# Einsatz genetischer Algorithmen als Optimierungsmethode bei der Synthese elektronischer Schaltungen

**ANDREAS FALKENBERG**

**1998**

## Abstract

Digitale Schaltungen begegnen uns inzwischen in fast allen Lebensbereichen. Die Entwicklung von hochintegrierten Elektronikbausteinen wie Mikroprozessoren, digitalen Signalprozessoren oder ASICs wird aufgrund der steigenden Komplexität der Schaltungen immer aufwendiger. Es müssen immer leistungsfähigere Werkzeuge, im wesentlichen Software, zur Unterstützung der Entwickler angeboten werden. Im Rahmen der vorliegenden Arbeit wird der Ablauf der Hardwareentwicklung ausführlich dargestellt. Die Möglichkeiten für den Einsatz von Werkzeugen zur Beschleunigung der Entwicklung und Optimierung der Entwicklungsergebnisse wird betrachtet und verglichen. Im Besonderen wird der Einsatz genetischer Algorithmen für die parallele Synthese auf Standardworkstations untersucht. In dieser Arbeit werden die benötigten Algorithmen sowie der theoretische Hintergrund entwickelt und ausführlich dargestellt. Durch die Nutzung der vorgestellten Methoden lassen sich sowohl eine Optimierung der Ergebnisse als auch eine Beschleunigung der Synthese erreichen.

## Stichwörter

VHDL, Synthese, genetische Algorithmen, digitale Schaltung, parallele Algorithmen, VLSI, ASIC

## Kurztitel

Falkenberg RTL-Synthese mit genetischen Algorithmen

## Danksagung

An dieser Stelle möchte ich Herrn Professor Dr. Tavangarian, Leiter des Instituts für Technische Informatik an der Universität Rostock, für die Betreuung der vorliegenden Arbeit, seiner steten Diskussionsbereitschaft und die Übernahme des Referates herzlich danken. Des weiteren gilt mein besonderer Dank auch Herrn Dr. Damaschke für die Übernahme des Koreferates, seiner sorgfältigen Begutachtung der Arbeit und der gewinnbringenden Hinweise und Verbesserungsvorschläge. Nicht zuletzt gilt mein Dank den Mitarbeitern des Lehrstuhls für Rechnerarchitektur der Universität Rostock, und dort besonders Herrn Burchert, ohne dessen Hinweise und Unterstützung einige wichtige Ergebnisse dieser Arbeit nicht hätten erzielt werden können. Des weiteren möchte ich Herrn Professor Dr. Verbeek und den Mitarbeitern des Lehrstuhls für theoretische Informatik der Fernuniversität Hagen meinen Dank aussprechen. Meiner Frau Tina möchte ich für ihre Geduld und ihre vielfältige Unterstützung in dieser Zeit besonders danken. Abschließend möchte ich mit Psalm 106,1 dem danken, dem wir alle zu größtem Dank verpflichtet sind und ohne den unsere Existenz und unser Handeln hinfällig wird: Unserem Herrn Jesus Christus.

Hagen, im Dezember 1998 Andreas Falkenberg

Für Tina und Johannes

# Inhaltsverzeichnis

**Danksagung** **III**

**Inhaltsverzeichnis** **V**

**Abkürzungen un Symbole** **IX**

**Kurzfassung** **XII**

**1 Motivation und Ziel der Arbeit** **1**

**2 Einleitung** **1**

**3 Einordnung und Vergleich verschiedener Synthese-Systeme** **5**

3.1 Vorstellung existierender Synthese-Systeme ... 6
3.1.1 Das Berkeley Synthese-System ... 7
3.1.2 Das Braunschweiger Synthese-System ... 7
3.1.3 Das Synthese-System Bridge ... 7
3.1.4 Das CADDY-Synthese-System ... 7
3.1.5 Das Synthese-System CATHEDRAL ... 7
3.1.6 Das Synthese-System Chippe ... 8
3.1.7 Das CMU Synthese-System ... 8
3.1.8 Das Synthese-System Easy/ESC ... 8
3.1.9 Das Synthese-System Flamel ... 8
3.1.10 Das HAL Synthese-System ... 8
3.1.11 Das MIMOLA Synthese-System ... 8
3.1.12 Das Synthese-System Olympus ... 9
3.1.13 Das Synthese-System Spade ... 9
3.1.14 Das USC-Synthese-System ... 9
3.1.15 Der Yorktown Silicon Compiler ... 9
3.1.16 Das Rostocker Synthese-System ... 9
3.2 Klassifikation der vorgestellten Synthese-Systeme ... 10
3.3 Auswertung ... 11

**4 Die Abstraktionsebenen beim Entwurf komplexer digitaler Systeme** **12**

4.1 System-Ebene oder algorithmische Ebene ... 13
4.2 Register-Transfer Ebene ... 14
4.3 Logik- oder Gatter-Ebene ... 14
4.4 Schaltkreis- oder Transistor-Ebene ... 15
4.5 Layout-Ebene ... 15
4.6 Bedeutung der Post-Synthese Verifikation im Systementwurf ... 16
4.7 Zusammenfassung ... 17

**5 Der Ablauf der Synthese** **19**

5.1 Die Bedeutung von VHDL für die Synthese ... 22
5.2 Schedulingalgorithmen ... 23
5.3 Time-constrained Scheduling Algorithmen ... 25
5.3.1 Integer Linear Programming ... 25
5.3.2 Force Directed Scheduling ... 26
5.3.3 Iterativ Refinement Method ... 27
5.4 Ressource-constrained Scheduling-Algorithmen ... 28
5.4.1 List Based Scheduling ... 28
5.4.2 Zufallsbasierte Methoden ... 28

5.4.3 Zieltechnologie einbeziehende Methoden ... 28
5.4.4 Vor- und Nachteile ... 29
5.5 Assignment und Allocation ... 29
5.6 Variablen-Register-Assignment ... 30
5.7 Optimierungsmöglichkeiten ... 33
5.7.1 Constant Propagation ... 33
5.7.2 Tree Height reduction ... 34
5.7.3 Spekulatives Scheduling ... 35
5.7.4 Nutzung verschiedener und multifunktionaler Bauteile ... 35
5.8 Synthetisierte Architektur ... 36

**6 Die Akzeleration der Synthese durch Partitionierung 37**

**7 Zufallsbasierte Optimierungsalgorithmen 39**

7.1 Simulated Annealing ... 40
7.2 Der Threshold Algorithmus ... 41
7.3 Sintflut Algorithmus ... 41

**8 Genetische Algorithmen 42**

8.1 Einführung ... 43
8.2 Codierung und Dekodierung ... 44
8.3 Bewertung ... 44
8.4 Selektion ... 45
8.5 Genetische Operatoren ... 45
8.6 Mutation ... 45
8.7 Crossover ... 46
8.8 Gesamtablauf ... 48
8.9 Lohnt sich der Einsatz genetischer Algorithmen ? ... 49
8.10 Einsatz im Workstationcluster ... 49
8.10.1 Der genetische Algorithmus im Workstationcluster ... 49
8.10.2 Erwartete Beschleunigung durch Nutzung eines Workstationclusters ... 50

**9 Theoretische Überlegungen zum Syntheseverfahren 52**

9.1 Die synthetisierbaren Befehle ... 52
9.2 Anmerkungen zu den theoretischen Überlegungen ... 53
9.3 Der elementare Befehl ... 53
9.4 Schleifendarstellung durch Befehlsblöcke ... 60
9.4.1 Darstellung einer repeat-until Schleife ... 61
9.4.2 Darstellung einer while-Schleife ... 62
9.4.3 Schleifenoptimierung ... 63
9.5 Umsetzung von VHDL Konstrukten in das Zwischenformat ... 66
9.5.1 Zuweisungen und arithmetische/logische Befehle ... 66
9.5.2 If-Statement ... 67
9.5.3 Schleifen ... 70
9.5.4 Ein- und Ausgabebefehle ... 72
9.6 Darstellung der Bauteile ... 72
9.6.1 Einleitung ... 72
9.6.2 Pipelining ... 74
9.6.3 Definition der Bauteile ... 76
9.7 Definition der Allocation ... 78
9.8 Definition des Assignment ... 79

9.8.1 Bauteilabhängigkeit ... 80
9.8.2 Ressourcenbedarf ... 83
9.9 Definition des Scheduling ... 84
9.10 Variablen Register Assignment ... 84
9.11 Der Syntheseablauf ... 85
9.12 Scheduling ... 86
9.13 Der Datenflußgraph und die Kantenbewertung ... 87
9.13.1 Bewertung der Abhängigkeiten ... 88
9.13.2 Bewertung der Datenabhängigkeit ... 88
9.13.3 Bewertung der Antidatenabhängigkeit ... 88
9.13.4 Bewertung der Ausgabeabhängigkeit ... 89
9.13.5 Bewertung der Bauteilabhängigkeiten ... 89
9.13.6 Schleifenkonstrukte ... 90
9.14 Zusammenfassung ... 92

**10 Anwendung genetischer Algorithmen für die Synthese 93**

10.1 Allocation mit einem genetischen Algorithmus ... 93
10.2 Codierung der Allocation ... 95
10.3 Die Dekodierung ... 96
10.4 Mutation ... 96
10.5 Crossover ... 97
10.6 Bewertung ... 97
10.6.1 Bewertungsfunktion ... 98
10.6.2 Maximaler sequentieller Zeitbedarf ... 98
10.6.3 Minimaler sequentieller Zeitbedarf ... 99
10.6.4 Anzahl allocierter Bauteile ... 99
10.6.5 Ressourcenbedarf ... 99
10.6.6 Balancierte Auslastung der Bauteile ... 99
10.6.7 Erwartete Ausführungszeit der Befehle ... 101
10.7 Selektionsfunktion für Allocation ... 101
10.8 Abbruchbedingung ... 102
10.9 Gesamtablauf der Allocation ... 102
10.10 Assignment und Scheduling mit genetischem Algorithmus ... 103
10.11 Die optimale Wahl der Ausführungsreihenfolge ist NP-vollständig ... 103
10.11.1 Einleitung ... 103
10.11.2 Beweis ... 104
10.12 Grundgerüst des genetischen Algorithmus für das Assignment ... 106
10.13 Codierung ... 108
10.14 Dekodierung ... 108
10.15 Mutation ... 109
10.16 Crossover ... 109
10.17 Zusätzliche Mutationsmöglichkeit ... 110
10.18 Die Bewertung ... 110
10.18.1 Anzahl Taktschritte ... 111
10.18.2 Bauteilbedarf ... 112
10.18.3 Ressourcenbedarf ... 112
10.18.4 Registeranzahl ... 112
10.18.5 Multiplexerbedarf ... 112
10.18.6 Verdrahtungsaufwand ... 113
10.18.7 Energiebedarf ... 114

10.18.8 Die Gesamtbewertung des Assignments ... 114
10.19 Selektion der Assignments ... 114
10.20 Abbruchbedingung ... 115
10.21 Der Gesamtablauf des genetischen Algorithmus für Assignment ... 115
10.22 Einsatz zusätzlicher heuristischer Methoden ... 116

**11 Wiederverwertbarkeit von optimierten Designs 116**

11.1 Einleitung ... 116
11.2 Synthese einer ALU ... 117
11.2.1 Die erzeugte Architektur ... 118
11.2.2 Vereinigung der Bauteilmengen ... 118
11.2.3 Zusammenfassung der Assignments ... 120
11.2.4 Gemeinsames Scheduling ... 121
11.2.5 Register Assignment ... 121

**12 Implementierung 122**

12.1 Befehl und Programm ... 122
12.2 Bauteil und Bauteilbibliothek ... 122
12.3 Allocation, Allocation_Code und Allocation_Generation ... 122
12.4 Schleife und Schleifen-Liste ... 123
12.5 Assignment und Assignment-Liste ... 123
12.6 Lösung ... 123
12.7 Variable und Variablen Liste ... 123
12.8 Eine Generation Lösungen ... 123
12.9 Netzliste, Kontroller und Gesamt_Netz ... 124
12.10 Die allgemein verwendbaren Klassen ... 124
12.11 Spezielle Klassen des genetischen Algorithmus ... 124
12.12 Der Gesamtprozeß ... 124
12.13 Parallelisierung durch Nutzung eines Workstationclusters ... 126

**13 Auswertung und Messergebnisse 128**

13.1 Das HAL-Beispiel ... 128
13.1.1 Vergleich verschiedener Syntheseansätze ... 129
13.1.2 Variablen Register Assignment ... 131
13.2 Weitere Messergebnisse ... 133
13.2.1 Die Darstellung des Lösungsraums ... 135
13.2.2 Optimierung durch genetischen Allocationsalgorithmus ... 140
13.2.3 Akzeleration der Synthese durch Abschätzung der Allocation ... 142
13.2.4 Akzeleration der Synthese durch Parallelisierung ... 145
13.2.5 Vergleich mit anderen Algorithmen ... 147

**14 Zusammenfassung und Fazit 151**

**Anhang A : Ablauf der Synthese anhand eines Beispiels 152**

**Anhang B : Der Quelltext des Synthesesystems 165**

**Literatur 256**

## Abkürzungen und Symbole

| | |
|---|---|
| A | Assignment |
| aa | Ausgabeabhängigkeit |
| $a_{ba}$ | Berechnet Bauteilmenge für Assignment |
| AC | Allocation |
| ada | Antidatenabhängigkeit |
| AFAP | As fast as possible |
| AL | Menge gültiger Bauteilmengen |
| ALAP | As late as possible |
| ALP | Allocationsprozeß |
| ALU | Arithmetical-Logical-Unit |
| anf | Erstes Auftreten einer Variablen |
| ASAP | As soon as possible |
| ASIC | Application Specific Integrated Circuit |
| ASP | Assignmentprozeß |
| B | Menge elementarer Befehle |
| BA | Menge allocierter Bauteile |
| bau | Liefert Typ einer Instanz eines Bauteils |
| $bel_{ba,j}$ | Belegungswert eines Bauteils |
| BiCMOS | Bipolar / Complementary Metall Oxide Semiconductor |
| BLIB | Bauteilbibliothek |
| BP | Bewertungsprozeß |
| Bres | Ressourcenbedarf eines Programms |
| Bt | Bewertungsfunktion für Scheduling |
| C | Codierungsfunktion |
| $C_{al}$ | Codierungsfunktion für Allocation |
| $C_{as}$ | Codierungsfunktion für Assignment |
| CMOS | Complementary Metall Oxide Semiconductor |
| cr | Crossoveroperator |
| $cr_{al}$ | Crossoveroperator für Allocation |
| $cr_{as}$ | Crossoveroperator für Assignment |
| $\delta_f$ | Anzahl der Operationen eines Bauteils |
| $\delta_{in}$ | Eingangskardinalität |
| $\delta_{out}$ | Ausgangskardinalität |
| D | Dekodierungsfunktion |
| $D_{al}$ | Dekodierungsfunktion für Allocation |
| $D_{as}$ | Dekodierungsfunktion für Assignment |
| da | Datenabhängigkeit |
| DRC | Design Rule Check |
| E | Erwartungswert |
| end | Letztes Auftreten einer Variablen |
| f | Funktion liefert Operationen eines Bauteils |
| FDS | Force Directed Scheduling |
| FPGA | Field Programmable Gate Array |
| G | Generation von Lösungen |
| $g_i$ | Gütekriterium |
| $g_{al,i}$ | Gütekriterium für Allocation |
| $g_{as,i}$ | Gütekriterium für Assignment |
| gen | Elementarer Schritt eines genetischen Algorithmus |

| | |
|---|---|
| $ges_w$ | Gesamtbewertung |
| gG | Bewertungsfunktion einer ganzen Generation |
| gO | Genetischer Operator |
| HDL | Hardware Description Language |
| HW | Hardware |
| ILP | Integer Linear Programing |
| id | Identifizierung |
| in | Funktion liefert Eingangsvariablen eines Befehls |
| IRM | Iterative Refinement Method |
| k | Bewertungsfunktion für Abhängigkeiten |
| L | Menge Lösungen von Assignmentproblem |
| LBS | List Based Scheduling |
| m | Mutationsoperator |
| $m_{al}$ | Mutationsoperator für Allocation |
| $m_{as}$ | Mutationsoperator für Assignment |
| $m_u$ | Weiterer Mutationsoperator für Assignment |
| minv | Minimale Anzahl benötigter Register |
| MOS | Metall Oxide Semiconductor |
| N | Menge natürlicher Zahlen |
| NP | Nichtdeterministisch Polynomiell |
| O(f) | Komplexitätsmaß |
| OP | Menge elementarer Operationen |
| out | Funktion liefert Ausgangsvariablen eines Befehls |
| P | Polynomiell |
| p | Abarbeitungsreihenfolge der Befehle |
| PE | Menge der Abarbeitungsreihenfolgen |
| PR | Programm |
| QAP | Quadratic Assignment Problem |
| r | Ressourcenbedarf |
| RAM | Random Access Memory |
| reg | Liefert Register für eine Variable |
| ROM | Read only Memory |
| RT | Register-Transfer |
| RTL | Register-Transfer Level |
| $\Sigma$ | Ein Alphabet |
| $\Sigma_2$ | Eine Komplexitätsklasse der polynomiellen Hierarchie |
| S | Befehlsblock |
| sch | Schedulingfunktion |
| $S_{nach}$ | Menge der Befehle nach Befehlsblock |
| $S_{vor}$ | Menge der Befehle vor Befehlsblock |
| SA | Simulated Annealing |
| sab | Selbstabhängigkeit |
| Sel | Selektionsfunktion |
| S-LBS | Static List Based Scheduling |
| $sp_{gen}$ | Speedup |
| SW | Software |
| $T_1$ | Gesamtzeitbedarf einer Operation auf einem Bauteil |
| $T_2$ | Zeit bis die erste Pipelinestufe wieder frei ist |
| $T_3$ | Zeit bis die Daten an den Eingängen übernommen wurden |
| $t_{com}$ | Zeitbedarf der Kommunikation eines genetischen Algorithmus |

| | |
|---|---|
| $terw_{b,i}$ | Erwartete Ausführungszeit |
| $t_{eq}$ | Zeitbedarf für die Bewertung einer Lösung |
| $t_{main}$ | Zeitbedarf des Hauptprozesses eines genetischen Algorithmus |
| $t_{par}$ | Paralleler Zeitbedarf eines genetischen Algorithmus |
| $T_R$ | Transitive Hülle von R |
| ts | Zeitbedarf eines Taktschrittes |
| $t_{seq}$ | Sequentieller Zeitbedarf eines genetischen Algorithmus |
| V | Variablenmenge |
| ve | Verteilungsfunktion |
| VHDL | Very High Speed Cicuit Hardware Description Language |
| VLSI | Very Low Scale Integration |
| WS | Workstation |
| $<<$ | Allgemeine Abhängigkeit |
| $<_P$ | Bauteilabhängigkeit |
| $<_{PR}$ | Ordnung eines Programmes |
| $<_S$ | Befehlsblockabhängigkeit |
| $\cong$ | Äquivalenzrelation auf Befehlen |
| $\equiv$ | Relation auf Variablen |

## Kurzfassung

Die Entwicklung von hochintegrierten Elektronikbausteinen wie Mikroprozessoren, digitalen Signalprozessoren oder ASICs ist durch die steigende Komplexität mit immer höherem Aufwand verbunden. Konnten die ersten Prozessoren, wie z.B. der INTEL 4004, noch von einem Entwickler in einigen Monaten entwickelt werden, so benötigt die Entwicklung heutiger Prozessoren hunderte von Mannjahren. Diesem Umstand entsprechend müssen immer leistungsfähigere Werkzeuge zur Unterstützung der Entwickler angeboten werden. Es handelt sich im wesentlichen um Softwarepakete, die ihrerseits einen großen Bedarf an Rechenleistung fordern, um sinnvoll eingesetzt werden zu können. Im Rahmen der vorliegenden Arbeit wird der Ablauf der Hardwareentwicklung ausführlich dargestellt. Die Möglichkeiten für den Einsatz von Werkzeugen zur Beschleunigung der Entwicklung und Optimierung der Entwicklungsergebnisse wird betrachtet und verglichen. Der Bereich der High-Level-Synthese wird für die näheren Untersuchungen und eigenen Softwareentwicklungen herausgegriffen. Das heißt, eine algorithmische Darstellung eines Hardwaresystems, z.B. in 'behavioural' VHDL, wird in eine Darstellung der Register-Transfer-Ebene (RTL) umgesetzt und liegt nach der Synthese z.B. als 'structural' VHDL vor. In dieser Arbeit werden die Probleme der derzeitig vorhandenen Werkzeuge aufgezeigt und eigene Lösungsvorschläge entwickelt. Hierbei handelt es sich zum einen um eine intensive Betrachtung des theoretischen Fundamentes der Syntheseprobleme. Des weiteren wird der Einsatz genetischer Algorithmen für die parallele Synthese auf Standardworkstations untersucht. Damit wird bei gleichzeitiger Beschleunigung der Berechnungen eine Optimierung der Ergebnisse erzielt. In dieser Arbeit werden die benötigten Algorithmen sowie der theoretische Hintergrund entwickelt und dargestellt. Durch die Nutzung der vorgestellten Methoden lassen sich sowohl eine Optimierung der Ergebnisse als auch eine Beschleunigung der Synthese erreichen.

# 1 Motivation und Ziel der Arbeit

Die Mikroelektronik hat in immer mehr Bereichen des täglichen Lebens Einzug gehalten. Digitale Schaltungen werden in den verschiedensten Anwendungsbereichen, wie z.B. der Telekommunikationstechnik oder der Automobiltechnik, eingesetzt. Durch hohe Stückzahlen lassen sich integrierte Schaltungen kostengünstig herstellen, wobei neben den reinen Herstellungskosten auch, gerade bei kleinen Stückzahlen, die Entwicklungskosten einen wesentlichen Faktor darstellen. Durch die rasante Entwicklung im Mikroelektronikbereich und den harten Konkurrenzkampf ist aber auch der Zeitpunkt der Markteinführung einer integrierten Schaltung ein wesentlicher Kostenfaktor, so daß eine schnelle Entwicklung einer Schaltung angestrebt wird. Durch die immer komplexeren Schaltungen nehmen der Entwicklungsaufwand und damit die Kosten zu. Da derartig komplexe Systeme ohne technische Hilfe gar nicht beherrschbar sind (ein moderner Prozessor hat über 10 Millionen Transistoren), ist es nötig, entsprechende Werkzeuge für die verschiedenen Ebenen des Schaltungsentwurfs, wie sie in Kapitel 4 dargestellt sind, und die Verifikation zu entwickeln. Die Entwurfswerkzeuge müssen dabei Ergebnisse liefern, die den Bedingungen, welche an die Geschwindigkeit und den Platzbedarf einer Schaltung gestellt werden, genügen. Andererseits muß der Ressourcenbedarf, den Entwurfswerkzeuge für einen Entwurf benötigen, in einem ertragbaren Rahmen bleiben. Die Verifikation von Entwürfen gegenüber den gegebenen Spezifikationen geschieht meist durch Simulation. Die Anforderung an einen Simulator ist einerseits die Genauigkeit der Ergebnisse, andrerseits ein niedriger Ressourcenbedarf. Synthesewerkzeuge unterstützen den automatischen Entwurf von Hardware. Wesentliche Methoden und Problemstellungen der Synthese und eine Auswahl aktueller Forschungsergebnisse werden in Kapitel 5 vorgestellt. Es gibt bisher nur unzureichend formale mathematische Definitionen der Probleme, die in der High-Level Synthese auftreten. Das Ziel dieser Arbeit besteht darin, sowohl eine theoretische Betrachtung der Probleme der High-Level Synthese, als auch die Ausarbeitung praktischer Lösungsansätze und deren Verifikation darzustellen. Des weiteren wird speziell die Anwendung genetischer Algorithmen zur Optimierung der Syntheseergebnisse forciert. Die Probleme der High-Level Synthese werden in Kapitel 9 formalisiert. In Kapitel 10 werden die formalen Ergebnisse aus Kapitel 9 auf den Bereich der genetischen Algorithmen, zu denen Kapitel 7 und Kapitel 8 eine Einleitung liefern, angewandt. Es werden Funktionen zur Bewertung formuliert, die einen Vergleich verschiedener synthetisierter Ergebnisse erlauben. Die Beschleunigung des Synthesevorgangs durch Parallelisierung der genetischen Algorithmen in einem Workstationcluster ist ein weiteres Ergebnis dieser Arbeit. Wesentliche Verbesserungen gegenüber anderen Syntheseverfahren bildet die explizite und intensive Betrachtung der Allocation, wie sie in Abschnitt 10.1 dargestellt wird. Die Ergebnisse des formalen Teils der Arbeit sind so allgemein gültig, daß sie als Grundlage weiterer praktischer und theoretischer Forschung dienen können. Im besonderen können die in Kapitel 10 formulierten Bewertungsfunktionen als allgemeine Grundlage zur Bewertung von beliebigen Syntheseergebnissen dienen. Die durchaus sinnvolle Verknüpfung von genetischen Algorithmen mit heuristischen Verfahren wird in dieser Arbeit nicht ausführlich betrachtet, da es den Rahmen der Arbeit bei weitem sprengen würde. In Abschnitt 10.22 wird dennoch kurz auf diese Möglichkeit eingegangen und in einigen Messungen in Kapitel 13 ebenfalls berücksichtigt. Teile der Arbeit [22] [40] [41] [42] [43] [44] [45] [46] sind bereits als Konferenzbeiträge veröffentlicht.

# 2 Einleitung

Die immer schneller wachsenden Anforderungen an elektronische Schaltungen, speziell Mikroprozessoren, erfordern eine Steigerung der Leistungsfähigkeit von Entwurfswerkzeugen [47], die dadurch ihrerseits wieder einer hohen Rechnerleistung bedürfen. Durch den sich hierdurch

ergebenden Kreislauf müssen neue Techniken entworfen werden, die die Akzeleration des Entwurfsprozesses, aber auch die Steigerung der Leistungsmerkmale der entworfenen Bauteile, Schaltungen und komplexen Systeme gewährleisten. Selbstverständlich ist es nicht nur der Selbstzweck, wie er in Bild 1 dargestellt ist, der die Forschung im Bereich der Mikroelektronik motiviert.

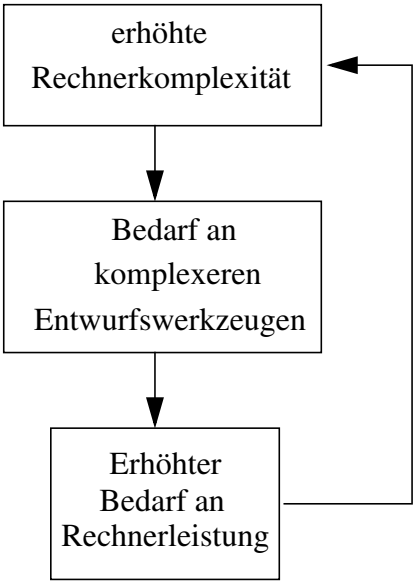



Bild 1: Selbstzweck des Rechnerentwurfs

Im Rahmen dieser Arbeit soll speziell auf den automatischen Entwurf digitalelektronischer Systeme eingegangen werden, welcher auch durch die Begriffe Synthese von Hardwarestrukturen [4][85] und Hardware/Software-Codesign [58] abgegrenzt werden kann. Es wird im speziellen der Bereich der Synthese betrachtet, der sich wiederum in High-Level und Low-Level Synthese unterteilen läßt. Die Low-Level Synthese ist der automatische Entwurf in den niedrigen Abstraktionsebenen, beispielsweise werden hier Werkzeuge zur automatischen Plazierung und Verdrahtung der Bauteile auf dem Chip betrachtet. Die Akzeleration der High-Level Synthese, also des automatischen Entwurfs in den hohen Abstraktionsebenen, wird in sofern betrachtet, als daß für die meisten Optimierungsaufgaben die einfach parallelisierbaren genetischen [57] oder evolutionären Algorithmen [83] genutzt werden, wobei hier einige spezielle Anpassungen für die Nutzung eines Workstation-Clusters [106] gemacht wurden. Die verschiedenen Abstraktions- oder Entwurfsebenen der Synthese werden ausführlich in Kapitel 4 dargestellt. Die Steigerung der Güte der Syntheseergebnisse wird durch verschiedene Optimierungsschritte erreicht, welche ebenfalls durch die Verwendung von genetischen Algorithmen implementiert und parallelisiert werden können. In Bild 2 wird der Syntheseprozeß in den übergeordneten Prozeß des Hardware/Software Codesign eingeordnet.

Eine Systembeschreibung, üblicherweise in einer Hardwarebeschreibungssprache wie VHDL [8] [28] [9] [92] [66] [99] oder Hardware-C, wird in Teile partitioniert, die entweder als Hardware oder als Software implementiert werden können. Dazu wird durch einen Precompiler jede Partition der Beschreibung sowohl als Software als auch als Hardwarebeschreibung zur Verfügung gestellt. Ein HW/SW Co-Design System [13] [54] [35] [48] stellt Werkzeuge zur Verfügung, mit denen die Auswahl der Teile welche als Hardware bzw. als Software implementiert werden, optimiert werden kann. Dabei wird eine möglichst hohe Beschleunigung des Gesamtsystems erwartet, wobei andrerseits der Hardwareaufwand unter einem zu definierenden Level bleiben soll. Die Synthese von Hardwarestrukturen läßt sich sehr einfach anhand der Bild 3 darstellen, welche sowohl die Aspekte der Synthese als auch der Analyse betrachtet. Das Y-Modell

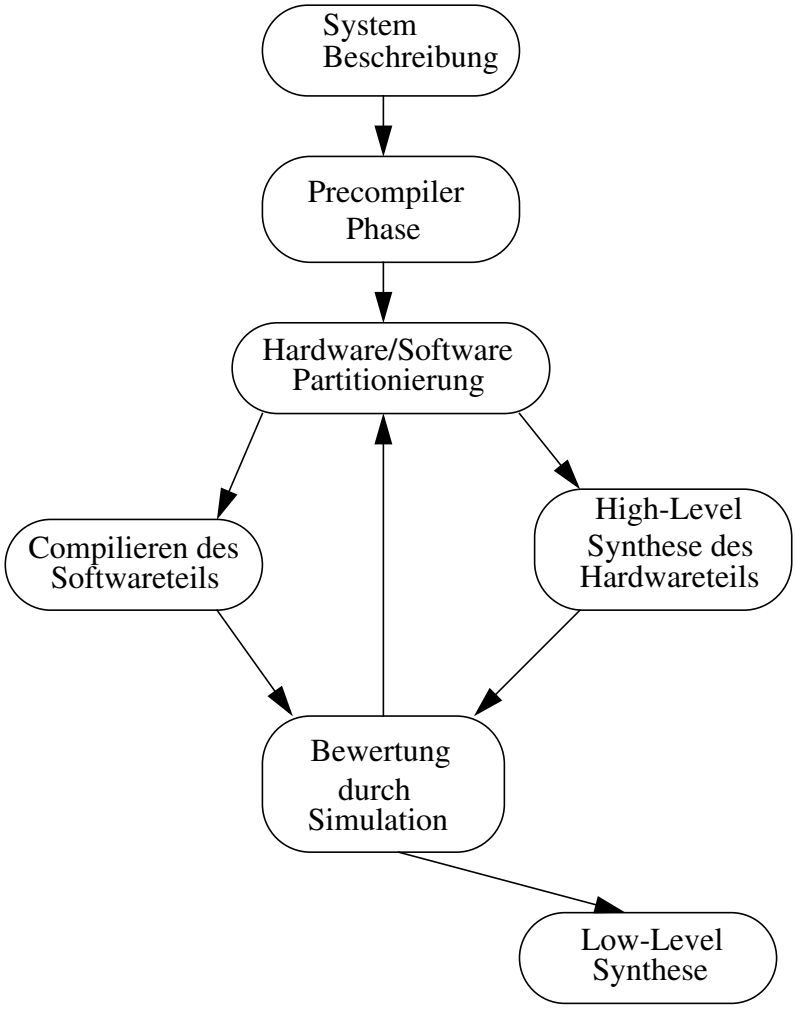



Bild 2: Allgemeiner Aufbau eines HW/SW Co-Design Systems

[50] veranschaulicht die unterschiedlichen Beschreibungsdomänen und -ebenen mit ihren Modellelementen.

Synthese bedeutet in Bild 3, einen Weg zum Mittelpunkt hin zu finden. Die Beschreibung der Hardwarestrukturen ist abstrakter je weiter die Beschreibungsebene vom Mittelpunkt entfernt dargestellt ist. Dabei wird in der Literatur zwischen High-Level und Low-Level Synthese unterschieden. High-Level Synthese ist dabei der Bereich der Synthese, der einen automatischen Übergang von der System-Ebene auf die Register-Transfer-Ebene (RTL) durchführt [50]. Die Synthese innerhalb der weiter unten/innen liegenden Schichten wird dann mit Low-Level Synthese bezeichnet. Im weiteren Verlauf wird es im wesentlichen um die Synthese von der Verhaltensbeschreibung in der System-Ebene auf die Strukturbeschreibung in der Logik-Ebene gehen. Wenn also im folgenden von Synthese die Rede ist, ist genau das, was in Bild 3 durch den Pfeil angedeutet ist, gemeint.

Ein Hilfsmittel zur Beschreibung von Systemen bieten Hardwarebeschreibungssprachen (Hardware Description Language HDL), die auf unterschiedlichen Abstraktionsebenen arbeiten. Im Rahmen der Arbeit soll im wesentlichen von einer speziellen Harwarebeschreibungssprache abstrahiert werden, da die vorgestellten Methoden davon unabhängig sind. Die Hardwarebeschreibungsprache VHDL hat zumindest in Europa die höchste Verbreitung gefunden hat. VHDL bietet schon in der Sprachdefinition die Möglichkeit, eine Schaltung auf verschiedenen Abstraktionsebenen zu beschreiben. Dabei ist zum einen die Verhaltensbeschreibung zu nen-

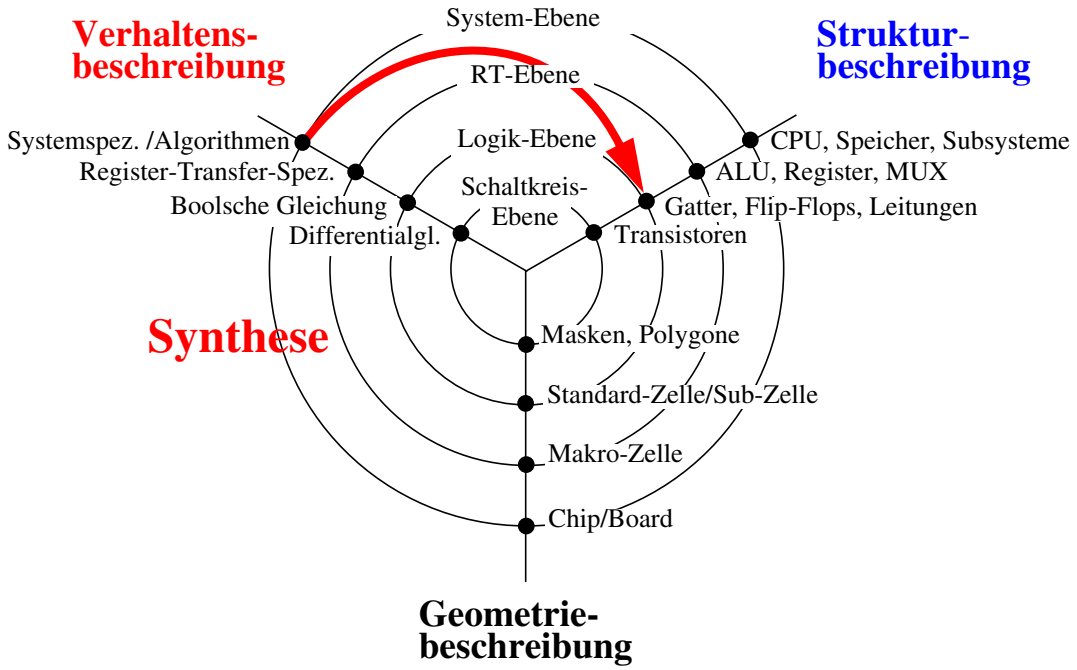



Bild 3: Das Y-Modell

nen. Hier kann das Verhalten einer Schaltung in Form eines Algorithmus beschrieben werden. Durch diese Beschreibung wird also noch keine Aussage über die Struktur der Hardware gemacht. Eine konkrete Strukturbeschreibung ist durch VHDL auch möglich. Die Strukturbeschreibung einer Schaltung kann als direkte Abbildung der Netzliste betrachtet werden und wird auch dementsprechend verarbeitet. Durch die Möglichkeit, eine Schaltung mit VHDL in verschiedenen Abstraktionsebenen zu beschreiben, ergeben sich für den Entwurf bessere Möglichkeiten. Das Gesamtsystem kann schon durch eine Simulation getestet werden, ohne daß jedes Bauteil in seiner Struktur bekannt ist [65]. Die Synthese, die hier betrachtet wird, kann somit als Umsetzung einer algorithmischen Beschreibung in eine strukturelle Beschreibung betrachtet werden. Die High-Level Synthese wird üblicherweise [75] in drei Teilschritte unterteilt, die in unterschiedlichen Ansätzen behandelt, implementiert und zusammengefaßt werden. Die Berechnung einer Bearbeitungsreihenfolge - im folgenden Scheduling genannt - bezieht sich auf die verschiedenen Befehle, die anhand einer Verhaltensbeschreibung ausgeführt werden müssen. Durch die Analyse der Datenabhängigkeiten kann eine Bearbeitungsreihenfolge gefunden werden, die eine möglichst hohe Parallelität der Berechnung gewährleistet. Im Rahmen dieser Arbeit wird davon ausgegangen, daß es sich auf der Systemebene um eine rein sequentielle Beschreibung der Algorithmen handelt, obwohl VHDL auch die Möglichkeit vorsieht Befehle parallel auszuführen. Die sequentielle Programmbeschreibung entspricht eher der Denkweise eines Programmierers und das langfristige Ziel eines jeden Synthesewerkzeugs sollte sein, einem Programmierer, welcher keine vertieften Kenntnisse im Hardwareentwurf und der parallelen Programmierung hat, Werkzeuge an die Hand zu geben, die den Hardwareentwurf und die Parallelisierung automatisieren. In Abschnitt 5.1 wird genauer auf die Merkmale von VHDL und deren Anwendungsbereiche im Systementwurf eingegangen. Als weitere Bedingung für den Entwurf einer den gegebenen Randbedingungen genügenden Schaltung müssen die zur Verfügung stehenden Bauteile betrachtet werden. Einerseits muß eine sinnvolle Auswahl an Schaltungskomponenten aus einer gegebenen Bibliothek getroffen werden (Allocation), zum anderen müssen die durch die Verhaltensbeschreibung spezifizierten Anweisungen für die Ausführung sinnvoll auf die Komponenten verteilt werden (Assignment). Diese drei Schritte effizient auszuführen, ist im wesentlichen eine schwierige Aufgabe (NP-vollständig, siehe Abschnitt 10.11),

wobei sich die Ziele teilweise widersprechen, so daß sich die Verwendung von genetischen Algorithmen mit Mehrzieloptimierung anbietet. In der Arbeit werden neue Verfahren für die genannten Schritte vorgestellt und bewertet.

In Kapitel 3 wird eine repräsentative Auswahl von Synthesesystemen kurz vorgestellt und eine vergleichende Bewertung durchgeführt. Kapitel 4 stellt die verschiedenen Abstraktionsebenen des Hardwareentwurfs dar. In Kapitel 5 werden dann einige grundlegende, in der Synthese benutzte Algorithmen vorgestellt. In Kapitel 6 wird auf Möglichkeiten der Akzeleration der Synthese eingegangen. Einige zufallsbasierte Optimierungsalgorithmen werden in Kapitel 7 dargestellt, wobei genetische Algorithmen allgemein in Kapitel 8 betrachtet werden. In Kapitel 9 wird eine theoretische Betrachtung der Synthese dargestellt, wobei auch das Gesamtkonzept vorgestellt wird. Kapitel 10 geht auf den genetischen Algorithmus für das Allocations-Verfahren ein. In Abschnitt 10.10 wird dann der genetische Algorithmus für das Assignment und das Scheduling dargestellt. In Kapitel 11 wird ein Verfahren vorgestellt, welches es ermöglicht, bereits zu Befehlen zugewiesene Hardwarekomponenten mehrfach und ohne Performance-Verluste zu nutzen. In Kapitel 12 wird die Implementierung dargestellt, und das Kapitel 13 stellt noch einige Meßergebnisse dar. Die Arbeit wird mit einer Zusammenfassung in Kapitel 14 abgeschlossen.

## 3 Einordnung und Vergleich verschiedener Synthese-Systeme

Eine Klassifikation existierender Synthesesysteme, wobei hier auch experimentelle Systeme betrachtet werden, kann bezüglich der Benutzersicht durchgeführt werden als auch aus der Systemsicht. Die Klassifikation, die ein Benutzer eines Synthesesystems durchführt, basiert auf der Frage nach der synthetisierten Architektur und dem Anwendungsfeld, welches durch das System erschlossen wird. Als Anwendungsfelder lassen sich mehrere Klassen differenzieren. Zum einen muß der Bereich der digitalen Signalverarbeitung, welcher sich dadurch auszeichnet, daß die Algorithmen konstante Laufzeit *O(1)* haben, also keine datenabhängige Verarbeitung vorliegt, genannt werden. Komplexere Algorithmen, die beispielsweise in Graphikbeschleunigern angewandt werden, sind im wesentlichen datenabhängig und verlangen nach einer besonderen Berücksichtigung durch das Synthesetool. Beispielsweise muß das Synthesetool den korrekten Umgang mit dynamischen Schleifen gewährleisten. Die oft angewandte Methode zum Abrollen der Schleifen [95] reicht nicht aus, da sie nur für Schleifen mit konstanter Anzahl Durchläufen nutzbar ist. In Bild 4 ist eine Klassifikation bzgl. der Zielarchitektur dargestellt. Eine Multiplexerarchitektur läßt sich ohne Einschränkungen auf eine Busarchitektur abbilden, so daß die vorgestellten Optimierungsmethoden für beide Architekturen gültig sind. Eine ganz andere Architektur bilden die systolischen Arrays [36], die von dem Nutzer so viel Wissen über die Struktur voraussetzen, daß keine Unterscheidung zwischen interner und externer Sicht gemacht werden kann. Auf systolische Arrays wird im Rahmen dieser Arbeit nicht weiter eingegangen.

Aus der Systemsicht muß eine Unterscheidung bezüglich der benutzten Algorithmen und den erlaubten Operationseinheiten oder Bauteilen gemacht werden. Operationseinheiten implementieren die geforderten Operationen. Die Frage, die sich stellt ist, ob arithmetische Einheiten, die mehrere Operationen implementieren, genutzt werden können. Diese Frage läßt sich bezüglich Pipelineeinheiten weiter differenzieren. Das heißt, es werden Operationseinheiten unterstützt, die eine Pipelinebearbeitung der Operationen zulassen. In Bild 5 ist eine Klassifikation aus der Systemsicht dargestellt, die sich im wesentlichen auf die benutzten bzw. unterstützten Hardwarekomponenten bezieht. Durch die Einbeziehung von Pipelinebauteilen und multifunktionalen Bauteilen erhöht sich einerseits die Komplexität der Probleme, die beim Entwurf eines entsprechenden Synthesesystems entstehen, wodurch sich die Laufzeit der Algorithmen eben-

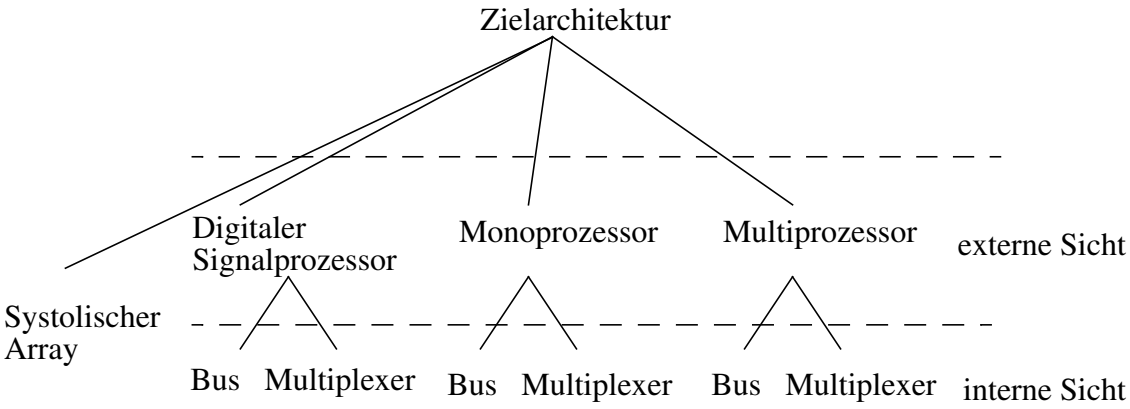



Bild 4: Klassifikation von Synthese-Systemen bezüglich der Zielarchitektur

falls erhöht, andrerseits werden aber, wie anhand des Beispiels in Abschnitt 13.1 verdeutlicht, verbesserte Ergebnisse erzielt, somit wird die Geschwindigkeit der synthetisierten Bauteile erhöht.

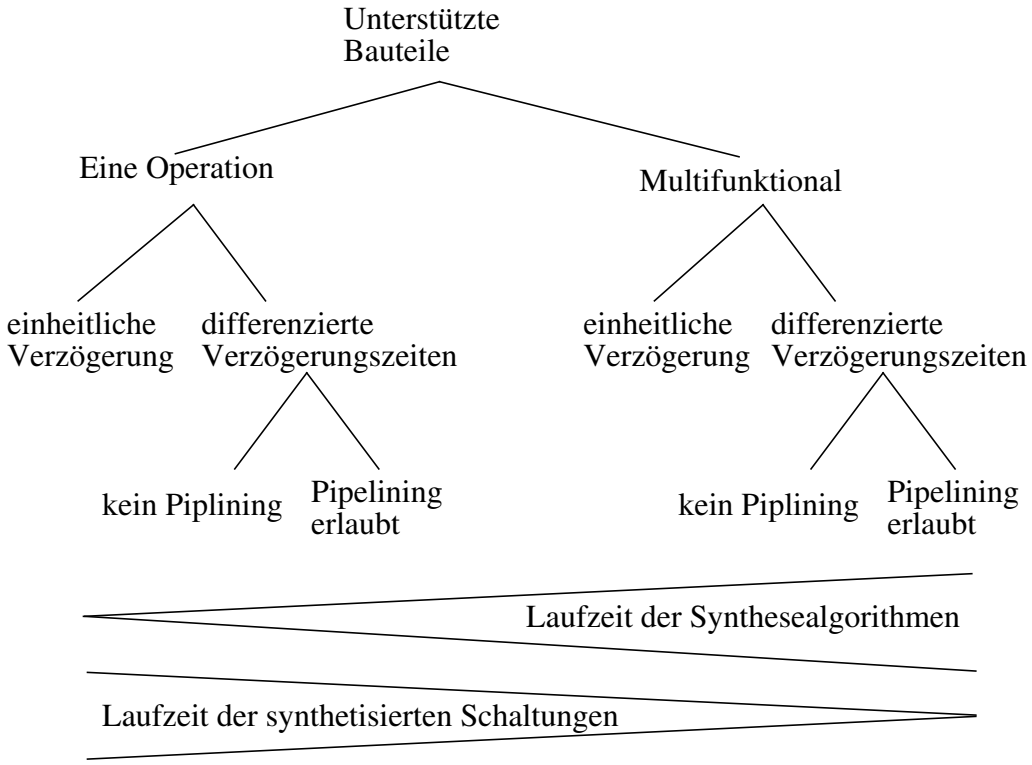



Bild 5: Klassifikation von Synthese-Systemen anhand der unterstützten Bauteile

### 3.1 Vorstellung existierender Synthese-Systeme

In der folgenden Übersicht über verschiedene Synthesesysteme werden im wesentlichen die Fragen nach der synthetisierten Architektur, den genutzten Algorithmen und den erlaubten Operationseinheiten geklärt. Die Systeme werden kurz beschrieben und im Anschluß in einer Tabelle bezüglich ihrer wesentlichen Eigenschaften verglichen. Kommerzielle Systeme werden hier nicht betrachtet, da die genutzten Algorithmen entsprechenden Forschungsprojekten entnommen sind.

### 3.1.1 Das Berkeley Synthese-System

Das an der Universität Berkeley entwickelte Synthesesystem [31] unterstützt den automatischen Entwurf von Monoprozessor-Architekturen. Die Optimierung des Scheduling, der Allokierung und des Assignments werden vollständig durch Simulated Annealing durchgeführt, was zu einer hohen sequentiellen Bearbeitungszeit führen kann. Es wird davon ausgegangen, daß eine Operation in einem Taktschritt durchgeführt wird. Dies führt dazu, daß die Taktschritte solange verzögert werden müssen, bis die langsamste Operation ausgeführt werden kann. Schnelle Bauteile können somit nicht optimal ausgenutzt werden.

### 3.1.2 Das Braunschweiger Synthese-System

Das Braunschweiger Synthese System [61], welches an der Universität Braunschweig im Rahmen des Hardware / Software Cosynthese Projektes COSYMA entwickelt wurde, ist im wesentlichen für den Entwurf von Coprozessoren optimiert. Die Eingabe, die Schleifen und datenabhängige Verzweigungen zuläßt, wird in ein Zwischenformat umgewandelt. Die Verarbeitungsgeschwindigkeit der erzeugten Schaltungen wird gegenüber anderen Systemen durch ein vorausschauendes Schedulingverfahren wesentlich gesteigert. Hierdurch entsteht aber ein erhöhter Hardwarebedarf. Der Nachteil des dargestellten Systems ist, daß es sich im wesentlichen auf die Optimierung des Schedulings beschränkt und die Allocation der geeigneten Bauteile benutzergesteuert vorgegeben werden muß.

### 3.1.3 Das Synthese-System Bridge

Das Bridge System [109] wurde im wesentlichen zur Synthese von prozessorartigen Strukturen entworfen. Nachdem die Eingabe in einen Daten- und Steuerflußgraphen umgewandelt wurde, wird das Scheduling mit Hilfe des ASAP (as soon as possible) Algorithmus durchgeführt. Dies führt dazu, daß zwar die schnellsten Schaltungen erzeugt werden können, aber eine Minimierung der Anzahl benutzter Hardwaremodule wird bei gleichbleibendem Zeitbedarf nicht durchgeführt.

### 3.1.4 Das CADDY-Synthese-System

Das an der Universität Karlsruhe entwickelte Synthese-System CADDY [27] wandelt die Eingabe in einen Datenflußgraphen um, wobei aus dem Compilerbau bekannte Transformationen auf der algorithmischen Ebene durchgeführt werden. Das Scheduling und Assignment geschehen interaktiv, weshalb das System für die Entwicklung großer Systeme nur bedingt geeignet ist bzw. einen erhöhten Arbeitseinsatz erfordert. Die Erweiterung des Systems um die Synthese paralleler Prozessorarchitekturen wird in [68] dargestellt. Das Werkzeug wurde erfolgreich in dem Siemens-Synthese-System CALLAS [16] eingesetzt. In diesem Rahmen wurde auch das System CASTOR [7] für die Synthese endlicher Automaten entwickelt.

### 3.1.5 Das Synthese-System CATHEDRAL

Die unter dem Namen CATHEDRAL entwickelten Systeme [73][84][52][103] sind im wesentlichen auf die Synthese von Schaltungen der digitalen Signalverarbeitung ausgerichtet. Es sind zwar datenabhängige Verzweigungen und Schleifen zugelassen, aber sie werden beispielsweise durch das Abrollen von Schleifen in zeitkonstante Strukturen umgewandelt. Nach der Zuordnung der elementaren Befehle zu Operationseinheiten wird ein Scheduling mit Hilfe eines ILP (Integer Linear Programming) Verfahrens, oder bei größeren Beschreibungen durch ein heuristisches Verfahren, durchgeführt.

### 3.1.6 Das Synthese-System Chippe

Das System Chippe [87][88] ist im wesentlichen für die Generierung von Monoprozessor Strukturen geeignet. Nachdem die Eingabe in einen Daten- und einen Steuerflußgraphen umgewandelt wurde, wird ein Scheduling nach einem LBS-Verfahren (List based Scheduling) durchgeführt. Mit Hilfe eines Branch and Bound Algorithmus wird das Assignment durchgeführt. Die exponentielle Laufzeit wird durch eine Greedy-Heuristik beschränkt, wodurch aber suboptimale Ergebnisse hingenommen werden müssen. Die verwendeten heuristischen und erschöpfenden Algorithmen lassen keine Bewertung der Ergebnisse bzgl. der Optimalität zu.

### 3.1.7 Das CMU Synthese-System

Das CMU Synthese System [110] besteht aus mehreren Werkzeugen, unter anderem einem Expertensystem für die Synthese. Nachdem die Eingabe in einen Daten- und Steuerflußgraphen umgewandelt wurde, wird das Scheduling der Basisblöcke durch einen Tiefensuche-Algorithmus durchgeführt. Die Befehle werden mit einem LBS-Verfahren in Kontrollschritte eingeordnet. Beim Assignment werden Schranken für die maximale Anzahl an Operationseinheiten und Zeitbedingungen mit berücksichtigt.

### 3.1.8 Das Synthese-System Easy/ESC

Das Easy/ESC System [91][100] der Universität Eindhoven ist für den Entwurf von Monoprozessorarchitekturen ausgelegt. Die Eingabe wird in einen Daten- und Steuerflußgraphen umgewandelt. Das Scheduling wird mit einem FDS (Force directed Scheduling) Verfahren durchgeführt. Die Zuweisung der Variablen zu Registern basiert auf Graphfärbe Algorithmen, wobei das Assignment der Befehle zu Operationseinheiten mit einem Simulated Annealing Verfahren durchgeführt wird.

### 3.1.9 Das Synthese-System Flamel

Die Eingabe des Synthese-Systems Flamel [108] wird in einen gerichteten azyklischen Graphen umgewandelt, wie dies auch in dem hier vorgestellten System geschieht. Im wesentlichen werden Transformationen auf der Verhaltensebene zur Optimierung durchgeführt. Schleifen werden abgerollt. Das Scheduling basiert auf dem ASAP Algorithmus. Nachdem das Scheduling durchgeführt wurde, werden die Operationen der elementaren Befehle den Operationseinheiten und die Variablen den Registern zugewiesen. Gleichartige Ressourcen werden in einem Allocationsschritt zusammengelegt.

### 3.1.10 Das HAL Synthese-System

Das HAL-System [90][91] ist für die Synthese von Monoprozessorarchitekturen ausgelegt. Die Eingabe wird in einen Graphen, der sowohl als Kontroll- als auch als Datenflußgraph fungiert, umgewandelt. Die elementaren Befehle werden dann mit Hilfe eines FDS (Force directed Scheduling) Verfahrens in Taktzyklen eingeordnet. Das Verfahren geht davon aus, daß eine Operation nur genau von einem Modultyp ausgeführt werden kann, es dürfen also keine zwei verschiedenen Operationseinheiten in der Lage sein, die gleiche Operation auszuführen. Der Vorteil ist jedoch, daß Operationseinheiten mit interner Fließbandverarbeitung unterstützt werden.

### 3.1.11 Das MIMOLA Synthese-System

Die Synthese einer Einprozessor-Architektur kann mit dem MIMOLA Synthese-System [74] der Universität Dortmund durchgeführt werden. Die aus der Eingabe erzeugten elementaren

Befehle werden mit einem ALAP (as late as possible) Verfahren in Kontrollschritte eingeordnet. Ein ILP (Integer Linear Programming) Verfahren führt die Zuordnung der Befehle zu Operationseinheiten durch. Es wird vorausgesetzt, daß alle Operationen in einem Taktschritt ausgeführt werden können. Diese Einschränkung wird aber in neueren Arbeiten aufgehoben.

### 3.1.12 Das Synthese-System Olympus

Das System Olympus [69] [80] wurde an der Universität Stanford entwickelt und besteht aus mehreren Modulen. Es ist für Monoprozessorarchitekturen entwickelt worden. Nachdem die Eingabe in ein Zwischenformat umgewandelt wurde, werden die elementaren Befehle den Operationseinheiten zugewiesen. Ein relatives Scheduling weist den elementaren Befehlen relative Ausführungszeitpunkte zu, wobei berücksichtigt werden muß, daß zwei Befehle, die derselben Operationseinheit zugewiesen wurden, nicht in demselben Taktschritt ausgeführt werden dürfen. Der konkrete Ausführungszeitpunkt der elementaren Befehle wird dynamisch zur Ausführungszeit festgelegt.

### 3.1.13 Das Synthese-System Spade

Das System Spade [55] ist für den Entwurf von Monoprozessorarchitekturen entwickelt, wobei Registerbänke in der Synthese benutzt werden können. Nachdem die Eingabe in einen Datenflußgraphen, bei dem auch Schleifen zugelassen sind, umgewandelt wurde, wird das Assignment mit einem Best-first-search Algorithmus vorgenommen. Danach wird ein ILP Verfahren unter Benutzung einer Dringlichkeitsheuristik für das Scheduling eingesetzt. Das System geht davon aus, daß jede Operation innerhalb eines Zweiphasentaktes ausgeführt werden kann.

### 3.1.14 Das USC-Synthese-System

Das USC-System [89][70] beinhaltet Module zur Generierung von Mono- und Multiprozessorsystemen. Nach der Generierung eines gerichteten azyklischen Graphen, welcher keine Schleifenkonstruktionen zuläßt, aus der Eingabe, wird ein FDS (Force directed Scheduling) Verfahren zur Einordnung der elementaren Befehle in Taktschritte und Minimierung der benötigten Komponenten benutzt. Es wird für die gleiche Operation nur ein Modultyp zugelassen, was dazu führt, daß keine weiteren Allokations- und Assignmentschritte durchgeführt werden. Der Vorteil ist, daß mit diesem System auch Pipelinestrukturen entworfen werden können.

### 3.1.15 Der Yorktown Silicon Compiler

Das Yorktown Synthese System [19] unterstützt im wesentlichen den Entwurf von Prozessorarchitekturen. Nachdem die Eingabe, in der Schleifen und Verzweigungen erlaubt sind, in das interne Format umgewandelt wurde, wird ein sogenanntes AFAP (as fast as possible) Scheduling Verfahren [26] durchgeführt. Bei dieser Ablaufplanung kann die Reihenfolge von elementaren Befehlen nicht verändert werden, weshalb elementare Befehle zwar parallel ausgeführt werden können, aber eine schlechte Ausnutzung der Komponenten zustande kommen kann. Die Allokierung wird interaktiv durch den Benutzer durchgeführt. Die Operationseinheiten werden nach Bedarf durch ein Logik-Synthese-Werkzeug direkt entworfen.

### 3.1.16 Das Rostocker Synthese-System

Bei dem an der Universität Rostock entworfenen und in dieser Arbeit beschriebenen experimentellen Synthese-System wird in erster Linie der Aspekt der parallelen Synthese betrachtet. Keiner der in Kapitel 5 dargestellten Algorithmen ist für eine Parallelisierung geeignet. Somit mußten an mehreren Stellen neue Ansätze geschaffen werden, um eine Synthese mit Hilfe der leicht parallelisierbaren genetischen Algorithmen durchführen zu können. Im Rahmen dieser

Arbeit wird die Synthese von Monoprozessorarchitekturen betrachtet. Dabei sind sowohl Verzweigungen als auch datenabhängige Schleifen erlaubt. Nachdem eine VHDL-Verhaltensbeschreibung in ein Zwischenformat umgewandelt wurde, wird eine Zuordnung der elementaren Befehle zu Operationseinheiten durchgeführt. Diese Zuordnung wird bewertet und durch genetische Algorithmen optimiert. Als Operationseinheiten sind Module mit mehreren Operationen zugelassen, wobei ein unterschiedlicher Zeitbedarf und Pipelinefähigkeiten berücksichtigt werden. Optimierungen, die auf Transformationen innerhalb der Verhaltensebene basieren, wie z.B. die in Kapitel 5.7.2 beschriebene Tree-Height-Reduction, werden im Rahmen dieser Arbeit nur vorgestellt, aber nicht weiter betrachtet.

### 3.2 Klassifikation der vorgestellten Synthese-Systeme

Um eine Übersicht über die Leistungsfähigkeit verschiedener Synthese-Systeme zu erhalten und einen Vergleich zu ermöglichen, sollen an dieser Stelle einige Eigenschaften von Synthese-Systemen definiert und den bisher genannten Systeme entsprechend zugeordnet werden. Diese Einteilung ist beschränkt auf die Übersetzung aus der Verhaltensbeschreibung in die Register-Transfer Beschreibung. Die folgenden Eigenschaften eines Synthese-Systems können differenziert werden:

- **Multifunktionalität** (MF): Operationseinheiten, die verschiedene Operationen ausführen können, (ALUs) werden unterstützt.
- **Multiprozessor** (MP): Der Entwurf von Multiprozessorsystemen wird unterstützt.
- **Pipeline** (P): Operationseinheiten, die eine Pipelineverarbeitung ermöglichen, werden voll unterstützt und können in Schaltungen eingesetzt werden.
- **Schleifen** (S): Die Synthese von Schleifen wird unterstützt, dabei wird unterschieden zwischen Schleifen **variabler** und **konstanter** Länge.
- **Transformationen** (TR): Transformationen werden auf der Verhaltensebene durchgeführt. Sie ermöglichen Optimierungen durch arithmetische Umformungen von Formeln.
- **Verzögerungszeitvariabilität** (VV): Operationseinheiten mit unterschiedlichem Zeitbedarf für die Ausführung der Operationen werden unterstützt.
- **Verzweigungen** (VZ): Die Synthese von datenabhängigen Verzweigungen wird unterstützt.

Weitere Eigenschaften beziehen sich auf die benutzten Algorithmen:

- **Deterministisch** (D): Der Algorithmus liefert bei gleicher Eingabe immer dasselbe Ergebnis. Es ist keine Optimierung durch längere Laufzeiten möglich.
- **Interaktiv** (I) : Das System ist nicht vollständig, und es müssen noch interaktiv Eingaben getätigt werden.
- **Probabilistisch** (PR): Der Algorithmus benutzt zufällig gewählte Parameter, wodurch eine laufzeitabhängige Optimierung möglich ist.
- **Parallel** (PA): Die Synthese kann auf mehrere parallel arbeitende Rechnereinheiten verteilt werden.

In der folgenden Tabelle werden die oben genannten Systeme anhand der Eigenschaften eingeordnet. Ein ( - ) bedeutet, daß die Eigenschaft nicht vorhanden ist, bei einem ( + ) ist sie vorhanden. Bei den Schleifen wird statt dem ( + ) entweder ein ( v ) oder ein ( k ) für variabel oder konstant eingesetzt. Das hier beschriebene Synthese-System wird in der Tabelle mit RSS (Rostock-Synthesis-System) bezeichnet.

**Tabelle 1: Eigenschaften von Synthese-Systemen**

| System | MF | MP | P | S | TR | VV | VZ | D | I | PR | PA |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Berkley | + | - | - | - | - | - | + | - | - | + | - |
| Braun | + | - | + | v | - | + | + | + | + | - | - |
| Bridge | - | - | - | k | - | - | + | + | - | - | - |
| Caddy | + | + | + | k | + | + | + | + | + | - | - |
| Cath | - | - | - | k | - | - | + | + | - | - | - |
| Chippe | - | - | - | k | - | - | + | + | - | - | - |
| CMU | - | - | - | k | + | - | + | + | + | - | - |
| Easy | - | - | - | k | - | - | + | + | - | + | - |
| Flamel | - | - | - | k | + | - | + | + | - | - | - |
| HAL | - | - | + | k | - | - | + | + | - | - | - |
| MIM | + | - | - | k | - | - | + | + | - | - | - |
| Olymp | + | - | - | k | - | - | + | + | + | - | - |
| Spade | - | - | - | v | - | - | + | + | - | - | - |
| USC | + | - | + | - | - | - | + | + | - | - | - |
| York | - | - | - | k | - | - | + | + | - | - | - |
| RSS | + | - | + | v | - | + | + | - | - | + | + |

### 3.3 Auswertung

Anhand der Tabelle 1 ist deutlich ersichtlich, daß die parallele Synthese beispielsweise in einem Workstation-Cluster bisher so gut wie gar nicht betrachtet wurde. Desweiteren unterstützen die meisten Systeme nur relativ einfache Bauteilbibliotheken. Außerdem werden oft nur Schleifen mit einer konstanten Anzahl an Durchläufen unterstützt. RSS unterstützt den Einsatz komplexer Operationseinheiten wie ALUs mit unterschiedlichen Laufzeiten der einzelnen Operationen, was zu differenzierteren und damit besseren Ergebnissen führt. Des weiteren können Pipeline-Bauteile eingesetzt werden. Außerdem sind datenabhängige Schleifen erlaubt, was bei den wenigsten Systemen der Fall ist. Viele probabilistische Algorithmen haben den Vorteil, daß sie nach sehr kurzer Laufzeit schon Ergebnisse liefern, mit denen weitergearbeitet werden kann. Optimierte Ergebnisse können dann berechnet werden, wenn vorhandene Workstations beispielsweise nachts wenig oder gar nicht benutzt werden. Das RSS bietet die Möglichkeit, sehr schnell Ergebnisse zu liefern, andrerseits aber unter Nutzung aller vorhandenen Ressourcen diese Ergebnisse parallel zu optimieren. Durch die Nutzung genetischer Algorithmen mit gewich-

teter Bewertungsfunktion lassen sich die Ergebnisse bedarfsgerecht optimieren. Sowohl Hardwareaufwand als auch Verzögerungszeiten können minimiert, bzw. ein Gleichgewicht gefunden werden. Die meisten der genannten Systeme versuchen laufzeitoptimale Ergebnisse zu liefern, wobei oft eine nur marginal langsamere Schaltung zu einer hohen Hardwareeinsparung führen kann, was bei RSS volle Berücksichtigung findet. Aus den dargestellten Eigenschaften des Systems folgt eine besonders gute Eignung für die Synthese von Coprozessoren, wie zum Beispiel Graphikbeschleuniger, die sich dadurch auszeichnen, daß die Laufzeit der Algorithmen oft sehr stark datenabhängig ist. In [93] sind sehr viele Graphik-Algorithmen dargestellt, die sehr gut zur Synthese mit dem dargestellten System geeignet erscheinen.

# 4 Die Abstraktionsebenen beim Entwurf komplexer digitaler Systeme

Wie schon in der Einleitung angedeutet wurde, wird im wesentlichen die Synthese in den oberen Abstraktionsebenen diskutiert, also die High-Level Synthese betrachtet, wobei selbstverständlich auch die anderen Ebenen erklärt werden.

Der Entwurfsprozeß komplexer digitaler Systeme kann in mehrere Abstraktionsebenen untergliedert werden [104]. Diese Ebenen wurden schon in Bild 3 dargestellt. In der oberen/äußeren Ebene, der System-Ebene, wird das zu entwerfende System algorithmisch bzw. von dem erwarteten Verhalten ausgehend beschrieben. Die Übersichtlichkeit kann gewahrt bleiben. Die untere Ebene liefert eine detaillierte Beschreibung des Systems, ist aber bei komplexen Systemen nicht mehr übersichtlich. Die unterste Ebene ist in unserem Fall die Layoutebene, die die endgültige Struktur des Layouts der Masken für den Entwurf der Chips liefert. Für jede Ebene müssen spezielle Simulatoren vorhanden sein, die eine Verifikation des Entwurfs gewährleisten. Andererseits müssen für jeden Übergang von einer Ebene in eine niedrigere Entwurfswerkzeuge, also Synthesetools, vorhanden sein, oder der Übergang muß manuell durchgeführt werden, was bei komplexen Systemen zu einem sehr hohen und damit kostenintensiven Arbeitsaufwand führen kann. Bei heutigen Mikroprozessoren kann es dabei schnell zu Arbeitszeiten kommen, die in hunderte von Mannjahren gehen. Durch die hohe Komplexität von modernen Mikroelektronikbausteinen wird dadurch der Bedarf an leistungsfähigen Werkzeugen ebenfalls gesteigert. Durch die hohe Komplexität der zu entwerfenden Systeme müssen gerade in der Planungsphase Möglichkeiten zur Verfügung stehen, die eine abstrakte Sichtweise des Systems erlauben. Der Entwurf geschieht also nicht mehr auf der Ebene des Schematic-Entry, wo manuell jedes funktionale Bauteil aus vorhandenen Bauteilen, wie z.B. Gattern, zusammengesetzt werden muß, sondern auf der Ebene der Verhaltensbeschreibung. Hier wird das Verhalten einer funktionalen Komponente durch eine Hardwarebeschreibungssprache (HDL) spezifiziert. Die konkrete Implementierung der Hardware und die Technologie brauchen dafür noch nicht bekannt zu sein. In Bild 6 werden verschiedene Entwurfsabläufe dargestellt, wobei bei dem rechten die Nutzung von Hardwarebeschreibungssprachen einbezogen wird.

Der Bereich der Synthese ist also ein wichtiger, die Arbeit des Hardware-Designers erleichternder und den Aufwand reduzierender Schritt in der Entwicklung fortschrittlicher und komplexer Hardware.

Am besten kann die Synthese anhand der Bild 7 erläutert werden. Eine auf einer hohen Ebene spezifizierte und durch Simulation verifizierte Beschreibung oder Darstellung einer Hardwarekomponente wird durch einen oder mehrere Syntheseschritte in einer niedrigeren Abstraktionsebene implementiert, wobei die in der Vorgabe spezifizierte Funktionalität erhalten bleiben muß. Die Implementierung geschieht durch eine Darstellung der gewünschten Funktion mit Hilfe von schon definierten Funktionen, wobei die benutzten Funktionen ihrerseits in einer beliebigen Abstraktionsebene vorhanden sein müssen. Oft ist es so, daß durch die Implementie-

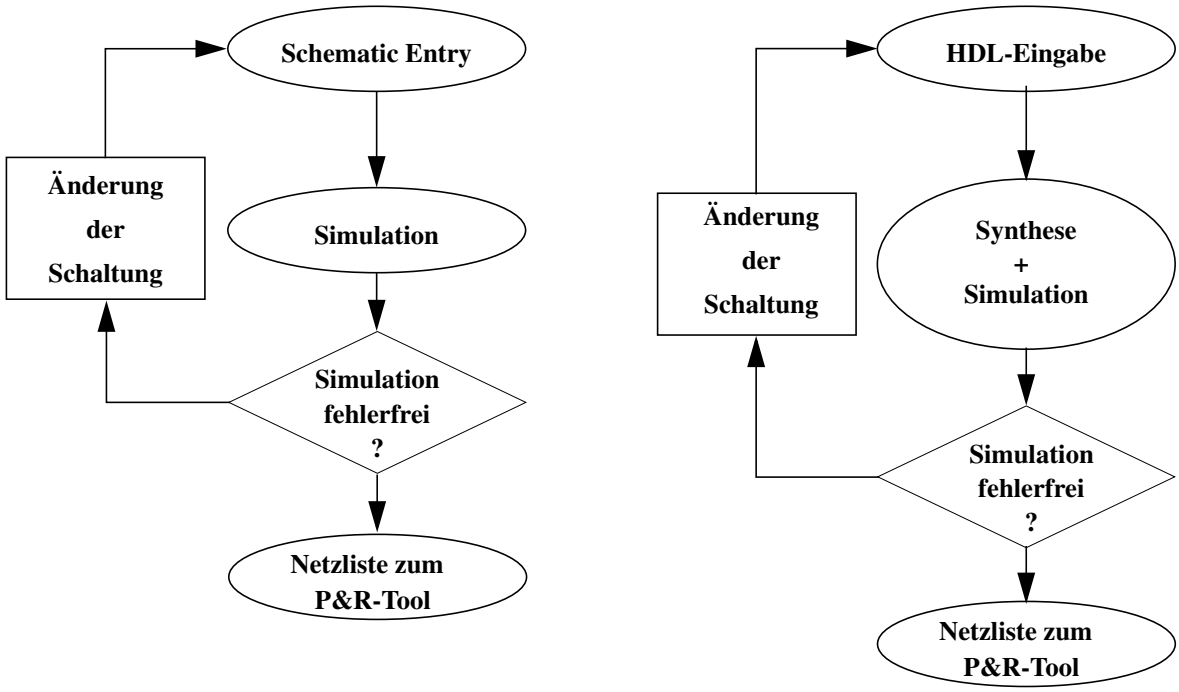



Bild 6: Verschiedene Entwurfsabläufe (design-flow)

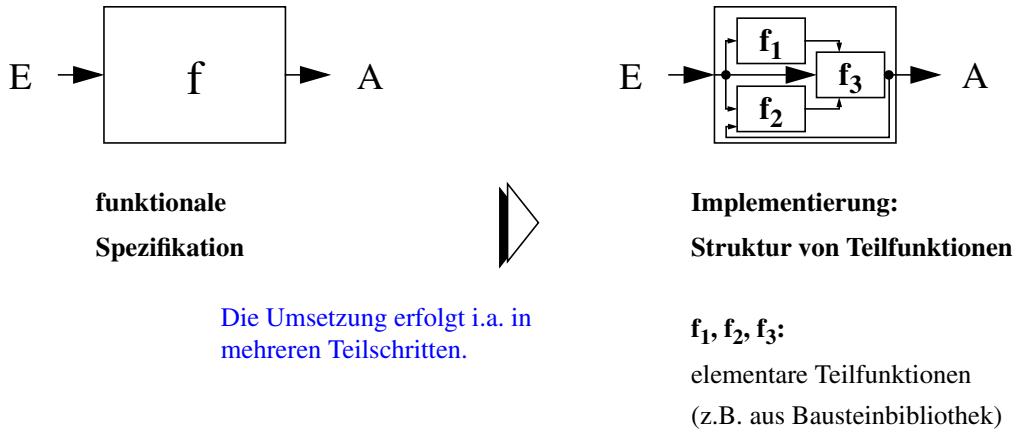



Bild 7: Die Umsetzung einer funktionalen Spezifikation in eine Implementierung

rung einer Beschreibung in einer niedrigeren Ebene keine isomorphe Abbildung entsteht. Das Wissen über die Implementierung wird erweitert und somit der Informationsgehalt erhöht. Beispielsweise können gerade bei der Beschreibung von Hardware in einer hohen Ebene wenig Aussagen über den Ressourcen- und Zeitbedarf der Schaltung gemacht werden. Diese Information wird immer präziser, je weiter das Abstraktionsniveau der Beschreibung absinkt und sich der endgültigen Implementierung nähert. Die einzelnen Abstraktionsebenen der Hardwarespezifikation und des Entwurfs werden im folgenden zuerst beschrieben. Dann wird in Abschnitt 4.6 auf die Bedeutung der Postsynthese-Verifikation für den Systementwurf eingegangen, hier im speziellen auf die Verifikation durch Simulation.

### 4.1 System-Ebene oder algorithmische Ebene

In der System-Ebene wird das Verhalten des Gesamtsystems auf einem sehr hohen Abstraktionsniveau beschrieben. Elemente, die in dieser Ebene unterschieden werden, sind im Fall des

Hardwareentwurfs Prozessoren, Speichereinheiten und Peripheriegeräte. Der Systementwurf kann in dieser Ebene unabhängig von der konkreten Implementierung in Hard- oder Software gemacht werden. Durch eine korrekte Spezifikation werden alle Bedingungen, die an ein System gestellt werden, meist umgangssprachlich in einem Pflichtenheft zusammengefaßt. Die System-Ebene gibt eine Beschreibung des Verhaltens des Systems in Form von Algorithmen auf der Basis der Spezifikation wieder. Hier kann eine Unterteilung derart erfolgen, daß Algorithmen mit einer Hochsprache oder einer maschinennahen Sprache beschrieben werden können. In dieser Ebene kann das Verhalten des beschriebenen Systems durch entsprechende Compiler oder Interpreter simuliert und damit die Korrektheit verifiziert werden. Für den Entwurf komplexer Hardware gibt es verschiedene Hardwarebeschreibungssprachen, die es ermöglichen, das Verhalten algorithmisch zu beschreiben. Sprachen wie z.B. VHDL [15] [105] oder MIMOLA [74] bieten neben speziellen Konstrukten zur Hardwarebeschreibung auch jene an, die aus normalen höheren Programmiersprachen bekannt sind. Zum Beispiel werden auf dieser Ebene die verschiedene Zahlendarstellungen unterstützt. Die auf dieser Ebene beobachtbaren Werte sind größen definierter Wertemengen, deren interne Darstellung hier noch nicht betrachtet wird. Eine Verifikation bzgl. der Spezifikation wird durch Simulationen erreicht, die bei größeren Systemen nicht unbedingt alle Möglichkeiten testen können. Dabei wird im wesentlichen die Kausalität des Systems überprüft. Das Zeitmodell ist hier nicht an ein konkretes Zeitschema gebunden, sondern es werden Ereignisse und Zustände betrachtet. Ist die Korrektheit durch die Simulation verifiziert, wird der Übergang zur nächsten Ebene z.B. mit Synthesetools vorgenommen. Früher mußte der Übergang ebenfalls manuell durchgeführt werden. An dieser Stelle ist es schon wichtig, die Zielarchitektur zu wählen. Die meisten Synthesetools haben Einschränkungen, die beachtet werden müssen: z.B. wird nicht der ganze VHDL-Sprachumfang durch Synthesetools unterstützt. Eine weitere Problematik ergibt sich durch gewisse Inkonsistenzen zwischen Simulatoren und Synthesewerkzeugen. Selbst ein synthetisierter Entwurf muß oft noch durch eine Simulation verifiziert werden.

