volwaardige processoren met lokale opslag van instructies. De PE's lopen echter wel synchroon, d.w.z. dat ze op elk moment dezelfde instructie uitvoeren (SIMD). Dit heeft onder andere tot gevolg dat er geen conditionele sprongen, afhankelijk van de invoer data gemaakt kunnen worden.

De globale besturing bestaat daarmee uit het programmeren van de cellen, en omdat dit programmeren ook iets dynamisch is, zal dit moeten gebeuren via de pre-processing unit en de cell control vector. Het programmeren gaat dan als volgt: De PE's worden in "program mode" gezet, bijvoorbeeld door een broadcast lijn van de interpreter, of door een extra bit in de cell control vector. Het programma wordt vervolgens instructie voor instructie de array ingeschoven. Een PE slaat de instructie op in het lokale geheugen, en geeft de instructie door naar zijn buurman.

Op deze manier ontstaat de meest uitgebreide hiërarchie van besturingen: de host computer geeft commando's aan de command interpreter, de command interpreter bestuurt de pre-processing unit, de pre-processing unit bestuurt de control units in de PE's (door op systolische wijze instructies de array in te sturen), en binnen een PE bestuurt de control unit het datapad.

Systolische arrays maken gebruik van het pipeline principe. Dat heeft tot gevolg dat het relatief lang duurt voordat het eerste resultaat van een berekening beschikbaar is. De daar op volgende resultaten komen echter bij iedere volgende cyclus van de communicatie klok beschikbaar. Naast de pipelining in de array kan er ook gebruik gemaakt worden van pipelining in de processing elements. Er ontstaat dan een "two-level pipelined array".

Nemen we als voorbeeld de array van fig.50, dan zou men voor de vermenigvuldiger een pipelined Wallace tree multiplier kunnen gebruiken zoals weergegeven in fig.24. De pipeline klok, reken klok en communicatie klok zijn in dit geval identiek. Er zijn nu meer klokslagen nodig voordat het eerste resultaat berekend is, maar de frequentie van de klok kan hoger gemaakt worden.

Het volgende voorbeeld dient om een idee te krijgen van de zeer hoge performance die te realiseren is met systolische arrays. Stel dat de (pipelined) vermenigvuldiger en de opteller in een PE iedere 100 ns een berekening uitvoeren. Dan is de throughput per PE 20 MOPS[10]. Aangezien een cel weinig logica bevat naast de vermenigvuldiger, kunnen waarschijnlijk meerdere PE's geïntegreerd worden in één IC. Gaan we uit van 16 PE's per IC, dan is met 16 van die IC's in serie de maximale throughput ruim 5 GOPS[11].

---

[10] 10^6 Operaties Per Seconde

[11] 10^9 Operaties Per Seconde

8. WAVEFRONT ARRAYS

De systolische arrays zoals die in het vorige hoofdstuk beschreven werden hebben een gemeenschappelijke communicatie klok die de timing van alle cellen in het systeem verzorgt. Indien een systolisch array programmeerbaar is, dan verloopt de executie van instructies in alle cellen synchroon. Afhankelijk van de data echter kan de uit te voeren berekening in de ene cel eenvoudiger zijn dan de uit te voeren berekening in een andere cel.

Nemen we als voorbeeld een twee-dimensionaal systolisch array voor de vermenigvuldiging van twee matrices. De vermenigvuldiging van twee getallen is veel sneller te berekenen indien één van de twee te vermenigvuldigen getallen gelijk aan nul is. De frequentie van de communicatie klok moet zo laag zijn dat de meest complexe operaties die kunnen voorkomen, bij de meest ongunstige invoergegevens, binnen de periodetijd van de klok uitgevoerd kunnen worden. In het geval dat de te vermenigvuldigen matrices veel nullen bevatten (zgn. "sparse matrices") kan dit een aanzienlijke beperking van de performance opleveren. Bovendien blijken grote twee-dimensionale systolische arrays moeilijk te synchroniseren, zodat ze niet onbeperkt uitbreidbaar zijn.

Om deze nadelen van systolische arrays te vermijden, maar de voordelen (de modulaire en regelmatige structuur) te behouden, kan men de eis laten vallen dat alle cellen op ieder tijdstip dezelfde instructie executeren, zodat een MIMD-systeem ontstaat. Dit heeft dan tot gevolg dat niet van te voren bekend is op welk tijdstip de communicatie tussen de cellen moet plaatsvinden.

Daarom maakt men gebruik van handshaking mechanismen tussen aangrenzende cellen. De communicatie klok komt dan te vervallen, en er ontstaan "self-timed" systemen die wavefront arrays genoemd worden [KUN87]. Wavefront arrays combineren het systolisch pipeline principe met het dataflow concept. Hoewel deze vorm van communicatie ook wel "asynchroon" genoemd wordt, hoeft

het systeem niet asynchroon ontworpen te zijn. De handshaking kan namelijk gesynchroniseerd zijn met de (gemeenschappelijke) reken klok.

Beschouwen we als voorbeeld een wavefront array (zie fig.52) voor de vermenigvuldiging van een M x N matrix $\mathbf{A} = \{a_{ij}\}$ en een N x M matrix $\mathbf{B} = \{b_{ij}\}$. Het resultaat van de vermenigvuldiging $C = \mathbf{A} \times \mathbf{B} = \{c_{ij}\}$ is een M x M matrix. De matrix **A** bestaat uit N kolommen $\mathbf{A}_j$, en de matrix B uit N rijen $\mathbf{B}_i$. Hiermee is **C** te schrijven als:

$$C = \mathbf{A}_1 * \mathbf{B}_1 + \mathbf{A}_2 * \mathbf{B}_2 + \ldots + \mathbf{A}_N * \mathbf{B}_N$$

waarin $\mathbf{A}_k * \mathbf{B}_k$ een matrix $\{d_{ij}^{(k)}\}$ is waarvoor geldt: $d_{ij}^{(k)}=a_{ik}b_{kj}$. De matrix vermenigvuldiging kan nu als volgt recursief geschreven worden:

$$C^{(k)} := C^{(k-1)} + \mathbf{A}_k * \mathbf{B}_k \quad \text{voor } k=1,2,\ldots,N.$$

De berekening van de eerste recursie $\mathbf{A}_1 * \mathbf{B}_1$ gaat als volgt (in fig.52 is uitgegaan van M=N=4). Het proces begint met PE (1,1) waar $c_{11}^{(1)} = c_{11}^{(0)} + a_{11}.b_{11}$ berekend wordt. Daarna geeft PE (1,1) de waarde van a_{11} door aan zijn rechter buurman (1,2), en de waarde van b_{11} aan zijn beneden buurman (2,1). De PE's (1,2) en (2,1) kunnen nu hun respectievelijke operaties uitvoeren en de operanden doorgeven aan hun naaste buren. Hierdoor worden de PE's (1,3), (2,2) en (3,1) geaktiveerd, enzovoort. Er ontstaat zo een golffront van verrichtte operaties in de array (aangegeven door de punt-streep lijn in de figuur), en zodra dit golffront PE (M,M) heeft bereikt is de eerste recursie $C^{(1)}$ klaar.

