Seminar: Datenflußrechner (Everling / Seebaur)

DFSP: Data Flow Signal Processor

Peter Schütt

Am Heidkamp 88

5090 Leverkusen 3

02171/81059

10. Semester

SS 1990 *

*überarbeitet (mit LaTeX) am 16. Februar 2003

Inhaltsverzeichnis

1 DSP: Digital Signal Processing

Digital Signal Processing (DSP) bedeutet, daß digitale Eingangssignale kontinuierlich zu digitalen Ausgabesignalen verarbeitet werden. Eine Folge X_n von Digitalsignalen wird zu einer Folge Y_n verarbeitet:

$$X_n \rightarrow \boxed{\text{DSP-Algorithmus}} \rightarrow Y_n \tag{1}$$

wobei X_n die **Eingabesignale (Inputs)** und Y_n die **Ausgabesignale (Outputs)** sind.
Um diese Folgen X_n und Y_n im Rechner repräsentieren zu können, verwendet man den Datentyp **Stream**. Ein Stream ist eine Folge von Elementen, die immer denselben Typ (z.B. Einzelwerttyp, Feldtyp, usw.). Zur Veranschaulichung: Wenn man sich einen Eingabe-Stream als Bit-Stream vorstellt, dann liest das DSP-Programm immer dieselbe Anzahl von Bits von diesem Bit-Stream.
Die meisten der in der Praxis verwendeten DSP-Algorithmen haben die Eigenschaft der *Datenwertunabhängigkeit*. Anders ausgedrückt: Der Ablauf des Algorithmus' ist — unabhängig vom Eingabewert — immer derselbe. In dieser Arbeit werden nur solche datenwertunabhängigen DSP-Algorithmen behandelt.
Aus der Eigenschaft der Datenwertunabhängigkeit folgt, daß in einem DSP-Algorithmus keine eingabebedingten Anweisungen wie z.B. eingabebedingte Verzweigungen, Sprünge vorkommen. Daraus folgt, daß ein DSP-Algorithmus vor seiner Ausführung schon optimal auf eine parallele Architektur umgesetzt werden kann. (Es gibt jedoch dafür noch keinen Algorithmus, der DSP-Algorithmen automatisch optimal auf parallele Architekturen umsetzt.)
DSP-Algorithmen werden hauptsächlich in Echtzeitanwendungen verwendet, was nachher in der Aufzählung der Anwendungsgebiete deutlich wird. Deswegen sind viele Bausteine, die zu DSP-Programm-Ausführungen verwendet werden, auf Geschwindigkeit hin optimiert.
DSP verwendet man hauptsächlich für:

- Digitalfilter (FIR, IIR)
- Spektralanalyse (FFT)

FIR-Filter (finite impuls result) sind *nicht-rekursive, digitale Filter*; sie brauchen also keine vorangegangenen Outputwerte für die nächsten Outputwerte. **IIR-Filter** (infinite impuls result) sind *rekursive, digitale Filter*; sie benötigen vorangegangene Outputwerte zur Berechnung des neuen Outputwert. IIR-Berechnungen sind deshalb nicht oder nur eingeschränkt **zyklusparallel** ausführbar, während FIR-Programme uneingeschränkt zykluspar-

allel ausführbar sind. Mit zyklusparallel ist eine gleichzeitige Ausführung von verschiedenen ganzen Programmdurchläufen gemeint.
Die **FFT (Fast Fourier Transformation)** wird zur Frequenzanalysen von Signalen und Funktionen verwendet und hat daher ein sehr weites Anwendungsfeld (zu FIR, IIR und FFT, siehe [1]).
Die daraus sich ergebenden Anwendungsgebiete für DSP sind sehr weit gefächert:

- Medizin
- Seismographie
- Bildverarbeitung
- Mustererkennung
- Radar- und Sonarüberwachung
- Akustik (z.B. CD-Player)
- Spracherkennung
- Telekommunikation

In den meisten dieser Anwendungsgebiete wird sehr schnelles und Echtzeit-DSP benötigt.
Nun gibt es zwei Parallelisierungsmöglichkeiten in DSP-Programmen. Zum einen sind es die *innerhalb eines Zyklus' gleichzeitig ausführbaren Operationen*, wobei hier mit Zyklus ein Programmdurchlauf gemeint ist.
Zum andern sind es *parallel ausführbare Zyklen*, welche bei den meisten Anwendungen die Hauptparallelisierungsquelle ist.
Diese Parallelisierung wird durch **Pipelining** realisiert, bei dem die Ergebnisse von Programmodul zu Programmodul wie durch eine Rohrleitung weitergeleitet werden. Falls Ergebnisse schneller ankommen, als daß sie weiterverarbeitet werden können, werden sie in einer Warteschlange zwischengespeichert. Bei IIR-Filtern ist dies durch die Rekursivität nur sehr eingeschränkt möglich und daher werden diese in dieser Arbeit nicht betrachtet.

2 DFSP

2.1 Allgemeines

In dieser Arbeit soll der **Data Flow Signal Processor (DFSP)** vorgestellt werden, der einen DSP-Prozessor durch einen **Datenflußrechner (DFR)** realisiert. Ein Datenflußrechner kann durch einen **Datenflußgraphen (DFG)** dargestellt werden (Beispiele, siehe Abb. 3 und 4). Die Knoten in einem Datenflußgraphen werden **Aktoren** genannt; sie stellen die auszuführenden Operationen dar. Einem Aktor wird zur Ausführung der entsprechenden Operation ein **Prozessor-Element (PE)** zugeordnet, das die Operation ausführt.
Auf den ersten Blick scheinen Datenflußrechner für DSP sehr geeignet zu sein. Zum Beispiel kommen bei einem DSP-Algorithmus keine eingabebedingten Anweisungen vor, welche auch im Datenflußkonzept gewisse Probleme bereiten. Statische Schleifen kann man problemlos sequentialisieren (aufrollen). Die übrigen bedingten Anweisungen kann man gegen entsprechende nicht bedingte Anweisungen austauschen, da der Ablauf eines DSP-Programmes immer derselbe ist.
Nun ist aber einiges zu beachten. Zum ersten, wie schon erwähnt, sind DFR nur für nicht rekursive DSP-Algorithmen geeignet, weil man bei rekursiven DSP-Algorithmen nicht mehrere Zyklen parallel ausführen kann und die Parallelisierungsmöglichkeiten innerhalb eines Zyklus' für eine sinnvolle Auslastung normalerweise nicht ausreichen.
Ein Programm muß **reentrant** sein, damit die DSP-Programme immer wiederholt werden können. Reentrant bedeutet, daß das Programm mit neuen Daten wiederholbar ist, u.U. auch bevor die alten zu Ende verarbeitet sind. Dies ist notwendig, damit man mehrere Zyklen parallel abarbeiten kann. Dies kann man auf zwei Arten bewerkstelligen:
Man kann zum einen für jede neuen Eingabe-Operanden den gesamten Programmcode kopieren. Die zweite Möglichkeit ist das **Coloring**. Dazu werden alle **Token**, durch die im Datenflußgraphen die Operanden repräsentiert werden, unterschiedlich markiert („gefärbt, colored"), so daß die Token von verschiedenen Programmdurchläufen unterschieden werden können. Im DFSP wurde die zweite Möglichkeit gewählt.
Das Wissen, daß die optimale Parallelisierung eines DSP-Algorithmus schon vorher ermittelt werden kann, wird vom DFR nicht ausgenutzt, da bei einem DFR die Zuteilungen der Recheneinheiten erst in Laufzeit geschieht. Nun kann man dieses Wissen dazu verwenden, daß man Operationen, die in jedem Fall hintereinander ausgeführt werden müssen, zu einer komplexeren Operation zusammenfaßt und dann in der Laufzeit die ganze komplexe Operation *einer* Recheneinheit zuteilt. So kann der Verwaltungsaufwand im DFSP erheblich

reduziert werden. Dieses Zusammenfassen muß aber vor der Programmausführung der Benutzer durchführen.
Man könnte auch parallel ausführbare Operationen zu einer komplexen Operation zusammenfassen; dabei ginge aber dann die Möglichkeit einer Parallelisierung verloren, wenn die dazugehörige Recheneinheit, die die komplexe Operation bearbeitet, nicht parallel arbeiten kann. Aber vorstellbar sind Fälle, wo die Einsparung von Verwaltungsaufwand einen höheren Zeitgewinn als ein Nicht-Nutzen von Parallelisierungsmöglichkeiten bringt.
Diese komplexen Operationen, die den Aktoren zugeordnet werden, werden **High-Level-Signal-Processing-Operationen (HLSP-Operationen)** oder auch nur **High-Level-Operationen** genannt. Mögliche HLSP-Operationen wären:

- Fast-Fourier-Transformation (FFT)
- FIR-filtering eines Signalarrays
- Linearisierung eines Pixelarrays
- u.a.