### 4.2 Register-Transfer Ebene

Eine weitere Ebene bildet die Register-Transfer Ebene (RT-Ebene), die auch als Mikroarchitekturebene bezeichnet wird. Die Beschreibung auf dieser Ebene benutzt schon konkrete Hardwareeinheiten, wie Register, Speicher und arithmetisch-logischen Einheiten (ALU), deren Verhalten bekannt sein muß. Die Zahlendarstellungen der Verhaltensebene werden in dieser Ebene durch entsprechende Bitfolgen ersetzt, so daß hier nur noch mit Bits und Bitstrings operiert wird. Auf dieser Ebene wird die Architektur mit ihren Elementen und Verbindungen sichtbar. Um hieraus eine Schaltung zu generieren, muß eine Abbildung der Elemente der RT-Ebene auf die Elemente einer Bibliothek erfolgen, die genau das geforderte Verhalten aufweisen, bzw. es müssen entsprechende Elemente konstruiert werden. Die Abbildung von Elementen der Register-Transfer Ebene auf Bibliothekselemente kann automatisiert werden. Bibliothekselemente bestehen im wesentlichen aus Komplexgattern mit einer fest definierten Funktionalität. In dieser Ebene wird ein abstraktes Zeitmodell mit relativen Größen benutzt, z.B. ein Taktschema. Der Übergang zur Logik-Ebene wird durch die Abbildung der Register und ALUs auf die Bibliothekselemente vollzogen, die ihrerseits in der niedrigeren Ebene vorhanden sind oder dem Syntheseprozeß unterzogen werden müssen. Es ist wichtig, daß die Elemente der Register-Transfer Ebene auf Bibliothekselemente abgebildet werden können, die genau die gleiche oder eine größere Funktionalität aufweisen.

### 4.3 Logik- oder Gatter-Ebene

Die einzelnen Elemente, die auf der Register-Transfer Ebene spezifiziert werden, bestehen aus Gattern und Flip-Flops und können jeweils als Gatter-Schaltungen oder Boolsche-Gleichungen

dargestellt werden. Die Darstellung auf dieser Ebene besteht aus einem Schaltplan, dessen Elemente die verschiedenen Gatter und Flip-Flops sind. Strukturen wie Register und ALUs sind auf dieser Ebene nicht mehr sichtbar. Jedem Gatter werden Verzögerungszeiten zugeordnet, die sich an den real existierenden Verzögerungszeiten von Gattern orientieren. Mit Hilfe der Gatterverzögerungszeiten können genauere Simulationen durchgeführt werden, die eine exaktere Aussage über die zu erwartende Geschwindigkeit der Schaltung liefern, als dies in den höheren Ebenen der Fall ist. Es existieren verschiedene Simulationsmodelle, die im wesentlichen auf der Auswertung der Booleschen Funktionen in mehrwertiger Logik basieren. Für diese Ebene existieren Werkzeuge, die den Übergang zur Schaltkreisebene durchführen, indem sie den Gattern und Flip-Flops Schaltungen aus Transistoren zuordnen. Manche Tools gehen noch einen Schritt weiter, indem sie jedem Gatter und jedem Flip-Flop direkt ein Layout zuordnen, wie es im sogenannten Standardzellenentwurf realisiert ist.

### 4.4 Schaltkreis- oder Transistor-Ebene

In der Schaltkreis-Ebene wird eine Schaltung auf der Ebene der Transistoren, Widerstände, Kondensatoren und Stromquellen beschrieben. Hier werden für die Simulation die Netzwerkgleichungen herangezogen. An dieser Stelle ist die konkrete Technologie bekannt, mit der die Schaltung aufgebaut werden soll. Es muß also zwischen Bipolar, CMOS, BiCMOS und anderen Technologien unterschieden werden. Die einzelnen Bits in der Logik-Ebene werden durch Spannungspotentiale, die als reele und komplexe Zahlen dargestellt werden, ersetzt. Eine Analyse mit einem Simulator basiert im wesentlichen auf der Lösung einer Menge von Differentialgleichungen. Um den Übergang zur Layoutebene zu schaffen, wird jedem Schaltungselement das entsprechende Layout zugeordnet und auf der Chipoberfläche plaziert und verdrahtet.

### 4.5 Layout-Ebene

In der Layout-Ebene wird der physikalische bzw. geometrische Aufbau einer Schaltung auf einem Chip im wesentlichen durch Polygone dargestellt. Das fertige Layout einer Schaltung wird in einzelne Belichtungsmasken unterteilt, was vom Fertigungsprozeß abhängig ist. Jede Maske dient als Vorlage für einen Prozeßschritt. Es gibt z.B. eine Maske, die die Diffusionsgebiete auf einem Chip darstellt. Fertige Layouts können optimiert werden, so daß der Flächenbedarf möglichst klein wird. Meist kann, aufgrund der Komplexität des Layouts, keine direkte Verifikation mit einem Simulator, welcher ebenfalls im wesentlichen eine Menge von Differenzialgleichungen auswertet, durchgeführt werden. Eine Maßnahme der Verifikation ist die Schaltungsextraktion, die eine Darstellung auf der Layout-Ebene wieder in eine Darstellung auf der Schaltkreis-Ebene überführt. Die Komplexität der Simulation kann dadurch erheblich reduziert werden und somit die Korrektheit schneller getestet werden. Ein Vergleich einer extrahierten Schaltung mit der ursprünglichen ist nicht möglich, da auch parasitäre Elemente extrahiert werden. Außerdem kann das Graphisomorphieproblem auf den Vergleich zweier Schaltungen reduziert werden. Layouts, deren Korrektheit schon durch den Einsatz des gefertigten Bauteils verifiziert sind, sollen oft, auch nach einer Technologieumstellung, weiterbenutzt werden können. Um nicht den ganzen Entwurfsprozeß wiederholen zu müssen, werden Layout-Kompaktierer eingesetzt. Auch für die Beschreibung in dieser Ebene gibt es Sprachen, die von den direkten Technologieparametern abstrahieren, z.B. kann mit DINGO-XT [21] ein Bauteil auf der Layoutebene beschrieben werden; eine Technologie ist damit nicht vorgegeben. Eine wichtige Problemstellung ist eine gute Plazierung und Verdrahtung der Bauteile zu finden. Bei vielen Werkzeugen müssen Vorgaben gemacht, z.B. einige komplexere Elemente manuell vorplaziert oder die Positionen der Anschlüsse manuell gewählt werden. Somit sind hier einige Versuche nötig, um ein gutes Ergebnis zu erhalten. Bei einer Rechenzeit von 6 h pro Plazierungs- und Verdrahtungsvorgang kann trotz des Einsatzes schneller Workstations eine Zeit von einem Monat zusam-

menkommen. Werden einzelne Elemente von Hand entworfen, so müssen diese den Design-Regeln entsprechen, die bei einem automatischen Layout durch das System berücksichtigt werden. Ein Design-Rule-Checker (DRC) verifiziert nach dem Entwurf des Layouts die Einhaltung von Technologieparametern.

### 4.6 Bedeutung der Post-Synthese Verifikation im Systementwurf

Eine Frage, die sich für jedes Synthesesystem stellt, ist die nach der Zuverlässigkeit der berechneten Ergebnisse. Das Ziel, welches beim Entwurf von Synthesewerkzeugen angestrebt wird, ist die Erfüllung des Paradigmas 'correct by construction' für den Gebrauch der Werkzeuge. Dies würde implizieren, daß eine Beschreibung eines Algorithmus auf Systemebene, welcher den gewünschten Anforderungen genügt, in eine äquivalente Hardware umgewandelt werden kann, ohne daß dieses Ergebnis noch einmal verifiziert werden muß. Da dieses Ziel an verschiedenen Stellen noch nicht erreicht ist und auch nicht in greifbarer Nähe scheint, müssen für jeden Syntheseschritt Verifikationswerkzeuge zur Verfügung gestellt werden, mit deren Hilfe der ursprüngliche Algorithmus mit dem synthetisierten Ergebnis verglichen werden kann. Hierbei kann die Methode der formalen Verifikation gewählt werden, welche aber heutzutage nur bei relativ kleinen Systemen mit höchstens ein paar Dutzend Programmzeilen anwendbar ist. Diese Möglichkeit wird daher an dieser Stelle nicht weiter vertieft. Der zweite Weg ist die Post-Synthese-Simulation. Das Bauteil, welches als Modell in der algorithmischen Beschreibung vorliegt, wird mit einer wohl definierten Stimulimenge angesteuert. Das synthetisierte Bauteil, welches nun als Register-Transfer-Modell vorliegt, wird dann mit der gleichen Stimulimenge angeregt. Diese Ergebnisse müssen verglichen werden. Bild 8 macht deutlich, wie eine derartige Simulationsumgebung gestaltet werden kann.

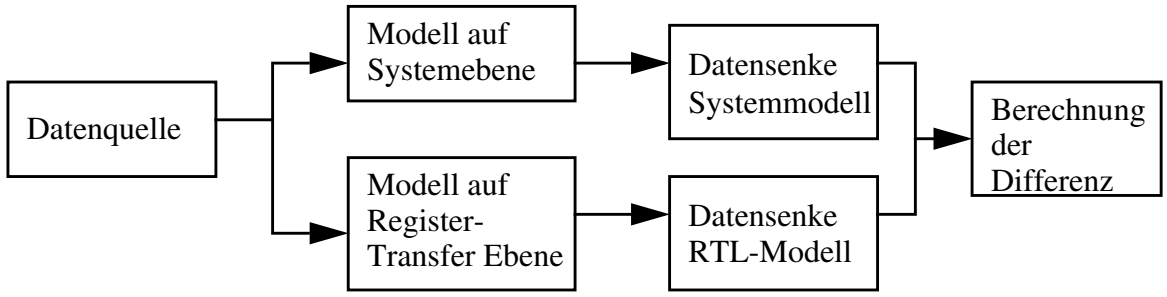



Bild 8: Umgebung für die Postsynthese Simulation

Wird für die Entwicklung der Hardware VHDL benutzt, so kann wie in [99] ausführlich dargestellt der gleiche Simulator für die verschiedenen Ebenen benutzt werden. Oft ist es aber so, daß das Verfahren der Cosimulation eingesetzt wird. Dabei werden von einem übergeordneten Werkzeug die verschiedenen Bausteine, welche auf unterschiedlichen Ebenen dargestellt sind, verwaltet. Wird nun ein spezieller Simulator benötigt, so wird dieser aufgerufen und automatisch mit den entsprechenden Parametern versehen und mit den aus der Datenquelle generierten Stimuli angesteuert. Im Optimalfall sollte in beiden Datensenken nach der Simulation das gleiche Ergebnis stehen. Der Vergleich der beiden Ergebnisse ist mit Schwierigkeiten verbunden, da in dem RTL-Modell Verzögerungszeiten mit modelliert werden, zum Beispiel in Form von Taktschritten, welche es in der algorithmischen Darstellung nicht gibt. Des weiteren ist ein direkter Vergleich nur dann möglich, wenn schon in der algorithmischen Darstellung die Wortbreiten, also die Anzahl der benötigten Bits, für die Zahlendarstellung berücksichtigt werden.

In einem normalen Systementwicklungsprozess werden aber an dieser Stelle fast optimale Annahmen gemacht und Fließkommazahlen benutzt. Daher ist eine Nachbearbeitung des Systems auf der algorithmischen Ebene nötig, um die geeignete Zahlendarstellung zu finden. Durch die unterschiedlichen Zahlendarstellungen kommen entsprechend unterschiedliche Ergebnisse bei der Simulation zustande. Hier liegt es an der Systemkenntnis und der Erfahrung des Entwicklers, Unterschiede als Fehler oder als Ungenauigkeiten zu interpretieren. Dieser Schritt kann zum Beispiel beim Entwurf digitaler Filter einen erheblichen Teil der Entwicklungszeit beanspruchen, da die Filterkoeffizienten gerade durch solche Ungenauigkeiten stark beeinflußt werden können. Eine Automatisierung dieses Schrittes sieht so aus, daß die Fließkommawerte durch Festkommawerte unterschiedlicher Wortbreiten ersetzt und die unterschiedlichen Bauteile simuliert werden. Die unterschiedlichen Simulationsergebnisse müssen nun aber manuell verglichen werden, und der Entwickler muß eine Entscheidung treffen, welche Ungenauigkeiten akzeptabel sind. Mit dem Wissen über die Wortbreiten kann erst eine Synthese im eigentlichen Sinne durchgeführt werden. Dann erst kann im Rahmen der Simulation ein Vergleich zwischen den Ergebnissen durchgeführt werden. Hierbei ist aber von dem Zeitverhalten zu abstrahieren. Die Simulationsergebnisse beider Modelle sind entsprechend dem unterschiedlichen Zeitverhalten zu interpretieren. Eigentlich kann beim algorithmischen Modell nicht von Zeitverhalten gesprochen werden, es handelt sich simulationstechnisch um Ereignisse. Des weiteren unterstützt nicht jedes Synthesetool alle in einer Beschreibungssprache wie VHDL spezifizierten Konstrukte. Daher können sich Simulation und Synthese unterschiedlich gegenüber einer gegebenen Beschreibung verhalten, bzw. es kann passieren, daß eine Beschreibung, für die ein Simulator korrekte Ergebnisse liefert, nicht durch ein Synthesetool akzeptiert wird. Dies führt ebenfalls zu einer Überarbeitung der Systembeschreibung. In Bild 9 ist die Einbettung der Postsynthese-Simulation in den gesamten Entwurfsablauf dargestellt. Anhand des in diesem Text Gesagten wird die Wichtigkeit der Simulation allgemein und speziell der Postsynthese-Simulation deutlich. Besonders wichtig in diesem Bereich ist Akzeleration der Simulation von VHDL-Modellen [65], da die Simulation einen Engpaß in der Entwicklung von Hardwaresystemen darstellt.

### 4.7 Zusammenfassung

Bei dem Entwurf eines VLSI-Systems wird eine Beschreibung des Systems auf einer niedrigeren Ebene (Implementierung) immer aus einer Beschreibung in einer höheren Abstraktionsebene (Spezifikation) gewonnen. Die Übergänge von einer hohen zu einer niedrigen Abstraktionsebene können manuell vorgenommen werden oder aber automatisch erfolgen (Synthese). Bei jedem Übergang von einer Ebene zur niedrigeren muß eine Verifikation des Verhaltens erfolgen. Simulatoren für die verschiedenen Ebenen erlauben einen Vergleich des Verhaltens einer Spezifikation mit ihrer Implementierung. Eine formale Verifikation des Verhaltens ist oft schwierig. Deshalb wird dieser Weg meist durch die Simulation von Beispielen umgangen. Die Schwierigkeit der Simulation ist ihre Unvollständigkeit. Das Problem, eine minimale vollständige Testmenge oder Stimulimenge zu finden, ist NP-vollständig [113]. Um alle möglichen Fehler einer Schaltung mit Sicherheit zu finden, müssen oft sehr große Testmengen erzeugt werden, die andererseits keine Garantie der Vollständigkeit liefern. Gerade bei komplexen Systemen ist es nötig, auf jeder Abstraktionsebene einen korrekten Entwurf zu garantieren. Einerseits bietet hier der manuelle Entwurf größere Freiheiten, andererseits kann eine Simulation durch ein korrektes System zur automatischen Synthese umgangen werden. Während die Synthese von VLSI-Strukturen aus der RT-Ebene meistens auf einer eindeutigen Abbildung der RT-Elemente auf Elemente aus einer Bauteilbibliothek beruht, deren Korrektheit ihrerseits einzeln verifiziert ist, bedarf es bei einer Synthese aus der Verhaltensbeschreibung oder der algorithmischen Ebene eines sehr viel höheren Aufwandes. Eine weitere Schwierigkeit bei der

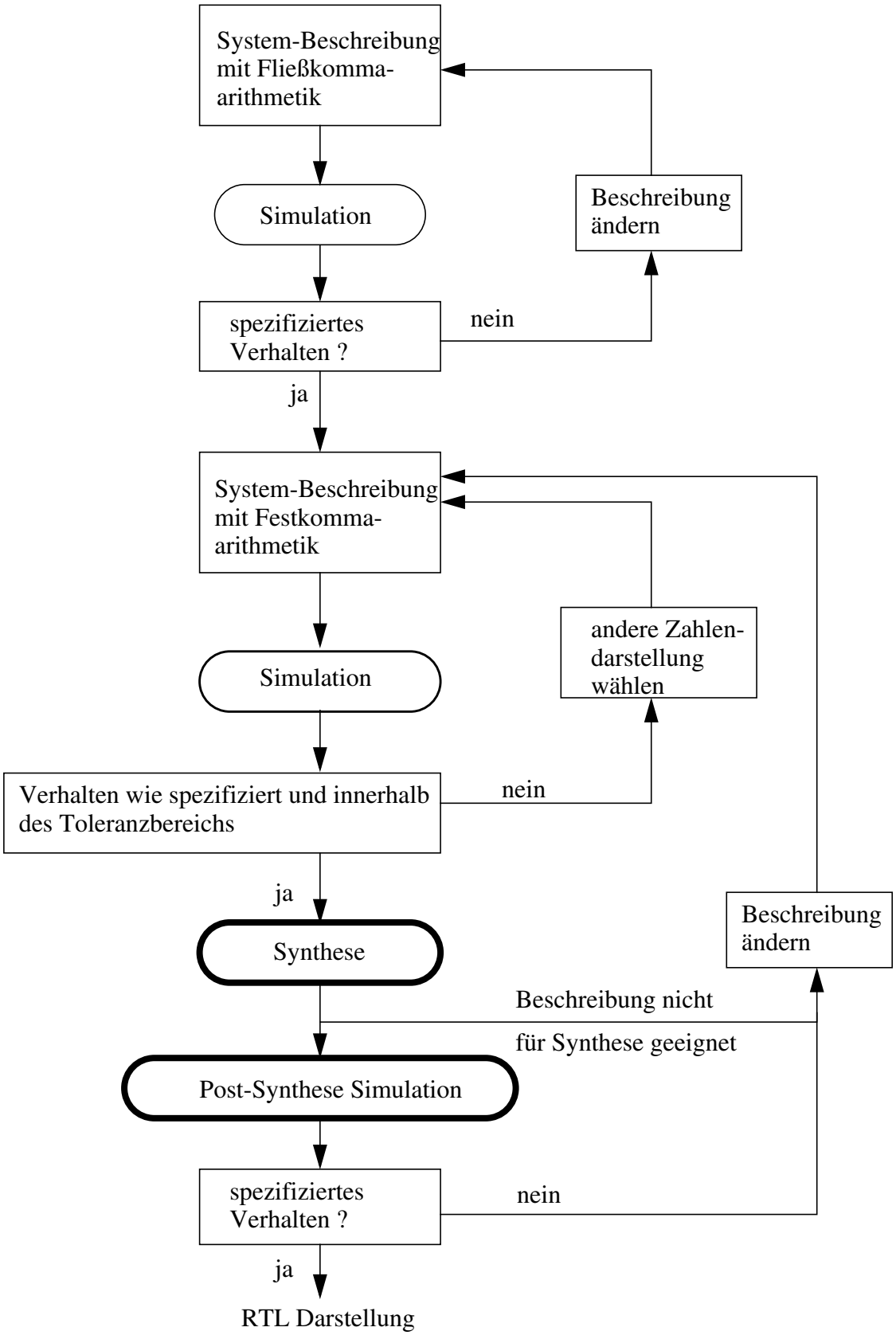



Bild 9: Einbettung der Postsynthese Simulation in den Entwurfsablauf

Simulation ist dadurch gegeben, daß es keine exakten Angaben über die Verzögerungszeiten und den Ressourcenbedarf der Schaltungen gibt. Es kann auf den sehr hohen Abstraktionsebenen nur mit Abschätzungen gearbeitet werden. Im Prototypenentwurf werden daher ideale Bedingungen angenommen, und durch iterative Optimierungen die gewünschte Geschwindigkeit angestrebt. Simulatoren arbeiten mit oberen und unteren Grenzen der Verzögerungszeiten. Die Signale werden in mehrwertiger Logik [17] dargestellt, wobei Signalwechsel als nicht definierte Werte erscheinen [75]. Aufgrund verschiedener Signallaufzeiten kann die Korrektheit von Simulationen nicht gewährleistet werden. Eine Akzeleration der Simulation [65] ist ein wesentlicher und wichtiger Schritt für die Beschleunigung des Entwurfs von VLSI-Systemen. Der automatische Entwurf von VLSI-Systemen wird für jeden Übergang zwischen den Ebenen angestrebt und verbessert. Es ist gerade in den niedrigeren Ebenen so, daß mit vielen Einschränkungen gearbeitet werden muß. Zum Beispiel werden Plazierer und Verdrahter für den automatischen Layoutentwurf eingesetzt. Die daraus resultierenden Entwürfe benötigen im Vergleich zu einem manuellen Entwurf oft die zwei- bis dreifache Chipfläche [71] [21]. Um diesen Nachteil des automatischen Entwurfs aufzufangen, werden Kompaktierer eingesetzt, die das Layout nachträglich bearbeiten [71]. Einige Systeme, die die Layoutkompaktierung durchführen, können zusätzlich eine Technologieanpassung vornehmen, durch die dann ein weiterer Flächengewinn erzielt wird. Ein Layout, welches manuell optimiert wurde, kann unter Umständen nicht einfach in eine neue Technologie umgesetzt werden. Ein Redesign mit einer neuen Technologie würde, wenn es einen Gewinn bringen soll, sehr lange dauern. Hier ist eine Automatisierung wichtig. Beim Entwurf in höheren Abstraktionsebenen müssen schon Entscheidungen getroffen werden, die erst in den niedrigeren wirksam werden. Zum Beispiel werden beim Entwurf in der RT-Ebene Entscheidungen über die Verbindungsstruktur der Schaltung - Bus- oder Multiplexer-Architektur - getroffen, deren Relevanz sich erst in der Layoutebene widerspiegelt, wenn der Platzbedarf der Verbindungsstruktur bekannt ist. Es ist im allgemeinen schwierig, eine Vorhersage über den Flächenbedarf des endgültigen Entwurfs zu machen, da gerade die Verbindungen zwischen den Komponenten bzw. deren Länge und damit ihr Flächenbedarf erst in den unteren Ebenen entschieden werden kann.

Im Rahmen dieser Arbeit wird der Übergang von der algorithmischen Ebene zur Register-Transfer Ebene betrachtet. Dazu werden die benötigten Eingabedaten spezifiziert, die als Zwischencode dargestellt werden, der direkt aus der Verhaltensbeschreibung durch Kompilation erzeugt werden kann. Die Bibliothek der zur Verfügung stehenden Elemente ist eine weitere Eingabe. Es muß eine Bibliothek angelegt werden, in der für jedes Bauteil die Funktionen, der Platz- und der Zeitbedarf abgelegt sind. Der Zeitbedarf wird für jede Funktion unabhängig betrachtet. Außerdem kann für jede Funktion eines Bauteils die Pipelineeigenschaft in Form der Verzögerungszeit einer Pipelinestufe angegeben werden. Als dritte Eingabe wird angegeben, wie lange die Eingänge der Schaltung mit den Eingangswerten mindestens belegt sein müssen. Wenn für jedes Element der Bibliothek eine genaue Angabe des Zeitbedarfs vorliegt, ist es durch das im folgenden erläuterte Synthesesystem möglich, eine optimale Nutzung der Ressourcen zu gewährleisten. In vielen anderen Synthesesystemen wird der unterschiedliche Zeitbedarf der verschiedenen Bauelemente nicht bzw. nur eingeschränkt betrachtet, so daß in einem System immer Bauteile vorhanden sind, die nicht optimal ausgenutzt werden. Im folgenden werden die Grundlagen der Synthese erläutert und das entwickelte System eingeordnet.

# 5 Der Ablauf der Synthese

Die Synthese, also die automatische Implementierung von abstrakt beschriebenen Schaltungen in ein Layout, kann - wie oben anhand der verschiedenen Ebenen schon gezeigt - in mehrere Teilschritte unterteilt werden. Wie Bild 7 zeigt, wird eine Funktion durch mehrere Teilfunktio-

nen und Verbindungsstrukturen implementiert. Der erste Schritt der Synthese besteht darin, für eine Funktion die Abbildung auf die Teilfunktionen zu finden.

```
INPUT A, B;
OUTPUT Y;
X := A * B;
Y := X + 1;
```

Bild 10: Funktion Y = f (A,B)

Beispielsweise kann die in Bild 10 dargestellte Funktion, welche aus einer Multiplikation und einer Addition besteht, mit Hilfe eines Multiplizierers und eines Addierers implementiert werden. Die Variablen *A* und *B* stellen die Eingänge und *Y* den Ausgang der Schaltung dar. Außerdem müssen eine Konstante 1 erzeugt und ein Register für die Variable *X* zur Verfügung gestellt werden. Bild 11 stellt eine mögliche Implementierung der Funktion dar. Aus dieser Implementierung geht hervor, welche Elemente benötigt werden, wobei aber nicht dargestellt ist, wie die Elemente ihrerseits aufgebaut sind. Außerdem wird an dieser Stelle noch keine Optimierung durchgeführt. Es ist offensichtlich, daß für die Variable *X* kein Register in der Implementierung benötigt wird, sondern eine einfache Verbindung vom Ausgang des Multiplizierers zum Eingang des Addierers ausreicht. Dieses Register wegzulassen, wäre schon ein Optimierungsschritt. Neben der Abbildung der einzelnen Befehle auf Bauteile muß eine Steuerung der Hardware realisiert werden. Beispielsweise muß das Register angesteuert werden, kurz nachdem die Eingänge mit den Variablen *A* und *B* belegt worden sind, um das Ergebnis der Multiplikation zu übernehmen. Es erfolgt eine zeitliche Einordnung der Befehle in Kontrollschritte, die nacheinander abgearbeitet werden, so daß jeder Befehl desselben Kontrollschrittes parallel ausgeführt wird. Ein Steuerwerk (Controller) übernimmt die Ansteuerung der Bauteile, gemäß der Einordnung der Befehle in die Kontrollschritte. Die Einordnung der Befehle in Kontrollschritte wird Scheduling genannt, wobei die Zuordnung der Befehle zu Bauteilen Assignment genannt wird.

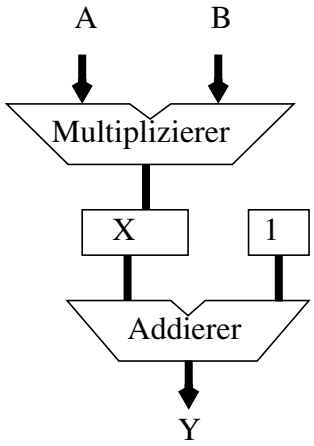



Bild 11: Die Funktion auf RT-Ebene

Nachdem das Assignment durchgeführt ist, wird die Möglichkeit, mehrere Befehle parallel ausführen zu können, genutzt. Anhand einer Analyse der Abhängigkeiten der Befehle untereinander können Parallelitäten berechnet werden. Mit verschiedenen Schedulingalgorithmen kann die Ablaufplanung optimiert werden, wobei unterschiedliche Ziele verfolgt werden können.

Zum Beispiel gibt es Schedulingalgorithmen, die optimale Ergebnisse bzgl. des Zeitbedarfs einer Schaltung berechnen. Andere Schedulingalgorithmen berechnen Ergebnisse mit geringem Bauteilbedarf, was aber zu schlechteren Ergebnissen beim Zeitbedarf führt. In den folgenden Abschnitten wird eine Auswahl dieser Algorithmen vorgestellt. In dem Beispiel aus Bild 10 gibt es keine Wahlmöglichkeiten für den Schedulingalgorithmus, da die Reihenfolge der Befehle durch die Datenabhängigkeiten festgelegt ist. Der zweite Befehl ist vom ersten abhängig, da er das Ergebnis des ersten Befehls benutzt. In vielen Synthesesystemen wird der Schritt des Scheduling zuerst ausgeführt, so daß jeder Befehl einem Kontrollschritt zugeordnet ist. Anhand dieser Zuordnung werden dann die Befehle den Bauteilen zugeordnet. Der Nachteil dieser Methode ist, daß der reale Zeitbedarf der Befehle beim Scheduling nicht berücksichtigt wird. Es kann verschiedene Implementierungsmöglichkeiten einer Funktion geben, zum Beispiel können unterschiedliche Addierer im gegebenen Beispiel zur Verfügung gestellt werden. An dieser Stelle muß dann ein Kompromiß zwischen einer schnellen, oft aber ressourcenintensiven Lösung und einer ressourcensparenden, dafür aber langsameren Lösung gefunden werden. Ist der reale Zeitbedarf der Befehle beim Scheduling nicht bekannt, werden Bauteile in die Schaltung eingefügt, die nicht optimal ausgenutzt werden. Der allgemeine Ablauf der Synthese wird in der Bild 12 dargestellt. Nachdem aus einer algorithmischen Beschreibung eines Modells ein Zwischenformat compiliert wurde, werden hieraus der Daten- und der Kontrollfluß generiert. Anhand des Datenflußes wird eine Datenpfad- oder Architektursynthese [101] durchgeführt. Für die Datenpfadsynthese werden die Schritte Scheduling, Allocation und Assignment benötigt, auf die noch eingegangen wird. Im nächsten Schritt wird das Steuerwerk synthetisiert. Dies geschieht in der Steuer- oder Kontrollflußsynthese. Hierbei werden der Steuerfluß und die Ergebnisse der Architektursynthese für den Entwurf des Controllers benötigt, was im weiteren noch intensiver betrachtet wird.

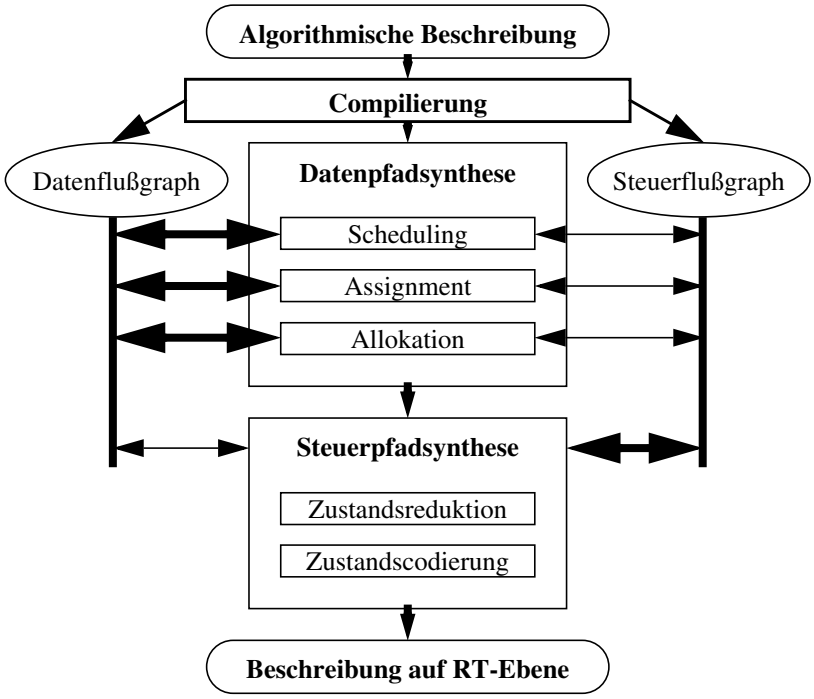



Bild 12: Allgemeiner Ablauf der High-Level Synthese

Die unterschiedlichen Abschnitte des Entwurfsprozesses digitaler Schaltungen werden besonders intensiv durch die Hardwarebeschreibungssprache VHDL unterstützt. Deshalb ist der fol-

gende Abschnitt im besonderen dieser Sprache gewidmet. Um dem allgemeinen Ablauf der Synthese gerecht zu werden, wird hier dann der Schritt des Scheduling, wie er in anderen Systemen implementiert ist, dargestellt. Dann wird das Assignment, die Zuordnung der Befehle zu den Bauteilen, betrachtet. Die Zurverfügungstellung der Bauteile, auch Allocation genannt, kann trotz der Verknüpfung mit dem Assignment als abgeschlossener Schritt betrachtet werden, so daß dieser ebenfalls im Rahmen eines Abschnittes dargestellt werden muß.

## 5.1 Die Bedeutung von VHDL für die Synthese

Die Hardwarebeschreibungssprache VHDL (Very High Speed Integrated Circuit Hardware Description Language) wurde ursprünglich für die Beschreibung, Spezifikation und Simulation von Hardware entwickelt. VHDL kann mit Hochsprachen wie C oder Pascal verglichen werden. Hochsprachliche Konstrukte sind ebenso definiert wie deren Möglichkeiten zur Strukturierung eines (Software/Hardware-)Systems. Das Verhalten von VHDL ist aber, und dies ist der wesentliche Unterschied, dem Verhalten realer Hardware angepaßt, deren Elemente parallel arbeiten. In den letzten Jahren wurde VHDL immer mehr zur Entwicklung von Hardwaresystemen eingesetzt. Auf einer hohen technologieunabhängigen Ebene wird ein System funktional beschrieben und simuliert. Durch die immer stärker werdende Unterstützung der Entwickler durch Synthesetools kann dann direkt eine automatische Umsetzung in eine strukturelle Beschreibung und weiter in die Gatterebene durchgeführt werden. Der Entwickler kann sein System also unabhängig von der aktuellen Technologie beschreiben. Erst zu einem späten Zeitpunkt wird dann die konkrete Implementierung, z.B. als ASIC oder FPGA, festgelegt. VHDL unterstützt den Entwickler auf den verschiedenen Abstraktionsebenen der Hardwareentwicklung. Hierzu kann der Entwickler z.B. Realzahlen für Berechnungen nutzen, ohne sich über die endgültigen Zahlenformate und deren Wortbreiten im Klaren zu sein. Im nächsten Schritt werden dann Realzahlen durch Integerzahlen bzw. Festkommazahlen ersetzt. Dies kann lokal geschehen, da entsprechende Konvertierungen zur Verfügung stehen, bzw. einfach beschrieben werden können. Andrerseits kann der Entwickler unabhängig von der Struktur der Hardware und von der verwendeten Technologie das System beschreiben, dazu steht die VHDL-Verhaltensbeschreibung zur Verfügung. Die Struktur eines Moduls wird entweder automatisch oder manuell aus der Verhaltensbeschreibung erzeugt. In VHDL stehen Sprachkonstrukte für die strukturelle Beschreibung eines Systems, also für die Register-Transfer-Ebene, zur Verfügung. In der Strukturbeschreibung können Timinginformationen mit eingebracht werden, so daß in einer ziemlich frühen Phase bereits Abschätzungen der Geschwindigkeit gemacht werden können. Des weiteren muß zwischen parallelem und sequentiellem VHDL unterschieden werden. Die Möglichkeiten, welche durch sequentielles VHDL gegeben sind, sind vergleichbar mit denen herkömmlicher Programmiersprachen, wie C oder Pascal. Andrerseits entspricht paralleles VHDL eher den Möglichkeiten realer Hardware. Ein Entwickler kann die verschiedenen Möglichkeiten beliebig mischen, so daß eine klare Trennung zwischen parallelem und sequentiellem VHDL oder Verhalten und Struktur in realen Systemen selten oder gar nicht zu finden ist. Es kann aber gesagt werden, daß eine Verhaltensbeschreibung in VHDL häufig die sequentiellen Sprachkonstrukte nutzt während die Strukturbeschreibung mehr auf den parallelen Konstrukten basiert. Im Rahmen dieser Arbeit wird daher von einer sequentiellen Verhaltensbeschreibung ausgegangen, aus der dann eine Strukturbeschreibung synthetisiert wird. Zusammenfassend kann gesagt werden, daß VHDL eine sehr komfortable und umfangreiche Sprache ist, die den Entwickler auf verschiedenen Ebenen, welche im Rahmen einer Systementwicklung durchlaufen werden, unterstützt. Durch die Etablierung von Entwicklungswerkzeugen, welche als Eingabemedium VHDL vorsehen, kommt es zu einer immer größeren Verbreitung dieser Sprache. Heutzutage kann man sicherlich nicht mehr von einer reinen Simulationssprache sprechen, wie es der ursprünglichen Intention entspricht, sondern es handelt sich bei VHDL vielmehr um eine

Hardwareentwicklungsumgebung. VHDL bildet außerdem eine Plattform, die die Kommunikation zwischen Systementwicklern und Halbleiterherstellern erleichtert.

### 5.2 Schedulingalgorithmen

Wie schon gesagt, wird jeder Befehl in einen Kontrollschritt eingefügt. Als Beispiel wird die Befehlsfolge in Bild 13 zugrundegelegt, um verschiedene Algorithmen darzulegen. Außerdem werden mögliche Optimierungen des Quelltextes, wie zum Beispiel die Veränderung der Klammerung (Assoziativgesetz), dargestellt.

```
-- entity Declaration für das Synthesebeispiel --
entity SYNEX1 is
port(A,B,C,D,E: in INTEGER; X,Y : out INTEGER);
end SYNEX1;
-- Architektur Deklaration --
architecture H_L of SYNEX1 is
begin
X := E * (A + B + C);
Y := (A + C) * (C + D);
end H_L;
```

Bild 13: Beispiel eines algorithmisch beschriebenen Bauteils

Im ersten Schritt wird der Datenflußgraph der zu synthetisierenden Beschreibung berechnet. Die genaue Definition des Datenflußes wird später erläutert. An dieser Stelle reicht folgende Aussage: zwei Befehle sind voneinander abhängig, wenn der zweite Befehl Ergebnisse des ersten benutzt. Um den Datenflußgraphen für das Beispiel aufstellen zu können, werden die elementaren Befehle, die im Beispiel vorkommen, in Form von Knoten mit der Operation und der Nummer des elementaren Befehls dargestellt. Die Abhängigkeiten der Knoten, bzw. der elementaren Befehle, werden als Kanten dargestellt. Der Datenflußgraph für das Beispiel ist in Bild 14 gegeben.

Durch den Datenflußgraphen sind Abhängigkeiten zwischen den Befehlen gegeben, die durch die Schedulingalgorithmen berücksichtigt werden müssen. Zwei relativ einfache Algorithmen für das Scheduling sind die Algorithmen ASAP (as soon as possible) und ALAP (as late as possible). Der ASAP Algorithmus ordnet alle Befehle so früh wie möglich in die Kontrollschritte ein. Der ASAP würde für das Beispiel das in Bild 15 dargestellte Ergebnis liefern.

Es werden genau drei Kontrollschritte benötigt, um das Ergebnis zu berechnen. Sie werden hier mit *K0*, *K1* und *K2* bezeichnet. Der ALAP Algorithmus rechnet ebenfalls ein Scheduling aus, für das nur drei Kontrollschritte benötigt werden. Dabei ist aber zu bedenken, daß beim ALAP die Ausführung der Befehle, bzw. die Einordnung der Befehle in Kontrollschritte, bis zum spähtest möglichen Zeitpunkt verzögert wird. In Bild 16 ist das Ergebnis, welches durch den ALAP Algorithmus geliefert wird, dargestellt. An diesem Beispiel wird deutlich, daß sowohl der ASAP als auch der ALAP Algorithmus zu dem gleichen Ergebnis führen, was den Zeitbedarf der Schaltung betrifft. Einschränkungen der Anzahl der zur Verfügung stehenden Bauteile werden nicht berücksichtigt.

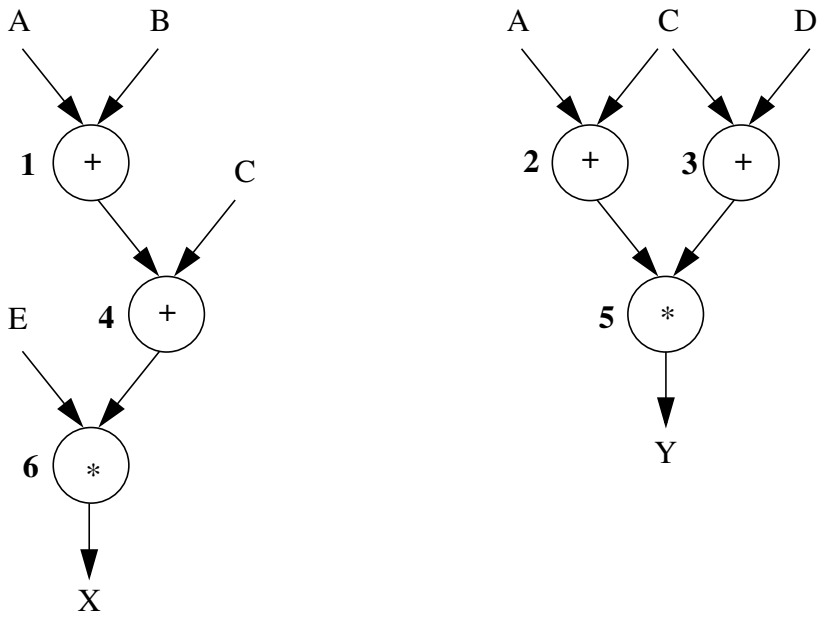



Bild 14: Der Datenflußgraph für das Synthesebeispiel

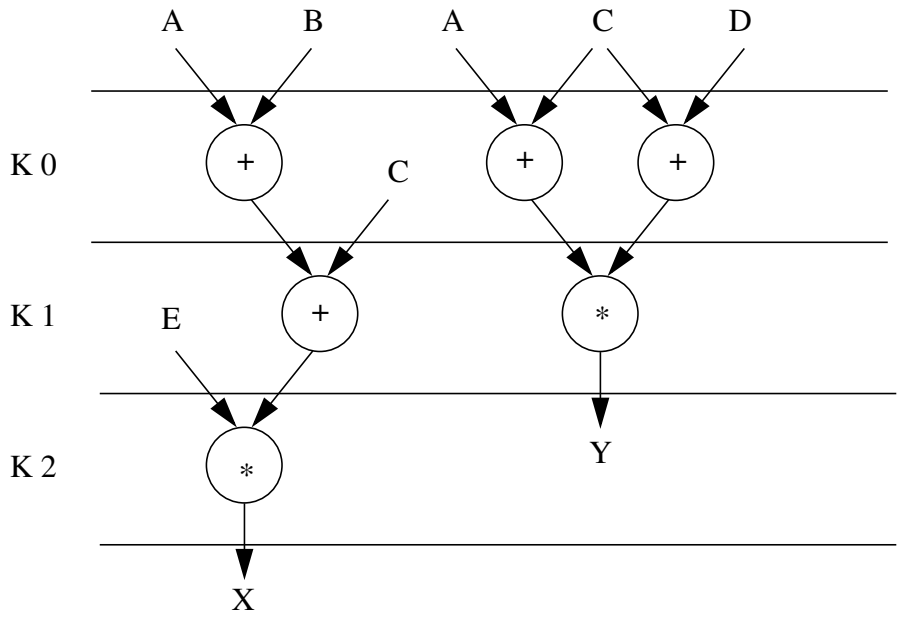



Bild 15: ASAP Scheduling Ergebnis

Wird nun jedem Befehl ein Bauteil zugeordnet, wobei die Mehrfachnutzung der Bauteile voll ausgenutzt werden soll, so fällt auf, daß im Fall des ASAP Scheduling nur ein Multiplizierer benötigt wird, welcher sowohl im zweiten als auch im dritten Kontrollschritt eingesetzt werden kann. Beim ALAP Scheduling werden hingegen zwei Multiplizierer benötigt, die parallel im dritten Kontrollschritt benutzt werden. In beiden Fällen werden drei Addierer benötigt, die bei ASAP im ersten Kontrollschritt bzw. bei ALAP im zweiten parallel arbeiten. Anhand dieser Bewertung der Scheduling Ergebnisse wird sicherlich bei diesem Beispiel die Entscheidung für ASAP fallen. Das Ergebnis wird für die weiteren Schritte benutzt. Da das optimale Ergebnis oft nicht so einfach zu finden ist, sondern irgendwo zwischen den Ergebnissen von ALAP und ASAP liegt, wurden einige Methoden entwickelt, die auf der Grundlage der ASAP und ALAP

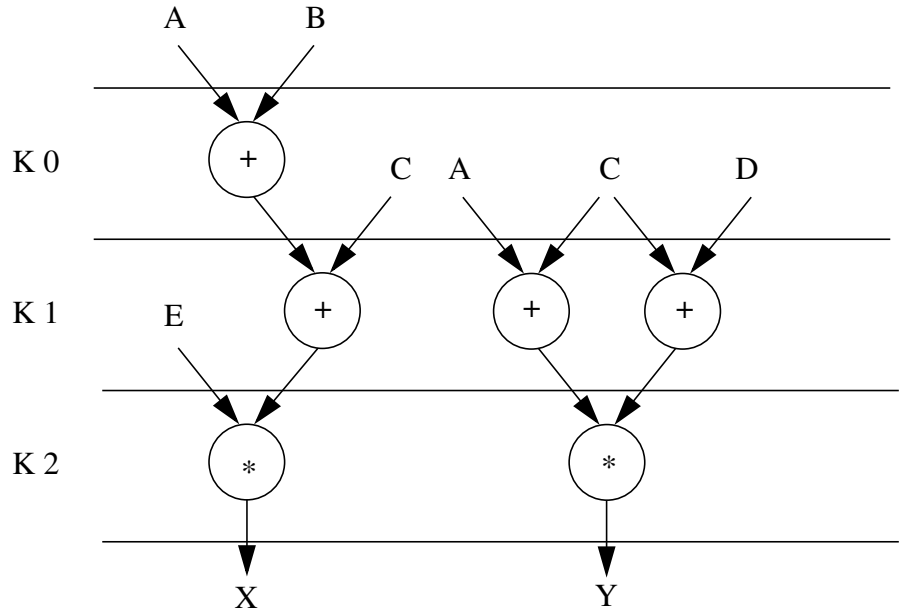



Bild 16: ALAP Scheduling Ergebnis

Algorithmen aufbauend die Befehle umordnen, um eine optimale Einordnung zu finden. Einige der Algorithmen bzw. der Klassen von Algorithmen, die auch in Bild 17 klassifiziert werden, sollen hier kurz dargestellt werden, um einen Eindruck von den verschiedenen Möglichkeiten zu bekommen. Da hier kein vollständiger Überblick über die vorhandenen Algorithmen dargestellt ist, wird auf [20] [25] [81] [116] verwiesen. Viele der in der Literatur dargestellten Scheduling-Algorithmen sind Verfeinerungen oder Spezialfälle der hier dargestellten Algorithmen, weshalb an dieser Stelle nur diese grundlegenden Algorithmen und Ideen ausgewählt wurden.

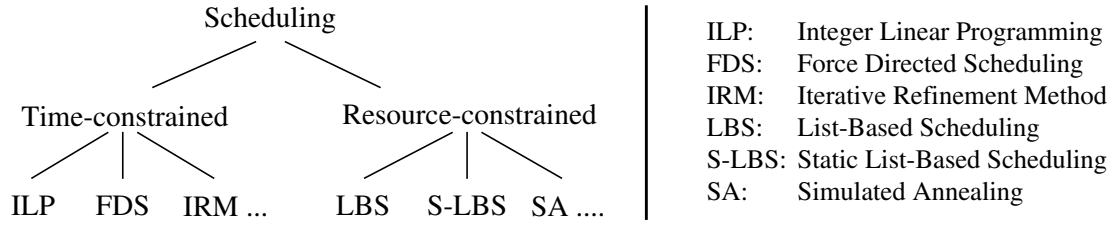



Bild 17: Klassifikation von Scheduling-Algorithmen

### 5.3 Time-constrained Scheduling Algorithmen

Wie aus der Bild 17 hervorgeht kann eine Unterscheidung der Schedulingalgorithmen anhand ihrer Zielfunktion durchgeführt werden. Time-constrained Algorithmen sind Schedulingalgorithmen, die von einer Zeitoptimalen Lösung ausgehen, die Beispielsweise mit einem ASAP Algorithmus gefunden wird. Es wird nun versucht den Bauteilbedarf zu minimieren, wobei der Zeitbedarf aber nicht zunehmen darf. Time-constrained bedeutet: Der Zeitbedarf ist festgelegt. In den folgenden Abschnitten werden einige Time-constrained Algorithmen vorgestellt.

#### 5.3.1 Integer Linear Programming

Wie in [2] dargestellt wird, können Methoden aus dem Gebiet des Integer Linear Programming (ILP) verwendet werden. Bei den ILP Methoden werden für jede Bedingung, die es im Rahmen des Scheduling zu berücksichtigen gibt, entsprechende Gleichungen oder Ungleichungen aufgestellt. Mit ILP versucht man nun, das Gleichungssystem zu lösen. Bei der korrekten Aufstel-

lung der Gleichungen können optimale Lösungen gefunden werden, was aber zu einer exponentiellen Rechenzeit führt. Einige Erweiterungen zu diesen Verfahren, die wichtige Aspekte von Allokation und Assignment berücksichtigen, sind in [72] wiedergegeben. Der Vorteil der ILP-Verfahren besteht darin, daß optimale Ergebnisse, allerdings bei exponentieller Zunahme der Laufzeit, berechnet werden können. Deshalb sind ILP-Methoden für kompliziertere Scheduling-Probleme nicht geeignet; sie werden vielmehr zur Verifikation anderer Scheduling-Verfahren eingesetzt, da sie optimale Ergebnisse liefern, die als Benchmark genutzt werden können.

### 5.3.2 Force Directed Scheduling

Eine anderes Verfahren, das mit FDS (Force Directed Scheduling) bezeichnet wird, basiert, wie viele andere Verfahren auch, auf der iterativen Optimierung von Ergebnissen, die der ASAP und ALAP Algorithmus berechnet haben [91]. Initial werden sowohl der ASAP als auch der ALAP Algorithmus ausgeführt, die zwei unterschiedliche Schedulingergebnisse liefern, wobei die Gesamtanzahl der benötigten Kontrollschritte gleich ist. Durch die Ausführung der beiden Algorithmen wird für jeden Befehl ein Intervall aus Kontrollschritten gebildet, in denen der Befehl ausführbar ist. Dabei berechnet der ASAP Algorithmus den frühest möglichen Kontrollschritt, in dem ein Befehl ausgeführt werden kann, und damit den Anfang des Intervalls, ebenso berechnet der ALAP Algorithmus den spätesten Zeitpunkt, in dem ein Befehl ausgeführt werden kann, und damit das Ende des Intervalls. Für die Befehle $b_0$ ... $b_n$ und den Kontrollschritten $k_0$ .. $k_m$ wird eine Matrix $M$ nach der folgenden Vorschrift gebildet : $m_{ij} = 0$, falls der Befehl $b_i$ nicht im Kontrollschritt $k_j$ ausgeführt werden kann und $m_{ij} = 1/p$, falls der Befehl $b_i$ in $k_j$ ausgeführt werden kann, wobei es insgesamt $p$ Kontrollschritte gibt, in denen $b_i$ ausgeführt werden kann. Nun muß iterativ die endgültige Zuordnung der Befehle zu den Kontrollschritten gefunden werden, indem einzelne Befehle fest an einen Kontrollschritt gebunden werden. In der Matrix wird an der Stelle $m_{ij} = 1$ gesetzt, wenn der Befehl $b_i$ endgültig dem Kontrollschritt $k_j$ zugewiesen wird. Die anderen Elemente der Spalte $i$ werden auf $0$ gesetzt. Für jeden Kontrollschritt werden die ihm zugewiesenen Werte, also die Elemente jeweils einer Zeile der Matrix, addiert. Eine Optimierung der Zuordnung wird erreicht, indem diese Summen balanciert werden. Dies bedeutet aber, daß in jedem Kontrollschritt möglichst die gleiche Anzahl von Befehlen ausgeführt wird. Dies führt zu einer gleichmäßigen Auslastung der Bauteile und zu einer Minimierung der benötigten Anzahl von Bauteilen. Das Beispiel aus Bild 14 führt initial zu dem in Bild 18 dargestellten Ergebnis. Der Befehl 0 wird dem Kontrollschritt 0, der Befehl 1 dem Kontrollschritt 1 und der Befehl 2 dem Kontrollschritt 2 zugewiesen, da ASAP und ALAP die gleichen Ergebnisse für diese Befehle errechnet haben.

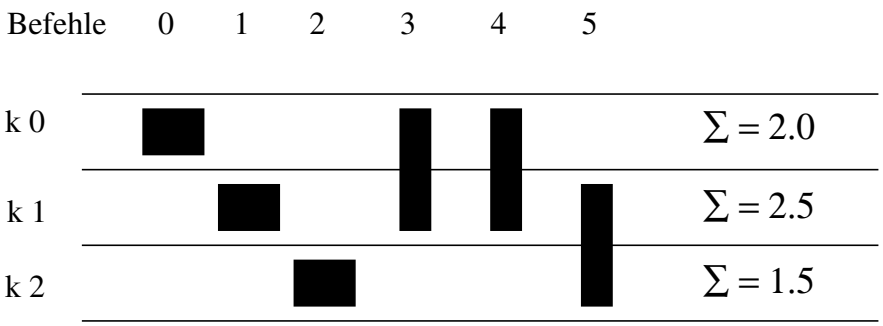



Bild 18: FDS nach der Initialisierung

Die Befehle 3 und 4 werden jeweils mit einem Wert von 0.5 in den Kontrollschritt $k_0$ oder $k_1$ und der Befehl 5 in die Kontrollschritte $k_1$ und $k_2$ eingeordnet, da es jeweils zwei mögliche Kontrollschritte für die Einordnung dieser Befehle gibt. Bei der endgültigen Einordnung der Befehle wird von dem Kontrollschritt mit der höchsten Summe ausgegangen und ein Befehl, welcher in mehreren Kontrollschritten ausgeführt werden kann, aus diesen herausgenommen. In dem dargestellten Beispiel wird Befehl 6 aus dem Kontrollschritt $k_1$ entfernt, wodurch der Wert von $k_1$ auf 2.0 sinkt. Der Wert von $k_2$ steigt auf 2.0 an, da der Befehl 6 jetzt nur noch in diesem Kontrollschritt ausgeführt werden kann. In Bild 19 ist das Ergebnis dargestellt. Dadurch, daß hier schon eine Balance für die Summen gegeben ist, wird nun einer der Befehle 4 oder 5 in dem ersten und der andere in dem zweiten Kontrollschritt fixiert. Auch hier ist das Auffinden eines optimalen Schedulings nicht garantiert, es können aber relativ gute Lösungen bei kurzer Laufzeit des Algorithmus errechnet werden.

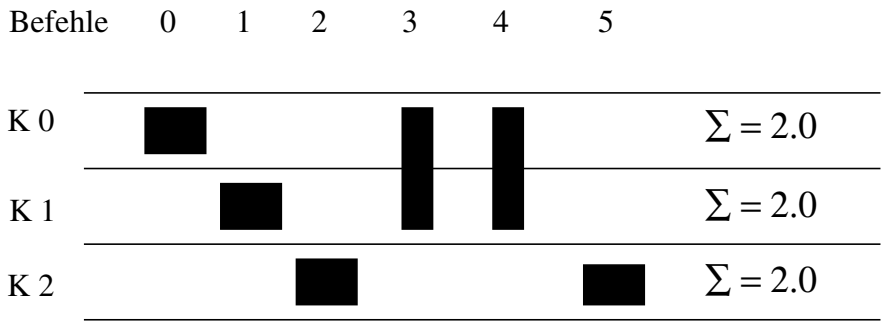



Bild 19: FDS nach der 1. Iteration

Die Methode wird dadurch erweitert, daß die verschiedenen Operationen getrennt betrachtet werden. Dies führt teilweise zu besseren Ergebnissen.

### 5.3.3 Iterativ Refinement Method

Ein anderes Scheduling-Verfahren ist die sogenannte Methode der iterativen Verfeinerung. Bei der IRM (Iterative refinement Method) wird wieder von einem gegebenen Scheduling, welches zum Beispiel durch einen ASAP oder ALAP Algorithmus berechnet wurde, ausgegangen und durch kleine Veränderungen des Schedulings eine bessere Lösung gesucht. Hier können verschiedene Optimierungstechniken eingesetzt werden, wie zum Beispiel Simulated Annealing. Allerdings ist auch bei diesem Ansatz nicht garantiert, daß die optimale Lösung gefunden wird. Zudem hängt die Qualität der Lösung entscheidend von dem anfänglichen Scheduling ab. Um eine Lösung in der Nähe des Optimums zu finden, wird der Algorithmus daher mehrmals mit unterschiedlichen, initialen Ablaufplänen und anschließender Auswahl der besten Lösung wiederholt. Aus diesem Grunde eignet sich dieser Algorithmus insbesondere zur parallelen Anwendung, bei der mehrere Workstations den Algorithmus mit verschiedenen initialen Schedulings durchführen. Im Prinzip kann bei IRM nicht von einer einzigen Methode gesprochen werden, vielmehr handelt es sich um eine ganze Klasse Methoden, die ähnliche Eigenschaften besitzen. Die wesentliche Eigenschaft ist der Nichtdeterminismus, da die Parameter für die Verfeinerung meist zufällig gewählt werden, somit bei gleicher Eingabe unterschiedliche Ergebnisse berechnet werden können.

Eine andere Klasse von Scheduling-Algorithmen ist dadurch charakterisiert, daß die maximale Anzahl der benötigten Bauteile oder Ressourcen durch eine obere Schranke begrenzt wird. Die Ressource-constrained und Time-constrained Methoden stehen gleichberechtigt nebeneinan-

der. Es werden lediglich unterschiedliche Ziele bei der Optimierung der Ergebnisse verfolgt.

### 5.4 Ressource-constrained Scheduling-Algorithmen

Die charakteristische Eigenschaft ressourcenorientierter (ressource-constrained) Scheduling-Verfahren ist, daß die Anzahl elementarer, zur Verfügung stehender Komponenten, bzw. die Gesamtfläche, die sie benötigen, durch eine vorgegebene, obere Grenze beschränkt ist. Unter Beachtung dieser Grenze wird dann die Anzahl der Kontrollschritte minimiert. In den meisten Fällen wird dabei eine Vereinfachung gemacht, indem die Anzahl der Operationen, die in jedem Schritt parallel ausgeführt werden können, beschränkt wird. In der Literatur werden mehrere ressourcenorientierte Methoden unterschieden. Ressource-constrained heißt also: die obere Schranke der zur Verfügung stehenden Bauteile oder Operationseinheiten wird festgelegt.

#### 5.4.1 List Based Scheduling

Bei der LBS (List-Based Scheduling) Methode werden jedem Befehl, der in einen Kontrollschritt einzuordnen ist, eine Priorität zugewiesen. Diese Priorität errechnet sich aus der Dringlichkeit, einen Befehl einzuordnen, also aus seiner Bedeutung für nachfolgende Befehle. Befehle, die auf dem sog. kritischen Pfad liegen, also diejenigen, bei denen sich eine Verzögerung unmittelbar auf das Gesamtergebnis auswirkt, bekommen die höchste Priorität. Die Prioritäten werden in einer sortierten dynamischen Liste gespeichert. Beim Scheduling werden die Befehle mit der höchsten Priorität zuerst eingeordnet. Dabei werden in jedem Kontrollschritt nur soviele Befehle eingeordnet, wie durch die Ressourcenschranke erlaubt ist. Die eingeordneten Befehle werden aus der Liste gelöscht. Soll die Liste nicht nach jedem Schritt neu berechnet werden, sondern am Anfang des Schedulings statisch zur Verfügung stehen, so wird das Static List Based Scheduling (S-LBS) angewandt. Das dynamische LBS berechnet in der Regel bessere Ergebnisse als S-LBS, wobei aber die Liste der Prioriäten mehrmals berechnet werden muß, was zu einer höheren Laufzeit von LBS gegenüber S-LBS führt. Beide Methoden haben polynomielle Laufzeit und liefern deterministische Ergebnisse.

#### 5.4.2 Zufallsbasierte Methoden

Unter den ressourcenorientierten Methoden finden sich auch Methoden, die auf zufallsbasierte Optimierungsalgorithmen wie z.B. Simulated Annealing (SA) basieren. Sie beginnen mit einer vorgegebenen Ablaufplanung, um die Zuweisung zu Kontrollschritten anschließend iterativ zu verbessern. Die Auswahl der zu verändernden Parameter geschieht zufällig. Es handelt sich also um eine Klasse von Methoden, die nicht deterministisch arbeiten. Manche dieser Methoden eignen sich daher für die Parallelisierung, indem mit unterschiedlichen initialen Schedulings, auf mehreren Rechnern optimiert wird, und das beste erzielte Ergebnis abschließend ausgewählt wird.

#### 5.4.3 Zieltechnologie einbeziehende Methoden

Ein wichtiges Verfahren aus dem Bereich der ressourcenorientierten Methoden wird in [68] von Krämer vorgestellt. Die Zuweisung von Befehlen zu den Bauteilen wird hier von einer Schätzung der Verzögerungszeiten der Komponenten abhängig gemacht. Jene Zuweisungen betrachten die unterschiedlichen Ausführungszeiten der Komponenten und benötigen daher die endgültige Komponentenbibliothek bereits innerhalb des Schedulingprozesses. Dieses Verfahren ist an die bereits beschriebene FDS Methode angelehnt, aber mit dem Unterschied, daß die konkrete Zielbibliothek berücksichtigt wird. Allerdings wird die abschließende Zuweisung der Befehle zu den Komponenten nach dem Scheduling ausgeführt, so daß optimale Ergebnisse nicht garantiert werden können. Das System von Krämer ist ein ressourcenorientiertes Verfah-

ren, wobei aber Zieltechnologie einbeziehende Methoden ebenso im Rahmen der zeitorientierten Verfahren entwickelt werden. Die Eigenschaft der Ressourcenorientiertheit ist also ist also keine zwingende Eigenschaft der Zieltechnologie einbeziehenden Methoden.

### 5.4.4 Vor- und Nachteile

Ein Nachteil der oben angegebenen Verfahren ist, daß sie innerhalb der Ablaufplanung nicht direkt die Komponenten betrachten, die in der Bauteilbibliothek bereitgestellt werden. Die Dauer eines Kontrollschrittes wird an die langsamsten Komponenten angepaßt. Daher ist die Auslastung der schnellen Komponenten u.U. sehr niedrig. Deshalb berücksichtigen aktuelle Erweiterungen solcher Scheduling-Verfahren die konkreten, verfügbaren Komponenten bei der Ablaufplanung. Hierbei muß ferner berücksichtigt werden, daß gleiche Operationen von unterschiedlichen Operationseinheiten ausgeführt werden können. In den meisten Fällen wird dabei die funktionale Überschneidung der von den Komponenten zur Verfügung gestellten Operationen vernachlässigt. In der Regel erfolgt die endgültige Zuweisung der elementaren Befehle zu den instanziierten Operationseinheiten nachdem das Scheduling beendet ist. Aus diesem Grund kann die Qualität des Schedulingergebnisses nicht endgültig eingeschätzt werden. Des weiteren beachten die meisten Scheduling-Algorithmen nicht die Ausführungszeit einer Operation, die zudem vom Typ der Operationseinheit abhängt, das die Operation ausführt. Obwohl die Methode, die von Krämer eingeführt wird, bereits eine Modulbibliothek berücksichtigt, weist sie Operationen lediglich unterschiedlichen Typen von Basisoperationseinheiten zu, ohne jedoch eine konkrete Instanzierung der Ressourcen vorzunehmen. Als eine Folge davon wird das Assignment getrennt, unter Berücksichtigung der Randbedingungen behandelt, die von dem bereits berechneten Scheduling vorgegeben sind. An dieser Stelle ist eine weitere Verfeinerung des Scheduling infolge der verschiedenen Möglichkeiten, die durch unterschiedlichen Assignments gegeben sind, nicht möglich. Anhand des Beispiels in Abschnitt 13.1 wird die enorme Beschleunigung, welche durch die Berücksichtigung der konkreten Operationseinheiten erreicht werden kann, deutlich. Im folgenden Abschnitt wird das Assignment genauer betrachtet.

## 5.5 Assignment und Allocation

Das Assignment, also die Zuweisung der Befehle an die Bauteile, ist sehr eng mit der Allocation, der Zurverfügungstellung der Bauteile, verbunden. Der Prozeß der Allocation und des Assignment ist im wesentlichen durch die Randbedingungen (Constraints) gekennzeichnet, die eine Vorgabe bzgl. des erlaubten Hardwarebedarfs oder der angestrebten Laufzeit machen. Eine sehr einfache Assignmenttechnik ist das "gierige" (greedy) Assignment. Die in Kontrollschritte eingeordneten Befehle werden der Reihe nach bzgl. der Kontrollschritte abgearbeitet und jedem der Befehle wird das schnellste zur Verfügung stehende Bauteil zugeordnet. Hierbei wird natürlich darauf geachtet, daß Bauteile mehr als einmal benutzt werden. Das heißt in jedem Schritt wird dann ein neues Bauteil zur Verfügung gestellt, wenn kein Bauteil mit der benötigten Funktion frei ist. Gierig ist diese Allocation/Assignment Methode deshalb, weil keine "Vorsorge" für die Zukunft getroffen wird. Der aktuelle Befehl wird möglichst schnell ausgeführt. Der ’gierige’ Algorithmus führt ziemlich schnell zu guten Ergebnissen, aber in den meisten Anwendungsfällen nicht zu den optimalen. Das andere Extrem ist die vollständige Suche nach dem Optimum, wozu alle Möglichkeiten erzeugt werden müssen. Die vollständige Suche garantiert zwar das optimale Ergebnis, aber die Rechenzeit ist in den meisten Anwendungsfällen zu hoch, da es einer exponentiellen Laufzeit bedarf. Dieses System ist also nur geeignet für kleine Anwendungen. Teilprobleme des Assignments, zum Beispiel die Zuweisung der Variablen zu Registern, können optimal und schnell mit dem left-edge-Algorithmus, welcher in Abschnitt 9.10 dargestellt wird, gelöst werden, wobei aber das Gesamtoptimum durch die Abhängigkeiten zwischen dem Register-Assignment und den anderen Assignmentschritten nicht immer erreicht

werden kann. Das heißt: ist die Registerzuordnung durch einen left-edge Algorithmus festgelegt, so kann dies dazu führen, daß die Anzahl der benötigten Multiplexer und Verbindungswege nicht minimal ist.

### 5.6 Variablen-Register-Assignment

Ein Schritt der Synthese ist die Zuweisung der Variablen an Register. Dazu werden die Lebenszeiten der Variablen berechnet, wodurch die Möglichkeit gegeben ist, mehrere Variablen dem gleichen Register zuzuordnen, wenn sich ihre Lebenszeiten nicht überschneiden. In Bild 20 werden für das Beispiel aus Bild 13 die Lebenszeiten der Variablen nach der Durchführung des ASAP Algorithmus dargestellt. Sind die Lebenszeiten bekannt, so können Variablen zusammengefaßt und dem gleichen Register zugeordnet werden. Bild 21 zeigt die Zuordnung der Variablen zu Registern. Jedes Zwischenergebnis, hier mit *Z1 ... Z4* benannt, wird dabei ebenfalls in einem Register zwischengespeichert. Für die Eingangsvariablen werden keine Register zur Verfügung gestellt.

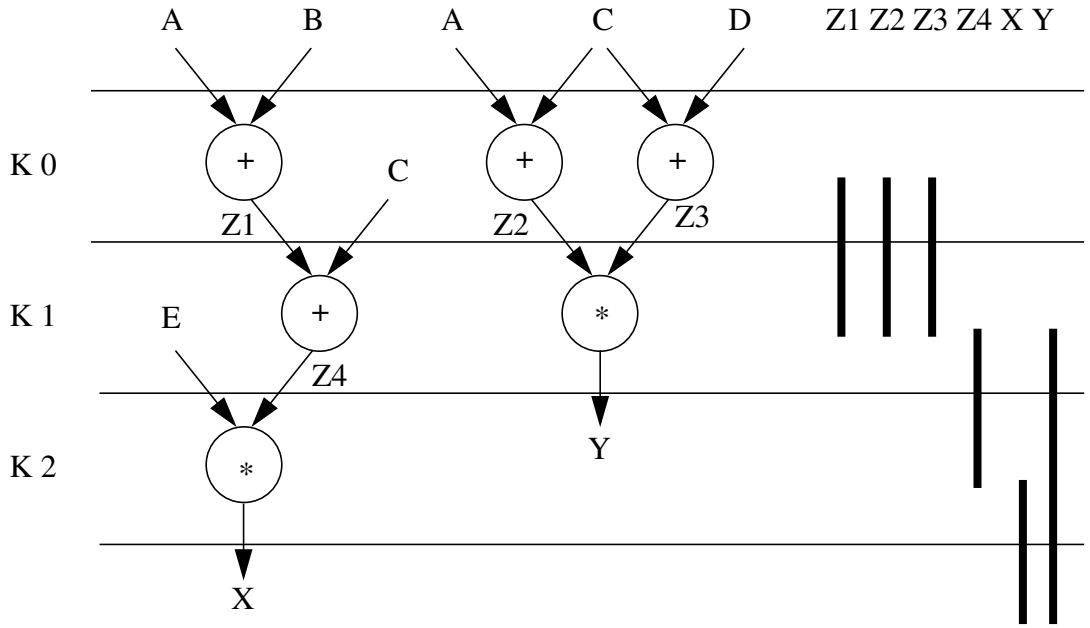



Bild 20: Lebenszeiten der Variablen

In Bild 21 ist eine Möglichkeit gezeigt, wie die Variablen den Registern zugeordnet werden können. Hierbei wurde jede Variable einem Register zugeordnet. Durch die Zusammenfassung der Variablen sind nur drei Register nötig, um alle Variablen darstellen zu können.

*Z1 -> R1*

*Z2 -> R2*

*Z3 -> R3*

*Z4 -> R1*

*X -> R2*

*Y -> R3*

Bild 21: Variablen Register Zuordnung

Das Ergebnis einer Synthese für eine Multiplexerarchitektur ist in Bild 22 gezeigt. Die Multiplexer und Register müssen dabei durch einen Controller gesteuert werden, so daß die richtige zeitliche Abfolge der Befehle möglich ist. Im ersten Schritt werden drei Additionen ausgeführt und die Ergebnisse in *R1*, *R2* und *R3* abgelegt. Im zweiten Schritt wird eine Addition und eine Multiplikation durchgeführt, wobei die Ergebnisse im Register *R1* und *R3* abgelegt werden. Im dritten Schritt wird wieder eine Multiplikation ausgeführt, deren Ergebnis in *R2* abgelegt wird. Die Ergebnisse der Befehlsfolge stehen jetzt in den Registern *R2* und *R3*.

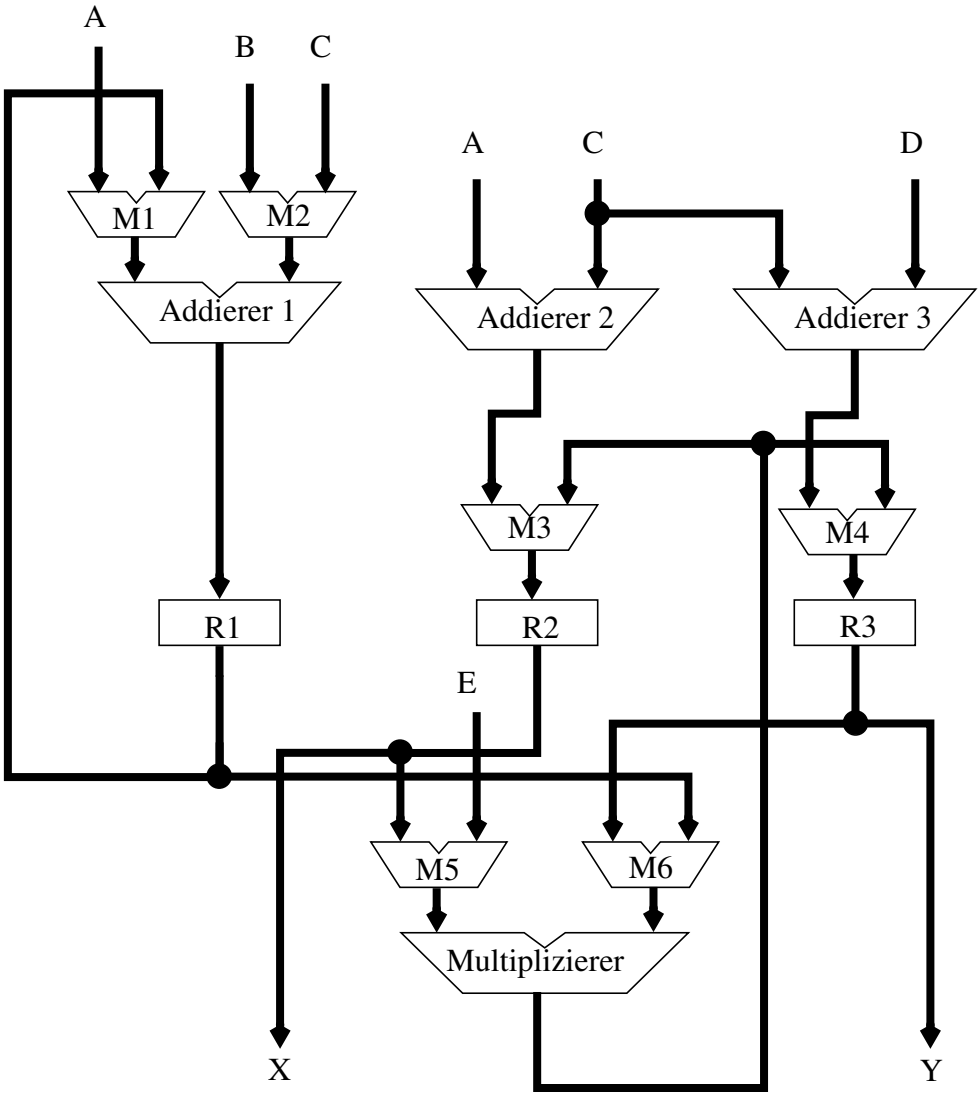



Bild 22: RT - Struktur nach Synthese mit ASAP Algorithmus

Die Ansteuerung der Bauteile wird in der Tabelle 2 dargestellt. Es wird dargelegt, in welchem Kontrollschritt welche Leitungen angesteuert werden. Die Register haben eine Steuerleitung. Sie sollen den anliegenden Wert übernehmen, wenn das Steuersignal auf ‚1' steht. Der Übernahmezeitpunkt der anliegenden Werte in die Register ist dabei in der Mitte der Kontrollschritte. Dies könnte dadurch gewährleistet werden, daß die Steuersignale sich synchron mit der positiven Taktflanke verändern, wobei die Register auf die negative Taktflanke reagieren. Die Multiplexer werden ebenfalls durch eine Steuerleitung angesteuert, wobei eine ‚0' den linken Eingang weiterschaltet und eine ‚1' den rechten. Das Timing der Register ist in Bild 23 dargestellt.

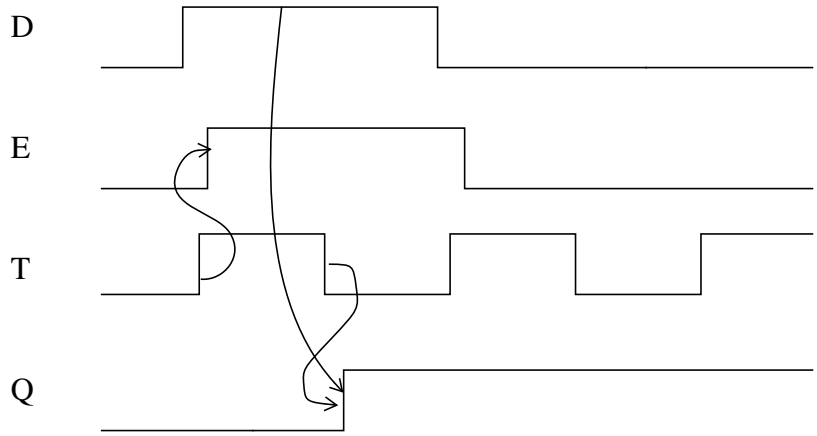



Bild 23: Das Timing eines Registers

Der Enableeingang *E* eines Registers wird durch den Controller gesteuert, der diesen bei einer positiven Taktflanke des Taktsignals *T* verändert. Ist der Eingang *E* auf ‚1' gesetzt, so werden die an *D* anliegenden Daten bei der folgenden negativen Taktflanke in das Register übernommen und damit auf den Ausgang *Q* des Registers geschaltet. Der Enable-Eingang *E*, welcher durch den Controller angesteuert wird, ist wichtig, da die an den Eingängen der Register befindlichen Daten aus verschiedenen ALUs stammen können, und umgekehrt jede ALU mehreren Registern vorgeschaltet werden kann, von denen nur eins das Ergebnis übernehmen soll.

**Tabelle 2: Die Ansteuerung der Bauteile durch den Controller**

| Kontroll-schritt | M1 | M2 | M3 | M4 | M5 | M6 | R1 | R2 | R3 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |

An dieser Stelle ist der erste Nachteil dieser Methode erkennbar: der eigentlichen Berechnung der Zwischenergebnisse wird nur die Länge eines halben Kontrollschrittes zur Verfügung gestellt. Dies kommt dadurch zustande, daß die Steuerleitungen vor den Taktflanken an den entsprechenden Registern anliegen müssen. Das heißt, ein Kontrollschritt dauert solange, bis alle Berechnungen in ihm abgeschlossen sind. Benötigt ein Addierer im Beispiel drei Taktschritte für eine Addition, so dauern der erste und zweite Kontrollschritt jeweils drei Taktschritte. Benötigt andererseits eine Multiplikation z.B. acht Taktschritte, so muß der zweite und dritte Kontrollschritt im Beispiel auf acht Taktschritte erweitert werden. Die Übernahme der errechneten Ergebnisse durch die Register darf dann nur im letzten Takt eines Kontrollschrittes geschehen. Das Ziel ist natürlich, einen Kontrollschritt so zu spezifizieren, daß er genau einem Taktschritt entspricht. Hierzu muß die Information über die Verzögerungszeit der einzelnen Bauteile vorliegen und mit berücksichtigt werden. Der Ansatz, zuerst ein Scheduling und dann das Assignment durchzuführen, erweist sich als ungünstig, denn es werden auf diese Weise nicht alle Möglichkeiten genutzt. Viele schnelle Bauteile liegen auf diese Weise die meiste Zeit brach. Das sollte vermieden werden. Die genannten Nachteile gelten für alle auf diesen Methoden auf-

setzenden Scheduling Algorithmen. Das erste Ziel dieser Arbeit ist also, eine Methode zu entwickeln und vorzustellen, die das Assignment, also die Zuordnung der Befehle zu den Bauteilen, mit in den Scheduling-Algorithmus einbezieht.