Op het moment dat het golffront PE (1,1) heeft verlaten is deze weer vrij om te beginnen aan de recursie $C^{(2)}$ hetgeen een tweede golffront veroorzaakt (aangegeven door de stippellijn). Door de pipelining die zo ontstaat kunnen een groot aantal PE's gelijktijdig aktief zijn.

Na de laatste recursie moet het resultaat dat in de diverse PE's staat (de matrix **C**), naar buiten uitgevoerd worden. In de

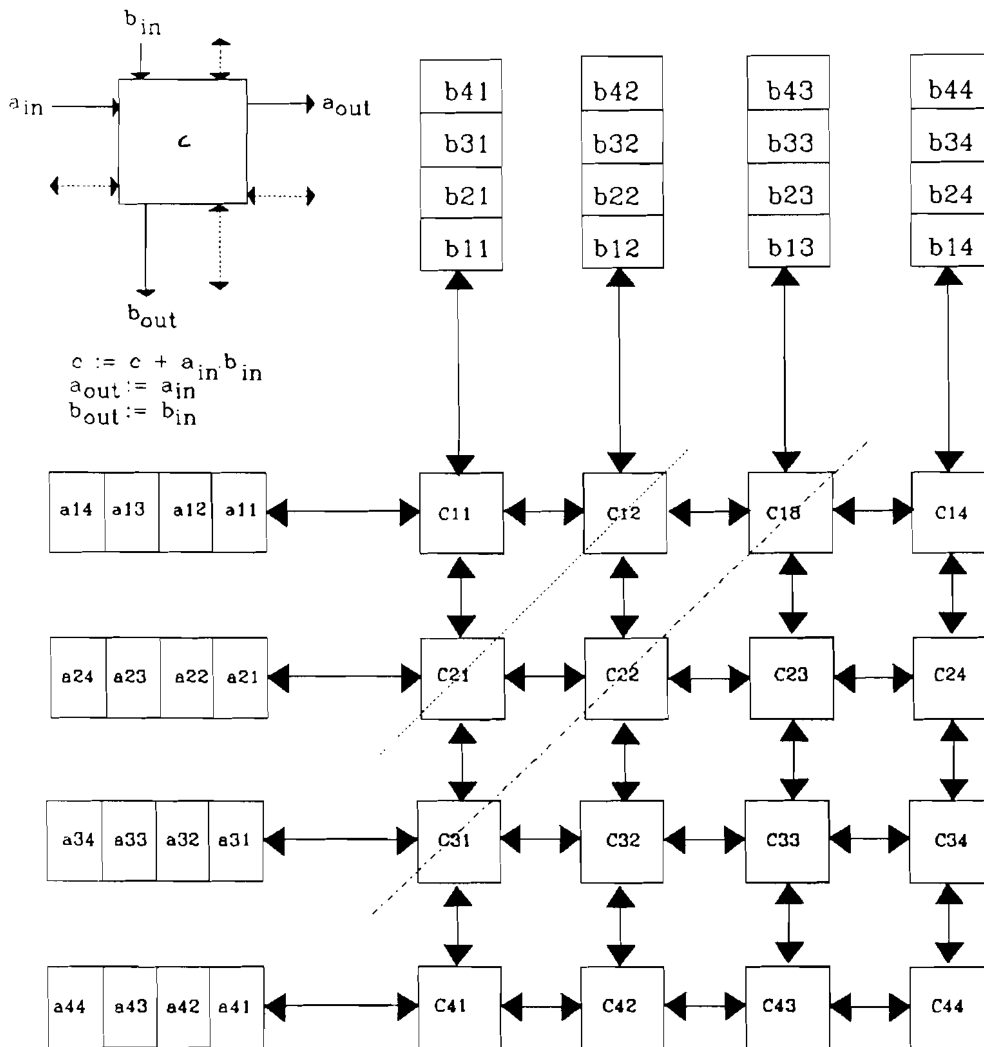

Fig.52. Wavefront array voor matrix vermenigvuldiging.

figuur is in het midden gelaten hoe deze uitvoer moet plaatsvinden. Een mogelijkheid is de meest rechtse cellen in de array, uit te breiden met een extra uitgang. Een cel (i,j) kan dan zijn resultaat c_{ij} doorgeven aan de cel (i,j+1) onmiddellijk nadat de laatste operand a_{iN} doorgegeven is. De uitvoer vindt nu plaats aan de cellen (i,M).

Om een idee te krijgen van de snelheidswinst ten opzichte van een systeem met één processing element, nemen we aan dat de berekening van een produkt en de accumulatie van het resultaat één tijdseenheid duurt. Voor ieder van de M^2 elementen van C zijn N van deze berekeningen nodig, zodat het systeem met één processing element M^2N tijdseenheden nodig heeft. Een array met

M x M PE's heeft voor de eerste recursie 2M-1 tijdseenheden nodig. Door de pipelining is voor elk van de volgende N-1 recursies slechts 1 tijdseenheid nodig, zodat er totaal 2M+N-1 tijdseenheden nodig zijn. De snelheidswinst wordt daarmee:

$$\frac{M^2N}{2M+N-2} \sim \frac{M^2N}{2M+N} \quad (M \gg 1).$$

In het geval dat N veel groter dan M is, levert dit een snelheidswinst op van ongeveer M^2 (in het ideale geval is elk PE vrijwel 100% van de tijd bezig). Als echter N gelijk aan M is, zoals aangenomen in fig.52, dan daalt de snelheidswinst tot $M^2/3$ (slechts één op de drie PE's wordt effectief gebruikt). In het meest ongunstige geval is N=1, en de snelheidswinst is dan ongeveer M/2.

Het verschil tussen systolische arrays en wavefront arrays is de "data-driven" eigenschap. In een wavefront array is er geen communicatie klok die de communicatie tussen processing elements synchroniseert, maar vindt de communicatie plaats door middel van een handshaking-protocol. Als een zendend PE data beschikbaar heeft, geeft hij dit door aan het ontvangend PE. De ontvanger geeft een bevestiging van ontvangst aan de zender, en zodra alle benodigde operanden ontvangen zijn kan een berekening beginnen.

