High-Level Synthese & PASS-Pedal

Burkhard Reike

23. November 1994

Seminar
der
Projektgruppe Systementwurf
WS94/95 und SS 95

Inhaltsverzeichnis

Kapitel 1

High-Level Synthese

1.1 Einführung

Betrachtet man die Entwicklung von Mikrochips in den letzten Jahrzehnten, so stellt man fest, daß auf immer kleineren Flächen immer komplexere Schaltungen integriert werden. Anfang der 90'er erreichte die VLSI[1]-Technologie Integrationsdichten von über einer Million Transistoren pro Chip. Es ist klar, daß der Entwurf solcher Systeme durch Planung jedes einzelnen Transistors von Hand oder durch Definition aller Signale für logische Gatter nur *sehr* schwer zu bewältigen ist. In dem Maße, wie die Komplexität von Systemen steigt, wächst auch daß Verlangen nach Automatisierung des Entwurfsprozesses.

In den frühen Tagen der Entwicklung des computergestützten Entwurfs verwendete man den Rechner lediglich zur Verifikation des Entwurfs (Simulation von Schaltkreisen, Verifikation von Layouts [2]). Später benutzte man den Rechner dann auch zum Entwurf des Layouts, ausgehend von einer Netzliste und einer Bibliothek von vorgefertigten Zellen. In den letzten zehn Jahren entwickelte sich ein Forschungszweig, der sich nun auch mit dem Entwurf von Netzlisten, ausgehend von einer abstrakten Beschreibung in einer Hochsprache, beschäftigt. Dieser Teil des Entwurfsprozesses wird *High-Level Synthese* genannt. Zur genauen Begriffsdefinition werfen wir einen kurzen Blick auf die verschiedenen Ebenen der Beschreibung von Hardware.

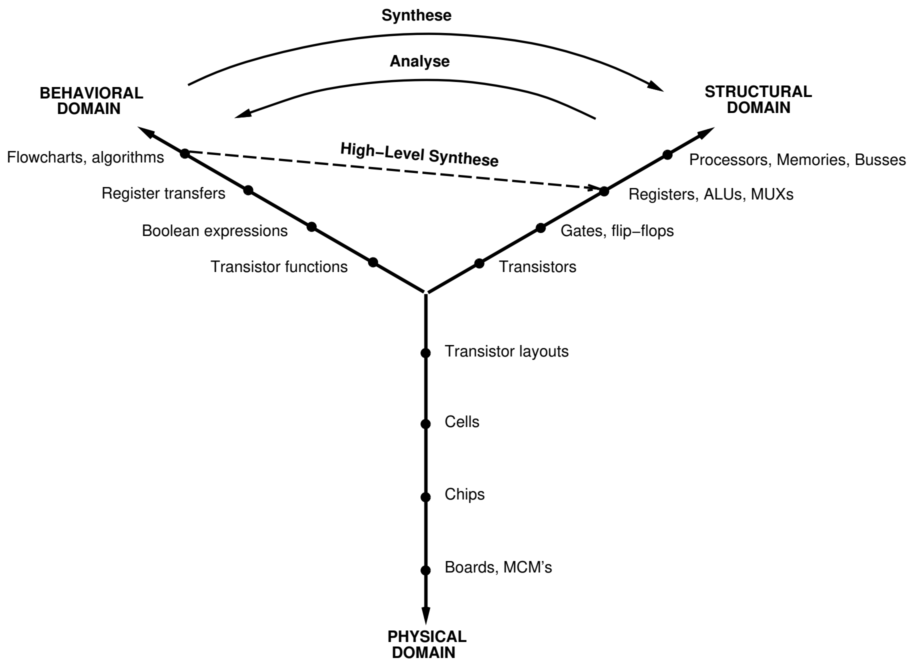

Abbildung 1.1: Y-Diagramm

Zur Klassifizierung der verschiedenen Arten von Hardwarebeschreibungen hat sich das sogenannte *Y-Diagram*[GaKu83] etabliert. Beschreibungen von Hardware werden dabei in drei Sichten aufgeteilt:

[1] VLSI = Very Large Scale Integration

[2] Das Layout ist das Endprodukt des Designs. Den einzelnen Transtoren und Verbindungen werden auf einer Fläche exakt positionierte Rechtecke und Linien zugeordnet.

- *Verhaltens-Sicht*
- *Struktur-Sicht*
- *Physikalisch/geometrische Sicht*

Alle drei Sichten sind in verschiedene Abstraktionsebenen aufgeteilt. Auf algorithmischer Ebene wird zum Beispiel Hardware in der Verhaltens-Sicht durch Algorithmen oder Flußdiagramme, auf Schaltkreisebene durch Differentialgleichungen dargestellt, auf Register-Transfer Ebene erfolgt die Darstellung durch Spezifikation von Registerkommunikationen. In der Struktur-Sicht ordnet man die Zusammenfassung von Bausteinen zu einem Prozessor der algorithmischen Ebene zu (weil ein Prozessor eine Einheit ist, die Algorithmen ausführt), die Darstellung als Netzliste der Register-Transfer Ebene, etc. Die physikalisch/geometrische Sicht bezieht auf die rein geometrische Darstellung des Schaltkreises, sprich dem Layout. Dieser Bereich interessiert uns aber im Moment nicht.

Man kann nun Beschreibungen aus einer der drei Sichten in eine Beschreibung aus einer der beiden anderen Sichten übertragen. Wenn wir eine Beschreibung aus der Struktur-Sicht in eine Beschreibung aus der Verhaltens-Sicht übersetzen, so nennt man diesen Vorgang *Analyse*. Wir können zum Beispiel eine Netzliste mit einer Beschreibung des Controllers untersuchen und daraus auf den Algorithmus schließen, der von der Schaltung ausgeführt wird.

Die *Synthese* ist das Gegenstück zur Analyse. Sie bezeichnet den Prozeß der Umformung einer Beschreibung aus der Verhaltens-Sicht in eine Beschreibung aus der Struktur-Sicht. Beispiel: Entwurf von Gattern und Flip-Flops nach Vorgabe von Boolschen Ausdrücken.

Als *High-Level Synthese* bezeichnet man den Prozeß der Übersetzung einer Darstellung aus der Verhaltens-Sicht auf algorithmischer Ebene in eine Darstellung der Struktur-Sicht auf Register-Transfer Ebene. Auf Deutsch: die Erstellung einer Netzliste nach Vorgabe eines Algorithmus.