## 5.7 Optimierungsmöglichkeiten

Um möglichst optimale Ergebnisse für die Synthese zu erhalten, muß der Platzbedarf berücksichtigt werden. Durch die Wahl eines Scheduling-Algorithmus, welcher auf den oben beschriebenen Algorithmen ASAP und ALAP aufbaut, kann garantiert werden, daß die kürzeste Ausführungszeit erreicht wird, da der Zeitbedarf oder besser der Kontrollschrittbedarf festgelegt wird. Zu optimieren ist der Ressourcenaufwand. In Bild 22 wurde schon eine Optimierung durchgeführt: die dem Register *R1* zugeordneten Variablen wurden jeweils durch den Addierer 1 berechnet. Deshalb brauchte trotz zweimaliger Zuweisung an dieses Register kein Multiplexer vorgeschaltet werden. Das heißt, bei der Zuweisung der Variablen an die Register und bei der Zuweisung der Befehle an die Bauteile muß darauf geachtet werden, daß die Anzahl der Multiplexer minimiert bzw. ein Kompromiß zwischen einer minimalen Anzahl von Registern und Multiplexern gefunden wird. Es kann also nicht garantiert werden, das optimale Syntheseergebnis zu finden. Einzelne Schritte der Synthese sind schon NP-vollständig. Es kann angenommen werden, daß - falls die polynomielle Hierarchie echt ist - der gesamte Syntheseprozeß noch komplexer ist, also mindestens $\Sigma_2$-vollständig. Für einen gewissen Teilbereich der Synthese, und zwar für die Schaltkreisminimierung, das 'minimum equivalent circuit problem', ist in [114] die $\Sigma_2$-vollständigkeit bewiesen. Im folgenden werden Transformationen auf der algorithmischen Ebene beschrieben, die zu Optimierungen führen. Diese Optimierungsmöglichkeiten werden nicht von jedem Synthesetool unterstützt.

### 5.7.1 Constant Propagation

Beim Constant Propagation Verfahren werden Berechnungen, die zu dem gleichen Ergebnis führen, nur einmal durchgeführt. In dem Beispiel aus Bild 13 kann eine Optimierung für die Synthese dadurch erfolgen, daß die Addition *A+C* nur einmal ausgeführt wird. Die VHDL Beschreibung dazu steht in Bild 24, wo die zusätzliche Variable *Z* eingefügt ist.

```
-- Architektur Deklaration --
architecture H_L_2 of SYNEX1 is
begin
Z := A + C
X := E * (B + Z);
Y := Z * (C + D);
end H_L_2;
```

Bild 24: Optimierung des Codes durch Mehrfachnutzung einer Berechnung

Der Datenflußgraph für dieses Beispiel ist in Bild 25 dargestellt, aus der ersichtlich ist, daß eine Addition weniger als bisher nötig ist. Führt man hierfür den ASAP Algorithmus durch, so zeigt sich, daß ein Addierer eingespart werden kann. Dieser Optimierungsschritt kann automatisiert werden, wobei dies nicht in allen Synthesewerkzeugen unterstützt wird, und der Entwerfer entsprechende Optimierungen vornehmen muß.

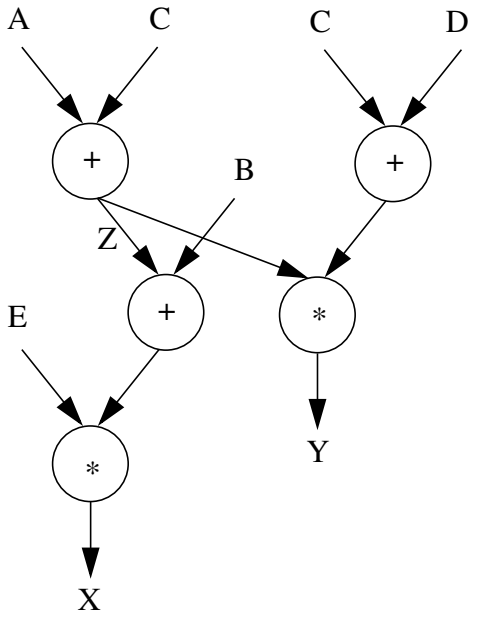



Bild 25: Der Datenflußgraph für H_L_2

### 5.7.2 Tree Height reduction

Um weitere Optimierungen auf der Ebene des Quelltextes durchführen zu können, müssen zusätzliche Informationen zu jeder Operation vorhanden sein, also ob sie z.B. assoziativ, distributiv oder kommutativ sind. Der Datenabhängigkeitsbaum, welcher sich zur Darstellung einer Formel ergibt, sollte balanciert, also die Tiefe der Teilbäume in etwa gleich sein [82], da dieser potentiell die schnellsten Schaltungen ermöglicht. Soll z.B. die Befehlszeile in Bild 26 synthetisiert werden, so ist der Aufbau des Datenflußgraphen abhängig von der Klammersetzung; im Beispiel wird eine Klammersetzung impliziert.

-- Architektur Deklaration --

*architecture H_L_3 of SYNEX1 is*

*begin*

*Y := A + B + C + D + E;*

*end H_L_2;*

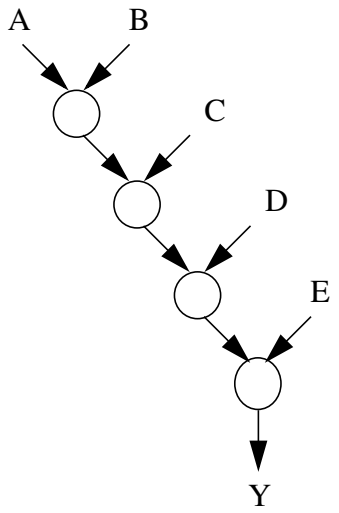



Bild 26: Beispiel für einen unbalancierten Datenflußgraphen/baum

Bild 27 zeigt dann, wie die gleiche Formel anders geklammert werden kann, so daß ein balancierter Baum entsteht.

Durch die Umstellung der Klammern entsteht ein Baum, dessen Tiefe um eins reduziert ist.

-- Architektur Deklaration --

*architecture H_L_3 of SYNEX1 is*

*begin*

*Y := (A + B + C) + (D + E);*

*end H_L_3;*

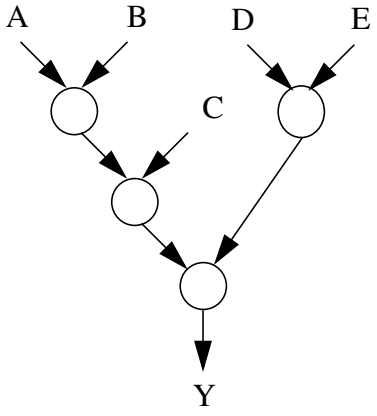



Bild 27: Beispiel für einen balancierten Datenflußgraphen/baum

Hierdurch erhöht sich die durch den Datenflußgraphen vorgegebene Parallelität, und die Erzeugung schnellerer Designs ist möglich. Es geht bei dieser Optimierung also um die potentielle Möglichkeit eine schnellere Schaltung zu entwerfen. In den weiteren Schritten der Synthese wird dann entschieden ob die Beschleunigung mit dem Ressourcenbedarf verträglich ist. Situationen wie diese müssen schon sehr früh vom Synthesetool oder vom Entwerfer der Schaltung erkannt werden.

### 5.7.3 Spekulatives Scheduling

Diese Methode stellt keinen gesonderten Schedulingalgorithmus dar, sondern kann in Kombination mit jedem Schedulingalgorithmus verwendet werden, weshalb sie im Rahmen der allgemeinen Optimierungsmöglichkeiten dargestellt wird. Ein wesentlicher Punkt bei der Optimierung der Synthese, speziell des Schedulings, erscheint in den nicht standard Situationen, die nicht ganz so einfach in das oben dargestellte Schema einzuordnen sind. Hierzu gehören Befehlsfolgen, die Schleifen und bedingte Anweisungen enthalten. Ein Schedulingalgorithmus muß in diesen Fällen weitere Abhängigkeiten, die teilweise erst zur Laufzeit feststehen, berücksichtigen können. Dies führt zu modifizierten Algorithmen, die nicht in die Menge der Standartalgorithmen eingeordnet werden können. Stehen genügend Hardwarekomponenten zur Verfügung, so kann mit einem System, wie es in [61] vorgestellt ist, spekulatives Scheduling durchgeführt werden. Sowohl bei bedingten Schleifen, also Schleifen deren Laufzeit abhängig von einem booleschen Wahrheitswert ist, als auch bei bedingten Ausdrücken, wie beispielsweise einer if-Anweisung, werden einige Befehle spekulativ zu einem frühen Zeitpunkt ausgeführt, wobei noch nicht bekannt ist, ob die Ergebnisse genutzt werden. Die Entscheidung über die Nutzung der Ergebnisse wird erst nach der Auswertung der Bedingung möglich. Somit kommt es dazu, daß einige Befehle überflüssigerweise ausgeführt werden. Durch die spezielle Darstellung der einzelnen Befehle als elementare Befehle ist das spekulative Scheduling in dem hier gewählten Ansatz impliziert. Um ein Synthesesystem zu entwickeln, wird aus der großen Anzahl an Auswahlmöglichkeiten im Rahmen dieser Arbeit ein Ansatz mit genetischen Algorithmen untersucht.

### 5.7.4 Nutzung verschiedener und multifunktionaler Bauteile

Ein wesentlicher Nachteil der bisher dargestellten Systeme ist, daß Bauteile bzw. Befehle mit unterschiedlicher Verzögerungszeit gleich behandelt werden. Anhand eines Beispiels soll dies

verdeutlicht werden. Die Multiplikation hat einen viel höheren Zeit- und Platzbedarf als eine Addition. Der Zeitbedarf eines Kontrollschrittes wird an den Befehl angepaßt, welcher den höchsten Zeitbedarf hat, also in dem Beispiel an den Zeitbedarf einer Multiplikation. Im Rahmen der Arbeit soll diese Einschränkung aufgehoben werden, wie dies auch schon in anderen Systemen geschieht [68]. Außerdem wird die Nutzung multifunktionaler Bauteile, wie z.B. der ALUs, bisher nicht oder nur selten berücksichtigt. Für ein und denselben Befehl gibt es verschiedene Möglichkeiten der Implementierung. Beispielsweise kann eine Addition mit einem langsamen sequentiellen Addierer oder einem Carry-Look-Ahead Addierer ausgeführt werden. Diese Unterschiede werden ebenfalls selten miteinbezogen. Im Rahmen dieser Arbeit wird die Nutzung multifunktionaler Bauteile mit Pipeline-Verarbeitung - wobei jede Funktion einen unterschiedlichen Zeitbedarf haben kann - in die Synthese einbezogen. Die Ausführungszeit für einen Befehl kann über einen Kontrollschritt hinausgehen, wobei ein Kontrollschritt einem Taktschritt entspricht. Die Länge eines Taktschrittes wird daher dem schnellsten vorhandenen Bauteil und nicht wie in anderen Systemen dem langsamsten angepaßt. Der Vorteil dieser Methode besteht darin, daß schnelle Bauteile optimal ausgenutzt werden, und es nicht zu 'unnötigen' Verzögerungen einzelner Befehle kommt. Um andererseits technologieunabhängig zu bleiben, können die Ausführungszeiten eines Befehls in Form von Gatterlaufzeiten angegeben werden, wie es bei Komplexitätsberechnungen üblich ist [112]. Dadurch kann die Länge eines Taktschrittes als eine gewisse Anzahl von Gatterlaufzeiten angegeben werden. Durch diese Darstellung der Bauteile werden viel mehr unterschiedliche Implementierungsmöglichkeiten berücksichtigt, was den Lösungsraum erheblich vergrößert. Hierdurch wird die für die Synthese benötigte Rechenzeit erhöht. Manche Algorithmen können nicht mehr oder nur bedingt angewandt werden. Um die Rechenzeit nicht eskalieren zu lassen, werden im folgenden genetische Algorithmen angewandt, die zu ihrer Beschleunigung in einem Cluster von Workstations verteilt ausgeführt werden können.

### 5.8 Synthetisierte Architektur

Um verschiedene Synthesesysteme und Algorithmen vergleichen zu können, sollte das Ergebnis aller zu vergleichenden Systeme äquivalent sein. Die synthetisierte Architektur wird im wesentlichen durch das zur Verfügung stehende System bestimmt. Die meisten in dieser Arbeit verfolgten Ansätze sind so allgemeingültig, daß die Architektur, die daraus synthetisiert werden soll, relativ frei gewählt werden kann. Im wesentlichen kann zwischen einer Multiplexerarchitektur und einer Busarchitektur unterschieden werden, wobei aber auch systolische Systeme [36] und spezielle Speicherstrukturen [22] und andere [10] betrachtet werden müssen. Die Unterscheidung zwischen Multiplexer- und Busarchitektur bezieht sich auf die unterschiedliche Ausführung der Verbindungsstrukturen. In einer Multiplexerstruktur, wie sie z.B. in Bild 22 schon dargestellt ist, werden mehrfach genutzte Eingänge durch die Vorschaltung eines Multiplexers vervielfacht. Andrerseits können die mehrfach genutzten Eingänge von Bauteilen durch einen Bus, auf den mit mehreren Bauteilen zugegriffen werden kann, angesteuert werden. Alle Ausgänge der Bauteile und Register müssen bei dieser Lösung als Tri-State Ausgänge ausgeführt sein, damit keine Zugriffskonflikte entstehen. Wird eine Busarchitektur gewählt, so gibt es hier wiederum die Optimierungsmöglichkeit, daß die Busse durch mehrere Ein- und Ausgänge der Bauteile genutzt werden. Hierbei muß das Scheduling berücksichtigt werden, so daß jedem schreibenden Ausgang auch ein Bus zur Verfügung steht. Das Thema der Verbindungsstruktursynthese wird in dieser Arbeit nicht weiter vertieft. Wie schon oben angedeutet sind die meisten in dieser Arbeit vorgestellten Ansätze von der Architektur unabhängig. Um aber einerseits Vergleichsmöglichkeiten und andrerseits konkrete Ergebnisse der Synthese zu erzielen, wurde hier eine Multiplexer-basierte Architektur gewählt. Besonders zu erwähnen ist, daß der Controller durch ein einfaches ROM realisiert werden kann, wobei die Adressierung

des ROMs durch einen Zähler geschieht. Die Optimierung des Controllers wird hier nicht weiter betrachtet [37].

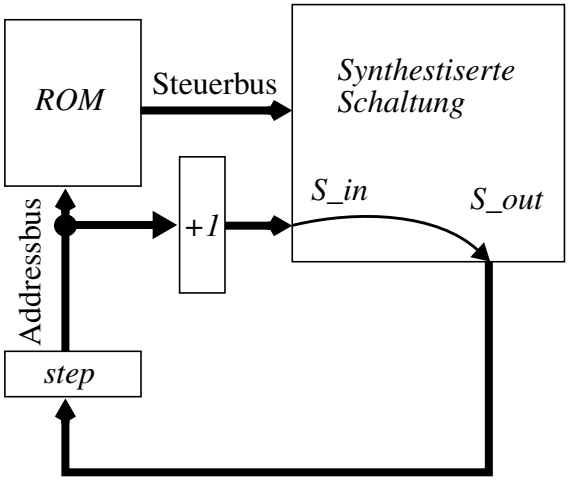



Bild 28: Synthetisierte Architektur

In der Bild 28 stellt die Variable *step* den Programmzähler dar, der in diesem Fall nicht durch den Controller sondern durch die synthetisierte Schaltung beeinflußt wird. Bei einem normalen Ablauf des Programms wird der Eingang *S_in* direkt wieder nach *S_out* durchgeschaltet. Bei einem Sprungbefehl wird aber der Ausgang *S_out* mit dem entsprechenden Sprungziel belegt, so daß der Zähler nicht um einen Schritt weiter gezählt, sondern direkt auf das Sprungziel gesetzt wird. Intern wird diese Umschaltung durch einen Multiplexer bewerkstelligt, wie das in Bild 29 dargestellt ist. Anhand der Bedingung wird entweder der normale Programablauf weitergeführt, oder es kommt zu einem Sprung.

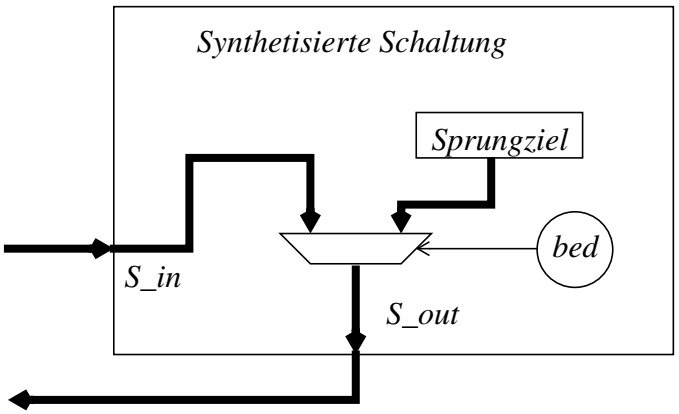



Bild 29: Realisierung eines Sprungbefehls

Eine so synthetisierte Schaltung ist so aufgebaut, daß sie einerseits unabhängig funktionsfähig ist oder wieder als Bauteil in eine andere Schaltung übernommen werden kann, wodurch eine hierarchische Synthese ermöglicht wird. Nachdem nun verschiedene Möglichkeiten der Optimierung angesprochen wurden, wird im nächsten Abschnitt auf die Akzeleration der Synthese eingegangen.

## 6 Die Akzeleration der Synthese durch Partitionierung

Aus der vorangegangenen Betrachtung ist deutlich geworden, daß die Synthese vor allem großer Schaltungen im besonderen dann sehr rechenintensiv ist, wenn akzeptable Ergebnisse erzielt werden sollen. Der Syntheseablauf ist dadurch gekennzeichnet, daß viele Schritte, die

sequentiell ausgeführt werden, einzeln betrachtet zwar gute Ergebnisse liefern, jedoch in ihrer Gesamtheit so voneinander abhängig sind, daß das Gesamtergebnis nicht optimal ist. Somit sind mehrere Iterationen der Synthese und viele Optimierungsschritte notwendig. Das Ziel ist nun, Techniken zu finden, die die Synthese beschleunigen, wobei die Qualität der Syntheseergebnisse erhalten bleibt. Ein wesentlicher Schritt ist die Partitionierung von Modellen in mehrere Submodelle, die unabhängig voneinander parallel synthetisiert werden können. Durch diesen Ansatz sollen in einem Cluster zur Verfügung stehende Workstations besser genutzt werden. In Bild 30 ist der Ablauf der durch Partitionierung ermöglichten parallelen Synthese dargestellt.

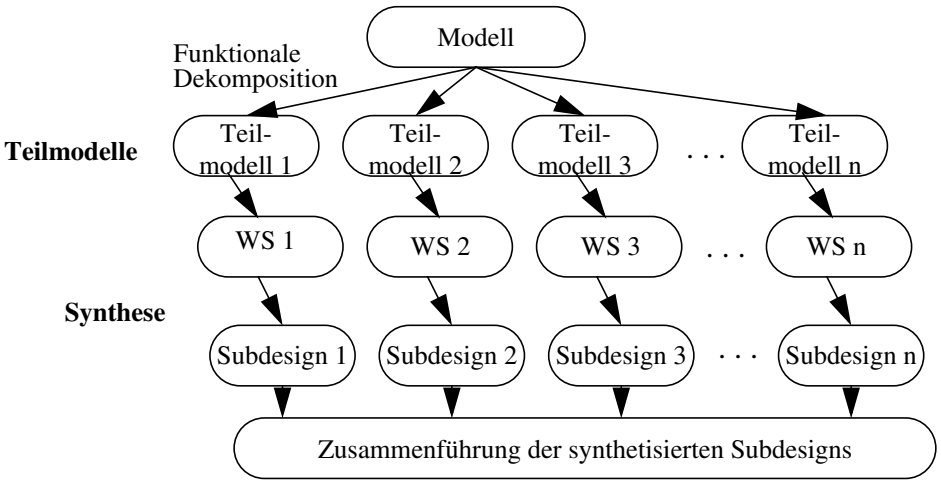



Bild 30: Prinzip der parallelen Synthese durch Partitionierung

Durch die Partitionierung eines Modells werden in einer ansonsten flachen Struktur Hierarchieebenen eingefügt. Als Beispiel sei das Modell, welches in Bild 31 dargestellt ist, gegeben. Es besteht aus sieben Befehlen, deren Abhängigkeiten als Kanten dargestellt sind.

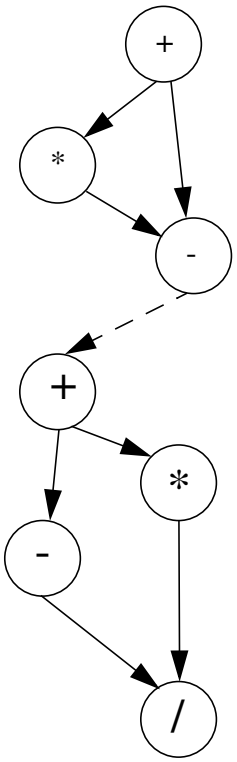



Bild 31: Datenflußgraph mit Partitionierungskante

Die Partitionierung soll nun an der gestrichelt dargestellten Kante durchgeführt werden. Durch die Partitionierung werden die drei oberen Befehle zu einer Funktion $f_1$ und die vier unteren Befehle zur Funktion $f_2$ zusammengefaßt. In Bild 32 ist dann links der resultierende Graph mit den beiden Funktionen $f_1$ und $f_2$ dargestellt, wobei rechts noch einmal die Graphen der Funktionen selber dargestellt sind.

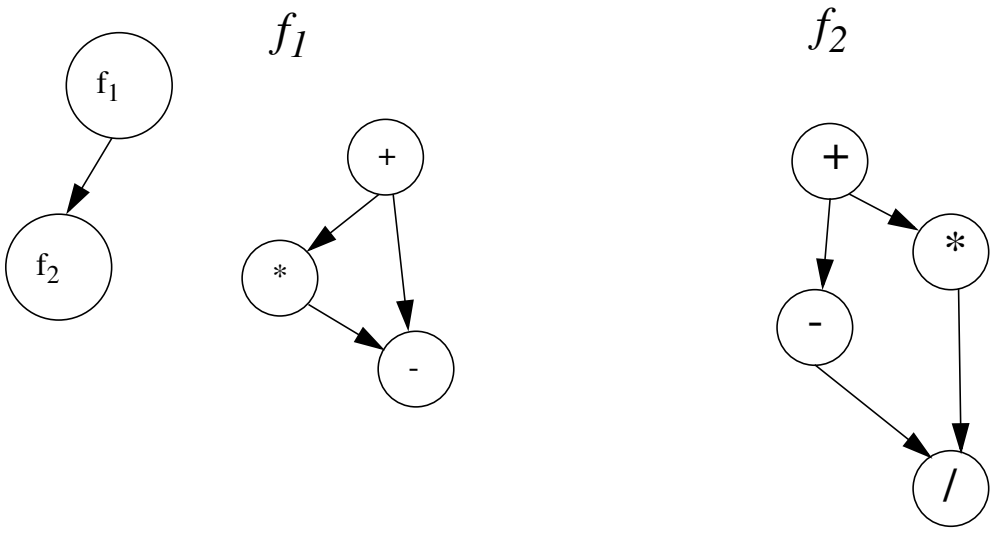



Bild 32: Ein Modell wird in zwei Teilmodelle partitioniert

An dieser Stelle soll nicht weiter auf die Partitionierung von Modellen eingegangen werden. Der Vorteil einer Partitionierung ist offensichtlich, denn die entstandenen Teilfunktionen können unabhängig voneinander synthetisiert werden. Da die Komplexität der Synthese bei den meisten Verfahren mindestens kubisch ist, macht sich eine Partitionierung positiv bemerkbar. Der Zeitbedarf der Synthese ist durch die Anzahl der zur Verfügung stehenden Workstations bestimmt. Gerade für große Modelle ist dies eine Möglichkeit, eine hohe Beschleunigung zu erreichen. Das Problem bei der unabhängigen Synthese ist, daß Ressourcen - also zur Verfügung gestellte Bauteile - nicht gemeinsam von den synthetisierten Schaltungen genutzt werden. Eine andere Möglichkeit der Akzeleration der Synthese ist die Benutzung von Algorithmen, die sich besonders gut für eine Parallelisierung eignen. Im Rahmen dieser Arbeit wird daher die Eignung genetischer Algorithmen für die Synthese untersucht [40]. Genetische Algorithmen sind optimal für die Parallelisierung in einem Workstationcluster geeignet. Da es sich bei den genetischen Algorithmen um zufallsbasierte Methoden handelt, wird im folgenden Abschnitt kurz auf einige zufallsbasierte Optimierungsalgorithmen eingegangen, um dann zu den genetischen Algorithmen überzuleiten.

# 7 Zufallsbasierte Optimierungsalgorithmen

Es gibt verschiedene Möglichkeiten, an diese Problemstellung heranzugehen. Eine Entscheidungshilfe gibt dabei die Komplexitätstheorie [114], mit deren Hilfe ein Problem in eine Komplexitätsklasse eingeordnet werden kann. Die Klasse *P* beinhaltet die in polynomiell beschränkter Zeit deterministisch berechenbaren Probleme. Kann für ein Problem gezeigt werden, daß es in *P* liegt, so sollte versucht werden, einen deterministischen Algorithmus hierfür zu finden. Die Klasse *NP* beinhaltet alle nicht-deterministisch polynomiell berechenbaren Probleme. Es bisher nicht möglich gewesen zu zeigen, ob sie der Klasse *P* entspricht. Es wird vermutet, daß $NP \neq P$ gilt, wobei aber zumindest $P \subseteq NP$ gilt. Die Klasse der *NP*-vollständigen Probleme ist diejenige Klasse, auf die alle Probleme aus *NP* polynomiell reduzierbar sind, damit enthält sie Probleme, die (vermutlich) nicht in *P* liegen. Es kann also nicht davon ausgegangen

werden für *NP*-vollständige Probleme deterministische polynomialzeit-Algorithmen zu finden. Häufig werden heuristische oder probabilistische Algorithmen formuliert, die zwar nicht immer das optimale, aber dennoch ein für die praktische Anwendung 'brauchbares' Ergebnis finden. Eine Klasse der probabilistischen Algorithmen ist die der Optimierungsalgorithmen, bei denen von einer Lösung des Problems ausgegangen wird, welches solange verbessert wird, bis die Güte der Lösung den Bedingungen genügt. Im folgenden wird kurz auf den bekannten Algorithmus Simulated Annealing, welcher auf dem Metropolis Algorithmus [78] beruht, und die daran angelehnten Verfahren Threshold-Algorithmus und Sintflut-Algorithmus eingegangen. Anschließend wird ein genetisches Verfahren beschrieben, welches in dieser Arbeit Anwendung findet. Da genetische oder evolutionäre Algorithmen [79] hier nicht intensiv behandelt werden sollen, sei auf [83], wo ein sehr guter und umfangreicher Überblick über evolutionäre Algorithmen gegeben wird, und auf [57] verwiesen; der Verfasser kommt schnell zu den wesentlichen Dingen der genetischen Algorithmen.

### 7.1 Simulated Annealing

Das Verfahren des Simulated Annealing basiert darauf, eine gegebene Lösung geringfügig zu verändern. Ist die so entstandene Lösung höherwertig bzgl. einer Gütefunktion, so wird mit der neuen Lösung weiter gerechnet, also wieder von vorne begonnen. Ist die Lösung schlechter als die alte, so wird mit der alten Lösung weitergerechnet oder die neue mit einer geringen Wahrscheinlichkeit übernommen.

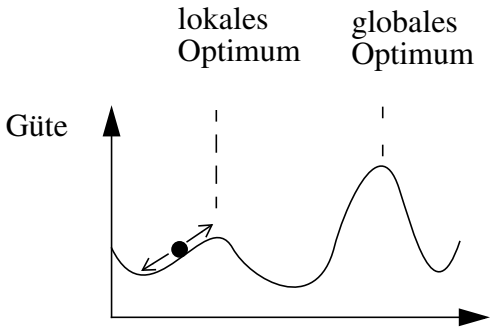



Bild 33: Gütefunktion mit Lösung

Am besten kann dies anhand eines ‚Gütegebirges' wie in Bild 33 verdeutlicht werden. Die Kugel visualisiert die Güte der aktuellen Lösung. Eine Veränderung der Lösung stellt sich in einer Verschiebung der Kugel nach links oder rechts dar. Das Optimum sei der höchste Punkt des Gebirges. Das lokale Optimum wird durch mehrere Veränderungen erreicht. Das globale Optimum kann nicht durch geringe Veränderungen erreicht werden außer in dem Fall, daß eine schlechtere Lösung zugelassen wird, so daß das Tal übersprungen werden kann.

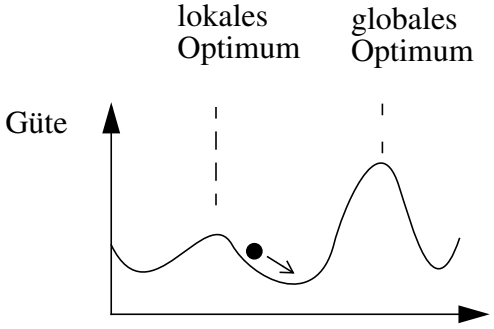



Bild 34: Schlechtere Lösung wird zugelassen

Bild 34 zeigt, daß die Kugel erst in das Tal muß, also eine schlechtere Lösung zugelassen wird, um dann den Weg zum globalen Optimum antreten zu können, was in Bild 35 dargestellt ist.

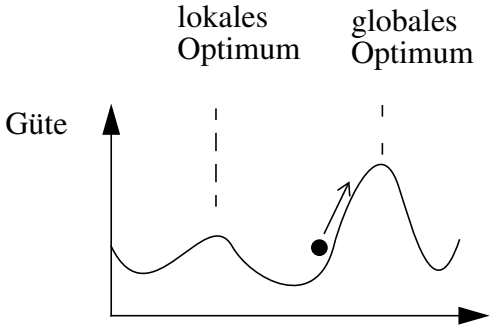



Bild 35: Es geht wieder aufwärts

Anhand dieser kurzen Darstellung wird schon die Problematik aller Optimierungsalgorithmen deutlich. Es kann sehr schnell passieren, daß der Algorithmus ein lokales Optimum erreicht, aber das eigentliche globale nicht findet. Bei dem Simulated Annealing Algorithmus werden schlechtere Lösungen nur manchmal zugelassen und ansonsten verworfen, z.B. mit der Wahrscheinlichkeit von 0.1.

### 7.2 Der Threshold Algorithmus

Der Threshold Algorithmus [33] basiert nicht darauf, daß zufällig eine schlechtere Lösung zugelassen wird, sondern er läßt schlechtere Lösungen zu, wenn ihre Güte noch in einem akzeptablen Abstand zur Güte der alten Lösung liegt. Das heißt ein gewisser Threshold-Bereich zur Güte der alten Lösung darf nicht überschritten werden. In Bild 36 wird die durch die weiße Kugel dargestellte Lösung nicht akzeptiert, da ihr Abstand zur alten Lösung - als schwarze Kugel dargestellt - zu groß ist.

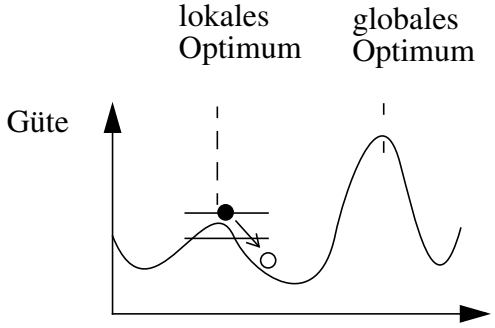



Bild 36: Lösung wird nicht akzeptiert

Das Tal muß mit einem Zwischenschritt überwunden werden. In Bild 37 gelangt die Kugel dadurch bis ins Tal, daß mehrere Male hintereinander die Lösung so verändert wurde, daß sie noch akzeptiert wird.

### 7.3 Sintflut Algorithmus

Bei dem Sintflut Algorithmus [33] kann sich die Lösung am Anfang frei im ‚Gütegebirge' bewegen Bild 38; der Pegel steigt aber langsam an, unter dem keine Lösung mehr akzeptiert wird.

In Bild 39 ist die ‚Flut' schon so weit gestiegen, daß die Lösung sich nicht mehr vom lokalen Optimum wegbewegen kann. Der Algorithmus liefert dann das Ergebnis, wenn die Lösung ‚er-

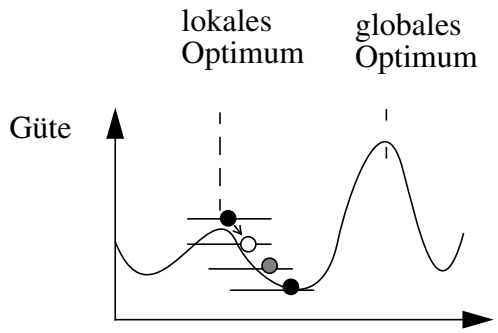



Bild 37: Lösungen werden akzeptiert

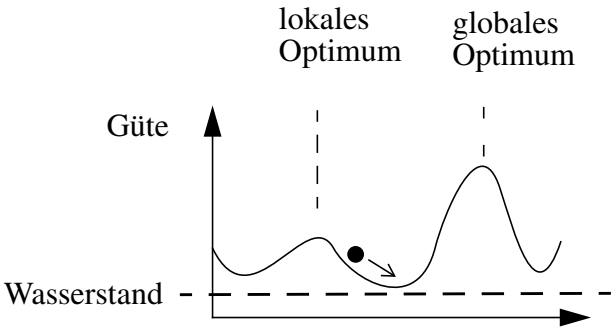



Bild 38: Lösung kann sich noch frei bewegen

trunken‘ ist. (Hier wird die überlebende Lösung, obwohl sie keine Arche hat, wenigstens gerettet, was bei der Original-Sintflut [56] ja schließlich nicht der Fall war).

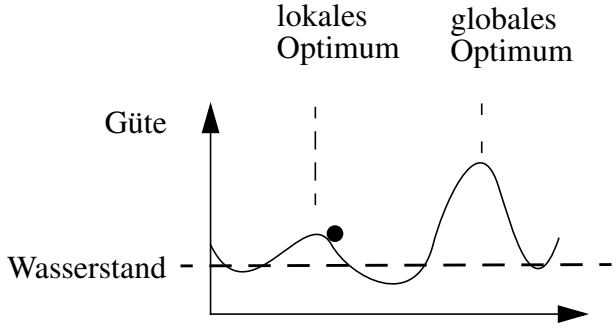



Bild 39: Gefangen im lokalen Optimum

Die Problematik dieser Algorithmen ist, daß sie immer nur mit einer Lösung operieren. Selbstverständlich ist es möglich, parallel mit mehreren Lösungen zu starten; einmal gewonnene Erkenntnisse werden nicht untereinander weitergegeben. Die Lösungen werden völlig unabhängig voneinander optimiert und dann das beste Ergebnis ausgewählt. Der Ansatz, der durch diese Algorithmen gegeben ist, wird mit genetischen oder evolutionären Algorithmen erweitert. Auch bei genetischen Algorithmen wird mit mehreren Lösungen gestartet, wobei aber der wesentliche Unterschied darin besteht, daß verschiedene Lösungen ihre ‚Erfahrungen‘ weitergeben können. Das heißt, es werden nicht nur Lösungen unabhängig voneinander verändert, sondern es werden auch verschiedene Lösungen zu neuen Lösungen vermischt.

## 8 Genetische Algorithmen

Die in der Biologie vorherrschende Theorie über die Entstehung des Lebens auf der Erde ist die

von Darwin [29] etablierte Idee der Evolution. Der Gedanke einer sich durch Evolution entwikkelten Natur geht zum Teil schon auf das griechische Denken zurück. Dies trifft zum Beispiel auf die Lehrsätze des frühen ionischen Philosophen Anaximander (ca. 610 -ca. 540 v. Chr.) zu, der sich alles aus dem Urprinzip des ‚Unbestimmten' entstanden dachte und glaubte, daß das Leben auf einer feuchten Erde entstanden sei, als diese durch die Sonne erwärmt wurde [86]. Die Idee von Darwin hat sich in den Köpfen vieler Biologen und Historiker festgesetzt. Darwin erklärt den Vorgang der Evolution als einen stufenweisen Prozeß der Höherentwicklung aus einem Zusammenspiel von Variation (zufälliger Abweichung) und Selektion (natürliche Zuchtwahl). Zufällige Abweichungen unter den Nachkommen führen zu unterschiedlicher Tauglichkeit. Die natürliche Auslese setzt an dieser unterschiedlichen Tauglichkeit der Individuen an. Die Bestangepaßten haben eine höhere Chance, ihre Erbanlagen weiterzugeben (Survival of the fittest). Ausgehend von diesen Ideen haben Rechenberg [94] und Schwefel [96] die evolutionären Algorithmen zuerst angewandt, wobei Holland [60] , De Jong [62] und Goldberg [51] den Begriff der genetischen Algorithmen geprägt haben. In der Literatur wird im wesentlichen zwischen genetischen Algorithmen, genetischer Programmierung, evolutionären Algorithmen und evolutionärer Programmierung unterschieden. Für die genaue Darlegung und Unterscheidung der einzelnen Methoden und Richtungen verweise ich auf entsprechende Literatur [83]. Im folgenden wird die hier benutzte Definition genau dargelegt und der Begriff ‚genetischer Algorithmus' benutzt.

## 8.1 Einführung

Genetische oder Evolutionäre Algorithmen werden im wesentlichen dort angewandt, wo es gar nicht oder nur unter großem Ressourceneinsatz möglich ist, eine gute Lösung für ein gegebenes Optimierungsproblem zu finden. Es handelt sich hierbei um die Klasse der NP-vollständigen und noch schwierigeren Optimierungsprobleme [63]. Oft ist es sehr einfach, bei dieser Klasse der Probleme überhaupt eine Lösung zu finden, aber der Lösungsraum ist so groß, daß nicht garantiert werden kann, auch die beste Lösung zu finden. Es gibt konstruktive Ansätze, die meistens mit heuristischen Mitteln genau eine Lösung berechnen. Andrerseits gibt es Optimierungsverfahren, die von einer gegebenen Lösung ausgehen, welche als solche natürlich erst konstruiert werden muß, und sie dann in mehreren gleichartigen Zwischenschritten optimieren. Die veränderbaren Parameter werden dabei meistens zufällig ausgewählt, und der Grad der Veränderung ist bei jedem Schritt relativ gering. Verfahren, wie Simulated Annealing oder Threshold Algorithmen, liefern in den meisten Fällen zufriedenstellende Ergebnisse. Ein wesentlicher Arbeitsaufwand, der für jedes Problem erneut geleistet werden muß, ist die Darstellung der Problems, so daß Veränderungen überhaupt erst durchführbar sind. Des weiteren ist die Frage der Bewertung der Lösungen zu klären. Oft kann zwar eine Veränderung einfach durchgeführt werden, aber es bedarf doch eines hohen Rechenaufwandes, die so konstruierte Lösung zu bewerten. Das heißt, auch bei Optimierungsverfahren wie Simulated Annealing [1] bedarf es einer hohen Rechenzeit. Durch genetische Algorithmen wird der Ansatz, der durch Simulated Annealing gegeben ist, erweitert. Neben der durch Simulated Annealing zur Verfügung gestellten Möglichkeit, eine gegebene Lösung zu verändern, wird auch das Mischen zweier vorhandener Lösungen als Möglichkeit betrachtet, eine bessere Lösung zu erhalten. Dadurch, daß diese Möglichkeit gewünscht ist, muß ein gewisser Aufwand in die Darstellung - oder Codierung der Lösungen - gesteckt werden, so daß sich der Mischvorgang einfach durchführen läßt. Das Mischen zweier Lösungen wird in Anlehnung an die Biologie Rekombination oder Crossover genannt, die Veränderung einer Lösung wird Mutation genannt. Im folgenden soll auf die Codierung einer Lösung, auf die genetischen Operatoren und auf die Bewertung einer Lösung eingegangen werden.

## 8.2 Codierung und Dekodierung

Wird mit einem genetischen Algorithmus ein Problem bearbeitet, so müssen die Lösungen in Form einer Codierung vorliegen, auf die dann die genetischen Operatoren angewandt werden können. Eine Codierung einer Lösung kann am besten in Form eines Strings dargestellt werden, der alle für eine eindeutige Unterscheidung der Lösung notwendigen Informationen enthält. Die folgende Definition macht dies deutlich:

*Definition 8.2-1 Sei ein Alphabet* $\Sigma$ *und eine Menge L gegeben, dann heißt die Abbildung*

$C: L \rightarrow \Sigma^n$ *Codierungsfunktion oder Codierung für einen Code der Länge n.*

*Definition 8.2-2 Die Codierungsfunktion heißt eindeutig, genau dann wenn für* $m_1, m_2 \in L$ *gilt*

$C(m_1) = C(m_2) \Leftrightarrow m_1 = m_2.$

Die Eindeutigkeit der Codierungsfunktion wird im folgenden vorausgesetzt, sie kann auch als charakteristische Eigenschaft der Codierungsfunktion betrachtet werden. Die Umkehrfunktion der Codierungsfunktion heißt Dekodierung:

*Definition 8.2-3 Die Dekodierung ist definiert als* $D: \Sigma^n \rightarrow L$ *mit* $D(C(m)) = m$ *für alle* $m \in L$.

Es existieren keine oder wenige Arbeiten über das generelle Vorgehen bei der Suche nach einem guten Code für eine gegebene Problemstellung. In den meisten Anwendungsfällen wird jedoch eine binäre Codierung gewählt, d.h $\Sigma = \{0, 1\}$, wobei man sich selbstverständlich auch andere Codierungen, z.B. mit ganzen oder reellen Zahlen vorstellen, oder irgendein anderes beliebiges Alphabet für die Codierung nutzen kann. In der Fachliteratur ist die binäre Codierung am intensivsten untersucht; im weiteren Verlauf dieser Arbeit wird die ganzzahlige Codierung benutzt.

## 8.3 Bewertung

Jede Lösung muß nach bestimmten, zu definierenden Gütekriterien bewertet werden. In den meisten Fällen heißt das, daß eine in Form des Codes vorliegende Lösung decodiert werden und eine Bewertung stattfinden muß. Diese Bewertung ist oft der aufwendigste Schritt eines genetischen Algorithmus. Die Bewertungsfunktion wird anhand der subjektiven oder objektiven Wünsche des Benutzers aufgestellt. Der Vorteil der genetischen Algorithmen ist, daß auch Mehrzieloptimierungen stattfinden können, d.h., es können verschiedene Gütekriterien in der Bewertungsfunktion berücksichtigt werden. Viele heuristische Algorithmen sind auf eine Einzieloptimierung festgelegt. Eine gewichtete Bewertungsfunktion kann dann wie folgt definiert werden.

*Definition 8.3-1 Die Funktionen* $g_1, \ldots, g_n: L \rightarrow \Re$ *heißen Gütekriterien wenn sie die Qualität einer Lösung* $l \in L$ *berechnen.*

*Definition 8.3-2 Die gewichtete Gesamtbewertung* $ges_w: L \rightarrow R$ *einer Lösung bezieht sich auf einen Gewichtsvektor* $w = (w_1, \ldots, w_n)$ *und ist definiert als*

$$ges_w(l) = \sum_{i=1}^{n} w_i g_i(l).$$

Durch die individuelle Wahl des Gewichtsvektors können mit ein und demselben genetischen Algorithmus verschiedene Bedürfnisse optimal befriedigt werden.

### 8.4 Selektion

Wurden alle vorhandenen Lösungen bewertet, so muß eine Selektion der am besten für die ‚Fortpflanzung' geeigneten Lösungen stattfinden. Diese Selektion richtet sich einzig und allein nach der Güte der einzelnen Lösungen. Es gibt verschiedene Selektionsmechanismen. Beispielsweise kann die Menge der Lösungen nach ihrer Güte sortiert werden, um dann die besseren Lösungen den genetischen Operatoren zur ‚Vermehrung' zuzuführen und die anderen zu löschen. Diese Methode wird im weiteren angewandt. Eine weitere Selektionsmethode besteht darin, jeweils zwei zufällig ausgewählte Lösungen miteinander zu vergleichen und die bessere 'überleben' zu lassen und somit den Lösungsraum solange zu reduzieren, bis die gewünschte Anzahl an Lösungen erreicht ist. Das gleicht einem Turnierschema, wie man es von der einen oder anderen Sportveranstaltung her kennt. Durch die zweite Methode der Selektion bleibt auf jeden Fall die beste Lösung erhalten, andererseits bleiben aber auch viele unterschiedliche Lösungen erhalten. Vor allem kann es aber passieren, daß die zweitbeste Lösung nicht 'überlebt', dafür aber die zweitschlechteste. Gerade am Anfang der Berechnungen eines genetischen Algorithmus ist es wichtig, noch viele unterschiedliche Lösungen zur Auswahl zu haben, um nicht in einem lokalen Optimum zu landen. Sind die Lösungen selektiert, so werden aus diesen Lösungen wieder neue erzeugt, so daß die ursprüngliche Anzahl Lösungen, die in einer Generation enthalten sind, wieder erreicht wird.

### 8.5 Genetische Operatoren

Als genetischer Operator kann eine Funktion bezeichnet werden, die aus einer Teilmenge der Lösungen bzw. deren Codes eine weitere Lösung bzw. dessen Code bildet.

*Definition 8.5-1 Ein genetischer Operator ist eine Funktion der Form* $gO:(\Sigma^n)^q \to \Sigma^n$.

Dadurch das die Codes betrachtet werden muß die in der folgenden Definition angegebene Bedingung für den genetischen Operator gelten, was nicht immer garantiert werden kann:

*Definition 8.5-2 Ein genetischer Operator gO heißt gültig genau dann wenn für beliebige* $m_1, \ldots, m_q \in L$ *gilt:* $D(gO(C(m_1), \ldots, C(m_q))) \in L$.

Das heißt also ein gültiger genetischer Operator ist über dem Bildbereich der Codierungsfunktion abgeschlossen. Diese Eigenschaft wird nicht von allen genetischen Operatoren erfüllt, so daß eine explizite Prüfung der entstandenen Lösung vorgenommen werden muß. Es werden im wesentlichen zwei Operatoren betrachtet und weiter differenziert. Einmal die Mutation, welche aus einer vorhandenen Lösung eine neue bildet ($q = 1$), und das sogenannte Crossover, welches aus zwei Lösungen eine neue produziert ($q = 2$). Im folgenden werden diese Operatoren genauer beschrieben.

### 8.6 Mutation

Die Mutation verändert eine gegebene Lösung in geringem Maße und produziert dadurch eine neue. Der Mutationsoperator wird nicht direkt auf eine Lösung sondern auf den Code der Lö-

sung angewandt. Bei der Definition des Mutationsoperators wird in dem hier dargestellten Fall davon ausgegangen, daß die Codes für alle Lösungen eine fixierte Länge haben. Es gibt genetische Algorithmen, bzw. Codes, die mit unterschiedlichen Längen arbeiten, was hier aber nicht weiter betrachtet werden soll.

*Definition 8.6-1 Die Mutation ist eine Funktion der Form* $m:\Sigma^n \to \Sigma^n$ *die aus einem Code einen neuen bildet, wobei maximal ein Buchstabe des Codewortes verändert wird.*

Diese Definition läßt offen wie der Buchstabe des Codewortes, der verändert wird, ausgewählt wird. In den meisten Anwendungen geschieht die Auswahl durch einen Zufallsgenerator. Durch die Anwendung der Codierung und Dekodierung erhält man die konkrete Lösung. Um beispielsweise aus einer Lösung $l_i$ durch Mutation eine neue Lösung $l_j$ zu berechnen, muß die Lösung codiert, mutiert und wieder dekodiert werden: $l_j = D(m(C(l_i)))$

Das Beispiel in Bild 40 zeigt die Mutation eines Strings, welcher die Lösung *l* codiert. Sie zeichnet sich dadurch aus, daß der String nur geringfügig, also nur an einer Stelle, verändert wird. Der mutierte Code wird nach der Mutation wieder decodiert, was zu einer neuen Lösung führt.

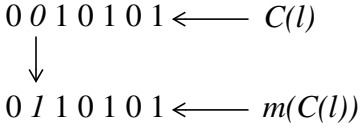



Bild 40: Mutation

Art und Umfang der Veränderung sind mehr oder weniger zufällig, meistens aber sehr gering. Durch eine geschickte Codierung soll erreicht werden, daß sich eine geringfügige Änderung in dem Code auch nur geringfügig auf die Lösung auswirkt. Zu beachten ist, daß der Mutationsoperator die Veränderungen so vornimmt, daß wieder eine Lösung erzeugt wird, der Mutationsoperator muß also der Bedingung der Gültigkeit entsprechen. Ist dies nicht der Fall so werden zusätzliche Mechanismen eingesetzt so daß nur Lösungen erzeugt werden. Das einfachste Verfahren eines genetischen Algorithmus ist nach dieser Definition der Simulated Annealing Algorithmus, der auf eine Lösung den Mutationsoperator anwendet und, falls eine Verbesserung des Ergebnisses vorliegt, die neue Lösung übernimmt und ansonsten mit der alten weitermacht. Es wird also bei diesem einfachen Algorithmus nur eine Lösung betrachtet und nur mit dem Mutationsoperator gearbeitet.

## 8.7 Crossover

Der wesentliche genetische Operator, welcher auch häufig genutzt wird, wird mit Rekombination oder Crossover bezeichnet; hierbei werden zwei Lösungen vermischt, um daraus eine neue Lösung zu bilden. Der Crossoveroperator wird sehr viel häufiger zur Erzeugung neuer Lösungen angewandt als der Mutationsoperator. Der Crossoveroperator bildet den wesentlichen Unterschied zu anderen Optimierungsalgorithmen, wie z.B. Simulated Annealing, welcher ja ebenfalls einen Mutationsoperator anwendet.

*Definition 8.7-1 Der Crossoveroperator auf der Ebene der Codierung ist eine Funktion der Form* $cr:\Sigma^n \times \Sigma^n \to \Sigma^n$.

Im ersten Schritt werden die beiden Lösungen, die vermischt werden sollen, codiert, dann wird der Crossoveroperator auf die Codes angewandt und somit der Code einer neuen Lösung produziert, und im letzten Schritt findet die Dekodierung statt. Also, wenn aus den Lösungen $l_i$ und

$l_j$ eine neue Lösung $l_k$ gebildet werden soll, dann sieht das folgendermaßen aus:

$D(cr(C(l_i), C(l_j))) = l_k$

Bei der Kreuzung von zwei Lösungen ist wiederum darauf zu achten, daß eine Lösung erzeugt wird. In Bild 41 wird das Verhalten des Crossoveroperators anhand eines ‚Gütegebirges' dargestellt. Die neue Lösung liegt irgendwo zwischen den beiden 'gekreuzten' Lösungen. Im Bild wird durch die Anwendung des Crossoveroperators fast das globale Optimum gefunden. Beide Lösungen allein hätten sich in mehreren Zwischenschritten auf das Optimum zubewegt.

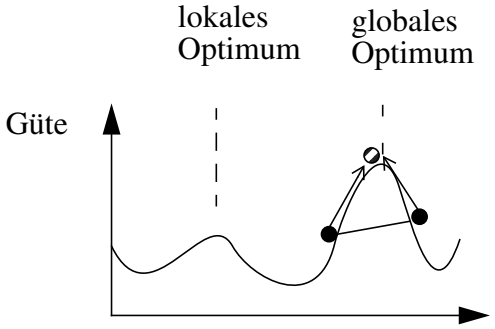



Bild 41: Crossover

Die Bild 42 zeigt ein Beispiel, in der die Codes der beiden Lösungen $l_1$ und $l_2$ mit Hilfe eines Crossoveroperators vermischt werden. Im Beispiel handelt es sich um das sogenannte Einpunkt-Crossover.

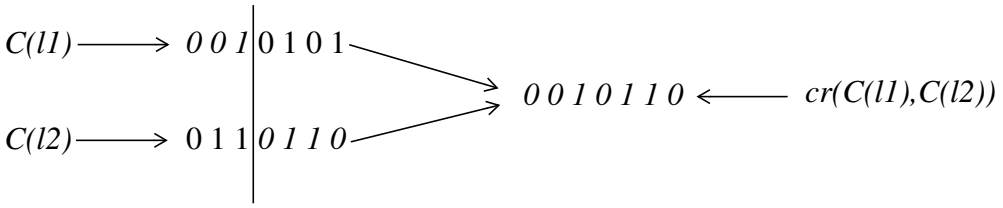



Bild 42: Der Crossoveroperator

Beim Einpunkt-Crossover muß ein Crossoverpunkt gewählt werden, anhand dessen entschieden wird, an welcher Stelle der Code aufgebrochen und mit dem entsprechenden anderen Teil der zweiten Lösung wieder zusammengesetzt wird. In Bild 42 werden die kursiv dargestellten Bits miteinander vermischt. Dabei behalten sie ihre Positionen im String bei. Der Code der so entstandenen neuen Lösung wird dekodiert, was in der Abbildung nicht mehr dargestellt ist. Falls für den Crossoveroperator die Bedingung der Gültigkeit nicht gegeben ist, muß für den erzeugten Code explizit überprüft werden ob es sich um eine Lösung handelt. Der dargestellte Crossoveroperator stellt nur eine der vielen Möglichkeiten dar, einen Crossoveroperator zu definieren. Statt den Crossoveroperator über den Crossoverpunkt zu definieren, kann auch eine Maske definiert werden, die für jedes einzelne Bit des Ergebnisses angibt, ob es von der ersten oder der zweiten Lösung genommen wird. Die Bild 43 stellt für diesen Fall den Crossovervorgang dar. Steht in der Maske eine 0, so wird von der ersten Lösung das entsprechende Bit genommen, wobei bei einer 1 in der Maske das entsprechende Bit von der zweiten Lösung in das Ergebnis übernommen wird.

Mit den definierten Operationen kann nun der Gesamtablauf eines genetischen Algorithmus wie in Bild 44 dargestellt werden.

*Maske :* 0 1 0 0 1 1 0

*C(l1)* ⟶ *0* 0 *1 0* 1 0 *1*

*C(l2)* ⟶ 0 *1* 1 0 *1 1* 0

*cr(C(l1),C(l2)* ⟶ *0 1 1 0 1 1 0*

Bild 43: Crossover mit Maske

### 8.8 Gesamtablauf

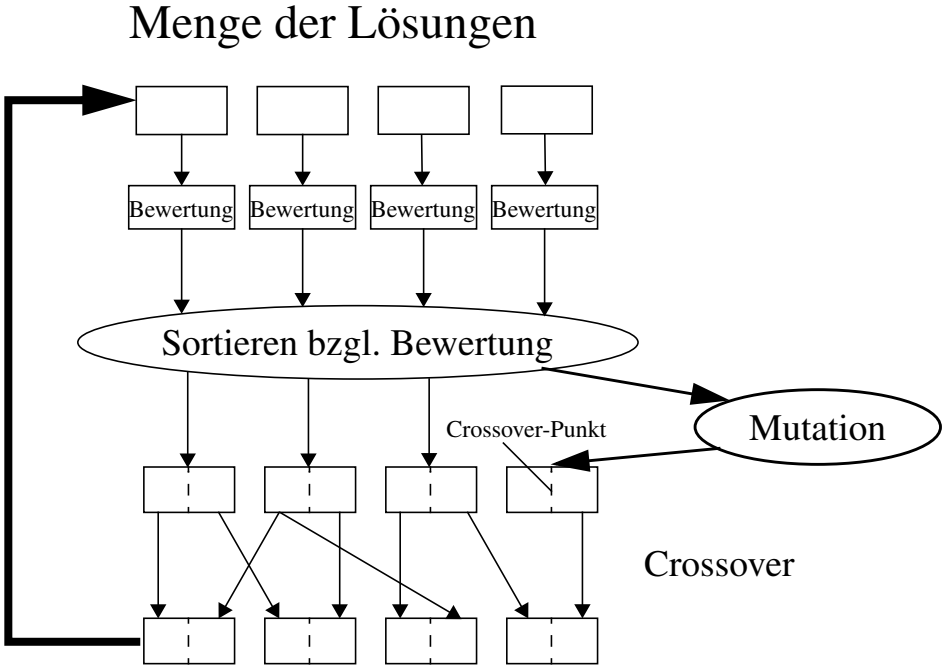



Bild 44: Grober Ablauf eines genetischen Algorithmus

Nachdem in einem Initialisierungsschritt eine Menge von möglichen Lösungen gebildet wurde, werden diese bewertet. Diese Teilmenge der Lösungen wird auch Generation genannt.

*Definition 8.8-1 Eine Generation $G \subseteq L$ ist eine Teilmenge aller Lösungen.*

Nachdem die erste Generation erzeugt wurde, findet eine Selektion statt; in dem hier dargestellten Fall werden dazu die Lösungen der Generation anhand der durch die Bewertungsfunktion gefundenen Güte sortiert.

*Definition 8.8-2 Die Selektion bildet aus einer Generation eine Teilmenge der Generation, es handelt sich also um eine Funktion der Form $Sel{:}2^L \to 2^L$ wobei gilt $Sel(G) \subseteq G$ für alle Teilmengen der Lösungsmenge.*

Es wird dann nur noch mit den ‚guten' Lösungen weitergearbeitet. Wieviele Lösungen man auf

diese Weise 'überleben' läßt, ist problemabhängig. Es bedarf einiger Experimente, diese Anzahl zu optimieren. Es gibt auch andere Selektionsmechanismen, die aus einer gegebenen Menge eine gewisse Anzahl an Lösungen auswählen, die für die weiteren Schritte geeignet erscheinen. Die ausgewählten Lösungen werden codiert und den Mutations- und Crossoveroperatoren zugeführt, welche daraus nach den dargestellten Methoden neue Lösungen bilden. Es müssen dabei so viel neue Lösungen gebildet werden, daß wieder die ursprüngliche Anzahl an Lösungen erreicht wird. Mit der neuen Menge an Lösungen startet der Prozeß erneut oder wird abgebrochen, falls die gewünschten Randbedingungen erreicht sind. Ein Durchlauf des genetischen Algorithmus läßt sich dann durch die folgende Definition beschreiben:

*Definition 8.8-3 Ein elementarer Schritt eines genetischen Algorithmus ist eine Abbildung der Form* $gen: 2^L \rightarrow 2^L$ *mit der Bedingung* $|gen(G)| = |G|$.

## 8.9 Lohnt sich der Einsatz genetischer Algorithmen ?

Der effiziente Einsatz eines genetischen Algorithmus muß für jeden Einzelfall neu überprüft werden, was im wesentlichen durch Experimente geschieht. Im folgenden wird eine theoretische Betrachtung der genetischen Algorithmen bzgl. ihres effizienten Einsatzes durchgeführt. Dazu werden einige Bedingungen angegeben. Die Bewertung muß insofern auf eine ganze Generation erweitert werden, als die Bewertung einer Generation sich aus der Bewertung der besten Lösung ergibt, wobei hier von einer Maximumbewertung ausgegangen wird: je größer der errechnete Wert, desto besser ist die Lösung.

*Definition 8.9-1 Die Bewertung einer Generation G ist gegeben durch die Funktion* $gG: 2^L \rightarrow \Re$ *und ist definiert als* $gG(G) = max\{ges_w(l) | l \in G\}$.

Das folgende Kriterium muß für jeden genetischen Algorithmus erfüllt sein : $E(gG(G)) < E(gG(gen(G)))$. Der Erwartungswert der Bewertung einer neu erzeugten Generation muß höher sein als der Erwartungswert der Bewertung der erzeugenden Generation. Durch diese Ungleichung wird von einem genetischen Algorithmus verlangt, daß er zu besseren Lösungen führt. Sind beide Erwartungswerte gleich, so handelt es sich um eine zufällige Suche im Lösungsraum. Ist der Erwartungswert der Güte der erzeugten Lösung kleiner als der, der erzeugenden Lösung, dann nimmt die Güte im wesentlichen ab, und der Algorithmus errechnet ein schlechtes Ergebnis. Lohnt sich also der Einsatz von genetischen Algorithmen nicht, was meistens nicht bewiesen werden kann, so muß auf andere Algorithmen zurückgegriffen werden. [111]

## 8.10 Einsatz im Workstationcluster

### 8.10.1 Der genetische Algorithmus im Workstationcluster

Wie aus Bild 44 ersichtlich wird, bieten sich gerade genetische Algorithmen zur parallelen Verarbeitung auf mehreren Workstations an [5]. Im speziellen können die oft zeitaufwendigen Bewertungsprozesse für eine Generation Lösungen parallel ausgeführt werden, da die Berechnungen unabhängig sind. Durch diese einfache Parallelisierbarkeit läßt sich bzgl. der Anzahl der zur Verfügung stehenden Workstations eine fast lineare Beschleunigung der Bewertung erreichen. Sortierung, Mutation und Crossover werden wieder auf einer einzelnen Workstation ausgeführt. In Bild 45 ist die Kommunikationsstruktur für genetische Algorithmen in einem Workstationcluster abgebildet.

Der Hauptprozeß wird auf einem Hostrechner ausgeführt. In dem Hauptprozeß werden die Lö-

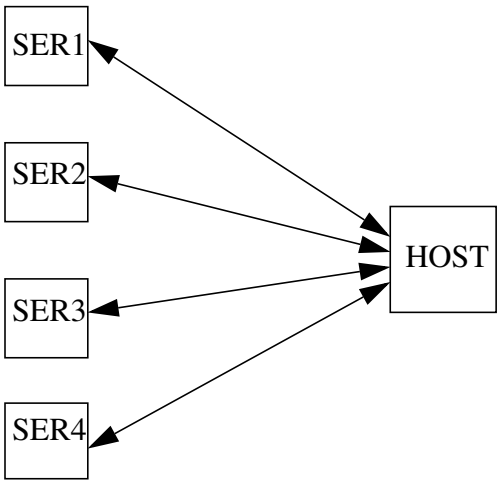


Bild 45: Komunikationsstruktur bei genetischen Algorithmen

sungen des Problems erzeugt und an mehrere Server verteilt, welche die Lösungen bewerten. Anhand der Bewertungen werden dann die besten Lösungen im Hauptprozeß gemäß einer Selektionsstrategie ausgewählt und durch Mutations- und Crossoveroperatoren miteinander verknüpft, so daß neue Lösungen entstehen. In Bild 46 ist rechts der Hauptprozeß dargestellt, welcher auf dem Hostrechner ausgeführt wird. Eine Menge Lösungen wird initial erzeugt. Dann wird jede Lösung auf einen Rechner verteilt, auf denen der Serverprozeß wartet. Der Serverprozeß führt dann die Bewertung der Lösung durch. Stehen *n* Rechner zur Verfügung, so können *n* Lösungen parallel bewertet werden. Da die Bewertung der Lösung im allgemeinen der rechenintensivste Prozeß eines genetischen Algorithmus ist, kommt eine fast lineare Geschwindigkeitssteigerung durch die Parallelisierung zustande. Steht die entsprechende Hardwarestruktur zur Verfügung, so kann die zu erwartende Beschleunigung wie im folgenden dargestellt berechnet werden.

### 8.10.2 Erwartete Beschleunigung durch Nutzung eines Workstationclusters

*Definition 8.10-1 Der sequentielle Zeitbedarf eines genetischen Algorithmus sei mit $t_{seq}$ bezeichnet und läßt sich berechnen durch $t_{seq} = t_{main} + nt_{eq}$. Dabei ist $t_{main}$ die Zeit, die der Hauptprozeß benötigt, um aus einer Menge gegebener Lösungen mit Hilfe der Selektion, der Mutation und des Crossover eine neue Menge Lösungen zu erzeugen. n ist die Anzahl der Lösungen, die bewertet werden sollen und $t_{eq}$ der Zeitbedarf der Bewertung für eine Lösung.*

Wird der genetische Algorithmus parallelisiert, so muß zu dem Zeitbedarf noch der Kommunikationsaufwand addiert werden.

*Definition 8.10-2 Der Zeitbedarf eines parallelen genetischen Algorithmus sei mit $t_{par}$ bezeichnet und läßt sich berechnen als $t_{par} = t_{main} + \frac{nt_{eq}}{m} + nt_{com}$. Dabei ist $t_{com}$ der Zeitbedarf, um eine Lösung vom Hauptprozeß an einen Serverprozeß zu übertragen einschließlich der Übertragung der Bewertung. Außerdem wird mit m die Anzahl der zur Verfügung stehenden Serverrechner bezeichnet.*

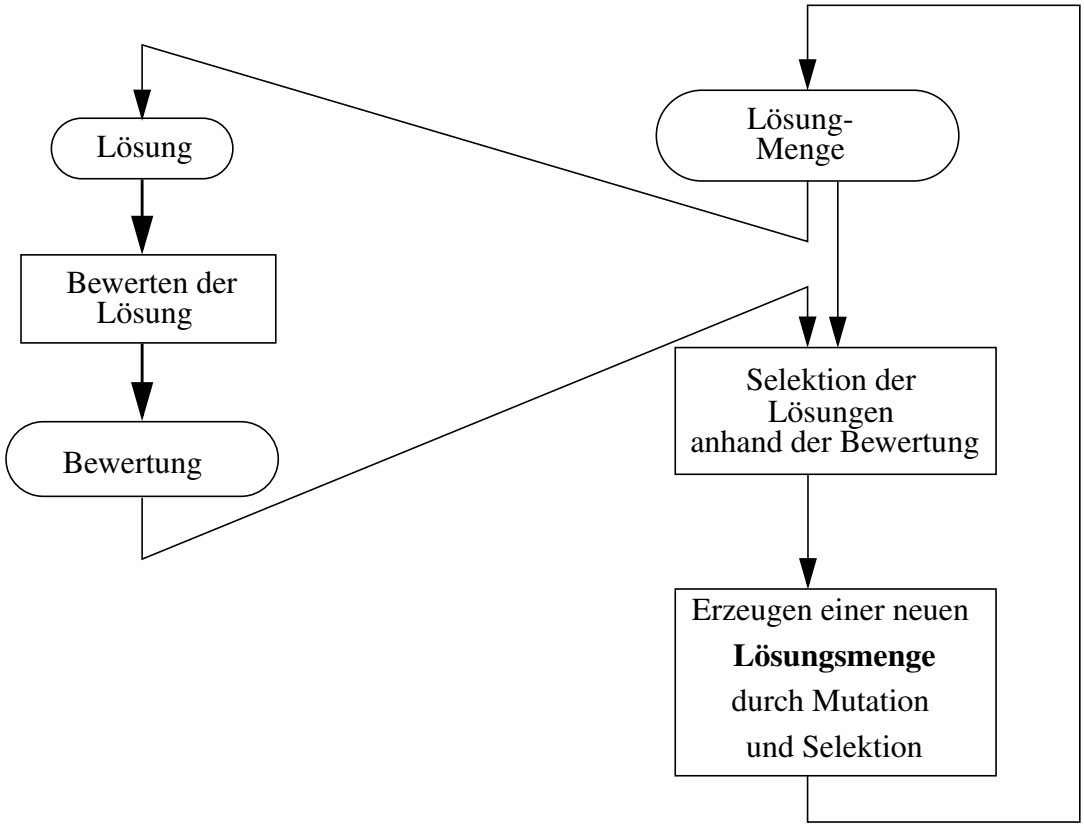



Bild 46: Allgemeiner Ablauf eines verteilten genetischen Algorithmus

Aus dem Verhältnis des sequentiellen zum parallelen Zeitbedarf läßt sich der Speedup - die Beschleunigung - eines genetischen Algorithmus berechnen.

*Definition 8.10-3 Der Speedup eines genetischen Algorithmus mit* $n = |G|$ *Lösungen pro Generation, dem m Rechner zur Verfügung stehen, gegenüber einem Rechner wird mit*

$$sp_{gen} = \frac{t_{seq}}{t_{par}} \text{ berechnet.}$$

An dieser Stelle interessiert im wesentlichen, ob und ab welcher Anzahl von Workstations es sich lohnt, eine Parallelisierung in Betracht zu ziehen, der *speedup* also größer eins ist.

*Satz 8.10-4 Eine Parallelisierung eines genetischen Algorithmus führt zu einer Beschleunigung, wenn für die Anzahl m der für die Bewertung zur Verfügung stehenden Rechner gilt* $n \geq m > \frac{t_{eq}}{t_{eq} - t_{com}}$ *und* $t_{com} < t_{eq}$.

Dies läßt sich zeigen, indem die Bedingung in $t_{par}$ eingesetzt und gezeigt wird, daß $sp_{gen} > 1$ gilt.

Genetische Algorithmen bieten also gegenüber anderen Optimierungsmethoden, z.B. dem Simulated Annealing, den Vorteil, sehr einfach parallelisierbar zu sein, wobei aber an dieser Stelle angemerkt werden muß, daß der Kommunikationsengpaß durch eine Architektur, wie z.B. in [106] beschrieben wird, nicht überwunden werden kann. Des weiteren erfolgt eine gleichmäßi-

gere Suche im Lösungsraum als beim Simulated Annealing. Andere Verfahren bleiben schneller in einem lokalen Optimum stecken und bieten bei kleinen Veränderungen keine Möglichkeiten, das globale Optimum zu finden. Durch die Suche mit genetischen Algorithmen wird dieser Aspekt im besonderen berücksichtigt, so daß auch die Qualität der Lösung verbessert werden kann. Gerade dann, wenn viele Workstations zur Verfügung stehen, die nur teilweise ausgelastet werden, bieten sich genetische Algorithmen an.

# 9 Theoretische Überlegungen zum Syntheseverfahren

Im folgenden wird das Verfahren abstrakt beschrieben. Dazu muß zuerst erklärt werden, wie die Synthese aussehen soll. In unserem Fall werden alle Kontrollstrukturen eines Programms, welche normalerweise durch den Kontrollfluß dargestellt und dann als Controller synthetisiert werden, ebenfalls in den Datenfluß mit einbezogen. Da im Rahmen dieser Arbeit im wesentlichen der Datenfluß und dessen Optimierung und Synthese betrachtet werden, muß für jeden Kontrollbefehl ein geeigneter Befehl, der auch als elementarer Befehl dargestellt werden kann, definiert werden. Im ersten Abschnitt wird gezeigt, wie Befehle, die in höheren Programmiersprachen vorkommen, in das hier definierte Zwischenformat übersetzt werden können. In [14] ist das Zwischenformat Treemola dargestellt, welches im Synthesesystem Mimola zum Einsatz kommt. Im zweiten Abschnitt wird eine allgemeine Form der Befehle definiert, die dann den Bauteilen zugewiesen wird, welche im dritten Teil allgemein dargestellt werden. In [76] ist ebenfalls eine recht allgemeine Darstellung der Bauteile vorgenommen worden. Die wesentlichen Schritte der High-Level-Synthese, die hier betrachtet werden sollen, sind die Schritte Allocation (also die Zurverfügungstellung einer gewissen Anzahl von Bauteilen), Assignment (also die Zuweisung der Befehle zu Bauteilen) und Scheduling, wodurch die Befehle gemäß ihrer Abhängigkeiten in Kontrollschritte eingefügt werden. Im folgenden werden die einzelnen Schritte definiert und die Anforderungen an jeden Schritt formuliert.

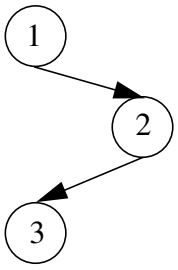
## 9.1 Die synthetisierbaren Befehle

Soll ein Programm in Hardware umgesetzt, also synthetisiert werden, dann muß für jeden Befehl ein Bauteil zur Verfügung gestellt werden, welcher diesen Befehl ausführen kann. Für arithmetische und logische Befehle ist dies auch kein oder nur ein geringes Problem, da hier arithmetische und logische Einheiten zur Verfügung stehen, die diese Befehle ausführen können. Dazu werden die Befehle den zur Verfügung stehenden Bauteilen zugeordnet. Die vorkommenden Variablen werden Registern zugeordnet, und durch eine entsprechende Verdrahtung werden die Befehle realisiert. Die Steuerung wird von einem Kontroller übernommen, der im wesentlichen eine Sequenz von Steuersignalen auf die Steuerleitungen gibt. Der Kontroller besteht im einfachsten Fall aus einem ROM, das die Steuersignale enthält, und einem Zähler, der die Adressierung vornimmt. Sind in einem Programmablauf Kontrollstrukturen enthalten, so muß der Kontroller einen endlichen Automaten realisieren, der auch Sprünge, Schleifen und bedingte Anweisungen zuläßt. Im hier vorgestellten Konzept besteht der Kontroller aus einem ROM und einem Zähler, also der einfachen Form. Die Zählvariable wird dabei als zusätzliche Variable im Programm bekanntgemacht, so daß diese Variable ebenfalls im Programm verändert werden kann, so daß Sprünge und Schleifen möglich sind. Außerdem enthält ein elementarer Befehl nur eine Operation. Somit müssen in dem Programmtext vorkommende Formeln aufgelöst werden. Eine *if*-Anweisung wird in diesem Konzept aufgefaßt als ein normaler arithmetischer Befehl, wobei der Multiplexer als Bauteil mit einer Funktion und drei Eingängen aufgefaßt wird. Somit läßt sich der *if*-Befehl einfach in dieses Konzept einordnen. Da die Programmzählervariable im Programm genau so behandelt werden kann wie jede andere Variable, ist es auch problemlos möglich, Schleifen und Sprünge zu realisieren. Im weiteren Verlauf wird die Programmzählervariable mit *step* bezeichnet. Nachdem die Definition der Befehle er-

folgt ist, wird gezeigt, wie einige beispielhaft gewählte Konstrukte aus VHDL in das Format umgesetzt werden können.

### 9.2 Anmerkungen zu den theoretischen Überlegungen

Im folgenden werden verschiedene Relationen definiert. Relationen können durch einen Graphen [30] [64] und asymmetrische Relationen durch einen gerichteten Graphen dargestellt werden. Im folgenden werden die Relationen *da*, *ada*, *aa*, <<, $<_S$ und $<_P$ definiert. In der Bild 47 ist an einem Beispiel dargestellt, wie eine Relationen als Graph dargestellt werden kann.




r(1,2)
r(2,3)

Bild 47: Darstellung der Relation r als Graph

Die Darstellung von Relationen als Graph soll hier nicht weiter betrachtet werden, da dies in den Bereich der Graphentheorie gehört. Hier gibt es genügend Literatur, auf die verwiesen sei [32] [53] [77].

### 9.3 Der elementare Befehl

Die für die Synthese interessante Darstellung eines High-Level Programms ist deren Zwischenformat, welches im wesentlichen aus einer Menge elementarer Befehle besteht. Ein elementarer Befehl hat eine bestimmte Menge an Eingangsgrößen und eine gewisse Menge an Ausgangsgrößen, welche durch Variablen spezifiziert werden. Des weiteren muß ein Operator vorhanden sein, welcher die Verknüpfung der Eingangsgrößen und die daraus folgende Berechnung der Ausgangsgrößen spezifiziert. Nähere Erläuterungen zu den Variablen werden weiter unten gemacht. Jeder Befehl hat Ein- und Ausgangsvariablen. Jede Variable wird durch einen Namen identifiziert. Die Menge der Variablen sei mit *V* bezeichnet. Es wird in dieser Definition nicht zwischen Ein- und Ausgangsvariablen unterschieden. Die Unterscheidung zwischen Ein- und Ausgangsvariablen wird durch bestimmte Befehle realisiert. Eine Variable wird schon an dieser Stelle festgelegt: in der Variable *step* wird der aktuelle Kontrollschritt abgelegt. Dies ist das Äquivalent zum Programmzähler in einem Prozessor. Die Variable *step* wird explizit benötigt, um Sprünge und Schleifen realisieren zu können. Durch die zur Verfügung stehende Variable *step* ist es möglich, alle Kontrollstrukturen als elementare Befehle zu definieren. Eine Aufteilung in Daten- und Kontrollfluß ist dadurch nicht mehr nötig. Die Funktion jedes Befehls wird durch die Operation festgelegt. Das Verfahren ist so ausgelegt, daß die Menge der elementaren Befehle durch zusätzliche Definitionen erweitert werden kann. Der Prozeß ist so ausgelegt, daß für alle nicht bekannten Funktionen bzw. Operationen eine Beschreibung vorhanden sein muß, die ihrerseits nur bekannte Funktionen enthält. Wird ein Syntheseprozeß gestartet, so wird davon ausgegangen, daß alle benutzten Operationen bekannt sind. Nach dem Syntheseprozeß wird die neu definierte Operation zu der Menge der Operationen hinzugefügt. Die Menge der Operationen sei mit *OP* bezeichnet. Jede Operation ist für eine feste Anzahl von Ein- und Ausgangsvariablen definiert, welche durch die zur Verfügung stehenden Operationseinheiten also Bauteile festgelegt werden. Das heißt also wenn es einen Addierer mit zwei Eingängen und einen Addierer mit drei Eingängen gibt, so muß es in *OP* zwei verschiedene Operationen für die

Additionen geben. Benutzt ein Befehl eine Operation, so muß sichergestellt werden, daß die gleiche Anzahl von Ein- und Ausgangsvariablen benutzt wird, wie für die Operation festgelegt ist.