Solche HLSP-Operationen müssen auch frei von Seiteneffekten und in sich selbst datenwertunabhängig sein; d.h. auch sie brauchen immer diesselbe Zeit, unabhängig vom Eingabewert. Bei den Operationen des DFSP's gibt es auch keine Einschränkung in der Anzahl oder Art der Operanden für jede Operation. So können solche HLSP-Operationen bspw. ein Array als Input und ein modifiziertes Array als Output haben. Diese Operationen können entweder durch spezielle Prozessor-Elemente (PE's) fest vorgegeben sein (wie z.B. FFT-Prozessoren) oder auch vom Benutzer individuell definiert werden und vor Start des Programm mit der übrigen Programminformation in die Lokalspeicher der Prozessor-Elemente (PE's) eingelesen werden. Dadurch wird das funktionale Programmieren sehr unterstützt.
Datenstrukturen, wo gemeinsam drauf zugegriffen werden darf, sind für DFSP's nicht geeignet. Da DSP-Algorithmen sehr schnell abgearbeitet und oft wiederholt werden, wären dann sehr viele Speicherzugriffe nötig, was Konflikte ergeben könnte. Außerdem wäre dann eine komplizierte Speicherverwaltung nötig. Man umgeht diese Probleme, indem man die Werte direkt in die Token des Datenflußgraphen (DFG) schreibt. Ein massives Kopieren von Daten durch Vervielfältigung von Token kann durch geeignete Zerlegung des Algorithmus' in geeignete HLSP-Operationen vermieden werden.
Da die meisten DSP-Anwendungen Echtzeit-Anwendungen sind, muß der DFSP die Daten schnell genug verarbeiten, damit nicht schneller Input-Daten eingelesen als Output-Daten

ausgegeben werden. (Dies würde zu einem Queue-Überlauf führen.) Dies ist zum einen durch einen effizienten Algorithmus zu erreichen. Zum andern kann man in den meisten Fällen die Ausführungszeit eines Programmes dadurch beschleunigen, indem man weitere PE's hinzufügt. So bietet die DFSP-Architektur die Möglichkeit an, *zwischen Materialkosten und Leistung abzuwägen, ohne daß das Programm modifiziert werden muß.*

2.2 Aufbau und Organisation des DFSP's

Der DFSP besteht zum einen aus einer **Execution Unit**, welche die Operationen ausführt. Andere Teile des DFSP's formen die **Control Section**, welche den Kontrollfluß steuert und das Pipelining organisiert. Dann gibt es noch die **DMA**'s, den **Speicher** und sonstige Komponenten.
Der Datenfluß ist physikalisch vom Kontrollfluß getrennt, indem eine Doppelbusarchitektur benutzt wird. Zusätzlich gibt es noch weitere Leitungen für Bestätigungen und ähnliches. In Abb. 1 ist der Aufbau des DFSP's schematisch dargestellt. Die dicken Leitungen sind Datenleitungen, die dünneren Leitungen Kontrolldatenleitungen, die Striche Bestätigungsleitungen.

Execution Unit

Die Execution Unit enthält einzelne Processor-Elemente (PE's). Diese können verschiedenen Typs sein und für eine häufig benötigte DSP-Funktion optimiert sein, so daß bspw. einige PE's Array-Manipulationen besonders effizient ausführen können. Man kann auch Spezialprozessoren verwenden, die nur eine Funktion berechnen können, z.B. FFT-Prozessoren. Die **Ein-/Ausgabefunktionen (I/O-Funktionen)** sind auch in der Execution Unit in speziellen PE's untergebracht.
Der **Host-Computer** ist die Verbindung zur Außenwelt. Er wird benötigt, um das Programm zu laden, und er liefert die Eingabesignale und nimmt die Ausgabesignale an.
Programme sind als getrennte High-Level-Operationen codiert und werden in die lokalen Speicher der PE's geladen. Im lokalen Speicher eines PE's stehen dann die Codierungen der zu dem PE passenden High-Level-Operationen und die Codierungen der Datenflußgraphen-Knoten, die diese Operationen verwenden. Somit steht das gesamte Programm in den lokalen Speichern der PE's.
Nach dieser Initialisierung läuft der DFSP ohne weitere Hilfe von außen.

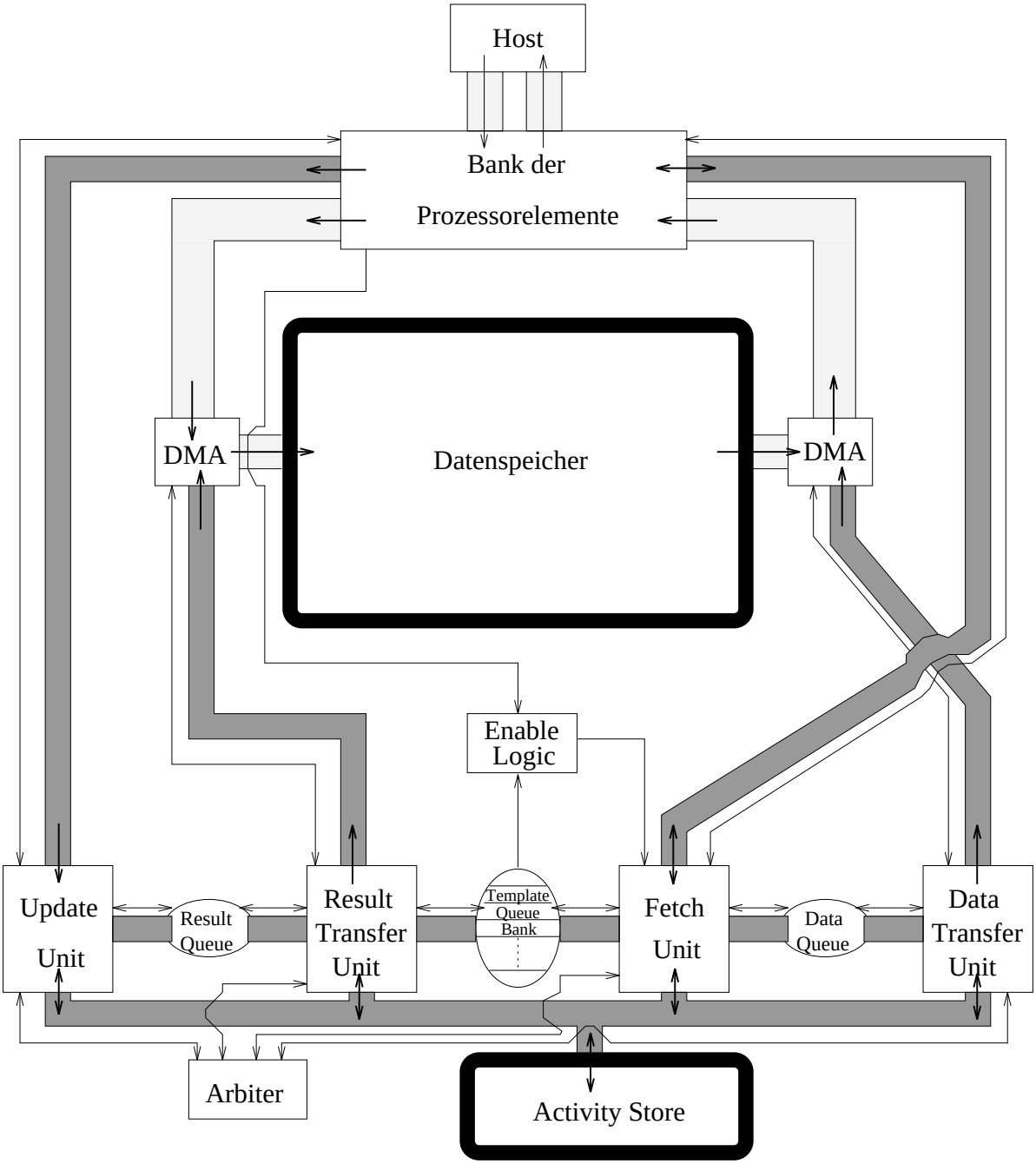


Abbildung 1: Schema des DFSP's

Am Ende der Beschreibung der Control Section wird noch einmal eine allgemeine Spezifikation für ein PE angegeben.

Control Section

In der Control Section gibt es vier Units:

- *Update Unit*

- *Result Transfer Unit*
- *Fetch Unit*
- *Data Transfer Unit*

Die *Update Unit* ist für das Zusammenfassen von Operanden der gleichen Zieloperation zuständig und teilt den Operanden Speicherplatz im Datenspeicher zu, die *Result Transfer Unit* kontrolliert den Datentransport von den PE's zum Datenspeicher und entdeckt ausführbare Operationen, die *Fetch Unit* weist diesen ausführbaren Operationen freie PE's zu, und die *Data Transfer Unit* transferiert die Operanden der ausführbaren Operation zu dem entsprechenden PE und gibt den für die Operanden zugeteilten Speicherplatz wieder frei. Standart-Semaphor-Mechanismen garantieren gegenseitigen Ausschluß bei kritischen Routinen, die auf den Datenspeicher oder auf den **Activity Store** zugreifen. Diese vier Units werden nachher in einer pascal-ähnlichen, informativen Spezifikation angegeben.
Die Kommunikation der Control Section mit der Execution Unit wird mit **Packets** durchgeführt, die ein festes Format haben. Die Token, welche zwischen den Aktoren des Datenflußgraphen (DFG) fließen, werden durch **Result Packets** repräsentiert. Ein Result Packet entsteht als Ergebnis einer Berechnung und ist folgendermaßen aufgebaut:
<Operation, Context, Totalsize, Result>

- Das Operation-Feld gibt den *Zielaktor* des DFG's für dieses Token an.
- Das Context-Feld bestimmt die Umgebung des Aufrufs; hierdurch ist also das Coloring realisiert.
- Das Totalsize-Feld gibt den gesamten physikalischen Speicherplatzbedarf aller Operanden des Zielaktors Operation an.
- Das Result-Feld gibt die Position innerhalb des Operandenblocks und den Typ des Operanden an, welchen das Result Packet repräsentiert. Aus dem Typ kann die physikalische Größe des Operanden ermittelt werden. Result könnte man als Record implementieren.