1.2 Start

Um die Syntheseaufgabe zu formulieren, benötigt man eine Spezifikation des Verhaltens des VLSI-Systems. Diese hat die Form eines Algorithmus, formuliert in einer imperativen Hochsprache[3]. Diese kann aus einem Subset einer gängigen Programmiersprache wie C oder PASCAL bestehen, aber gewöhnlich benötigt man eine spezielle Hardwarebeschreibungssprache (HDL[4]) um z.B. Nebenläufigkeiten ausdrücken zu können. Es gibt eine Fülle von HDLs, z.B. AHPL[Hill74], CAP/DSDL[Ramm80], CONLAN[Pilo83], DACAPO[DOS87]. Um eine Standardisierung zu ermöglichen, wurde die Sprache VHDL[IEEE88] normiert. VHDL ist hauptsächlich für die Simulation von Hardware konzipiert und nicht vollständig für die Synthese verwendbar. Wenn sie für Synthese benutzt wird, darf man bestimmte Teile der Sprache nicht benutzen. Abbildung 1.2 zeigt ein Beispiel eines VHDL-Programms, daß die Differentialgleichung

$$\frac{d^2y}{dx^2} + 5\frac{dy}{dx}x + 3y = 0$$

löst.[5]

1.3 Ziel

Das Ziel des Syntheseprozesses ist eine Darstellung des Chips auf Register-Transferebene. Dazu gehören:

- Eine *Netzliste*, die alle funktionalen Einheiten wie ALUs, Addierer, etc., alle Speichereinheiten wie Register, Registerfiles [6] , RAM, etc. und alle Verbindungshardware wie Busse, Multiplexer und Treiber enthält (siehe Abbildung 1.4).

---

[3] Die am IMEC in Leuven entwickelte CATHEDRAL-Synthesewerkzeuge verwenden zur Spezifikation die funktionale Sprache SILAGE[Man87]. Meist werden aber imperative Sprachen benutzt.

[4] HDL = Hardware Description Language

[5] Von [Paul88]. Diese Gleichung ist ein häufig benutztes Benchmark zum Testen von Synthesewerkzeugen.

[6] Ein Registerfile besteht aus mehreren Registern, die zu einer Einheit mit nur einem Eingang und Ausgang zusammengefaßt worden sind.

```
use work.syn_pack.all;

  ENTITY diffeq IS
  PORT ( inport  : IN  integer;
         outport : OUT integer;
         sysclock :IN  bit);
  END diffeq;

  ARCHITECTURE hls_synthesis OF diffeq IS
  BEGIN
    PROCESS
      VARIABLE a, dx, x, u, y: integer;
      VARIABLE x1, y1: integer;
    BEGIN
      cycles(sysclock,1);
      a := inport;
      cycles(sysclock,1);
      dx := inport;
      cycles(sysclock,1);
      y := inport;
      cycles(sysclock,1);
      x := inport;
      cycles(sysclock,1);
      u := inport;
      LOOP
        cycles(sysclock,7);
        x1 := x + dx;
        y1 := y + (u * dx);
        u := u - 5 * x * (u * dx) - 3 * y * dx;
        x:= x1; y := y1;
        EXIT WHEN NOT (x1 < a);
      END LOOP;
      outport <= y;
    END PROCESS;
  END hls_synthesis;
```

Abbildung 1.2: VHDL code für die Differential Gleichung

- Eine Spezifikation des Controllers in Form eines symbolischen *State-Transition-Table* (siehe Tabelle 1.3).

- Ein Kontrollnetz, über das der Controller die funktionalen Einheiten, Register, Multiplexer und Treiber steuert und ein Zustandsnetz, das dem Controller Ergebnisse von Testausdrücken mitteilt.

Wie die Ausgabe nun genau aussieht, hängt vor allem von der verwendeten Architektur ab, ob man also Busse oder nur Multiplexer benutzt, ob ein Taktzyklus zweiphasig ist, etc.

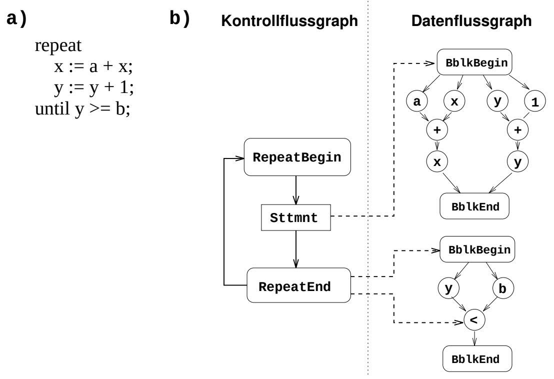

Abbildung 1.3: a) Einfache Repeat-Schleife, b) Kontroll- und Datenflußgraph

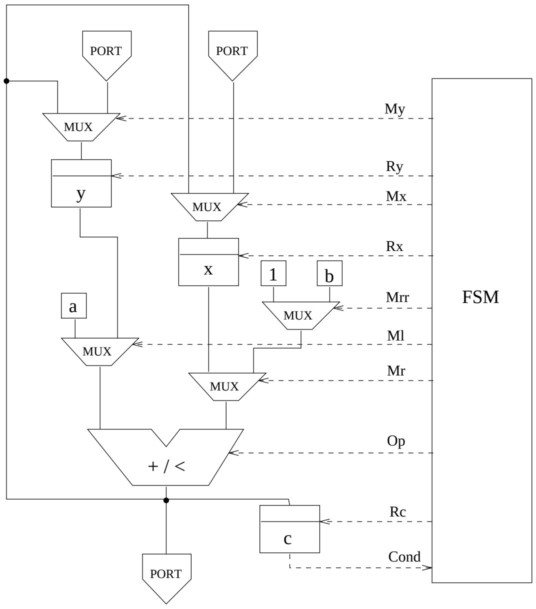

Abbildung 1.4: Netzliste für das Repeat-Beispiel

Cond	State	Next	Rc	My	Ry	Mx	Rx	Mrr	Ml	Mr	Op
x	...	S1	x	0	1	0	1	x	x	x	x
x	S1	S2	x	x	0	1	1	x	1	1	1
x	S2	S3	x	1	1	x	0	1	0	0	1
x	S3	S4	1	x	0	x	0	0	0	0	0
0	S4	S2	x	x	0	1	1	x	1	1	1
1	S4	...	x	x	0	x	x	x	x	x	x

Tabelle 1.1: State-Transistion-Table des Controllers für das Repeat-Beispiel

1.4 Anforderungen an den Entwurf

In der Regel soll der Entwurf bestimmten Vorgaben genügen. Diese können sich z.B. auf die verwendete Architektur, Leistungsaufnahme, Datendurchsatz, Schnittstellen (wie kommuniziert der Chip mit seiner Umgebung?) und Testbarkeit beziehen. Zwei miteinander konkurrierende Leistungskriterien sind:

1. Die Zeit, die der Chip zur Bearbeitung der Aufgabe benötigt.

2. Die Fläche, die der Chip in Anspruch nimmt.

Eine hohe Geschwindigkeit ist z.B. für einen hohen Datendurchsatz erforderlich.