*Definition 9.3-1 Die Eingangskardinalität, also die Anzahl der möglichen Eingangsvariablen einer Operation, sei durch die Funktion* $\delta_{in}: OP \rightarrow N$ *gegeben und für jede Operation definiert.*

*Definition 9.3-2 Die Ausgangskardinalität, also die Anzahl der Ausgangsvariablen einer Operation, sei durch die Funktion* $\delta_{out}: OP \rightarrow N$ *gegeben und für jede Operation definiert.*

Meistens erlaubt eine Operation nur eine Ausgangsvariable, aber hier soll eine allgemeine Betrachtung stattfinden. Die folgende Definition beschreibt nun den Befehl:

*Definition 9.3-3 Ein Befehl ist ein 3-Tupel* $(i, o, op) \in V^{\delta_{in}(op)} \times V^{\delta_{out}(op)} \times OP$ *wobei* $i = (i_1, \ldots, i_{\delta_{in}(op)})$ *das Tupel der Eingangsvariablen und* $o = (o_1, \ldots, o_{\delta_{out}(op)})$ *das Tupel der Ausgangsvariablen ist. V ist die Menge der Variablen.*

Diese Definition muß noch erweitert werden. Damit Sprünge und Schleifen realisiert werden können, kann jedem Befehl eine Sprungmarke zugewiesen werden. Dies ist eine Variable, die nach dem Scheduling den Taktschritt zugewiesen bekommt, in dem der Befehl startet.

*Definition 9.3-4 Es sei b ein Befehl nach Def. 9.3-3 und* $l \in V$*, dann ist b ein elementarer Befehl oder l:b ist ein elementarer Befehl.*

Im weiteren Text wird nur noch der elementare Befehl nach Def. 9.3-4 benötigt, so daß teilweise nur Befehl geschrieben wird.

Die Bild 48 zeigt, wie ein in herkömmlicher Form dargestellter Befehl als 3-Tupel dargestellt werden kann.

$$x = a + b \longrightarrow ((a,b),(x),+)$$

Bild 48: Darstellung eines Befehls als Tupel

Eine Menge elementarer Befehle wird im folgenden mit *B* bezeichnet, diese führt dann zu der Definition eines Programms. Aber zuerst noch einige Definitionen, die den Zugriff auf die Befehle darstellen. Die einzelnen Elemente eines Befehls können durch die folgenden Funktionen abgerufen werden:

*Definition 9.3-5 Die Funktion* $f: B \rightarrow OP$ *liefert die Operation eines Befehls.*

*Definition 9.3-6 Die Funktion* $in: B \rightarrow 2^V$ *liefert die Menge der Eingangsvariablen eines Befehls, außerdem ist* $in_j: B \rightarrow V$ *mit* $j = 1 \ldots \delta_{in}(f(b))$ *die j-te Eingangsvariable eines Befehls b.*

*Definition 9.3-7 Die Funktion* $out: B \rightarrow 2^V$ *liefert die Menge der Ausgangsvariablen eines Be-*

*fehls, außerdem ist* $out_j: B \to V$ *mit* $j = 1 \ldots \delta_{out}(f(b))$ *die j-te Ausgangsvariable eines Befehls b.*

Die Kardinalitätsfunktionen lassen sich direkt für einen Befehl definieren, so daß folgende Definition im weiteren verwendet werden kann, wobei davon ausgegangen wird, daß die Anzahl der Ein- und Ausgangsvariablen eines Befehls mit der Anzahl der durch die Operation vorgegebene Ein- und Ausgänge übereinstimmt.

*Definition 9.3-8 Die Eingangskardinalität eines Befehls b, also die Anzahl der Eingangsvariablen, ist durch die Funktion* $\delta_{in}: B \to N$ *gegeben und durch die Kardinalität der zugehörigen Operation definiert* $\delta_{in}(b) = \delta_{in}(f(b))$.

*Definition 9.3-9 Die Ausgangskardinalität eines Befehls b, also die Anzahl der Ausgangsvariablen, ist durch die Funktion* $\delta_{out}: B \to N$ *gegeben und durch die Kardinalität der zugehörigen Operation definiert* $\delta_{out}(b) = \delta_{out}(f(b))$.

Die Kardinalitätsfunktionen werden hier überladen, so daß im späteren Verlauf eine einfachere Schreibweise möglich ist. Aus dem Operanden sollte hervorgehen, welche Definition gemeint ist. Diese Definition eines Befehls impliziert die Vorstellung, daß es für gleiche Operatoren mit einer unterschiedlichen Anzahl an Eingängen auch verschiedene Bezeichnungen gibt. Da jede Variable mehreren Ein- oder Ausgängen zugewiesen werden kann, folgt:

*Satz 9.3-10 Für die Kardinalitäten der Befehle gilt :* $|in(b)| \le \delta_{in}(b)$ *und* $|out(b)| \le \delta_{out}(b)$.

Die Menge *B* wird im folgenden genau dann benutzt, wenn die Reihenfolge der Befehle für die Betrachtung nicht relevant ist. Da ein Befehl immer im Zusammenhang mit einem Programm auftritt, ist hier eine Definition des sequentiellen Programms dargestellt. Die Reihenfolge der Befehle ist durch das Programm festgelegt und kann nur unter bestimmten Randbedingungen verändert werden.

*Definition 9.3-11 Ein Programm* $PR = (b_1, \ldots, b_n)$ *oder* $PR = (B, <)$ *ist eine Ordnung von B. Wir schreiben* $b_i <_{PR} b_j$ *falls* $i < j$. *Wenn es eindeutig ist, um welches Programm es sich handelt, kann auch* $b_i < b_j$ *geschrieben werden.*

Die sequentielle Reihenfolge der Befehle einer Befehlsmenge *B* muß durch Abhängigkeiten erweitert werden, die Aufschluß über Möglichkeiten der parallelen Bearbeitung einzelner Befehle liefern. Die verschiedenen Abhängigkeiten werden im folgenden definiert.

*Definition 9.3-12 Die Datenabhängigkeit* $da \subseteq \{(b_i, b_j) \mid b_i, b_j \in B\}$ *von zwei Befehlen* $b_i$ *und* $b_j$ *in einem Programm PR ist definiert als:*

$$(b_i, b_j) \in da \Leftrightarrow (\exists l \exists k (out_l(b_i) = in_k(b_j)) \wedge (b_i <_{PR} b_j))$$
$$\Leftrightarrow (out(b_i) \cap in(b_j) \neq \varnothing \wedge (b_i <_{PR} b_j))$$

*Der Befehl* $b_j$ *ist datenabhängig von* $b_i$, *wenn er einen von* $b_i$ *berechneten Wert*

*benutzt. Diese Relation ist asymmetrisch.*

Die Bild 49 macht den Zusammenhang der Datenabhängigkeiten deutlich, wobei links die Befehle in herkömmlicher und rechts in dem definierten Format dargestellt sind. Existiert eine Datenabhängigkeit zwischen zwei Befehlen, können sie nicht parallel oder in umgekehrter Reihenfolge abgearbeitet werden. Ebenso verhält es sich mit der Antidatenabhängigkeit und der Ausgabeabhängigkeit, die ebenfalls im folgenden definiert werden.

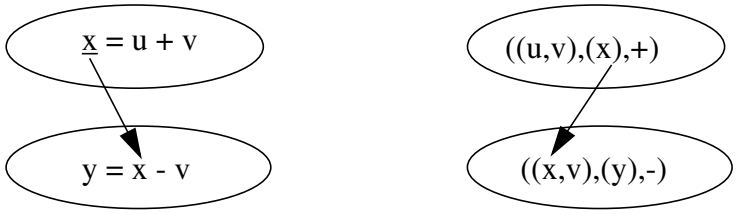



Bild 49: Datenabhängigkeit

*Definition 9.3-13 Die Antidatenabhängigkeit* $ada \subseteq \{(b_i, b_j) \mid b_i, b_j \in B\}$ *von zwei Befehlen* $b_i$ *und* $b_j$ *in einem Programm PR ist definiert als:*

$(b_i, b_j) \in ada \Leftrightarrow \exists l \exists k (in_l(b_i) = out_k(b_j)) \wedge (b_i <_{PR} b_j)$ *also: der Befehl* $b_j$ *überschreibt eine Variable, die von* $b_i$ *vorher benutzt wird, so daß* $b_i$ *noch den alten Wert der Variablen benötigt. Diese Relation ist ebenfalls asymmetrisch.*

In Bild 50 ist die Antidatenabhängigkeit zwischen zwei Befehlen dargestellt, wobei links der Befehl in herkömmlicher Form und rechts im definierten Befehlsformat wiedergegeben ist.

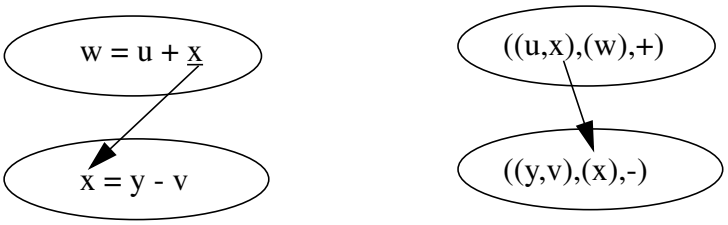



Bild 50: Antidatenabhängigkeit

*Definition 9.3-14 Die Ausgabeabhängigkeit* $aa \subseteq \{(b_i, b_j) \mid b_i, b_j \in B\}$ *von zwei Befehlen* $b_i$ *und* $b_j$ *in einem Programm PR ist definiert als:*

$(b_i, b_j) \in aa \Leftrightarrow \exists l \exists k (out_l(b_i) = out_k(b_j)) \wedge (b_i <_{PR} b_j)$ *also der Befehl* $b_j$ *ist ausgabeabhängig von* $b_i$*, denn beide berechnen dieselbe Variable und überschreiben sie. Das heißt, auch hier muß die Reihenfolge berücksichtigt werden.*

In Bild 51 ist dann die Ausgabeabhängigkeit dargestellt, die dadurch gegeben ist, daß zwei Befehle die gleiche Variable berechnen.

In der weiteren Betrachtung wird die Vereinigung und die transitive Hülle der drei Relationen benötigt, da auch indirekte Abhängigkeiten wichtig für die Einordnung in Kontrollschritte sind.

*Definition 9.3-15 Als Zusammenfassung der drei Relationen wird die allgemeine Abhängigkeit*

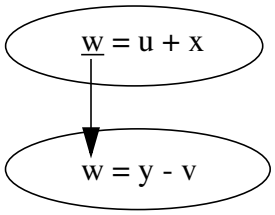



Bild 51: Ausgabeabhängigkeit

$$als\ b_i \ll b_j \Leftrightarrow (b_i, b_j) \in da \cup ada \cup aa\ definiert.$$

Da der Datenfluß über alle drei Abhängigkeiten definiert werden kann, wurde hier die allgemeine Abhängigkeit noch einmal explizit definiert. Der Datenflußgraph ist dann eine Visualisierung genau dieser allgemeinen Abhängigkeit. Benötigt wird die transitive Hülle von einigen hier definierten Relationen.

*Definition 9.3-16 Die transitive Hülle $T_R$ einer Relation $R$ wird rekursiv folgendermaßen definiert:* $(x, y) \in T_R \Leftrightarrow (x, y) \in R \vee \exists z((x, z) \in R \wedge (z, y) \in T_R)$

Es gibt natürlich auch indirekte Abhängigkeiten. Das heißt: zwei Befehle sind von einander abhängig, wenn es einen Befehl gibt, der von dem ersten abhängig ist, und der zweite Befehl von diesem Befehl abhängig ist. Dadurch ist die Reihenfolge dieser indirekt abhängigen Befehle festgelegt. Die transitive Hülle der Abhängigkeiten liefert genau die benötigte Information. In der Bild 52 ist ein Beispiel dargestellt. Der untere Graph zeigt die transitive Hülle des oberen Graphen.

Da die Relationen so definiert sind, daß sie nicht reflexiv sind, wird im folgenden die Selbstabhängigkeit, die im weiteren ebenfalls benötigt wird, dargestellt.

*Definition 9.3-17 Ein Befehl ist von sich selber abhängig, wenn es eine Rückkopplung gibt, also ein Ausgangsvariable gleich einer Eingangsvariablen ist. Die Selbstabhängigkeit kann definiert werden mit* $b \in sab \Leftrightarrow out(b) \cap in(b) \neq \varnothing$.

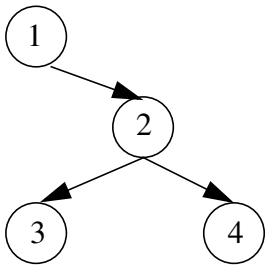
Im folgenden wird das Beispiel aus Bild 53, welches aus vier Befehlen besteht, betrachtet.

| | |
|---|---|
| 1) x:=a+b | *((a,b),(x),+)* |
| 2) y:=c*d | *((c,d),(y),*)* |
| 3) z:=x+y | *((x,y),(z),+)* |
| 4) u:=a+u | *((a,u),(u),+)* |

Bild 53: Beispiel

Das Beispiel kann in der oben definierten Form geschrieben werden als: $PR = (((a, b), (x), +), ((c, d), (y), *), ((x, y), (z), +), ((a, d), (u), +))$.

Um nicht immer den Befehl als solchen angeben zu müssen, werden die Befehle im folgenden mit ihrer Nummer bezeichnet. Für das so definierte Programm sind dann die Datenabhängigkeiten durch $da = \{(1, 3), (2, 3)\}$ gegeben. Denn die Ausgangsvariable x des ersten und die Ausgangsvariable y des zweiten sind die Eingangsvariablen des dritten Befehls. Die Selbstabhängigkeit ist nur für den letzten Befehl gegeben. Antidatenabhängigkeiten und Ausgabeabhän-




Normale Darstellung:

$(1, 2) \in da$

$(2, 3) \in ada$

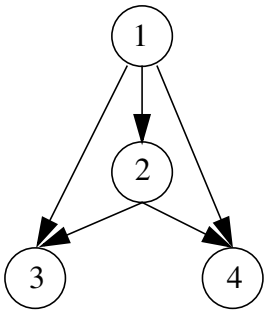
$(2, 4) \in aa$

Vereinigung der Relationen

$1 \ll 2$

$2 \ll 3$

$2 \ll 4$




Transitive Hülle

$(1, 2) \in T_{<<}$

$(1, 3) \in T_{<<}$

$(1, 4) \in T_{<<}$

$(2, 3) \in T_{<<}$

$(2, 4) \in T_{<<}$

Bild 52: Vereinigung und transitive Hülle der Relationen

gigkeiten sind in diesem Beispiel nicht vorhanden.

Eine Befehlsfolge läßt sich sehr gut in einem Graphen darstellen, dem Datenflußgraphen. Die Knoten des Graphen stellen dabei die elementaren Befehle dar, wobei die Abhängigkeiten als Kanten zwischen den Knoten dargestellt werden. Es wird also die allgemeine Abhängigkeit visualisiert. Der Graph für das Beispiel aus Bild 53 ist in Bild 54 dargestellt

.

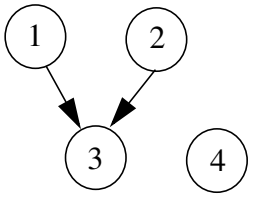



Bild 54: Datenflußgraph

Die Problematik bei dieser Definition der Befehle ist, daß hier nur der reine Datenfluß definiert werden kann. Um Kontrollmöglichkeiten mit in die Befehlsfolge einzubeziehen, müssen auch Schleifen und bedingte Verzweigungen möglich sein. Um eine Schleife zu definieren, wird die Menge der Befehle in Teilmengen unterteilt, die die Position der Befehle bzgl. der Schleife spezifizieren. Zuerst wird in Bild 55 die Umsetzung eines *if*-Statements in das interne Befehlsformat dargestellt.

| | |
|---|---|
| *if (u = 1)* | ((a, b), (t), +) |
| *then* | ((a, c), (e), -) |
| *x := a + b* | ((u, 1),(bed), =) |
| *else* | |
| *x := a - c* | ((bed, t, e),(x), if) |

Bild 55: Umsetzung eines If-Statements

Um das gezeigte *if*-Statement im Rahmen der definierten Befehle realisieren zu können, müssen zusätzliche Variablen eingefügt werden. Und zwar realisiert der *if*-Befehl eine Zuweisung der Variablen *t* oder *e* an die Variable *x* in Abhängigkeit von der Bedingung *bed*. Diese drei Variablen müssen zusätzlich eingefügt werden. Die anderen Befehle realisieren dann die eigentlichen Verknüpfungen, die normalerweise erst in dem *then*- bzw. *else*-Teil des Statements ausgeführt werden. Durch diese Codierung wird schon das sog. spekulative Scheduling impliziert, welches in [61] ausführlich beschrieben ist. Stehen genug Bauteile für die Berechnung der Variablen zur Verfügung, so kann das im Beispiel gezeigte *if*-Statement in zwei Schritten berechnet werden, was auch das Ziel des spekulativen Scheduling ist. In unserem Fall wird aber, wie später darzustellen ist, kein besonderer Scheduling-Algorithmus benötigt. Das heißt, auch die Vorteile des spekulativen Scheduling werden impliziert. Bild 56 zeigt links die Darstellung eines *if*-Statements in allgemeiner Form als Kontrollfluß. Innerhalb der Anweisungsblöcke, des then- und des else-Teiles, sind bei dieser Darstellung Datenflußgraphen enthalten. Die rechte Seite der Abbildung zeigt eine Möglichkeit, wie die Trennung zwischen Kontroll- und Datenfluß überwunden werden kann. Die Definitionen des nun folgenden Abschittes liefern eine Möglichkeit, Schleifen darzustellen.

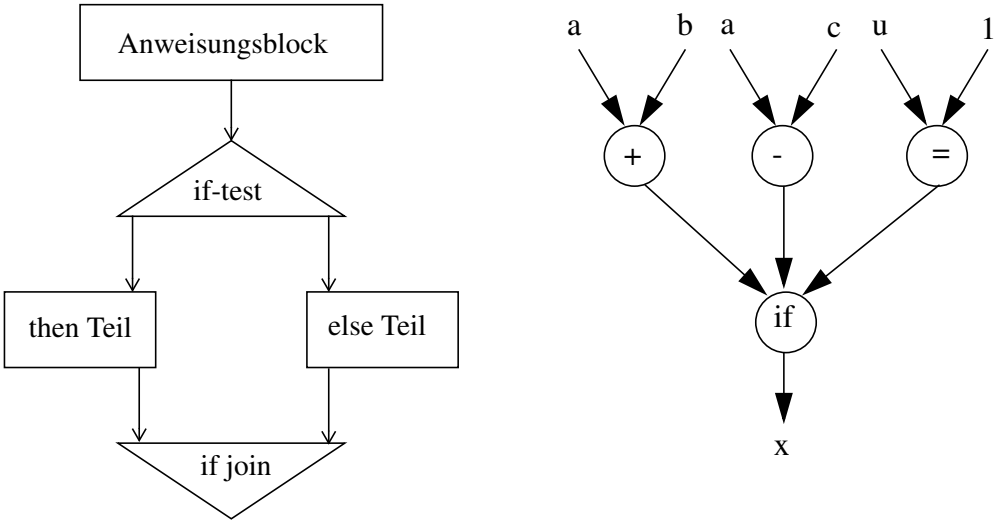



Bild 56: If-Bedingung als Kontrollgraph und Datenflußgraph

### 9.4 Schleifendarstellung durch Befehlsblöcke

Die Behandlung von Schleifen wird in der Synthese oft durch Abrollen der Schleifen realisiert, wobei aber Schleifen, deren Grenzen dynamisch errechnet werden, dann nicht synthetisierbar sind. Schleifen können verschiedene Strukturen haben, die unterschiedlich behandelt werden,

es seien hier *for*-, *while*- und *repeat-until*-Schleifen genannt. Im folgenden wird der Begriff Befehlsblock definiert, auf den sich Schleifen zurückführen lassen. Nachdem die Umsetzung der Schleifen in Befehlsblöcke dargestellt ist, wird in Abschnitt 9.4.3 ein Optimierungsalgorithmus vorgestellt, der auf den Befehlsblöcken arbeitet und somit von der ursprünglichen Schleifenstruktur abstrahiert. Ein Befehlsblock ist durch einen ersten und einen letzten Befehl in einem sequentiellen Programm gegeben. Durch die daraus hervorgehenden Blockabhängigkeiten, die in einem Scheduling mit berücksichtigt werden müssen, wird sichergestellt, daß die Blockstruktur bei der Synthese nicht verändert wird. Die hier definierte Blockstruktur ist nicht zu vergleichen mit den in [61] dargestellten Blöcken. Soll der Befehlsblock eine Schleife darstellen, dann ist der letzte Befehl ein Sprungbefehl, welcher die Verzweigung zu dem ersten Befehl durchführt.

*Definition 9.4-1 Eine Menge S wird Befehlsblock bezüglich des Programms* $PR = (B, <)$ *und der Befehle* $sa, se \in B$ *genannt, wenn gilt:* $S = \{b | (b \in B)(sa <_{PR} b <_{PR} se)\} \cup \{sa, se\}$. *Wir schreiben im weiteren auch* $S = (PR, sa, se)$.

Ein Befehlsblock partitioniert ein Programm somit in drei Teile. Neben dem Befehlsblock selber werden noch die folgenden Mengen definiert:

*Definition 9.4-2 Gegeben sei ein Befehlsblock* $S = (PR, sa, se)$ *mit* $PR = (B, <)$.

*Die Menge* $S_{vor} = \{b | (b \in B)(b <_{PR} sa)\}$ *ist die Menge der Befehle vor dem Befehlsblock.*

*Die Menge* $S_{nach} = \{b | (b \in B)(se <_{PR} b)\}$ *ist die Menge der Befehle nach dem Befehlsblock.*

Aufgrund dieser Definition kann beispielsweise eine *repeat-until* Schleife dargestellt werden, indem der erste Befehl der Schleife den ersten Befehl des Befehlsblocks definiert und der letzte Befehl, also der Rücksprung zum Anfang der Schleife, den letzten Befehl des Befehlsblocks definiert.

*Definition 9.4-3 Für jedes Befehlpaar werden durch einen Befehlsblock neue Abhängigkeiten gebildet, die Blockabhängigkeiten genannt werden. Zwei Befehle* $b_i$ *und* $b_j$ *sind blockabhängig* $b_i <_S b_j$ *bzgl. des Befehlsblocks S, genau dann wenn gilt:* $b_i <_S b_j \Leftrightarrow ((b_i \in S_{vor}) \wedge (b_j \in S \cup S_{nach})) \vee ((b_i \in S_{vor} \cup S) \wedge (b_j \in S_{nach}))$

*Außerdem gilt für den ersten und letzten Befehl des Befehlsblocks:* $sa <_S b_i \Leftarrow b_i \in S/sa$ *und* $b_i <_S se \Leftarrow b_i \in S/se$.

Durch die Blockabhängigkeit der Befehle eines Programmes kann also die durch einen Befehlsblock gegebene Ausführungsreihenfolge ausgedrückt werden. Anhand der Definition der Blockabhängigkeit ist erkennbar, daß für jeden Befehl gilt, daß er entweder blockabhängig von einem anderen Befehl ist, oder daß ein Befehl von ihm blockabhängig ist, falls ein Befehlsblock für ein Programm definiert ist. Die Definition ist auch für andere Kontrollstrukturen in Programmen anwendbar, z.B. kann ein *if-then*-Statement auf diese Weise definiert werden, wobei die Befehle des *then*-Teils in einem Befehlsblock untergebracht werden. Jedes Programm wird

also durch mehrere Befehlsblöcke erweitert, die beispielsweise die Schleifen des Programms angeben.

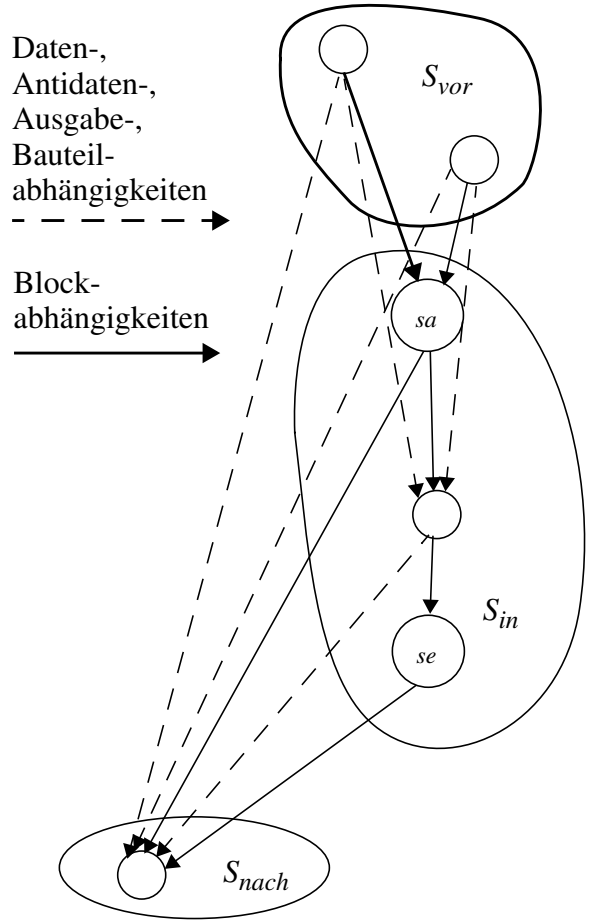



Bild 57: Datenfluß aus Blockabhängigkeiten

In Bild 57 ist ein Beispiel für die Blockabhängigkeiten dargestellt. Soll der Befehlsblock eine Schleife darstellen, so müssen spezielle Befehle dafür sorgen, daß die Variable *step*, welche den Verarbeitungs- oder Kontrollschritt explizit angibt, so gespeichert wird, daß der Kontroller am Ende einer Schleife an die richtige Stelle springt und dort die Ausführung fortführt. Die Variable kann deshalb wie jede andere Variable benutzt werden. Da die Semantik der Befehle durch die gegebenen Bauteile, welche ja genau diese Befehle ausführen sollen, definiert wird, wird für jeden vorkommenden Befehl mindestens ein Bauteil benötigt, welches den Befehl ausführen kann. In den folgenden Abschnitten wird dargestellt, wie eine Schleife in dem definierten Befehlsformat aussehen kann.

### 9.4.1 Darstellung einer *repeat-until* Schleife

In Bild 58 ist ein Beispiel für die Umsetzung einer *repeat-until* Schleife dargestellt. Der Befehlsblock und die dadurch definierten Mengen sind $S = \{1, 2, 3, 4\}$, wobei der erste Befehl $sa = 1$ und der letzte Befehl $se = 4$ sind. Die anderen Mengen werden an dieser Stelle nicht benötigt, da weder Befehle vor der Schleife noch nach der Schleife vorhanden sind. Nachdem das Scheduling durchgeführt wurde, wird die eingefügte Variable $Sanf$ auf den Wert des Kontrollschrittes, in dem der erste Befehl der Schleife ist, gesetzt. Der vierte Befehl - also die *until*-Anweisung - muß nun so definiert werden, daß der Variablen *step*, anhand der Bedingung *bed*, eine der beiden Variablen *Sanf* oder *step* zugewiesen wird. Dabei ist zu beachten, daß *step* um

eins inkrementiert werden muß, wenn die Bedingung falsch ist. Das Bauteil, mit dem eine *until* Anweisung implementiert wird, besteht also aus einem Multiplexer, wobei dem zweiten Eingang noch ein Inkrementer vorgeschaltet ist. Durch diese Realisierung der Schleife liegt die gesamte Kontrolle in dem Datenfluß. Der Vorteil, eine reine Datenflußsynthese durchzuführen, liegt darin, daß alle Parallelitäten voll ausgenutzt werden.

| | |
|---|---|
| *repeat* | 1) ((a,b), (x), +) |
| $x = a + b$ | 2) ((a,c), (y), -) |
| $y = a - c$ | 3) ((u,1), (bed), =) |
| *until (u = 1)* | 4) ((Sanf, step , bed), (step), until) |

Bild 58: Umsetzung eines *repeat-until* Statements

### 9.4.2 Darstellung einer *while*-Schleife

Eine Beispiel für eine *while*-Schleife ist in Bild 59 dargestellt. Der wesentliche Unterschied zu einer *repeat-until*-Schleife besteht darin, daß die Bedingung vor der Ausführung der Schleife geprüft wird. Das bedeutet aber, daß es vorkommen kann, daß der Rumpf nicht ausgeführt wird im Gegensatz zur *repeat-until*-Schleife, wo der Rumpf mindestens einmal ausgeführt wird.

| | |
|---|---|
| *while (u = 1) do* | 1) ((u,1), (bed), =) |
| $x = a + b$ | 2) ((Se2, step, bed), (step), while) |
| $y = a - c$ | 3) ((a,b), (x), +) |
| $z = x * y$ | 4) ((a,c), (y), -) |
| | 5) ((Sa1), (step), :=) |
| | 6) ((x,y), (z), *) |

Bild 59: Umsetzung eines *while-do* Statements

Um diese Schleife in Form von Befehlsblöcken darzustellen, werden zwei verschachtelte Blökke benötigt. Der erste Block $S_1 = (PR, 1, 5)$ ist so gestaltet, daß der Rücksprung am Ende des Schleifenrumpfes korrekt durchgeführt wird. Es reicht nicht aus, genau einen Befehl vor den *while*-Befehl zu springen, da die Berechnung der Bedingung aus mehreren Befehlen bestehen kann. Das heißt, der Variablen *Sa1* muß die Nummer des Kontrollschrittes zugewiesen werden, in den der erste Befehl eingeordnet wird. Die Zuweisung in Befehl 5 bildet also den Rücksprung an den Anfang der Schleife. Mit dem zweiten Befehlsblock $S_2 = (PR, 2, 6)$ wird der Variablen *Se2* der Wert des Kontrollschrittes zugewiesen, in den der Befehl 6 eingeordnet wird. Das *while*-Statement weist bei einer Belegung der Variablen *bed* mit 0, also wenn die Bedingung falsch ist, der Variablen *step* den Wert *Se2* zu. Umgekehrt wird der Variablen *step* wieder der Wert *step* zugewiesen, der aber um eins inkrementiert ist. Es muß also an dieser Stelle wieder ein Bauteil vorhanden sein, welches die Operation *while* korrekt ausführen kann. *For*-Schleifen lassen sich ebenso darstellen wie *repeat-until* Schleifen, da es sehr einfach möglich ist, jede *for*-Schleife in eine *repeat-until*-Schleife zu übersetzen. Die Darstellung mit zwei Befehlsblöcken ist damit zu erklären, daß dadurch ein einheitliches Format erreicht und die Optimierung auf ei-

nem Befehlsblock ausgeführt werden kann, und nicht Schleifenspezifisch ist. Eine Lösung mit nur einem Befehlsblock kann zu Problemen führen, da dann nicht alle Abhängigkeiten korrekt gesetzt werden.

Im Folgenden wird dargestellt, wie einzelne Befehle aus dem Befehlsblock entfernt werden können. Die entsprechenden Blockabhängigkeiten fallen somit weg, und eine Realisierung durch einen Schedulingalgorithmus wird flexibler, was in vielen Fällen dazu führen kann, daß ein schnelleres Design ensteht.

### 9.4.3 Schleifenoptimierung

Die Darstellung von Schleifen durch Befehlsblöcke führt mit Def. 9.4-3 zu einer enormen Anzahl zusätzlich zu berücksichtigender Abhängigkeiten. Die hierbei entstehenden Abhängigkeiten schöpfen dabei keine Optimierungsmöglichkeiten, die zu einem schnelleren Design führen, aus. Durch den folgenden Algorithmus werden genau die Befehle gesucht und aus dem Befehlsblock entfernt, die isolierte Knoten im Datenflußgraphen, eingeschränkt auf den Befehlsblock, sind. Das heißt, die Ergebnisse, die diese Befehle liefern, sind in jedem Schleifendurchlauf gleich und müssen daher nur einmal berechnet werden. Eine Schleifenoptimierung [3] ist nur dann sinnvoll, wenn die ursprüngliche Semantik des Programms nicht verändert wird. Eine Schleife wird auf einen Befehlsblock *S* wie oben gezeigt zurückgeführt. Um die einzelnen Schritte der Optimierung darzustellen, wird das Beispiel in Bild 60 gewählt.

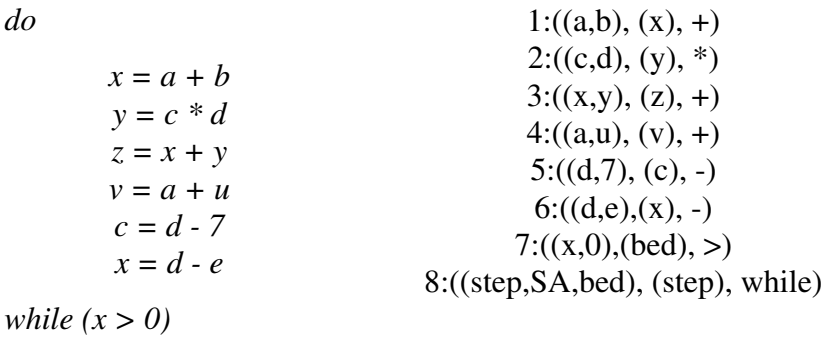



Bild 60: Beispiel

Die Befehle werden gemäß ihrer Reihenfolge durchnumeriert. Zu beachten ist, daß die Bedingung für die Schleife in Befehl 7 berechnet wird. In Befehl 8 wird dann anhand der Bedingung entweder der Wert der Variablen *SA* der Variablen *step* zugewiesen oder wieder die Variable *step* selber. Der Wert der Variablen *SA* ist noch nicht bekannt und wird erst dann festgelegt, wenn bekannt ist, in welchem Kontrollschritt der erste Befehl der Schleife durchgeführt werden soll. Durch diese Zuweisung wird ein bedingter Sprungbefehl realisiert, da *step* ja den Programm- bzw. den Kontrollschrittzähler repräsentiert.

Für die durch den Befehlsblock *S*, welcher die Schleife spezifiziert, definierten Mengen gilt dann:

$S = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$S_{vor} = \{\ \}$

$S_{nach} = \{\ \}$

$sa = 1$

$se = 8$

Das heißt also: alle dargestellten Befehle befinden sich in dem Befehlsblock. Der erste Befehl

- mit der Nummer 1 - ist auch der Anfangsbefehl der Schleife, und der letzte Befehl - mit der Nummer 8 - ist der letzte Befehl der Schleife. Der Datenflußgraph für das Beispiel ist in Bild 61 dargestellt. Hier wird auch die Selbstabhängigkeit für den Befehl 8 mit dargestellt.

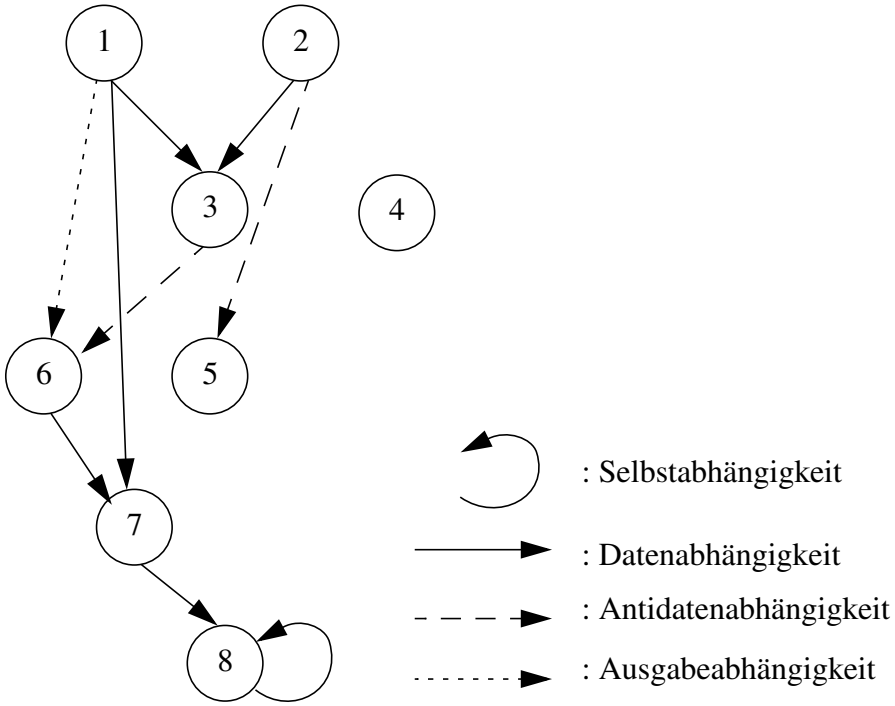



Bild 61: Der Datenflußgraph für die Schleife

In Bild 62 ist der Optimierungsalgorithmus dargestellt, welcher aufgrund der Abhängigkeiten der Befehlsfolge prüft, welche Befehle aus dem Befehlsblock und damit aus der Schleife entfernt werden können. Für die gefundenen Befehle ist dann die Einordnung in Kontrollschritte frei wählbar in Bezug auf die Abhängigkeiten, die durch die Blockbildung hinzugekommen sind. In dem Algorithmus werden die Mengen $S'$, $S'_{nach} = S_{nach}$ und $S'_{vor} = S_{vor}$ berechnet, die gemäß Def. 9.4-3 die neuen Abhängigkeiten definieren. Der erste und der letzte Befehl der Schleife sind in der Menge $S'$ enthalten. Im ersten Schritt werden alle selbstabhängigen Befehle aus $S$ nach $S'$ übertragen, es wird also die Bedingung $b \in sab \Rightarrow b \in S'$ für jeden Befehl geprüft. Es ist offensichtlich, daß selbstabhängige Befehle nicht aus einem Befehlsblock entfernt werden dürfen, da hier in Abhängigkeit von der Anzahl der Durchläufe Variablen geändert werden, wie z.B. Zählvariablen. Für das Beispiel heißt das, daß der Befehl 8 in die Menge $S'$ übertragen wird. Es folgt also $S' = \{8\}$. Antidatenabhängigkeiten sind innerhalb von Blöcken besonders zu berücksichtigen, da sie bei Wiederholungen zu Datenabhängigkeiten werden. Die Bedingung $(b_i, b_j) \in ada \Rightarrow b_i, b_j \in S'$ überträgt alle Paare von antidatenabhängigen Befehlen in die Menge $S'$. Für das Beispiel folgt daraus $S' = \{2, 3, 5, 6, 8\}$. Die Ausgabeabhängigkeit muß in der dritten Bedingung $(b_j, b_k \in S') \wedge ((b_i, b_j) \in da) \wedge ((b_i, b_k) \in aa) \Rightarrow b_i \in S'$ ebenfalls berücksichtigt werden. Würde der Befehl $b_i$ nicht innerhalb des Blocks erhalten bleiben, so wird bei einer Wiederholung eine falsche Variablenbelegung für die weitere Berechnung herangezogen. In dem Beispiel ist dieser Fall für die Befehle 1, 3 und 6 gegeben, so daß der Befehl 1 in die Menge $S'$ mit aufgenommen wird. Daraus folgt dann $S' = \{1, 2, 3, 5, 6, 8\}$. Die letzte betrachtete Bedingung des dargestellten Optimierungsalgorithmus $(b_i, b_k \in S') \wedge ((b_i, b_j) \in T_{da}) \wedge ((b_j, b_k) \in T_{da}) \Rightarrow b_j \in S'$ führt dazu, daß Befehle, die datenabhängig zueinander sind, bzw. von Befehlen, die schon innerhalb des Blockes liegen, datenabhängig 'eingeschlossen' werden, innerhalb eines Blockes bleiben. Da die Befehle 6 und 8 des Beispiels in der Menge $S'$ enthalten sind, muß Befehl 7 ebenfalls hinzugefügt werden. Damit

Eingabe: $S$, $S_{vor}$, $S_{nach}$, $sa$, $se$

$S'_{vor} = S_{vor}$; $S'_{nach} = S_{nach}$; $S' = \{\ \}$

for all $b \in S$

$b \in sab \Rightarrow b \in S'$

for all $b_i, b_j \in S$

$(b_i, b_j) \in ada \Rightarrow b_i, b_j \in S'$

repeat

for all $b_i, b_j, b_k \in S$

$(b_j, b_k \in S') \wedge ((b_i, b_j) \in da) \wedge ((b_i, b_k) \in aa) \Rightarrow b_i \in S'$

$(b_i, b_k \in S') \wedge ((b_i, b_j) \in T_{da}) \wedge ((b_j, b_k) \in T_{da}) \Rightarrow b_j \in S'$

until (keine Veränderung von im $S'$ im letzten Durchlauf)

Bild 62: Befehlsblockoptimierung

berechnet der Algorithmus $S' = \{1, 2, 3, 5, 6, 7, 8\}$. In diesem Beispiel wird also der Befehl 4 aus dem Block entfernt. Dieser Befehl muß also nur einmal ausgeführt werden, was zu einer Beschleunigung der Ausführung führen kann. In Bild 62 wird der Algorithmus explizit dargestellt. Der so gebildete Befehlsblock $S'$ führt zu einer Reduzierung der Blockabhängigkeiten, die in Abhängigkeit von $S'$ gebildet werden. Das heißt also, es kann jetzt mit den Blockabhängigkeiten $<_{S'}$ weitergearbeitet werden.

In Bild 63 sind die für die Optimierung relevanten Abhängigkeiten für das Beispiel noch einmal explizit dargestellt. Es ist deutlich ersichtlich, daß der Befehl 4 keine Abhängigkeiten zu anderen Befehlen des Befehlsblocks besitzt. Somit kann dieser Befehl auch außerhalb des Befehlsblockes ausgeführt werden.

Das Verfahren hat den Vorteil, daß es mit weniger Ressourcenbedarf die gleichen Ergebnisse erzielt wie ein Loop-Folding Verfahren, welches von jedem Befehl in der Schleife eine Kopie anfertigt. Da keine weiteren Abhängigkeiten zwischen Befehlen in der Schleife und außerhalb der Schleife bestehen und andrerseits davon ausgegangen wird, daß die Schleife mindestens einmal ausgeführt wird, können die Befehle, die außerhalb der Schleife stehen, an beliebiger Stelle ausgeführt werden. Ist dieser Algorithmus durchgeführt und eine optimierte Schleife erzeugt worden, so kann diese optimierte Schleife im weiteren für die Synthese benutzt werden. Für das Beispiel aus Bild 58 sieht die Optimierung so aus, daß die Befehle 2 und 3 aus der Schleife genommen werden, da es keine Veränderungen in der Schleife gibt. Die Schleife besteht dann nur noch aus den Befehlen 1 und 4. Selbst der erste Befehl der Schleife könnte in dem speziellen Fall der *while-do* Schleife entfernt werden. Aber da sich der Algorithmus auf allgemeine Schleifen bezieht, wird dieser Speziallfall nicht berücksichtigt. Die Optimierung der Schleifen ist auf jeden Fall sinnvoll, da durch die automatische Erzeugung des Befehlsformates viele Befehle entstehen können, die nur einmal aufgerufen werden müssen, aber dennoch Ressourcen und vor allem Zeit in einer Schleife konsumieren. [12]

### 9.5 Umsetzung von VHDL Konstrukten in das Zwischenformat

Das oben dargestellte Befehlsformat soll nun in die konkrete Abhängigkeit zu einer real existierenden Hardwarebeschreibungssprache gebracht werden. Wie schon in der Einleitung darge-

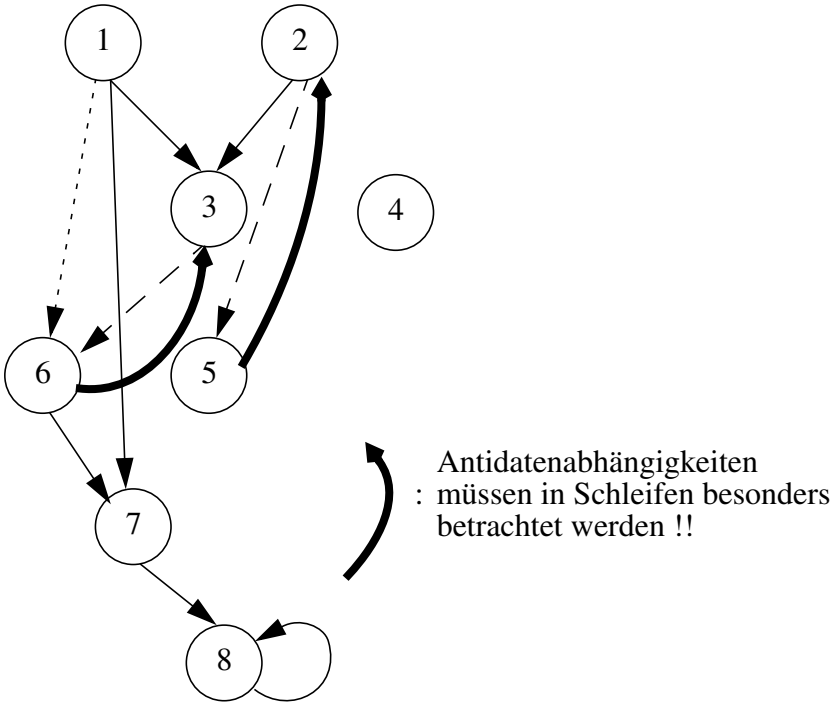



Bild 63: Antidatenabhängigkeiten in Schleifen

stellt, gibt es mehrere Sprachen, die für die Beschreibung von Schaltungen auf algorithmischer Ebene geeignet oder gedacht sind. An dieser Stelle sei auf entsprechende Literatur verwiesen: beispielsweise sind in [75] und [68] verschiedene Synthesesprachen und Systeme dargestellt. Da zumindest in Europa die Hardwarebeschreibungssprache VHDL am weitesten verbreitet ist, werden die folgenden Beispiele auf VHDL basieren, die ursprünglich als Spezifikations- und Simulationssprache konzipiert worden ist. Der Aspekt der Synthese ist erst später hinzugekommen, so daß die bisher existierenden Synthesesysteme nur eine Teilmenge von VHDL als synthetisierbare Eingabe akzeptieren. Hier sollen, angefangen bei einer einfachen Zuweisung, *if*-Anweisungen und die Implementierung von Schleifen dargestellt werden. *Case*-Anweisungen werden hier nicht betrachtet, da sie auch als *if*-Anweisungen darstellbar sind. Außerdem wird hier nur die *while*-Schleife betrachtet, die durch eine Bedingung abgebrochen wird. Schleifen mit einer statischen Anzahl von Durchläufen können mit dem hier vorgestellten Verfahren realisiert werden, aber auch mit den hier nicht weiter betrachteten Verfahren des Loop-Unrolling [52] oder des Loop-Pipelining.

### 9.5.1 Zuweisungen und arithmetische/logische Befehle

Eine einfache Zuweisung wie z.B. $x := a + b$ wird durch ein Bauteil realisiert, welches genau die Funktion ausführt. Hier ist es also ein Addierer.

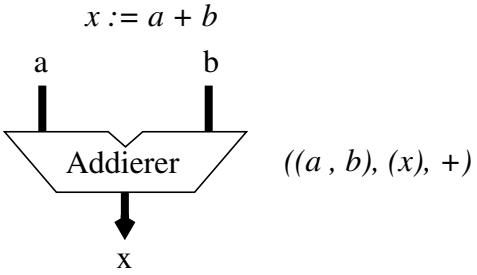



Bild 64: Umsetzung einer Addition

An dieser Stelle kann noch nicht entschieden werden, ob die Variablen als Register implementiert werden. In der Bild 64 ist die Signalzuweisung, wie sie in VHDL vorkommen könnte, oben dargestellt. Rechts neben der Schaltung ist in der Abbildung das definierte Format der Anweisung dargestellt. Die Umsetzung ist an dieser Stelle sehr einfach, da eine direkte Umsetzung der Zuweisung in Hardware möglich ist. Im nächsten Schritt soll die Realisierung einer Formel dargestellt werden. Z.B. kann die Anweisung *x := a+b * c * d * e* auf mehrere Weisen umgesetzt werden. Hierbei ist es wichtig, wie oben schon dargestellt, daß eine schnelle Verarbeitung der Anweisung durch die synthetisierte Hardware möglich ist. Da davon ausgegangen wird, daß alle benutzten Bauteile nur zwei Eingänge haben, also z.B. nur Multiplizierer mit zwei Eingängen existieren, müssen im Zwischenformat mehrere Anweisungen für die Umsetzung erzeugt werden. Bild 65 zeigt zwei Möglichkeit für das Zwischenformat der Anweisung, die sich durch verschiedene Klammersetzungen unterscheiden.

*x := a + (((b * c) * d) * e)*

*((b , c) , (v1), *)*
*((v1 , d) , (v2), *)*
*((v2 , e) , (v3), *)*
*((a , v3) , (x), +)*

*x := a + ((b * c) * (d * e))*

*((b , c) , ((v1), *)*
*((d , e) , (v2), *)*
*((v1 , v2) , (v3), *)*
*((a , v3) , (x), +)*

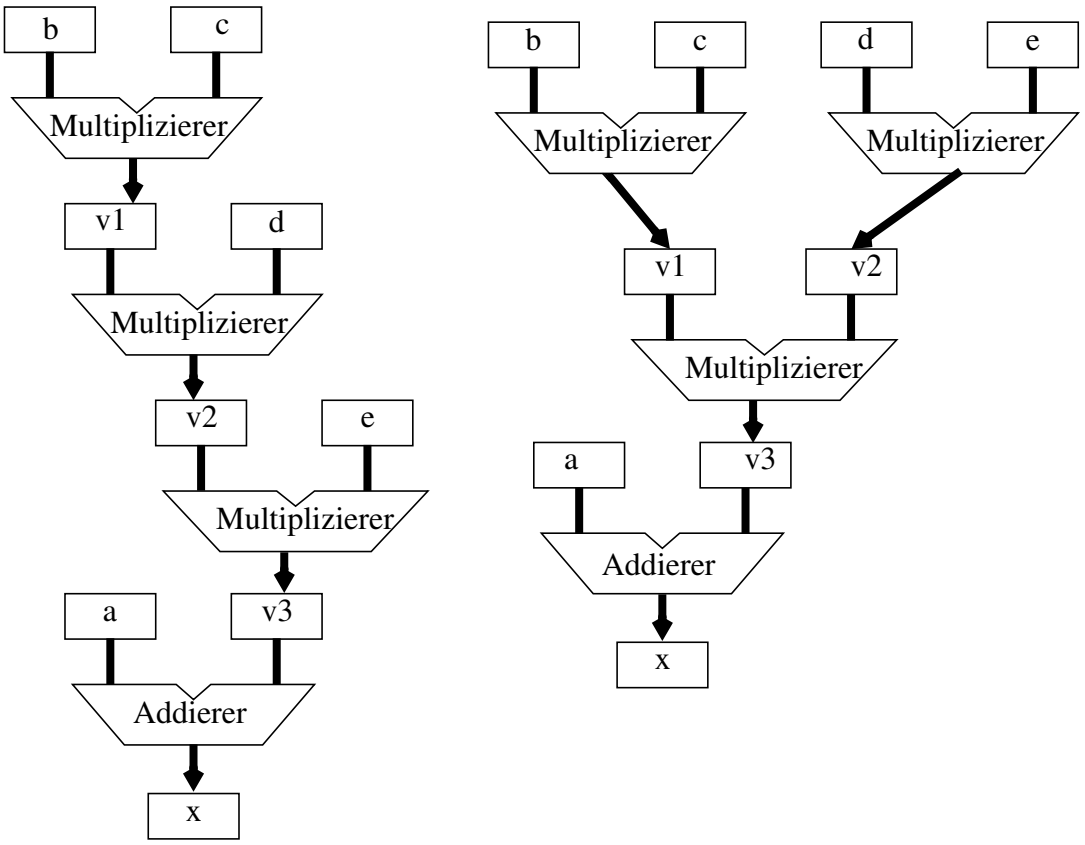



Bild 65: Realisierung einer Gleichung

Für die Umsetzung werden zusätzliche Variablen erzeugt, um die Zwischenergebnisse darzu-

stellen. Daß hier Variablen und Multiplizierer mehrfach genutzt werden können, soll an dieser Stelle noch nicht berücksichtigt werden. Zumindest ist durch die Klammersetzung, die in Bild 65 auf der rechten Seite gegeben ist, die Möglichkeit vorhanden, eine schnellere Schaltung dadurch zu erzeugen, daß die ersten beiden Multiplikationen parallel ausgeführt werden können.

### 9.5.2 If-Statement

Im nächsten Schritt soll die Umsetzung von *if*-Anweisungen genauer betrachtet werden. Eine *if*-Anweisung wird dazu zuerst in das definierte Format umgesetzt. Bild 66 zeigt, wie eine einfache *if*-Anweisung mit einem Multiplexer umgesetzt werden kann. Dabei ist die Variable *bedingung* schon berechnet worden.

*if (bedingung) then* ((bedingung, x, a), (x))

*x := a*

*end if*

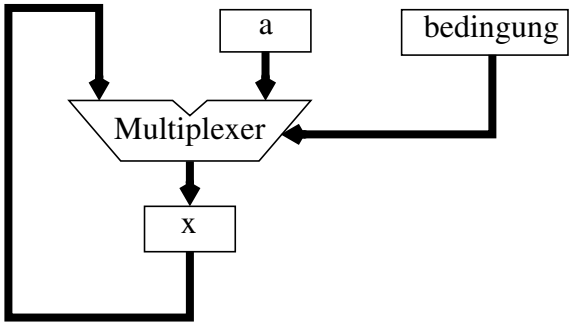



Bild 66: Realisierung einer *if*-Anweisung mit einem Multiplexer

Die Implementierung einer *if*-Anweisung beruht darauf, daß, falls der Wahrheitswert der Bedingung wahr ist, also im Beispiel *bedingung = 1* gilt, der Eingang *eins* des Multiplexers auf den Ausgang geschaltet wird, und falls die Bedingung falsch ist, also *bedingung = 0* gilt, der Wert, der am Eingang *null* des Multiplexers liegt, auf den Ausgang geschaltet wird. Im Beispiel verändert sich der Wert der Variablen *x* nur im ersten Fall, also der Eingang *null* mit der Variablen *x* selber belegt wird. In Bild 67 ist eine Alternative dargestellt, welche darauf beruht die Steuerleitung des Ergebnisregisters, also die Enable-Leitung, über die Bedingung anzusteuern.

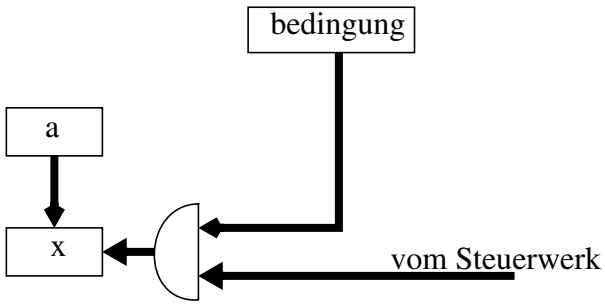



Bild 67: Realisierung einer *if*-Anweisung mit *UND*-Gatter und Register

Hierbei wird das Register über ein *UND*-Gatter geschaltet. Der Eingang wird nur dann übernommen, wenn die Bedingung wahr ist, also *bedingung = 1* gilt, und das Register durch das Steuerwerk angesteuert wird. Es handelt sich um eine Optimierung, die immer dann angewandt werden kann, wenn nur an dieser Stelle in das Ergebnisregister geschrieben wird. Wird die Ergebnisvariable noch anderweitig benutzt, so wird die Steuerlogik entsprechend komplexer. Au-

ßerdem wird an dieser Stelle die Trennung zwischen Steuerwerk und Rechenwerk aufgehoben, was die benötigte Rechenleistung des Synthesesystems erhöht, da komplexere Algorithmen nötig sind. Dieser Optimierungsschritt muß nicht weiter für den Vergleich der Ergebnisse betrachtet werden, da er auf alle Systeme nachträglich angewandt werden kann.

Die Umsetzung einer *if-then-else*-Anweisung ist durch eine Erweiterung der in Bild 66 dargestellten Lösung möglich. Die Bild 68 zeigt die Umsetzung einer einfachen *if-then-else*-Anweisung. Anzumerken ist an dieser Stelle, daß die Register, in denen die Variablen gespeichert werden, nicht zur eigentlichen Implementierung der *if-then-else*-Anweisung gehören. Sie stellen die Implementierung der Variablen dar. An anderer Stelle wird gezeigt, wie das Optimierungsverfahren zur Reduzierung der benötigten Register aussieht.

*if (bedingung)* *((bedingung, b, a), (x))*

*then x := a;*

*else x := b;*

*end if*

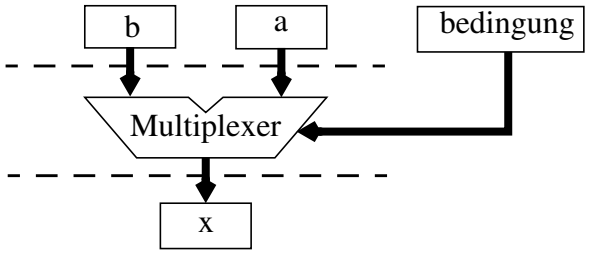



Bild 68: Realisierung einer *if-then-else*-Anweisung mit einem Multiplexer

Eine *if-then-else*-Anweisung kann also sehr einfach mit einem Multiplexer realisiert werden, wobei aber vorausgesetzt werden muß, daß bei den Zuweisungen im *then*- und im *else*-Zweig die gleiche Variable auf der linken Seite steht. Im folgenden wird darauf aufbauend die Realisierung einer komplexeren *if-then-else*-Anweisung dargestellt. Den Variablen werden jetzt Formeln zugewiesen. Außerdem sind die Zielvariablen im *then*- und im *else*-Zweig unterschiedlich.

Im ersten Schritt wird die Bedingung berechnet und in die Variable *v1* geschrieben. Dann werden alle in der Anweisung vorkommenden Befehle dargestellt, wobei ihre Ergebnisse in neu erzeugten Variablen abgelegt werden. Nun muß für jede Variable, an die ein Wert zugewiesen wird, ein Multiplexer erzeugt werden. Bei der Darstellung der Befehle im definierten Format ist darauf zu achten, daß es hier nur *if*-Befehle gibt, wobei die Eingänge - je nachdem ob der *else*- oder der *then*-Zweig realisiert werden soll - entsprechend belegt werden. Der Vorteil dieser Darstellung ist, daß mehrere Befehle parallel bearbeitet werden können. Die ersten vier Befehle werden parallel bearbeitet, und die letzten drei Befehle können ebenfalls parallel ausgeführt werden. Bild 69 zeigt das erzeugte Rechenwerk, die Steuerleitungen sind in der Abbildung nicht dargestellt. Durch diese Darstellung einer *if-then-else*-Anweisung werden die in [61] dargestellten Vorteile des spekulativen Scheduling ebenfalls berücksichtigt. Die Realisierung von *case*-Anweisungen wird entsprechend durchgeführt, was an dieser Stelle aber nicht vertieft werden soll. Nachdem nun die Realisierung von einfachen Anweisungen und Bedingungen dargestellt wurde, wird jetzt die Umsetzung einer Schleife gezeigt.

| | |
|---|---|
| *if (bed > 3)* | *((bed, 3), (v1), >)* |
| *then* | *((a, b), (v2), *)* |
| *x := a * b;* | *((c, d), (v3), -)* |
| *y := c - d;* | *((b, 2), (v4), +)* |
| *else* | *((v1, x, v2), (x), if)* |
| *z := b + 2;* | *((v1, y, v3), (y), if)* |
| *end if;* | *((v1, v4, z), (z), if)* |

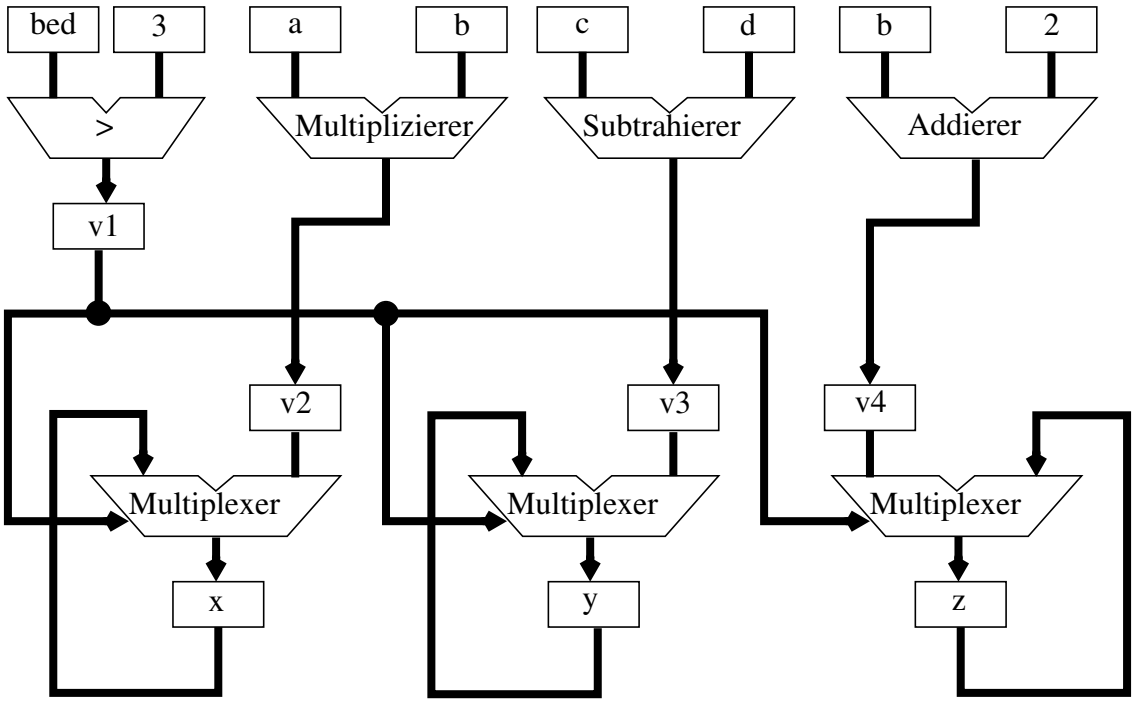



Bild 69: Beispiel zur Realisierung einer komplexen *if-then-else* Anweisung

### 9.5.3 Schleifen

Wie schon oben dargestellt ist, muß dazu die Variable *step* zur Verfügung stehen. Sie kann genauso wie andere Variablen in den Befehlen des Zwischenformates benutzt werden. An dieser Stelle soll die *do-while*-Schleife betrachtet werden, die solange ausgeführt wird, bis die Bedingung erfüllt ist. Diese Schleife wird von den meisten Synthesetools nicht akzeptiert. Deshalb soll an dieser Stelle eine Umsetzung dargestellt werden. Zur Realisierung einer Schleife wird die Variable *step*, die den Kontrollschrittzähler oder Programzähler für den Mikrocode angibt, anhand der Bedingung entweder auf den nächsten Kontrollschritt oder auf den Kontrollschritt, in den der erste Befehl der Schleife eingeordnet ist, gelegt. Die oben genannten Bedingungen für eine Schleife sind dabei zu beachten. In Bild 70 wird die Realisierung einer *do-while* Schleife dargestellt.

Die Bedingung wird dabei in der Schleife berechnet und der Wahrheitswert der Bedingung einer neu erzeugten Variablen *v1* zugewiesen. In der Variablen *sa* ist die Nummer des Kontrollschrittes abgelegt, in dem der erste Befehl der Schleife ausgeführt wird. Diese Variable kann

| | |
|---|---|
| *do* | *((a ,b), (x), +)* |
| *x = a + b* | *((a, c), (a), -)* |
| *a = a - c* | *((x, 7), (v1), >)* |
| *while (x > 7)* | *((v1, step, sa), (step), while)* |

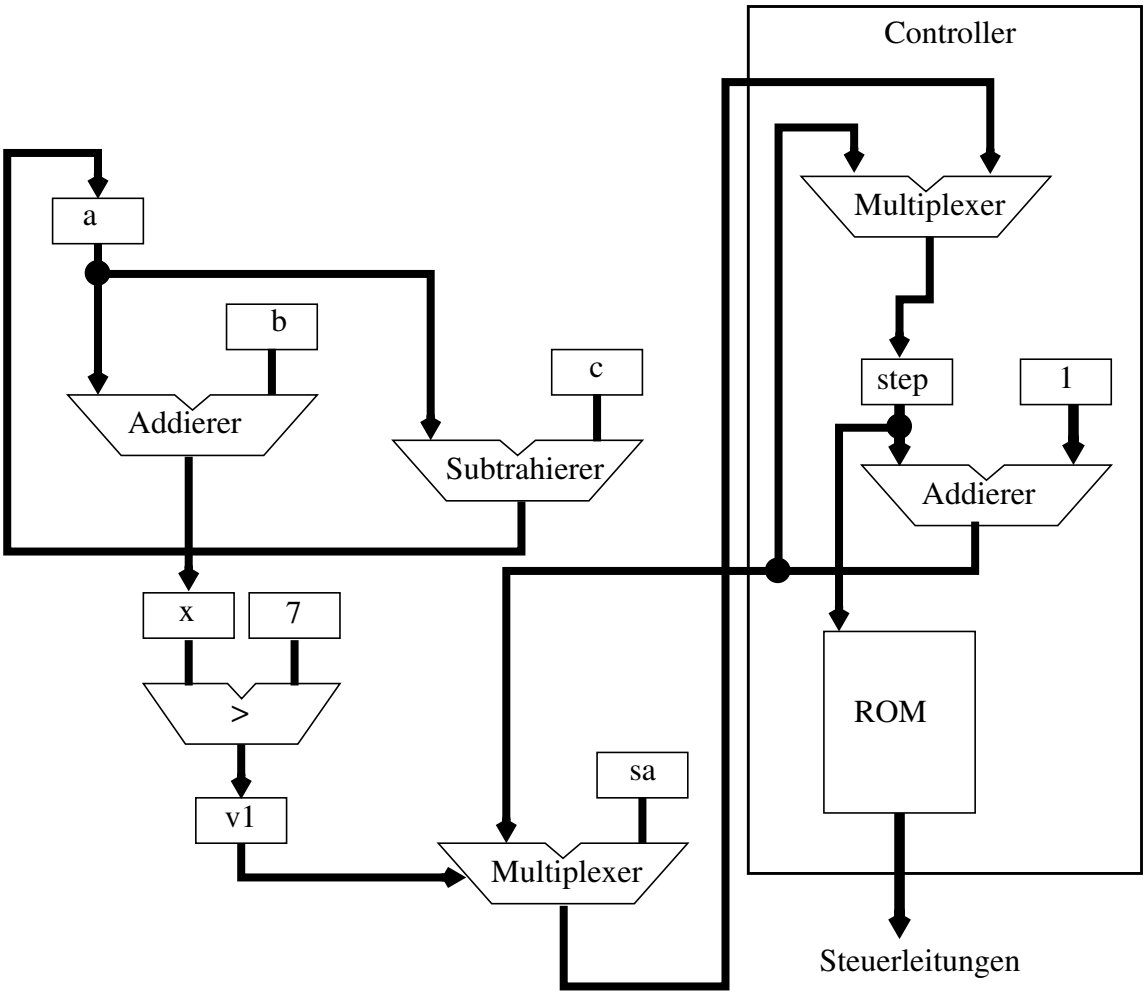



Bild 70: Realisierung einer Schleife

erst dann mit einem Wert belegt werden, wenn die Einordnung der Befehle in Kontrollschritte mit einem Schedulingalgorithmus erfolgt ist. Je nachdem, welche Bedingung gegeben ist, schaltet der Multiplexer entweder den am Eingang eins oder zwei anliegenden Wert weiter. Die Variable *step* wird also entweder um eins erhöht, was durch den im Controller liegenden Addierer geschieht, oder auf den ersten Kontrollschritt gelegt, in dem der erste Befehl der Schleife ausgeführt werden soll. Die Steuerleitungen, die jedes Register und die Bauteile ansteuern, sind hier der Übersicht halber nur angedeutet. Ein Multiplexer, welcher in dem Kontroller liegt, erlaubt die externe Veränderung der Variablen *step* und wird immer dann auf ‚extern' geschaltet, wenn ein Befehl vorkommt, der die Variable *step* als Ausgang benutzt. Der wesentliche Vorteil, Schleifen in dieser Form darzustellen, ist die einheitliche Darstellung der Befehle im definierten Format. Das Zwischenformat braucht nicht um Kontrollstrukturen erweitert zu werden. Außerdem ist der Nachteil, der durch die Betrachtung der Schleife als unabhängigem Block zustande kommt, hiermit aufgelöst. Durch ein einheitliches Zwischenformat kann eine Schaltung als

Ganzes betrachtet werden, was gerade beim Assignment zu besseren Ergebnissen führt. Bei allen bisher betrachteten Beispielen wurde nicht auf Optimalität bzgl. Zeit und Platzbedarf geachtet. Sie sollten nur die generelle Vorgehensweise veranschaulichen.

### 9.5.4 Ein- und Ausgabebefehle

Eine besondere Stellung im Zwischenformat haben die Befehle, welche die Ein- und Ausgänge einer Schaltung darstellen. Es gibt besonders bezeichnete Befehle *in* und *out*, die keinem Bauteil zugeordnet werden. Am Anfang jeder Schaltungsbeschreibung muß ein *in*-Befehl stehen, der die Eingangsvariablen definiert. Der *out*-Befehl am Ende einer Schaltungsbeschreibung gibt die Ausgangsvariablen an. Bild 71 zeigt die Definitionen der Befehle.

*((), (a, b, c), in)*

*((c, d, e), (), out)*

Bild 71: Ein- und Ausgänge

Der Befehl *in* ist so definiert, daß er die Eingangsvariablen erzeugt, wobei der Ausgangsbefehl die Ausgangsvariablen konsumiert. Durch diese Definition ist es möglich, die Struktur der Befehle beizubehalten. Jede in einer Schaltung vorkommende Variable muß erzeugt und konsumiert werden. Im nächsten Abschnitt wird dargestellt, daß der Eingangs- und Ausgangsbefehl ebenfalls Bauteilen zugeordnet wird, die aber eher eine virtuelle als eine reale Funktion haben.

Für die Synthese muß es eine Menge Bauteile geben, welche die Befehle ausführen können. Die Bauteile werden durch den Allocationsprozeß, welcher aus einer Bauteilbibliothek eine gewisse Menge an Bauteilen auswählt, zur Verfügung gestellt. Der Assignmentprozeß ordnet dann jedem Befehl ein Bauteil zu, welches die entsprechende Funktion zur Verfügung stellt. Wie oben schon dargestellt ist, werden auch *if*-Anweisungen und bedingte Schleifen genauso wie arithmetische und logische Befehle betrachtet. Somit müssen auch Multiplexer in das folgende Konzept der Bauteildefinition mit eingeordnet werden. Ein Multiplexer stellt also im Folgenden ein Bauteil dar, welches die Funktion *if* ausführen kann. Im folgenden Abschnitt wird eine Definition der Bauteile und der Bauteilbibliothek geliefert.

## 9.6 Darstellung der Bauteile

### 9.6.1 Einleitung

Im Abschnitt 9.6.3 wird eine allgemeine Definition der zur Verfügung stehenden Bauteile gegeben. Es wurde versucht, diese Definition so allgemein zu halten, daß sich sehr verschiedene real existierende Bauteile, die für die Synthese von Bedeutung sind, damit darstellen lassen. Daher ist es sinnvoll, an dieser Stelle einiges über Eigenschaften real existierender Bauteile darzustellen. Im allgemeinen kann gesagt werden, daß jedes Bauteil eine oder verschiedene Operationen durchführen kann, die über Steuerleitungen ausgewählt werden. Hier ergeben sich viele Möglichkeiten der Implementierung mit unterschiedlicher Auswirkung auf das Zeitverhalten, den Platzbedarf und den Energiebedarf einer Schaltung. Zum einen gibt es verschiedene Algorithmen zur Implementierung der gleichen Operationen, die sich bzgl. der Laufzeit und der Anzahl benötigter Gatter, also des Platzbedarfs, unterscheiden. Des weiteren kann bzgl. der zugrundeliegenden Technologie unterschieden werden. Zum Beispiel ist es möglich, mit einem BiCMOS-Fertigungsprozeß [38] sowohl Bipolar-Gatter, welche sich durch eine höhere Geschwindigkeit bei größerem Flächen- und Energiebedarf auszeichnen, als auch CMOS-Gatter zu nutzen, die bedeutend kleiner sind. Des weiteren gibt es auch Gatter, welche sowohl Bipolar

als auch MOS Transistoren beinhalten. Es muß noch zwischen den verschiedenen Bipolar Techniken unterschieden werden, wie z. B. TTL oder ECL. In Bild 72 wird für das Beispiel des Addierers verdeutlicht, welche Unterscheidungen getroffen werden können, wobei die Möglichkeiten nicht vollständig wiedergegeben werden können. Die in Bild 72 genannten Algorithmen sind [112] entnommen. An dieser Stelle soll nur deutlich gemacht werden, wie umfangreich eine Bauteilbibliothek werden kann. Die Additionsalgorithmen zu beschreiben, würde den Rahmen dieser Arbeit sicherlich sprengen.

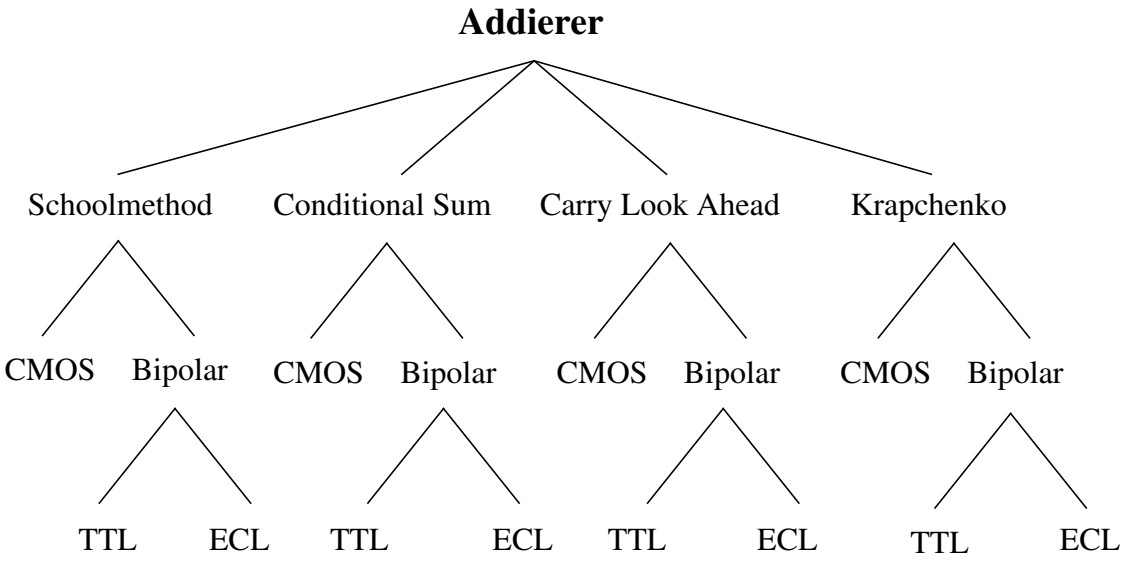



Bild 72: Implementierungsmöglichkeiten eines Addierers

Ähnliche Überlegungen können nun für andere Bauteile wie Multiplizierer, Dividierer und Subtrahierer angestellt werden. Bei der Konstruktion von ALUs werden mehrere Funktionen zusammengefaßt, wobei hier beliebige Kombinationen der Funktionen, die in einem Bauteil zur Verfügung stehen, vorstellbar sind. So ist es für die meisten Fälle nicht unbedingt günstig, eine sehr komplexe ALU zu verwenden, bei der viele Einheiten oft brachliegen. Andrerseits können Funktionen so geschickt in eine ALU integriert werden, daß eine große Platzersparnis gegenüber der Summe des Platzbedarfs der Elemente mit einer Funktion entsteht. Ein Addierer kann durch das Hinzufügen von wenigen Gattern zum Subtrahierer erweitert werden. Die Entscheidung über die Auswahl der Bauteile, die in einer Schaltung benutzt werden, ist im wesentlichen von den Zeit- und Platzanforderungen abhängig. Außerdem muß eine optimale Abbildung des zu realisierenden Algorithmus gewährleistet werden. Schon an diesen wenigen Beispielen ist ersichtlich, daß die Auswahl der Bauteilmenge, also der benutzten Teilmenge der in einer Bibliothek zur Verfügung stehenden Bauteile, entscheidende Optimierungsmöglichkeiten für den Entwurf digitaler Schaltungen bietet. Dies wird im Rahmen dieser Arbeit deshalb genauer betrachtet. Ein Bauteil ist in unserem Falle gegeben durch eine Menge von Eingängen und Ausgängen und eine Menge von Operationen, die dieses Bauteil beherrscht. Jedem Bauteil werden Parameter zugewiesen, die Auskunft über den Zeitbedarf der einzelnen Operationen und über den Platzbedarf des Bauteils geben. Wie schon dargestellt, kann es sein, daß verschiedene Additionsalgorithmen in den Bauteilen implementiert sind, welche mit unterschiedlichem Ressourcenaufwand und unterschiedlichem Zeitbedarf addieren können. Oder es sind arithmetische Einheiten vorhanden, die mehrere verschiedene Operationen ausführen können. Um die im folgenden gemachte Darstellung der Bauteile plausibel zu machen, wird hier kurz dargestellt, wie

der Zeitbedarf einer Operation zu unterscheiden ist. Hierzu ist ein kurzer Abstecher zur Prozessorarchitektur [11] nötig, wobei speziell das Thema Pipelining interessant ist.