Ein PE versendet so ein Result Packet als Ergebnis einer Operation an die *Update Unit*, wobei das Result Packet auch von einem Input-PE stammen kann.
Die *Update Unit* kann fogendermaßen spezifiziert werden:
Update Unit:

```
1. do
2.   halt until ein PE sendet ein Result Packet;
3.   k := PE-Kennung;
4.   empfange Result Packet;
5.   i := hash (Operation, Context);
6.   suche das passende Activity Template im i-ten Bucket;
7.   if die Suche nicht fündig geworden then
8.       new_Activity_store(Template);
9.       new_Datenspeicher(Template.Location ,Totalsize);
10.      kopiere_Inhalte(Result Packet, Template);
11.      Template.Trigger:=Template.Totalsize;
12.      ordne Template in Template-liste ein;
13.  fi;
14.  schicke ein transfer_command(^Template, k, Result) in die Result Queue;
15. od
```

(^Template stellt die Template-Adresse dar; Buckets werden in der Beschreibung des Activity Store erläutert.)

Die *Update Unit* wartet also auf ein Result Packet von einem PE und sucht dann das passende Template im Activity Store dazu. Wenn die *Update Unit* kein Template findet, das zu dem eingesandten Result Packet paßt, dann repräsentiert dieses Result Packet den ersten fertigen Operand von der Zieloperation. Es wird nun ein neues Template beschrieben. An welchen Stellen neuer Speicher reserviert wird, wird nachher erläutert.

Das Activity Store enthält Repräsentationen aller Zieloperationen, bei denen noch nicht alle, aber schon mindestens ein Operand vorliegt. Das Activity Store enthält daher eine Repräsentation des momentan aktiven Teils des Datenflußgraphen, mit Ausnahme der Knoten, die gerade berechnet werden. Die Repräsentationen der Zieloperationen stehen in starren Strukturen, Activity Templates oder kurz Templates genannt:

<Linkage, Operation, Context, Totalsize, Trigger, Location>

- Das Linkage-Feld enthält 2 Pointer, die die Struktur im Activity Store aufbauen und später erläutert werden.
- Operation, Context und Totalsize ist wie bei dem Result Packet.
- Das Trigger-Feld wird bei Initialisierung des Templates auf Totalsize gesetzt und bei jeder Hinzufügung eines neuen Operanden zu den übrigen Operanden im Speicherblock des Templates wird Trigger um den Speicherplatzbedarf dieses neuen Operanden vermindert. Wenn Trigger = 0, dann kann die Operation, die das Template repräsentiert ausgeführt werden, weil dann alle Operanden vorhanden sind.
- Das Location-Feld enthält die Startadresse des dem Template zugeordneten Speicherblocks. In diesem Speicherblock stehen alle die Operanden, von denen schon ein Result Packet gesendet wurde.

Das Activity Store unterstützt folgende Funktionen:

1. Zusammenfassung der zu einer Operation gehörigen Operanden
2. Entdecken von ausführbaren Operationen
3. Speicherverwaltung für die Operanden

Der Aufbau des Activity Store im Einzelnen wird später beschrieben.
Der **Arbiter** sorgt dafür, daß beim Zugriff auf das Activity Store keine Konflikte entstehen. Er dient als Semaphore, um kritische Operationssequenzen auf gemeinsame Daten im Datenspeicher unteilbar zu machen.
Das **transfer_command**(ˆTemplate, k, Result) von der *Update Unit* zur *Result Transfer Unit* enthält die Adresse des Templates, die Kennung k des PE's, von dem der zum Result Packet zugehörige Operand empfangen werden soll, und Result, damit der Operand an die richtige Stelle in dem zum Template gehörigen Speicherblock geschrieben werden kann. Durch dieses transfer_command(ˆTemplate, k, Result) kann die *Result Transfer Unit* den Operanden in den zum Template gehörenden Speicherblock schreiben. Die PE-Kennung k empfängt die *Update Unit* über die Bestätigungsleitung.
Die *Result Transfer Unit* kann folgendermaßen spezifiziert werden:
Result Transfer Unit:

1. **do**

```
2.  halt until transfer_command(^Template, k, Result) ist in Result Queue;
3.  hole transfer_command(^Template, k, Result) aus Result Queue;
4.  start Datentransfer des Result Data Block's von PE_k
      nach Template.Location + Result.Position durch Startsignal;
5.  Trigger := Trigger - Speicherplatzbedarf(Result.Typ);
6.  halt until der Datentransfer von PE_k ist beendet;
7.  if Trigger=0 then
8.      wähle die richtige Template Queue,
          gemäß dem Operation-Feld des Templates;
9.      schreibe(^Template, Template_Queue);
10. fi
11. od
```

(Der Result Data Block ist der Operand, den das Result Packet repräsentiert. k ist die Kennung des aktuellen PE's.)
(Location + Result.Position) gibt die Adresse an, an die der gerade empfangene Operand geschrieben werden soll. Wenn Trigger=0 wird, dann sind jetzt alle Operanden empfangen worden, und damit kann die Operation, welche durch das Template repräsentiert wird, ausgeführt werden. Die Template Queues sind die Verbindungen von der *Result Transfer Unit* zur *Fetch Unit*. Warum das hier mehrere sind, wird gleich klar.
Nun eine Spezifikation der *Fetch Unit*:
Fetch Unit:

```
1.  do
2.  halt until ein passendes Paar eines leerlaufenden PE's
      und einer nichtleeren Template Queue sind vorhanden;
3.  k:=PE-Kennung;
4.  hole ^Template von der Template Queue;
```

5. `Operation Packet := (Template.Operation, Template.Context);`
6. `schicke Operation Packet zu leerlaufendem` PE_k`;`
7. `schreibe data_transfer_command(^Template,` k`) in die Data Queue;`
8. **od**

Der Sinn der mehreren Template Queue's liegt darin, daß die Template Queue gemäß Operation ausgewählt wird. Die High-Level-Operationen stehen ja nicht bei jedem PE im lokalen Speicher, sondern nur bei den dazu passenden bzw. dazu optimierten PE's. So ist jeder Template Queue einer Gruppe von PE's zugeordnet. Diese Zuordnung regelt die Enable Logic. Über eine Leitung werden kontinuierlich die PE-Kennungen der leerlaufenden PE's von der PE-Bank zur Enable Logic gesendet, die diese Kennungen mit der Template Queue Bank vergleicht und —wenn eine PE-Kennung zu einer nichtleeren Template Queue paßt— diese PE-Kennung einfach an die *Fetch Unit* schickt und die *Fetch Unit* sich dann die passende Template Queue ausliest.
Das Operation Packet ist folgendermaßen aufgebaut: <Operation, Context>.
Operation und Context haben diesselbe Bedeutung wie im Result Packet. Context muß nur übergeben werden, damit das PE Context korrekt in das Result Packet nach Ausführung der Operation schreiben kann. Context muß immer die korrekte Umgebung angeben.
Da alle statischen Schleifen sequentialisiert wurden, braucht Context nur ein integer-Wert zu sein, dessen Wertebereich größer als die maximale Anzahl von Programmzyklen ist, die gleichzeitig bearbeitet werden können. Dann wird Context bei jedem neuen Zyklus um einen erhöht, modulo dem Wertebereich von Context.
Das data_transfer_command(^Template, k) enthält die Template-Adresse und die Kennung des ausführenden PE's.
Eine Spezifikation der *Data Transfer Unit*:
Data Transfer Unit:

1. **do**
2. **halt until** `data_transfer_command(^Template,` k`) in der Data Queue ist;`
3. `hole das data_transfer_command(^Template,` k`) aus der Data Queue;`
4. `beginne den Transfer des Operand Data Block's durch Startsignal;`
5. **halt until** `Transfer beendet;`

```
6.  entferne das Template aus der Template-Liste und markiere es als frei;
7. od
```