Eine kleine Fläche bedeutet nicht nur oft geringere Hardwarekosten, sondern ist auch oft dann von Nöten, wenn der Chip in eine bereits vorhandene Umgebung eingebaut werden soll und nicht soviel Platz wegnehmen darf.

Für eine hohe Geschwindigkeit braucht man aber ein hohes Maß an Parallelität, also mehr Hardware. Umgekehrt passen auf eine kleine Fläche nicht so viele Bausteine, weshalb die Berechnungen sequentiell durchgeführt werden müssen, wodurch der Chip langsamer wird.

Während man die Anzahl von Zyklen, die der Chip für die Bearbeitung benötigt, relativ leicht während des Syntheseprozesses bestimmen kann, stellt die Vorgabe der Chipfläche ein Problem dar. Denn welche Fläche letztendlich beansprucht wird, hängt vom Layout ab, also von der Frage, wo genau die einzelnen

Bausteine, Busse, etc. plaziert sind. Die Erstellung des Layouts erfolgt aber erst *nach* der High-Level-Synthese! Um dieses Problem zu mildern, kann man die Fläche des Chips während der High-Level Synthese nach Erfahrungswerten abschätzen. Außerdem werden meist mehrere verschiedene Designs entworfen und modifiziert, bis man schließlich einen Entwurf wählt (oder auch nicht).

1.5 Klassische Teilaufgaben der High-Level Synthese

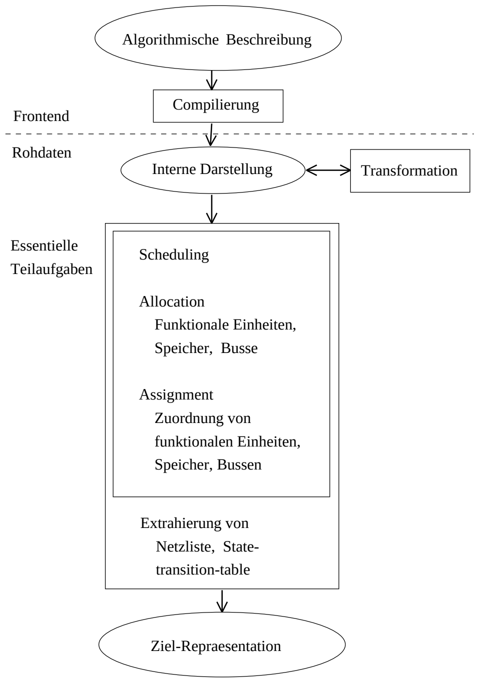

Abbildung 1.5: Klassische Teilaufgaben der High-Level Synthese

Während der High-Level Synthese müssen einige Teilaufgaben bewältigt werden. Dazu gehören:

- Erstellung einer *Internen Darstellung.*
- *Scheduling*
- *Allocation*
- *Assignment*

Die interne Darstellung hat die Form eines Kontrollfluß- und Datenfluß-Graphen (siehe Abbildung 1.3 b)). Dafür wird das HDL-Programm compiliert. Anschließend können Manipulationen am Datenflußgraphen (z.B. Verlagerung von Operationen durch Anwendung des Assoziativgesetzes) und am Kontrollflußgraphen (z.B. Umwandlung von Schleifen mit konstanter Laufzeit in Sequenzen) durchgeführt werden.

Beim Scheduling wird jede Operation einem bestimmten Takzyklus zugeordnet.

Bei der Allocation bestimmt man Anzahl und Typ von funktionalen Einheiten (ALUs, Addierer, etc.), Registern und Bussen.

Das Assignment ordnet dann die Operationen den verwendeten funktionalen Einheiten, die Variablen den verwendeten Registern und die Kanten des Datenflußgraphen den verwendeten Bussen zu.

Scheduling, Allocation und Assignment sind im hohen Maße voneinander abhängig, man kann also diese Teilaufgaben nicht voneinander getrennt bearbeiten. Auch die Reihenfolge der Durchführung kann variieren. Oft werden Allocation und Assignment zusammen in einem Durchgang erledigt, oder Anzahl und Typ von Bausteinen sind bereits vorgegeben, so daß eine Allocation überflüssig ist.

Hat man alle diese Teilaufgaben erledigt, so verfügt man über alle nötigen Informationen, um die Zielrepresentation auf Register-Transfer-Ebene, sprich eine Netzliste mit allen Bausteinen, Registern, Bussen und Verbindungen und eine Darstellung des Controllers in Tabellenform, auszugeben.

1.5.1 Interne Darstellung

Das als Eingabe benutzte HDL-Programm enthält implizit alle Informationen, die man zur Synthese braucht. Allerdings ist eine rein textuelle Eingabe nur umständlich zu bearbeiten, deshalb wird das Programm compiliert und in eine interne Darstellung in Form eines Datenfluß- und Kontrollfluß-Graphen übersetzt.

Das Konzept des Kontrollflusses stammt vom *von Neumann Rechner*, in welchem – kontrolliert von einem Programmzähler – Instruktionen zur sequentiellen Ausführung ausgewählt werden. Imperative Programmiersprachen enthalten Konstrukte zur expliziten Spezifikation des Kontrollflusses wie Sequenzen, bedingte Sprünge und Schleifen. Der Kontrollfluß läßt sich in einem Graphen darstellen, dessen Knoten aus Basisblöcken bestehen und dessen Kanten den Kontrollfluß beschreiben. Basisblöcke sind Codesequenzen, innerhalb derer sich der Kontrollfluß nicht ändert.

Der Datenflußgraph enthält als Knoten alle verwendeten Operationen (und ggf. Variablen). Er enthält eine Kante (v1,v2) genau dann, wenn Knoten v2 das Ergebnis von Operation v1 (bzw. die Variable v1) als Eingabe benötigt.

Kontroll- und Datenflußgraph können entweder getrennt oder in einem Hypergraphen miteinander vereint implementiert werden. Abbildung 1.3 b) zeigt ein Beispiel für getrennte Graphen.

1.5.2 Scheduling

Beim Scheduling werden die einzelnen Operationen des Datenflußgraphen bestimmeten Taktzyklen, in denen sie ausgeführt werden sollen, zugeordnet. Hierbei gibt es zwei verschiedene Ansätze:

1. Es wird die Zeit (in Taktzyklen) vorgegeben, in der der Chip den Algorithmus abarbeiten soll mit dem Ziel, die Operationen so zu schedulen, daß möglichst wenig Hardwarebausteine benötigt werden. Diese Art des Scheduling wird *time-constrained* Scheduling genannt.