### 9.6.2 Pipelining

Ein in neueren Prozessoren oft zur Geschwindigkeitssteigerung angewandtes Prinzip ist das des Pipelining, welches auf der Tatsache beruht, daß Befehle in einem Verarbeitungszyklus durch mehrere verschiedene Verarbeitungsstufen laufen müssen. Befehle werden also wie an einem Fließband in mehreren Stufen verarbeitet. Dies kann man sich zunutze machen und dadurch eine Parallelisierung der Verarbeitung erreichen. In einem getakteten Prozessor wird, nachdem die Pipeline einmal gefüllt wurde, in jedem Taktschritt ein Befehl fertig. Der Zeitbedarf für einen Befehl eines Maschinenprogramms, welches aus *n* Befehlen besteht, konvergiert also mit der Anzahl der Befehle, die bearbeitet werden, gegen eins. Besteht eine Verarbeitungseinheit in einem Prozessor aus *m* Pipelinestufen, so wird ein Befehl erst *m* Takte, nachdem er eingelesen wurde, fertig. Hierdurch ergibt sich für Sprungbefehle und Berechnungen die Konsequenz, daß ein Ergebnis, welches berechnet wird, nicht im direkt folgenden Befehl benutzt werden darf, da es sich hierbei noch um einen veralteten Wert handelt. Bei Sprüngen werden die *m-1* Befehle, die direkt hinter dem Sprungbefehl stehen, noch mit ausgeführt, bevor der Sprungbefehl ausgeführt wird. Dies hat Konsequenzen für einen Compiler, der diese Randbedingungen eines Prozessors mit Pipelineausführung mit berücksichtigen muß. Im Normalfall wird eine Pipeline über die Angabe ihrer Stufen definiert, wobei eine Stufe in genau einem Taktschritt abgearbeitet wird. An dieser Stelle soll nun eine allgemeine Definition für Pipelinebausteine erfolgen. Es erweist sich als sinnvoll, für einen Baustein zwei bzw. drei Zeitangaben für jede Funktion, die von diesem Bauteil ausgeführt werden kann, zu definieren: zum einen die Verzögerungszeit, also die Zeit, die auch in allen anderen Betrachtungen berücksichtigt wird. Die Verzögerungszeit gibt an, wie lange eine Funktion braucht, um berechnet zu werden. Des weiteren wird die Latenzzeit, also beim Pipelining die Zeit, bis die nächste Berechnung starten kann, angegeben. Für die hier synthetisierten Bauteile hat es sich als sinnvoll erwiesen, zum dritten die Zeit zu betrachten, in der der Eingang eines Bausteins mit einem festen Eingangswert belegt werden muß. Werden zum Beispiel die Eingangswerte direkt in Registern gespeichert, so reicht es, die Eingangswerte genau einen Taktzyklus lang an den Eingang zu legen. Eine weitere Bedingung für die Bausteine, die wir betrachten und synthetisieren wollen, ist dadurch gegeben, daß die Pipelineeigenschaft so verstanden werden soll, daß der nächste Befehl zu einem beliebigen Zeitpunkt, der größer oder gleich der Latenzzeit nach dem Startzeitpunkt des aktuellen Befehls ist, gestartet werden kann.

Die Abbildungen Bild 73, Bild 74 und Bild 75 machen das Pipelining deutlich, welches einmal allgemein in Prozessoren angewandt wird, andrerseits aber in unserem Fall durch die synthetisierten Bausteine ermöglicht werden soll. Die synthetisierten Bausteine werden wieder als Grundelemente für weitere Syntheseschritte benutzt. In Bild 73 wird vierstufiges Pipelining dargestellt.

Wie aus der Abbildung ersichtlich ist, wird die Pipeline so aufgebaut, daß jeder Pipelineschritt genau einen Taktschritt beträgt. In der Bild 74 wird eine Erweiterung dargestellt. Hier wird jeder Befehl nach *n* Taktschritten fertig, und der nächste Befehl kann erst nach *m* Taktschritten gestartet werden.

In der Bild 75 wird hiervon die Verallgemeinerung dargestellt, welche besagt, daß der nächste Befehl frühestens nach *m* Taktschritten gestartet werden kann. Dieser Fall kann dann eintreten, wenn die Verarbeitungseinheiten, welche in einer Pipeline benutzt werden, unterschiedliche Ausführungszeiten benötigen. Es muß eine Synchronisierung der Verarbeitungseinheiten mit dem gegebenen Taktraster geschehen. Beispielsweise gibt es Verarbeitungseinheiten, die sich

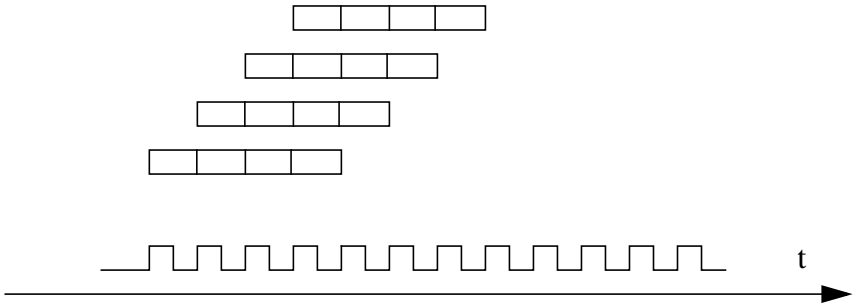



Bild 73: Pipelining in vier Stufen

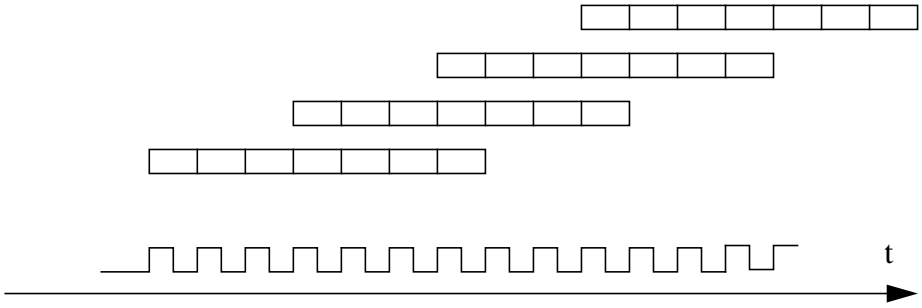



Bild 74: Pipelining in n = 7 und m = 3

nicht oder sehr schlecht in weitere Einheiten unterteilen lassen. Wenn deren Zeitbedarf größer ist als das festgelegte Taktraster, dann muß diese Einheit erst wieder frei werden, bevor der nächste Befehl bearbeitet werden kann. Das *m* läßt sich also auch als der größte vorkommende Zeitbedarf der Pipelineeinheiten interpretieren.

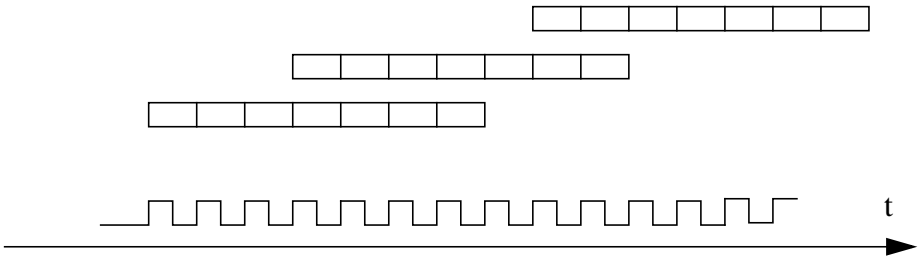



Bild 75: Pipelining in n = 7 und m > 2

Die Latenzzeit kann also in dem dargestellten Fall größer als zwei sein, muß aber kein Vielfaches von 3 sein. Damit ist eine sehr viel größere Flexibilität bei der Nutzung solcher Bausteine möglich. Andrerseits können herkömmliche Pipelinebausteine, die ja eine Latenzzeit von einem Taktschritt haben, ebenfalls in diese Definition eingeordnet werden. Um solche Bausteine benutzen zu können, müssen diese beiden Zeiten bekannt sein und berücksichtigt werden. Des weiteren muß der Zeitraum berücksichtigt werden, in der die Eingabe korrekt anliegen muß. Bei dem in der Bild 75 dargestellten Pipeline-Baustein könnte man sich vorstellen, daß die Eingabe direkt im ersten Schritt in ein internes Register übernommen wird. Somit müssen an den Eingängen des Bauteils nur im ersten Taktschritt konstante Werte anliegen.

### 9.6.3 Definition der Bauteile

Aufgrund der im letzten Abschnitt dargestellten Bedingungen kann die Bauteilbibliothek wie

folgt definiert werden, wobei eine möglichst reale Darstellung eines Bauteils angestrebt wird.

*Definition 9.6-1 Ein Bauteil ist als 4n+1-Tupel folgendermaßen definiert:*

$$(o_0, \ldots, o_{n-1}, t_{10}, \ldots, t_{1(n-1)}, t_{20}, \ldots, t_{2(n-1)}, t_{30}, \ldots, t_{3(n-1)}, s) \quad . \\ \in OP^n \times N^{3n} \times \Re$$

*Dabei haben die einzelnen Elemente des 4n+1-Tupels die folgende Bedeutung: die ersten n Elemente $o_i$ geben die n Operationen an, die mit diesem Bauteil ausgeführt werden können. Die Elemente $t_{1i}$ geben den Zeitbedarf der Operationen $o_i$ an, also die Zeit, bis das Bauteil nach dem Start einer Operation das Ergebnis liefert. Die Elemente $t_{2i}$ geben den Zeitbedarf an, den das Bauteil benötigt, bis es für eine neue Eingabe bereit ist. Die Elemente $t_{3i}$ geben an, wie lange die Eingabe anliegen muß. Das letzte Element s gibt den Flächen- bzw. Ressourcenbedarf des Bauteils an.*

Der Zeitbedarf kann in dieser Definition als ganze Zahl mit beliebiger Zeiteinheit dargestellt werden. Mit dieser Definition sind viele Bauteile definierbar, wobei sicherlich noch feinere Unterscheidungen denkbar sind, indem weitere Parameter definiert werden, wie z.B. der Leistungsbedarf eines Bauteils für die Ausführung einer Operation. Diese Arbeit soll mit den dargestellten Definitionen vor allem das Verfahren und die grundsätzlichen Ideen deutlich machen. Eine exaktere Darstellung eines Bauteils ist z.B. in [68] gegeben, wobei es sich aber bei sehr viel höherem Aufwand in der Bauteilbeschreibung nur um marginale Verbesserungen handelt. In Tabelle 3 werden beispielhaft verschiedene Bibliothekselemente dargestellt. Addierer können auf unterschiedliche Weisen aufgebaut werden. Der einfache sequentielle Addierer besteht nur aus hintereinandergeschalteten Volladdierern. Die Ausführungszeit ist dabei zwar ziemlich hoch, aber es ist nur ein minimaler Hardwarebedarf nötig. In der Tabelle 3werden nur relative Werte angeben, die als Beispiele dienen sollen. Wird eine Taktlänge festgelegt, so kann der Zeitbedarf bzgl. dieser Taktlänge in Taktschritten angegeben werden. Der Ressourcenbedarf kann als Flächenbedarf in einer beliebigen Einheit aufgefaßt werden. Das Beispiel macht

**Tabelle 3: Beispiel einer Bauteilbibliothek**

| Bezeichnung | Hardwarebedarf | Funktionen | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|---|---|
| Addierer | 5 | + | 3 | 3 | 3 |
| Addierer | 8 | + | 2 | 2 | 2 |
| Addierer | 11 | + | 1 | 1 | 1 |
| Subtrahierer | 7 | - | 3 | 3 | 3 |
| ALU | 9 | + , - | 3 | 3 | 3 |
| ALU | 13 | + , - | 1 | 1 | 1 |
| Multiplizierer | 20 | * | 12 | 6 | 1 |
| Dividierer | 25 | / | 20 | 10 | 1 |

deutlich, daß es viele Möglichkeiten gibt, eine Addition und eine Subtraktion zu realisieren. In

realen Systemen gibt es mindestens ebensoviele Möglichkeiten für die Realisierung der Multiplikation und der Division.

*Definition 9.6-2 Die Menge der Bauteile* $BLIB \subseteq \bigcup_{j=1}^{m} (OP^j \times N^{3j} \times \Re)$ *ist ein Element der Potenzmenge* $2^{(OP^* \times N^* \times \Re)}$ *für ein festzulegendes m. Im weiteren wird diese Menge auch Bauteilbibliothek genannt.*

Die Menge *BLIB* beinhaltet also die vorhandenen Bauteiltypen und enthält jedes vorhandene Bauteil nur einmal. Natürlich ist der Zugriff auf die Daten der Bauteile wichtig. Um dies darstellen zu können, werden entsprechende Funktionen definiert.

*Definition 9.6-3 Die überladenen Funktion* $f:BLIB \rightarrow 2^{OP}$ *gibt die Menge der Operationen, die ein Bauteil beinhaltet zurück. Die Funktion* $f_i:BLIB \rightarrow OP$ *gibt die i-te Operation zurück.*

Um die Lesbarkeit zu vereinfachen, werden Funktionen überladen, die eindeutig anhand des Operanden zu identifizieren sind.

*Definition 9.6-4 Der Zeitbedarf des Bauteils bzgl. einer Operation wird durch die Funktionen* $T_1:BLIB \times OP \rightarrow N$ , $T_2:BLIB \times OP \rightarrow N$ *und* $T_3:BLIB \times OP \rightarrow N$ *berechnet.*

Hier wird die oben gemachte Unterscheidung in drei Zeiten genutzt.

*Definition 9.6-5 Der Ressourcenbedarf wird durch* $r:BLIB \times OP \rightarrow \Re$ *berechnet.*

*Definition 9.6-6 Die Anzahl der Funktionen, die durch ein Bauteil zur Verfügung gestellt werden, sei mit* $\delta_f:OP \rightarrow N$ *bezeichnet.*

Diese Funktion gibt also für ein Bauteil das oben benötigte *n* an.

Was diese Definition nicht leisten kann, ist, eine ALU zu beschreiben, welche eine echte Parallelverarbeitung ermöglicht. Denn es gibt z.B. ALUs, deren Multiplikation acht Taktschritte benötigt, wobei diese parallel zu vier Additionen ausgeführt werden kann, die jeweils nur zwei Taktschritte benötigen. In [39] ist eine entsprechende ALU entwickelt und beschrieben worden. Um die Definition zu verdeutlichen, soll hier beispielhaft der Multiplizierer aus Tabelle 3 dargestellt werden, er läßt sich durch das Tupel $Multiplizierer = (*, 12, 6, 1, 20)$ beschreiben. Da im Rahmen vieler Synthesesysteme eine Technologieunabhängigkeit gewünscht wird, kann der Zeit- und Platzbedarf auch mit relativen Einheiten belegt werden. So wird zum Beispiel in [112] der Platzbedarf anhand der Anzahl der benötigten Gatter dargestellt, wobei der Zeitbedarf an der Anzahl der Gatterlaufzeiten gemessen wird. Durch die Darstellung der Komplexität, also des Zeit- und Platzbedarfs von Bauteilen bzw. Funktionen, wie dies auch in der Komplexitätstheorie üblich ist, ist es möglich, die Optimierung unabhängig von der konkreten Technologie durchzuführen. Die Länge eines Taktschrittes wird ebenfalls entsprechend angepaßt, so daß sie nicht absolut sondern relativ definiert wird. Im folgenden ist mit einem Kontrollschritt genau ein Taktschritt gemeint, so daß diese Begriffe synonym verwendet werden. In vielen Synthese-

systemen wird davon ausgegangen, daß jede Funktion in einem Kontrollschritt abgearbeitet werden kann, was aber hier nicht der Fall sein muß. Der große Vorteil ist der, daß schnelle Bauteile nicht auf langsame warten müssen, was zu einer insgesamt höheren Geschwindigkeit führen kann. Besonders zu erwähnen sind die beiden Bauteile, die die Ein- und Ausgänge realisieren. Diese Bauteile müssen nur jeweils einmal vorhanden sein, und ihnen werden die Befehle *in* und *out* zugewiesen.

Wie schon erwähnt, wird durch die Allocation eine gewisse Menge an Bauteilen ausgewählt und zur Verfügung gestellt. Die Allocation kann also wie folgt definiert werden.

### 9.7 Definition der Allocation

*Definition 9.7-1 Die Allocation ist eine linkseindeutige Relation* $AC{:}BLIB \rightarrow BA$. *Sie bildet aus der Bibliothek BLIB die Menge der Bauteilinstanzen BA.*

Durch die Definition der Allocation ist ein Instrument vorhanden, welches, ähnlich der Deklaration von Variablen in Programmiersprachen, eine Instanzenmenge aus der vorhandenen Bibliothek bildet. Die so deklarierten Instanzen werden Beispielsweise durch ihre Namen oder durch eine eindeutige Numerierung unterschieden. Die Umkehrfunktion der Allocation liefert den Typ einer Bauteilinstanz zurück, und damit kann auf die Parameter des Bauteils zugegriffen werden.

*Definition 9.7-2 Der zu einer Instanz gehörende Typ wird durch die Funktion* $bau{:}BA \rightarrow BLIB$ *geliefert, sie ist definiert als* $bau(ba) = AC^{-1}(ba)$.

*Definition 9.7-3 Die Operationen einer Instanz eines Bauteils werden mit der überladenen Funktion* $f{:}BA \rightarrow 2^{OP}$ *geliefert, wobei* $f_i{:}BA \rightarrow OP$ *die i-te Operation liefert. Das heißt also, daß mit* $ba \in BA$ *die Funktion* $f(bau(ba))$ *auch geschrieben wird als* $f(ba) = f(bau(ba))$ *und die Funktionen* $f_i(bau(ba))$ *als* $f_i(ba) = f_i(bau(ba))$.

Hier werden Funktionen wieder überladen, was zu einer vereinfachten Schreibweise und besseren Lesbarkeit führen soll.

*Definition 9.7-4 Der Zeitbedarf einer Instanz eines Bauteils bzgl. einer Operation wird mit den überladenen Funktionen* $T_1{:}BA \times OP \rightarrow N$, $T_2{:}BA \times OP \rightarrow N$ *und* $T_3{:}BA \times OP \rightarrow N$ *berechnet.*

*Definition 9.7-5 Der Ressourcenbedarf wird mit* $r{:}BA \rightarrow \Re$ *berechnet.*

Auch hier sollte eindeutig sein, was gemeint ist. Die Eigenschaften der Typen werden vollständig auf die Instanzen übertragen.

*Definition 9.7-6 Eine allocierte Bauteilmenge BA ist vollständig bzgl. eines Programms* $PR = (b_1, b_2, \ldots, b_n) \in B^n$, *wenn gilt* $\forall(0 < i \leq n)\exists(ba \in BA)(f(b_i) \in f(ba))$.

Durch die letzte Definition ist die Aufgabe des Allocationsprozesses eindeutig festgelegt. Er muß eine Menge *BA* finden, welche vollständig ist bzgl. eines Programms *PR*.

*Definition 9.7-7 Die Menge aller gültigen Instanzenmengen bzgl. eines Programms* $PR = (B, <)$ *und einer Bauteilbibliothek BLIB ist durch*

$$\mathfrak{A}L(PR, BLIB) = \{BA \mid \forall (b \in B)\exists (ba \in BA)(f(b_i) \in f(ba))\}$$ *definiert.*

Als Beispiel werden die beiden Bauteile $ALU = (+, *, 300, 600, 300, 400, 100, 100, 2)$ und $ADD = (+, 200, 200, 200, 1)$ definiert und stehen in der Bauteilmenge zur Verfügung. Hierbei kann die Menge der instanziierten Bauteile als $BA = \{ALU, ADD\}$ aufgefaßt werden. Wobei die Bibliothek der Bauteile z.B. durch die folgende Menge gegeben sein kann:
$BLIB = \{(+, 200, 200, 200, 1), (+, *, 300, 600, 300, 400, 100, 100, 2),$
$(-, 300, 300, 100, 1), (+, 400, 400, 200, 1)\}$

Bei der so gemachten Definition wird auf die Vergabe eines Namens für die Bauteile der Bibliothek verzichtet. Mit dieser Bauteilbibliothek gilt dann für die allocierten Bauteile beispielsweise $f(ALU) = f(bau(ALU)) = \{+, *\}$ und $f_0(ALU) = f_0(bau(ALU)) = +$.

Der Zeitbedarf ist dann z.B. durch $T_1(ALU, +) = T_1(bau(ALU), +) = 300$ gegeben.

Das wesentliche Problem bei der Allocation ist dadurch gegeben, daß eine möglichst gute Lösung gefunden werden muß, so daß die Syntheseergebnisse allen Randbedingungen genügen. Steht eine Auswahl an Bauteilinstanzen zur Verfügung, dann wird jedem Befehl ein Bauteil zugeordnet, welches diesen auszuführen hat. Das heißt, es müssen auch Bauteile vorhanden sein, die z.B. einen IF-Befehl realisieren können. Eine Zuweisung ist dabei auch als Operation definiert, welche durch ein Bauteil realisiert werden muß. Dieses besteht dann natürlich nur aus einer Verbindung zwischen dem Eingang und dem Ausgang. An anderer Stelle werden die Bauteile für spezielle Befehle noch einmal explizit definiert, wodurch die Semantik der Befehle gegeben ist. Nachdem eine Bauteilmenge zur Verfügung gestellt worden ist, kann jeder Befehl einem Bauteil zugewiesen werden, wofür der Assignmentprozeß zuständig ist.

## 9.8 Definition des Assignment

Der Assignmentprozeß ordnet jedem Befehl eines Programms *PR* ein Bauteil zu. Dabei ist darauf zu achten, daß die Operation des Befehls auch vom zugeordneten Bauteil ausgeführt werden kann. Im folgenden ist mit der Befehlsmenge *B* immer genau die Menge an Befehlen gemeint, die auch im Programm vorhanden ist. Da für das Assignment die Reihenfolge der Befehle unerheblich ist, kann auf *PR* verzichtet und im folgenden mit *B* gearbeitet werden.

*Definition 9.8-1 Für jedes Bauteil einer Instanzenmenge bzw. der Bibliothek ergibt sich eine Menge der Befehle, die auf diesem Bauteil ausführbar sind. Die Funktion* $a_{BA}: B \to 2^{BA}$ *liefert für jeden Befehl die Bauteile, die dessen Operation ausführen können, also mit* $b \in B$ *folgt* $\mathfrak{a}_{BA}(b) = \{u \mid (u \in BA) f(b) \in f(u)\}$. *Steht die Bauteilmenge fest, wird nur a geschrieben.*

Es sei besonders auf die überladene Funktion *f* hingewiesen, die einmal ein Element und das andere Mal eine Menge liefert. Durch die definierte Funktion kann also für jeden einzelnen Befehl herausgefunden werden, welchen Bauteilen er zugeordnet werden darf.

*Definition 9.8-2 Eine konkrete, also eindeutige Zuordnung eines Befehls zu einem Bauteil ist durch eine Funktion* $A: B \to BA$ *gegeben, die mit* $b \in B$ *die Bedingung*

$A(b) \in a(b)$ *erfüllen muß.*

Diese Funktion definiert das Assignment. Durch die Definition der Assignmentfunktion A mit Hilfe der Funktion a kann jeder Befehl nur genau einem Bauteil zugeordnet werden und es wird für jedes Assignment die Gültigkeit sichergestellt. Da die Gültigkeit oder Korrektheit eines Assignment impliziert wird, wird sie hier nicht noch einmal explizit definiert. Da es viele verschiedene Möglichkeiten gibt, die Befehle den Bauteilen zuzuordnen, wird hier die Menge der gültigen Assignments angegeben. Durch die Größe dieser Menge ist es oft nicht möglich, diese explizit anzugeben.

*Definition 9.8-3 Die Menge der möglichen Zuordnungen, die sich aus der Funktion a ergeben, bezeichnen wir als* $L(a)$ *oder als* $L(B, BA)$ *bzgl. B und BA, und sie ist definiert durch* $L(a) = \{A \mid \forall (b \in B) A(b) \in a(b)\}$.

Das Ziel eines guten Assignmentalgorithmus kann dadurch spezifiziert werden, daß ein möglichst optimales Assignment bzgl. einer Bewertungsfunktion gesucht wird. Dies ist sicherlich keine einfache Aufgabe, denn die Kardinalität der Menge *L(a)* ist sehr groß und läßt sich mit der folgenden Formel berechnen:

*Satz 9.8-4 Die Anzahl der möglichen Assignments kann berechnet werden durch*

$$|L(B, BA)| = \prod_{b \in B} |a(b)|.$$

Die Anzahl der möglichen Zuordnungen wächst also exponentiell mit der Anzahl der Befehle. Durch die Zuordnung der Befehle zu den Bauteilen werden die Eigenschaften der Bauteile auf die Befehle übertragen. Im besonderen ist der Zeitbedarf eines Befehls von der Realisierung der Funktion in dem zugeordneten Bauteil abhängig.

*Definition 9.8-5 Der Zeitbedarf eines Befehls* $b \in B$ *bzgl. einer Zuordnung* $A \in L(a)$ *wird mit* $T_1: B \times L(a) \to N$, $T_2: B \times L(a) \to N$ *und* $T_3: B \times L(a) \to N$ *bezeichnet und ist definiert durch* $T_1(b, A) = T_1(A(b), f(b))$, $T_2(b, A) = T_2(A(b), f(b))$ *und* $T_3(b, A) = T_3(A(b), f(b))$. *Auf der rechten Seite sind jeweils die oben schon gegebenen Funktionen gemeint. Ist die Zuordnung A eindeutig gegeben, dann kann auch* $T_1(b)$, $T_2(b)$ *und* $T_3(b)$ *geschrieben werden.*

Jedem Befehl wird durch diese Definition der aufgrund eines Assignments benötigte Zeitbedarf zugeordnet. Diese Funktionen sind wieder als überladene Funktionen dargestellt, um eine einfachere Handhabung zu gewährleisten.

### 9.8.1 Bauteilabhängigkeit

Durch die Definition des Assignments ergibt sich zusätzlich zu den schon definierten Daten-, Antidaten- und Ausgabeabhängigkeiten die Bauteilabhängigkeit. Sie existiert für jeweils zwei Befehle, die dem gleichen Bauteil zugeordnet sind. Die Motivation, den Begriff der Bauteilabhängigkeit einzuführen, ist damit begründet, daß zwei Befehle nicht gleichzeitig auf einem Bauteil ausgeführt werden können, so daß diese Abhängigkeit im Schedulingalgorithmus mit berücksichtigt werden muß [98]. Die Bild 76 zeigt einen Graphen für das in Bild 53 schon gegebene Beispiel, in dem die Daten-, Antidaten- und Ausgabeabhängigkeiten dargestellt sind.

An diesem Beispiel sollen die Bauteilabhängigkeit und die weiteren Definitionen verdeutlicht werden.

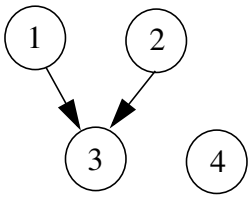



Bild 76: Datenflußgraph

*Definition 9.8-6 Die Zuordnung der Befehle zu den Bauteilen bildet eine Relation, die die Befehle in Äquivalenzklassen unterteilt und zwar in diejenigen Befehle, die durch das gleiche Bauteil ausgeführt werden. Für* $b_1, b_2 \in B$ *gilt* $b_1 \cong_A b_2 \Leftrightarrow A(b_1) = A(b_2)$. *Ist A eindeutig festgelegt, so kann auch* $b_1 \cong b_2$ *geschrieben werden.*

Werden in dem Beispiel die Befehle *1* und *2* dem Bauteil *A1* und die Befehle *3* und *4* dem Bauteil *A2* zugeordnet, so gelten die Äquivalenzen $1 \cong 2$ und $3 \cong 4$.

*Definition 9.8-7 Da die Relation* $\cong$ *symmetrisch, transitiv und reflexiv ist, bildet sie auf der Menge der Befehle Äquivalenzklassen, die mit* $\overline{b_A} = \{b_i \mid b \cong_A b_i\} \subseteq B$ *bezeichnet werden. Wenn die Zuordnung A festgelegt ist, wird hier auch* $\bar{b}$ *geschrieben.*

In der Bild 77 werden die in diesem Sinne äquivalenten Befehle mit ungerichteten gestrichelten Kanten verbunden.

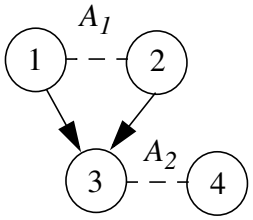



Bild 77: ‚Äquivalenzen' der Befehle

*Definition 9.8-8 Dir verschiedenen Abarbeitungsreihenfolgen der Befehle auf einem Bauteil werden als Permutationen dargestellt:* $p = (b_1, b_2, \ldots, b_m)$ *mit* $n = \left|\overline{b_A}\right|$ *und* $b_1, b_2, \ldots, b_n \in \overline{b_A}$ *sind alle Elemente von* $\overline{b_A}$ . *Die dazugehörige Ordnung auf den Befehlen läßt sich mit* $b_i <_p b_j \Leftrightarrow i < j$ *beschreiben und wird im weiteren als Bauteilabhängigkeit bezeichnet. Ist die Permutation für ein Bauteil* $u \in BA$ *eindeutig festgelegt, so kann auch* $b_i <_u b_j$ *geschrieben werden.*

Für jedes Bauteil sind in dem Beispiel zwei mögliche Reihenfolgen wählbar: das Bauteil *A1* hat als mögliche Permutationen *(1,2)* und *(2,1),* das Bauteil *A2* hat die Permutationen *(3,4)* und

*(4,3)*. Die Reihenfolge der Befehle, also die Bauteilabhängigkeit, läßt sich im Graphen durch gerichtete Kanten darstellen. Wählt man für jedes Bauteil im Beispiel jeweils die erste Permutation, dann ergibt sich der Graph in Bild 78 .

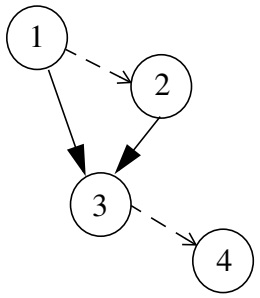



Bild 78: Bauteilabhängigkeiten als gerichtete Kanten

Der Begriff Bauteilabhängigkeit soll die Verwandtschaft zu den oben definierten Abhängigkeiten darstellen. Das heißt, es wird durch die Bauteilabhängigkeit eine weitere Abhängigkeit der Befehle untereinander impliziert, welche im weiteren ihre Berücksichtigung finden muß. Dies wird auch durch die Darstellung im Graphen deutlich.

*Definition 9.8-9 Die Menge der verschiedenen Reihenfolgen, also der Permutationen, sei mit* $PE(\overline{b_A})$ *bezeichnet und besteht aus* $\left|PE(\overline{b_A})\right| = \left|\overline{b_A}\right|!$ *Elementen. Mit* $b \in \overline{b_A}$ *und* $A(b) = u$ *kann auch* $PE(u) = PE(\overline{b_A})$ *geschrieben werden.*

Die Anzahl der möglichen Permutationen für ein einziges Bauteil läßt sich aus der Fakultät der Anzahl ihm zugeordneter Befehle berechnen. Dadurch ergibt sich für eine feste Zuordnung *A* die Gesamtanzahl verschiedener Möglichkeiten, die Bauteilabhängigkeiten zu definieren, folgendermaßen:

*Satz 9.8-10 Die Gesamtanzahl verschiedener Möglichkeiten von Bauteilabhängigkeiten für ein gegebenes Assignment A läßt sich mit* $\prod_{u \in BA} |PE(u)|$ *berechnen.*

Die Anzahl der verschiedenen Möglichkeiten, Permutationen für das angegebene Beispiel zu finden, ist also vier. Auch hier ist ein Entscheidungsspielraum gegeben, so daß eine Entscheidung getroffen werden muß, um die beste Lösung zu finden. Etwas erleichtert wird die Suche nach der besten Lösung durch die folgende Definition, die eine Verringerung der Anzahl der gültigen Permutationen in Aussicht stellt.

*Definition 9.8-11 Es gibt gültige und nicht gültige Permutationen. Eine Permutation* $p \in PE(\overline{b_A})$ *ist gültig genau dann, wenn für alle* $b_i, b_j \in \overline{b_A}$ *gilt* $(b_i, b_j) \in T_{<< \cup <_s} \Rightarrow b_i <_p b_j$.

Durch diese Definitionen werden zwei Randbedingungen abgefangen. Jede Zuordnung eines Befehls zu einem Bauteil muß gültig sein, das heißt, der Befehl muß auf dem Bauteil auch ausführbar sein. Werden zwei Befehle dem gleichen Bauteil zugeordnet, so können diese Befehle nicht gleichzeitig ausgeführt werden, und es muß eine Reihenfolge festgelegt werden, die aber den anderen Abhängigkeiten nicht widersprechen darf. Hierzu muß die allgemeinen Abhängig-

keit mit der Schleifenabhängigkeit vereinigt werden. Von dieser Vereinigungsmenge muß die transitive Hülle gebildet und eine topologische Ordnung [18] der Befehle gebildet werden. Werden die Permutationen, die anhand der Bauteilzuordnungen gebildet wurden, gemäß der topologischen Ordnung gebildet, so sind alle Permutationen gültig und es gilt der folgende Satz:

*Satz 9.8-12 Sei* $p_i \in PE(u_i)$ *für alle Bauteile* $u_1, \ldots, u_n \in BA$ *gültig, dann gilt für alle* $p_i$ *und die Befehle* $x, y \in B$*:*

$$x <_{p_i} y \Rightarrow (y, x) \notin T_{<< \cup <_S \cup <_{p_0} \cup \ldots \cup <_{p_{i-1}} \cup <_{p_{i+1}} \cup \ldots \cup <_{p_n}}$$

Faßt man die Abhängigkeiten wieder als Graph auf, so besagt die Bedingung: falls es eine gerichtete Kante zwischen zwei Befehlen $x, y \in B$ gibt, dann darf es keinen Weg in die entgegengesetzte Richtung, also von $y$ nach $x$ geben, was zu einem Kreis führen würde.


Bild 79: Permutationen werden gewählt und getestet

In Bild 79 ist ein Beispiel gezeigt, hier wird die Permutation $p_1 = (1, 2)$ für das erste Bauteil und die Permutation $p_2 = (3, 4)$ für das zweite Bauteil gewählt. Die entsprechenden Kanten werden in den Graphen eingefügt. Es zeigt sich, daß kein Kreis in dem Graphen entstanden ist, die Permutationen sind also gültig.

### 9.8.2 Ressourcenbedarf

Ist jeder Befehl einem Bauteil zugewiesen, so kann an dieser Stelle schon der Ressourcenbedarf berechnet werden. Der Ressourcenbedarf ist ein wesentlicher Faktor der Bewertung einer synthetisierten Schaltung [34].

*Definition 9.8-13 Der Bauteilbedarf, der durch ein Assignment A für die Realisierung der Befehlsmenge B spezifiziert wurde, ergibt sich als Funktion* $Bres: A \rightarrow \Re$ *und läßt sich berechnen als* $Bres(A) = \sum_{v \mid (\exists b \in B)(v = A(b))} r(v)$*. Es werden also nur die Bauteile betrachtet und in die Berechnung mit einbezogen, die auch tatsächlich benutzt werden.*

Diese Berechnung der Ressourcen ist natürlich unvollständig, denn hier wird nur die Zuordnung der Befehle zu den Bauteilen betrachtet. Sehr wichtig ist auch die Zuordnung der Variablen zu den Registern. Eine Zuordnung der Variablen zu Registern ist erst dann möglich, wenn das Scheduling abgeschlossen ist. Aus diesem Grunde wird an dieser Stelle das Scheduling definiert, um dann direkt das Registerassignment zu behandeln. Die differenzierte Betrachtung des Scheduling kommt erst in einem späteren Abschnitt.

### 9.9 Definition des Scheduling

Nun folgt der Schritt der Synthese, der als Ablaufplanung oder Scheduling bezeichnet wird. Nachdem jedem Befehl ein Bauteil zugeordnet ist, müssen die Befehle so in Kontrollschritte eingeordnet werden, daß alle Abhängigkeiten gewahrt bleiben. Dabei wird der Zeitbedarf der Befehle berücksichtigt, welcher sich aus der Bauteilzuordnung ergibt. Das Scheduling ist eine Abbildung der Befehle in Kontrollschritte. Die folgende Definition ist sozusagen die Grundanforderung an ein Scheduling.

*Definition 9.9-1 Ein Scheduling* $sch{:}B \rightarrow N$ *ist eine Funktion, die jedem Befehl einen Steuer- oder Kontrollschritt zuordnet, so daß mit* $b_i, b_j \in B$ *gilt:* $(b_i \ll b_j) \vee (b_i <_S b_j) \vee (b_i <_u b_j) \Rightarrow sch(b_i) < sch(b_j)$.

Das heißt, daß die Einordnung der Befehle in Kontrollschritte durch die allgemeine Abhängigkeit, die Schleifenabhängigkeit oder durch die Bauteilabhängigkeit impliziert wird, wobei das Scheduling keinesfalls eindeutig ist. Diese Definition muß für jedes Scheduling erfüllt sein. Eine genauere Spezifizierung des Scheduling ist durch den Zeitbedarf der Module gegeben, worauf später noch eingegangen wird. Nachdem das Scheduling durchgeführt wurde, wird jeder Variablen ein Register zugeordnet. Da die endgültigen Lebenszeiten der Variablen erst nach dem Scheduling bekannt sind, wird dieser Schritt nach dem Scheduling ausgeführt.

### 9.10 Variablen Register Assignment

Die folgenden Definitionen verdeutlichen die Zusammenfassung von mehreren Variablen in einem Register. Um Variablen zusammenfassen zu können, müssen die Lebenszeiten, welche als Intervalle dargestellt werden, bekannt sein. Die Lebenszeiten ergeben sich erst nach dem Scheduling.

*Definition 9.10-1 Eine Variable v ist als 3-Tupel* $v = (name, a, e)$ *dargestellt, wobei das Intervall der Lebenszeit* $[a, e]$ *definiert ist durch* $a = min\{s | s = sch(b) + T_1(b) \wedge v \in out(b)\}$ *für das erste Auftreten der Variablen und* $e = max\{s | s = sch(b) + T_3(b) - 1 \wedge v \in in(b)\}$ *für das letzte Auftreten der Variablen. Die Lebenszeiten einer Variablen werden mit den Funktionen* $anf{:}V \rightarrow N$ *und* $end{:}V \rightarrow N$ *abgefragt.*

Selbstverständlich muß jeder Variablen auch ein Typ zugewiesen werden, aber für die Darstellung der Verfahren kann davon abstrahiert werden. Werden Schleifen benutzt, so muß diese Definition erweitert werden, da innerhalb von Schleifen keine Variablen zusammengefaßt werden d ürfen und die Lebenszeiten der Variablen die gesamte Schleifenlaufzeit überdecken müssen.

*Definition 9.10-2 Sei eine Variable* $v = (name, a, e)$ *gegeben. Falls es eine Schleife, die durch einen Befehlsblock definiert ist, gibt mit* $sch(sa) \leq a \leq sch(se)$ *dann folgt* $a = sch(sa)$. *Falls gilt* $sch(sa) \leq e \leq sch(se)$ *folgt* $e = sch(se)$.

*Definition 9.10-3 Es gibt eine symmetrische Relation zwischen Variablen, die besagt, daß zwei Variablen im gleichen Register abgelegt werden können: für* $v_1, v_2 \in V$ *ist sie de-*

$$\textit{finiert durch } v_1 \equiv_V v_2 \Leftrightarrow (anf(v_2) > end(v_1)) \vee (anf(v_1) > end(v_2)).$$

Durch die letzte Definition wird deutlich, wann eine Zuordnung zweier Variablen zu einem einzigen Register möglich ist. Aus dieser Definition folgt dann der Satz folgende für die minimale Anzahl an benötigten Registern.

*Satz 9.10-4 Die minimale Anzahl benötigter Register n zur Realisierung einer Variablenmenge* $V' \subseteq V$ *entspricht der minimalen Größe einer Partition. Wobei* $pa_i \subseteq V'$ *mit* $1 \le i \le n$ *die Mengen der Partition sind, für die gilt:* $k \neq j \Rightarrow pa_k \cap pa_j = \{\ \}$ *und* $\bigcup_{i \le n} pa_i = V'$. *Für jedes Variablenpaar* $v_r, v_s \in V'$, *welches in der gleichen Partition liegt, gilt:* $v_r, v_s \in pa \Rightarrow v_1 \equiv_{V'} v_2$.

Die folgende Definition wird im weiteren benutzt um die minimale Anzahl benötigter Variablen darzustellen.

*Definition 9.10-5 Die Anzahl der benötigten Register zur Realisierung der Variablenmenge V wird durch die Funktion* $minv: 2^V \rightarrow N$ *berechnet.*

*Definition 9.10-6 Ist eine Partition festgelegt, so liefert die Funktion* $reg: V \rightarrow 2^V$ *die Teilmenge in der die Variable liegt, und damit das entsprechende Register.*

Die Definition ist konstruktiv, denn neben der reinen Anzahl der Variablen werden die Variablen, die zusammengefaßt werden können, in einer Partition untergebracht, so daß eine Teilmenge einem Register entspricht. Der left-edge Algorithmus beispielsweise sortiert die Variablen bzgl. ihrer Lebenszeiten und durchläuft dann jede Variable und faßt Variablen zusammen, für die die Relation gilt. In der Bild 80 ist ein Beispiel für die Einordnung von neun Befehlen gegeben, wobei rechts daneben die Lebenszeiten der Variablen aufgezeichnet sind. Der left-edge Algorithmus sortiert im ersten Schritt die Variablen nach den Lebenszeiten. Der Algorithmus macht dazu zwei Sortierungsläufe: der erste sortiert die Variablen anhand der Endzeiten in absteigender Reihenfolge; der zweite Durchgang des reihenfolgeerhaltenden Algorithmus sortiert nach den Startzeiten, und zwar in ansteigender Reihenfolge. Somit erhält man die in der Bild 80 angegebene Sortierung der Variablen. Wird nun die erste Variable gewählt und mit jeder Variablen verglichen, so kann die Variable *a* nur mit der Variablen *f* in einem Register untergebracht werden. Durch die Pfeile ist angedeutet, welche Variablen zusammengefaßt werden können.

### 9.11 Der Syntheseablauf

Nachdem die wesentlichen Schritte der Synthese definiert sind, wird hier der Ablauf der Synthese dargestellt. Im ersten Schritt wird eine Menge von Bauteilen zur Verfügung gestellt, also der Allocationsschritt durchgeführt. Hierbei ist die wesentliche Problematik dadurch gegeben, daß eine gute Auswahl der zur Verfügung stehenden Menge erst dann möglich ist, wenn die Ergebnisse der anderen Schritte bekannt sind. Somit ist eine Optimierung nur möglich, wenn iterativ alle Schritte der Synthese mehrfach ausgeführt werden.

Unter der Annahme, daß die Bauteilmenge *BA* allociert wurde, muß im nächsten Schritt die

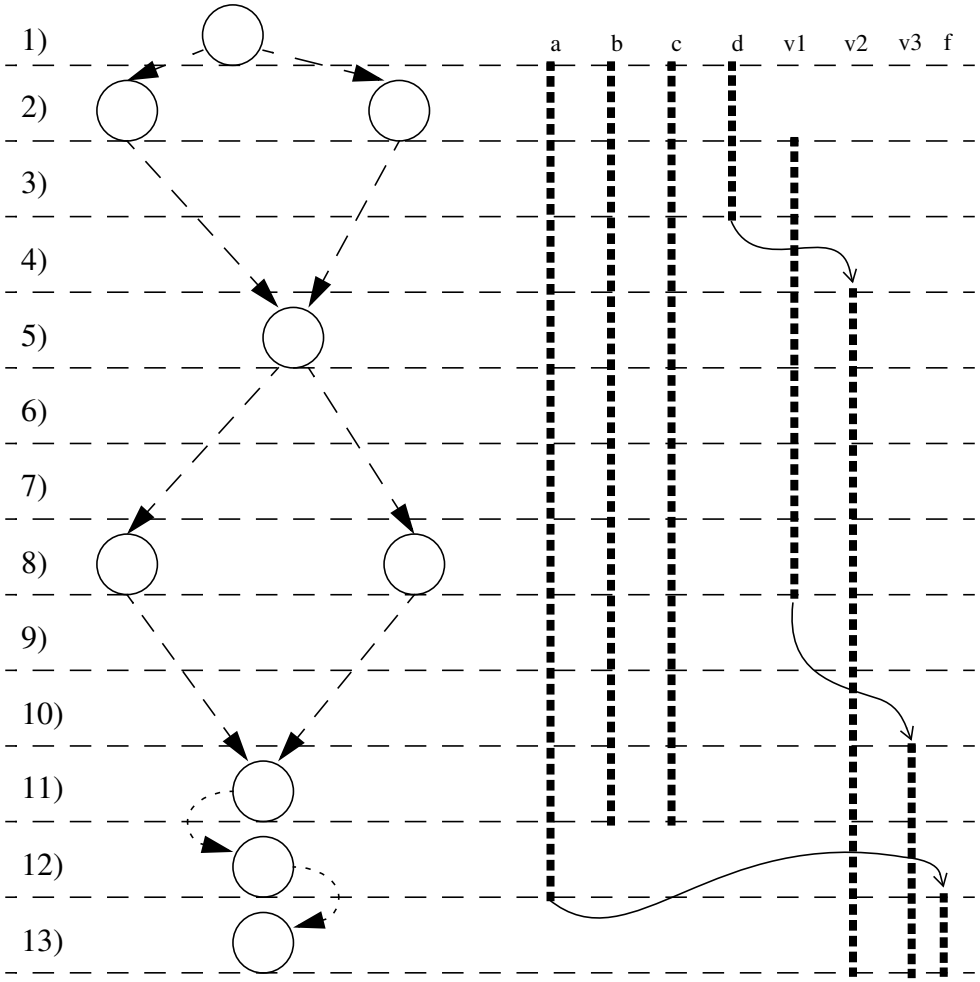



Bild 80: Der Left-Edge Algorithmus faßt Variablen zusammen

Vollständigkeit der allocierten Bauteilmenge bzgl. der Befehlsmenge *B* verifiziert werden. Nun kann eine eindeutige Zuordnung der Befehle zu den Bauteilen stattfinden. Der Assignmentprozeß realisiert diese Zuordnung, wobei die Zuordnung der Befehle zu Bauteilen eindeutig sein muß, was laut der Definition deren Korrektheit impliziert. Im letzten Schritt wird das Scheduling durchgeführt, was die Bauteilabhängigkeiten, die Schleifenabhängigkeiten, die Daten-, Antidaten- und Ausgabeabhängigkeiten berücksichtigt. Im folgenden Abschnitt wird genau auf das Scheduling eingegangen.

### 9.12 Scheduling

Das Scheduling wird allen definierten Abhängigkeiten gemäß durchgeführt. Jeder Befehl wird dabei einem Taktschritt zugeordnet, so daß alle durch die Abhängigkeiten gegebenen Bedingungen erfüllt sind. Ein Befehl darf erst dann ausgeführt werden, wenn alle von ihm benutzten Werte berechnet sind. Außerdem ist durch die Bauteilabhängigkeit eine gewisse Reihenfolge bei der Abarbeitung zweier Befehle festgelegt, die dem gleichen Bauteil zugeordnet sind. Dazu ist es wichtig zu wissen, wann der zweite Befehl auf einem Bauteil gestartet werden darf. Außerdem darf kein Befehl, der in einer Schleife vorkommt, vor dem ersten und nach dem letzten Schleifenbefehl ausgeführt werden. Befehle, die in der Schleife gestartet werden, müssen auch innerhalb der Schleife abgeschlossen werden. Im folgenden Abschnitt wird auf eben diese Problematik eingegangen und eine Funktion definiert, die diese Randbedingungen berücksichtigt.

### 9.13 Der Datenflußgraph und die Kantenbewertung

Oben wurden schon die Abhängigkeiten der Befehle definiert, wobei die allgemeine Abhängigkeit aus der Vereinigung der Datenabhängigkeiten, der Antidatenabhängigkeiten und der Ausgabeabhängigkeiten hervorgeht. Ein weiterer Aspekt, der einen wesentlichen Einfluß auf die Bearbeitungsreihenfolge hat, ist die Zuordnung der Befehle zu den Bauteilen und die damit einhergehende Bearbeitungsreihenfolge der Befehle auf einem Bauteil. Im folgenden wird das Beispiel aus Bild 81 betrachtet.

| | |
|---|---|
| *1) x := a + b* | *((a,b),(x),+)* |
| *2) y := c * d* | *((c,d),(y),*)* |
| *3) z := x + y* | *((x,y),(z),+)* |
| *4) u := a + u* | *((a,u),(u),+)* |
| *5) c := d - 7* | *((d,7),(c),-)* |
| *6) x := d - e* | *((d,e),(x),-)* |

Bild 81: Beispiel

Die im folgenden dargestellten Bedingungen müssen beachtet werden. Die Datenabhängigkeit ist zwischen den Befehlen (1,3) und den Befehlen (2,3) gegeben, außerdem besteht die Ausgabeabhängigkeit (1,6) und die Antidatenabhängigkeit (2,5). Die Reihenfolgen der Bearbeitung seien durch Permutationen festgelegt, die die Bauteilabhängigkeiten repräsentieren. Für eine ALU sei die Reihenfolge der Befehlsbearbeitung durch (2,4,5) und für einen Addierer durch (1,3) gegeben, außerdem wird die Subtraktion (6) durch einen Subtrahierer berechnet. Diese Zusammenhänge können durch einen Graphen dargestellt werden. Bild 82 zeigt den Graphen für das Beispiel, in dem die Knoten die Befehle darstellen und die Kanten die Reihenfolge der Befehlsausführung.

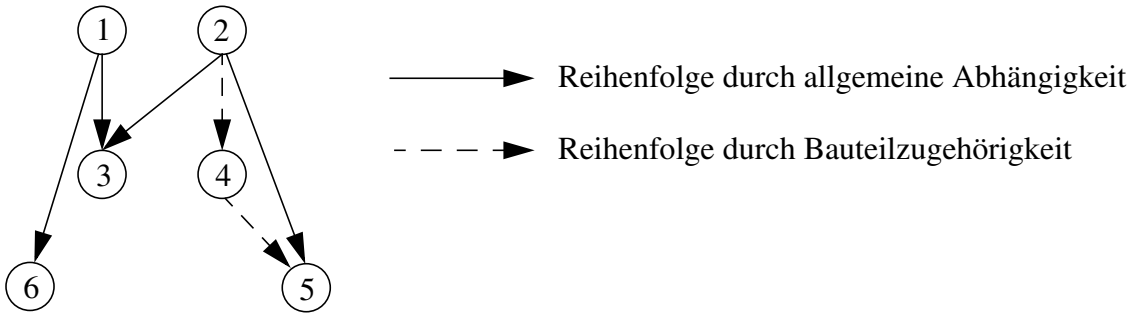



Bild 82: Datenabhängigkeitsgraph mit Bauteilabhängigkeit

Wie wir gesehen haben, ist die Reihenfolge, die durch die Daten-, Antidaten- und Ausgabeabhängigkeit gegeben ist, zwingend. Anders aber die Abhängigkeit, die durch die Bauteilzugehörigkeit gegeben ist. Diese Reihenfolge kann gewählt werden, bzw. die Zuordnung der Befehle zu Bauteilen bietet Wahlmöglichkeiten an. Im Graphen können diese Abhängigkeiten durch fixierte Kanten dargestellt werden. Das heißt also, eine gerichtete Kante wird dann zwischen zwei Knoten gesetzt, wenn zwischen diesen Knoten entweder die Datenabhängigkeit, die Antidatenabhängigkeit oder die Ausgabeabhängigkeit besteht. In manchen Systemen können die Antidaten- und die Ausgabeabhängigkeiten nicht vorkommen, da allen Variablen nur einmal ein Wert zugewiesen wird. Das sogenannte single-assignment-Prinzip wird genutzt, welches intern für jede beschriebene Variable einen neuen Namen vergibt. Um die Anwendungsmöglichkeiten des Systems möglichst allgemein zu halten, wird dieser Fall hier nicht betrachtet. Werden die Ab-

hängigkeiten in einem Graphen dargestellt, so wird die Bewertung der Laufzeit einer Lösung mit Hilfe der Berechnung des längsten Weges durchgeführt.

### 9.13.1 Bewertung der Abhängigkeiten

Die Auswirkungen der verschiedenen Abhängigkeiten auf die Einordnung der Befehle in Bearbeitungsschritte sind unterschiedlich. Das Ziel des Scheduling ist es die insgesamt benötigte Anzahl Bearbeitungs bzw. Kontrollschritte zu minimieren, wobei kein Widerspruch zu den Abhängigkeiten entstehen darf. Im folgenden wird eine Bewertung der Abhängigkeiten dargestellt, die dazu führt, daß die Anzahl der Kontrollschritte potentiell minimal wird. Die in der folgenden Definition eingeführte Funktion wird in den folgenden Abschnitten genauer betrachtet. Sie bildet die grundlegende Bewertungsfunktion für das Schedulingverfahren. Im Prinzip handelt es sich um eine Kantenbewertungsfunktion.

*Definition 9.13-1 Die Bewertung der Abhängigkeiten ist eine Abbildung der Form:* $k: B \times B \rightarrow N$, *wobei B die Menge der Befehle ist.*

### 9.13.2 Bewertung der Datenabhängigkeit

Zwei Befehle, die voneinander datenabhängig sind, z.B. die Befehle 1 und 3 in dem Beispiel, müssen so bearbeitet werden, daß der zweite Befehl erst dann gestartet wird, wenn die Bearbeitung des ersten Befehls abgeschlossen ist. Diese Abhängigkeit muß also mit $k(1, 3) = T_1(1)$ bewertet werden, da durch $T_1$ die Bearbeitungszeit für einen Befehl angegeben wird. Diese Funktion ist abhängig von der Zuordnung der Befehle zu den Bauteilen. Es wird die überladene Funktion $T_1$ unter der Annahme benutzt, daß das Assignment durchgeführt wurde. Die Bild 83 macht den Zusammenhang deutlich:

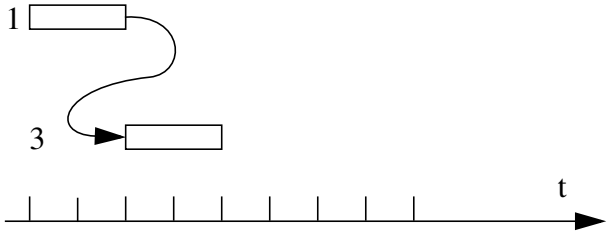



Bild 83: Datenabhängigkeit

Durch den Pfeil wird die Abhängigkeit des dritten Befehls von dem Ergebnis des ersten Befehls verdeutlicht. Der Befehl kann frühestens am Anfang des dritten Taktschrittes starten.

### 9.13.3 Bewertung der Antidatenabhängigkeit

Zwei Befehle sind voneinander antidatenabhängig, wenn der von dem ersten Befehl benutzte Wert durch den zweiten Befehl überschrieben wird. Im angegebenen Beispiel, in Bild 81, gibt es eine Antidatenabhängigkeit zwischen den Befehlen zwei und fünf. Die Bewertung der so gegebenen Antidatenabhängigkeit ist dadurch gekennzeichnet, daß der benutzte Wert erst dann überschrieben werden darf, wenn die Eingänge des Bauteils, welches den ersten Befehl ausführt, die Daten übernommen haben. Im Beispiel wird angenommen, daß die ALU, welche den Befehl zwei ausführt, im ersten Taktschritt die anliegenden Eingangsdaten übernimmt. Die Ausgangsvariable bzw. Eingangsvariable wird in Befehl fünf im letzten Schritt überschrieben. Es wird angenommen, daß durch die flankengesteuerte Taktung in dem System eine korrekte zeitliche Abfolge gewährleistet ist. Die Bild 84 macht den Zusammenhang deutlich.

Der gestrichelte Pfeil in Bild 84 zeigt, wann die Variable überschrieben werden darf. Der durch-

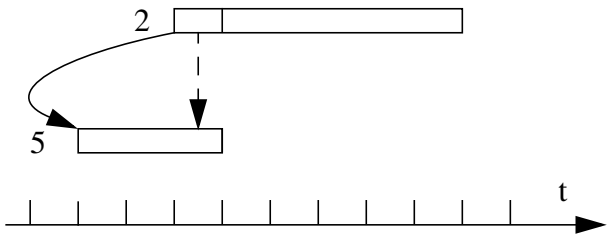



Bild 84: Antidatenabhängigkeit

gehende Pfeil macht deutlich, daß in diesem Fall eine negative Bewertung der Antidatenabhängigkeit auftritt. Die Bewertung läßt sich durch $k(2,5) = T_3(2) - T_1(5)$ berechnen. $T_3$ bezeichnet den Zeitbedarf, der angibt, wie lange die aktuellen Eingabedaten anliegen müssen. Eine negative Bewertung ist unproblematisch, was anhand des in Bild 88 dargestellten ASAP Algorithmus deutlich wird.

### 9.13.4 Bewertung der Ausgabeabhängigkeit

In dem Beispiel aus Bild 81 existiert eine Ausgabeabhängigkeit. Da der Befehl sechs die Variable $x$ überschreibt, die schon von dem Befehl eins überschrieben wird, ist die Bedingung der Ausgabeabhängigkeit für die Befehle eins und sechs gegeben. Außerdem ist die Antidatenabhängigkeit zwischen den Befehlen drei und sechs gegeben. Durch die Antidatenabhängigkeit wird sichergestellt, daß der Befehl sechs nicht zu früh gestartet wird. Die Ausgabeabhängigkeit muß betrachtet werden, damit die nachfolgenden Befehle auf jeden Fall die richtige Eingabe bekommen. Die Ausgabeabhängigkeit bewirkt also, daß der zweite Befehl auch wirklich das Ergebnis des ersten überschreibt. Das heißt, die Befehle müssen mindestens einen Taktschritt versetzt fertig werden. Die Bild 85 macht den Zusammenhang deutlich:

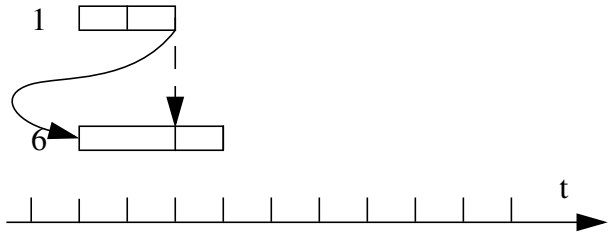



Bild 85: Ausgabeabhängigkeit

Da zwei Befehle nicht gleichzeitig auf eine Variable schreiben dürfen, reicht hier der Abstand von genau einem Taktschritt aus, denn der Befehl sechs wird später fertig als Befehl eins. Der Startzeitpunkt des zweiten Befehls kann wieder vor dem des ersten Befehls liegen. In diesem Fall wird die Ausgabeabhängigkeit mit null bewertet. Die Bewertung läßt sich berechnen durch: $k(1,6) = T_1(1) - T_1(6) + 100$ wobei 100 als Länge eines Taktschrittes gewählt wurde. Es muß also genau der zeitliche Abstand eines Taktschrittes eingehalten werden.

### 9.13.5 Bewertung der Bauteilabhängigkeiten

Die Abhängigkeiten, die durch die Zuordnung der Befehle zu den Bauteilen zustandegekommen sind, müssen nun ebenfalls bewertet werden. Werden zwei Befehle durch das gleiche Bauteil ausgeführt, so existiert eine Bauteilabhängigkeit zwischen diesen Befehlen. Die Abhängigkeit wird dadurch bewertet, daß zwei Befehle nicht gleichzeitig bearbeitet werden können. Dafür gibt es die Zeitangabe $T_2$, die eine Aussage darüber macht, nach welchem Zeitablauf der nächste Befehl auf einem Bauteil starten darf. Bei den meisten Bauteilen gilt $T_1 = T_2$. Anders sieht es bei Bauteilen aus, die eine Pipelineverarbeitung zur Verfügung stellen. Hierdurch ist es möglich, daß der nächste Befehl in der Liste schon starten kann, wenn der

erste noch nicht fertig ist. Beispielsweise werden die Befehle zwei und vier durch die gleiche ALU ausgeführt. Die Bewertung der Abhängigkeit zwischen den Befehlen zwei und vier kann also mit $k(2, 4) = T_2(2)$ festgelegt werden. Die Bild 86 macht den Zusammenhang deutlich.

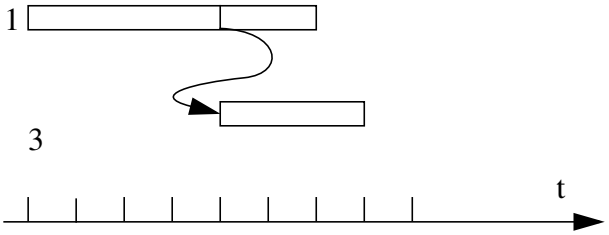



Bild 86: Bauteilabhängigkeit

Aufgrund dieser Darstellung findet eine Bewertung statt, die alle Möglichkeiten der Abhängigkeiten einschließt und auch für die Verwendung von Pipelinebausteinen geeignet ist. Treten mehrere Abhängigkeiten zwischen zwei Befehlen auf, dann muß die maximale Bewertung gewählt werden. Die folgende Formel, die auch im implementierten Experimentalsystem genutzt wird, faßt die Ergebnisse zusammen. Dazu muß zuerst die Länge eines Taktschrittes festgelegt werden.

*Definition 9.13-2 Die Länge eines Taktschrittes wird mit $ts \in N$ bezeichnet, ts gibt den Zeitbedarf eines Taktschrittes an.*

Seien *x* und *y* zwei Befehle, dann gilt für die Bewertungsfunktion *k*, wenn *ts* die Taktschrittlänge ist, folgendes:

$$k(x, y) = max\left\{ k \mid \begin{array}{c} k = T_1(x) \Leftarrow da(x, y) = 1 \\ k = T_2(x) \Leftarrow x <_P y \\ k = T_1(x) - T_1(y) + ts \Leftarrow aa(x, y) = 1 \\ k = T_3(x) - T_1(y) \Leftarrow ada(x, y) = 1 \end{array} \right\}$$

Das heißt, es können auch mehrere Bedingungen erfüllt sein. Dann wird das Maximum des angegebenen Wertes als Bewertung genommen. Die Anforderungen an ein System zur Optimierung des Zeitbedarfs ist somit durch diese Bewertungsfunktion gegeben.

### 9.13.6 Schleifenkonstrukte

Die Verwendung von Schleifen kann ebenfalls im Rahmen dieses Systems betrachtet werden, und zwar sei eine Schleife hier ausgehend von Def. 9.4-2 als Befehlsblock dargestellt. Die Bewertung der Blockabhängigkeiten entspricht der Bewertung der Datenabhängigkeiten. Eine Schleife kann beispielsweise wie in Bild 87 dargestellt werden. Die Befehle 1,2,3 sollen in einer Schleife liegen. Dann werden im ersten Schritt die zusätzlichen Abhängigkeiten eingefügt, wobei die sequentielle Numerierung berücksichtigt wird.

Bewertet werden die zusätzlichen Blockabhängigkeiten genau wie die Datenabhängigkeiten. Somit kann die Bewertungsfunktion, wie folgt, erweitert werden:

Dabei ist *p* die Permutation, welche die Reihenfolge der Befehle angibt, falls sie auf demselben Bauteil ausgeführt werden. *S* stellt den Befehlsblock einer Schleife dar. Der Schedulingalgorithmus errechnet für jedes Paar von Befehlen den durch die Bewertungsfunktion definierten Ab-

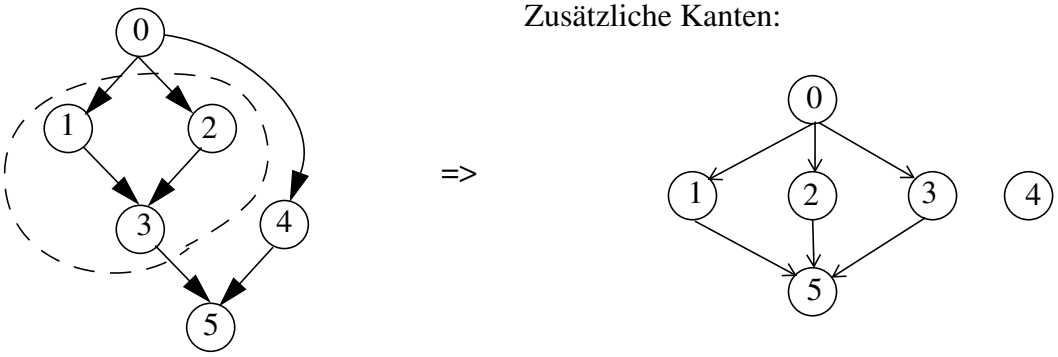



Bild 87: Blockabhängigkeiten als Kanten dargestellt

$$k(x,y) = max\left\{ k \,\middle|\, \begin{array}{c} k = T_1(x) \Leftarrow da(x,y) = 1 \\ k = T_2(x) \Leftarrow x <_P y \\ k = T_1(x) - T_1(y) + ts \Leftarrow aa(x,y) = 1 \\ k = T_3(x) - T_1(y) \Leftarrow ada(x,y) = 1 \\ k = T_1(x) \Leftarrow x <_S y \end{array} \right\}$$

stand und ordnet die Befehle in die Kontrollschritte ein. Ein an den ASAP (as soon as possible) Algorithmus angelehnter erweiterter Algorithmus ist in Bild 88 vorgestellt. Initialisiert wird die Funktion *sch* so, daß für jeden Befehl $b$ aus der Befehlsmenge $B$ $sch(b) = 0$ gilt. Dann werden alle Abhängigkeiten zwischen den Befehlen geprüft und der zeitliche Abstand zwischen den Befehlen berechnet. Die definierte Bewertungsfunktion $k$ wird hier eingesetzt.

Init:

*for all* $b \in B$
  *sch(b) = 0*

*ASAP:*
*repeat*
  *for all* $b_i \in B$
  *for all* $b_j \in B$

$$if\,(b_i \ll b_j) \vee (b_i <_P b_j) \vee (b_i <_S b_j)$$

$$sch(b_j) = max\left(sch(b_j), sch(b_i) + \left\lceil \frac{k(b_i, b_j)}{ts} \right\rceil\right)$$

  *end for*
  *end for*
*until (keine Veränderung von* $sch(b_i)$*)*

Bild 88: Modifizierter ASAP Algorithmus

Andere Schedulingalgorithmen berechnen eine Lösung mit dem ASAP Algorithmus und eine weitere Lösung mit dem ALAP (as late as possible) Algorithmus. Durch diese Berechnungen ergeben sich für jeden Befehl Freiheitsgrade, die genutzt werden, um den Hardwarebedarf zu reduzieren. In unserem Fall ist das nicht nötig, da der Hardwarebedarf schon durch die vorhergehenden Allocations- und Assignmentfunktionen festgelegt wurde. Der Bedarf an Registern

wird andrerseits erst nach dem Scheduling berechnet, so daß hier noch Optimierungsmöglichkeiten bestehen. Der gesamte Ablauf ist darauf angelegt, daß die Allocations- und Assignmentschritte zu guten, bzw. optimalen Ergebnissen führen. Da dieses Problem aber NP-vollständig ist, sind optimale Lösungen nur mit hohem Aufwand auffindbar. Um eine Allocation und ein Assignment bewerten zu können, werden der oben schon definierte Ressourcenbedarf und andrerseits der Zeitbedarf benötigt, der aus dem Scheduling hervorgeht. Die Bewertung einer Schaltung ist der Zeitbedarf in Taktschritten und kann folgendermaßen berechnet werden:

*Definition 9.13-3 Die Bewertungsfunktion für den Zeitbedarf einer Schaltung geht aus dem Scheduling hervor, welches seinerseits von der Befehlsmenge B und dem Assignment A abhängt. Sie ist eine Funktion der Form* $Bt: B^n \times L \rightarrow N$ *und ist definiert als*

$$Bt(PR, A) = ts \cdot max\left\{ sch(b_i) + \left\lceil \frac{T_1(b_i)}{ts} \right\rceil \middle| b_i \text{ ist der i-te Befehl aus } PR \right\}.$$

Das heißt also, der Zeitbedarf errechnet sich aus der Einordnung der Befehle in die Taktschritte, wobei der letzte benutzte Taktschritt für die Berechnung interessant ist. Werden die Relationen als Graph mit gewichteten Kanten dargestellt, so ist die Berechnung des Zeitbedarfs äquivalent zur Berechnung des längsten Weges in einem gerichteten, bewerteten Graphen unter der Voraussetzung, daß keine Schleifen existieren. Im Rahmen dieser Arbeit ist der Begriff Zeitbedarf daher immer so zu verstehen, daß angenommen wird, daß vorkommende Schleifen nur einmal durchlaufen werden. Der kritische Pfad wird der Vollständigkeit halber an dieser Stelle definiert.

*Definition 9.13-4 Eine Folge von Befehlen* $(b_0, \ldots, b_n)$ *mit* $b_i \in B$ *ist ein kritischer Pfad bzgl. des Programms PR und des Assignments A genau dann, wenn gilt:*

$$\sum_{i=0}^{n-1} k(b_i, b_{i+1}) + \left\lceil \frac{T_1(b_n)}{ts} \right\rceil = Bt(PR, A).$$

Es kann mehrere kritische Pfade geben. Wird im folgenden von den Befehlen auf dem kritischen Pfad gesprochen, so sind damit alle Befehle gemeint, die auf irgendeinem der kritischen Pfade liegen.

## 9.14 Zusammenfassung

In diesem Kapitel wurden Befehle und Bauteile definiert sowie die Funktionen Allocation, Assignment und Scheduling. Außerdem wurden Schleifen auf die Definition eines Befehlsblockes zurückgeführt. Die Allocationsfunktion stellt eine Menge von Bauteilen für die weiteren Schritte zur Verfügung, wobei die zur Verfügung gestellte Menge gültig sein muß bzgl. eines Programms. Das Assignment ordnet jedem Befehl einen Bauteil zu, wobei das Assignment ebenfalls gültig sein muß. Durch das Assignment wird jedem Befehl die Ausführungszeit zugeordnet. Anhand dieser Ausführungszeiten kann dann ein Scheduling durchgeführt werden, wobei hier aber die Schleifen-Konstrukte und das Assignment beachtet werden müssen. Der Vorteil dieser Vorgehensweise ist, daß die unterschiedlichen Verzögerungszeiten der Bauteile voll berücksichtigt werden. Es wird also nicht davon ausgegangen, daß jeder Befehl in einem Kontrollschritt ausführbar ist, sondern es werden im wesentlichen auch Befehle zugelassen, die eine höhere Ausführungszeit haben. Die Ausführungszeit ist aber im wesentlichen von dem Al-

gorithmus abhängig, der zur Ausführung einer Funktion implementiert ist. Werden verschiedenen Bauteile mit unterschiedlicher Implementierung der gleichen Funktion zur Verfügung gestellt, was zu unterschiedlichem Ressourcen- und Zeitbedarf führt, so kann dies berücksichtigt werden. Es ist andrerseits kein Problem, die Technologieunabhängigkeit zu wahren, indem die Verzögerungszeiten durch Gatterlaufzeiten - also die Anzahl der durchlaufenen Gatterebenen - dargestellt werden. Werden die Laufzeiten für ein Gatter einheitlich auf eins gesetzt, so kann die Synthese unabhängig von der benutzten Technologie durchgeführt werden. Die Definitionen sind andrerseits so allgemeingültig, daß die Laufzeiten der Bauteile mit dem Wert eins festgelegt werden können, wodurch andere bekannte Methoden mit abgedeckt werden. Der Nachteil ist, daß es viel mehr Lösungsmöglichkeiten gibt, weshalb einige Algorithmen nicht direkt übertragen werden können. Im Rahmen dieser Arbeit wird die Eignung genetischer Algorithmen für den Einsatz in der High-Level Synthese betrachtet. Das nächste Kapitel wird im wesentlichen diesen Teil behandeln, wobei hier gesagt werden kann, daß die theoretischen Betrachtungen der Synthese allgemeingültig sind, so daß auch andere Algorithmen in diesen theoretischen Rahmen eingefügt werden können.

# 10 Anwendung genetischer Algorithmen für die Synthese

Wie schon gezeigt, gibt es viele Möglichkeiten, eine Menge von Bauteilen für eine Menge von zu synthetisierenden Befehlen zur Verfügung zu stellen. Ist einmal eine Menge Bauteile festgelegt, so gibt es wieder viele Möglichkeiten, die Befehle den Bauteilen zuzuweisen. Der Ansatz für eine optimale Menge an Bauteilen und Zuordnungen ist, den Allocations- und Assignmentprozes durch genetische Algorithmen ausführen zu lassen. Um die Durchführung dieser Prozesse durch genetische Algorithmen zu ermöglichen, müssen Codierungsfunktionen gefunden werden, die am besten eine bijektive Darstellung der Allocation und des Assignments auf den Code darstellen. Auch in [115] und [97] wurde mit Hilfe von genetischen Algorithmen versucht, Syntheseergebnisse zu optimieren. Zuerst wird die Codierung der Allocation betrachtet.

## 10.1 Allocation mit einem genetischen Algorithmus

Die Allocation von Bauteilen realisiert die Auswahl von Bauteilen, die in den weiteren Syntheseschritten zur Verfügung stehen sollen. Dazu wird angenommen, daß in einer Modul- oder Bauteilbibliothek nur die Typen der Bauteile spezifiziert werden. Durch den Allocationsschritt werden dann die Instanzen der Bauteile gebildet, mit denen gearbeitet werden kann.

Betrachtet man den Schritt der Allocation im Zusammenhang mit den anderen Syntheseschritten, so fällt auf, daß dieser Schritt zwar ziemlich früh ausgeführt werden muß, aber die Bewertung erst zum Schluß - also im Rahmen der Gesamtbewertung - geschehen kann. Ein genetischer Algorithmus für die Allocation sieht vom Konzept her so aus, wie es in Bild 89 dargestellt ist.

In Abhängigkeit von dem zu synthetisierenden Modell, welches in einer Hardwarebeschreibungssprache - zum Beispiel VHDL - zur Verfügung steht, werden aus der Bauteilbibliothek Bauteilmengen allociert. Aufgrund der Bewertung der allocierten Mengen anhand einer Bewertungsfunktion, die spezifiziert werden muß, werden einige Mengen selektiert und den genetischen Operatoren Mutation und Crossover zugeführt. Aus den selektierten Allocationen werden dann so viele neue Bauteilmengen erzeugt, daß die gegebene Anzahl einer Generation wieder erreicht ist. Der Prozeß startet erneut mit der Bewertung der erzeugten Allocationen. Ist die Abbruchbedingung erfüllt, so wird die beste gefundene Allocation als Ergebnis geliefert. Die eigentliche Problematik ist hierbei die Bewertung der allocierten Bauteilmengen. Es gibt zwei Möglichkeiten, eine Bewertung durchzuführen: einerseits indem die Schritte Synthese, Assignment und Scheduling durchgeführt werden, was wieder zu mehreren Optimierungsschritten

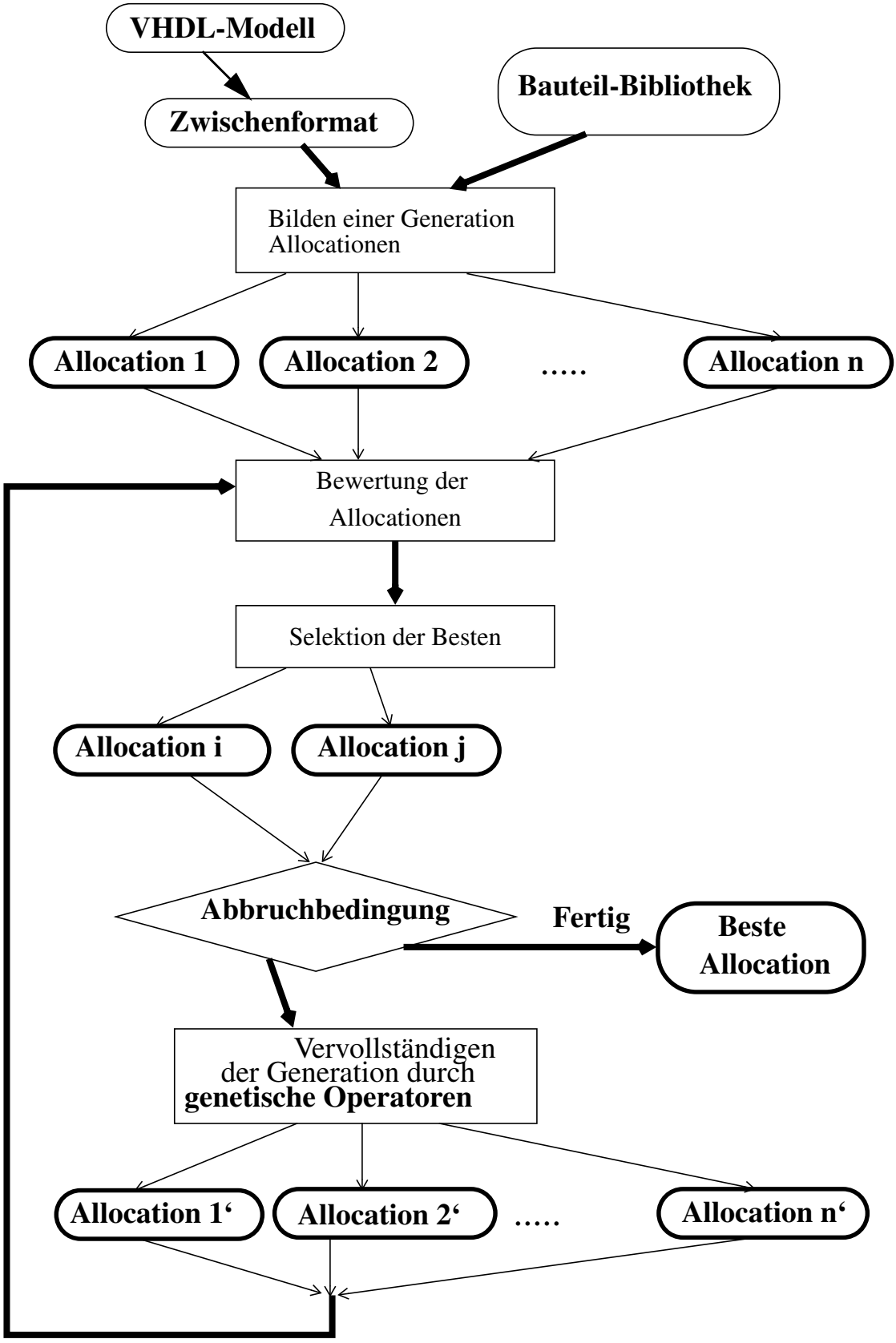



Bild 89: Der Allocationsprozeß

führt, da jede Bewertung wieder einen genetischen Algorithmus nach sich zieht. Die allocierte Menge wird anhand der Qualität der Synthese berechnet. Andrerseits könnte eine Vorhersage anhand einer diskreten Funktion getroffen werden, die eine gegebene Menge an Bauteilen bzgl. ihrer Güte bewertet. Diese Vorhersagefunktion ist entsprechend zu spezifizieren. Im folgenden sollen die Codierungs-, Mutations- und Crossoverfunktion, die angewandt werden, dargestellt werden. Außerdem wird genauer auf die Bewertung der Allocationen eingegangen.