Het voordeel van deze methode is dat de timing minder kritisch is, waardoor de array in principe onbeperkt uitbreidbaar is. Het handshaking mechanisme veroorzaakt weliswaar een tijdsvertraging die terugkeert bij elke data-uitwisseling, maar heeft ook het voordeel dat snelle berekeningen minder opgehouden worden door langzamere. Hiervoor is het nodig dat meerdere woorden gebufferd kunnen worden tussen twee cellen. Verder is het gewenst dat de data transfers gelijktijdig met berekeningen kunnen plaats vinden. Net als bij systolische arrays vindt er alleen lokale communicatie plaats tussen naburige PE's, hetgeen het voordeel heeft dat er vrijwel geen lange verbindingen nodig zijn.

Voor wat de besturing van wavefront arrays betreft, gelden vergelijkbare overwegingen als bij programmeerbare systolische arrays. Het programmeren van de cellen zal dus ook via een cell control vector (m.b.v. handshaking) moeten gebeuren.

Voor wat de realisatie van de PE's betreft, zijn er twee mogelijkheden. Ten eerste kan men deze zelf ontwerpen, zodat ze optimaal geschikt zijn voor een bepaalde toepassing. Hierbij valt te denken aan PE's die:

- volledig programmeerbaar zijn;
- programmeerbaar zijn door verwisseling van een (EP)ROM[12];
- niet-programmeerbaar zijn.

In het voorbeeld van fig.52 zijn de PE's dan gespecialiseerd in het vermenigvuldigen van twee getallen. Bovendien zullen de PE's meestal zo eenvoudig zijn dat er meerdere in één IC geïntegreerd kunnen worden.

Indien men een systeem wil ontwikkelen dat een groter toepassingsgebied heeft, of indien men in korte tijd een prototype wil maken voor een specifieke toepassing, heeft het voordeel uit te gaan van bestaande programmeerbare bouwstenen ("universal building blocks"). Een voorbeeld van een chip die hiervoor gebruikt kan worden is de "transputer" van de firma INMOS [GRA87, NEW86]. Deze heeft voor de communicatie met de naburige cellen de beschikking over vier bidirectionele seriële verbindingen (zogenaamde "links"). Het programmeren van een transputer in deze toepassingen gebeurt door het programma via de links in het lokale geheugen te laden.

---

[12] (Erasable Programable) Read Only Memory

9. VECTOR PROCESSING

Onder vector processing (ook wel synchroon parallel processing genoemd) verstaan we hier de uitvoering van identieke operaties op de elementen van een aantal vectoren met behulp van een array van parallel geschakelde processoren die allemaal dezelfde instructie executeren (SIMD). Het onderstaande stukje programma dat de som van twee vectoren berekend, zou op n processoren n keer zo snel uitgevoerd kunnen worden als op één processor, omdat elke processor één element voor zijn rekening neemt:

```
for i:=1 to n do c[i]:=a[i]+b[i] od;
```

In de praktijk zijn er echter weinig toepassingen die op deze manier gevectoriseerd kunnen worden, omdat vrijwel altijd een gedeelte van de uit te voeren operaties sequentieel moet gebeuren. De wet van Amdahl [RAF88 p.339] laat zien dat de efficiëntie van vector processing hoofdzakelijk bepaald wordt door dat gedeelte van een programma dat niet vectoriseerbaar is [DES84 p.100].

Om dit toe te lichten bekijken we een programma dat bestaat uit m operaties die serieel uitgevoerd moeten worden, en n operaties die parallel uitgevoerd kunnen worden:

```
s[1] := a[1] op1 s[0];
s[2] := a[2] op1 s[1];
  :
s[m] := a[m] op1 s[m-1];
p[1] := b[1] op2 c[1];
p[2] := b[2] op2 c[2];
  :
p[n] := b[n] op2 c[n];
```

In dit voorbeeld kan de berekening van s[i] niet plaatsvinden voordat de berekening van s[i-1] klaar is, zodat deze operaties achter elkaar moeten gebeuren. De n berekeningen van p[i]

kunnen echter wel gelijktijdig plaatsvinden. Verder gaan we voorlopig uit van de (vrij ideale) situatie dat er ook n processoren beschikbaar zijn. Als de tijd die nodig is voor een operatie gelijk is aan T, dan is de totale tijd voor de executie van het programma op n processoren gelijk aan (m+1)T. Aangezien de executietijd op één processor gelijk is aan (m+n)T is de snelheidswinst S(n) op n processoren gelijk aan:

$$S(n) = \frac{(m+n)T}{(m+1)T} = \frac{m+n}{m+1}$$

Als $\beta=m/(m+n)$ de fractie is van de operaties die serieel uitgevoerd moeten worden, dan geldt dus: $m=\beta n/(1-\beta)$. Hiermee vinden we voor de snelheidswinst:

$$S(n) = \frac{\beta n/(1-\beta) + n}{\beta n/(1-\beta) + 1} = \frac{\beta n+n(1-\beta)}{\beta n + 1 - \beta} = \frac{n}{\beta(n-1)+1}$$

In het algemene geval zal het aantal operaties dat parallel uitgevoerd kan worden niet gelijk zijn aan het aantal processoren. Daarom geeft bovenstaande formule een bovengrens voor de te behalen snelheidswinst. Dit resultaat dat bekend staat als de wet van Amdahl is weergegeven in fig.53 voor een drietal waarden van n. Het is duidelijk dat de snelheidswinst snel afneemt naarmate de fractie van de operaties die serieel uitgevoerd moeten worden toeneemt.

Net als bij pipelining wordt bij vector processing het begrip efficiency gebruikt. De efficiency van een systeem met n processoren is gedefinieerd als de snelheidswinst ten opzichte van een systeem met 1 processor, gedeeld door het aantal geïnstalleerde processoren n:

$$E = \frac{S(n)}{n} \leq \frac{1}{\beta(n-1)+1}$$

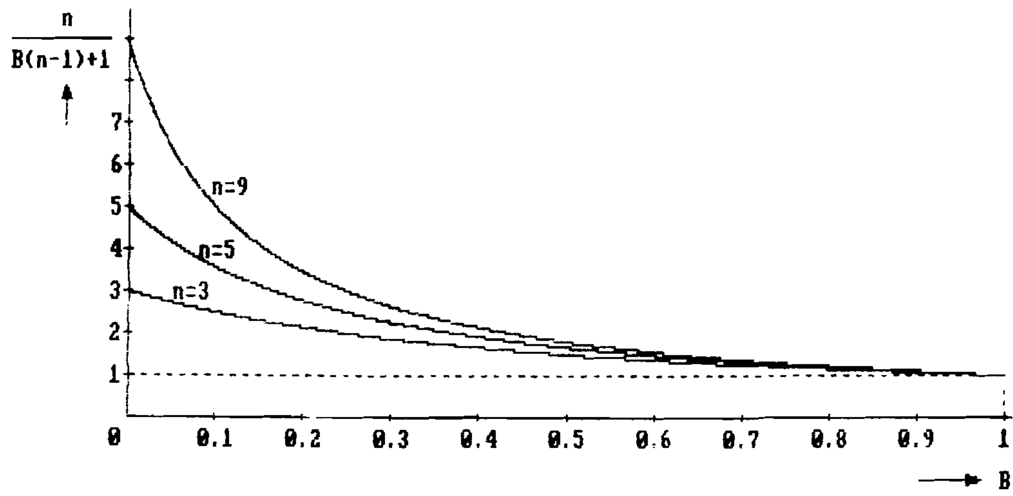

Fig.53. Wet van Amdahl (snelheidswinst als functie van de niet-vectoriseerbare fractie).