(k ist die Kennung des aktuellen PE's.)
Es wird hier deutlich, daß PE_k die Operanden nur als einen kompletten Datenblock übergeben kriegt. Es ist also alleiniges Problem der als Programm im lokalen Speicher des PE's abgelegten Operation diesen Datenblock in die einzelnen Operanden wieder zu zerlegen. Aber da der zu jeder Operation entsprechende Operandenblock immer diesselbe Struktur und Größe hat, ist das keine Schwierigkeit.
Mit der Freigabe des Templates und dessen Entfernen aus der Template-Liste wird auch gleichzeitig der dazugehörige Datenblock freigegeben. Dies wird später bei der genauen Beschreibung der Speicherverwaltung deutlich.
Abschließend wird noch eine allgemeine Spezifikation eines PE's angegeben:
PE:

```
1. do
2.  informiere Fetch Unit über eigenes Leerlaufen;
3.  halt until Bestätigung;
4.  akzeptiere Operation Packet von Fetch Unit;
5.  halt until Data Transfer Unit sendet Startsignal für Operandentransfer;
6.  empfange Operanden für die Operation;
7.  führe Operation durch und produziere Result Packets;
8.  while noch nicht alle Result Packets abgeschickt
9.  do
10.     informiere Update Unit, daß ein Result Packet abgeschickt werden kann;
11.     halt until Bestätigung;
12.     sende nächstes Result Packet;
13.     halt until Result Transfer Unit sendet Startsignal;
```

```
14.       sende Result Data Block;
15.   od
16. od
```

Speicherverwaltung

Das Activity Store und der Datenspeicher werden beide dynamisch verwaltet. Das Activity Store ist ein Pool von einer festen Anzahl Templates, wobei nicht benutzte Templates als frei markiert sind. Die benutzten Templates sind zum einen durch *Hashing* geordnet, mit Hilfe einer Hash-Funktion, die aus Operation und Context einen Hash-Wert berechnet. Wenn nun mehrere Templates denselben Hash-Wert haben, dann werden diese hintereinander doppelt verkettet. So eine doppelt-verkettete Liste von Templates mit gleichem Hash-Wert nennt man auch „Activity Template Buckets“ oder kurz „Buckets“. In Abb. 2 ist diese Struktur durch die Template-Einträge B_Link und durch die durchgezogenen Pfeile dargestellt. Die locker gepunkteten, waagerechten Linien stellen eine optische Bucket-Begrenzung dar.
Da das Activity Store einfach nur ein Pool von Templates ist und damit zweckmäßig als Array organisiert ist, wird bei einer Reservierung eines neuen Templates einfach von dem letzten zugegriffenen Template ausgegangen und das nächste, das frei ist, genommen und, wenn der entsprechende Eintrag der Hash Index Table auf NIL zeigt, dieser Eintrag mit dem neuen Template doppelt verkettet. Ansonsten wird das neue Template einfach mit dem letzten Bucket-Element doppelt verkettet. Dann werden von der *Update Unit* die Inhalte des Result Packets in die entsprechenden Einträge des Templates kopiert.
Nun sind alle nicht-freien Templates zusätzlich zur Hash-Ordnung auch noch im Kreis doppelt-verkettet miteinander verbunden. In der Abb. 2 ist diese Ordnung durch die gepunkteten Pfeile, die von C_Link ausgehen, dargestellt. Auf den Nachfolger wird in Abb. 2 immer von rechts gezeigt. Die Reihenfolge der Operandenblöcke im Datenspeicher entspricht durch eine geeignete Organisation immer der Reihenfolge der Templates in dieser Kreisordnung. Dazu geht die Routine, die den Datenspeicher für das Template reserviert, vom letzten zugegriffenen Template aus, sucht von dessen Operandenblock aus im Datenspeicher den ersten genügend großen ($\geq$Totalsize) freien Speicherblock (First Fit) und geht dazu die Kreisliste der Templates Template für Template durch. Die Anfangsadresse dieses Speicherblockes wird in Location geschrieben. Zwischen den Templates, zwischen deren Operandenblöcke der neu reservierte Speicherblock liegt, wird das neue Template in die Kreisliste eingehängt. Zu beachten ist, daß diese Kreisliste unabhängig von der Hash-Organisation ist. Wenn nun ein

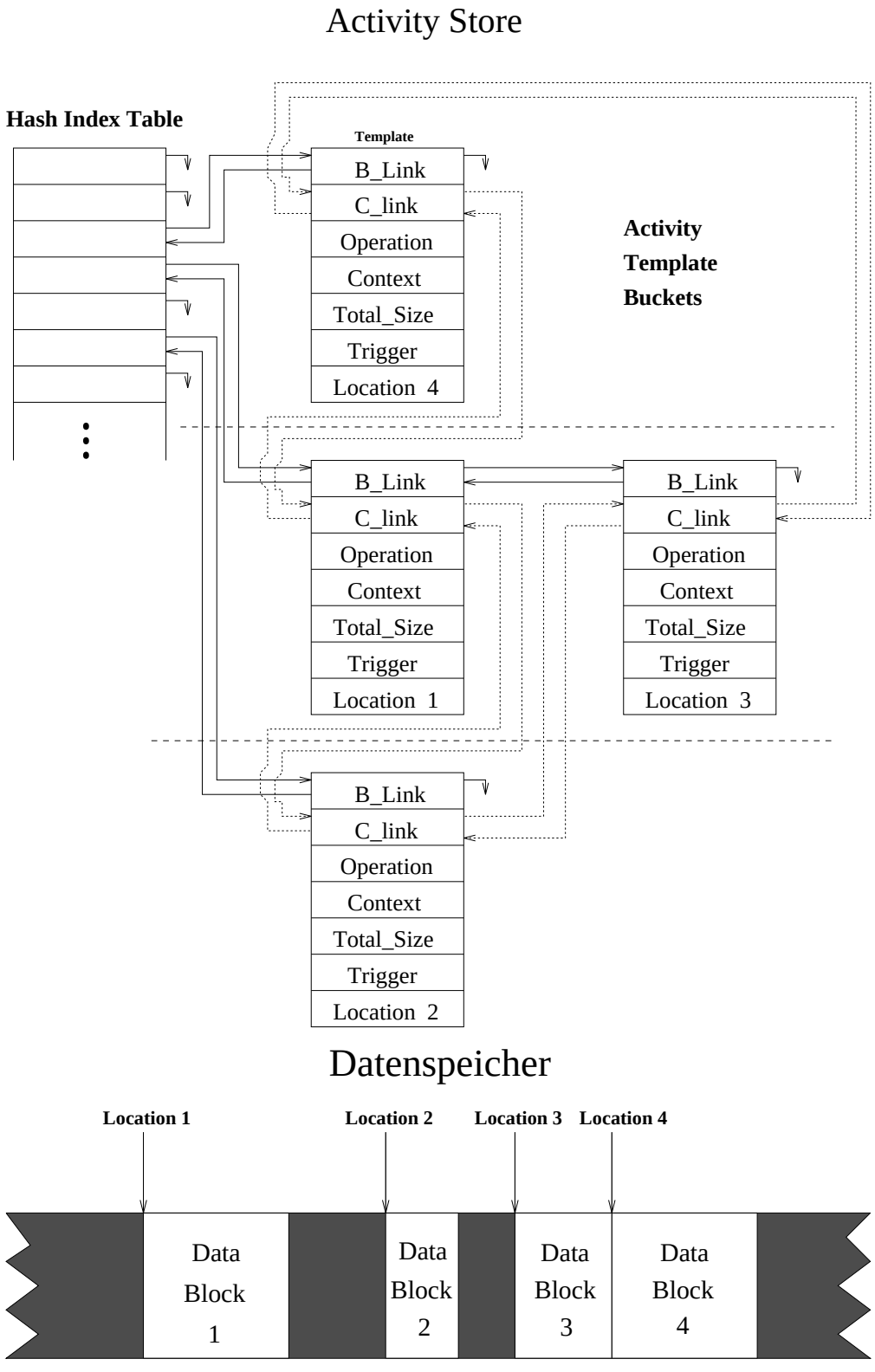


Abbildung 2: Speicherverwaltung

Template entfernt werden soll, weil die dazugehörigen Operanden verarbeitet wurden, dann wird einfach das Template aus dem Bucket und aus der Kreisliste entfernt und als frei mar-

kiert. Dadurch ist auch automatisch der diesem Template zugeordnete Speicherbereich wieder frei, da ja die Routine, die Datenspeicher reserviert, sich allein an der Template-Kreisliste orientiert. Die Kreisorganisation wird durch eine doppelt verkettete Liste bewerkstelligt, weil dann ein Entfernen und Einfügen von Templates erheblich vereinfacht wird. Diese Speicherverwaltung erspart komplizierte Verwaltungsmechanismen, wie z.B. *garbage collection*.
Bei diesem Verfahren ist es vorstellbar, daß eine starke Fragmentierung des Speichers auftreten könnte, doch bei Simulationen von verschiedenen DSP-Algorithmen trat eine Fragmentierung des Speichers nicht auf. Daher braucht der DFSP keinen Mechanismus, um den Speicher durch Zusammenschieben entzufragmentieren.

3 Simulator

3.1 Beschreibung

Es wurde ein deterministischer Simulator für diskrete Signale entwickelt, um die Leistung des DFSP's in verschiedenen Anwendungen zu testen. Der DFSP wird durch verschiedene sequentielle Prozesse simuliert, welche parallel arbeiten und sich gegenseitig beeinflussen können, z.B. durch einen Zugriff auf einen gemeinsamen Speicher. Die Detailliertheit des Modells ist ein Kompromiß zwischen Meßgenauigkeit und Simulationsgeschwindigkeit.