2. Es werden Anzahl und Typ von Hardwarebausteinen, auf denen die Operationen ausgeführt werden sollen, vorgegeben mit dem Ziel, ein Schedule zu finden, daß möglichst wenig Taktzyklen beansprucht. Diese Art des Scheduling wird *resource-constrained* Scheduling genannt.

1.5.2.1 Time-constrained Scheduling

Beim time-constrained Scheduling ist die Anzahl von Zyklen, in denen der Algorithmus bearbeitet werden soll, vorgegeben. Gesucht ist eine Verteilung von Operationen auf Zyklen, bei der möglichst wenig Hardwarebausteine benötigt werden. Betrachten wir als Beispiel den Datenflußgraphen von Ab-

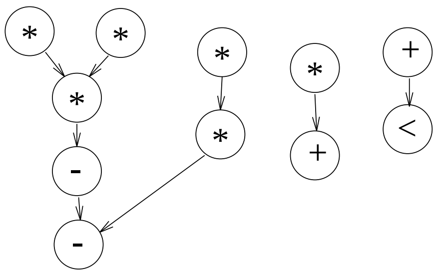

Abbildung 1.6: Datenflußgraph (schematische Darstellung)

bildung 1.6. Dieser Datenflußgraph soll nun auf 4 Takzyklen geschedult werden. Dieses können wir z.B. tun, indem wir annehmen, daß alle Operationen so früh wie möglich ausgeführt werden sollen.

Wir ordnen also eine Operation dem ersten Taktzyklus zu, zu dem alle ihre benötigten Operanden bereits berechnet sind. Diese Art des Schedulens nennt man *ASAP*[7]*-Scheduling*. Analog können wir versuchen, die Operationen zum spätest möglichen Zeitpunkt zu schedulen. Diese Art des Schedulens nennt man entsprechend *ALAP*[8]*-Scheduling*. Durch ASAP- und ALAP-Scheduling haben wir nun auch für jede Operation den Zeitrahmen gefunden, innerhalb dessen sie geschedult werden kann (siehe Abbildung 1.7. Die Minus-Operationen können nur zu einem Zeitpunkt geschedult werden, ihre Timeframelängen betragen also 1. Die Plus-Operationen können auf je 3 Taktzyklen geschedult werden, ihre Timeframelängen haben daher je die Länge 3).

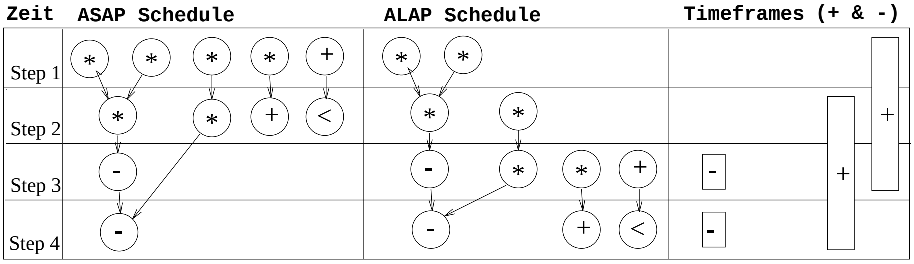

Abbildung 1.7: ASAP-/ALAP-Schedules und Timeframes der Operationen + und −.

Gehen wir nun davon aus, daß die Operationen +, − und < durch ALUs und die Multiplikation durch Multiplizierer mit einem Delay[9] von je einem Takzyklus ausgeführt werden sollen[10]. Dann benötigen wir im Fall des ASAP-Schedules 4 Multiplizierer, da im ersten Takt 4 Multiplikationen gleichzeitig ausgeführt werden müssen. Die beiden Multiplikationen in Takt 2 können dann auf je einem dieser Multiplizierer ausgeführt werden, da die Multiplizierer nach dem ersten Takt wieder frei sind. Außerdem brauchen wir 2 ALUs, da im zweiten Takt gleichzeitig ein '+' und ein '<' ausgeführt werden. Im Falle des ALAP-Schedules brauchen wir 2 Multiplizierer (für die Takte 1, 2 und 3) und 3 ALUs (weil im vierten Takt 3 ALU-Operationen ausgeführt werden).

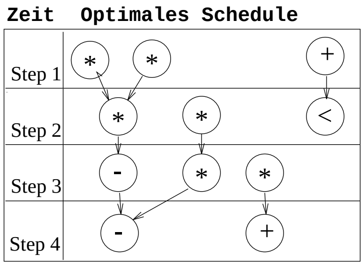

Abbildung 1.8: Optimales Schedule

Die ASAP- und ALAP-Schedules sind aber nicht optimal. Im Falle eines optimalen Schedules benötigen wir nur 2 Multiplizierer und 2 ALUS (siehe Abbildung 1.8.).

Das Problem: die Aufgabe, für einen Datenflußgraphen ein optimales Schedule zu ermitteln, ist *NP-vollständig*. Es gibt aber heuristische Verfahren, die in polynomieller Zeit arbeiten und in der Praxis gute bis sehr gute Ergebnisse liefern, so zum Beispiel das *Force-Directed Scheduling*[Paul88, PaKn89b].

[7] ASAP = As Soon As Possible

[8] ALAP = As Late As Possible

[9] Delay = Die Zeit, die nach Vorhandensein der Operanden vergeht, bis das Ergebnis am Ausgang einer funktionalen Einheit steht.

[10] Es gibt auch Einheiten mit einem Delay > 1, z.B. bestimmte Typen von Multiplizierern, die 2 oder mehr Taktzyklen benötigen (das nennt man dann *Multicycling*). Es kann auch sein, daß z.B. Addierer so schnell sind, daß zwei Additionen in einem Takt ausgeführt werden können. Werden zwei Operationen in einem Takt hintereinander gehängt, nennt man das *Chaining*.

1.5.2.2 Resource-constrained Scheduling

Beim resource-constrained Scheduling sind Typ und Anzahl von funktionalen Einheiten vorgegeben. Ziel ist es, ein Schedule zu finden, das diese Vorgabe erfüllt und möglichst wenig Taktzyklen benötigt. Leider ist auch dieses Problem *NP-vollständig*. Es gibt aber auch hier wieder heuristische Verfahren, die akzeptable Ergebnisse in polynomieller Zeit liefern. Als Beispiel wird das *List-Scheduling* vorgestellt.

Beim List-Scheduling versucht man erst, so viele Operationen wie möglich auf den ersten Taktzyklus zu schedulen, dann werden möglichst viele der restlichen Operationen auf den zweiten Zyklus, dritten Zyklus usw. geschedult. Damit eine Operation für eine Zuordnung zu einem bestimmten Zyklus in Frage kommt, müssen alle Vorgänger-Operationen dieser Operation (im Datenflußgraphen) bereits geschedult sein.