10. VOORBEELD VAN EEN COMBINATIE VAN ARCHITECTUREN

In grotere systemen, zoals mainframes en (mini)supercomputers zijn vaak meerdere van de genoemde architecturen en technieken terug te vinden. Als voorbeeld bekijken we de Alliant FX/8 minisupercomputer waarin bijna alle vormen van parallellisme voorkomen. De processoren in een FX/8 worden CE's (Computational Elements) genoemd [ALL86].

Allereerst is het mogelijk één CE te gebruiken in een SISD-configuratie, waarbij één of meer gebruikers op de processor zijn aangesloten. Op deze ene processor kunnen via time-sharing meerdere processen op schijnbaar hetzelfde moment verwerkt worden. Het voordeel hiervan is dat wanneer één van de processen moet wachten op de voltooiing van een I/O-operatie, dat dan de processor verder kan gaan met een ander proces. In feite is dit een vorm van parallellisme waarvoor vrijwel geen extra hardware nodig is.

De tweede vorm van parallellisme in de FX/8 ontstaat door gebruik te maken van pipelining (MISD). De executie van instructies gebeurt in een 5-segments pipeline, en bovendien heeft een CE arithmetische pipelines in de vorm van floating point units (voor vermenigvuldigen, delen en optellen/aftrekken), zie fig.54. Het aantal segmenten in deze vector units is maximaal 12. Om de aan- en afvoer van data voor de vector units te verzorgen is een CE voorzien van vector registers. Hierin kunnen 8 vectoren van 32 elementen van elk 64 bit breed worden opgeslagen. Twee andere belangrijke onderdelen van het blokschema in fig.54 zijn de "instruction processor" en de "control section".

De instruction processor bestaat uit een address unit en een integer/logic unit, en deze zijn elk weer opgebouwd uit een control unit en een datapad. Het datapad van de address unit bevat onder andere 8 registers, de programmateller, een 32-bit opteller en hulpregisters voor het implementeren van diverse adresseringsmethoden. Het datapad van de integer/logic unit

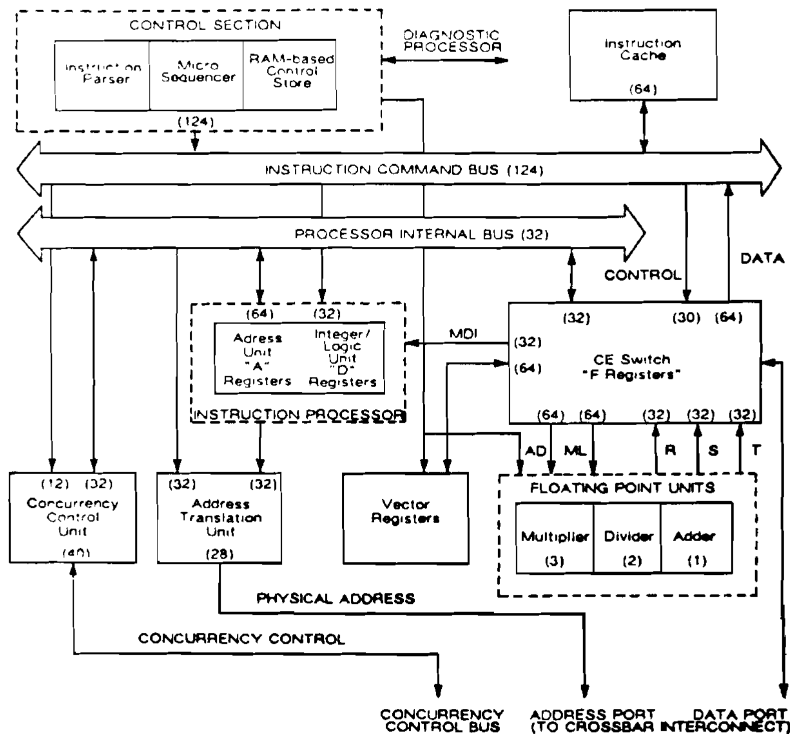

Fig.54. Computational Element (CE) van de Alliant FX/8.

bevat onder meer 8 data registers, een ALU en een barrel shifter.

Het control section gedeelte zorgt voor de besturing van de diverse onderdelen van een CE. De "instruction parser" hierin decodeert de opcodes en genereert startadressen voor de microprogrammeerbare besturing. De microcode bevat onder andere implementaties van transcendente functies als sin(x), arctan(x) en exp(x).

De derde vorm van parallellisme wordt gevormd doordat de FX/8 meerdere CE's kan hebben (tot maximaal 8). Omdat de CE's een gemeenschappelijk geheugen hebben (zie fig.55) en er sprake is van slechts één operating system ontstaat dan een echte multiprocessor (MIMD). Op elk van de CE's kunnen onafhankelijk van elkaar jobs verwerkt worden, zodat de throughput (het aantal jobs per tijdseenheid) vergroot kan worden. Door het toevoegen van CE's wordt ook de latency (de tijd die verstrijkt voordat een job klaar is) kleiner, omdat het gemiddelde aantal jobs per CE kleiner wordt. Zou men echter voor elke job die verwerkt

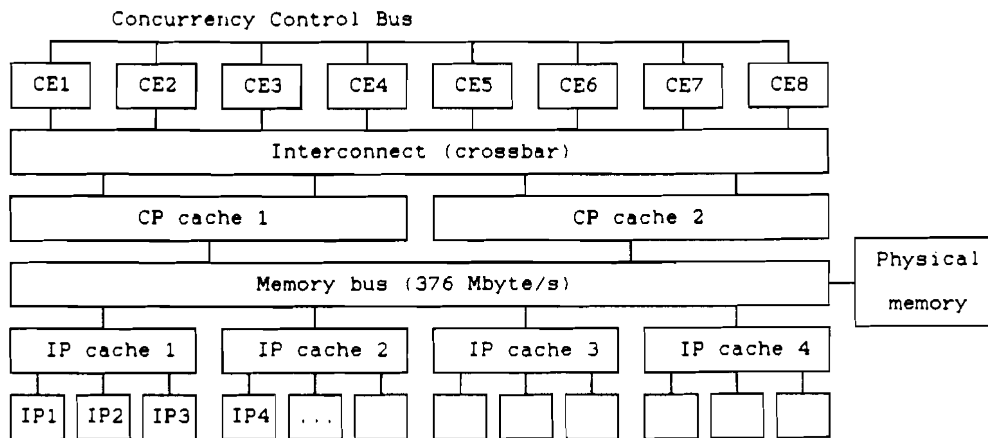

Fig.55. FX/8 Multiprocessor of Computational Complex.

moet worden een apart CE ter beschikking hebben, dan heeft het toevoegen van meer CE's geen enkel effekt.