### 10.2 Codierung der Allocation

Die Menge der allocierten Bauteile kann auf mehrere Arten codiert werden. An dieser Stelle soll nur eine der untersuchten Möglichkeiten vorgestellt werden. Wie bereits erwähnt, stellt die Codierung einer Lösung eines Problems eine Abbildung auf einen String dar, wobei eine sehr wichtige Entscheidung die Wahl des Alphabets betrifft. Jedes in einer Bibliothek vorkommende Bauteil wird im Code durch eine Stringposition repräsentiert. Durch die Wahl des Alphabets als der Menge der natürlichen Zahlen mit Null kann für jedes Bauteil die Anzahl der Instanzen als natürliche Zahl an der entsprechenden Position im String angegeben werden. Somit kann jede beliebige Auswahl an Bauteilen bzgl. einer gegebenen Bibliothek codiert werden. Eine Codierung für die Allocation kann also folgendermaßen definiert werden:

*Definition 10.2-1 Eine Codierung für die Allocationen bzgl. eines Programms PR und einer Bibliothek BLIB ist eine Abbildung* $C_{al}: AL(PR, BLIB) \rightarrow \Sigma^{|BLIB|}$.

Jedem Element aus der Bibliothek wird durch diese Definition eine Stelle im String zugeordnet. Es sei also im folgenden beispielsweise die Bauteilbibliothek $BLIB = \{ALU1, ALU2, ALU3, ALU4, ALU5\}$ gegeben, bei der jedes Element der Bauteilbibliothek nur wenige Funktionen zur Verfügung zu stellen braucht. Zum Beispiel können *ALU1* ein Addierer und *ALU3* ein Multiplizierer sein. Es sind an dieser Stelle im wesentlichen die ALUs oder Bauteile definiert, die von einer realen Bibliothek zur Verfügung gestellt werden. Das Alphabet für die Codierung sei gegeben durch $\Sigma = N_0$ die Menge der natürlichen Zahlen und Null. Dann kann jede allocierte Bauteilmenge durch ein Element der Menge $\Sigma^5$ dargestellt werden. In Bild 90 sind als Beispiel die beiden Bauteilmengen $L_1, L_2 \in AL(PR, BLIB)$ dargestellt. In der Bauteilmenge $L_1$ sind also 3 Bauteile vom Typ *ALU1* enthalten, zwei vom Typ *ALU2* usw.. Durch die Definition ist für die Menge der Allocationen *AL* sichergestellt, daß sie gültig gemäß Def. 9.7-7 sind.

L1:

| ALU1 | ALU2 | ALU3 | ALU4 | ALU5 |
|---|---|---|---|---|
| 3 | 2 | 5 | 1 | 0 |

=> 32510

L2:

| ALU1 | ALU2 | ALU3 | ALU4 | ALU5 |
|---|---|---|---|---|
| 2 | 4 | 1 | 2 | 3 |

=> 24123

Bild 90: Codierung der Allocation

Durch die Wahl der Codierung ist umgekehrt die Möglichkeit gegeben, daß es Strings gibt, die Bauteilmengen darstellen, die nicht gültig sind. Die Gültigkeit codierter Bauteilmengen muß also explizit gewährleistet werden.

### 10.3 Die Dekodierung

Die Umkehrfunktion der Codierung, also die Dekodierung eines Codes, erzeugt aus einem Code wieder eine entsprechende Bauteilmenge.

*Definition 10.3-1 Die Dekodierungsfunktion der Allocation ist eine Abbildung* $D_{al}:\Sigma^{|BLIB|} \rightarrow AL(PR, BLIB)$ *es handelt sich dabei um die Umkehrfunktion von* $C_{al}$.

Die Dekodierungsfunktion erzeugt dem Code gemäß eine Menge Bauteile. Dabei wird impliziert, daß die Bauteilmenge gültig ist. Da die Reihenfolge der Bauteile durch die Repräsentation in der Bibliothek vorgegeben und diese Reihenfolge nicht veränderbar ist, kann eine eindeutige Zuordnung der Codes zu den allocierten Bauteilen gefunden werden.

### 10.4 Mutation

Der Mutationsoperator bewirkt eine kleine Veränderung einer gegebenen Lösung. Im Fall des Allocationsproblems gibt es zwei Möglichkeiten der geringen Veränderung einer Menge von Bauteilen. Einmal wird die Anzahl der vorhandenen Bauteile verändert, das heißt also, die Anzahl der Instanzen gleichen Typs wird erhöht oder erniedrigt. Andrerseits wird ein vorhandenes Bauteil durch ein ähnliches ersetzt. Zwei Veränderungen des ersten Typs können somit eine Veränderung des zweiten Typs ersetzen. Mit der gegebenen Codierung bietet sich der Mutationsoperator ersten Typs an, der auch verwendet wird. Auf Codeebene sieht eine Mutation dann so aus, daß eine Zahl, die im String vorkommt, verändert wird.

*Definition 10.4-1 Ein Mutationsoperator für die Allocation* $m_{al}:\Sigma^{|BLIB|} \rightarrow \Sigma^{|BLIB|}$ *ist eine Abbildung von der Menge der Codes der Allocationen auf die Menge der Codes der Allocationen.*

Soll im obigen Beispiel die Bauteilmenge $L_1$ mutiert werden, so bedeutet dies eine geringe Veränderung der Menge. Zum Beispiel kann sich die Anzahl der Bauteile des Typs *ALU3* verändern. In der Bild 91 wird die Veränderung durch die kursiv dargestellten Elemente des Codes visualisiert. Man kann annehmen, daß eine Mutation auch bei der Bewertung keine allzu großen Unterschiede bewirkt.

$L_1$:

| ALU1 | ALU2 | ALU3 | ALU4 | ALU5 |
|---|---|---|---|---|
| 3 | 2 | *5* | 1 | 0 |

=> 32*5*10

$m(L_1)$:

| ALU1 | ALU2 | ALU3 | ALU4 | ALU5 |
|---|---|---|---|---|
| 3 | 2 | *4* | 1 | 0 |

=> 32*4*10

Bild 91: Mutation einer Allocation

Da es Codes für ungültige Bauteilmengen gibt und diese durch eine Mutation erzeugt werden können, ist die Mutationsfunktion nicht gültig gemäß Def. 8.5-2. Wird zum Beispiel die Instanzenanzahl eines Bauteiltyps durch eine Mutation auf null Instanzen reduziert, und gibt es kein

anderes Bauteil, das eine bestimmte Operation, die gerade benötigt wird, zur Verfügung stellt, dann ist die Bauteilmenge ungültig. Die Gültigkeit der Bauteilmengen muß explizit gewährleistet werden.

### 10.5 Crossover

Der Crossoveroperator soll aus zwei gegebenen Lösungen eine neue generieren. Das heißt, aus zwei Mengen an Bauteilen soll eine erzeugt werden. Hier gibt es wieder viele Möglichkeiten. Die Nutzung herkömmlicher Mengenoperatoren wie Vereinigungs- und Schnittmengenbildung bietet sich nicht an, da die Kardinalität der resultierenden Menge stark verändert wird. Des weiteren soll der Crossoveroperator einfach und auf der Ebene des Codes durchführbar sein. Crossover heißt auf der Ebene des Codes, daß zwei Strings an einem Punkt aufgebrochen werden und die Teilstücke sozusagen ‚über Kreuz' wieder miteinander verbunden werden.

*Definition 10.5-1 Der Crossoveroperator für die Allocation ist eine Abbildung* $cr_{al}: \Sigma^{|BLIB|} \times \Sigma^{|BLIB|} \rightarrow \Sigma^{|BLIB|}$ *von zwei Codemengen von Allocation auf eine Codemenge.*

Sollen beispielsweise die beiden Bauteilmengen $L_1$ und $L_2$ mit einem Crossoveroperator vermischt und so eine neue Bauteilmenge $L_3$ gebildet werden, so geschieht dies über die Codierung der Mengen, wie es in Bild 92 dargestellt ist.

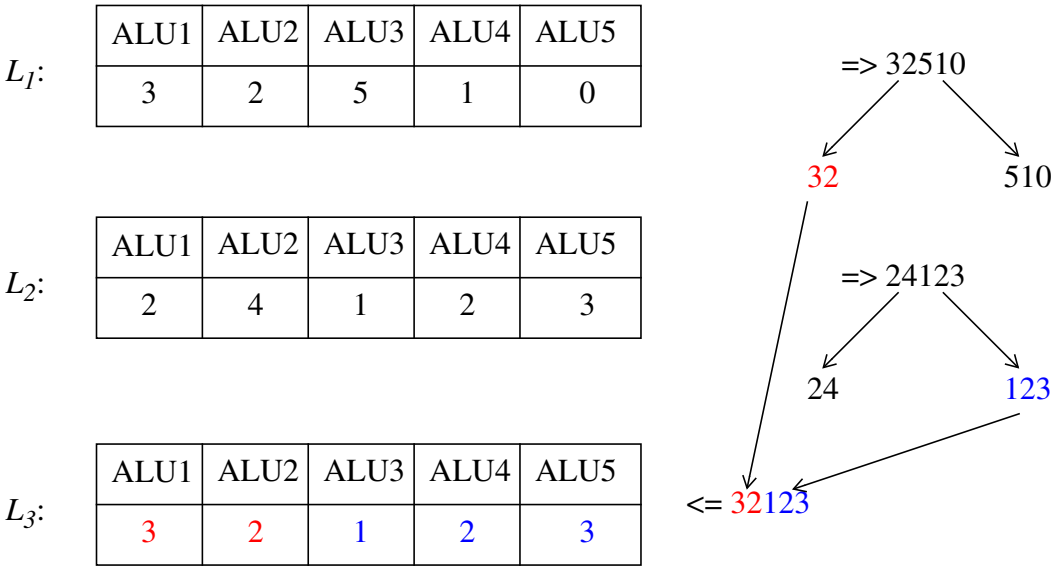



Bild 92: Crossover von zwei Allocationen

Eine schlechte Kombination zweier Bauteilmengen kann auch beim Crossoveroperator dazu führen, daß es zu einer ungültigen Allocation kommt. Dies ist der Fall, wenn eine bestimmte Operation in beiden zu kombinierenden Bauteilmengen nur jeweils durch ein Bauteil ausgeführt werden kann. Beide Bauteile werden durch ein Crossover nicht in die neue Bauteilmenge übernommen.

### 10.6 Bewertung

In jedem Schritt eines genetischen Algorithmus wird eine Generation von Lösungen erzeugt, bewertet, sortiert und mit den genetischen Operatoren weiter verarbeitet. Die Idee der geneti-

schen Algorithmen, nur die guten Lösungen weiterzuverwenden, wird dadurch realisiert, daß eine Bewertung jeder Lösung vorgenommen wird und die Lösungen nach ihrer Güte sortiert werden. Die besten Lösungen werden dann ausgewählt, um auf ihnen die genetischen Operatoren durchzuführen und somit eine neue Menge an Lösungen zu generieren. Der aufwendigste Bestandteil eines genetischen Algorithmus ist die Bewertungsfunktion. In dem dargestellten Fall ist eine objektive Bewertung der allocierten Bauteilmengen nur dann möglich, wenn die nächsten Schritte der Synthese ebenfalls durchgeführt werden, also das Assignment und das Scheduling. Somit werden zur Bewertung einer Allocation alle im nachfolgenden beschriebenen Syntheseschritte durchgeführt. Diese Lösung des Bewertungsproblems ist natürlich aus Laufzeitgründen nicht befriedigend, weshalb nach einer diskreten Bewertungsfunktion gesucht werden muß, die von der Bauteilmenge und der Befehlsmenge abhängig ist. Im folgenden werden einige mögliche Bewertungsfunktionen dargestellt, die aber alle der Einschränkung unterliegen, daß sie das Ergebnis, welches mit expliziter Ausführung der Folgeschritte erreicht wird, nur annähern.

### 10.6.1 Bewertungsfunktion

Wie schon in Def. 8.3-1 und Def. 8.3-2 dargestellt, kann sich die Bewertung einer im Rahmen eines genetischen Algorithmus gefundenen Lösung auf verschiedene Kriterien beziehen, was zu einer Mehrzieloptimierung führt. Die Kriterien, die für eine möglichst sinnvolle Bewertung der allocierten Bauteilmenge bzgl. einer Befehlsmenge zu guten Ergebnissen führen, können nur subjektiv sein. Denn die endgültige Bewertung läßt sich erst im Zusammenhang mit dem Assignment und dem Scheduling durchführen. Die in Bild 93 angegebenen Bewertungsfunktionen lassen sich unterscheiden. Die Funktionen $g_{al,1}$ ... $g_{al,6}$ gehen mit einem Gewichtsvektor $w_{al}$ in die Gesamtbewertung $ges_{al}$ der Allocation ein. Im folgenden wird auf jede Bewertungsfunktion im einzelnen eingegangen.

$g_{al,1}$ : Maximaler sequentieller Zeitbedarf

$g_{al,2}$ : Minimaler sequentieller Zeitbedarf

$g_{al,3}$ : Anzahl allocierter Bauteile

$g_{al,4}$ : Ressourcenbedarf

$g_{al,5}$ : Balancierte Auslastung

$g_{al,6}$ : Erwarteter gewichteter Zeitbedarf

Bild 93: Bewertungsfunktionen der Allocation

### 10.6.2 Maximaler sequentieller Zeitbedarf

Da der endgültige Zeitbedarf erst nach dem Scheduling feststeht, wird an dieser Stelle angenommen, daß alle Befehle sequentiell ausgeführt werden. Für jeden Befehl wird dann der maximale Zeitbedarf, der sich anhand der allocierten Bauteile ergibt, angenommen. Für die Bauteilmenge *BA* kann die Bewertungsfunktion $g_{al,1}$ folgendermaßen definiert werden:

*Definition 10.6-1 Der maximale sequentielle Zeitbedarf zur Ausführung einer Befehlsmenge B auf einer Bauteilmenge BA wird durch die Funktion*

$$g_{al,1} = \sum_{\forall b \in B} max\{T_1(u, f(b)) | u \in BA, f(b) \in f(u)\}$$ *berechnet.*

Für jeden Befehl wird also das Bauteil gesucht, welches den maximalen Zeitbedarf hat, um den Befehl auszuführen. Die so berechneten Verzögerungszeiten werden addiert. Hierdurch kann der maximale Zeitbedarf einer Schaltung bzgl. einer Bauteilmenge berechnet werden. Der Sinn dieser Bewertungsfunktion ist also, den worst-case Fall zu verbessern.

### 10.6.3 Minimaler sequentieller Zeitbedarf

Bei der Berechnung des minimalen sequentiellen Zeitbedarfs einer Schaltung bzgl. einer allocierten Bauteilmenge wird wieder angenommen, daß die Befehle sequentiell ausgeführt werden, wobei aber für den Zeitbedarf der Ausführung eines Befehls angenommen wird, daß der Befehl dem schnellsten Bauteil zugewiesen wird. Die Funktion $g_{al,2}$ wird dann folgendermaßen definiert:

*Definition 10.6-2 Der minimale sequentielle Zeitbedarf zur Ausführung einer Befehlsmenge B auf einer Bauteilmenge BA wird durch die Funktion*

$$g_{al,2} = \sum_{\forall b \in B} min\{T_1(u, f(b)) | u \in BA, f(b) \in f(u)\}$$ *berechnet.*

Für jeden Befehl wird das Bauteil gesucht, welches ihn am schnellsten ausführt. Die so erhaltenen Ausführungszeiten werden dann addiert.

### 10.6.4 Anzahl allocierter Bauteile

Die dritte Bewertungsfunktion für eine Menge allocierter Bauteile ist deren Anzahl. Die Bewertungsfunktion kann also definiert werden als:

*Definition 10.6-3 Die Anzahl allocierter Bauteile einer Bauteilmenge ist gegeben durch*

$$g_{al,3} = |BA|.$$

### 10.6.5 Ressourcenbedarf

Neben der Anzahl der allocierten Bauteile kann als weitere Bewertungsfunktion der Gesamtbedarf an Ressourcen berechnet werden. Durch die in Def. 9.7-5 angegebene Funktion kann für jedes allocierte Bauteil der Ressourcenbedarf berechnet werden. Dieser wird für die gesamte Menge addiert. Die Funktion $g_{al,4}$ ist dann folgendermaßen definiert:

*Definition 10.6-4 Der Ressourcenbedarf einer allocierten Bauteilmenge wird berechnet als*

$$g_{al,4} = \sum_{u \in BA} r(u).$$

### 10.6.6 Balancierte Auslastung der Bauteile

Eine Bedingung, die bei der Allocation angegeben werden kann, ist die balancierte Auslastung der Bauteile. Ähnlich wie bei dem schon dargestellten FDS Schedulingverfahren wird versucht, eine Menge von Bauteilen zu finden, bei der die zu erwartende Auslastung jedes Bauteils balanciert ist. Hierzu wird für jeden Befehl die Anzahl der Bauteile berechnet, die diesen Befehl ausführen können. Jedem Bauteil wird dann der Wahrscheinlichkeitswert, daß der Befehl auf diesem Bauteil ausgeführt wird, zugewiesen. Die Vorgehensweise der Berechnung der Bewertungsfunktion $g_{al,5}$ soll im folgenden dargestellt werden. Die Befehle einer Befehlsmenge *B* werden auf die horizontale Achse einer Matrix aufgetragen, die Bauteile der allocierten Bauteil-

menge *BA* werden auf die vertikale Achse aufgetragen. Die Elemente der Matrix werden mit $w_{i,j}$ bezeichnet. In Bild 94 ist die so entstandene Wahrscheinlichkeitsmatrix dargestellt für $m = 3$ Bauteile und $n = 3$ Befehle.

| | $ba_1$ | $ba_2$ | $ba_3$ |
|---|---|---|---|
| $b_1$ | $w_{1,1}$ | $w_{1,2}$ | $w_{1,3}$ |
| $b_2$ | $w_{2,1}$ | $w_{2,2}$ | $w_{2,3}$ |
| $b_3$ | $w_{3,1}$ | $w_{3,2}$ | $w_{3,3}$ |

Bild 94: Wahrscheinlichkeitsmatrix

*Definition 10.6-5 Die Wahrscheinlichkeitsmatrix für die wahrscheinliche Verteilung der Befehle der Befehlsmenge B auf die Bauteile der Bauteilmenge BA wird definiert*

$$durch\ (b_i) \in f(ba_j) \Rightarrow w_{i,j} = \frac{1}{\left|\{u|(u \in BA)(f(b_i) \in f(u))\}\right|}$$

$$und\ (b_i) \notin f(ba_j) \Rightarrow w_{i,j} = 0.$$

*Satz 10.6-6 Für die Wahrscheinlichkeitsmatrix gilt* $\sum_{j=1}^{m} w_{i,j} = 1$ *für ein festes i.*

Für jedes Bauteil wird nun der Belegungswert $bel_{ba}$ berechnet.

*Definition 10.6-7 Der Belegungswert* $bel_{ba,j}$ *des Bauteils* $ba_j$ *wird definiert als*

$$bel_{ba,j} = \sum_{i=1}^{n} w_{ij}.$$

Der Belegungswert eines Bauteils ist der Erwartungswert, wieviele Befehle auf diesem Bauteil voraussichtlich ausgeführt werden. Somit wird eine balancierte Auslastung der Bauteile dann erreicht, wenn die quadratische Abweichung vom Mittelwert oder Streuung der Belegungswerte minimiert wird.

*Definition 10.6-8 Die Bewertungsfunktion für eine balancierte Auslastung der Bauteile* $g_{al,5}$

$$wird\ als\ g_{al,5} = \sqrt{\sum_{j=1}^{m}\left(bel_{ba,j} - \frac{\sum_{j=1}^{m} bel_{ba,j}}{m}\right)^2} = \sqrt{\sum_{j=1}^{m}\left(bel_{ba,j} - \frac{n}{m}\right)^2}\ definiert.$$

Durch eine balancierte Auslastung der Bauteile soll erreicht werden, daß nicht überflüssige Bauteile allociert werden. Je mehr Bauteile für den Assignmentprozeß zur Verfügung gestellt werden, um so größer ist dort die Auswahlmöglichkeit, und die Rechenzeit, um ein gutes Assignment zu finden, erhöht sich entsprechend. Eine Erweiterung der balancierten Auslastung der Bauteile ergibt sich aus der Berechnung der erwarteten Laufzeit für jeden Befehl.

### 10.6.7 Erwartete Ausführungszeit der Befehle

Der im folgenden dargestellte Ansatz, der die erwartete Ausführungszeit der Befehle bzgl. einer allocierten Menge an Bauteilen abschätzt, berechnet den Durchschnitt der Laufzeiten der Befehle, wobei aber die Laufzeiten mit dem Belegungswert des entsprechenden Bauteils gewichtet sind. Ein Bauteil, welches einen hohen Belegungswert hat, kann zwar einerseits den Befehl als solchen schnell berechnen, andrerseits muß aber eine lange Wartezeit in Kauf genommen werden. Für jeden Befehl kann die gewichtete, zu erwartende Ausführungszeit wie folgt berechnet werden:

*Definition 10.6-9 Die zu erwartende Ausführungsszeit für einen Befehl $b_i \in B$ bzgl. einer Bauteilmenge BA kann berechnet werden durch die folgende Gleichung*

$$erw_{b,i} = \frac{\sum_{j|(ba_j \in BA)(f(b_i) \in f(ba_j))} T_1(ba_j, f(b_i)) \cdot bel_{ba,}}{\left|\{k|f(b_i) \in f(ba_k)\}\right|}.$$

Hier wird also die durchschnittliche Ausführungsszeit eines Befehls anhand der zur Verfügung stehenden Bauteile berechnet, wobei die Ausführungszeiten anhand der Belegung gewichtet werden. Die Bewertungsfunktion, welche für den genetischen Algorithmus benutzt wird, ist die Summe der gewichteten, erwarteten Ausführungszeiten aller Befehle.

*Definition 10.6-10 Die Bewertungsfunktion für die gewichtete, zu erwartende Ausführungszeit der Befehlsmenge B mit m Befehlen ist gegeben durch* $g_{al,6} = \sum_{i=1}^{m} terw_{b,i}$.

Selbstverständlich können auch die Zeitangaben $T_2$ und $T_3$ für eine entsprechende Bewertungsfunktion benutzt werden. Die verschiedenen Bewertungsfunktionen gehen mit entsprechender Gewichtung $w_{al}$ in die Gesamtbewertungsfunktion $ges_{al}$ ein.

*Definition 10.6-11 Die Gesamtbewertungsfunktion der Allocation ist definiert als*

$ges_{al} = \sum_{i=1}^{6} w_{al,i} \cdot g_{al,i}$, *wobei der Gewichtsvektor* $w_{al} = (w_{al,1}, \ldots, w_{al,6})$ *gewählt werden muß.*

Nachdem die Bewertung der Lösungen abgeschlossen ist, werden die ‚guten' Lösungen selektiert und der weiteren Verarbeitung zugeführt. Die Bewertungsfunktionen sind dabei so definiert, daß sie einen niedrigen Wert berechnen, wenn Güte der Lösung hoch ist.

## 10.7 Selektionsfunktion für Allocation

Die Selektionsfunktion wählt aus einer Generation Lösungen, also hier Bauteilmengen, eine Teilmenge aus, die dann den Funktionen Mutation und Crossover zur Verfügung gestellt werden, so daß soviele neue Bauteilmengen erzeugt werden, daß wieder eine vollständige Generation entsteht. Die Selektion einer Teilmenge der Lösungen findet im dargestellten Fall dadurch statt, daß die bewerteten Bauteilmengen anhand ihrer Güte bzw. Bewertung sortiert werden. Die schlechteren Bauteilmengen werden dann verworfen, wobei die besseren durch den Crossoveroperator so vermehrt werden, daß die ursprüngliche Anzahl der Bauteilmengen erhalten bleibt. Der Mutationsoperator wird dann auf einige der ausgewählten Bauteilmengen angewandt. In

der Bild 95 ist die Einordnung der Selektion in den Ablauf dargestellt, der auch in der konkreten Ausführung benutzt wird.

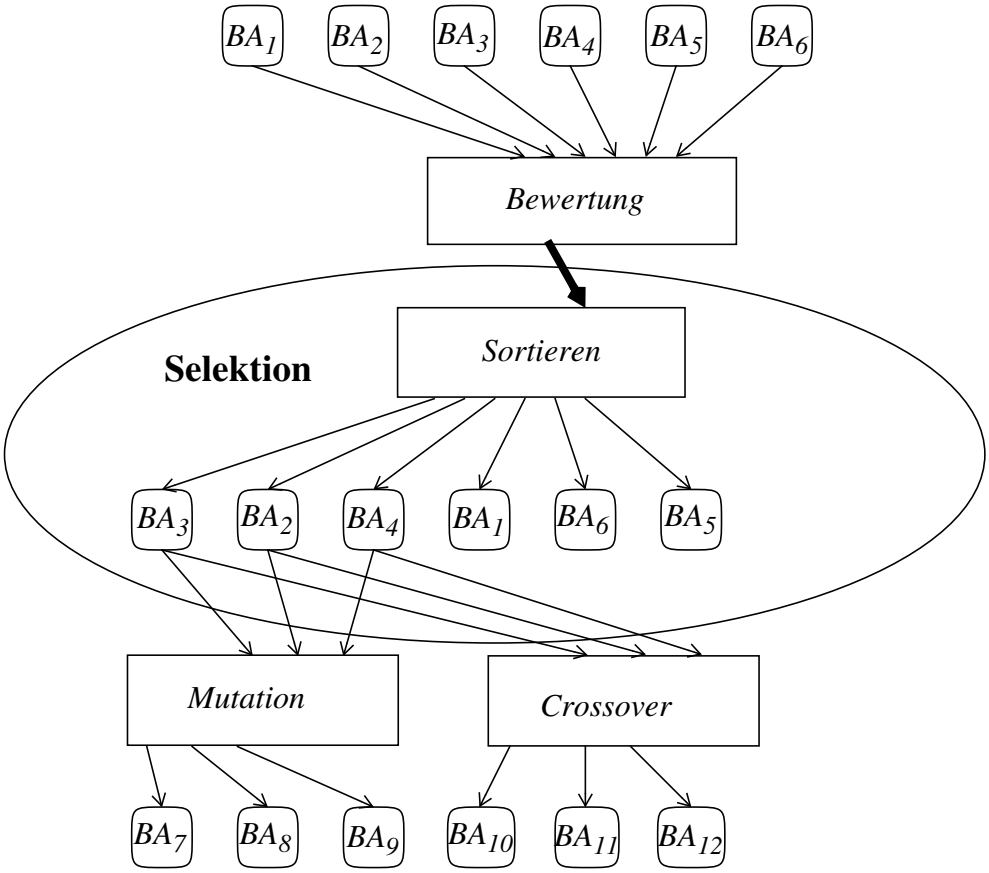



Bild 95: Die Einordnung der Selektionsfunktion

### 10.8 Abbruchbedingung

Wie schon oben dargelegt, muß eine Abbruchbedingung definiert werden. Der genetische Algorithmus erzeugt eine gewisse Anzahl von Generationen mit Bauteilmengen. Die Abbruchbedingung besagt, wann der Algorithmus gestoppt und das Ergebnis weiter verarbeitet werden kann. Eine Abbruchbedingung kann dadurch gegeben sein, daß eine feste Anzahl an Generationen vorgegeben und das erreichte Ergebnis dann benutzt wird. Diese Abbruchbedingung wurde meistens in der implementierten Version benutzt. Andrerseits kann der genetische Algorithmus dann abgebrochen werden, wenn eine gewisse Anzahl an hintereinander erzeugten Generationen von Lösungen zu keiner Verbesserung des Ergebnisses führt.

Anhand dieser definierten Funktionen kann nun der Gesamtaufbau des genetischen Algorithmus speziell für die Allocation dargestellt werden.

### 10.9 Gesamtablauf der Allocation

Der Gesamtablauf des genetischen Algorithmus für die Allocation ist in Bild 96 dargestellt.

Ist der Schritt der Allocation beendet, so kann die so gefundenen Bauteilmenge im Rahmen des Assignments weiterverwendet werden, was im folgenden Abschnitt genauer dargelegt wird. Die Nutzung von genetischen Algorithmen bietet sich zur Verfeinerung schon gefundener Lösungen, im besonderen dann an, wenn ein Workstationcluster zur Verfügung steht, da gerade

1)Erzeuge Generation $G_{al} \subseteq AL(PR, BLIB)$ verschiedener Assignments

2)Bilde Codierung $C_{al}(BA)$ aller Allocationen $BA \in G_{al}$

3)Bilde die Bewertung $ges_{al}(A)$ aller Allocationen

4)Bilde eine Teilmenge $Sel(G_{al}) \subseteq G_{al}$ der Allocationen durch Selektion

5)Bilde eine neue Generation Allocationen $G'_{al} = gen(G_{al})$

6)Starte mit $G'_{al}$ wieder bei 2), falls Abbruchbedingung nicht erfüllt

Bild 96: Ablauf des Genetischen Algorithmus für Allocation

die Bewertung der Lösungen für jede Lösung unabhängig von einander parallel durchführbar ist.

## 10.10 Assignment und Scheduling mit genetischem Algorithmus

Nachdem eine gewisse Menge an Bauteilen durch ein geeignetes Allocationsverfahren zur Verfügung gestellt wurde, kann nach Def. 9.8-1 jeder Befehl einem Bauteil zugeordnet werden, welches die Operation des Befehls ausführt. Dabei ist zu beachten, daß die Operationen der Befehle auch auf den Bauteilen ausführbar sind, also eine korrekte Zuordnung der Befehle zu den Bauteilen durchgeführt wird. Nach dem Assignment ist für jeden Befehl der Zeitbedarf bekannt, da dieser der Ausführungszeit der entsprechenden Operation auf dem zugeordneten Bauteil entspricht. Ein wesentlicher Schritt, um eine konkrete Zuordnung zu bewerten, ist das Scheduling, welches die Befehle in Kontrollschritte einordnet und somit die benötigte Anzahl der Kontrollschritte berechnet. Wird das Assignment vor dem Scheduling durchgeführt, dann müssen die durch das Assignment zustandegekommenen Abhängigkeiten der Befehle beim Scheduling mit berücksichtigt werden, also der Zeitbedarf jedes einzelnen Befehls und vor allem die Tatsache, daß ein Bauteil nur einen Befehl gleichzeitig ausführen kann. Umgekehrt, also wenn das Scheduling vor dem Assignment ausgeführt wird, dann muß durch das Assignment berücksichtigt werden, daß Befehle, die in den gleichen Taktschritt eingeordnet wurden, auch jeweils unterschiedlichen Bauteilen zugeordnet werden. Im folgenden wird der Weg dargestellt, wie ein konkretes Assignment modifiziert werden kann, damit eine Optimierung möglich ist. Wie im folgenden Abschnitt gezeigt wird, ist schon das Problem NP-vollständig die Reihenfolge der Befehle auf den Bauteilen so zu wählen, daß die Anzahl der benötigten Taktschritte minimal wird. Hierdurch wird die Nutzung von Optimierungsalgorithmen, in diesem Fall den genetischen Algorithmen, motiviert. Die gesamte Optimierung des Assignment stellt außerdem eine Möglichkeit dar, eine gegebene Allocation, welche als Grundlage benutzt wird, zu bewerten.

## 10.11 Die optimale Wahl der Ausführungsreihenfolge ist NP-vollständig

### 10.11.1 Einleitung

Nach dem Assignment wird, wie in Abschnitt 9.8.1 dargestellt, die Ausführungsreihenfolge der Befehle auf den Bauteilen gewählt. Hierdurch wird nach Def. 9.8-9 die Bauteilabhängigkeit definiert. Die optimale Wahl der Ausführungsreihenfolge der Befehle auf den Bauteilen bildet, unter berücksichtigung der Daten-, Antidaten- und Ausgabeabhängigkeiten, ein Optimierungsproblem, welches NP-vollständig ist. Optimal bedeutet an dieser Stelle, daß die Ausführungsreihenfolge so gewählt wird, das die Gesamtausführungszeit minimal wird. Im folgenden wird hierfür der Beweis geliefert. Die Befehle werden als Knoten in der Knotenmenge *V* dargestellt. Die Abhängigkeiten zwischen den Befehlen werden als Kanten in der Kantenmenge *E* darge-

stellt. Es wird zwischen den allgemeinen Abhängigkeiten (Def. 9.3-15), die aus den Daten- (Def. 9.3-12), Antidaten- (Def. 9.3-13) und Ausgabeabhängigkeiten (Def. 9.3-14) bestehen, und der Bauteilabhängigkeit (Def. 9.8-8) unterschieden, indem zusätzlich eine Menge $U \subseteq E$ gebildet wird, die die Bauteilabhängigkeiten darstellt. Ein Richtungswechsel in der Ausführungsreihenfolge der Befehle wird als Richtungswechsel der entsprechenden Kanten aus der Menge $U$ dargestellt. Die Gesamtausführungszeit ist dann minimal, wenn der längste Weg zwischen zwei ausgezeichneten Knoten minimal ist und der Graph azyklisch ist. In dem folgenden Satz wird das Optimierungsproblem, dessen NP-vollständigkeit bewiesen werden soll, formuliert:

*Satz 10.11-1 Das folgende Optimierungsproblem ist NP-vollständig: In einem Graphen $G = (V, E)$ mit einer ausgezeichneten Menge $U \subseteq E$ werden die Kanten aus U so ausgerichtet, daß der längste gerichtete Weg zwischen zwei ausgezeichneten Knoten $a, e \in V$ minimal und der Graph azyklisch ist.*

Die beiden ausgezeichneten Knoten entsprechen den in Bild 71 dargestellten Ein- und Ausgabebefehlen. Für den Beweis wird zuerst ein weiterer Satz, durch eine Reduktion des NP-vollständigen Problems 3-SAT (3-satisfiability problem), bewiesen. Das Ergebnis wird dann für den Beweis von Satz 10.11-1 benutzt.

### 10.11.2 Beweis

Zuerst wird der folgende Satz bewiesen:

*Satz 10.11-2 Folgendes Entscheidungsproblem ist NP-vollständig: In einem Graphen $G = (V, E)$ mit einer ausgezeichneten Menge $U \subseteq E$ werden die Kanten aus U so ausgerichtet, daß es zwischen zwei ausgezeichneten Knoten $a, e \in V$ einen gerichteten Weg gibt und der Graph azyklisch ist.*

Gegeben sei eine konjunktive Normalform (KNF) $F = C_1 \wedge \ldots \wedge C_p$ mit den Klauseln $C_1, \ldots, C_p$ und die Variablen $b_1, \ldots, b_n$. Es wird der Graph $G = (V, E)$ und die Menge $U \subseteq E$ konstruiert so das gilt: $F$ ist erfüllbar genau dann wenn es einen gerichteten Weg zwischen zwei ausgezeichneten Knoten gibt und $G$ ist ein gerichteter azyklischer Graph. Dabei werden für jede Klausel $C_i = (x_{i1}^{\alpha_{i1}} \vee x_{i2}^{\alpha_{i2}} \vee x_{i3}^{\alpha_{i3}})$ mit $x_{i1}, x_{i2}, x_{i3} \in \{b_1, \ldots, b_n\}$ und $\alpha_{i1}, \alpha_{i2}, \alpha_{i3} \in \{0, 1\}$ acht Knoten definiert: $a_i, e_i, u_{i1}, u'_{i1}, u_{i2}, u_{i2}', u_{i3}, u_{i3}' \in V$.

Als ausgezeichneter Startknoten des Weges durch den Graphen wird der Knoten $a_1$ und als Endknoten $e_p$ festgelegt.

Des weiteren werden folgende Kanten definiert $\{u_{i1}, u'_{i1}\}, \{u_{i2}, u_{i2}'\}, \{u_{i3}, u_{i3}'\} \in U \subseteq E$, deren Richtungen noch nicht festgelegt sind. Die gerichteten Kanten werden abhängig von den Literalen für $k \in \{1, 2, 3\}$ wie folgt definiert:

Falls $\alpha_{ik} = 0$ also die Variable $x_{ik}$ in $C_i$ negiert ist, werden die gerichteten Kanten $(a_i, u_{ik}'), (u_{ik}, e_i) \in E/U$ definiert.

Falls $\alpha_{ik} = 1$ also die Variable $x_{ik}$ in $C_i$ positiv ist, werden die gerichteten Kanten $(a_i, u_{ik}), (u_{ik}', e_i) \in E/U$ definiert.

Bild 97 zeigt die Darstellung einer Klausel als Teilgraph. Die so konstruierten Teilgraphen werden dann mit gerichteten Kanten verbunden $(e_i, a_{i+1}) \in E/U$.

Des weiteren werden noch Kanten eingefügt, die gewährleisten sollen, daß die gleiche Variable

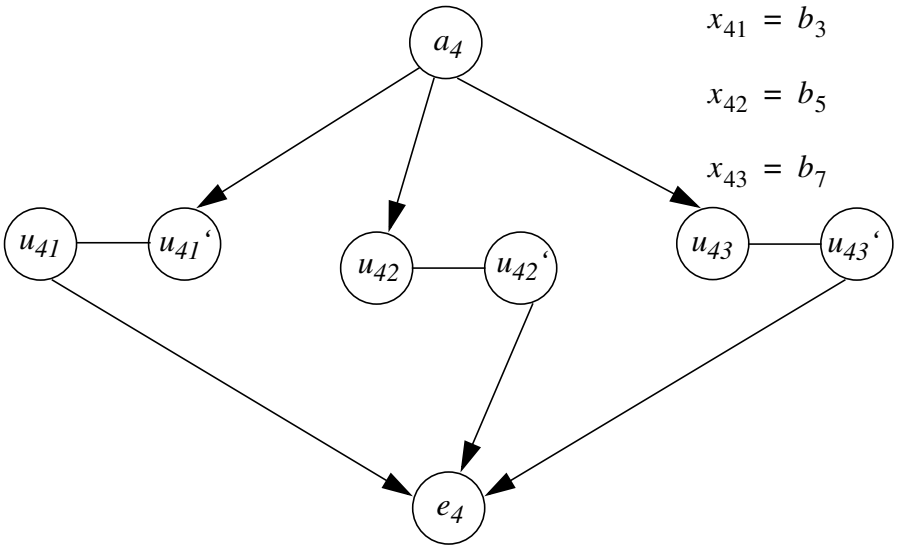



Bild 97: Darstellung der Klausel $C_4 = (\neg b_3 \vee b_5 \vee b_7)$

in allen Klauseln gleich belegt wird, wobei der entstehende Graph bei einer erfüllenden Belegung kreisfrei ist. Seien $C_i$ und $C_j$ zwei Klauseln mit $i < j$ und die Variable $\neg b_l$ mit $l \in \{1, \ldots, n\}$ kommt in $C_i$ vor und die Variable $b_l$ kommt in $C_j$ vor. Es gibt also Variablen $x_{ik} = x_{jm} = b_l$ mit $k, m \in \{1, 2, 3\}$ für die gilt $\alpha_{ik} = 0$ und $\alpha_{jm} = 1$. In diesem Fall wird die Kante $(u_{jm}', u_{ik}') \in E$ hinzugefügt. In Bild 98 wird ein Beispielgraph für diesen Fall dargestellt.

Umgekehrt, also falls $\alpha_{ik} = 1$ und $\alpha_{jm} = 0$ wird die Kante $(u_{jm}, u_{ik}) \in E$ eingefügt. Kommt in zwei Klauseln das gleiche Literal vor, so brauchen diese nicht miteinander verbunden werden. Falls es eine Belegung der Variablen gibt, die $F$ erfüllt, so gibt es einen Weg von $a_1$ nach $e_p$ wenn die ungerichteten Kanten wie folgt, abhängig von der Belegung der Variablen, gerichtet werden:

Falls eine Variable $x_{ik}$ mit 0 belegt wird, wird die entsprechende Kante $\{u_{ik}, u'_{ik}\} \in U \subseteq E$ so gerichtet, daß sie von $u_{ik}'$ nach $u_{ik}$ verläuft. Umgekehrt, falls $x_{ik}$ mit 1 belegt wird, wird die Kante so gerichtet, daß sie von $u_{ik}$ nach $u_{ik}'$ verläuft. Eine erfüllende Belegung einer Klausel $C_i = (x_{i1}^{\alpha_{i1}} \vee x_{i2}^{\alpha_{i2}} \vee x_{i3}^{\alpha_{i3}})$ führt in dem entsprechenden Teilgraphen dazu, daß es mindestens einen gerichteten Weg vom Knoten $a_i$ nach $e_i$ gibt, denn falls $\alpha_{ik} = 0$ führt eine Belegung der Variablen $x_{ik}$ mit 0 zu einer Erfüllung der Klausel, dies bedeutet aber auch, daß es den gerichteten Weg $(a_i, u'_{ik}, u_{ik}, e_i)$ gibt. Eine Belegung mit 1 führt nicht zu einem gerichteten Weg. Umgekehrt also falls $\alpha_{ik} = 1$ führt eine Belegung der Variablen $x_{ik}$ mit 1 zu einer Erfüllung der Klausel, dies bedeutet aber, daß es den gerichteten Weg $(a_i, u_{ik}, u_{ik}', e_i)$ gibt. Da gleiche Variablen in verschiedenen Klauseln auch gleich belegt werden, also die entsprechenden Kanten in die gleiche Richtung zeigen, kann es nicht zu einer Kreisbildung kommen. Gibt es für jede Klausel eine erfüllende Belegung so gibt es einen gerichteten Weg durch jeden Teilgraphen und damit durch den Graphen.

Umgekehrt ist zu zeigen, daß falls es eine Ausrichtung der ungerichteten Kanten aus der Kantenmenge $U$ gibt, so daß es einen Weg von $a_1$ nach $e_p$ gibt und der Graph kreisfrei ist, es auch eine erfüllende Belegung gibt. Es werden dazu nur die Kanten aus $U$ betrachtet, die auf dem gerichteten Weg liegen. Liegt die Kante $(u_{ik}, u_{ik}') \in U$ auf dem gerichteten Weg, so wird $x_{ik}$ mit 1 belegt, liegt umgekehrt $(u_{ik}', u_{ik}) \in U$ auf dem gerichteten Weg so wird $x_{ik}$ mit 0 belegt. Entsprechend wird die Variable $b_l = x_{ik}$ belegt. Variablen, die auf diese Weise nicht festgelegt sind können beliebig gewählt werden. Falls es eine weitere Klausel $C_j$ gibt in der $b_l$ vorkommt

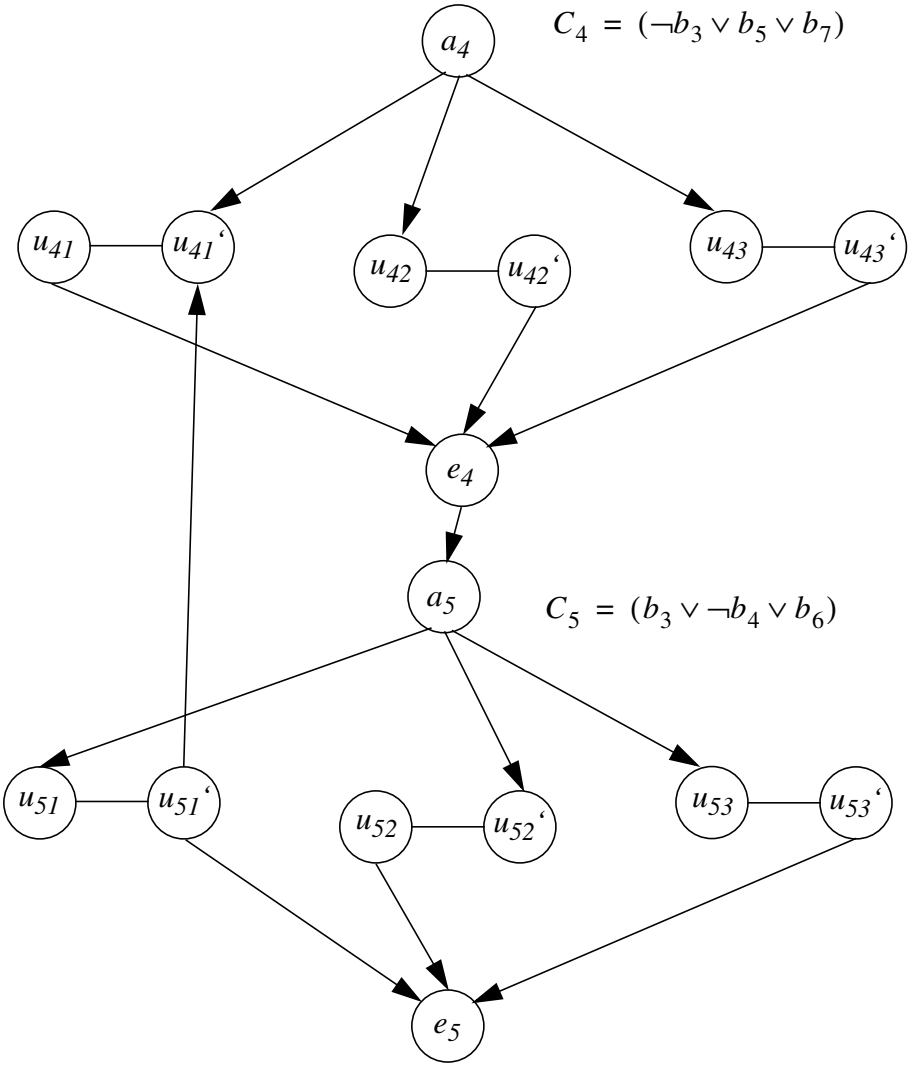



Bild 98: Verbindung zweier Klauseln mit gleichen Variablen

also $b_l = x_{jm}$, so muß wie folgt unterschieden werden: Die Kante $(u_{jm}', u_{jm}) \in U$ oder $(u_{jm}, u_{jm}') \in U$ liegt nicht auf dem gerichteten Weg, dann wird sie nicht für die Belegung von $b_l$ berücksichtigt und die Belegung ist eindeutig. Liegt sie auf dem gerichteten Weg, so zeigt sie entweder in die gleiche Richtung, was ebenfalls kein Problem darstellt, oder in die entgegengesetzte. Dies kann aber nur dann auftreten, wenn $\neg b_l$ in $C_j$ und $b_l$ in $C_i$ oder umgekehrt vorkommt. In diesem Fall gibt es aber eine Kante $(u_{jm}, u_{ik}) \in E$ bzw. umgekehrt die Kante $(u_{jm}', u_{ik}') \in E$. Liegen beide Kanten auf dem gerichteten Weg so entsteht aber der Weg $(a_i, u_{ik}, u_{ik}', e_i, \ldots, a_j, u_{jm}', u_{jm}, u_{ik})$, der einen Kreis enthält, bzw. umgekehrt $(a_i, u_{ik}', u_{ik}, e_i, \ldots, a_j, u_{jm}, u_{jm}', u_{ik}')$. Damit ist der Beweis für Satz 10.11-2 erbracht. Eine Lösung für das Optimierungsproblem aus Satz 10.11-1 ist immer auch eine Lösung für das Entscheidungsproblem aus Satz 10.11-2, womit auch dieser Satz bewiesen ist.

### 10.12 Grundgerüst des genetischen Algorithmus für das Assignment

Die Bild 99 stellt das Vorgehen im Rahmen eines genetischen Algorithmus dar. Im ersten Schritt werden anhand der allocierten Bauteilmenge verschiedene Zuordnungen der Befehle zu den Bauteilen generiert. Die Bewertung dieser Zuordnungen findet durch das Scheduling statt, welches die Anzahl der Kontrollschritte liefert. Außerdem kann an dieser Stelle eine Voraussa-

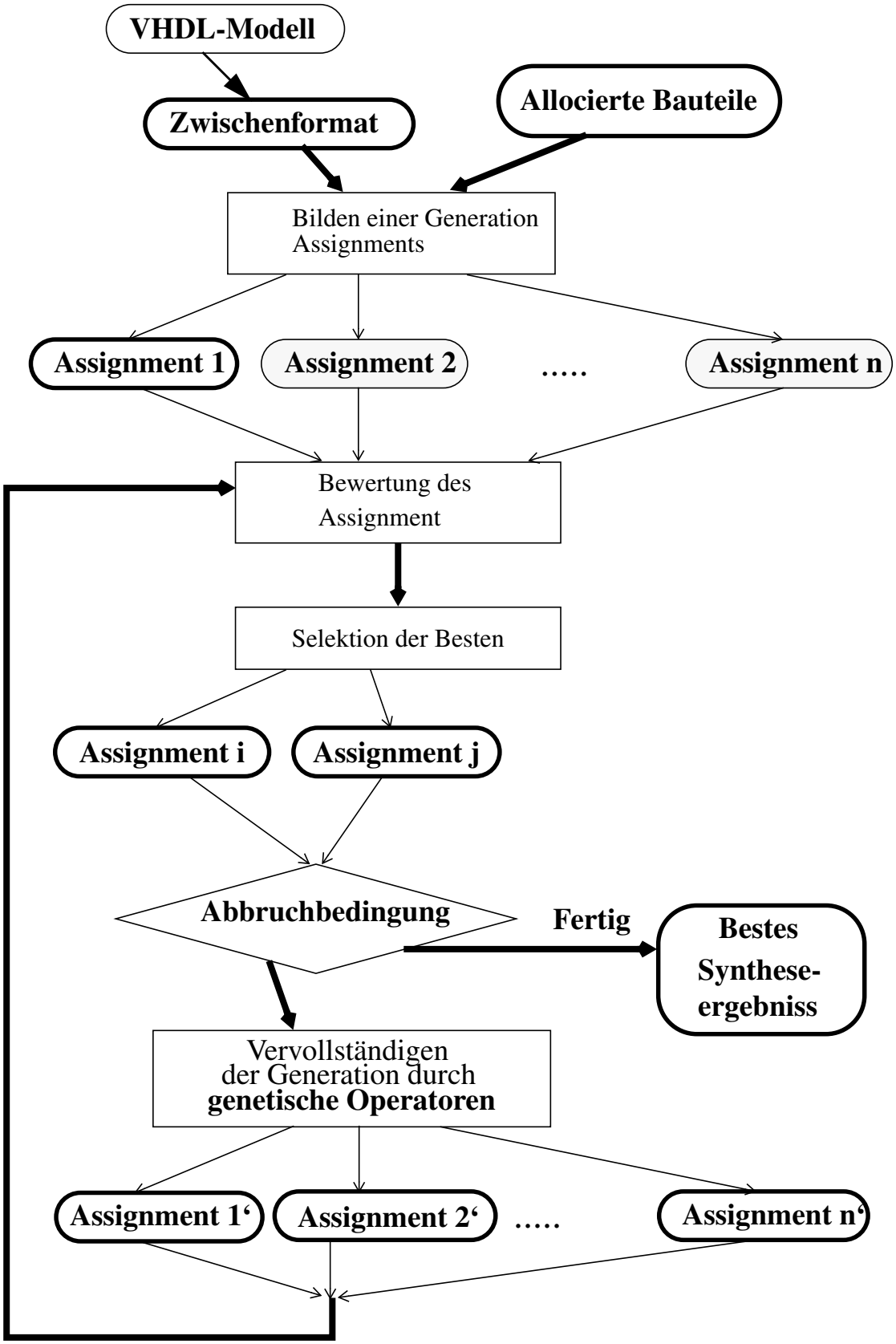



Bild 99: Der Assignmentprozess

ge bzgl. des Ressourcenbedarfs gemacht werden, indem der Ressourcenbedarf der benutzten Bauteile summiert wird. Hieraus geht bei einer Multiplexerarchitektur die Anzahl und Größe der benutzten Multiplexer hervor. Die Variablen werden Registern zugeordnet, wobei die Lebenszeiten der Variablen berücksichtigt werden und somit eine Zuordnung mehrerer Variablen auf ein Register möglich ist. Hierdurch kann der Ressourcenbedarf eingeschätzt werden. Was fehlt, ist eine Abschätzung über den Verdrahtungsaufwand der Schaltung auf dem Chip. Nachdem jedes Assignment bewertet wurde, findet eine Selektion der besten Assignments statt. Aus diesen werden dann mit Hilfe der genetischen Operatoren Mutation und Crossover soviele neue Assignments gebildet, daß eine vollständige Generation zur Verfügung steht. Die Vorgänge des genetischen Algorithmus wiederholen sich dann mit der neuen Generation Assignments. Wird die Abbruchbedingung erfüllt, so wird das bis dahin beste gefundene Assignment ausgegeben.

In dem folgenden Abschnitt soll nun die Codierung eines Assignments und die Mutations- und Crossoverfunktionen für einen genetischen Algorithmus dargestellt werden.

### 10.13 Codierung

Für die Zuordnung der Befehle zu den Bauteilen gibt es mehrere Möglichkeiten, die in Form eines Codes dargestellt werden müssen. Eine Möglichkeit, diese Zuordnung darzustellen, ist, für jeden Befehl ein Zeichen im String zu reservieren. Auf der anderen Seite werden die allocierten Bauteile, die in einer Liste stehen, gemäß ihrer Reihenfolge durchnumeriert. Eine Codierung der Zuordnung der Befehle zu den Bauteilen sieht nun so aus, daß in jeder Stringposition, die ja einen Befehl repräsentiert, die Nummer des Bauteils steht, dem der korrespondierende Befehl zugeordnet ist.

*Definition 10.13-1 Eine Codierung für das Assignment ist eine Abbildung*

$$C_{as}: L(B, BA) \rightarrow \Sigma^{|B|}.$$

Das heißt, jedem Befehl wird ein Buchstabe aus dem Alphabet $\Sigma = N$ zugewiesen. Auch hier ist die Codierung nicht widerspruchsfrei, da es Codes geben kann, die zu einem nicht gültigen Assignment führen.

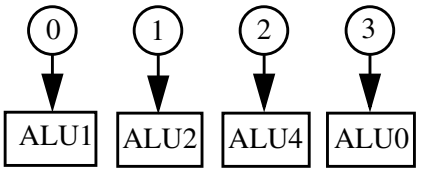



Bild 100: Assignmentcodierung

In Bild 100 ist das Beispiel einer Codierung eines Assignments von vier Befehlen dargestellt, die den Bauteilen *ALU0*, *ALU1*, *ALU2* und *ALU4* zugewiesen werden. Jedem Befehl, welcher hier als Kreis dargestellt wird, wird ein Bauteil zugewiesen, wobei natürlich auch mehrere Befehle dem gleichen Bauteil zugewiesen werden können. Die Gültigkeit der Zuweisungen ist - wie oben schon definiert - dadurch gegeben, daß das Bauteil, welches einem Befehl zugewiesen wird, auch die Operation des Befehls zur Verfügung stellt.

### 10.14 Dekodierung

Die Dekodierung eines Codes ist die Umkehrfunktion der Codierung.

*Definition 10.14-1 Die Dekodierungsfunktion des Assignments ist eine Abbildung*

$D_{as}: \Sigma^{|B|} \rightarrow L(B, BA)$, *es handelt sich dabei um die Umkehrfunktion von* $C_{as}$.

Die Dekodierungsfunktion errechnet das Assignment $A$ aus einem Code. Da für jeden Befehl genau eine Position im Code fixiert ist, kann die Zuordnung dadurch eindeutig festgelegt werden. Da auch nicht gültige Assignments codiert werden können, muß jeder Code bzw. jedes Assignment in der Implementierung auf seine Gültigkeit getestet werden.

### 10.15 Mutation

Die Mutationsfunktion wird auf eine codierte Lösung des Assignmentproblems angewandt, so daß eine neue Lösung entsteht. Eine geringe Veränderung der Zuordnung ist beispielsweise dann gegeben, wenn nur ein Befehl einem anderen Bauteil zugewiesen wird, welches für die Ausführung zuständig ist. Hierbei ist natürlich darauf zu achten, daß die Mutation nicht zu einer Zuordnung führt, die ungültig ist, das heißt der Befehl einem Bauteil zugeordnet wird, welcher die entsprechende Funktion nicht ausführen kann.

*Definition 10.15-1 Die Mutationsfunktion für das Assignment ist eine Abbildung* $m_{as}: \Sigma^{|B|} \rightarrow \Sigma^{|B|}$ *der Codierung der Assignments auf die Codierung.*

Die Mutationsfunktion stellt eine Zuordnung der Befehle zu den Bauteilen her, wobei auch hier der Fall eintreten kann, daß eine ungültige Zuordnung produziert wird.

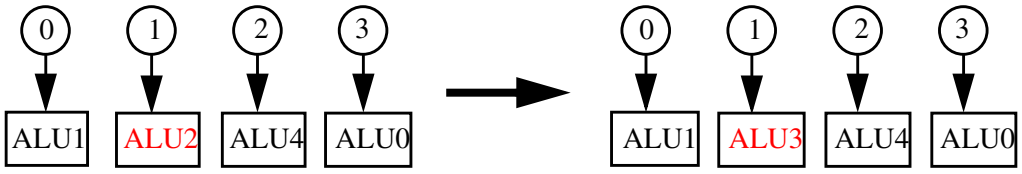



Bild 101: Mutation eines Assignment

In Bild 101 ist das Beispiel einer Mutation dargestellt, bei der die Zuweisung des Befehls eins verändert wird. Das Beispiel macht deutlich, daß sich die Veränderung eines Assignments nur auf die Zuweisung genau eines Befehls zu einem Bauteil auswirkt. Es sind selbstverständlich auch Mutationen möglich, die gröbere Veränderungen nach sich ziehen.

### 10.16 Crossover

Der Crossoveroperator, der zwei verschiedene Assignment-Lösungen mischt, sieht bei der oben definierten Codierung der Lösungen so aus, daß die Befehlsmenge an einer Stelle aufgebrochen wird und die so entstandenen Teile über Kreuz wieder zusammengefügt werden.

*Definition 10.16-1 Der Crossoveroperator für das Assignment ist eine Abbildung* $cr_{as}: \Sigma^{|B|} \times \Sigma^{|B|} \rightarrow \Sigma^{|B|}$ *von zwei Mengen der Codes auf eine Menge von Codes.*

In der Bild 102 ist für ein Beispiel dargestellt, wie sich die Implementierung des Crossoveroperators auswirkt. Durch das Crossover kann keine ungültige Lösung zustandekommen, wenn zwei gültige Lösungen gekreuzt werden.

In Bild 102 ist das Crossover zweier Assignments dargestellt, der Crossoverpunkt liegt in diesem Beispiel genau in der Mitte. Im Beispiel werden die Zuweisungen der Befehle eins und zwei von der ersten Lösung und die Zuweisungen der Befehle drei und vier von der zweiten Lö-

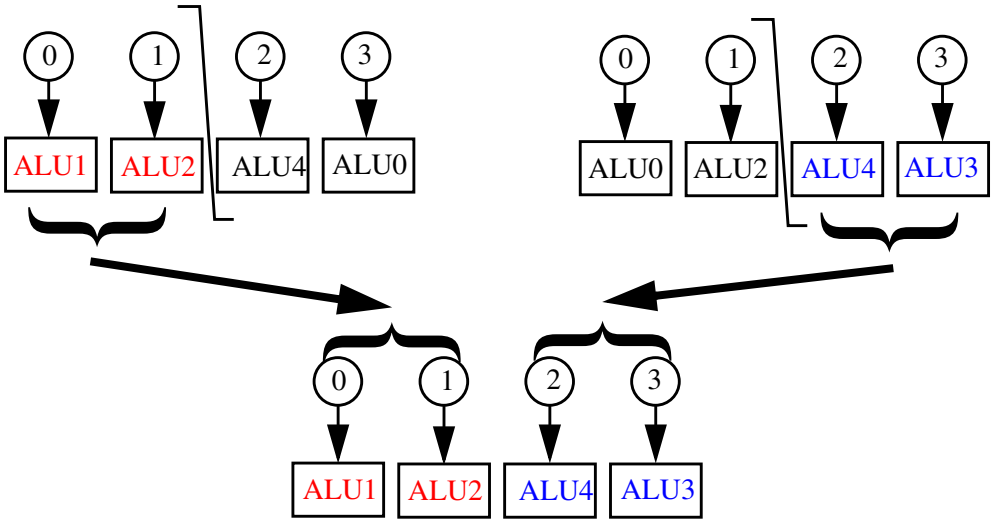



Bild 102: Crossover zweier Assignments

sung übernommen.

### 10.17 Zusätzliche Mutationsmöglichkeit

Eine weitere Möglichkeit, eine Mutation, also eine kleine Veränderung, zuzulassen, ist auf der Ebene der Bauteilabhängigkeiten gegeben. Durch die Zuordnung der Befehle zu den Bauteilen wird zwar für jedes Bauteil die Menge der ihm zugeordneten Befehle definiert, aber die Reihenfolge, also die sich daraus ergebende Permutation, ist noch offen. Daraus folgt: für eine Optimierung der Schedulingergebnisse müssen diese Abhängigkeiten veränderbar sein. Es muß also ein weiterer Mutationsoperator definiert werden, welcher es ermöglicht, Bauteilabhängigkeiten zu verändern. Für diese Veränderung hat es sich als sinnvoll erwiesen, dies nicht auf dem Code durchzuführen, der speziell für den schon beschriebenen Crossover- und Mutationsoperator definiert wurde. Um die Bauteilabhängigkeit zu verändern, werden die Permutationen umsortiert.

*Definition 10.17-1 Eine Mutation der Bauteilabhängigkeit für ein Bauteil u besteht in einer Umsortierung der entsprechenden Permutation* $p \in PE(u)$*, es handelt sich also um eine Abbildung der Form* $m_u: PE(u) \rightarrow PE(u)$.

In der Bild 103 ist die Mutation der Bauteilabhängigkeit dargestellt. Nachdem eine Reihenfolge der Befehle gewählt wurde, kann diese für jedes Bauteil durch eine Permutation dargestellt werden. Der Mutationsoperator sortiert eine Permutation um, so daß die Bauteilabhängigkeit verändert wird. Im Graphen kann dies durch die Umkehrung der Richtung der entsprechenden Kante dargestellt werden. Beide hier dargestellten Mutationsoperatoren werden durch die Anwendung einer einfachen Heuristik optimiert: Die für eine Mutation zugelassenen Befehle werden aus der Menge der Befehle ausgewählt, die auf dem kritischen bzw. längsten Pfad des Abhängigkeitsgraphen liegen. Dieser Pfad wird vor jedem Mutationsschritt neu berechnet.

### 10.18 Die Bewertung

Die Bewertungsfunktion eines Assignments wird durch die Berechnung der real genutzten Ressourcen einerseits und des Bedarfs an Kontrollschritten andrerseits durchgeführt. Somit muß für jede Bewertung einer Lösung der Schedulingalgorithmus durchgeführt werden. Die Bewertungsfunktionen für das Assignment werden mit $g_{as,i}$ definiert, woraus sich dann unter Benut-

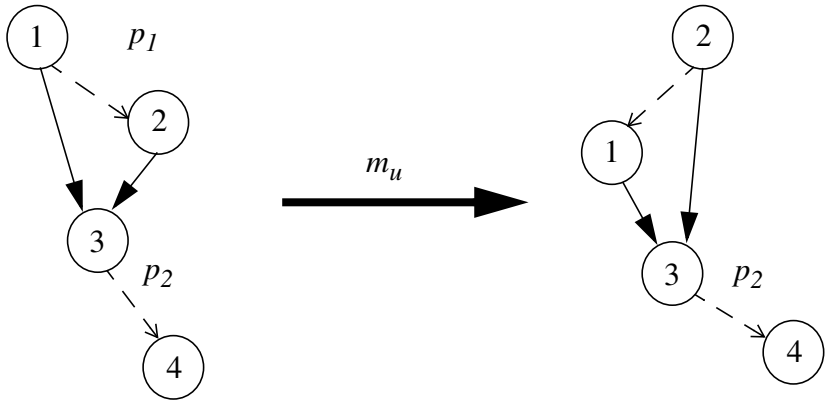



Bild 103: Mutation der Permutation $p_1$

zung eines Gewichtsvektors $w_{as}$ die Funktion für die Gesamtbewertung $ges_{as}$ zusammensetzt. Es gibt also, wie schon oben dargestellt, keine eindeutige Bewertungsfunktion, sondern die Bewertung einer Lösung muß differenziert werden. Durch die verschiedenen Ziele, die bei der Bewertung betrachtet werden müssen, sind bei der Mehrzieloptimierung viele Algorithmen überfordert, da sie eher eine Spezialität von genetischen Algorithmen ist. Die in Bild 104 angegebenen Bewertungsfunktionen lassen sich dann für den hier dargestellten Fall unterscheiden.

$g_{as,1}$ : Anzahl Taktschritte

$g_{as,2}$ : Bauteilbedarf (Anzahl)

$g_{as,3}$ : Bauteilbedarf (Ressourcen)

$g_{as,4}$ : Registeranzahl

$g_{as,5}$ : Multiplexerbedarf

$g_{as,6}$ : Verdrahtungsaufwand

$g_{as,7}$ : Energiebedarf

Bild 104: Bewertungsfunktionen

Einige Bewertungsfunktionen wurden schon definiert, so daß in diesen Fällen auf die jeweilige Definition verwiesen wird. Ansonsten werden alle Bewertungsfunktionen im folgenden genau dargestellt.

### 10.18.1 Anzahl Taktschritte

Die Anzahl der benötigten Taktschritte hängt von der Zuordnung der Befehle zu den Bauteilen ab. Der Zeitbedarf der einzelnen Befehle wird in Taktschritten angegeben, der sich aus dem größten gemeinsamen Teiler aller Zeiten, die in der Bauteilbibliothek vorkommen, zusammensetzt. Durch den Schedulingalgorithmus wird die maximale Anzahl an benötigten Taktschritten berechnet. Die Bewertungsfunktion kann also anhand der Def. 9.13-3 dargestellt werden. Das Gewicht wird für diese Funktion meistens relativ hoch gewählt, da es in den meisten Applikation mehr auf den Zeitgewinn als auf den Ressourcenbedarf ankommt.

*Definition 10.18-1 Die Anzahl der Taktschritte einer Schaltung für das Programm PR und das Assignment A mit gegebener Taktschrittlänge ts ist definiert als* $g_{as,1} = Bt(PR, A)/ts$.

### 10.18.2 Bauteilbedarf

Die Berechnung des Bauteilbedarfs kann anhand der Anzahl benötigter Bauteile geschehen. Es wird sozusagen für jedes Bauteil der Einheitswert angegeben. Diese Gewichtsfunktion ist dann sinnvoll, wenn über die Implementierung des Bauteils noch nichts bekannt ist. Das heißt, wenn jedes verwendete Bauteil als Black-Box betrachtet wird. Mit Hilfe dieser Bewertungsfunktion ist also das Erreichen guter Syntheseergebnisse möglich, wenn noch nicht alle Daten zur exakten Bewertung vorhanden sind. Die Funktion $g_{as,2}$ läßt sich als Anzahl der benutzten Bauteile definieren:

*Definition 10.18-2 Die Anzahl benötigter Bauteile einer Bauteilmenge BA für die Realisierung einer Befehlsmenge B ist bzgl. des Assignments A gegeben durch* $g_{as,2} = \left|\{v|(v \in BA)(\exists b \in B)(v = A(b))\}\right|$.

### 10.18.3 Ressourcenbedarf

Weiter differenzieren läßt sich der Bauteilbedarf, wenn der benötigte Ressourcenbedarf für die Bauteile bekannt ist. Dann kann die Bewertungsfunktion für den Ressourcenbedarf, wie in Def. 9.8-13 schon angegeben, dargestellt werden.

*Definition 10.18-3 Der Ressourcenbedarf für die Realisierung einer Befehlsmenge B ist bzgl. des Assignments A definiert durch* $g_{as,3} = Bres(A)$.

Mit dieser Bewertungsfunktion soll nicht die Anzahl der benötigten Bauteile, sondern der tatsächliche Ressourcenbedarf minimiert werden. Hierzu muß der Ressourcenbedarf, falls er nicht bekannt ist, zumindest abschätzbar sein.

### 10.18.4 Registeranzahl

Die Anzahl der benötigten Register ist erst dann bekannt, wenn das Scheduling durchgeführt wurde. Da ein etwas anderer Mechanismus bei der Bewertung des Registerbedarfs als bei der des Ressourcenbedarfs vorhanden ist, sind diese beiden Punkte getrennt voneinander zu betrachten. Der Ressourcenbedarf läßt sich, ohne ein Scheduling durchgeführt zu haben, allein anhand des Assignments errechnen. Der endgültige Registerbedarf ist erst dann abschätzbar, wenn das Scheduling durchgeführt ist und die Variablen z.B. durch einen left-edge Algorithmus zusammengefaßt wurden. Die in Def. 9.10-5 angegebene Definition läßt sich hier als Bewertungsfunktion einsetzen.

*Definition 10.18-4 Die Anzahl benötigter Register für die Realisierung einer Variablenmenge V ist gegeben durch* $g_{as,4} = minv(V)$.

Durch die Zusammenfassung und mehrfache Nutzung verschiedener Variablen steigt die Anzahl benötigter Multiplexer, weshalb hier ebenfalls eine entsprechende Bewertung durchgeführt werden muß.

### 10.18.5 Multiplexerbedarf

Der Bedarf an Multiplexern und deren Größe, also die Anzahl der Eingänge, ist erst berechenbar, wenn die konkrete Netzliste aus den vorhandenen Daten generiert worden ist. Für die Berechnung der Größe und der Anzahl der benötigten Multiplexer heißt das, daß für jedes Register die Anzahl der Bauteile berechnet werden muß, die auf dieses Register schreiben. Für die Bau-

teile heißt das, daß für jeden Eingang die Anzahl der Register herausgefunden werden muß, die diesen Eingängen vorgeschaltet sind. Ist einem Eingang nur ein Register bzw. Bauteil vorgeschaltet, so wird hier kein Multiplexer benötigt. Die Bewertungsfunktion für den Multiplexerbedarf könnte noch bzgl. der Anzahl der Eingänge der Multiplexer differenziert werden. Im Rahmen dieser Arbeit soll diese Unterscheidung nicht gemacht werden. Die folgenden Bedingungen können zur Berechnung des Multiplexerbedarfs angegeben werden, wobei auf die Def. 9.10-5 verwiesen werden soll.

*Satz 10.18-5 Einem Register muß genau dann ein Multiplexer vorgeschaltet werden, wenn gilt:*

*1) die dem Register entsprechende Partition* $pa \subseteq V$ *enthält mindestens zwei Variablen. also* $|pa| > 1$ *oder*

*2) falls* $|pa| = 1$ *muß gelten: die Anzahl der Befehle, die auf das Register schreiben, beträgt mindestens 2.*

*Satz 10.18-6 Einem Bauteil u muß genau dann Multiplexer vorgeschaltet werden, wenn die Anzahl der ihm zugewiesenen Befehle größer als 1 ist, also* $\left|A^{-1}(u)\right| > 1$.

Für den Bedarf an Multiplexern wird die Funktion $g_{as,5}$ reserviert. Aus dem Aufwand für Multiplexer kann der Aufwand für die Verdrahtung abgeleitet werden.

*Definition 10.18-7 Die Anzahl Multiplexer, die benötigt wird, um eine Befehlsmenge B mit einer Bauteilmenge BA zu synthetisieren - wobei A das Assignment ist und* $pa_i$ *die Menge der Variablen, die dem Register* $reg_i$ *zugewiesen sind, darstellen - wird mit der Funktion* $g_{as,5}$ *dargestellt und berechnet durch:*

$$g_{as,5} = |\{pa|(|pa| > 1)\}| + |\{pa|(|pa| = 1) \wedge (v \in pa) \wedge (|\{b|(b \in B) \wedge (v \in out(b))\}| > 1)\}| + |\{u|(u \in BA) \wedge (A(u) > 1)\}|$$

In der Definition zur Berechnung des Multiplexerbedarfs werden also die in Satz 10.18-5 und Satz 10.18-6 dargestellten Bedingungen umgesetzt.

### 10.18.6 Verdrahtungsaufwand

Eine endgültige Bewertung des Platzbedarfs vorzunehmen, ist erst möglich, wenn der Verdrahtungsaufwand für eine Schaltung bekannt ist. Das Problem ist, daß die Verdrahtung auf einem Chip einen sehr großen Flächenanteil (bis 90% der Fläche) beansprucht. Berechenbar jedoch ist die Anzahl der Verbindungen, die nötig sind. Diese ergibt sich aus der Zuordnung der Befehle zu den Bauteilen und der Zuordnung der Variablen zu den Registern. Anhand der Befehle müssen nun Verbindungen von den Registern zu den Bauteilen bzw. von den Bauteilen zu den Registern generiert werden. Dabei ist auf die Benutzung von Multiplexern zu achten, durch die sich die Anzahl der Verbindungen noch einmal erhöht. Unter der Annahme, daß die Anzahl der Verbindungen berechnet wurde, kann für die Berechnung einer Näherung die folgende Überlegung angestellt werden. Wenn zu einem Chip, der mit einer bestimmten Fläche $F$ - die durch schon plazierte und verdrahtete Bauteile zustande gekommen ist - gegeben ist, nun eine Verbindung hinzugefügt wird, so kann davon ausgegangen werden, daß ihre Länge mit der Kantenlänge des Chips korreliert. Die folgende rekursive Formel bildet eine Schätzfunktion für den Platzbedarf, der durch die Verdrahtung zustandekommt. Die Formel berechnet dabei die Flä-

che, die durch Hinzufügen einer Verbindung zustandekommt.

$$F_{i+1} = F_i + c\sqrt{F_i}$$

Dabei ist $F_0$ der Platzbedarf, der durch die Bauteile, Register und Multiplexer gegeben ist. Damit kann der Platzbedarf auch in geschlossener Form durch $F_n = \Theta(n^{3/2})$ angenähert werden.

*Definition 10.18-8 Die Bewertungsfunktion für die Verdrahtung der Bauteile mit n Verbindungen kann als* $g_{as,6} = F_n - F_0$ *angegeben werden.*

Durch diese Definition des Verdrahtungsaufwandes wird nur die Verdrahtung betrachtet. Der Platzbedarf der Bauteile wird mit anderen Bewertungsfunktionen abgedeckt.

### 10.18.7 Energiebedarf

Die Berechnung des Energiebedarf einer Schaltung ist im wesentlichen von dem Energiebedarf bei der Umschaltung von Gatterein- und ausgängen abhängig. Beispielsweise fließt in einer CMOS-Schlatung ein vernachläßigbar geringer Ruhestrom. Wird nun ein Eingang eines Gatters umgeschaltet, so bedeutet dies einen Umladevorgang eines Kondensators über einen Widerstand. Ist der Energiebedarf einer Operationseinheit für die Berechnung einer Operation bekannt, so kann der Energiebedarf einer synthestisierten Schaltung grob durch Summenbildung abgeschätzt werden. Eine genaue Berechnung ist nicht möglich, da die Widerstände proportional zur Länge der Verbindungen sind, die noch nicht bekannt sind. Die einfachste Abschätzung, mit der aber zumindest schon eine Richtung angegeben werden kann, ist die Annahme, daß der Energiebedarf proportional zur Anzahl der Befehle eines Algorithmus ist. Diese Annahme ist auch einsichtig, da nicht die Anzahl der Bauteile die benutzt werden in die Berechnung eingehen, sondern die Anzahl der Umschaltvorgänge, oder wie oft die Bauteile genutzt werden, was ja für jede zu bewertende Lösung gleich ist. In dieser Arbeit wurde daher auf eine eingehende Betrachtung des Energiebedarfs einer Schaltung verzichtet.

### 10.18.8 Die Gesamtbewertung des Assignments

Die Gesamtbewertung eines Assignments setzt sich zusammen aus den einzelnen Bewertungsfunktionen des Assignments, die mit dem Gewichtsvektor $w_{as}$ in die Bewertungsfunktion $ges_{as}$ eingehen, und ist definiert als:

*Definition 10.18-9 Die Gesamtbewertung eines Assignment ist definiert als*

$ges_{as} = \sum_{i=1}^{6} w_{as,i} \cdot g_{as,i}$, *wobei der Gewichtsvektor* $w_{as} = (w_{as,1}, \ldots, w_{as,6})$ *gewählt werden muß.*

Für jede Lösung ist neben der Bewertung zu klären, ob das Assignment zulässig ist, und wenn die Bauteilabhängigkeiten feststehen, ob die transitive Hülle der Abhängigkeiten gebildet werden kann. Hieraus ergibt sich der weiter unten dargelegte Algorithmus.

## 10.19 Selektion der Assignments

Die Selektionsfunktion sucht aus einer bewerteten Generation von Lösungen eine Teilmenge an Lösungen aus, die der Mutation und dem Crossover zur Verfügung gestellt und aus denen dann wieder so viele Lösungen gebildet werden, daß eine vollständige Generation entsteht. Die Selektion wird dabei anhand der Güte oder der Bewertung der Lösung vorgenommen. In Def. 8.8-2 wurde der allgemeine Selektionsoperator definiert. Die Selektion besteht in dem hier darge-

stellten Fall darin, daß die Lösungen einer Generation bzgl. ihrer Güte sortiert werden. Dann werden die schlechteren Lösungen verworfen und durch Lösungen ersetzt, die durch das Crossover der besseren Lösungen entstehen. Die erhalten gebliebenen Lösungen werden dann mutiert. In Bild 105 ist die Einordnung der Selektionsfunktion in den Gesamtablauf dargestellt.