Die Operationen, die für das Schedulen auf einen bestimmten Taktzyklus in Frage kommen, werden nach einer Prioritätsfunktion sortiert, so daß Operationen mit hoher Priorität zuerst geschedult werden. Sind in einem Zyklus alle Resourcen, die einen bestimmten Operationstyp ausführen, bereits vergeben, so müssen Operationen dieses Typs, die noch nicht geschedult sind, zu einem späteren Zeitpunkt geschedult werden. Als Prioritätsfunktion kann man zum Beispiel die Beweglichkeit (*mobility*)

Eingabe: V = Menge der Operationen des Datenflußgraphen
Ausgabe: $Schedule \subset V \times \mathbb{N}$

$cstep := 0;$
$Schedule := \phi;$
while $(V \neq \phi)$ **do**
  $cstep := cstep + 1;$
  $Ops := Ready\_Ops(V);$
  $Ops := Sort(Ops, \mathcal{P});$
  $Release\_Resources();$
  while $(Ops \neq \phi)$ **do**
    $op :=$ HEAD(Ops); $Ops :=$ TAIL(Ops);
    if $Resource\_Free\_For(op)$ **then**
      $Schedule := Schedule \cup \{(op, cstep)\};$
      $Reserve\_Resource(op);$
      $V := V \setminus \{op\};$
    endif
  endwhile
endwhile

Abbildung 1.9: Algorithmus *List-Scheduling*

einer Operation wählen. Die Beweglichkeit einer Operation ist die Länge ihres Timeframes, den man durch die ASAP- und ALAP-Schedules erhält, wenn die minimale Anzahl von Taktzyklen vorgegeben ist. Die minimale Anzahl von Taktzyklen wiederum ist gleich der Länge des längsten Pfades im Datenflußgraphen (welchen man auch *kritischen Pfad* nennt). Je geringer die Mobilität einer Operation ist, desto größer ist ihre Priorität.

Abbildung 1.9 zeigt den Algorithmus. *Ready_Ops* liefert die Menge von Operationen zurück, die geschedult werden können, deren Operanden also bereits berechnet sind. *Sort* sortiert diese Operationen nach der Prioritätsfunktion $\mathcal{P}$. *Release_Resources* zeigt an, daß die im vorherigen Schritt benutzten Resourcen wieder freigegeben werden. *Resource_Free_For* ist wahr, wenn für die Operation noch eine Resource frei ist, und *Reserve_Resource* markiert diese Resource als benutzt.

1.5.3 Allocation & Assignment

Bei der Allocation müssen Anzahl und Typ von funktionalen Einheiten und die Anzahl von Bussen und Registern ermittelt werden. Beim Assignment werden dann die einzelnen Operationen funktionalen Einheiten, die Variablen Registern und die Datentransfere Bussen zugeordnet.

Falls nicht schon vorher Typ und Anzahl von funktionalen Einheiten vorgegeben sind, werden diese nach dem Scheduling ermittelt. Die Anzahl von Modulen eines bestimmten Typs ist gleich der maximalen Anzahl von Operationen, die in einem Taktzyklus parallel ausgeführt werden und diesen Modultyp benötigen. Beim Assignment kann man nun z.B. so vorgehen, daß zwei Operationen bevorzugt auf dem gleichen Modul ausgeführt werden, wenn sie eine möglichst große Anzahl von gemeinsamen Vorgängern und Nachfolgern besitzen. Dann nämlich spart man mit hoher Wahrscheinlichkeit ein paar Multiplexer und Treiber.

Für die Allocation von Registern betrachtet man die *Lifetime-Analyse* der Variablen. Eine Variable wird *geboren*, wenn sie erzeugt wird, und sie *stirbt*, wenn sie das letzte Mal benutzt wird. Die Anzahl von benötigten Registern ist dann die maximale Anzahl von gleichzeitig lebenden Variablen zu einem beliebigen Zeitpunkt. Abbilding 1.10 zeigt einen Datenflußgraphen und die zugehörigen Lifetime-Intervalle der Variablen. Da zu jedem Zeitpunkt maximal 3 Variablen leben, werden 3 Register benötigt. Um die Variablen jetzt den einzelnen Registern zuzuordnen, kann man wie bei den Operationen die gemeinsamen Vorgänger und Nachfolger berücksichtigen, damit die Anzahl von Multiplexern und Treibern minimiert wird.

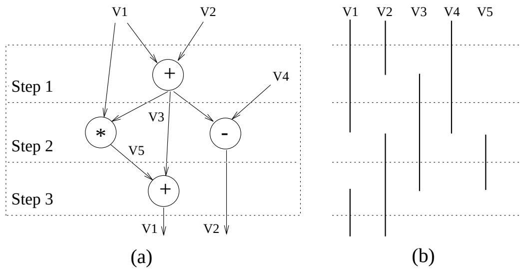

Abbildung 1.10: (a) Datenflußgraph, (b) zugehörige Liftime-Intervalle der Variablen

Um die Anzahl von Bussen zu bestimmen, betrachtet man die maximale Anzahl von gleichzeitig durchgeführten Datentransferen. Zwei Datentransfere werden dann bevorzugt auf dem gleichen Bus durchgeführt, wenn ihre Quellen und/oder Ziele identisch sind, so daß man auch hier Multiplexer und Treiber spart.

Die Allocation ist in allen drei Fällen einfach und kann sehr schnell durchgeführt werden. Das Problem, ein optimales Assignment mit einer minimalen Anzahl von Multiplexern und/oder Treibern zu finden, ist allerdings wieder *NP-vollständig*. Aber auch hier gibt es heuristische Verfahren, die in polynomieller Zeit laufen, siehe den *Weighted Allocation Algorithmus*, der im zweiten Kapitel beschrieben wird.

1.5.4 Zur Literatur

Der größte Teil dieses Kapitels wurde mit Hilfe des Buches [MiLaDu92] erstellt. Als weitere Einführung in den Bereich High-Level Synthese eignet sich außerdem [GaDu92].

Kapitel 2

PASS

2.1 Einführung

Das High-Level-Synthese-Tool PASS[1] wurde von Christian Ewering und Gunter Gerhardt an der Universität Paderborn entwickelt und 1990 auf dem *Sixteenth Symposium on Microprocessing and Microprogramming* [EwGe90] vorgestellt. Dabei ging es vor allem um zwei neue Methoden, die PASS von anderen Synthese-Tools unterscheidet:

1. Beim Schedulen werden bedingte Sprünge berücksichtigt, d.h. nicht nur der Datenfluß, sondern auch der Kontrollfluß beeinflußt das Ergebnis.

2. Für Allocation und Assignment von funktionalen Einheiten, Registern, Registerfiles und Bussen wird *ein* zentraler Algorithmus verwendet.