Er is echter ook een mogelijkheid voor het samengroeperen van meerdere CE's tot één enkele resource (een "computational complex"). In dat geval werken de CE's gezamenlijk dezelfde instructiestroom af, maar op verschillende data items (SIMD). Met deze vierde vorm van parallellisme (door Alliant "concurrent processing" genoemd) wordt het wél mogelijk de executietijd van een job te verkorten. In het (zeldzame) ideale geval duurt de executie van een job op een complex van 8 CE's, één achtste van de tijd die nodig zou zijn bij executie van de job op één enkel CE.

In een applicatie met geneste loops wordt geprobeerd de binnenste loop te "vectoriseren". Dit houdt in dat nagegaan wordt of er data afhankelijkheden voorkomen tussen verschillende iteraties van de loop. Is dat niet het geval, dan wordt de binnenste loop verwerkt door een vector unit. De op één na binnenste loop wordt dan verwerkt door concurrent processing (mits er voldoende CE's beschikbaar zijn).

In het blokschema van fig.55 komen we ook nog IP's (interactive Processors) tegen. Dit zijn ook weer processoren die zelfstandig kunnen werken. Reken-intensieve taken worden uitgevoerd op

de CE's, terwijl de IP's zorgen voor de verwerking van interaktieve jobs, I/O-taken en operating system aktiviteiten.

We zien hier dat er sprake is van een hiërarchie van architecturen. Op het bovenste niveau is de machine een MIMD multiprocessor, maar door CE's parallel te schakelen ontstaat een SIMD architectuur. De individuele CE's op het lagere niveau vallen in de MISD categorie.

11. EEN OVERZICHT VAN ARCHITECTUREN

11.1. Inleiding

In dit hoofdstuk wordt een samenvatting gegeven van de onderzochte architecturen. Hierbij zal het accent liggen op de voordelen en nadelen van de diverse architecturen, de bottlenecks, de toepassingen en de manier waarop de besturing plaatsvindt. Verder zal aangegeven worden waar er meerdere niveaus van architectuur of besturing mogelijk zijn. Het uiteindelijke doel is te komen tot een schema waaruit afgeleid kan worden welke architectuur het meest geschikt is voor een bepaalde toepassing.

11.2. Array processoren

Het kenmerkende van een array processor is dat er meerdere identieke datapaden zijn (bijvoorbeeld ALU en registers) die door een gemeenschappelijke control unit bestuurd worden (fig.56). Dit heeft tot gevolg dat de datapaden synchroon moeten werken, en dat er maar één stroom van instructies is.

De toepassingen waarvoor array processoren geschikt zijn, maken dan ook gebruik van regelmatige strukturen, zoals vectoren en

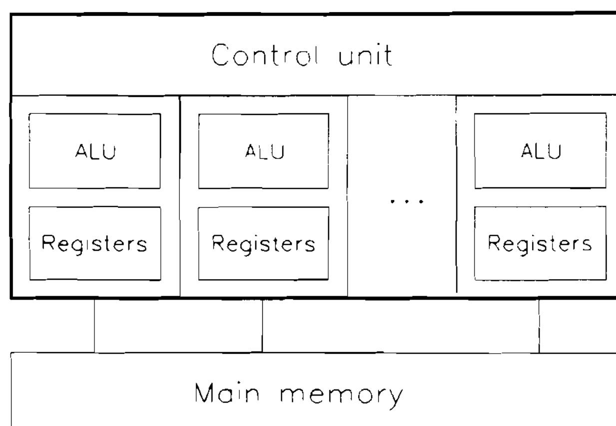

Fig.56. Array processor.

matrices. Elk element van een vector of matrix wordt dan bewerkt door één processing element uit de array.

11.3. Multiprocessoren

Een multiprocessor systeem bestaat uit meerdere processoren die onderling verbonden zijn door een interconnectie systeem (fig.7). De individuele processoren bestaan ieder uit een besturing en een datapad (fig.8). Dit maakt dat de processoren in staat zijn ieder een op zich staand gedeelte van een toepassing te verwerken, en dat deze stukken gelijktijdig verwerkt kunnen worden.

In de meeste multiprocessor systemen vormt het interconnectie systeem de bottleneck. Het is dan ook noodzakelijk de communicatie tussen de processoren zo veel mogelijk te beperken. Dit is mogelijk indien de applicatie opgesplitst kan worden in grote stukken die onafhankelijk van elkaar werken. Het voordeel van een multiprocessor systeem is de flexibiliteit: de opsplitsing kan voor iedere toepassing anders zijn.

De besturing van een multiprocessor systeem kan bijvoorbeeld plaatsvinden door één van de processoren de taken van het operating system te laten vervullen. Deze ene processor (de master) geeft dan commando's aan de andere processoren (de slaves).

11.4. Dataflow computers

Een conventioneel (control flow) programma bestaat uit instructies die uitgevoerd kunnen worden als de voorgaande instructies klaar zijn. Er kan slechts één instructie uitgevoerd worden op een bepaald tijdstip. Control flow programma's zijn dus in principe sequentieel.

Een dataflow programma bestaat uit instructies die uitgevoerd kunnen worden zodra alle benodigde operanden beschikbaar zijn.

De executie van een instructie levert een resultaat op dat operand kan zijn van meer dan één andere instructies. Op een bepaald tijdstip kunnen dus meerdere instructies geëxecuteerd worden. De processen die parallel uitgevoerd kunnen worden zijn dus klein, en men spreekt dan ook van parallelle processen met een "fijne korrel". Dit in tegenstelling tot de "grove korrel" die ontstaat indien complete procedures parallel kunnen verlopen. Door er nu voor te zorgen dat er meerdere processing elements beschikbaar zijn, wordt het mogelijk deze vorm van parallellisme te benutten.

Dataflow systemen zijn geschikt voor algemene toepassingen die veel inherent parallellisme bezitten, zonder dat het nodig is dat er een regelmatige struktuur in het parallellisme aanwezig is.

In principe is het mogelijk de performance van een dataflow computer te vergroten zolang de toepassing voldoende parallellisme bezit. Nadeel is echter dat door het toevoegen van PE's ook de communicatie overhead toeneemt. De bottleneck in een dataflow computer is vaak het communicatie netwerk, zodat het nodig is dat de communicatie tussen PE's beperkt wordt.

De besturing in een dataflow computer verloopt ook nogal onconventioneel (fig.42): een host computer bestuurt de PE's door pakketjes informatie (zowel instructies als data) het communicatie netwerk in te sturen.