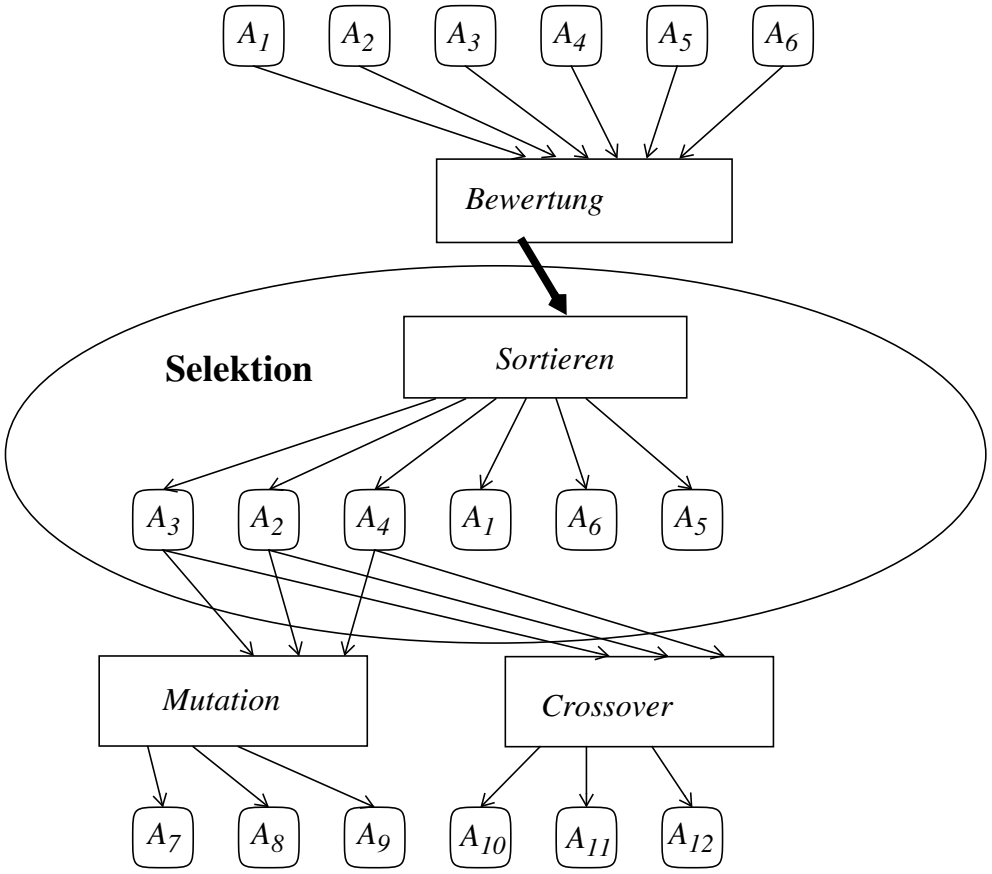



Bild 105: Die Einordnung der Selektionsfunktion

### 10.20 Abbruchbedingung

Nachdem ein genetischer Algorithmus mehrere Generationen erzeugt hat - und so hofft man - bessere Lösungen gefunden sind, muß es ein Abbruchkriterium geben. Das einfachste Abbruchkriterium eines genetischen Algorithmus besteht darin, den Algorithmus nur eine bestimmte festgelegte Anzahl an Generationen laufen zu lassen, wie es in unserem Fall angewandt wurde. Andrerseits kann ein Abbruchkriterium formuliert werden, das den genetischen Algorithmus dann abbricht, wenn über eine gewisse Zeit, bzw. eine Anzahl an Generationen, keine Verbesserung der Ergebnisse erreicht wurde.

### 10.21 Der Gesamtablauf des genetischen Algorithmus für Assignment

Der Gesamtablauf des genetischen Algorithmus für das Assignment ist in Bild 106 dargestellt.

Zu diesem genetischen Algorithmus ist zu sagen, daß die Selektionsfunkion hier nicht im Detail dargestellt wurde. Sie besteht im wesentlichen aus dem Sortieren der Menge bzgl. der Bewertung und der Auswahl der guten Lösungen. Aus den ausgewählten Lösungen wird dann mit den Funktionen Crossover und Mutation eine neue Generation von Lösungen gebildet, bei der darauf zu achten ist, daß sie nur gültige Lösungen enthält.

1)Erzeuge Generation $G_{as} \subseteq L(B, BA)$ verschiedener Assignments

2)Bilde Codierung $C_{as}(A)$ aller Assignments $A \in G_{as}$

3)Bilde die Bewertung $ges_{as}(A)$ aller Assignments

4)Bilde eine Teilmenge $Sel(G_{as}) \subseteq G_{as}$ der Assignments durch Selektion

5)Bilde eine neue Generation Assignments $G'_{as} = gen(G_{as})$

6)Starte mit $G'_{as}$ wieder bei 2), falls Abbruchbedingung nicht erfüllt

Bild 106: Ablauf des genetischen Algorithmus für Assignment

### 10.22 Einsatz zusätzlicher heuristischer Methoden

Der Einsatz heuristischer Verfahren in Verbindung mit genetischen Algorithmen bietet sich gerade in dieser Anwendung an. Genetische Algorithmen können verschiedene gute Ergebnisse liefern, die aber durch die zufälligen Mutationen nur noch sehr langsam und über mehrere Generationen optimiert werden. Unter Umständen kann hier ein einfaches heuristisches Verfahren, angewandt auf das beste Lösung einer Generation, schneller zu guten Ergebnissen führen. Die in Abschnitt 10.17 dargestellte Mutation wird für jede Lösung nur einmal pro Generation auf einen zufällig gewählten Befehl angewandt. Im Kapitel 13 ist dargestellt, zu welchen Verbesserungen ein einfacher heuristischer Algorithmus, angewandt auf das beste Ergebnis jeder Generation, führt. Der Algorithmus wählt dazu die beste Lösung einer Generation und führt die Mutation gezielt auf eine Teilmenge der Befehle auf dem kritischen Pfad aus. Nach jeder Mutation wird geprüft, ob eine Verbesserung zustande gekommen ist. Das so erhaltene Ergebnis wird zur aktuellen Generation hinzugefügt. Die Anzahl der Mutationen wird so gewählt, daß der Algorithmus keine Verzögerung des genetischen Algorithmus in einer parallelen Workstation Umgebung bewirkt. Er wird auf dem Hauptrechner ausgeführt, der auf die Ergebnisse der Bewertungsfunktion, die auf anderen Rechner durchgeführt werden, wartet.

## 11 Wiederverwertbarkeit von optimierten Designs

### 11.1 Einleitung

Die High-Level Synthese beschäftigt sich im wesentlichen mit der automatischen Implementierung von abstrakt formulierten Beschreibungen einer Schaltung, wobei die Implementierung eine Realisierung der Funktion mit Hilfe verschiedener Unterfunktionen darstellt, die ihrerseits wieder auf unterschiedlichen Abstraktionsebenen beschrieben werden können. Der Graph in Bild 107 zeigt zum Beispiel die Funktionen $f_1$ und $f_2$ und deren Realisierung als Graph.

Synthetisiert man nun die Funktionen $f_1$ und $f_2$ unabhängig voneinander, so werden zwei Schaltungen erzeugt, wobei ALUs für die Funktionen +, -, * zweimal zur Verfügung gestellt werden müssen. Durch den Graphen in der höheren Hierarchieebene ist ersichtlich, daß die beiden Funktionen nicht gleichzeitig ausgeführt werden sollen. Somit ist hier eine Ressourcenverschwendung gegeben. Um sie zu umgehen, wird die Hierarchie aufgelöst. Diese Darstellung ist dann als einziger Graph zu betrachten und entsprechend zu synthetisieren. Der Nachteil einer solchen Lösung ist, daß die Funktionen bzw. die Bauteile, die die beiden Funktionen realisieren, nicht noch einmal oder in einem anderen Zusammenhang benutzt werden können. Außerdem ist der Zeitaufwand für die Optimierung großer Schaltungen sehr viel höher als für die Optimierung mehrerer kleiner. Der Vorteil, der durch Partitionierungsalgorithmen zur Beschleunigung

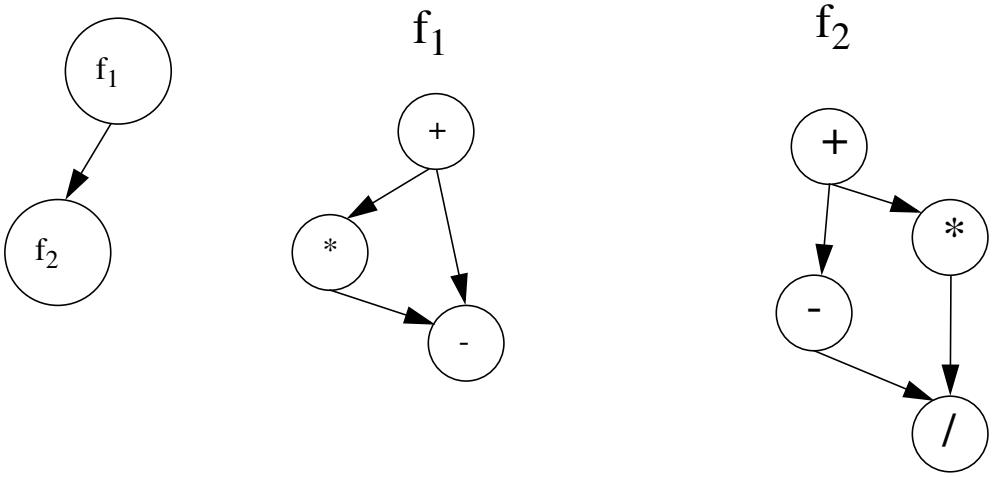



Bild 107: Die Datenflußgraphen der Funktionen $f_1$ und $f_2$

der Synthese erreicht wird, wird wieder durch den erhöhten Ressourcenbedarf zunichte gemacht. Im folgenden wird kurz eine Methode vorgestellt, die es erlaubt, Schaltungen unabhängig voneinander zu synthetisieren, wobei sie dann aber in einer einzigen ALU untergebracht und die von beiden benötigten Bauteile gemeinsam genutzt werden.

### 11.2 Synthese einer ALU

Der Ablauf der Synthese sieht so aus, daß im ersten Schritt eine unabhängige Synthese für beide Funktionen durchgeführt wird. Nachdem der Bauteilbedarf und die Zuordnung der Befehle zu den Bauteilen, das Scheduling und der Registerbedarf beider Schaltungen berechnet sind, wird die Vereinigungsmenge der beiden Mengen real genutzter Bauteile gebildet. Die Vereinigungsmenge wird so gebildet, daß für jeden Bauteiltyp so viele Instanzen gebildet werden, wie sie von einer Funktion maximal benötigt werden. Benötigt also eine Schaltung drei Addierer und die andere nur einen, so werden drei Addierer zur Verfügung gestellt. Wichtig ist, daß die errechneten Zuweisungen der Befehle zu Bauteilen erhalten bleiben. Außerdem wird nur einmal die maximal benötigte Menge an Registern zur Verfügung gestellt. Das Scheduling bleibt ebenfalls erhalten. Diese Information reicht aus, um eine Netzliste zu erzeugen. Es können entweder zwei unabhängige Kontroller erzeugt werden oder ein Kontroller, der es erlaubt, durch ein Auswahlbit entweder das eine oder das andere Programm ablaufen zu lassen. In Bild 108 ist die

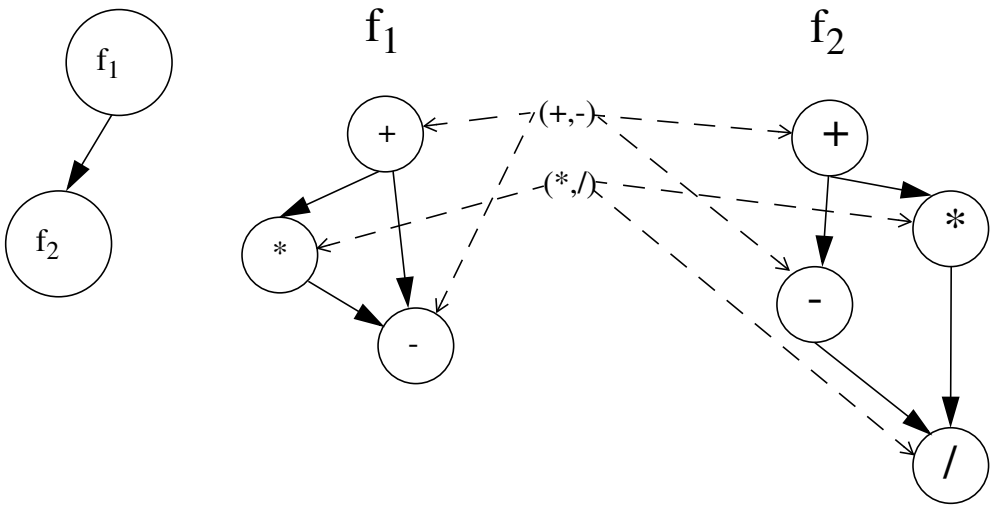



Bild 108: Gemeinsame Nutzung einer Bauteilmenge

gemeinsame Nutzung einer Bauteilmenge dargestellt. Wie hier ersichtlich ist, werden im wesentlichen die Assignments auf diese Weise modifiziert.

### 11.2.1 Die erzeugte Architektur

Die Architektur einer so erzeugten ALU ist in Bild 109 dargestellt. Die Komponenten Netzliste-1 und Netzliste-2 werden durch die gemeinsame Netzliste ersetzt, wobei die Kontroller erhalten bleiben, aber eine Umschaltung der Steuerleitungen durch einen Multiplexer durchgeführt wird. Durch die Steuerleitung des Multiplexers wird die Funktion ausgewählt, so daß hier der Zusammenhang zu den schon definierten Bauteilen besteht.

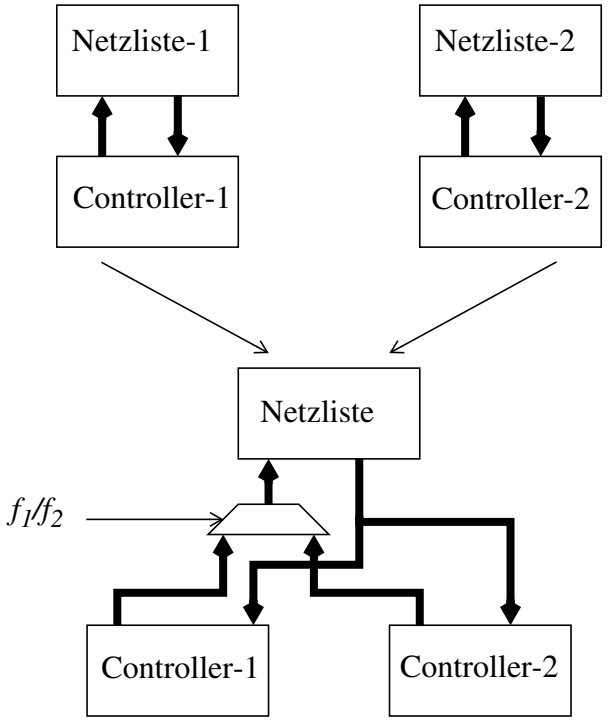



Bild 109: ALU Architektur für zwei Funktionen

Im folgenden ist der formalisierte Ablauf dieser Synthese dargestellt. Die Funktion $f_1$ sei durch die Befehlsmenge $B_1$ gegeben, und es stehen die Bauteile $BA_1$ zur Verfügung. Das Assignment $A_1$ ist durch die oben dargestellten Syntheseschritte berechnet. Ebenso verhält es sich mit der Funktion $f_2$, deren Befehlsmenge durch $B_2$ gegeben ist. Das Assignment $A_2$ weist jedem Befehl ein Bauteil aus $BA_2$ zu. Außerdem ist das Scheduling der beiden Befehlsmengen durchgeführt worden und wird hier als $sch_1$ und $sch_2$ bezeichnet. Die Allocationsfunktionen werden mit $bau_1$ und $bau_2$ bezeichnet. Sind diese Schritte der ‚Standartsynthese' durchgeführt, so werden die Bauteilmengen zusammengefaßt. Es muß eine Vereinigung der beiden Bauteilmengen stattfinden. Sie muß speziell definiert werden, da die Bauteile anhand ihrer Namen identifiziert werden.

### 11.2.2 Vereinigung der Bauteilmengen

Der entscheidende Schritt bei der Synthese von ALUs ist die modifizierte Vereinigung der beiden benutzten Bauteilmenge. Die erzeugt Bauteilmenge enthält die maximale Anzahl der Instanzen gleicher Typen der beiden Bauteilmengen.

*Definition 11.2-1 Die modifizierte Vereinigungsmenge BA zweier Bauteilmengen $BA_1$ und $BA_2$*

*wird durch den folgenden Algorithmus erzeugt, wobei die Umbenennung der Bauteile aus* $BA_2$ *mit Hilfe der Identifizierungsfunktion* $id{:}BA_2 \rightarrow BA$ *stattfindet:*

*begin*

$BA = \{\ \}$

*for all* $ba_1 \in BA_1$

$BA = BA \cup \{ba_1\}$

*if* $\exists(ba_2 \in BA_2)(bau_1(ba_1) = bau_2(ba_2))$

*then (Dieser Schritt wird für genau ein solches* $ba_2$ *durchgeführt)*

$id(ba_2) = ba_1$

$BA_2 = BA_2 / \{ba_2\}$

*end then*

*end if*

*end for*

$BA = BA \cup BA_2$

*end*

Bei dieser Definition der modifizierten Vereinigungsmenge wird eine neue Bauteilmenge aus zwei vorhandenen gebildet, indem für jeden vorkommenden Bauteiltyp nur die in den beiden vorgegebenen Mengen maximal vorkommende Anzahl Instanzen allociert wird. Jedes Bauteil aus $BA_2$ welches einem Bauteil in der Menge $BA_1$ vom Typ her entspricht, wird aus der Menge $BA_2$ gestrichen, wobei es aber mit dem entsprechenden Bauteil der Menge *BA* identifiziert werden muß. Falls kein entsprechendes Bauteil in $BA_1$ vorkommt, wird es der Bauteilmenge *BA* hinzugefügt. In Bild 110 ist ein Beispiel für zwei allocierte Mengen gezeigt. Hieraus wird dann

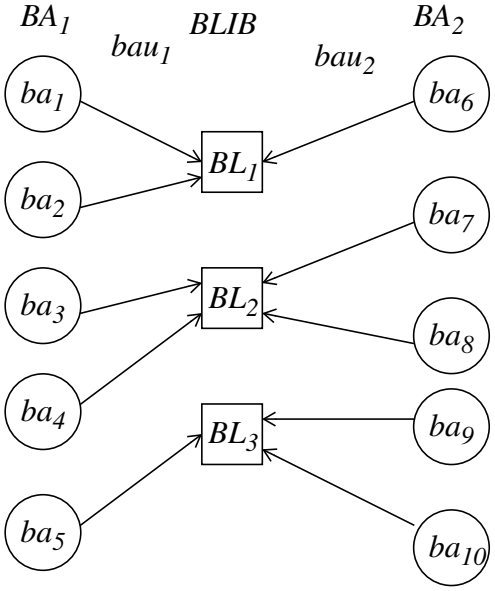



Bild 110: Zwei allocierte Mengen

die Allocation, die in Bild 111 dargestellt ist, mit der oben dargestellten Methode erzeugt. Es

werden die Bauteile $ba_1$ bis $ba_5$ in die Menge *BA* übernommen und für jedes Bauteil der Menge $BA_2$ wird die oben dargestellte Bedingung überprüft. Dadurch, daß es schon entsprechende Bauteile gibt, fallen die Bauteile $ba_6$ bis $ba_9$ weg, da sie mit den Bauteilen $ba_1$ und $ba_3$ ... $ba_5$ identifiziert werden können. Es wird nur $ba_{10}$ in die resultierende Menge *BA* übernommen.

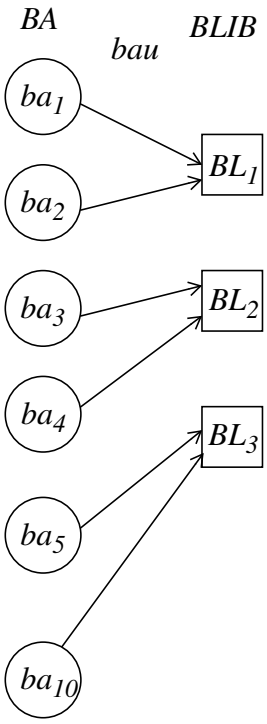



Bild 111: Durch Vereinigung erzeugt Allocation

### 11.2.3 Zusammenfassung der Assignments

Sind die beiden allocierten Mengen, wie oben dargestellt, zu einer zusammengefaßt, so müssen die Assignments im nächsten Schritt vereinigt werden. Hierbei ist zu beachten, daß einige Bauteile nicht mehr existieren, sondern nur noch das mit ihnen identifizierte Bauteil. Dadurch, daß nicht mehr alle Bauteile, die für $A_2$ benutzt werden, vorhanden sind, muß eine Umbenennung der Bauteile stattfinden und das Assignment $A_2$ entsprechend modifiziert werden. Die Zuordnung der Befehle zu den Bauteilen bleibt für $A_1$ erhalten und wird direkt in das Gesamtassignment *A* übernommen. Das neue Assignment wird dann folgendermaßen erzeugt.

*Definition 11.2-2 Die Abbildung* $A{:}B \rightarrow BA$ *ist die Vereinigung der Assignments* $A_1{:}B_1 \rightarrow BA_1$ *und* $A_2{:}B_2 \rightarrow BA_2$ *mit* $B_1 \cup B_2 = B$ *und ist folgendermaßen definiert:*

*1)* $A(b) = A_1(b)$ *falls gilt* $b \in B_1$

*2)* $A(b) = A_2(b)$ *falls gilt* $b \in B_2$ *und* $A_2(b) \in BA$

*3)* $A(b) = id(A_2(b))$ *sonst.*

Durch die Definition ist eine Vereinigung der Assignments gegeben, welche auf der Vereinigung der Bauteilmengen basiert. Konsequent zu der in Def. 11.2-1 gemachten Berechnung der Vereinigungsmenge wird hier in Punkt 3) das Assignment mit identifizierten Bauteilen durch-

geführt. Außerdem werden die Befehlsmengen ‚ganz normal' in einer Befehlsmenge zusammengefaßt. Schon in Bild 108 ist dargestellt, wie ein zusammengefaßtes Assignment aussehen kann.

### 11.2.4 Gemeinsames Scheduling

Im nächsten Schritt wird ein gemeinsames Scheduling *sch* aus $sch_1$ und $sch_2$ berechnet.

*Definition 11.2-3 Die Vereinigung sch der Schedulings $sch_1$ und $sch_2$ zweier Befehlsmengen $B_1$ und $B_2$ ist eine Abbildung der Form $sch: B \rightarrow N$ mit $B_1 \cup B_2 = B$ und ist folgendermaßen definiert:*

*1)* $b \in B_1 \Rightarrow sch(b) = sch_1(b)$

*2)* $b \in B_2 \Rightarrow sch(b) = sch_2(b)$

Die einmal berechneten Schedulings bleiben erhalten, und es ist keine neue Berechnung des neuen Schedulings nötig.

### 11.2.5 Register Assignment

Abschließend wird dargestellt, wie das Variablen-Register-Assignment zusammengefaßt werden kann. Da die Variablen der beiden Befehlsmengen unabhängig voneinander sind und niemals gleichzeitig benutzt werden können, ist hier eine Zusammenfassung der Variablenmengen möglich. Seien die Variablenmengen $V_1$ und $V_2$ für die dargestellten Befehlsmengen gegeben, dann gibt es nach Def. 9.10-5 für beide Mengen Partitionen mit einer minimalen Anzahl Mengen bzw. Registern, deren Anzahl durch $n_1 = minv(V_1)$ und $n_2 = minv(V_2)$ berechnet wird. Die Mengen der Partitionen können o.B.d.A durch $pa1_i$ mit $i = 1 \ldots n_1$ und $pa2_j$ mit $j = 1 \ldots n_2$ dargestellt werden. Dann kann die Vereinigungsmenge der Register-Variablen-Assignments folgendermaßen definiert werden:

*Definition 11.2-4 Die Vereinigung* $pa = pa1 \overset{\circ}{\cup} pa2$ *zweier Variablen-Register-Zuordnungen ist gegeben durch die Vereinigung der Partitionen pa1 und pa2 der Variablenmengen $V_1$ und $V_2$. Die Mengen der Partitionen werden mit $pa1_i$ und $pa2_j$ bezeichnet, wobei gilt $n_1 = minv(V_1)$, $n_2 = minv(V_2)$, $i = 1 \ldots n_1$ und $j = 1 \ldots n_2$. Für die Partition pa bestehend aus den Mengen $pa_k$ mit $k = 1 \ldots n$ und $n = max\{n_1, n_2\}$ gilt:*

*1)* $(k \leq n_1) \wedge (k \leq n_2) \Rightarrow pa_k = pa1_k \cup pa2_k$

*2)* $(k \leq n_1) \wedge (k > n_2) \Rightarrow pa_k = pa1_k$

*3)* $(k > n_1) \wedge (k \leq n_2) \Rightarrow pa_k = pa2_k$

Die Funktion *reg*, die durch die Definition Def. 9.10-5 gegeben ist, gilt auch für die Vereinigung $V = V_1 \cup V_2$ der entsprechenden Variablenmengen.

Der durch die Partitionierung eines zu synthetisierenden Modells gegebene Nachteil, daß jedes Teilmodell unabhängig voneinander synthetisiert wird und die allocierten Bauteilmengen ebenfalls unabhängig voneinander betrachtet werden - somit auch keine gemeinsame Nutzung der

Bauteile ermöglicht wird - wird durch das in diesem Abschnitt vorgestellte System ausgeglichen. Der Vorteil einer Synthesebeschleunigung durch die Partitionierung bleibt aufgrund der Möglichkeit der gemeinsamen Nutzung der Bauelemente erhalten. Es entsteht also ein ressourcensparendes System, wobei der errechnete Zeitbedarf der Ergebnisse erhalten bleibt.

# 12 Implementierung

Die Implementierung eines experimentellen Systems wurde objektorientiert in C++ durchgeführt. Die wichtigsten Klassen werden im folgenden dargestellt. Da das bisher Gesagte unabhängig von einer speziellen Programmiersprache ist, dient die Implementierung nur der Verifikation. Bei der Implementierung wurde darauf geachtet, daß die Hauptklassen und Module unabhängig voneinander sind. Dies ist im besonderen deshalb wichtig, weil das ganze System in einem lokalen Workstationcluster verteilt arbeiten soll. Praktisch wurde die Parallelisierung der Synthese mit dem System LSF (Load Sharing Facility) [49][102] durchgeführt. Die einzelnen Module wurden hierfür angepaßt und optimiert. Im folgenden werden die einzelnen Klassen kurz vorgestellt.

## 12.1 Befehl und Programm

Die Klasse Befehl repräsentiert den in Def. 9.3-3 definierten elementaren Befehl. Durch entsprechende Methoden kann auf die einzelnen Elemente, also die Variablen und den Operanden, zugegriffen werden. Außerdem wird der Kontrollschritt, in den ein Befehl eingeordnet ist, ebenfalls in dieser Klasse abgelegt und kann hier abgefragt werden. Ein Befehl kann daraufhin abgefragt werden, ob er einen ersten oder letzten Befehl einer Schleife darstellt. Mehrere Befehle werden in der Klasse Befehlsliste zu einem Programm zusammengefaßt. Die Klasse Befehlsliste enthält Methoden, um die Abhängigkeiten der Befehle untereinander abzufragen: also die Datenabhängigkeit, Antidatenabhängigkeit und Ausgabeabhängigkeit. Außerdem wird in dieser Klasse eine Methode implementiert, die die transitive Hülle der Abhängigkeiten berechnet.

## 12.2 Bauteil und Bauteilbibliothek

Die Klasse Bauteil repräsentiert das in der Def. 9.6-1 dargestellte Bauteil und stellt Methoden zur Verfügung, um die definierten Zugriffe zu realisieren, also den Zugriff auf den Zeit- und Platzbedarf und die zur Verfügung stehenden Operationen. Außerdem werden Methoden zur Verfügung gestellt, die einen Test erlauben, ob ein bestimmter Befehl diesem Bauteil zugewiesen werden kann. Die Klasse Bauteil-Liste wird als Bauteil-Bibliothek benutzt.

## 12.3 Allocation, Allocation_Code und Allocation_Generation

Die Klasse Allocation stellt eine allocierte Bauteilmenge dar. Diese Klasse wird von der Bauteil-Liste abgeleitet. Diese Klasse stellt die zur Nutzung fertige Allocation dar. Zur Optimierung im Rahmen des genetischen Algorithmus steht die Klasse Allocation_Code zur Verfügung, die die Codierung der Allocation bildet. Hier werden Bewertungsfunktionen definiert, unter anderem auch die Abfrage, ob die codierte Bauteilmenge gültig bzgl. einer Menge von Befehlen ist. Um die allocierte Menge geliefert zu bekommen, gibt es eine Funktion, die die Dekodierung vornimmt und eine Bauteilliste liefert. Die Klasse Allocation_Generation verwaltet eine Generation Allocationen bzw. deren Codes. Eine Sortierfunktion sortiert anhand gegebener Bewertungen die einzelnen Allocationen. Außerdem werden Mutations- und Crossoveroperatoren für die Implementierung des genetischen Algorithmus zur Verfügung gestellt.

## 12.4 Schleife und Schleifen-Liste

Die Klasse Schleife stellt genau eine Schleife dar. In dieser Klasse wird die in Satz 9.4-2 gege-

bene Definition einer Schleife umgesetzt, und es werden Methoden für den Zugriff auf die Elemente der Schleife zur Verfügung gestellt. Eine Methode, um die Schleifenabhängigkeit zwischen zwei Befehlen zu berechnen, ist - wie in Satz 9.4-3 gegeben - ebenfalls in dieser Klasse implementiert. Der Algorithmus zur Schleifenoptimierung wird ebenfalls durch die Schleifen-Klasse zur Verfügung gestellt. In der Klasse der Schleifen-Liste werden alle vorhandenen Schleifen in eine Liste eingetragen, wobei die Abhängigkeiten, die durch alle Schleifen zustandekommen, hier abgefragt werden können.

### 12.5 Assignment und Assignment-Liste

Die Klasse Assignment ist abgeleitet von der Bauteil Klasse. Sie stellt genau einen Bauteil dar mit den Befehlen, die diesem Bauteil zugeordnet sind. Um die Menge der Befehle zu verwalten, die einem Bauteil zugeordnet sind, ist die Assignment Klasse außerdem von einer Mengen Klasse abgeleitet, die Mengenoperatoren und Operatoren auf Permutationen zur Verfügung stellt. Eine wie in Def. 9.8-8 definierte Permutation für ein Bauteil wird also mit der Klasse Assignment dargestellt. Die Assignment-Liste ist dann die Klasse, die alle Bauteile und deren Zuordnungen zusammenfaßt. Das heißt, das im Text als Assignment bezeichnete Objekt wird in der Implementierung als Assignment-Liste bezeichnet. Der Test der Assignments auf Gültigkeit wird - wie in Satz 9.8-12 dargestellt - von dieser Klasse durchgeführt.

### 12.6 Lösung

Ein Assignment ist in der Klasse Lösung implementiert. Diese Klasse ist von der Assignment-Liste abgeleitet und stellt alle Methoden zur Verfügung, die die Manipulation eines Assignments betreffen, also die in Def. 10.15-1 und Def. 10.17-1 dargestellten Mutationsfunktionen und die Crossoverfunktion aus Def. 10.16-1. Außerdem werden hier die Bewertungsfunktionen der Lösung zur Verfügung gestellt. Das heißt, hier ist beispielsweise auch das Scheduling implementiert. Um eine Optimierung bei der Mutation zu erreichen, wird der längste Weg durch den als Graphen dargestellten Datenfluß - oder den kritischen Pfad - markiert. Mutationen werden in erster Linie auf den so markierten Befehlen durchgeführt. Somit ist hier auch eine Heuristik implementiert. Die Klasse Lösung kann, wegen ihrer Wichtigkeit, als die zentrale Klasse des Synthesesystems verstanden werden.

### 12.7 Variable und Variablen Liste

Eine Liste aller vorkommender Variablen wird von der Klasse Variablen_Liste verwaltet. Eine Lösung wird als initialer Wert übergeben und daraus werden zum einen die vorkommenden Variablen extrahiert und zum anderen wird der Left-Edge Algorithmus ausgeführt. Die einzelnen Variablen sind Instanzen der Klasse Varible, hier werden Parameter, wie der Typ der Variablen verwaltet. Außerdem werden Lebenszeitintervalle hier verwaltet, so daß einem Optimierungsalgorithmus alle benötigten Daten zur Verfügung stehen. Im wesentlichen werden die in Abschnitt 9.10 dargestellten Eigenschaften und Methoden in diesen Klassen implementiert.

### 12.8 Eine Generation Lösungen

Eine Generation an Lösungen wird in der Klasse Generation implementiert. Hier werden die Lösungen anhand ihrer Bewertung sortiert und eine neue Generation Lösungen erzeugt. Der Crossoveroperator steht in dieser Klasse zur Verfügung.

### 12.9 Netzliste, Kontroller und Gesamt_Netz

Die Klasse Netzliste erzeugt aus einer Lösung eine Netzliste, wobei der Kontroller mit Hilfe der Klasse Kontroller erzeugt werden kann. Durch Austausch dieser Klassen können andere Archi-

tekturen mit dem System erzeugt werden. Die Klasse Gesamt_Netz erzeugt die Verbindungsstruktur zwischen dem Kontroller und der Netzliste.

### 12.10 Die allgemein verwendbaren Klassen

Die Struktur des implementierten Experimentalsystems ist in Bild 112 dargestellt. Hier sind nur die Zusammenhänge der Klassen dargestellt, die auch in den Definitionen auftauchen. Die Klassen, die für die Umsetzung des genetischen Algorithmus zuständig sind, werden in der Abbildung nicht dargestellt. Das unten dargestellte System ist unabhängig von den Optimierungsalgorithmen, kann also als Grundlage für die weitere Forschung im Bereich der Optimierungsalgorithmen in der Synthese dienen.

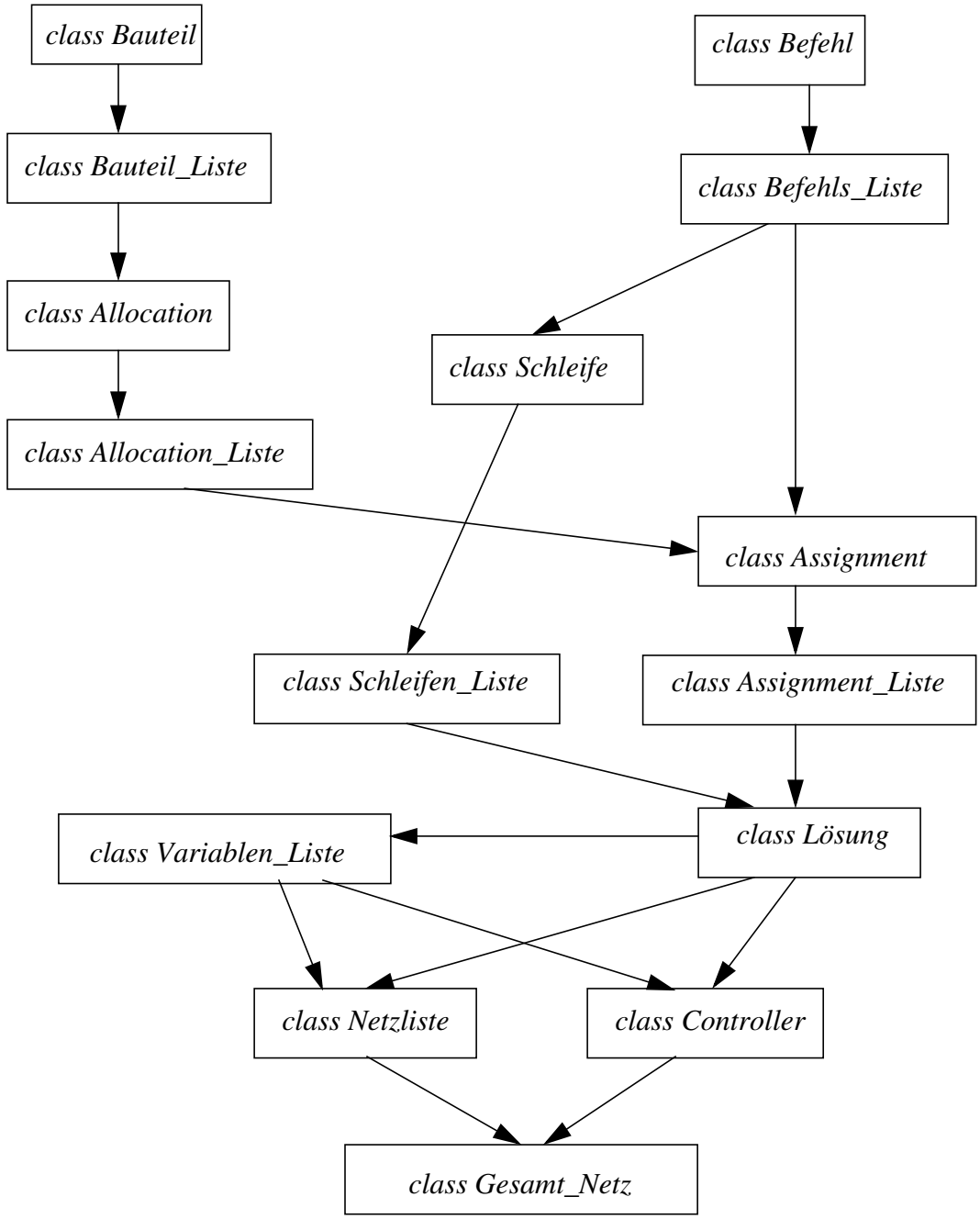



Bild 112: Der Zusammenhang der wesentlichen implementierten Klassen

### 12.11 Spezielle Klassen des genetischen Algorithmus

Der genetische Algorithmus wurde oben schon ausführlich dargelegt. In der Bild 113 ist der Zusammenhang der Klassen dargestellt, die für die Implementierung des genetischen Algorithmus für die Allocation zuständig sind.

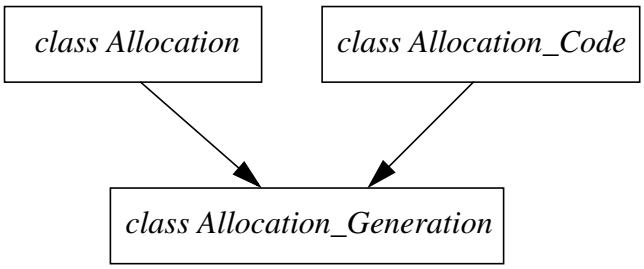



Bild 113: Die Klassen für die Allocation

Die Bild 114 stellt die Zusammenhänge der Klassen dar, die für die Implementierung des genetischen Algorithmus für das Assignment genutzt werden.

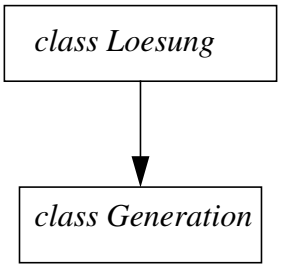



Bild 114: Die Klassen für das Assignment

In den Abbildungen - Bild 112 , Bild 113 und Bild 114 - wurden die Zusammenhänge der wesentlichen Klassen dargestellt. Unter Nutzung der dargestellten Klassenhierarchie wurden drei Prozesse implementiert, die zusammenarbeiten.

### 12.12 Der Gesamtprozeß

Der Gesamtprozeß besteht aus drei Prozessen. Der Hauptprozeß des genetischen Algorithmus für die Allocation ist die Implementierung des in Bild 89 dargestellten genetischen Algorithmus. Die Bewertung der einzelnen Bauteilmengen kann mit den in “Bewertung” auf Seite 97 definierten Funktionen geschehen, oder es wird der genetische Algorithmus für das Assignment zur Bewertung der Allocation benutzt. Der genetische Algorithmus für das Assignment ist seinerseits wieder aufgeteilt in einen Hauptprozeß und den Bewertungsprozeß. Bild 115 zeigt den Zusammenhang der drei Prozesse. Ganz rechts ist der Prozeß der Allocation dargestellt. Hier werden verschiedene Allocationen erzeugt. Mit diesen unterschiedlichen Allocationen wird der Assignmentprozeß aufgerufen, der verschiedene Assignments erzeugt, die dann mit dem ganz links dargestellten Prozeß bewertet werden. Für jede Allocation wird der Assignmentprozeß mehrmals durchlaufen. Die Bewertung des besten Ergebnisses wird dann zur Bewertung der Allocation benutzt.

Durch die Dreiteilung des Gesamtprozesses ist eine Parallelisierung möglich. Der Allocationsprozeß erzeugt mehrere Assignmentprozesse, die wiederum mehrere Bewertungsprozesse nach sich ziehen. Diese Prozesse können in einem Workstationcluster verteilt ausgeführt werden.

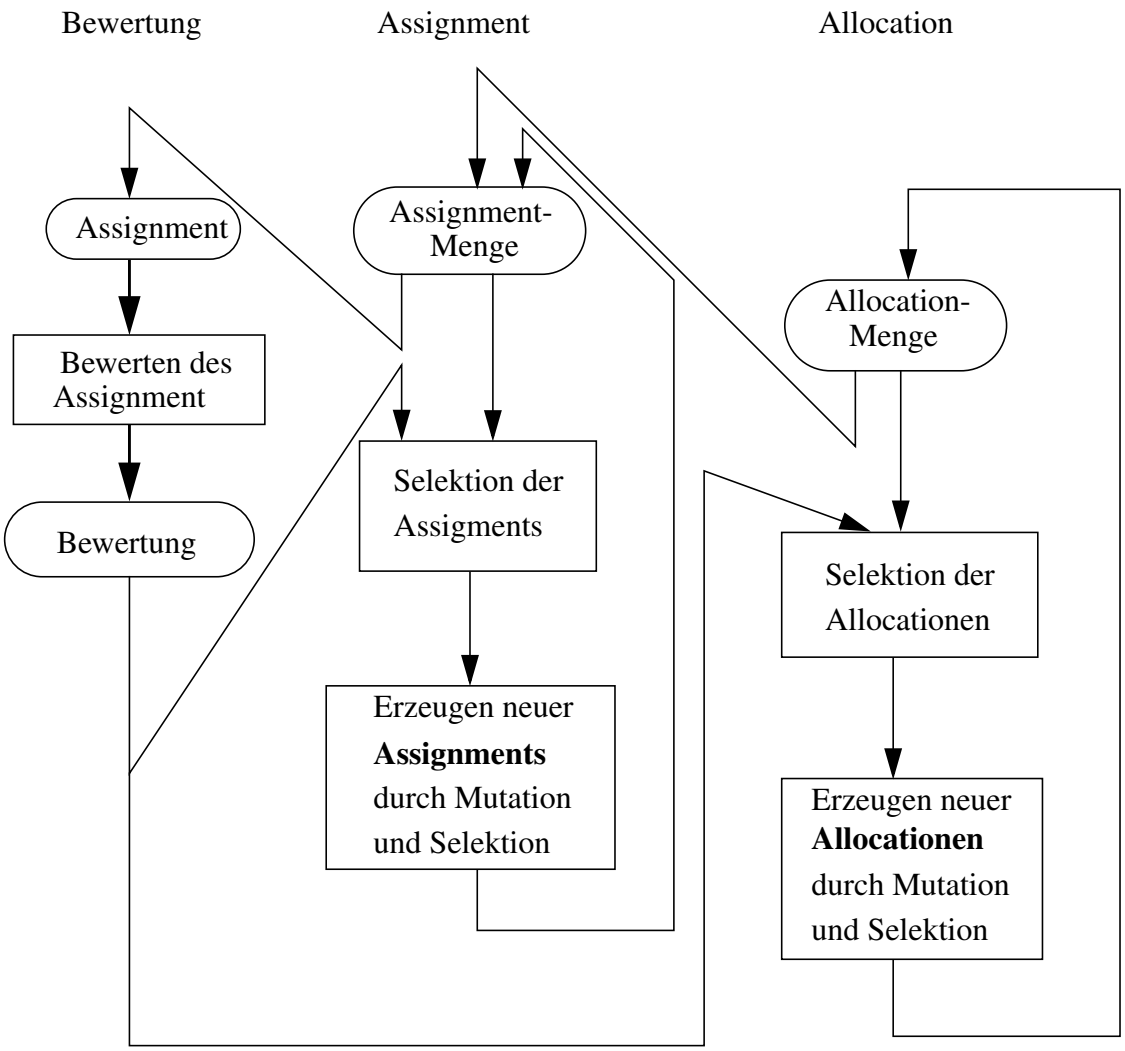



Bild 115: Gesamtprozeß

### 12.13 Parallelisierung durch Nutzung eines Workstationclusters

Da hier verschiedene Workstationclusterarchitekturen nicht betrachtet werden sollen, sei auf die entsprechende Literatur verwiesen [107]. Anhand der in [59] dargestellten Workstationclusterarchitektur soll hier gezeigt werden, wie eine Verteilung der Prozesse aussehen kann. Eine feinere Unterteilung der Prozesse in Unterprozesse, als die hier gewählte, hat sich als nicht sinnvoll erwiesen, da der Anteil der Kommunikation in einem Workstationcluster den Zeitgewinn, welcher durch stärkere Parallelisierung erreicht wird, wieder zunichte macht.

*Definition 12.13-1 Der Hauptprozeß der Allocation sei mit ALP bezeichnet.*

Mit *ALP* ist also der ganz rechts in Bild 115 dargestellte Teil bezeichnet. Der Allocationsprozeß erzeugt mehrere Assignmentprozesse.

*Definition 12.13-2 Die von ALP erzeugten n Assignmentprozesse werden mit $ASP_1$, ..., $ASP_n$ bezeichnet.*

Jeder Assignmentprozeß $ASP_i$ erzeugt wiederum mehrere Bewertungsprozesse.

*Definition 12.13-3 Die von einem Assignmentprozeß $ASP_i$ erzeugten m Bewertungsprozesse werden mit $BP_{i,1}$ , ... , $BP_{i,m}$ bezeichnet.*

In der Bild 116 ist die in [59] dargestellte Architektur gezeigt, die sich dadurch auszeichnet, daß der Komunikationsengpaß, der entsteht, wenn Workstations über einen Kommunikationskanal verbunden sind, dadurch beseitigt wird, daß mehrere Kommunikationskanäle genutzt werden.

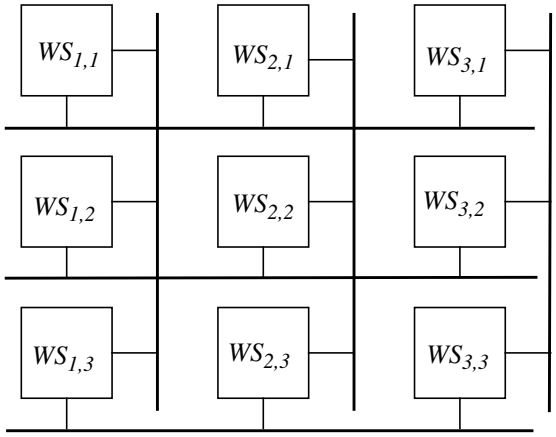



Bild 116: Workstationclusterarchitektur

*Definition 12.13-4 Eine zweidimensionales Workstationcluster - mit ConCAR bezeichnet - besteht aus den Workstations $WS_{i,j}$ mit $i = 1 \ldots o$ und $j = 1 \ldots p$, wobei o die Anzahl der Workstations in horizontaler und p die Anzahl der Workstations in vertikaler Richtung angibt und alle Workstations $WS_{i,j}$ und $WS_{k,l}$, für die gilt $(i = k) \vee (j = l)$, an dem gleichen Kommunikationskanal angeschlossen sind.*

*Satz 12.13-5 Ein Workstationcluster aus $o \times p$ Workstations besitzt $o + p$ Kommunikationskanäle.*

Die Verteilung von Prozessen auf Workstations kann folgendermaßen definiert werden.

*Definition 12.13-6 Die Verteilung von Prozessen des Gesamtsystems auf Workstations ist durch die Abbildung*

$$\nu e: \{ALP, ASP_1, \ldots, ASP_n, BP_1, \ldots, BP_n\} \rightarrow \{WS_{i,j} \mid (i = 1 \ldots o), (j = 1 \ldots p)\}$$

*gegeben.*

In der Bild 117 ist eine mögliche Verteilung der Prozesse auf die Workstations dargestellt, die einen sehr geringen Kommunikationsaufwand erfordert. Der Prozeß *ALP* braucht nur mit den *ASP* Prozessen zu kommunizieren, was über den fett dargestellten Kommunikationskanal geschieht. Die *ASP* Prozesse kommunizieren in erster Linie mit den zu ihnen gehörenden *BP* Prozessen.

Durch eine Verteilung der Prozesse, wie in der Bild 117 dargestellt, werden die Vorzüge, die die in Bild 116 dargestellte Architektur bietet, voll genutzt. Der Kommunikationsengpaß, der

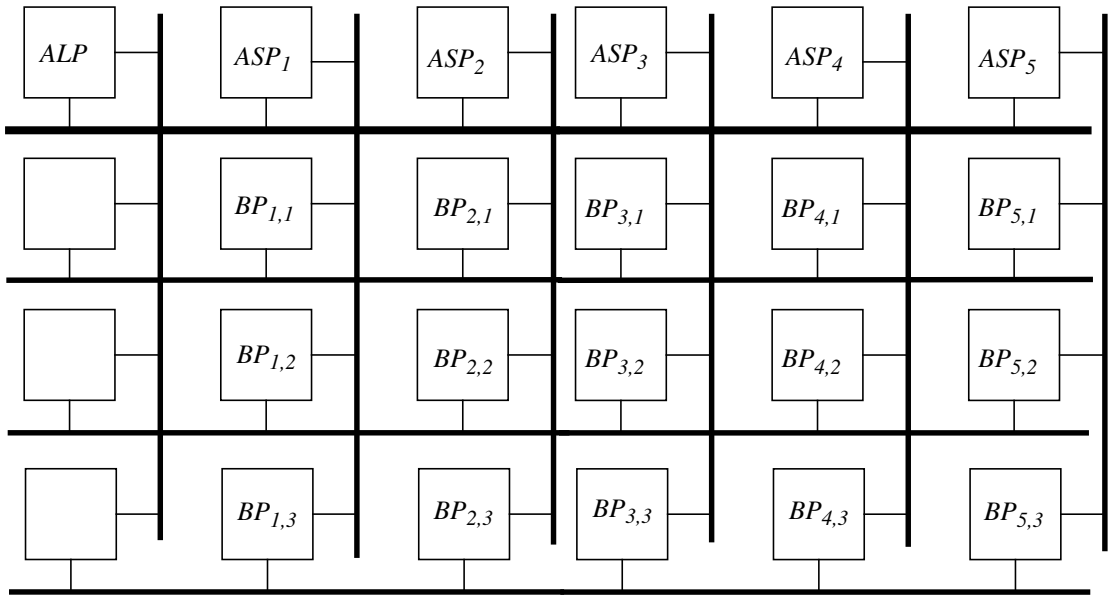



Bild 117: Verteilung der Prozesse auf ein Workstationcluster

durch die Nutzung nur eines Kommunikationskanals entsteht, ist hier umgangen. Durch die günstige Verteilung der Prozesse auf die Workstations ist es nicht nötig, ein zeitaufwendiges Routing durchzuführen.

# 13 Auswertung und Messergebnisse

In den folgenden Abschnitten werden die in dieser Arbeit vorgestellten Methoden anhand von Beispielen verglichen und untermauert. Zum einen wird anhand eines kleineren Beispiels, welches ausführlich diskutiert wird, dargestellt, welcher Gewinn durch verschiedene grundsätzliche Methoden erzielt werden kann. Hier ist im wesentlich die im Rahmen der Arbeit entwickelte Methode zur Darstellung der Bauteile zu nennen. Des weiteren wird das ebenso bekannte wie praxisnahe Beispiel des Elliptical Wave Filters dargestellt. Die selbstgewählten Beispiele sollen zeigen, daß der Laufzeitgewinn welcher durch die Parallelisierung der genetischen Algorithmen erzielt wird, wesentlich von der Größe der Beispiele abhängig ist.

## 13.1 Das HAL-Beispiel

Ein typisches Beispiel, welches immer wieder in der Literatur anzutreffen ist und an dem verschiedene Syntheseverfahren verglichen werden können, ist der lineare Differenzierer, welches auch als HAL-Beispiel bekannt ist. Anhand dieses Beispiels sollen die Einflüsse verschiedener Methoden auf das Syntheseergebnis dargestellt werden. Der wesentliche Punkt bei diesem Vergleich ist der Einfluß der verschiedenen zugelassenen Module auf das Ergebnis. Im ersten Fall werden nur Module betrachtet, für die der Zeitbedarf der einzelnen Operationen nicht durch das Synthesetool berücksichtigt wird. Im nächsten Fall wird dann davon ausgegangen, daß ein unterschiedlicher Zeitbedarf der Befehle erlaubt ist. Im dritten Fall wird zusätzlich der Einsatz von Pipelinebausteinen durch das Synthesetool erlaubt, was dazu führt, daß eine höhere Geschwindigkeit des synthetisierten Bauteils erreicht wird. Anhand dieses relativ kleinen Beispiels lassen sich keine weiteren Aussagen über die Güte der verwendeten parallelen genetischen Algorithmen machen, da der Zeitverlust durch den erhöhten Kommunikationsaufwand im Netzwerk den Gewinn, der durch die parallele Bewertung gegeben ist, ausgleicht.

### 13.1.1 Vergleich verschiedener Syntheseansätze

In Bild 118 ist das HAL-Beispiel bzw. der lineare Differenzierer als Datenflußgraph dargestellt. Bei einfachen Synthesesystemen werden nur Bauteile mit genau einer Funktion unterstützt. Eine weitere Einschränkung ist dadurch gegeben, daß davon ausgegangen wird, daß der Zeitbedarf jeder Funktion äquivalent ist und somit jede Funktion in einem Taktschritt abgearbeitet werden kann.

```
while (x<a) do
   x1 := x + dx
   u1 := u - (3*x*u*dx) - (3*y*dx)
   y1 := y + (u * dx)
   x := x1
   u := u1
   y := y1
endwhile
```

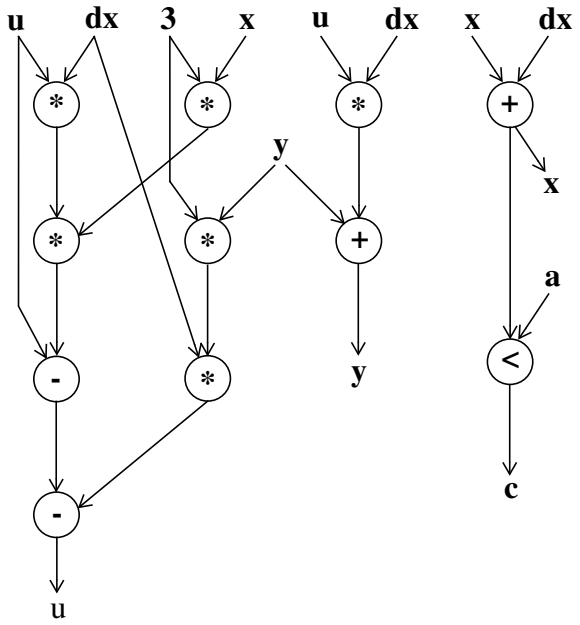



Bild 118: Das HAL-Beispiel

In Bild 119 wird das Ergebnis des ASAP-Scheduling dargestellt. Es werden lediglich vier Taktperioden benötigt, wenn eine entsprechende Anzahl Bauteile zur Verfügung stehen. Zu einem späteren Zeitpunkt des Entwurfs wird die Länge einer Taktperiode so gewählt, daß das langsamste Bauteil seine Operation innerhalb einer Taktperiode ausführen kann.

Der Nachteil dieser einfachen Methode ist, daß schnelle Bauteile oft brach liegen. Außerdem können Bauteile, die mehrere Funktionen unterstützen und damit einen Flächengewinn bringen, nicht verwendet werden. Ein Schritt, um schnelle Bauteile besser auszunutzen, besteht darin, jedem Kontrollschritt nur die benötigte Anzahl von Taktschritten zuzuweisen. Eine Taktperiode ist dann an das schnellste Bauteil angepaßt, und jedem Kontrollschritt werden soviele Taktschritte wie nötig zugewiesen. Werden in einem Kontrollschritt nur schnelle Bauteile benutzt, so kann ihre volle Geschwindigkeit ausgenutzt werden. Schnelle Bauteile liegen oft immer noch brach, wenn in dem gleichen Kontrollschritt, in dem sie benutzt werden, auch langsame Bauteile aktiv sind. Unter der Annahme, daß auch Bauteile mit mehreren Operationen, also ALUs, erlaubt sind und unterstützt werden, ergibt sich bei der Allocation der in Tabelle 4 dargestellten Bauteile das folgende Ergebnis: Im ersten Fall werden für die Länge einer Taktperiode 60 ns

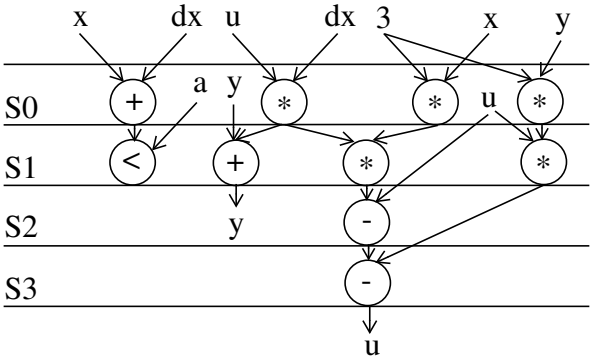



Bild 119: ASAP Scheduling

gewählt, was zu einer Gesamtverzögerungszeit von 240 ns führt. Im zweiten Fall werden 20 ns für die Länge einer Taktperiode festgelegt. Die Kontrollschritte *S0* und *S1* benötigen für eine korrekte Ausführung aller Operationen drei Taktperioden, wobei die Kontrollschritte *S2* und *S3* mit jeweils einer Taktperiode auskommen. Hieraus ergibt sich ein Gesamtzeitbedarf von 160 ns.

**Tabelle 4: Bauteile**

| ALU | A1 | A2 | A3 | A4 | A5 |
|---|---|---|---|---|---|
| Function | +,- | * | * | * | < |
| Delay in ns | 20 | 40 | 60 | 60 | 20 |

Werden bei der Berechnung der minimal benötigten Bauteile nur die Kontrollschritte in Betracht gezogen, nicht aber die real benötigten Taktschritte, so wird in diesem Beispiel die Anzahl der zur Verfügung gestellten Multiplizierer auf zwei reduziert. Die Anzahl der benötigten Kontrollschritte bleibt bei vier, und im ersten Fall ist ein echter Flächengewinn vorhanden. Im zweiten Fall muß die Anzahl der benötigten Taktschritte für den Kontrollschritt *S2* auf zwei erhöht werden, da hier die letzte Multiplikation ausgeführt wird. Der Zeitbedarf erhöht sich auf 180 ns. Bild 120 zeigt die Verteilung der Operationen auf die Kontrollschritte.

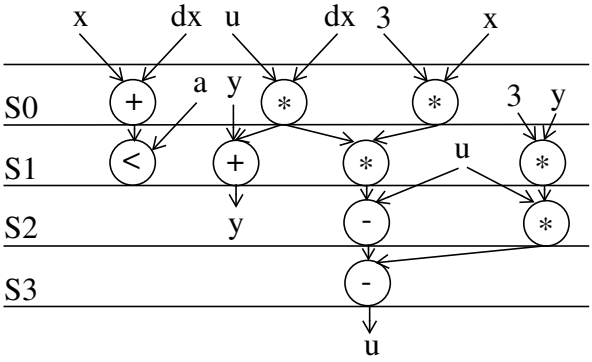



Bild 120: Nur zwei Multiplizierer werden verwendet

Solange das Scheduling unabhängig von realen Verzögerungszeiten der Bauteile durchgeführt wird, gibt es immer Bauteile, die die meiste Zeit brach liegen. Im folgenden wird gezeigt, daß

eine direkte Betrachtung der benötigten Taktschritte schon zur Schedulingzeit einen Zeit- und Platzgewinn bringen kann. Hierzu ist es wichtig, daß das Assignment, also die konkrete Zuordnung der Befehle zu den Bauteilen, vor dem Scheduling stattfindet. Die Verzögerungszeit eines Taktschrittes wird an das schnellste vorhandene Bauteil angepaßt, also beträgt in dem Beispiel eine Taktperiode 20 ns. In Bild 121 ist das Ergebnis dargestellt, wenn die Operationen direkt in Taktschritte eingeordnet werden, wobei auch hier nur zwei Multiplizierer benutzt wurden. Es werden lediglich acht Taktschritte benötigt, und die Verzögerungszeit beträgt 160 ns. Dieses Ergebnis wurde bisher nur mit einem erhöhten Hardwarebedarf erreicht.

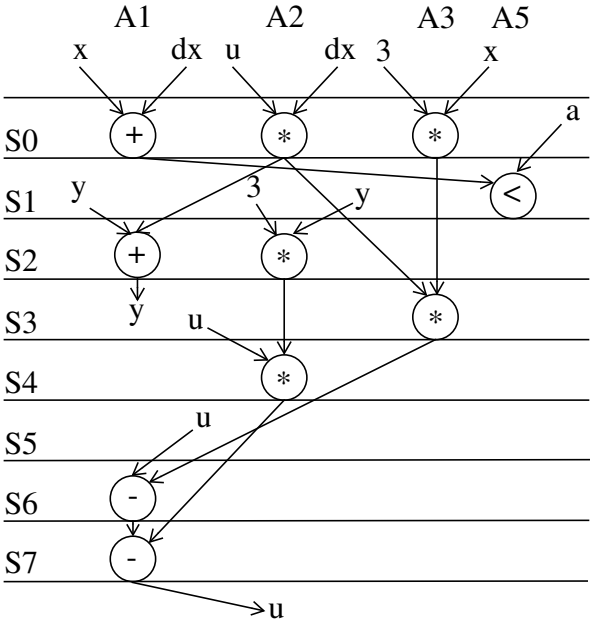



Bild 121: Betrachtung der Taktschritte

Angemerkt sei hier, daß eine Erhöhung der Multipliziereranzahl auf drei, wie im ursprünglichen Fall, zu einem Zeitbedarf von 140 ns führt. Eine weitere Optimierung liegt in der Unterstützung von Pipelinebauteilen, die im Rahmen dieser Arbeit ebenfalls betrachtet wird. Der 40 ns Multiplizierer soll in dem Beispiel durch eine Pipelineeigenschaft erweitert werden, und zwar kann schon nach 20 ns die nächste Multiplikation gestartet werden. Das Ergebnis ist in Bild 122 dargestellt. Der Übersichtlichkeit halber wurden die zu einem Bauteil gehörenden Operationen in dieser Abbildung nicht untereinander gezeichnet. Durch die Nutzung der Pipeline-Eigenschaft des Multiplizierers konnte eine Taktperiode gespart werden, so daß sich der Gesamtzeitbedarf für die Berechnung auf 140 ns reduziert hat.

### 13.1.2 Variablen Register Assignment

Der nächste Schritt der Synthese besteht in der Zuordnung der Variablen an Register. Hierbei ist es wichtig zu wissen, wie lange die Werte an den Eingängen des Bauteils anliegen müssen. Normalerweise wird davon ausgegangen, daß die Eingänge solange belegt sind, bis das berechnete Ergebnis in ein Register geschrieben wird. Unter der Annahme, daß die Register immer in der Mitte einer Taktperiode geschrieben werden, können die in Bild 123 dargestellten Lebenszeiten für das betrachtete HAL-Beispiel berechnet werden. Hierbei wird angenommen, daß die Ergebnisse bereits eine halbe Taktperiode, bevor sie benutzt werden, existieren.

Ein einfacher left-edge Algorithmus kann benutzt werden, um Variablen zusammenzufassen, was gleichbedeutend ist, daß sie in einem Register abgelegt werden. In dem dargestellten Fall

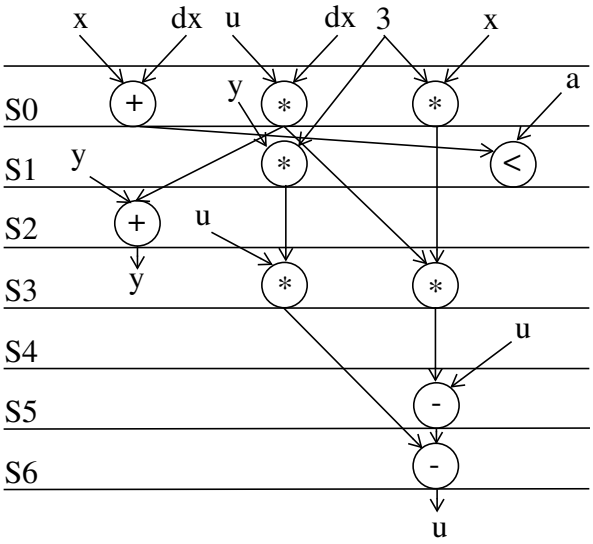



Bild 122: Pipeline-Multiplizierer

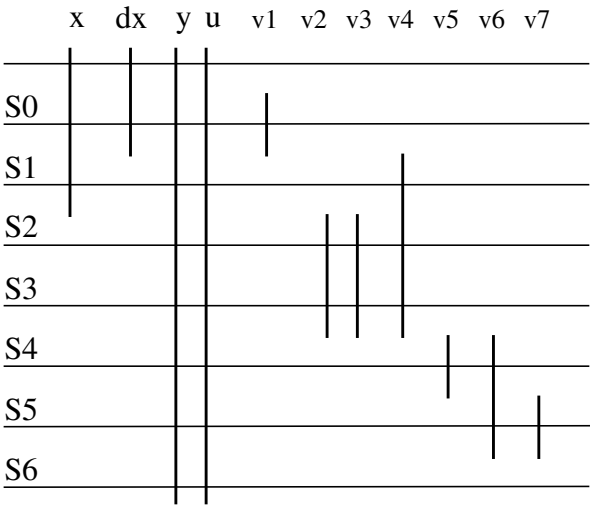



Bild 123: Lebenszeitintervalle der Variablen

werden fünf Register für die Speicherung der Variablen benötigt. Bei vielen Bauteilen ist es aber nun so, daß die relevanten Werte an den Eingänge nicht bis zur endgültigen Berechnung des Ergebnisses anliegen müssen. Ist der für die Eingänge eines Bauteils relevante Zeitbedarf bekannt, so ergeben sich andere Lebenszeitintervalle der Variablen, was dazu führen kann, daß Register zur Speicherung der Variablen eingespart werden können. Unter der Annahme, daß die Multiplikationsbausteine so beschaffen seien, daß nur über den Zeitraum einer Taktperiode die relevanten Werte an den Eingängen anliegen müssen, ergibt sich die in Bild 124 dargestellte Situation.

Die Anwendung des left-edge Algorithmus führt zu dem Ergebnis, daß nur vier Register zur Darstellung der Variablen ausreichen. Hierdurch ist ersichtlich, daß eine differenzierte Betrachtung der Bauteile, wie in dieser Arbeit geschehen, zu verbesserten Ergebnissen in der Synthese führt.

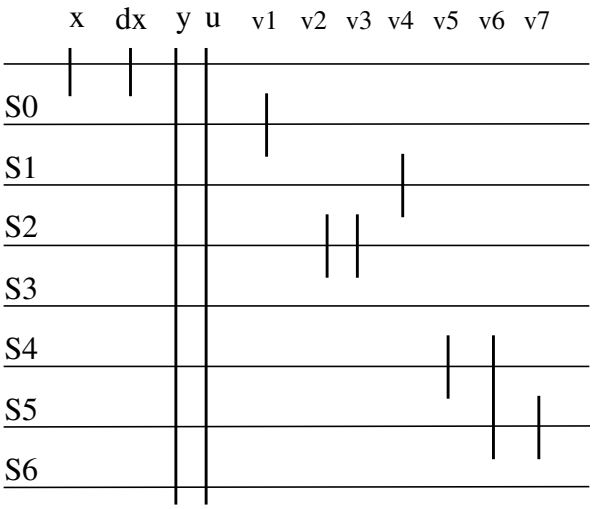



Bild 124: Verkürzte Intervalle

### 13.2 Weitere Messergebnisse

Im folgenden werden einige Meßergebnisse dargestellt. Im Fall der Optimierung von Syntheseergebnissen können - wie oben schon dargestellt - mehrere Ziele verfolgt werden, bzgl. denen eine Optimierung stattfinden soll. Wir betrachten hier zum einen den Zeitbedarf der synthetisierten Schaltung und andererseits den Platzbedarf, der sich seinerseits aus den dargestellten Faktoren Platzbedarf der Bauteile, der Register und der Verdrahtung zusammensetzt. Es gibt einen minimalen Zeitbedarf, der z.B. durch einen ASAP Algorithmus berechnet werden kann, wobei jedem Befehl das schnellste Bauteil zugeordnet wird, welches diese Operation ausführt.

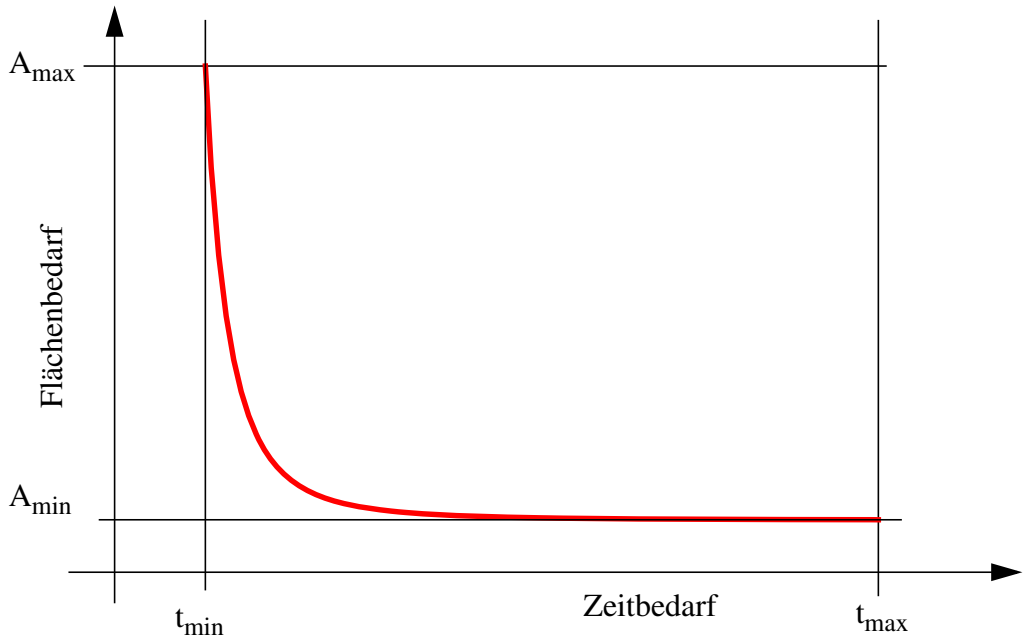



Bild 125: Typischer Zusammenhang zwischen Zeit- und Flächenbedarf

In der Bild 125 ist der typische Zusammenhang zwischen dem Zeit- und dem Platzbedarf einer Schaltung dargestellt. Es gibt einen maximalen Flächenbedarf, der dann gegeben ist, wenn jeder Befehl sein eigenes Bauteil zur Verfügung hat. Dieser maximale Platzbedarf führt dann gleich-

zeitig zu dem minimal möglichen Zeitbedarf. Andererseits gibt es einen maximalen Zeitbedarf, der gegeben ist, wenn alle Befehle sequentiell auf einem langsamen Bauteil ausgeführt werden. Das heißt also, dieser Zeitbedarf geht mit dem minimalen Platzbedarf einher. Werden alle Lösungen als Punkte in die Bild 125 eingetragen, so liegen sie rechts bzw. über der dargestellten Grenze. Ziel der Optimierung ist, eine Lösung zu finden, die möglichst auf der Linie liegt. Dabei wird, je nachdem ob eine schnelle Schaltung oder eine platzsparende Schaltung benötigt wird, der optimale Punkt näher an $t_{min}$ bzw. näher an $A_{min}$ liegen. Eine Optimierung heißt dann, daß die Lösungen so verändert werden, daß der korrespondierende Bewertungspunkt in Richtung des Ursprungs wandert. Bild 126 zeigt diesen Zusammenhang.

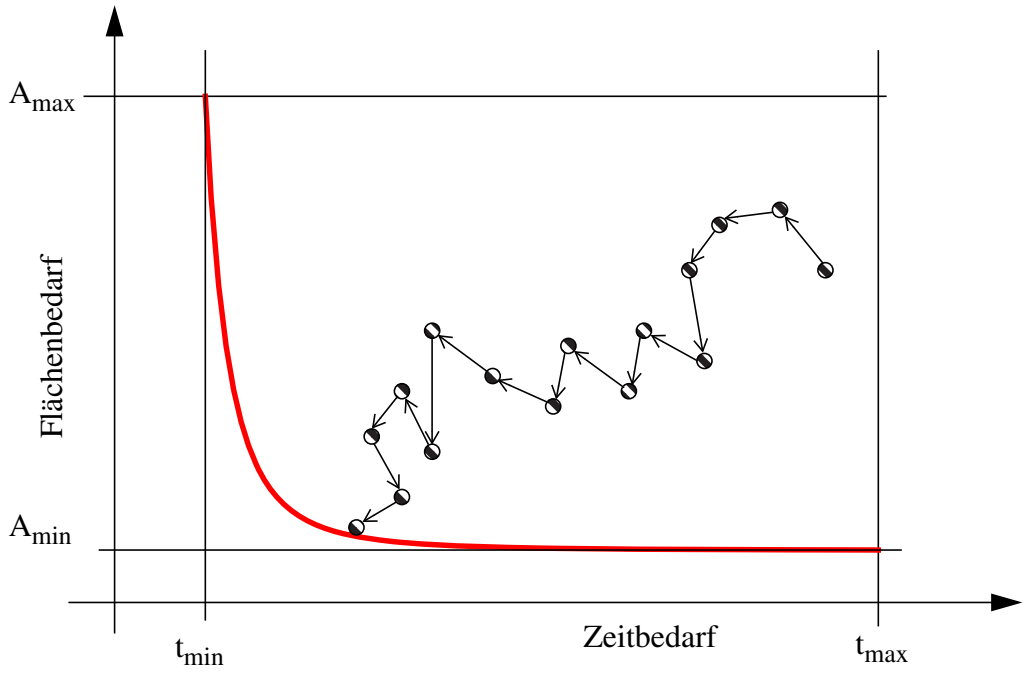



Bild 126: Optimierung heißt Suche im Lösungsraum

Meistens wird eine Optimierung mit einer zufällig erzeugten aber gültigen Lösung gestartet und dann iterativ verbessert. Für den genetischen Algorithmus wurde dargestellt, wie Veränderungen an den Lösungen durchgeführt werden können. Die große Verwandschaft der betrachteten Problemstellung zu dem QAP (Quadratic Assignment Problem), welches in [67] eingeführt wurde, und für das in [23] auf sein asymptotisches Verhalten hingewiesen wurde, läßt nach [24] die Vermutung zu, daß bei zufällig erzeugten Lösungen für große Problemstellungen die Qualität gut ist. Das heißt, unter Umständen erzeugt eine rein zufällige Suche im Lösungsraum ebenfalls gute Ergebnisse. Für einige Beispiele soll ein Ausschnitt des Lösungsraums dargestellt werden. Die Lösungen wurden durch die Anwendung des genetischen Algorithmus zur Optimierung des Assignment bei fest vorgegebenen Allocationen erzeugt. Es wurden verschiedene große Bauteilmengen zur Verfügung gestellt, so daß die Synthese schneller Schaltungen gewährleistet ist. Danach wird der Lösungsraum, der durch die Anwendung des genetische Algorithmus zur Bestimmung der Allocation zustandekommt, dargestellt, wobei die Bewertung durch die Anwendung des genetischen Algorithmus für das Assignment durchgeführt wird, wie dies in Bild 115 dargestellt ist. Da dieser Prozeß zu langen Rechenzeiten führt, wird daraufhin die Akzeleration der Synthese durch die Anwendung der Bewertungsfunktion, die in Def. 10.6-11 dargestellt wird, betrachtet. Dazu wird der genetische Algorithmus für die Suche nach guten

Allocationen über mehrere Generationen mit den definierten Bewertungsfunktionen ausgeführt. Die so erhaltene Menge wird dann als Eingabedatei für den genetischen Algorithmus zur Assignment- und Schedulingoptimierung herangezogen.