Control Section

Es wurde eine konventionelle 16-Bit-Mikroprozessor-Architektur mit einem orthogonalen Instruktionen-Set spezifiziert und die angegebenen Spezifikationen der vier Units aus der Control Section werden auf diesen Instruktionen-Set angepaßt. Diese Maschinenprogramme werden auch dazu verwendet, um markante Punkte in diesen Prozessen genau lokalisieren zu können (z.B. wann Datenübertragung beginnt und endet) und um die Ausführungszeit der aktiven Phasen genau messen zu können. Die interne Organisation des Activity Store wurde völlig implementiert, damit man die Korrektheit der Konzeption überprüfen und den jeweils aktuellen Speicherbedarf ermitteln kann. Daneben kann man dadurch auch noch die Effekte des gegenseitigen Ausschlusses in kritischen Programmregionen beobachten (Warten auf Semaphore). Ein präzises Modell für den Hauptspeicherzugriff inklusive der Leistungsminderung durch gegenseitiges Blockieren beim Speicherzugriff konnte nicht realisiert werden.

Execution Unit

Es wurde ein generelles Modell eines PE's entwickelt, welches in verschiedendsten DFSP-Konfigurationen und -Applikationen verwendet werden kann. Das Modell definiert die Bus-Interfaces zu der Control Section und die Interfaces der I/O-PE's zum Host-Computer. Alle System-Transaktionen zwischen Host und DFSP-Rechner sind wohldefiniert und relativ selten. Die innere Struktur eines PE's wird nicht repräsentiert, weil die Leistung eines speziellen PE's nicht interessiert. Der Simulator simuliert eine spezielle Applikation, indem er ein Konfigurations-File und das Applikationsprogramm lädt. Das Konfigurations-File spezifiziert die Anzahl und die Typen der PE's und das Auftreten der externen I/O-Ereignisse. Das Applikationsprogramm ist nur eine Sammlung von einfachen Prozeduren, die nur die Form des Result Packet und die Verzögerung angibt, die das entsprechende PE produziert (Weil das Ergebnis für die Simulation uninteressant ist und DSP-Algorithmen datenwertunabhängig sind, kann eine DSP-Operation als eine konstante Zeitverzögerung aufgefaßt werden).

Messungen

1. Als erstes wird die gesamte Durchsatzzeit gemessen, wie lange ein Input-Signal braucht, bis es zu einem Output-Signal verarbeitet worden ist (und die ist vom Inhalt des Inputs unabhängig). Die Input-Rate ist vom Konfigurations-File festgelegt. Ob nun die Verarbeitung schnell genug ist, damit die Input-Rate nicht höher als die Output-Rate ist und kein Speicherüberlauf entsteht, wird durch Lauf des Simulators festgestellt.

2. Für jede Komponente wird die verbrauchte Zeit der verschiedenen Phasen (aktiv oder passiv) während einer Ausführung aufgezeichnet.

3. Auslastungsraten der Speicher und Vorhandensein von Messages in Queues werden gemessen, indem der Zustand der entsprechenden Modellkomponente aufgezeichnet wird;

3.2 Testbeschreibungen

Drei-Sensor-Problem

1. Drei Eingabeströme mit gleicher Frequenz, welche durch einen FIR-Filter vorgefiltert werden;

2. Diese drei Ergebnisse werden kreuzweise miteinander verknüpft.

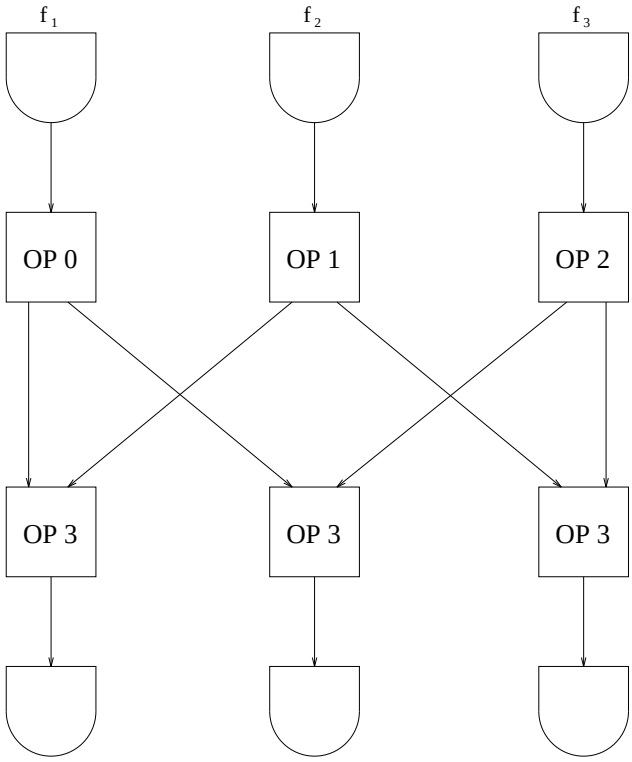


Abbildung 3: Datenflußgraph des Drei-Sensor-Problems

(siehe Abb. 3) Die Mathematik, die dahinter steckt, ist durch die Datenwertunabhängigkeit für die Simulation nicht von Bedeutung und wird in [2, 3] beschrieben.
Die DFSP-Architektur für diese Simulation ist folgende:

1. Drei Eingabe-PE's mit einer Verzögerung von 50 μs für jedes Result Packet;
2. Drei Ausgabe-PE's mit einer Verzögerung von 30 μs für jedes Result Packet;
3. 64 PE's zur normalen Berechnung; für die Vorfilterung und die Verknüpfung werden zusammen 87.1 ms benötigt.

Im folgenden wird für das Drei-Sensor-Problem die Abkürzung **TSP** (Three-Sensor-Problem) verwendet.

Echtzeitbildverarbeitung

Eine weitere mögliche Anwendung für DFSP-Rechner ist Echtzeitbildverarbeitung. Dieses Beispiel besteht aus drei Charakteristika:

1. Die Bildverarbeitungsoperation ist eine **Point Spread Function (PSF)** [2, 3].

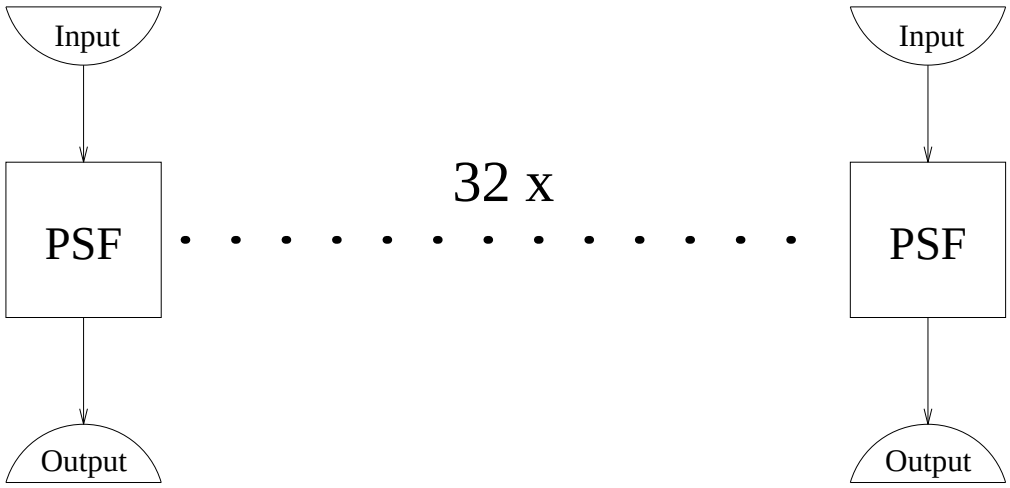


Abbildung 4: Datenflußgraph der Echzeitbildverarbeitung

2. Das Eingabebild besteht aus 256*256 Punkten, wobei jeder Punkt durch 16 Bits dargestellt wird.

3. Die Input-Rate ist vergleichbar mit der Fernseh-Frequenz, 25 Bilder pro Sekunde.

Als DFSP-Konfiguration wird die vorige wieder verwendet: 1 Eingabe-PE, 1 Ausgabe-PE und 64 Berechnungs-PE's. Die Eingabe-PE's verzögern um 50 μs und die Ausgabe-PE's um 30 μs.
Jedes Eingabebild wird in 32 gleichgroße Segmente partitioniert; der Datenflußgraph besteht aus 32 getrennten Subgraphen, die jeweils einen Eingabeknoten, einen Funktionknoten und einen Ausgabeknoten haben (Abb. 4). Die Ausführung der Funktion benötigt 9038 μs.
Die Segmente werden alle 1.25 ms vom Eingabe-PE eingelesen. Der Wert 1.25 ms ergibt sich aus:

$$25 \text{ Bilder}/s = (25 * 32 \text{ Segmente})/s = 800 \text{ Segmente}/s$$

$$\Rightarrow \qquad \text{Alle } 1/800s = 0.00125s = 1.25ms \text{ wird ein Segment eingelesen}$$