11.5. Systolische arrays en wavefront arrays

Belangrijkste eigenschappen van systolische/wavefront arrays zijn de modulaire en uitbreidbare struktuur en het feit dat er alleen verbindingen tussen naburige processing elements voorkomen ("nearest neighbor connections"). De modulaire en uitbreidbare struktuur zorgen er voor dat de performance en de kosten van een systolisch/wavefront array evenredig zijn met het aantal processing elements.

Het voordeel van de tweede eigenschap wordt duidelijk als men bedenkt dat een steeds groter gedeelte van de oppervlakte van een IC nodig is voor verbindingen. Naast de hoeveelheid oppervlakte die nodig is, is ook van belang de vertragingen die de verbindingen opleveren. Naarmate de technologie vordert worden de devices (componenten) op een IC steeds kleiner, terwijl de grootte van de IC's steeds verder toeneemt ("scaling"). Dit heeft tot gevolg dat de devices op zich steeds sneller worden. De verbindingen tussen de devices worden door de scaling echter niet sneller, maar eerder langzamer [MEI87 p.61, TRE84 p.4]. Op grond hiervan is het niet onwaarschijnlijk dat in de toekomst de lengte van de verbindingen op een chip de bepalende factor wordt in de snelheid van de schakeling.

Een ander kenmerk van systolische/wavefront arrays is de hiërarchie van besturingen (fig.51): de host geeft commando's aan de command interpreter, en de command interpreter bestuurt de PE's. Bovendien kan in een (intelligent) PE het datapad weer bestuurd worden door een controller. De manier waarop de command interpreter de PE's bestuurt is vrij bijzonder: om de voordelen van de lokale verbindingen te behouden, wordt besturingsinformatie net als de data op systolische manier de array ingestuurd.

Systolische/wavefront architecturen zijn geschikt voor toepassingen waarin steeds weer dezelfde operaties uitgevoerd moeten worden (i.v.m. de identieke PE's), en de operanden meerdere keren gebruikt worden (i.v.m. het "systolisch stromen" door de array). Voorbeelden zijn dan ook te vinden in de digitale signaalbewerking (FIR[13]-filters en FFT[14]) en beeldbewerking (patroonherkenning).

Nadeel van deze systemen is dat ze maar voor één toepassing te gebruiken zijn. Verbeteringen zijn weliswaar mogelijk door de PE's programmeerbaar te maken, maar dit gaat weer ten koste van de eenvoud.

---

[13]Finite Impulse Response

[14]Fast Fourier Transform

Bij systolische arrays vindt de uitwisseling van data tussen PE's plaats op van te voren vastgestelde tijdstippen, onafhankelijk van de werkelijke tijd die een PE nodig heeft voor een operatie. Bij wavefront arrays gebeurt de uitwisseling van informatie met behulp van handshaking op het moment dat een PE data te versturen heeft, terwijl de data tussen twee PE's gebufferd wordt. Dit heeft tot gevolg dat de PE's efficiënter gebruikt worden, zodat wavefront arrays sneller zijn dan systolische arrays. Bovendien is dankzij de handshaking een wavefront makkelijker uit te breiden. Het synchroniseren van een systolisch array kan namelijk grote moeilijkheden opleveren.

11.6. Tradeoffs

Bij het ontwerpen van een computer architectuur zal men vele afwegingen moeten maken. Een grotere performance (snelheid) is in het algemeen alleen te realiseren ten koste van een grotere complexiteit (en dus hogere kosten). Hieronder wordt een beperkte opsomming gegeven van de diverse tradeoffs:

Interne bussen:
Voor de inwendige struktuur van de processor kan men in het algemeen kiezen uit één, twee of drie bussen (fig.4,5,6). Hoe meer bussen des te minder klokcycli zijn nodig voor de executie van een operatie op twee registers. Meer bussen betekend echter dat de registers multiported moeten zijn, en de bussen nemen een (relatief grote) oppervlakte van het IC in beslag.

Externe bussen:
Voor de externe struktuur kan men kiezen uit het wel of niet toepassen van de (gemodificeerde) Harvard architectuur (fig.3). Hierbij is het van belang hoeveel aansluitpennen er beschikbaar zijn, en of de bus een bottleneck vormt in het systeem. Bij multiprocessor systemen is er een scala van keuzes, variërend van één gemeenschappelijke bus (fig.9), meerdere bussen (fig.10) tot een aparte bus voor elke processor.

Besturing:
Voor de besturing in een processor is er onder andere de keuze tussen hardwired en microprogrammeerbaar. Een hardwired besturing is sneller, maar ook minder flexibel. Bovendien zal voor grote systemen de besturing zo complex worden dat alleen microprogrammering in aanmerking komt. Hier is dus sprake van een afweging van performance tegen ontwerpkosten.

Registers:
Het aantal interne registers heeft invloed op het aantal geheugenreferenties. Meer registers gebruiken betekent een hogere snelheid, maar kost oppervlakte op de chip. Bovendien moet men nagaan of bij een process swap het redden van registerinhouden niet het genoemde voordeel teniet doet.

Instructie executie pipelining:
Door de verschillende fasen in de instructie executie te laten overlappen kan een behoorlijke snelheidswinst behaald worden. Er zijn echter veel factoren die een negatieve invloed hebben op de werkelijk behaalde snelheidswinst. Om deze invloeden te beperken kan er veel extra hardware en/of software (optimizing compiler) nodig zijn.

Multiple PE's/CPU's:
Bij sommige architecturen is een vergroting mogelijk van de performance door het toevoegen van processing elements. Afgezien van de kosten, is het mogelijk dat door te veel PE's te gebruiken de communicatie een bottleneck gaat vormen.

Implementatie van operaties:
In het algemeen zijn er vele mogelijkheden voor de implementatie van arithmetische operaties (bijvoorbeeld vermenigvuldigen). Enkele voorbeelden in oplopende volgorde van snelheid / hardware zijn: implementatie in software, in het microprogramma, in de ALU, in een aparte unit (die gelijktijdig met de ALU kan werken), en in een vector unit.

Toepasbaarheid:
In sommige gevallen is het mogelijk een systeem meer of minder algemeen toepasbaar te maken (bijvoorbeeld systolische arrays). In het algemeen is "general purpose" trager en/of complexer dan "special purpose" (of "dedicated").