Die Ausführungszeit des Chips hängt von der Anzahl von Controlsteps und dem Delay pro Stufe ab. Das Delay einer Stufe setzt sich aus dem maximalen Delay der funktionalen Einheiten dieser Stufe und dem Delay des Controllers zusammen.
Beispiel: in einer Stufe wird eine Subtraktion durchgeführt, und der Controller sendet Signale, um den nächsten Schritt einzuleiten.
Diese Aktionen können parallel ausgeführt werden, allerdings gibt es meist eine Verzögerung bei bedingten Sprüngen, da der Controller erst auf das Ergebnis des Tests warten muß. PASS versucht, ein Schedule zu finden, in dem so eine Verzögerung vermieden wird, wodurch sich die gesamte Ausführungszeit verringert.

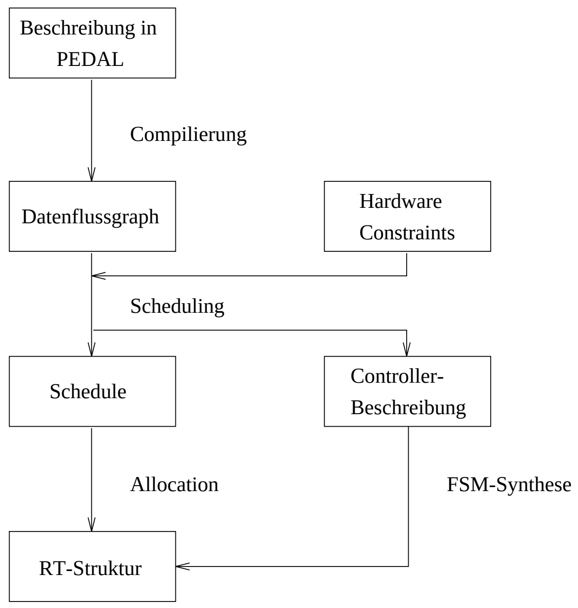

Abbildung 2.1: Überblick über das System

[1]PASS = PAderborner Synthesis System

2.2 PEDAL

Zur Spezifikation der Aufgabe wird die funktionale, blockorientierte Datenflußsprache PEDAL[2] benutzt. Ziel beim Entwurf dieser Sprache war es, eine kleine Sprache zu erzeugen, die Hardwaresynthese unterstützt. In einer *Datenflußsprache* korrespondiert jedes Sprachkonstrukt mit einem Subgraphen des zugehörigen Datenflußgraphen, weshalb der Datenflußgraph ohne großen Aufwand erzeugt werden kann. So werden zum Beispiel potentielle Nebenläufigkeiten einfach daran erkannt, daß zwischen Operationen keine Datenabhängigkeiten bestehen. Auf spezielle Kontrukte zum Ausdrücken von Nebenläufigkeiten kann man völlig verzichten. In Abbildung 2.2 sind die add-Funktion und das if-Konstrukt parallel, da es zwischen beiden keine Datenabhängigkeiten gibt.

```
.
.
.
if less(a, sub(b, c))
   then ...
   else ...
fi,
add(d, e),
.
.
.
```

Abbildung 2.2: PEDAL Programmfragment

PEDAL wird hauptsächlich zur Beschreibung von Algorithmen und nicht von Hardwarestrukturen benutzt, deshalb beschränken sich die zur Beschreibung von Hardware verwendeten Konstrukte auf die Spezifikation der Schnittstellen und der verwendeten Bitbreite. Alle anderen Konstrukte sind Standardkonzepte, wie explizite E/A, überladene Funktionen und if-, case- und loop-Ausdrücke. Die Sprache enthält keine vordefinierten Funktionen oder Operationen, statt dessen werden diese von einer benutzerdefinierten Hardwarebibliothek bereitgestellt. Funktionen können einfach durch Angabe des Namens und der Anzahl und Bitbreite von Input- und Output-Ports deklariert werden.

2.3 Scheduling

Zum Schedulen des Datenflußgraphen wird das in 1.5.2.2 beschriebene List-Scheduling benutzt. Der Benutzer kann dabei zwischen zwei Prioritätsfunktionen wählen:

1. Operationen auf dem kritischen Pfad (das sind die Operationen, die bei einem zeitlich minimalen Schedule nicht verschoben werden können, ohne das Schedule zu verlängern) werden bevorzugt.

2. Operationen mit vielen (nicht notwendigerweise direkten) Nachfolgern werden bevorzugt.

Die letzte Funktion führte bei Testentwürfen immer zu den kürzesten Schedules.

2.3.1 Hardware constraints

Es wird resource-constrained Scheduling durchgeführt, deshalb werden am Anfang der Schedulingphase Anzahl und Typ von funktionalen Einheiten eingegeben. Die für die Synthese wichtigen Informationen über die funktionalen Einheiten sind in einer Hardwarebibliothek gespeichert. Pro funktionaler Einheit werden folgende Informationen abgefragt:

- Die Fläche, die von der Komponente beansprucht wird

[2] PEDAL = Paderborner Experimental Data flow Language

- Ein- und Ausgabeports (Name und Bitbreite)
- Eine Liste aller von der Komponente ausführbaren Funktionen. Für jede Funktion wird angegeben:
  - Name
  - Benutzte Ports
  - Delay
  - Latency

Als *latency* (Latenz) einer Komponente bezeichnet man die Zeit, die nach dem Anliegen eines Eingangssignal vergehen muß, ehe ein neues Eingangssignal an die Komponente gesendet werden darf. Ein Beispiel für einen einfachen Bibliothekseintrag zeigt Abbildung 2.3.

```
CONFIGURATION onemultifunc IS
MODULE multi SPACE 50 INSTANCES 1
IN a 8, b 8
OUT c 8, d 1
  FUNC sub  DELAY 1 LATENCY 1
            IN a, b OUT c
  FUNC add  DELAY 1 LATENCY 1
            IN a, b OUT c
  FUNC less DELAY 1 LATENCY 1
            IN a, b OUT d
```

Abbildung 2.3: Hardware Konfiguration

Falls vor dem Scheduling keine Hardwareconstraints angegeben werden, so wird ein maximal paralleles Schedule ermittelt, und die dazu notwendige Hardwarekonfiguration wird nach dem Schedulen ausgegeben.