Dieser Wert ist wichtig, um die Abb. 6 zu verstehen. In dieser Grafik sind alle benutzten Prozessoren, die DMA's und alle Control Units aufgeführt; die aktiven Phasen sind schattiert, die passiven nicht. Man kann hier in dieser Grafik deutlich sehen, daß das Eingabe-PE alle 1.25 ms ein Segment einliest. Die Mehrfach-Striche zeigen die verschiedenen Phasen an (aktiv oder passiv). Bei der *Result Transfer Unit* ist der Ablauf in die Phasen

1. Warten auf ein Kommando

2. Transfer vorbereiten und beginnen

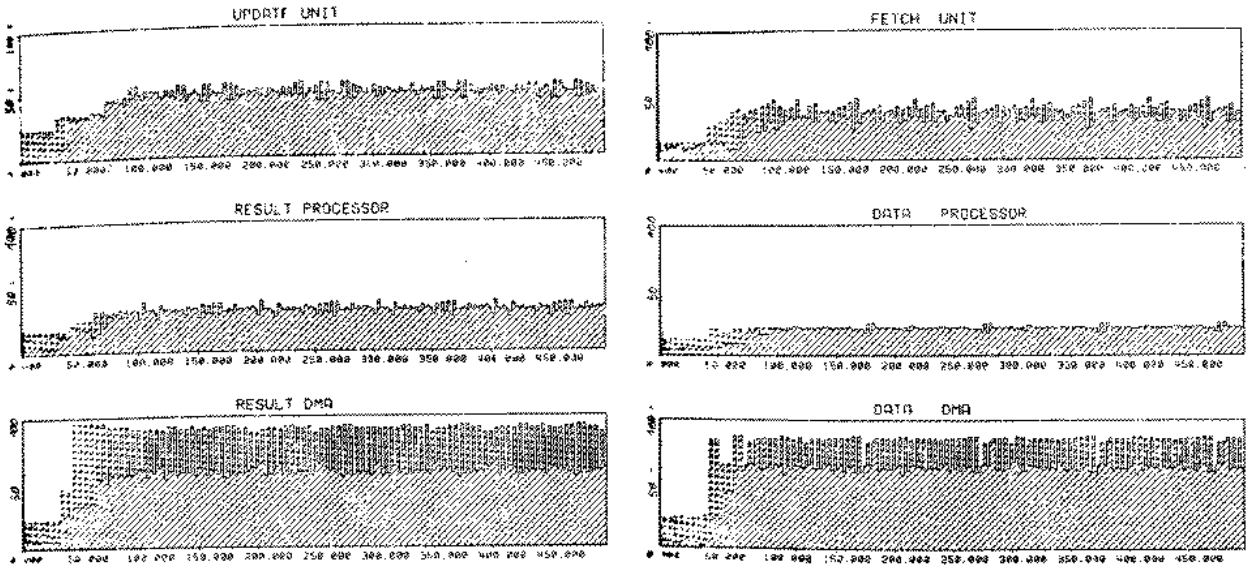
3. Warten auf Beendigung des Transfers

unterteilt. Bei den PE's sind mit den Doppelstrich-Passivphasen die Abschnitte gemeint, wo das Ergebnis abgeschickt und wo die Operanden eingelesen werden.
Im folgenden wird für diesen Test die Abkürzung **PSF** verwendet.

3.3 Meßergebnisse

Gesamte Durchsatzzeit und Verwaltungsaufwand

DFSP System Load Diagram



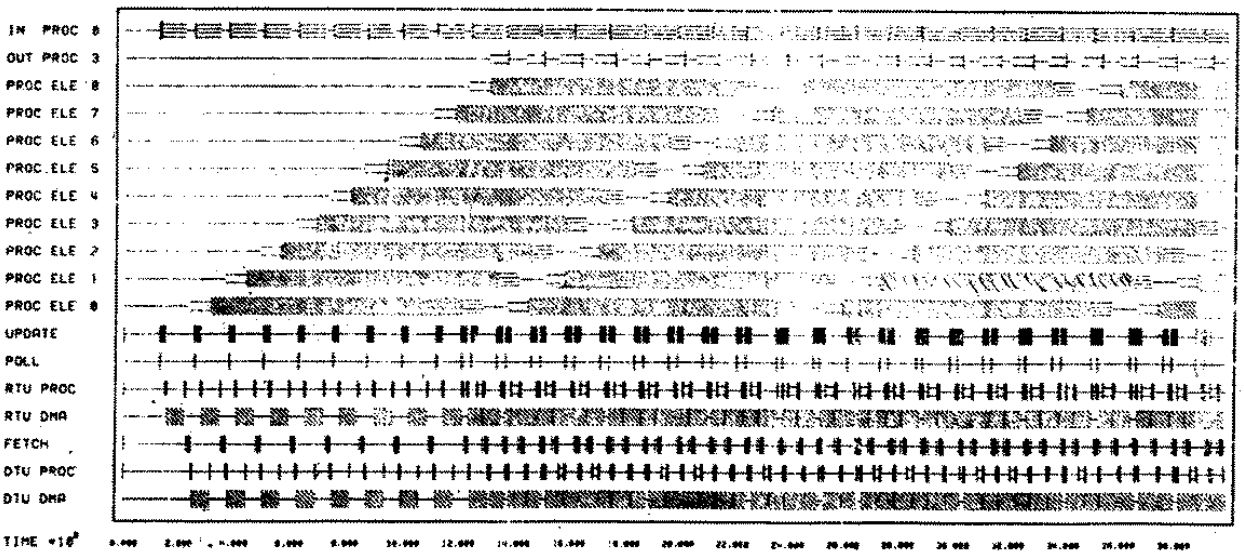
Zeiteinheit: μs Erstellungsdatum 5 1 1982

Abbildung 5: Auslastung der Komponenten in der *Drei-Sensor-Simulation (TSP)*

für TSP: Die berechnete Durchsatzzeit ist 87.2 ms, die tatsächlich gemessene beträgt 95.5 ms. Diese zusätzliche Verzögerung von 8.3 ms ist für den Verwaltungsaufwand angemessen (Datenflußkontrolle, Datentransfer, usw.).

für PSF: Hier ist der Verwaltungsaufwand prozentual etwas höher; die berechnete Durchsatzzeit beträgt 9.1 ms, die gemessene 13.9 ms. Das ist ein Verwaltungsaufwand von 4.8 ms. Dies sind über 50% mehr. Dieser Wert ist aber nicht so kritisch, wenn die Eingabebilder entsprechend zur Eingaberate immer noch schnell genug abgearbeitet werden können.

DFSP System Load Diagram



Zeiteinheit: μs

Abbildung 6: Auslastung der Komponenten in der *Echtzeitbildverarbeitungsimulation (PSF)*

Zeitverbrauch jeder Komponente in der aktiven Phase

Solche Zeiten sind nur durch die Abb. 6 für PSF angedeutet. Da für TSP und PSF diesselbe DFSP-Konfiguration verwendet wurde, gelten diese Zeiten natürlich auch für TSP (außer natürlich von den PE's und den DMA's). Diese absoluten Zeiten sind aber weniger interessant. Was interessant ist, ist die Frage, ob die einzelnen Komponenten schnell genug arbeiten, um einen Queue-Overflow zu vermeiden. Und das ist eine Frage der Auslastung.

Auslastungen der Speicher, Queues und Komponenten

Control Units Bei TSP war die *Update Unit* zu ca 50 % ausgelastet, die *Result Transfer Unit* und die *Fetch Unit* zu ca 40 % und die *Data Transfer Unit* zu ca 25 % ausgelastet. Bei PSF sind die Werte für die *Update Unit* wieder am höchsten, aber für alle liegt die Auslastung unter 50 % (siehe Abb. 6). In beiden Beispielen liegt folglich der wesentliche Engpaß nicht bei den Control Units, da die Datenflußgraphen (DFG) relativ einfach sind. Bei komplizierteren DFG's kann aber die *Update Unit* zu einem Engpaß in der Control Section werden, denn die *Update Unit* muß als einzige Unit Suchoperationen im Activity Store und im Hauptspeicher durchführen bzw. durchführen lassen. Die *Fetch Unit* ist weniger kritisch, da sie nur weniger komplizierte und umfangreiche Aktionen durchführen muß und oft mehrere Result Packets für eine Operation benötigt werden. Die anderen beiden Units, die

	Drei-Sensor-Problem			Bildverarbeitung		
(in %)	Min.	Max.	Durchschn.	Min.	Max.	Durchschn.
Activity Store	3.0	18.8	16.0	2.9	5.8	4.3
Datenspeicher	24.6	36.9	29.7	3.1	7.0	5.4
Template Queue	0.0	5.0	0.2	0.0	5.0	0.5
Result Queue	0.0	5.0	1.2	0.0	5.0	0.2
Data Queue	0.0	10.0	2.0	0.0	10.0	0.4

Tabelle 1: Auslastungen einzelner Komponenten bei den Tests

den Datentransport steuern, sind noch weniger kompliziert und verursachen keine Engpässe.

DMA Es traten Probleme an den beiden DMA's auf. Bei TSP lagen die Auslastungen bei der Result DMA teilweise bei über 90 %, bei der Data DMA bei 80 % (siehe Abb. 5) . Bei PSF wurden beide DMA's bis zu 90 % ausgelastet. Bei beiden Tests ist es nun nicht möglich, die Input-Rate wesentlich zu erhöhen (bei PSF könnte es auch eine Vergrößerung der Bilder bedeuten), da die Datentransferkapazität nahezu erschöpft ist. Der einzige Ausweg wäre hier eine Verbreiterung des Datenbusses.