Fig.57 geeft globaal weer hoe de kosten als funktie van de performance er uit zouden kunnen zien. In de meeste gevallen nemen de kosten exponentieel toe, zodat er een punt is waarbij de prijs/prestatie verhouding optimaal is (het raakpunt). Verder is in de figuur met "basic machine" de "natuurlijke" implementatie aangegeven, d.w.z. die machine waarbij er geen speciale moeite gedaan is om de performance te vergroten of de kosten te verkleinen. Vaak blijkt de natuurlijke implementatie de optimale prijs/prestatie verhouding te hebben (zoals weergegeven in de figuur). [ANC86 pp.20,25]

11.7. Van specificatie naar architectuur

Uit het voorgaande overzicht van de onderzochte architecturen is een stroomschema samen te stellen dat een antwoord geeft op de vraag: Welke architectuur is het meest geschikt voor een bepaalde toepassing of klasse van toepassingen? Zie fig.58. Men moet wel bedenken dat dit ruwe indicaties zijn. Voor zover het überhaupt mogelijk zou zijn precies te bepalen welke architec-

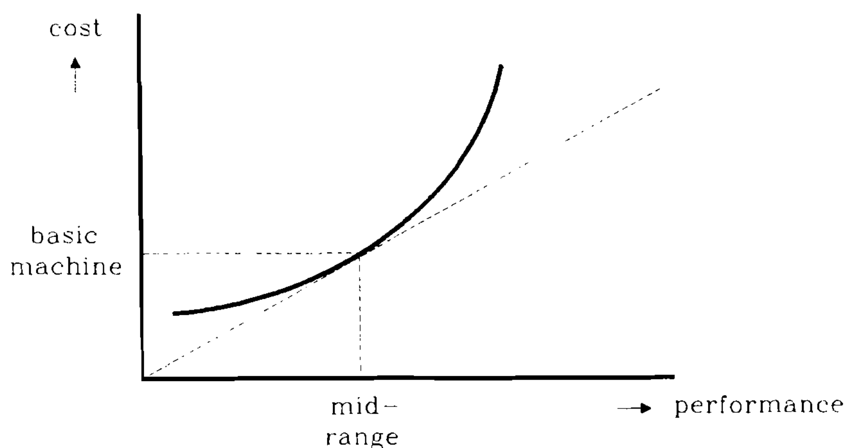

Fig.57. Tradeoff van kosten tegen performance.

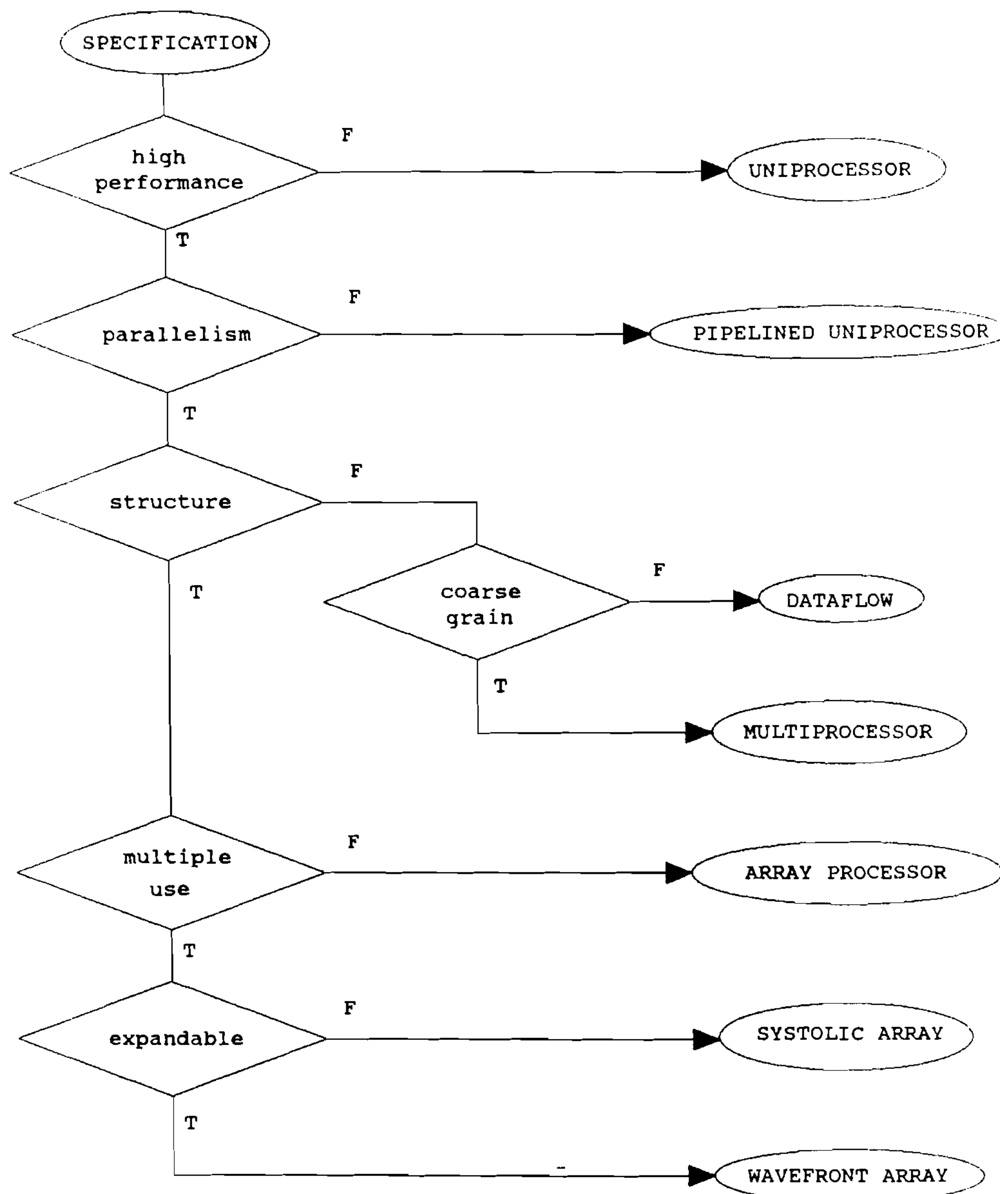

Fig.58. Van specificatie naar architectuur.

tuur de beste is, zou daarvoor in ieder geval de toepassing precies vast moeten liggen. Verder zijn er bij de meeste onderzochte architecturen diverse variaties mogelijk, denk bijvoorbeeld aan de hiërarchie van architecturen die mogelijk is.

De diverse afwegingen zijn:

1. Is een grote performance vereist? Zo nee, dan kan waarschijnlijk volstaan worden met de uniprocessor.

2. Bevat de toepassing (of de klasse van toepassingen) veel inherent parallellisme? Zo nee, dan is ook hier weer de uniprocessor de juiste keus, maar door bijvoorbeeld pipelining van de instructie executie toe te passen kan wellicht voldaan worden aan de eis van grote performance. Verder hebben we bij de tradeoffs gezien dat er nog vele technieken zijn die er mogelijk voor kunnen zorgen dat de gewenste snelheid gehaald wordt.
3. Heeft het parallellisme een regelmatige struktuur? Zo nee, ga verder met 4. Zo ja, ga verder met 5.
4. Is er sprake van parallellisme in grote korrels (d.w.z. de stukken programma die gelijktijdig uitgevoerd kunnen worden zijn relatief groot)? Zo nee, kies dataflow. Zo ja, multiprocessor.
5. Worden dezelfde operanden vaak gebruikt? Zo nee, array processor.
6. Moet de performance (onbeperkt) uitbreidbaar zijn? Zo nee, systolisch array. Zo ja, wavefront array.