2.3.2 Bedingte Sprünge

Es gibt zwei Möglichkeiten, wie Daten- und Kontrollteil zusammen arbeiten können: parallel mit einer Phase oder sequentiell mit zwei Phasen. Im sequentiellen Fall ist in der ersten Phase der Datenteil aktiv und der Kontrollteil inaktiv, in der zweiten Phase ist es umgekehrt. Im parallelen Fall sind beide Teile zur gleichen Zeit aktiv. Betrachten wir zum Datenflußgraphen in Abbildung 2.4 die Schedules

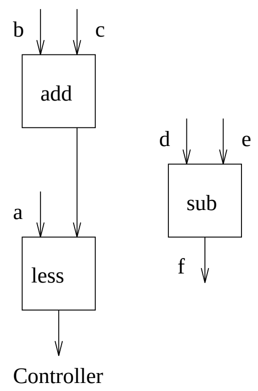

Abbildung 2.4: Datenflußgraph

Control Step	Phase	sequentiell	
		Datenteil	Kontrollteil
1	a	sub	
	b		$1 -> 2$
2	a	add	
	b		$2 -> 3$
3	a	less	
	b		cond. branch

Control Step	parallel	
	Datenteil	Kontrollteil
1	sub	$1 -> 2$
2	add	$2 -> 3$
3	less	$3 -> 4$
4		cond. branch

Abbildung 2.5: Schedule A

für den sequentiellen und parallelen Fall in Abbildung 2.5. Im parallelen Fall tritt ein Problem auf: in Schritt 4 ist der Datenteil inaktiv, da der Kontrollteil erst das Ergebnis der less-Operation aus Schritt 3 auswerten muß. Ein kürzeres Schedule zeigt Abbildung 2.6. Dort wird das *less* gleich nach dem *add* durchgeführt, so daß im letzten Schritt während der Auswertung des Ergebnisses von *less* noch parallel das *sub* ausgeführt werden kann.

Genau diese Optimierung wird auch beim Schedulen in PASS unterstützt. Dazu werden während des Schedulens konditionale Operationen wie *less* bevorzugt geschedult. Hinter diesen Operationen wird dann im Schedule eine Lücke geschaffen, und es wird versucht, diese Lücke mit nicht-konditionalen Operationen zu füllen.

Control Step	parallel	
	Datenteil	Kontrollteil
1	add	$1 -> 2$
2	less	$2 -> 3$
3	sub	cond. branch

Abbildung 2.6: Schedule B

2.4 Allokation

In der Allokationsphase werden die Anzahlen der funktionalen Einheiten, Register, Registerfiles und der Busse festgelegt. Symbolische Resourcen werden realen Resourcen zugeordnet, z.B. Operationen zu funktionalen Einheiten, Kanten zu Registern etc.

Diese Teilaufgaben können als Färbungsproblem auf einem Graphen formuliert werden. So kann man z.B. Operationen als Knoten auffassen, und Kanten zwischen zwei Knoten existieren genau dann, wenn die zugehörigen Operationen sich zeitlich überschneiden. Färbt man dann den Graphen so, daß adjazente Knoten unterschiedliche Farben haben, so kann man alle Operationen einer Farbe auf der gleichen funktionalen Einheit ausführen. Jede Farbe korrespondiert mit einer funktionalen Einheit, also erhält man eine Zuordnung von Operationen auf eine minimale Anzahl von funktionalen Einheiten, wenn der Graph mit der minimalen Anzahl von Farben gefärbt wurde.

Eine minimale Färbung ist aber nicht unbedingt das primäre Ziel. Die Anzahl von Farben sagt nämlich nichts darüber aus, wie die Verbindungsstruktur des resultierenden Entwurfs aussieht. Die Anzahl von Multiplexern und "Drähten" hängt z.B. davon ab, wie eng benachbart die Operationen, die auf einer funktionalen Einheit ausgeführt werden, sind. Aus diesem Grund wird ein Präferenzgraph gebildet, dessen Kanten mit Gewichten versehen sind. Ist das Gewicht einer Kante hoch, so sollen die adjazenten Knoten bevorzugt mit der gleichen Farbe gefärbt werden.

PASS verwendet zur Allokation von funktionalen Einheiten, Registern, Registerfiles und Bussen den sogenannten *Weighted Allocation Algorithmus*. Die Eingabe für den Algorithmus besteht aus zwei Graphen: dem *Interferenzgraphen* (IG) und dem *Präferenzgraphen* (PG).

Eine Kante verbindet zwei Knoten in IG, wenn diese nicht der gleichen Resource zugeordnet werden dürfen. In PG gibt es eine Kante zwischen zwei Knoten, wenn deren Zuordnung zu der gleichen Resource eine Verbesserung der Verbindungsstruktur verspricht. Außerdem besitzen die Kanten in PG ein Gewicht, daß die Priorität ausdrückt, mit der zwei Knoten einer Resource zugeordnet werden.

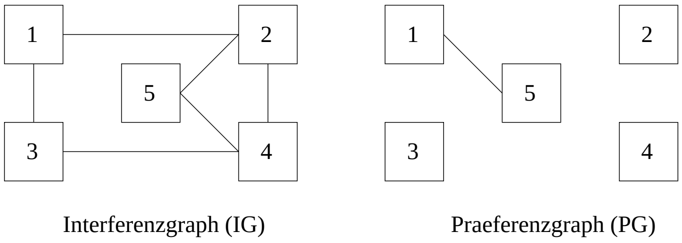

Abbildung 2.7: Datenstruktur zur Allokation

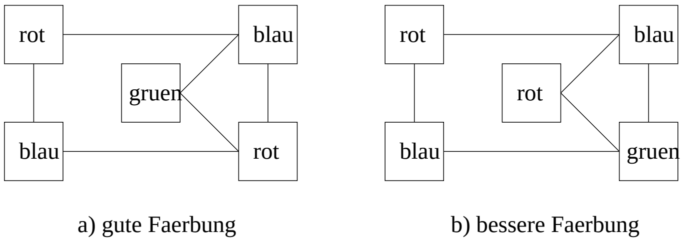

Abbildung 2.8: Gefärbte Graphen

Abbildung 2.7 zeigt ein Beispiel für die beiden Graphen. Die Knoten werden durch Zahlen identifiziert. Abbildung 2.8 a) zeigt eine mögliche Dreifärbung von IG ohne Berücksichtigung von PG. In Abbildung 2.8 b) wird dagegen die Präferenz der Knoten 1 und 5 berücksichtigt, so daß sie die gleiche Farbe erhalten.