Prozessorelemente Für TSP war die Zahl der PE's zu hoch; es wurden 56 PE's teilweise (60-87%) genutzt, die übrigen 8 wurden gar nicht genutzt. Eine Erhöhung der PE-Anzahl oder eine Verminderung von bis zu 8 PE's hätte daher keine Veränderung des Ergebnisses gebracht. Eine weitere Reduzierung der PE-Anzahl hätte die Ausführungszeit natürlich verschlechtert. Nur 9 PE's wurden bei der Bildverarbeitung PSF benutzt, während die anderen vollkommen ungenutzt blieben (Abb. 6). Aber die, welche benutzt wurden, wurden zu 80 % ausgelastet, obwohl sie nur einen einzigen Operationstyp zu leisten hatten. Durch die Parallelität des DFSP's konnten sie effizient ausgenutzt werden.

Activity Store Verglichen mit dem Datenspeicher wurde relativ wenig Speicher für die Activity Templates reserviert (die Hash Index Table bekam ein Viertel des Activity Stores). Der Speicherbedarf des Activity Stores für TSP, war sehr gering, weil maximal zweistellige High-Level-Operationen verwendet wurden. Dies führt zu einer kurzen Lebensdauer der Templates, da zwei Operanden relativ schnell verfügbar sind und außerdem genügend freie PE's zur Verfügung standen. Bei PSF wurden sogar nur einstellige Operationen verwendet, so daß, da immer freie PE's zur Verfügung standen, der Input direkt „durchgereicht“ werden

konnte. Dies wird auch in der Tabelle 1 deutlich: Die maximale Auslastung lag hier gerade bei 18.8% für TSP, und bei PSF lag sie bei 5.8% .

Queues Die Queues waren die meiste Zeit während beider Simulationen leer (siehe Tabelle 1). Es traten keine Overflows auf, obwohl die Queues sehr knapp kalkuliert waren:
— 20 Words für die Result Queue und 10 Words für die Data Queue;
— 20 Words für die Template Queue.
Die Template Queue Bank bestand nur aus zwei Template Queues, da nur ein Berechnungs-PE-Typ und ein Ausgabe-PE verwendet wurde. Da das Ausgabe-PE im Vergleich zu den anderen Aktionen sehr schnell arbeitete (30 μs), benötigte dessen Template Queue nur die Größe eines Ausgabe-Elements. Die 20 Words gehören also zur Template Queue der Berechnungs-PE's. Für die Auslastung der Queues siehe Tabelle 1.

Datenspeicher Die first-fit-Strategie bei der Speicherzuweisung funtionierte gut. Der Speicher blieb unfragmentiert, was für eine kurze Ausführungszeit der Zuweisungsroutine sorgte. Bei Anwendungen, wo die Lebensdauer der Templates sehr unterschiedlich sind, kann Fragmentierung Probleme hervorrufen. Was einer Fragmentierung natürlich entgegenwirkt, ist die Tatsache, daß die Datenblöcke der High-Level-Operationen eher größer sind. Im Datenspeicher eines DFSP's sind also normalerweise eher weniger und dafür größere Datenblöcke vorhanden.

4 Mögliche Anwendungen

Der DFSP wurde für eine spezielle Klasse von DSP-Anwendungen entwickelt, denn bei vielen DSP-Anwendungen tritt ein Konflikt zwischen hoher Leistung und Flexibilität deutlich auf. Die höchste Leistung erhält man, indem man einen Algorithmus direkt in Hardware realisiert (z.B. FFT-Prozessoren). Dies ist natürlich unflexibel und deswegen will man eine programmierbare Architektur erstellen. Die Leistungsfähigkeit einer DFSP-Architektur kann durch geeignete Wahl der PE's individuell gestaltet werden, da diese Architektur keine Einschränkung in Anzahl und Typ der PE's vorgibt. Dadurch kann eine hohe Leistungsfähigkeit erreicht werden. Gleichzeitig ist solch eine DFSP-Architektur frei programmierbar und damit ist die geforderte Flexibilität erreicht.
Die DFSP-Architektur kann jedoch auch bei anderen Anwendungen verwendet werden, wenn folgende Bedingungen erfüllt sind:

1. Die Anwendung verläuft kontinuierlich und ein fester Satz von Programmen wird während der Anwendung verwendet. Wenn die Anwendung läuft, kann kein weiteres Programm nachgeladen werden.

2. Es wird im hohen Maße mit lokalen Datenstrukturen gearbeitet, welche einen Transfer von ganzen Strukturen in die PE's rechtfertigen müssen. Globale Daten können nicht verwendet werden.

3. Die Anwendung muß gut in High-Level-Operationen zerlegbar sein.

4. Rekursive Algorithmen sind in den meisten Fällen für DFSP's nicht geeignet, da sie normalerweise nur schlecht parallelisierbar sind.

Die DFSP-Architektur hat viele positive Eigenschaften für Software-Entwicklung. Der Programmierer muß sich nicht um die Synchronisation bei parallelen Prozessen kümmern, da dies durch die Datenfluß-Mechanismen abgedeckt wird. Er bekommt fast „kostenlos“ den hohen Grad an Parallelität, den das Datenflußprinzip bietet. Dies ist zum Beispiel bei Anwendungen wichtig, die mehrere Eingaben einlesen und diese geeignet synchronisieren müssen. Die DFSP-Architektur unterstützt auch strukturiertes Programmieren, weil die Programme in High-Level-Operationen zerlegt werden müssen.

5 Hardwareimplementation des DFSP's

Durch die Modularität der DFSP-Architektur ist diese ein geeignetes, flexibles Hardware-System für verschiedene DSP-Anwendungen. Ein charakteristisches Merkmal des DFSP's ist die Möglichkeit der freien Konfiguration der PE-Bank. Jeder Prozessor kann als PE verwendet werden, vorrausgesetzt er kann an das Bussystem angeschlossen werden und hat genügend internen Speicher, um Daten zu empfangen und zu berechnen. Für die Implementation der Control Section gibt es zwei Möglichkeiten:

1. Die Benutzung von Standart-Mikroprozessoren

2. Implementation in VLSI

Im ersten Fall ist eine hohe Flexibilität gegeben, da die Funktionen der Control-Section-Komponenten durch Mikroprogramme realisiert werden. Das System kann so auf verschiedene Geräte leicht angepaßt werden. Die Implementation in VLSI sorgt für einen hohen Durchsatz auf Kosten dieser Flexibilität.

5.1 Standart-Mikroprozessoren

Für die vier Control-Units kann man herkömmliche 16-Bit-Mikroprozessoren nehmen. Die Funktionen dieser Units sind anwendungsunabhängig, so daß diese Funktionen in Mikroprogrammen realisiert werden können. Die Kommunikations dieser Units wird über gemeinsame Speicher (Queues, Activity Store) realisiert. Diese Speicher sind getrennte Multiport-Speichermodule (Multiport für den gemeinsamen Zugriff). Auf das Activity Store muß natürlich jede Unit zugreifen können. Der Datenspeicher ist so realisiert, daß gleichzeitig in verschiedene Adressen geschrieben und gelesen werden kann. Die Daten werden von DMA-Bausteinen von den lokalen Speichern der PE's zum Datenspeicher und zurück geschoben. Konfliktfreies gleichzeitiges Adressieren ist durch die Speicherverwaltung des DFSP's garantiert.

5.2 VLSI

Der Durchsatz der Control Section kann natürlich durch die Implementierung der Control Section in spezielle Hardware verbessert werden. In [4] wurde die DFSP-Control-Section auf einem einzelnen VLSI-Chip vorgeschlagen. Die funktionale Struktur des Chips ist wie in Abb. 1 angegeben und zwar ist alles unterhalb des Datenspeicher, also die gesamte Control Section inklusive Activity Store, auf dem Chip realisiert. Die Anordnung der einzelnen Elemente auf dem Chip ist in Abb. 7 abgebildet.
Die Kommunikation der PE's mit der Control Section verläuft über zwei physikalisch getrennte Bussysteme, die Verbindung mit der *Update Unit* und die die Verbindung zur *Fetch Unit* über die Enable Logic. Bei der Kommunikation mit der *Update Unit* hat immer das PE Vorrang, das bisher die längste Zeit warten mußte.
Bei der Kommunikation mit der *Fetch Unit* gilt dieses Verfahren für jede Klasse von PE's. Als erstes werden von den leerlaufenden PE's einer Hardware-Klasse und den nicht leeren Template Queues der entsprechenden Template-Queue-Menge die Template Queue und das PE einander zugeordnet, die die längste Zeit warten mußten.
Die Chip-Prozessoren haben mikroprogrammierbare Controller und die Mikroprogrammierung wurde durch Mechanismen für Unterprogrammaufrufe und durch ein 2 Kword ROM erleichtert. Die Programme für die Chip-Prozessoren wurden alle komplett in Mikroprogrammen realisiert, so daß keine Mechanismen zum Lesen externer Instruktionen benötigt werden.
Die Queues zwischen den Control Units sind nicht synchronisiert. Die einzelnen Elemente einer Queue, die als Eingang einer Transfer Unit fungiert, sind einfach aufgebaut und haben genau die Größe des Kommandos, das über die Queue geschickt wird.