12. CONCLUSIES

Dit verslag beschrijft een onderzoek naar architecturen die gebruikt kunnen worden voor parallel processing. Het gegeven overzicht is verre van volledig. Enerzijds zijn niet alle klassen van architecturen onderzocht, anderzijds zijn er meerdere variaties mogelijk binnen een bepaalde klasse. In dit verslag is dan ook getracht om enkele algemene principes weer te geven.

In de meeste besproken systemen wordt de bottleneck gevormd door de noodzakelijke communicatie tussen de diverse parallelle processen. Architecturen die dit nadeel trachten te beperken zijn grofweg in twee klassen te verdelen:

- systemen die de communicatie tussen processen beperken;
- systemen die de communicatie tussen processen snel maken.

Het blijkt dat in het algemeen een afweging gemaakt moet worden van de voor- en nadelen van een bepaalde architectuur. Een systeem met een grotere performance heeft hogere kosten tot gevolg, waarbij in de meeste gevallen de kosten exponentieel toenemen met de performance. Uitzonderingen zijn mogelijk door systemen te ontwerpen voor één bepaalde toepassing, of een beperkte klasse van toepassingen.

SLOTWOORD

Tot slot wil ik alle medewerkers en studenten van de vakgroep digitale systemen bedanken voor de prettige samenwerking gedurende mijn afstudeerperiode.

In het bijzonder gaat mijn dank uit naar mijn begeleider ing. P.H.A. van der Putten, van wie ik vele nuttige aanwijzingen heb ontvangen.

Mijn vriendin Annie wil ik bedanken voor het eindeloze geduld dat ze kon opbrengen. Zij gaf mij de motivatie die nodig was om mijn TU-studie af te ronden.

LITERATUURLIJST

[ALL85]
"Computer Architecture for Digital Signal Processing"
Jonathan Allen
Proceedings of the IEEE, vol.73, no.5, May 1985, pp.852-873

[ALL86]
"Alliant FX/Series Product Summary"
Alliant Computer Systems Corporation, Massachusetts, Oct. 1986

[AMD87]
"Am29000 Streamlined Instruction Processor"
Advanced Micro Devices, Sunnyvale California
Data sheet, Feb. 1987

[ANC86]
"The Architecture of Microprocessors"
François Anceau
Addison-Wesley Publishing Company, 1986
ISBN 0-201-14401-8 (DDT 86 ANC)

[BAS87]
"Parallel processing systems: a nomenclature based on their characteristics"
A. Basu
IEE Proceedings, Vol.134, Pt.E, No.3, May 1987, pp.143-147

[BRI86]
"A Perspective on RISC"
H.M. Brinkley Sprunt et al.
pp.5-47 in: "32-Bit Microprocessors"
H.J. Mitchell (ed.)
Collins Professional and Technical Books, London
ISBN 0-00-383067-5 (DCS 86 BIT)

[BUR82]
"Instruction set design issues relating to a static dataflow computer"
F.J. Burkowski
Proceedings of the 9th Annual Symposium on Computer Architecture, April 1982 (SIGARCH Newsletter, Vol.10, number 3, Computer Architecture News)

[DEN79]
"The varieties of data flow computers"
Jack B. Dennis
Proceedings of the First International Conference on Distributed Computing Systems, October 1979, pp.430-439

[DES84]
"Notes on computer architecture for high performance"
Alvin M. Despain
pp.59-138 in: "New computer architectures"
J. Tilberghien (ed.)
Academic Press, 1984
ISBN 0-12-690980-6 (DDT 82 NEW)

[ENS77]
"Multiprocessor Organization - A Survey"
Philip H. Enslow jr.
Computing Surveys, vol.9, no.1, March 1977, pp.103-129

[FOX87]
"Advanced Computer Architectures"
Geoffrey C. Fox & Paul C. Messina
Scientific American, October 1987, Vol.257, Number 4, pp.44-52

[GRA87]
"De transputer"
Jang Graat & Merik Voswinkel
Databus juni 1987, pp.13-21

[JOH87]
"System considerations in the design of the Am29000"
Mike Johnson
Advanced Micro Devices, January 1987

[KUN82]
"Why systolic architectures?"
H.T. Kung
Computer, January 1982, pp.37-46

[KUN87]
"Wavefront Array Processors - Concept to Implementation"
S.Y. Kung, S.C. Lo, S.N. Jean & J.N. Hwang
Computer, July 1987, pp.18-33

[LEO88]
"RISC microprocessors: many architectures thrive"
Milt Leonard
Electronic Design, July 1988, pp.49-58

[MEI87]
"Chips for Advanced Computing"
James D. Meindl
Scientific American, October 1987, Vol.257, Number 4, pp.54-62

[NEW86]
"The Inmos Transputer"
J.R. Newport
pp.93-129 in: "32-Bit Microprocessors"
H.J. Mitchell (ed.)
Collins Professional and Technical Books, London
ISBN 0-00-383067-5 (DCS 86 BIT)

[OBE88]
"Side by Side"
Klaus K. Obermeier
BYTE, November 1988, pp.275-283

[RAF88]
"Modern Computer Architecture"
Mohamed Rafiquzzaman & Rajan Chandra
West Publishing Company, St. Paul, 1988
ISBN 0-314-60714-0

[STA86]
"Reduced Instruction Set Computers"
W. Stallings
pp.58-81 in: "Tutorial: Reduced Instruction Set Computers"
William Stallings (ed.)
IEEE Computer Society Press, 1986
ISBN 0-8186-0713-0 (DDT 86 STA)

[TAN84]
"Structured computer organization"
Andrew S. Tanenbaum
Prentice-Hall, second edition, 1984 (DDM 84 TAN)

[TRE84]
"Decentralised computer architecture"
Philip C. Treleaven
pp.1-58 in: "New computer architectures"
J. Tilberghien (ed.)
Academic Press, 1984
ISBN 0-12-690980-6 (DDT 84 NEW)

[VEE88]
"Dataflow Machines"
Arthur H. Veen
in: "Mode(rne) architecturen"
Syllabus v.h. seminar op 21 juni 1988 te Den Haag
Afdeling/Sectie Informatietechniek van KIvI/NGI

[YAU77]
"Associative Processor Architecture - A Survey"
S.S. Yau & H.S. Fung
Computing Surveys, Vol.9, No.1, March 1977, pp.3-27