Um das Allokationsproblem zu lösen, wird IG durch den Algorithmus von Abbildung 2.9 gefärbt. Dabei gibt es zwei Strategien: durch den Aufruf *allocate(PG,IG)* maximieren wir die Anzahl von erfüllten Präferenzen, durch den Aufruf *allocate(IG,IG)* minimieren wir die Anzahl benötigter Resourcen. Der

proc *allocate(RG,IG)*
while *RG is not empty* **do**
    k = node with lowest degree in RG;
    push(stack, k);
    RG := RG - k;
endwhile*;*
restore RG;
while *not empty(stack)* **do**
    k = pop(stack);
    select a color with respect to IG
    which maximizes the weighted preferences;
endwhile*;*
endproc

Abbildung 2.9: Weighted Allocation Algorithmus

erste Parameter RG bestimmt die Reihenfolge, in der die Knoten gefärbt werden. Ist ein Knoten nur mit einer Kante verbunden, so sollte zunächst der Nachbarknoten gefärbt werden. Um dieses zu erreichen, werden sukzessive die Knoten mit dem kleinsten Eingangsgrad gelöscht und auf einen Stapel gepackt, so daß in der zweiten Schleife, wenn die Knoten einzeln vom Stapel geholt werden, die Knoten mit großen Eingangsgraden zuerst gefärbt werden. Um den Präferenzen gerecht zu werden, erhält in der zweiten Schleife ein Knoten die Farbe, die die Summe von erfüllten gewichteten Präferenzen maximiert, d.h. wenn ein Knoten 1 mit bereits gefärbten Knoten 2 und 3 verbunden ist, dann erhält er die Farbe von Knoten 2, falls das Gewicht von (1,2) gröser ist als das von (1,3). Andernfalls erhält er die Farbe von Knoten 3.

Die Kanten der Graphen und ihre Gewichte werden so berechnet:

- **Allokation funktionaler Einheiten**: Eine Kante wird in IG zwischen zwei Operationen eingefügt, wenn sie sich zeitlich überschneiden, denn dann können diese Operationen nicht auf der gleichen Einheit ausgeführt werden.

  Eine Präferenz mit Gewicht 2 wird in PG eingefügt, falls die Operationen von den gleichen symbolischen Registern lesen. Falls diese Operationen auf der gleichen Einheit ausgeführt werden, sparen wir einen Multiplexereingang. Eine Präferenz mit Gewicht 1 wird eingefügt, falls der Output der einen Operation der Input der anderen ist, denn wenn diese Operationen auf der gleichen Einheit ausgeführt werden, sparen wir Drähte.

- **Allokation von Registern**: Eine Kante wird in IG zwischen zwei Variablen eingefügt, wenn sich ihre Lebenszeiten überlappen.

  Eine Präferenz mit Gewicht 2 wird in PG eingefügt, falls die Eingabe von zwei Variablen von der gleichen funktionalen Einheit stammt. Werden diese zwei Variablen dem gleichen realen Register zugewiesen, sparen wir einen Multiplexereingang. Eine Präferenz mit Gewicht 1 wird eingefügt, falls zwei Variablen von der gleichen funktionalen Einheit gelesen werden. Werden diese Variablen dem gleichen Register zugewiesen, so sparen wir Drähte und evtl. einen Multiplexer.

- **Allokation von Registerfiles**: Eine Kante wird in IG zwischen zwei Registern eingefügt, die zu einem Zeitpunkt beide gelesen oder geschrieben werden.

  Eine Präferenz wird in PG eingefügt, falls die Register auf den gleichen Multiplexer schreiben. Das Gewicht ist die Anzahl von gemeinsamen Zielen.

- **Allokation von Bussen**: Eine Kante wird in IG zwischen allen Multiplexern eingefügt, die zu einem Zeitpunkt gleichzeitig aktiv sind.

  Eine Präferenz wird in PG eingefügt, falls die Inputs zweier Multiplexer von den gleichen Quellen stammen. Das Gewicht ist die Anzahl gemeinsamer Quellen.

Die Gewichte der verschiedenen Präferenztypen sind ein Parameter des Allokationsalgorithmus. Werden die Gewichte geändert, so ändern sich auch die Eigenschaften des Entwurfs. Die oben genannten Gewichte dienen zur Minimierung von Multiplexerinputs.

2.5 Ergebnisse von Experimenten

	HAL	PASS
C-steps	19	19
Register	12	10
Busse	6	9
Bus-Inputs	26	39
CPU sec.	960	2

Abbildung 2.10: Entwurf für den *fifth-order wave digital elliptic filter*, verglichen mit den Ergebnissen von HAL

Es sei hier kurz des Resultat des Entwurfs von PASS für den *fifth-order wave digital elliptic filter* [KWK85] mit dem eines anderen Synthesesystems, HAL[PaKn89a], verglichen. Die Anzahl von Businputs ist größer, weil zum Zeitpunkt des Experiments bestimmte Optimierungen weggelassen wurden. HAL benutzt zwei zusätzliche Register, um die Anzahl von Bussen und Businputs zu minimieren und benötigt 16 Minuten für Scheduling und Allokation auf einer Xerox 1108 (PASS lief auf einer SUN 3/50).

Literaturverzeichnis

[DOS87] Fa. DOSIS, *DACAPO II, User Manual, Version 3.0*, DOSIS GmbH, Dortmund, 1987.

[EwGe90] C.Ewering, G.Gerhardt, "PASS: High Level Synthesis" *16th Symposium Microprocessing and Microprogramming*, 8, 1990.

[GaDu92] D.Gajski, N.Dutt, *High-Level Synthesis. Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[GaKu83] D.Gajski, R.Kuhn, "Guest Editors' Introduction: New VLSI Tools" *IEEE Computer*, vol. 16, no. 12, pp. 11-14, December 1983.

[Hill74] F.J.Hill, "Introducing AHPL" *IEEE Computer*, 7(12), 12 1974.

[IEEE88] Design Automation Standards Subcommittee of the IEEE, *IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-87)*, IEEE Inc., New York, 1988

[KWK85] S.Y.Kung, H.J.Whitehouse, T.Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, pages 256-264, 1985.

[Man87] H.De Man, J.Rabaey, "CATHEDRAL II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip" in: *G.De Micheli: Design Systems for VLSI Circuits-Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1987.

[Mar93] P.Marwedel, *Synthese und Simulation von VLSI-Systemen*, Hanser Studien Bücher, 1993.

[MiLaDu92] P.Michel, U.Lauther, P.Duzy *The Synthesis Approach to Digital System Design*, Kluwer Academic Publishers, 1992.

[PaKn89a] P.G.Paulin, J.P.Knight, "Scheduling and Binding Algorithms for High-Level Synthesis" *Proceedings of the 26th ACM/IEEE Design Automation Conference (DAC)*, pages 1-6, 1989.

[PaKn89b] P.G.Paulin, J.P.Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs" *IEEE Transactions on CAD*, pages 661-679, June 1989.

[Paul88] P.G.Paulin, *High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms*, PhD thesis, Carleton University, 1988.

[Pilo83] R.Piloty, M.Barbacci, D.Borrione, D.Dietmeyer, F.Hill, P.Skelly, "CONLAN Report" *Lecture Notes in Computer Science No. 151*, Springer, 1983.

[Ramm80] F.J.Rammig, *Preliminary CAP/DSDL Language Reference Manual. Technical Report 129*, University of Dortmund, Dept. Computer Science, 1980.