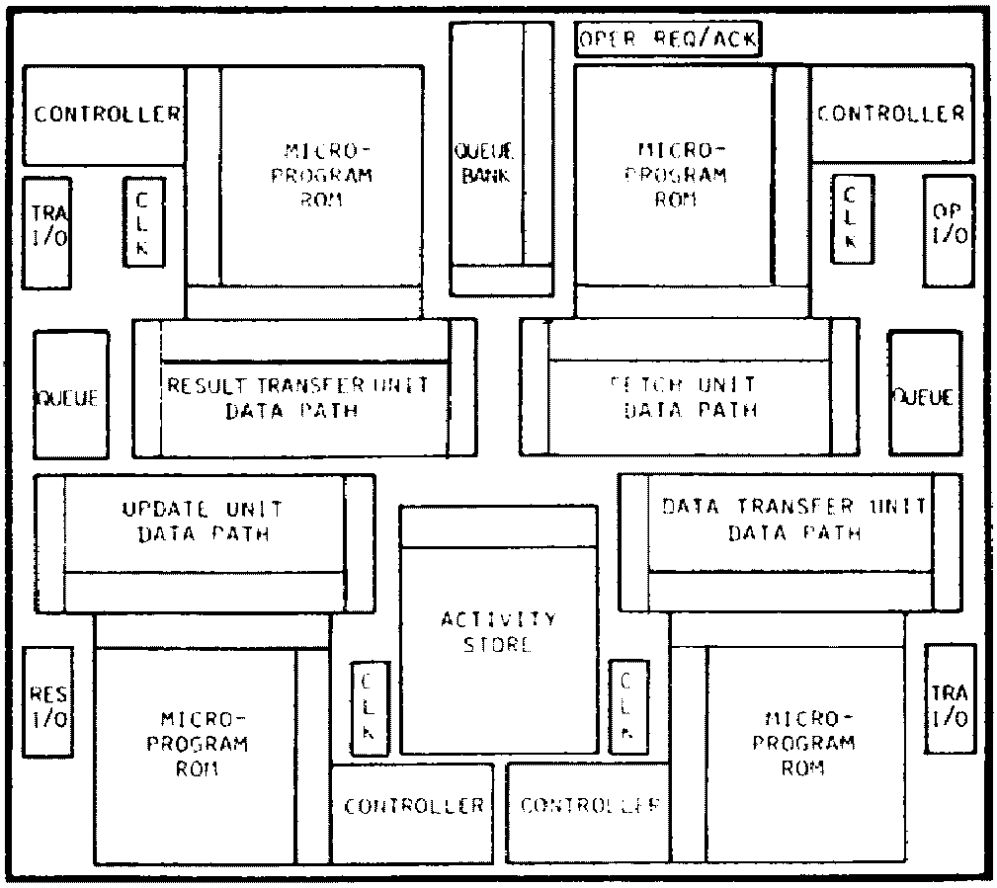


Abbildung 7: Implementation der DFSP-Control-Section auf einem VLSI-Chip

Die Template Queue Bank besteht aus acht Template Queues.
Das Activity Store Subsystem besteht aus 1 Kword RAM und aus einem Arbiter, der zum einen Konflikte regelt, die beim Zugriff auf das Activity Store entstehen können. Zum anderen dient der Arbiter als Hardware-Semaphor, um kritische Operationssequenzen beim Zugriff auf gemeinsame Datenstrukturen unteilbar zu machen. Die Größe des RAM's, 1 Kword, hat sich in ausgedehnten Simulationsversuchen als ausreichend erwiesen.
Weitere insbesondere technische Einzelheiten sind in [4] beschrieben.

Literatur

[1] Abraham Peled and Bede Lu. *Digital Signal Processing.* John Wiley & Sons.

[2] Iiro Hartimo, Klaus Kronlöf, Olli Simula, and Jorma Skyttä. Dfsp: A data flow signal processor. *IEEE Transactions on computers*, c-35(1), January 1986.

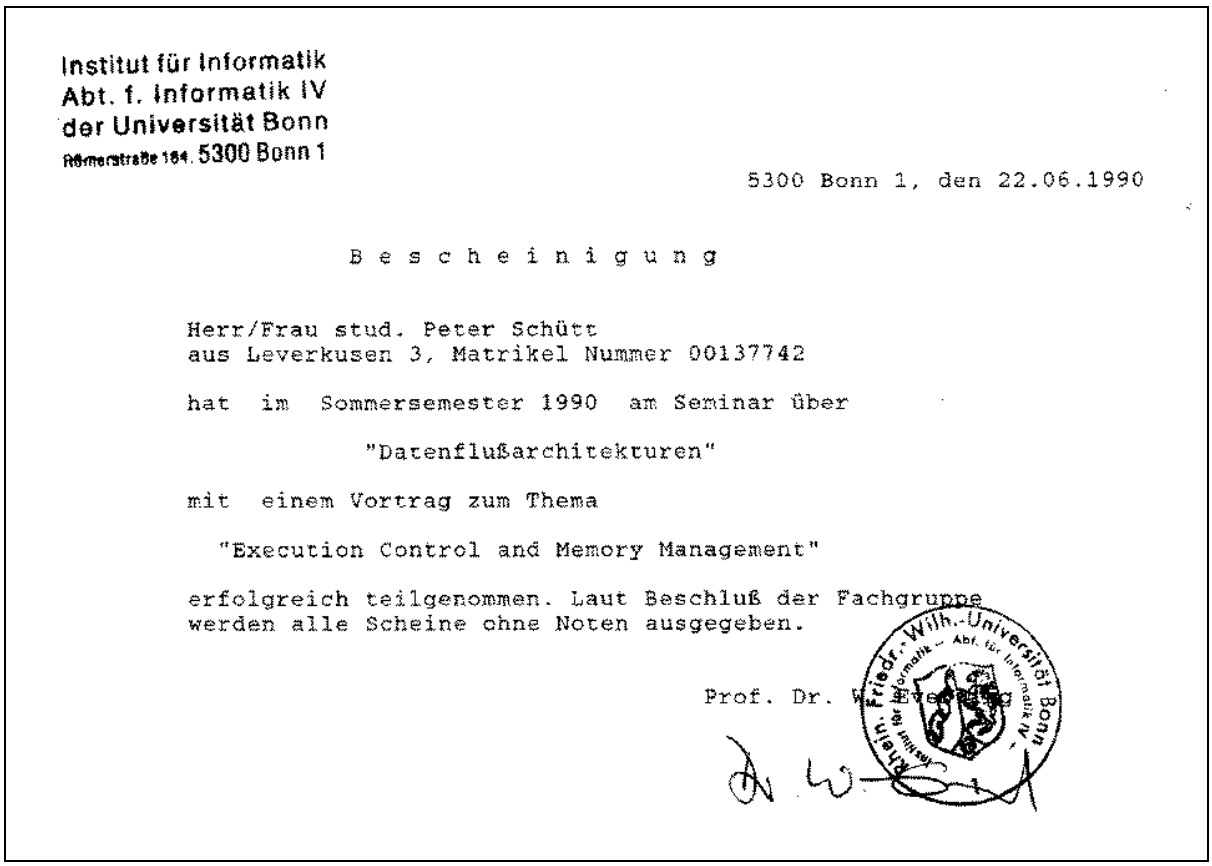
[3] Iiro Hartimo, Klaus Kronlöf, Olli Simula, and Jorma Skyttä. Performance of an experimental data flow architecture for signal processing. *IEEE Trans. Acoust., Speech, Signal Processing*, 2:695–698, May 1982.

[4] Iiro Hartimo, Klaus Kronlöf, and Olli Simula. On the vlsi implementation of a data flow architecture signal processor. *Proc. ICCC*, 2:594–597, October 1982.

[5] Klaus Kronlöf. Execution control and memory managment of a data flow signal processor. *IEEE 10th intern. Symposium on Computer Arch.*, 1983.

Abbildungsverzeichnis

Note



Institut für Informatik
Abt. f. Informatik IV
der Universität Bonn
Römerstraße 164, 5300 Bonn 1

5300 Bonn 1, den 22.06.1990

B e s c h e i n i g u n g

Herr/Frau stud. Peter Schütt
aus Leverkusen 3, Matrikel Nummer 00137742

hat im Sommersemester 1990 am Seminar über

"Datenflußarchitekturen"

mit einem Vortrag zum Thema

"Execution Control and Memory Management"

erfolgreich teilgenommen. Laut Beschluß der Fachgruppe
werden alle Scheine ohne Noten ausgegeben.

Prof. Dr. W. Brauer

Friedr.-Wilh.-Universität Bonn
Institut für Informatik – Abt. für Informatik IV

Abbildung 8: Seminarschein

Nach einem Beschluß der Fachgruppe werden Seminararbeiten nicht benotet. Wenn diese Arbeit aber benotet würde,
dann würde ich ihr die Note geben.

gez. Seebaur