### 13.2.1 Die Darstellung des Lösungsraums

Das erste Beispiel ist in dem dieser Arbeit zugrunde liegenden Format in Bild 127 dargestellt. In Bild 128 ist der Deutlichkeit halber das Beispiel in einer Pseudo-Programmiersprache dargestellt. Es besteht aus 28 elementaren Befehlen, wobei eine Schleife in dem Beispiel vorkommt. In der Bauteilbibliothek, die in Tabelle 5 dargestellt ist, sind 17 Bauteile enthalten. Aus dieser Bibliothek wird die Menge der Bauteile für alle folgenden Beispiele allociert. Von den Beispielen 2 und 3 werden nur der Pseudo-Code angegeben. Es handelt sich hierbei um reine sequentielle Programme und entspricht somit der traditionellen Programmierung, wie es durch die Arbeit auch angestrebt wurde

.

```
(in,(),(u,x,ei,SA_1,SA_2,STEP,y2,SE_1,SE_2))
(+,(u,x),(HV_1))
(+,(HV_1,x),(HV_2))
(*,(u,u),(HV_3))
(-,(HV_2,HV_3),(y1))
(+,(u,u),(HV_4))
(-,(u,y1),(HV_5))
(+,(HV_4,HV_5),(HV_6))
(<,(y2,HV_6),(HV_7))
(+,(y1,x),(HV_8))
(if,(HV_7,y1,HV_8),(y1))
(+,(x,ei),(HV_9))
(if,(HV_7,HV_9,x),(x))
(+,(u,u),(HV_10))
(+,(u,u),(HV_11))
(+,(HV_10,HV_11),(x))
SA_1: (+,(u,ei),(HV_12))
(>,(x,HV_12),(HV_13))
SA_2: (whilea,(HV_13,SE_2,STEP),(STEP))
(+,(y1,x),(y1))
(-,(x,ei),(x))
(+,(y1,ei),(y1))
(*,(y1,y1),(HV_14))
(+,(x,ei),(HV_15))
(-,(HV_14,HV_15),(y2))
SE_1: (whilee,(SA_1),(STEP))
SE_2: (+,(y2,ei),(y2))
(out,(y1,y2,STEP),())
```

Bild 127: Beispiel 1

In Bild 129 wird der Einfluß der verschiedenen Allocationen auf die Qualität der Schaltung dargestellt. Jeder Punkt in der Abbildung entspricht einer Lösung im Lösungsraum. Es sind nicht

```
INPUT: u,x,ei : INTEGER;
OUTPUT: y1,y2 : INTEGER;

y1 := (u + x) + x - (u * u);
if (y2 < (u+u+(u-y1)))
 then
  y1 := y1 + x;
 else
  x := x + ei;
end if;
x := ((u+u)+(u+u));
while (x > (u+ei)) loop
   y1 := y1 + x;
   x := x - ei;
   y1 := y1 + ei;
   y2 := (y1 * y1) - (x + ei);
end loop;
y2 := y2 + ei;
```

Bild 128: Pseudo-Code von Beispiel 1

alle möglichen Lösungen dargestellt. Anhand der Häufungen ist der Einfluß der verschiedenen - zur Verfügung gestellten - allocierten Mengen ersichtlich.

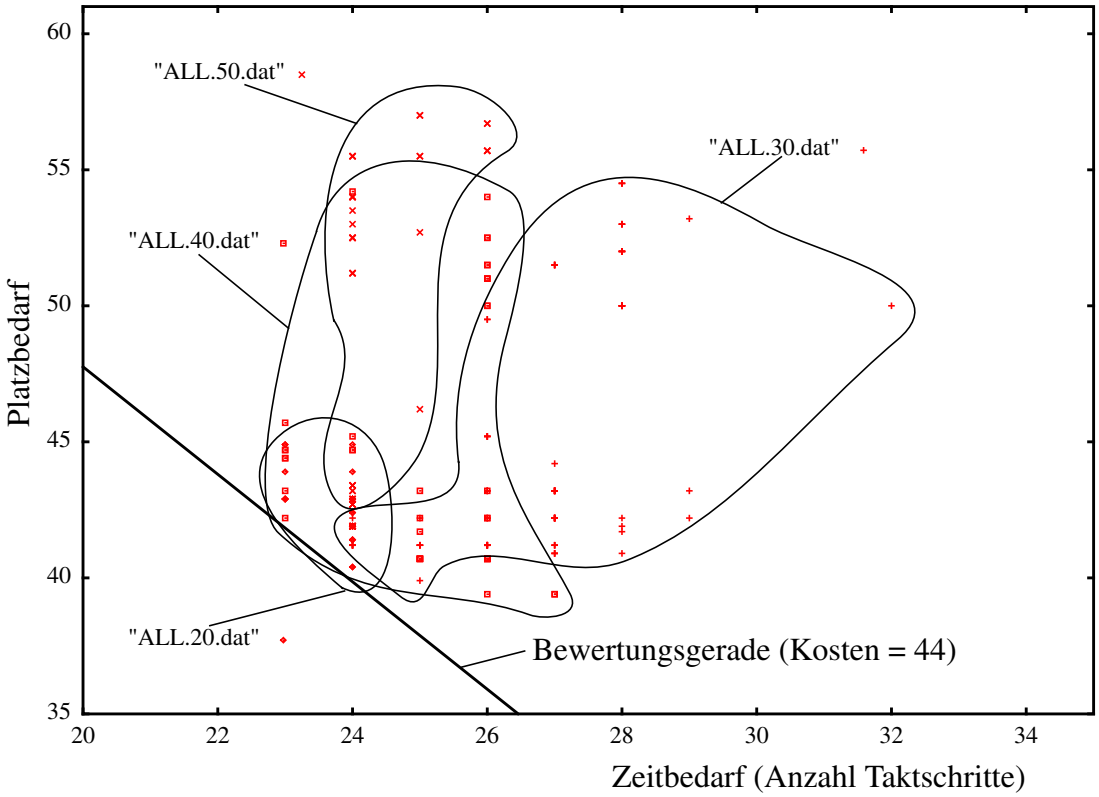



Bild 129: Lösungen in Abhängigkeit verschiedener Allocationen für Beispiel 1

**Tabelle 5: Bauteilbibliothek**

| Bezeichnung | Hardwarebedarf | Funktionen | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|---|---|
| EQUAL | 2 | <, = | 1,1 | 1,1 | 1,1 |
| IN | 1 | in | 1 | 1 | 1 |
| OUT | 1 | out | 1 | 1 | 1 |
| Adder | 1.5 | + | 1 | 1 | 1 |
| ALU | 3.2 | + , - | 2,2 | 2,2 | 2,2 |
| WHILEA | 1.2 | whilea | 1 | 1 | 1 |
| CMP1 | 3 | < , > | 1, 1 | 1, 1 | 1, 1 |
| CMP2 | 1 | < | 1 | 1 | 1 |
| CMP3 | 2.1 | > | 1 | 1 | 1 |
| MULT | 8.3 | * | 4 | 4 | 4 |
| MULT_2 | 5.3 | * | 6 | 6 | 6 |
| MULT_3 | 9.3 | * | 4 | 2 | 2 |
| WHILEE | 0.5 | whilee | 1 | 1 | 1 |
| SUB | 2 | - | 1 | 1 | 1 |
| ALU2 | 9.8 | +, -,* , eq | 2, 2, 6, 1 | 2, 2, 6, 1 | 2, 2, 6, 1 |
| IF | 1.3 | if | 1 | 1 | 1 |
| ADDER_2 | 1.0 | + | 2 | 2 | 2 |
| SUB_2 | 1.5 | - | 2 | 2 | 2 |

Aus den in der Abbildung dargestellten Lösungen läßt sich anhand der Bewertungsfunktion - wie sie in Def. 10.18-9 definiert ist - die beste herausfinden. In dem Beispiel wurde die Bewertungsfunktion so gewählt, daß die Anzahl der Variablen mit 0.5 bewertet wurde, der Ressourcenbedarf ebenfalls mit 0.5 und der Zeitbedarf mit 1. Also wurde der Gewichtsvektor als $w_{as} = (1, 0, 0.5 , 0.5 , 0, 0)$ festgelegt. In der Abbildung stellt die y-Achse die Summe der Anzahl der Variablen und des Ressourcenbedarfs dar. Die x-Achse stellt die Anzahl der Taktschritte dar. Aus der Wahl der Gewichte für die Bewertungsfunktion folgt, daß Lösungen mit der gleichen Bewertung auf einer Geraden liegen müssen, welche parallel zur eingezeichneten Bewertungsgeraden ist. Die schnellste mögliche Schaltung hat für das Beispiel 23 Taktschritte. Dieser Wert wird auch im Lösungsraum erreicht. Das nächste Beispiel besteht aus 53 Befehlen, und es sind außerdem zwei Schleifen enthalten. Der Pseudo-Code von Beispiel 2 ist in Bild 130 dargestellt.

Der Lösungsraum ist in Bild 131 dargestellt. Die minimale Anzahl der benötigten Taktschritte beträgt 39. Die höhere Anzahl an Befehlen bedingt einen größeren Lösungsraum, wodurch ein höherer Zeitbedarf für die Optimierung zustandekommt.

Der Pseudo-Code von Beispiel 3 ist in Bild 132 dargestellt.

```
INPUT: u, x,ei: INTEGER ;
OUTPUT: y1,y2 : INTEGER ;

y1 := (u + x) + x - (u * u);
if (y2 < (u+u+(u-y1)))
then
   y1 := y1 + x;
else
   x := x + ei;
end if;

x := ((u+u)+(u+u));

while (x > (u+ei)) loop
   y1 := y1 + x;
   x := x - ei;
   y1 := y1 + ei;
   y2 := (y1 * y1) - (x + ei);
end loop;

y2 := y2 + ei;

while (x > (u+ei)) loop
   y1 := y1 + x;
   x := x - ei;
   y1 := y1 + ei;
   y2 := (y1 * y1) - (x + ei);
end loop;

y1 := (u + x) + x - (u * u);
if (y2 < (u+u+(u-y1)))
then
   y1 := y1 + x;
else
   x := x + ei;
end if;

x := ((u+u)+(u+u));
```

Bild 130: Beispiel 2

Der Lösungsraum für das dritte Beispiel ist in Bild 133 dargestellt. In dem Beispiel kommen keine Schleifen vor, und es besteht aus 49 elementaren Befehlen. Der minimale Bedarf an Taktschritten beträgt 24. Dadurch, daß keine Schleifen vorhanden sind, wird die Prozedur der Schleifenoptimierung nicht benötigt, was sich positiv auf den Zeitbedarf des Algorithmus auswirkt.

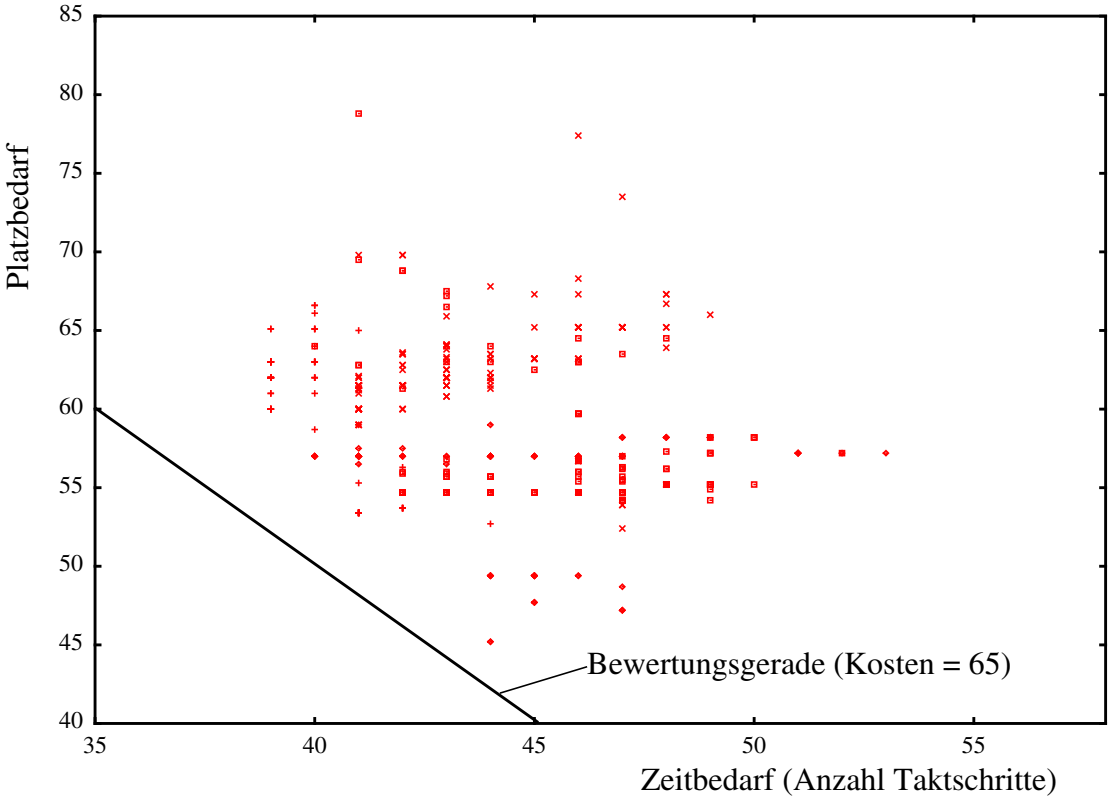



Bild 131: Lösungsraum für Beispiel 2

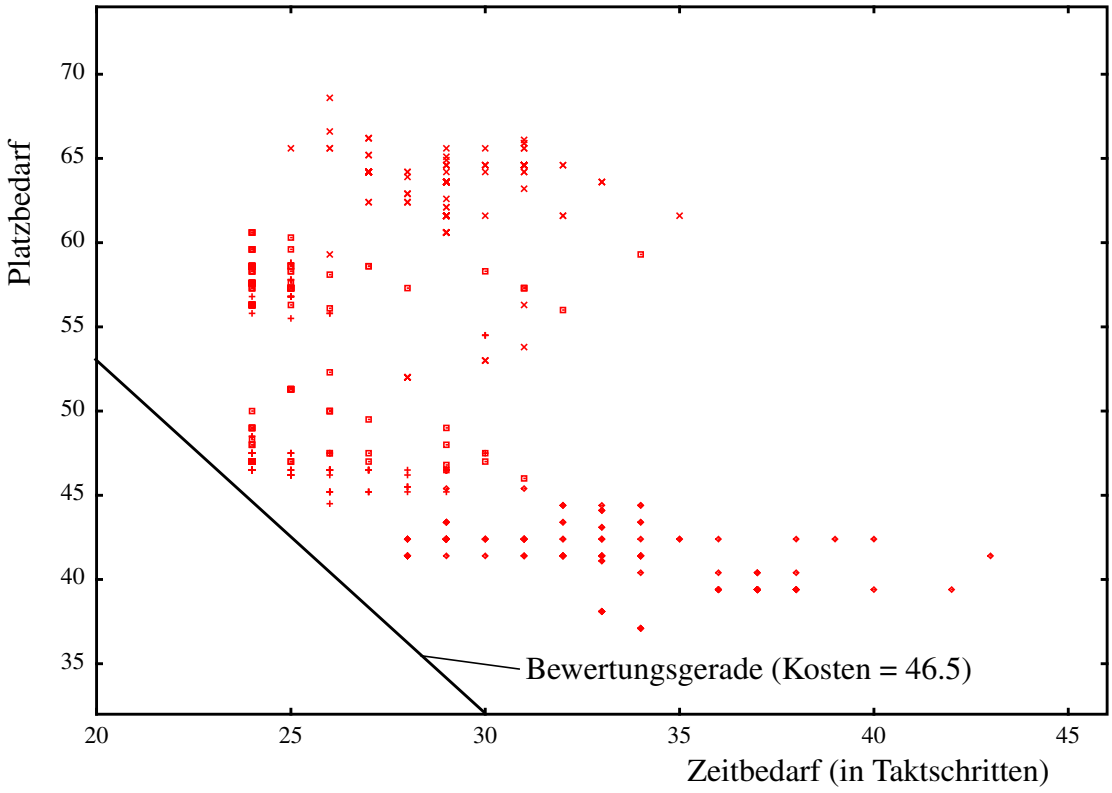



Bild 133: Lösungsraum für Beispiel 3

```
INPUT : u, x,ei : INTEGER;
OUTPUT : y1,y2 : INTEGER

y1 := (u + x) + x - (u * u);
if (y2 < (u+u+(u-y1)))
then
  y1 := y1 + x;
else
  x := x + ei;
end if;

x := ((u+u)+(u+u));
u := (u+ei)
y1 := y1 + x;
x := x - ei;
y1 := y1 + ei;
y2 := (y1 * y1) - (x + ei);
y2 := y2 + ei;
u := u+ei
y1 := y1 + x;
x := x - ei;
y1 := y1 + ei;
y2 := (y1 * y1) - (x + ei);
y1 := (u + x) + x - (u * u);

if (y2 < (u+u+(u-y1)))
then
  y1 := y1 + x;
else
  x := x + ei;
end if;
x := ((u+u)+(u+u));
```

Bild 132: Beispiel 3

### 13.2.2 Optimierung durch genetischen Allocationsalgorithmus

In der Bild 134 wird die Verteilung der Lösungen für das erste Beispiel dargestellt, wobei der genetische Algorithmus, wie er in Bild 115 dargestellt ist, zur Wahl der allocierten Bauteilmenge angewandt wurde. Es ist deutlich ersichtlich, daß eine erhebliche Verbesserung der Lösungen auf diese Weise möglich ist, gerade was den Ressourcenbedarf angeht.

Das erste Beispiel zeigt, daß der Platzbedarf der Lösungen reduziert werden kann. Der Nachteil der Berechnung mit Hilfe des genetischen Algorithmus ist seine Laufzeit, weshalb im folgenden dargestellt werden soll, wie sich die beschleunigte Allocation, die die in Def. 10.6-11 dargestellten Bewertungsfunktionen nutzt, auf die Lösungen auswirkt. In der Bild 135 werden die gefundenen Lösungen für das Beispiel 2 dargestellt. Auch hier ergibt sich eine Reduzierung des Platzbedarfs, wodurch eine Gütesteigerung erreicht wird.

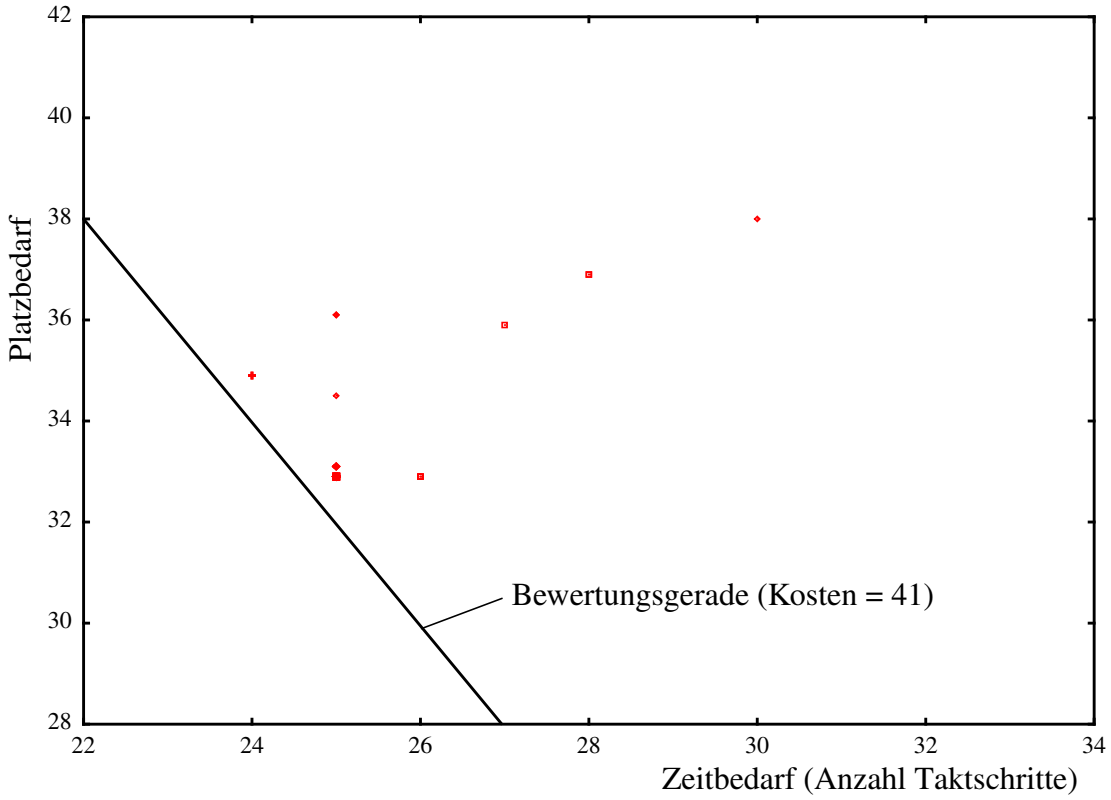



Bild 134: Mit genetischem Algorithmus gefundene Lösungen für Beispiel 1

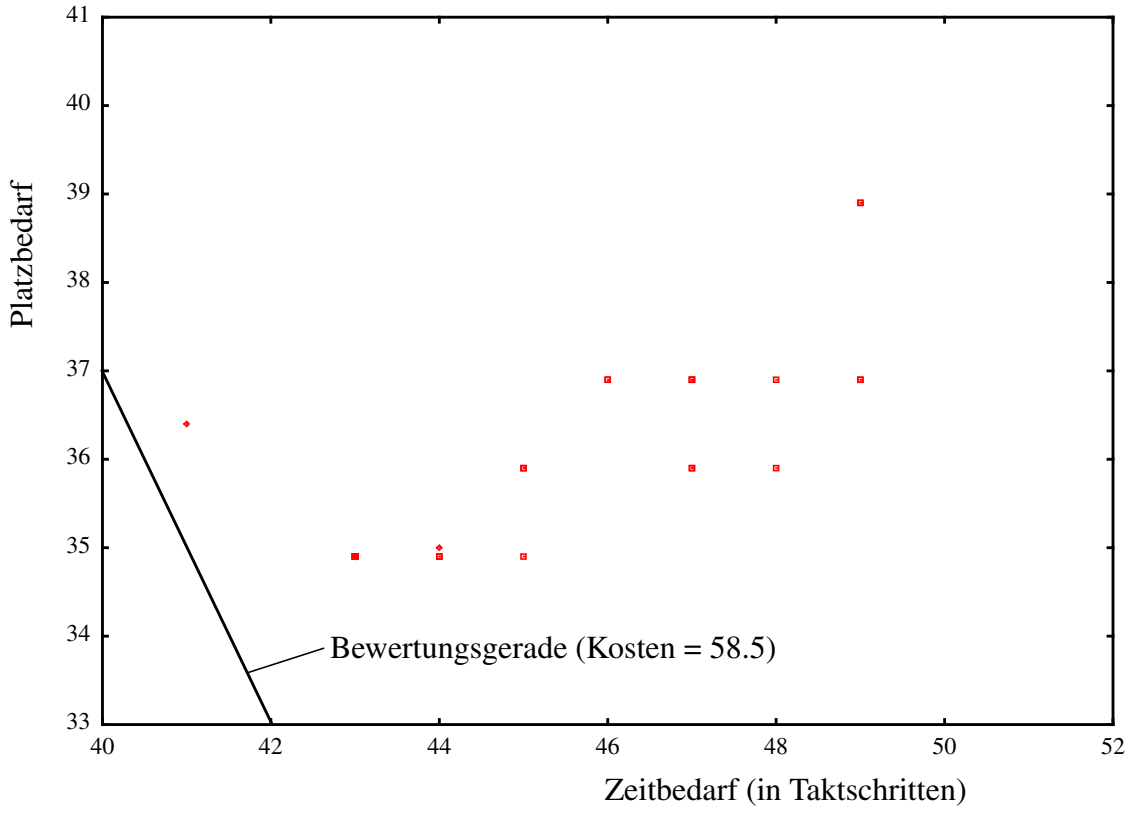



Bild 135: Mit genetischem Algorithmus gefundene Ergebnisse für Beispiel 2

Die Lösungen, die der beschleunigte genetische Algorithmus für das dritte Beispiel liefert, ist in Bild 136 dargestellt. Da hier keine Schleifen enthalten sind, werden weniger Taktschritte benötigt. Es ist auch bei diesem Beispiel ersichtlich, daß die Güte gesteigert werden kann. Da die Laufzeit der verwendeten Algorithmen nicht linear sondern teilweise sogar kubisch bzgl. der Anzahl der Befehle ist, werden die Algorithmen sehr langsam, weshalb eine Akzeleration der Synthese sinnvoll und nötig ist.

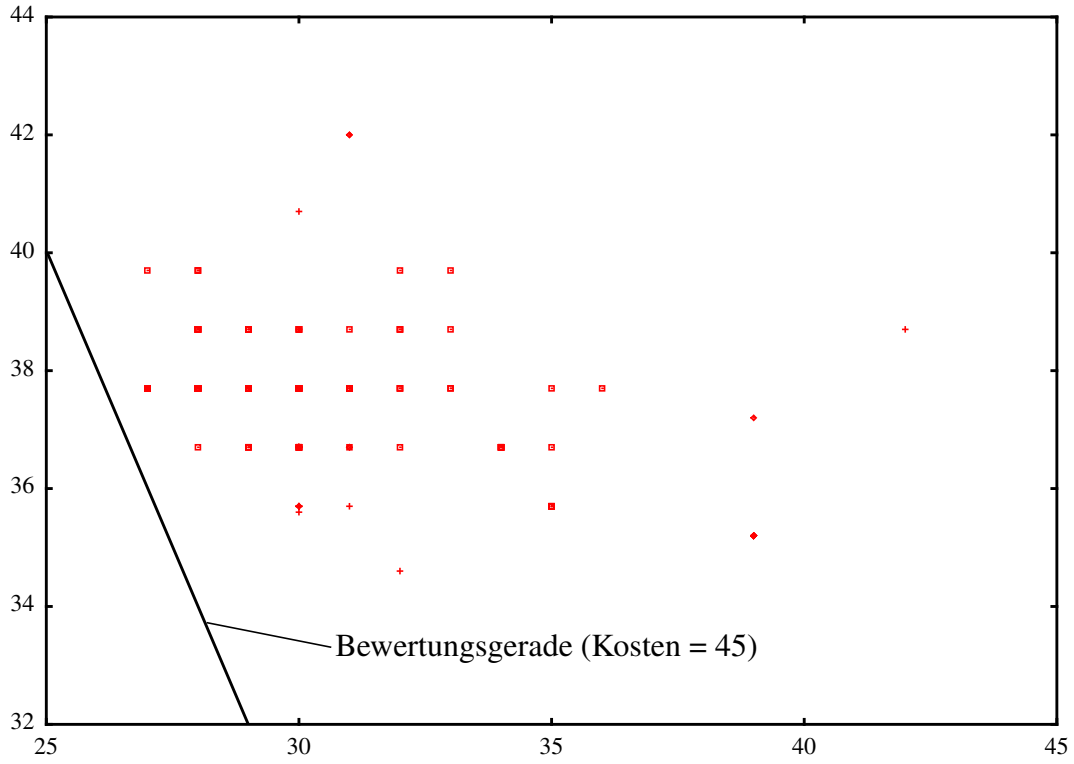



Bild 136: Mit genetischem Algorithmus gefundenen Lösungen für Beispiel 3

### 13.2.3 Akzeleration der Synthese durch Abschätzung der Allocation

Nutzt man die in Def. 10.6-11 definierte Bewertungsfunktion für die Allocation, dann kann der genetische Algorithmus zum Auffinden einer geeigneten Bauteilmenge beschleunigt werden. Wird dem genetischen Algorithmus für das Assignment eine Bauteilliste zur Verfügung gestellt, die relativ groß ist, wird der Lösungsraum entsprechend groß. Durch eine Einschränkung der allocierten Bauteilmenge wird auch der Lösungsraum eingeschränkt. Man verkürzt damit die Rechenzeit. Unter Anwendung der in Def. 10.6-11 angegebenen Bewertungsfunktionen wird der genetische Algorithmus gestartet. Dieser genetische Algorithmus ist wegen der schnellen Berechnung der Bewertung der Allocationen schneller als der in Bild 115 dargestellte. Steht die allocierte Bauteilmenge fest, wird der Algorithmus zur Optimierung des Assignment angewandt. In Bild 137 ist der Vergleich von vier Durchläufen des genetischen Algorithmus zur Optimierung des Assignments dargestellt. Dazu wurde einmal eine relativ große Bauteilmenge allociert und zur Verfügung gestellt, das andere Mal wurde die Berechnung der allocierten Bauteilmenge mit dem dargestellten beschleunigten genetischen Algorithmus ausgeführt. Die Wahl einer relativ großen Menge an Bauteilen kann damit motiviert werden, daß die maximal mögliche Parallelität erreicht werden soll. Die beschleunigte Allocation garantiert nicht die Güte der Ergebnisse, die durch den oben dargestellten genetischen Algorithmus berechnet werden, liefert aber deutlich bessere Ergebnisse, als dies bei großen allocierten Bauteilmengen der Fall ist.

Wichtiger ist die deutliche Akzeleration des genetischen Algorithmus, die in Bild 137 verdeutlicht wird. Zusätzlich wurden Messungen durchgeführt, die das Verhalten des genetischen Algorithmus darstellen, wenn eine Heuristik zur Optimierung der Zwischenergebnisse genutzt wird. Es zeigt sich, daß hier noch einmal eine deutliche Steigerung der Qualität zu erreichen ist. Die besten Ergebnisse werden, wie zu erwarten ist, dann erzielt, wenn sowohl eine automatisch erzeugte optimierte Bauteilmenge zur Verfügung gestellt, als auch der heuristische Algorithmus in Kombination mit dem genetischen Algorithmus angewandt wird.

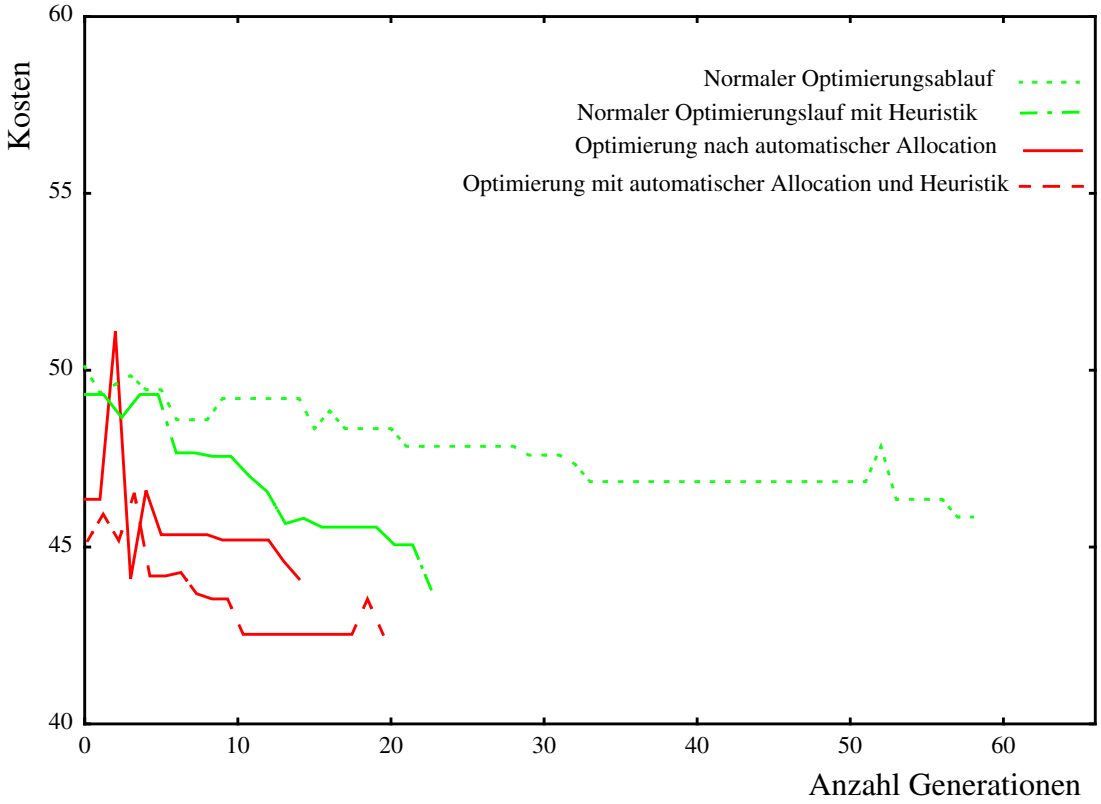



Bild 137: Akzeleration durch automatische Allocation für Beispiel 1

Der Zeitbedarf für die Durchführung des genetischen Algorithmus zur Optimierung des Assignments wird in diesem Fall auf ein Viertel reduziert. Addiert man noch den Zeitbedarf zum Auffinden einer Bauteilmenge hinzu, so läßt sich der Zeitbedarf in diesem Fall immer noch auf die Hälfte reduzieren. Bei größeren Befehlsmengen erhöht sich der Zeitgewinn weiter. Neben der definierten Methode für die Allocation der Bauteile wird eine heuristische Methode, wie dies in Abschnitt 10.22 dargestellt ist, mit dem genetischen Algorithmus verknüpft. Dies führt zu weiter verbesserten Ergebnissen. Gerade die Verknüpfung des genetischen Algorithmus mit der dargestellten optimierten Allocation und einem heuristischen Algorithmus bringt eine enorme Optimierung der Ergebnisse und eine Akzeleration der Berechnung. In Bild 138 ist der Verlauf der Optimierung für das Beispiel 2 dargestellt. Durch die automatische Allocation ist die Lösungsmenge stark eingeschränkt, so daß schon die initialen Lösungen gute Ergebnisse darstellen.

Für das dritte Beispiel ist der Vergleich in Bild 139 dargestellt. Durch den eingeschränkten Suchraum ist hier schon nach ziemlich wenigen Generationen eine Verbesserung der Ergebnisse gegeben. Bei der großen Anzahl an Befehlen macht sich die Rechenzeit für die beschleunigte Allocation in dem Gesamtablauf kaum noch bemerkbar. Man kann hier von einer Verdopplung der Geschwindigkeit sprechen, bzw. die Rechenzeit beliebig verkürzen, da schon initial sehr

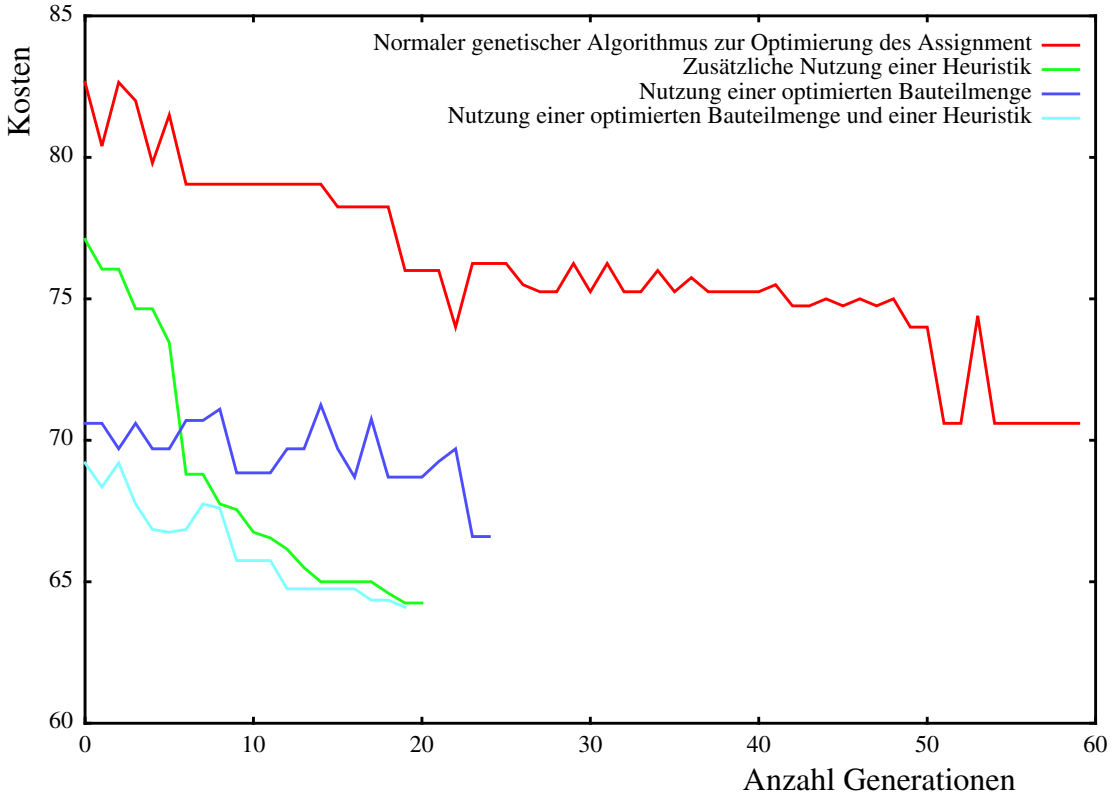



Bild 138: Akzeleration durch automatische Allocation für Beispiel 2

gute Ergebnisse gefunden werden. Dieser Effekt ist nur damit zu erklären, daß die zur Verfügung gestellte Bauteilmenge optimal an die Befehlsmenge angepaßt ist, wodurch der zu durchsuchende Lösungsraum reduziert ist. In der Kombination mit einer Heuristik sind hier relativ schnell sehr gute Ergebnisse zu erzielen.

Anhand der Beispiele wird deutlich, daß mit den genannten Verfahren sehr gute Ergebnisse erzielt werden können. Die besten Ergebnisse werden mit dem genetischen Algorithmus aus Bild 115 erzielt. Da dieses Verfahren trotz Parallelisierung recht zeitintensiv ist, wird alternativ dazu eine große Menge an Bauteilen zur Verfügung gestellt. Die Optimierung beschränkt sich dann auf die Optimierung des Assignments, welches schneller zu Lösungen kommt, wobei die Güte der Lösungen nicht immer die Erwartungen erfüllt. Daraufhin wird eine Abschätzung der Güte der allocierten Bauteilmengen gemacht, wie sie in Def. 10.6-11 dargestellt ist. Es entstehen hierdurch zwei unabhängige genetische Algorithmen, einmal der Algorithmus, welcher eine Allocation berechnet, und der Algorithmus zur Optimierung des Assignments. Da die Berechnung der Bewertung einer Allocation durch die Schätzfunktion sehr schnell durchgeführt werden kann, ist der Algorithmus sehr schnell und fällt in der Gesamtrechenzeit kaum ins Gewicht. Außerdem werden Verbesserungen der Ergebnisse erzielt. Der zusätzliche Einsatz von heuristischen Methoden bringt - wie dies schon in Abschnitt 10.22 angedeutet wurde - noch einmal einen Gewinn. In Bild 140 werden die Meßergebnisse für die Synthese des Elliptic Wave Filter [76] dargestellt, um zu zeigen, daß auch für einen Standartbenchmark äquivalente Ergebnisse erzielt werden und damit das System auch in der Praxis ohne Einschränkungen angewandt werden kann. Tabelle 6 zeigt die absoluten Ergebnisse, welche für das Elliptic Wave Filter erzielt wurden, wobei der Flächenbedarf des Multiplizierers vier mal so hoch angenommen wurde, als der des Addierers, außerdem wurde der Zeitbedarf des Multiplizierers mit zwei Kontrollschritten festgelegt, wobei der Addierer nur einen benötigt.

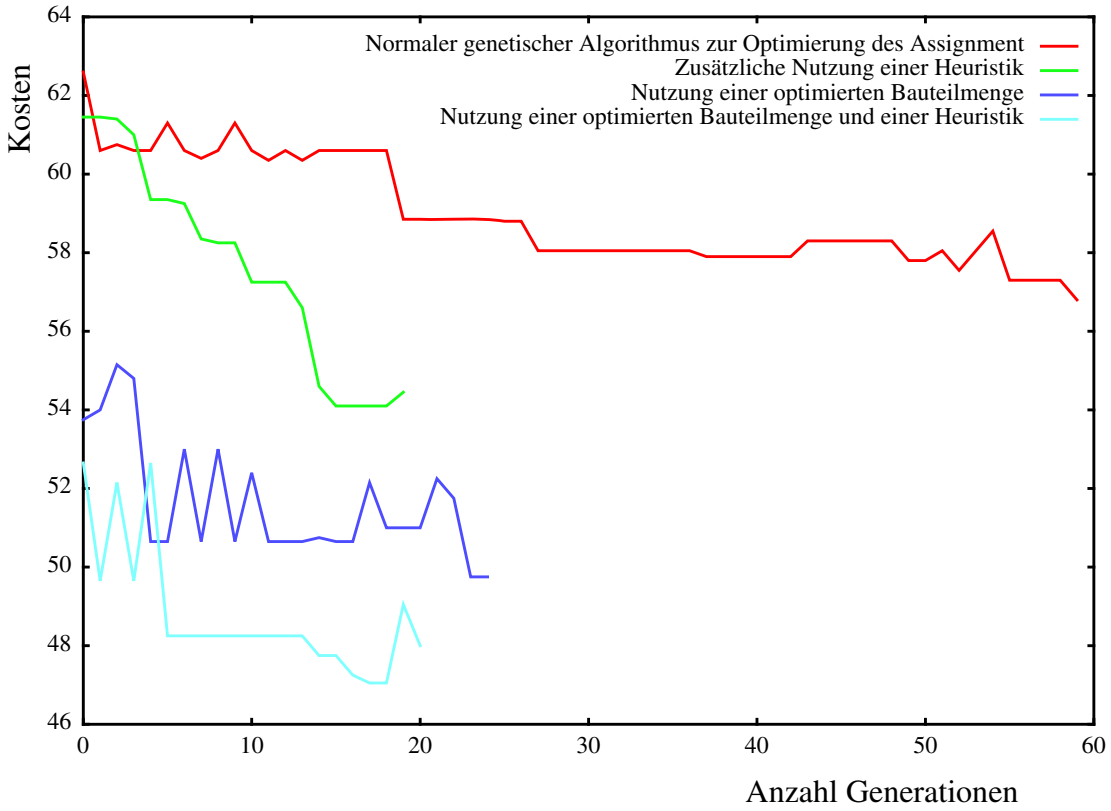



Bild 139: Akzeleration durch automatische Allocation für Beispiel 3

**Tabelle 6: Elliptical Wave Filter**

| Verfahren | Kontroll schritte | Addierer | Multi-plizierer |
|---|---|---|---|
| Normal | 18 | 3 | 5 |
| opt. Allocation | 19 | 3 | 3 |
| Heuristik | 18 | 2 | 5 |
| opt. Alloc und Heu. | 17 | 5 | 3 |

Im folgenden wird die Akzeleration der Synthese durch parallele Bewertung der Lösungen im Workstationcluster dargestellt.

### 13.2.4 Akzeleration der Synthese durch Parallelisierung

Die erreichte Beschleunigung des genetischen Algorithmus wird für die Beispiele in den folgenden Darstellungen verdeutlicht. Bild 141 zeigt die Beschleunigung des genetischen Algorithmus für das erste Beispiel.

An dem ersten Beispiel wird deutlich, daß sich eine Beschleunigung durch die Verwendung von mehreren Workstations ergibt. Mehr als fünf Workstations zu verwenden, lohnt sich nicht, da der Kommunikationsaufwand steigt und die Anzahl der parallelen Zugriffe auf die Daten begrenzt ist. In der Bild 142 wird die Beschleunigung der Synthese für Beispiel 2 dargestellt. An-

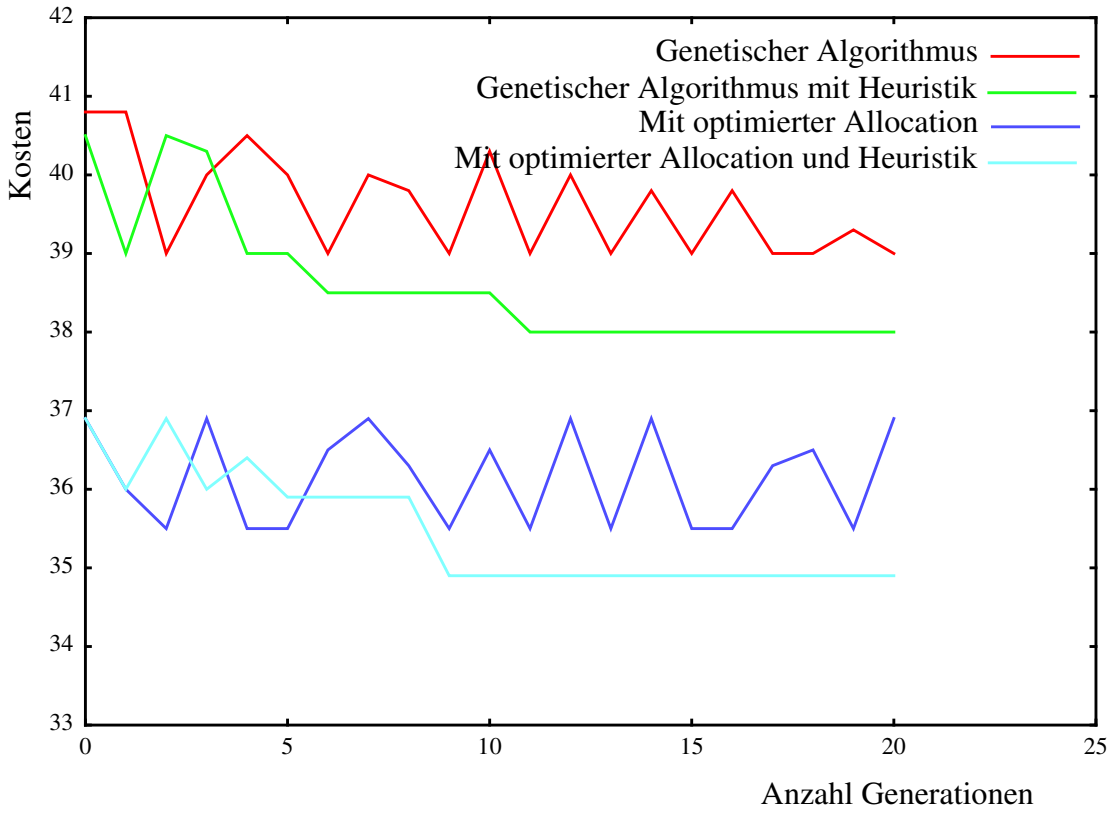



Bild 140: Ergebnisse Elliptical Wave Filter

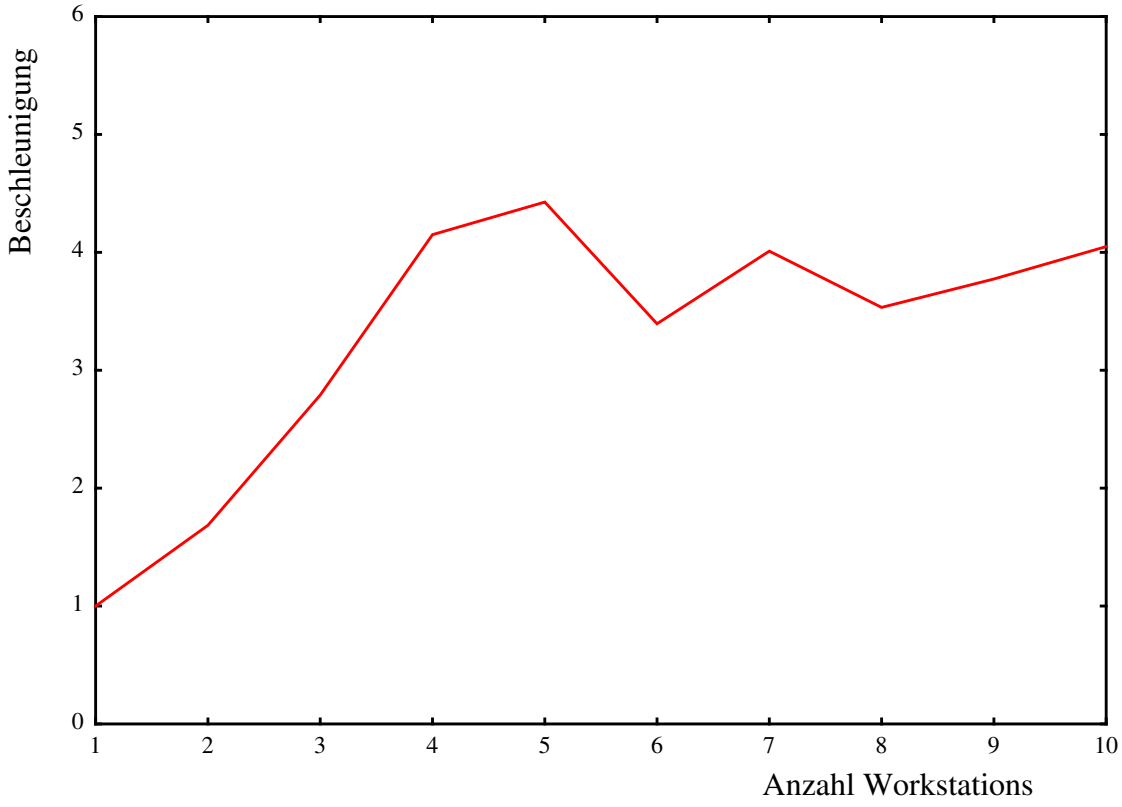



Bild 141: Speedup des genetischen Algorithmus für Beispiel 1

hand des Beispiels wird deutlich, daß der Einsatz von sieben Workstations einen deutlichen Gewinn bringt. Bei der Verwendung von mehr als sieben Workstations wird der Verwaltungsaufwand zu hoch, so daß kein weiterer Gewinn zu erzielen ist.

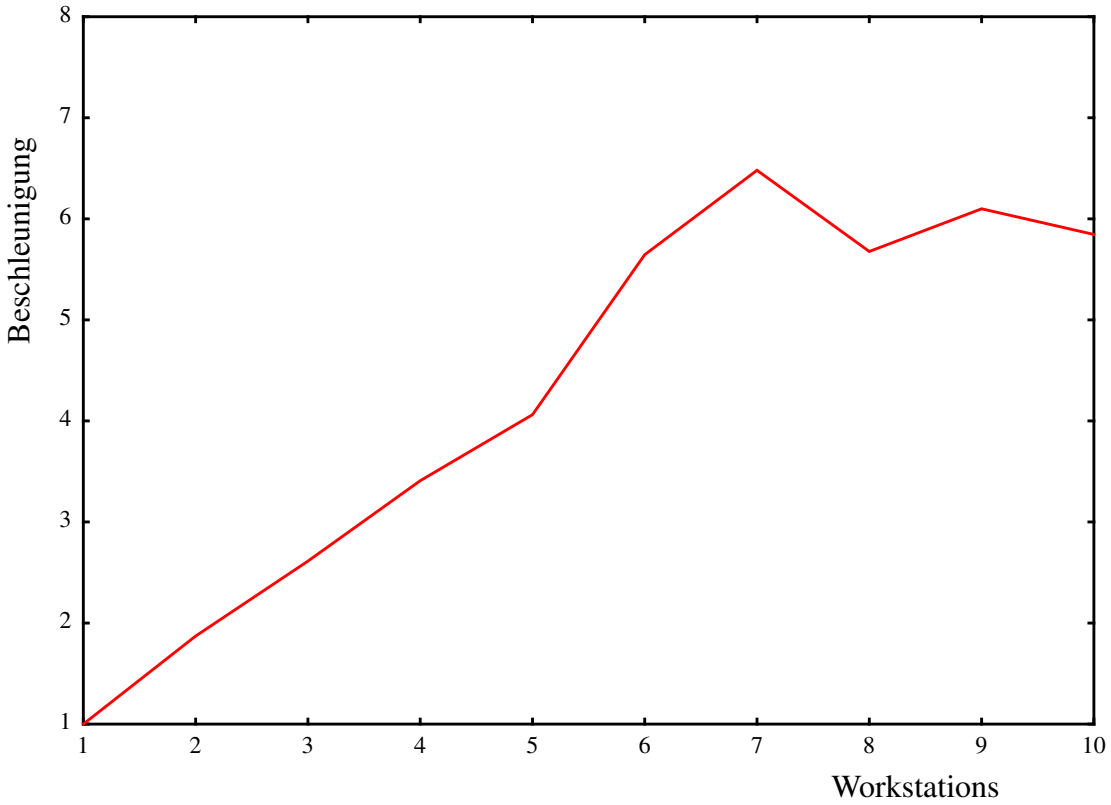



Bild 142: Beschleunigung des genetischen Algorithmus für Beispiel 2

In der Bild 143 wird die Beschleunigung der Synthese für das Beispiel 3 dargestellt. Wie die weiteren Messungen gezeigt haben, ist die Obergrenze für die Anzahl der Workstations bei zehn erreicht. Danach nimmt die Beschleunigung auch für dieses Beispiel wieder ab, bzw. bleibt etwa auf dem erreichten Niveau.

### 13.2.5 Vergleich mit anderen Algorithmen

An dieser Stelle soll noch der Vergleich des genetischen Algorithmus mit Simulated Annealing und mit dem schon oben erwähnten und benutzten heuristischen Algorithmus gemacht werden. Das Problem bei Simulated Annealing und dem heuristischen Algorithmus ist, daß es nicht möglich ist, die Verfahren zu parallelisieren. Für den genetischen Algorithmus mit 18 Lösungen pro Generation, Simulated Annealing und dem heuristischen Verfahren zur Optimierung des Assignments werden in den folgenden Abbildungen - Bild 144, Bild 145 und Bild 146 - auf der x-Achse der Zeitbedarf und auf der y-Achse die Bewertung der Lösungen dargestellt. Für jedes der drei Verfahren wurde die gleiche Menge allocierter Bauteile zur Verfügung gestellt. Aus diesen und den Darstellungen des letzten Abschnitts kann entnommen werden, bei wieviel zur Verfügung stehenden Workstations es sinnvoll ist, den genetischen Algorithmus anzuwenden. In Bild 144 wird für Beispiel 1 das Verhalten von Simulated Annealing, dem heuristischen Algorithmus und dem sequentiellen genetischen Algorithmus dargestellt. Dies führt zu ähnlichen Optimierungsergebnissen. Die entscheidende Verbesserung sowohl der Ergebnisse als auch der Laufzeit des genetischen Algorithmus wird durch die Parallelisierung und die zusätzliche Nut-

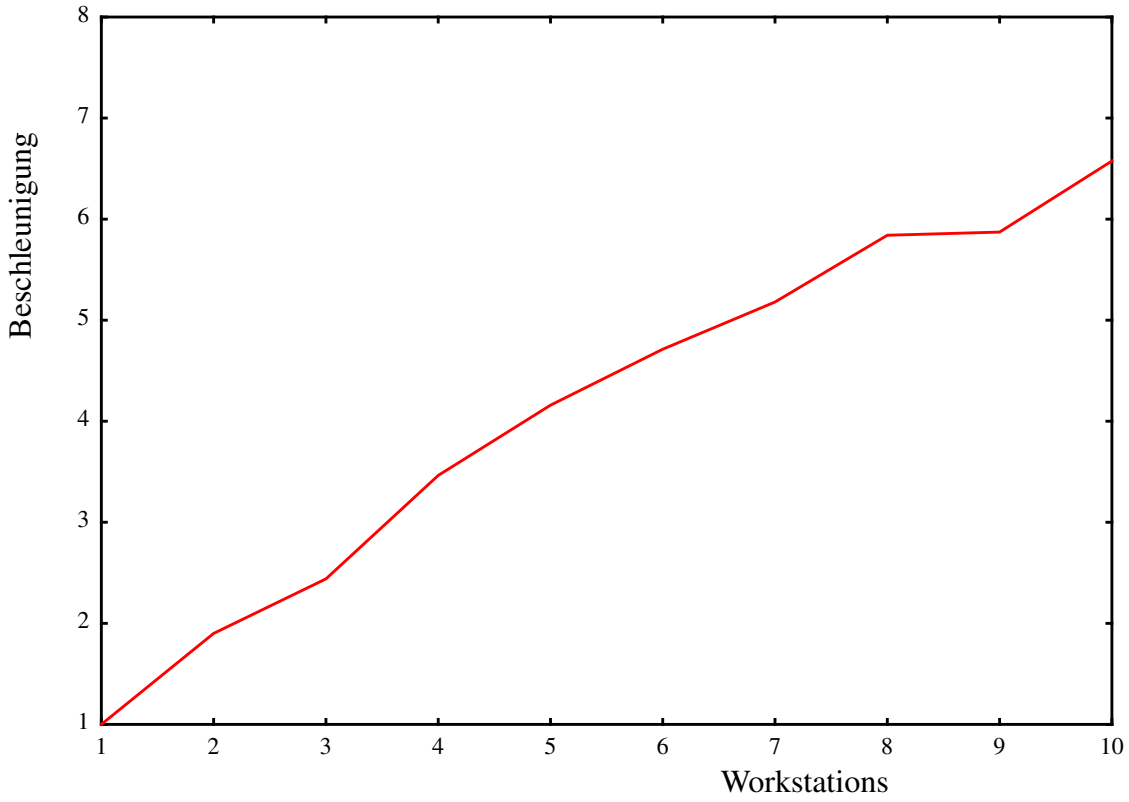



Bild 143: Beschleunigung des genetischen Algorithmus für Beispiel 3

zung einer Heuristik erreicht. Durch die gemischte Anwendung von Heuristik und genetischem Algorithmus kann sowohl eine Beschleunigung als auch eine weitere Steigerung der Qualität erreicht werden.

Für das erste Beispiel ist es schon dann sinnvoll, den genetischen Algorithmus anzuwenden, wenn zwei Workstations zur Verfügung stehen. Bei zwei Rechnern wird eine Beschleunigung von 1.8 erreicht, wodurch der genetische Algorithmus schneller ist als Simulated Annealing. In der Bild 145 wird der Vergleich für das Beispiel 2 dargestellt.

In diesem Beispiel würde eine Beschleunigung um den Faktor 3 ausreichen, damit der genetische Algorithmus in der gleichen Zeit zu gleichen oder zu besseren Ergebnissen kommt als Simulated Annealing. Eine Beschleunigung um den Faktor 3 kann erreicht werden, wenn mindestens 4 Workstations zur Verfügung stehen. Eine weitere Verbesserung der Ergebnisse wird hier wiederum durch die Kombination mit der Heuristik erreicht, in der Abbildung ist der Optimierungslauf bei paralleler Nutzung von fünf Workstations dargestellt. In der Bild 146 wird der Vergleich für das Beispiel 3 dargestellt, die Ergebnisse von Simulated Annealing konnten erst durch die zusätzliche Nutzung der Heuristik übertroffen werden. In der Abbildung ist der Optimierungslauf, welcher durch die parallele Nutzung von fünf Workstations erreicht wird, dargestellt.

Aus den Abbildungen wird ersichtlich, daß die kombinierte Nutzung sowohl von Heuristischen Methoden als auch von genetischen Algorithmen, welche parallelisiert werden, zu optimalen Ergebnissen führt. In Bild 147 sind abschließend die Ergebnisse von Simulated Annealing und dem Genetischen Algorithmus mit und ohne Optimierungen für das Beispiel des Elliptical Wave Filter dargestellt.

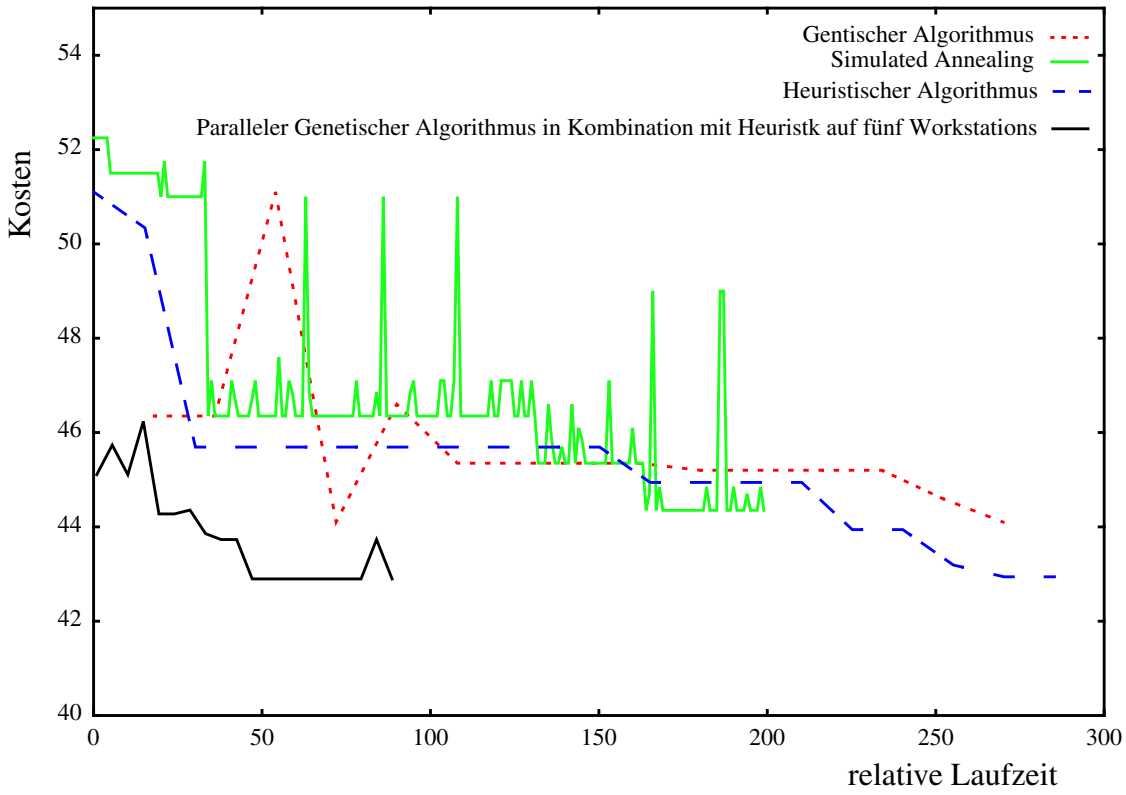



Bild 144: Vergleich mit anderen Algorithmen für Beispiel 1

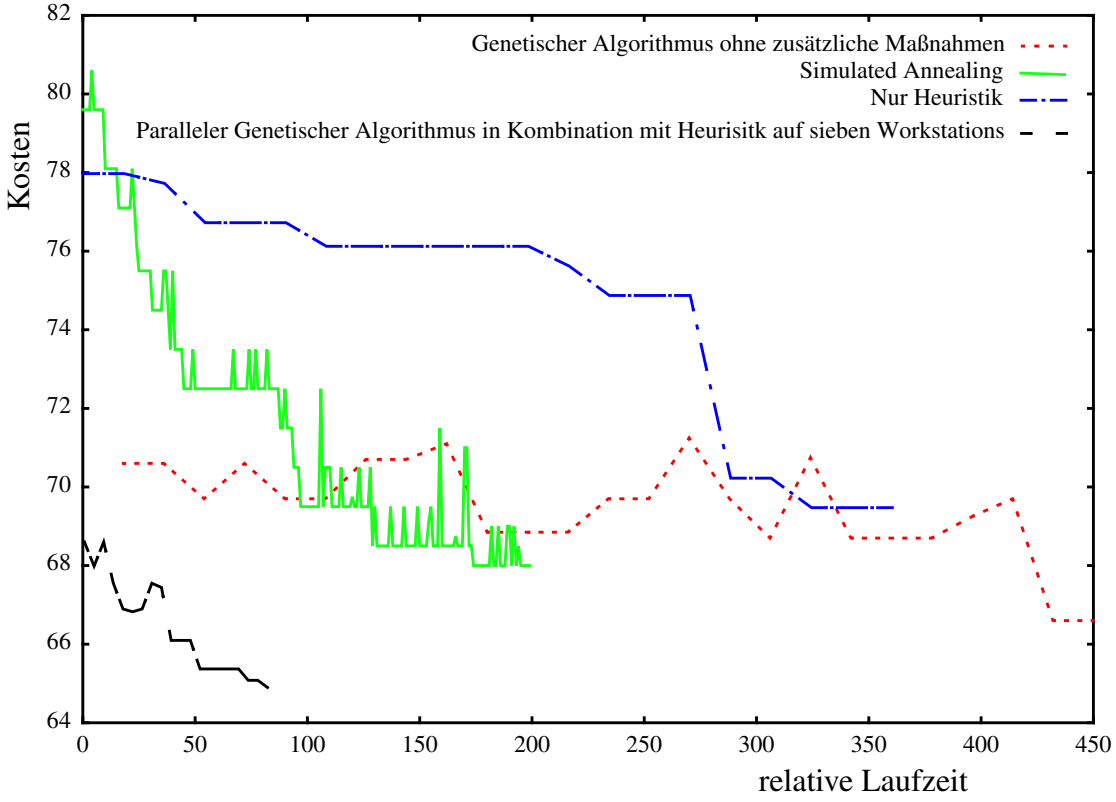



Bild 145: Vergleich mit Simulated Annealing für Beispiel 2

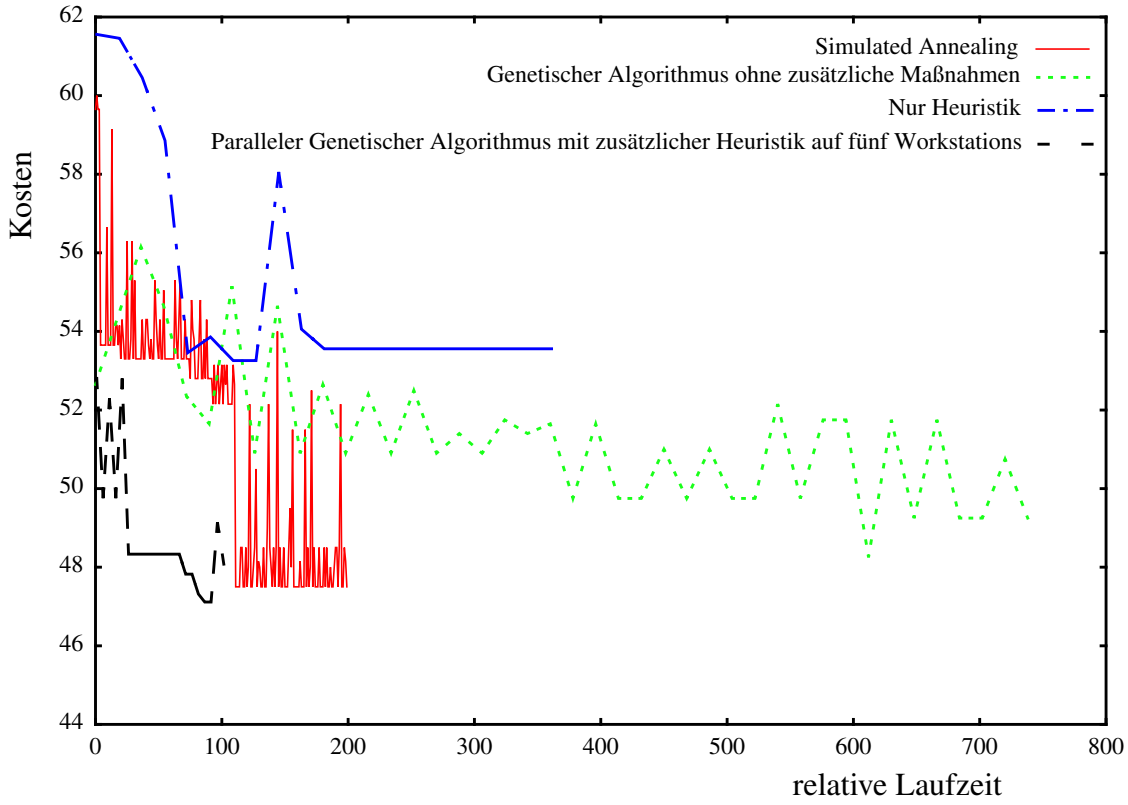



Bild 146: Vergleich mit Simulated Annealing für Beispiel 3

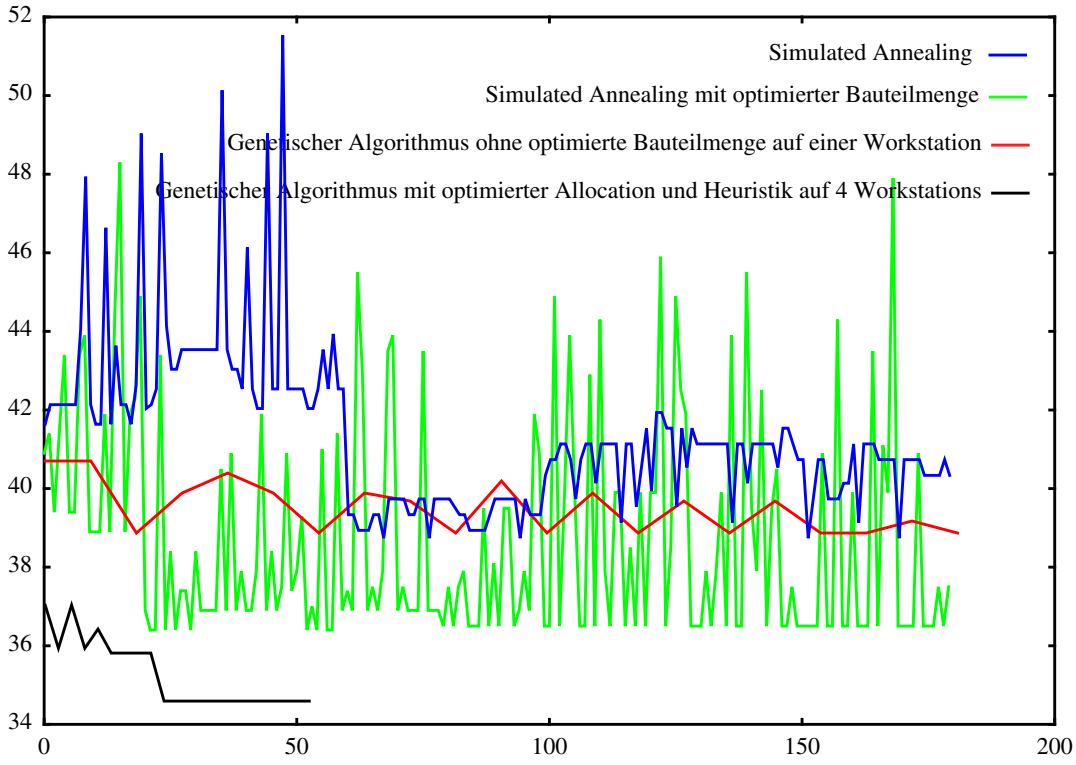



Bild 147: Verschiedene Algorithmen für den Elliptical Wave Filter

Die Optimierung der Bauteilmenge ist von der Nutzung des genetischen Algorithmus unabhängig und kann auch bei anderen Algorithmen als Optimierungsschritt eingesetzt werden. Aus allen Beispielen ist ersichtlich, daß die Nutzung genetischer Algorithmen nur dann von Nutzen ist, wenn die Möglichkeit der parallelen Nutzung mehrere Rechner besteht und zum anderen wenn zusätzliche heuristische Methoden zum Einsatz kommen. Gerade die Kombination verschiedener Algorithmen führt zu einer Optimierung der Ergebnisse.

# 14 Zusammenfassung und Fazit

Im Rahmen dieser Arbeit wurde eine intensive theoretische Betrachtung der High-Level Synthese durchgeführt. Die Erkenntnisse sind nicht auf eine Hardwarebeschreibungssprache beschränkt sondern allgemeingültig. Nachdem ein Befehlsformat definiert wurde, wurde die Umsetzung typischer Hochsprachenkonstrukte in das Befehlsformat dargestellt. Durch die Umsetzung wurde ein spekulatives Scheduling in das System integriert. Des weiteren wurde es ermöglicht, Schleifen mit einem dynamischen Abbruchkriterium zu synthetisieren, wobei ein Verfahren zur Optimierung der Schleifen ebenfalls vorgestellt wurde. Bauteile wurden so definiert, daß multifunktionale Bauteile mit Pipelining mit berücksichtigt werden können. Für jedes Bauteil wurde der Zeitbedarf bzgl. einer Operation definiert, welcher dann nach der Zuweisung der elementaren Befehle zu den Bauteilen ebenfalls beim Scheduling berücksichtigt werden konnte. Durch diese differenzierte Definition der Bauteile wird die Anzahl der möglichen Lösungen größer als bei anderen Systemen, wodurch der Einsatz genetischer Algorithmen gerechtfertigt ist. Es wurden in der Arbeit zwei genetische Algorithmen definiert, einmal der für die Allocation, des weiteren der für das Assignment und das Scheduling. Die Bewertung einer Allocation geschieht dabei durch die Bewertung des Assignments und des Scheduling, welches sich daraus ergibt. Sehr gute Ergebnisses wurden durch die explizite Betrachtung und Bewertung der allocierten Bauteilmengen erreicht. Das Verfahren zur Auswahl einer geeigneten Bauteilmenge führt einerseits zu einer Beschleunigung der Synthese, des weiteren zu besseren Syntheseergebnissen. Ein weiterer Vorteil des Verfahrens ist, daß es mit unterschiedlichen Methoden zur Optimierung des Assignment und Scheduling, kombiniert werden kann. Der Problematik der hohen Laufzeiten genetischer Algorithmen wird durch dessen Parallelisierung begegnet. Des weiteren wurde die Synthese von multifunktionalen Bauteilen betrachtet, wodurch Ergebnisse erzielt werden, die bei gleichen Qualtitätsmerkmalen mit geringerem Ressourcenbedarf auskommen. Zum Schluß wurde dann die Beschleunigung der genetischen Algorithmen durch die Parallelisierung in einem Workstationcluster dargestellt. Die Messungen zeigen, daß die Anwendung genetischer Algorithmen dann vorteilhaft ist, wenn mehrere Workstations parallel genutzt werden können, wie dies z.B. nachts der Fall ist, wenn die Auslastung der Workstations relativ gering ist. Die allgemeine Darstellung der Problematik erlaubt es, auf einfache Weise andere Algorithmen in das System einzuordnen. In der Arbeit wurden einige Messungen dargestellt, die eine weitere Optimierung und Beschleunigung des genetischen Algorithmus durch die Kombination mit heuristischen Verfahren zeigen. Sicherlich kann eine genauere Betrachtung von Heuristiken zu weiteren Ergebnissen führen. Die Arbeit kann und soll an dieser Stelle nicht als abgeschlossen betrachtet werden, sondern eine theoretische und praktische Grundlage liefern, auf der weiter gearbeitet werden kann. Die Optimierung der Strategien der genetischen Algorithmen muß speziell für diese Anwendungen noch intensiv betrachtet werden. Ein weiterer Aspekt, welcher noch vertieft werden kann, ist die Nutzung von Heuristiken gerade in der Kombination mit genetischen Algorithmen. Zusammenfassend kann also gesagt werden, daß eine Beschleunigung der Synthese durch die Nutzung von genetischen Algorithmen dann möglich und sinnvoll ist, wenn mehrere Rechner parallel genutzt werden können. Außerdem bilden Kombinationen von genetischen und anderen Verfahren eine Grundlage für optimale Synthesesysteme.

## Anhang A : Ablauf der Synthese anhand eines Beispiels

Die Messungen wurden an mehreren verschiedenen Beispielen durchgeführt. Zum einen wurden die bekannten Beispiele 'Differential Equasion Solver' und 'Elliptical Wave Filter' analysiert. Des weiteren wurden drei Beispiele selber generiert. In den Beispielen wurde darauf Wert gelegt, daß zum einen Schleifen als auch bedingte Anweisungen Verwendung finden, wobei die angeführten Beispiele keine reale Anwendung haben. Da es im Rahmen der Untersuchungen nicht darauf ankam, daß alle in VHDL vorkommenden Konstrukte synthetisiert werden können wurde ein sehr einfacher experimenteller Compiler für die Erzeugung des Zwischenformates implementiert. Teilweise wurde das Zwischenformat manuell modifiziert da die Grundlage des Synthesesystems das Zwischenformat und nicht VHDL ist. Des weiteren wurden Bauteilbibliotheken ebenfalls manuell generiert, die aber realen Bauteilen und ihren Parametern entsprechen. Um die Ergebnisse zu verifizieren wurde eine VHDL-Strukturbeschreibung erzeugt und anhand von Simulationen mit der VHDL-Verhaltensbeschreibung verglichen. In diesem Anhang soll dieser Ablauf nochmal anhand des 'Elliptical Wave Filter' dargestellt werden. Das hier dargestellte VHDL kann manchmal unübersichtlich sein, da es maschinell erzeugt wurde. Als erstes ist das Zwischenformat welches aus 35 elementaren Befehlen besteht und in einer Datei abgelegt ist dargestellt. Im ersten Befehl werden alle Eingänge definiert.

*(in,{},{v1,v2,v3,v4,v5,v6,v7,v8})*
*(+,{v1,v2},{y1})*
*(+,{y1,v3},{y2})*
*(+,{v6,v8},{y4})*
*(+,{y2,v5},{y3})*
*(+,{y3,y4},{y5})*
*(*,{zwo,y5},{y6})*
*(+,{y6,y2},{y7})*
*(+,{y6,y4},{y8})*
*(+,{y2,y7},{y10})*
*(+,{y5,y7},{y9})*
*(*,{y10,zwo},{y11})*
*(+,{y4,y8},{y12})*
*(*,{y12,zwo},{y13})*
*(+,{y1,y11},{y14})*
*(+,{y14,y7},{y15})*
*(+,{y13,v8},{y16})*
*(+,{y16,y8},{y17})*
*(+,{y1,y14},{y18})*
*(+,{y16,v8},{y19})*
*(*,{zwo,y18},{y20})*
*(+,{v4,y15},{y21})*
*(+,{v6,y17},{y22})*
*(*,{zwo,y19},{y23})*
*(+,{y16,y23},{w7})*
*(+,{y20,v1},{y24})*
*(+,{y24,y14},{w1})*
*(*,{zwo,y21},{y25})*
*(+,{y25,v4},{w3})*
*(+,{w3,y21},{w2})*
*(+,{y9,y8},{w4})*

*(*,{zwo,y22},{y26})*
*(+,{y26,v7},{w6})*
*(+,{y22,w6},{w5})*
*(out,{w1,w2,w3,w4,w5,w6,w7})*

Des weiteren sind die zur Verfügung stehenden Bauteile ebenfalls in einer Datei in einfachem ASCII-Format abgelegt.

*Add2* -- Name des Bauteils
*1* -- Platzbedarf des Bauteils
*+(1,1,1)* -- Das Bauteil kann die Funktion + und es gilt $T_1(+) = 1$, $T_2(+) = 1$, $T_3(+) = 1$

*IN*
*1*
*in(1,1,1)*

*Add1*
*1*
*+(1,1,1)*

*MU1*
*4*
**(2,2,2)*

*Add2*
*1*
*+(1,1,1)*

*Add2*
*1*
*+(1,1,1)*

*Add1*
*1*
*+(1,1,1)*

*MU3*
*4*
**(2,2,2)*

*OUT*
*1*
*out(1,1,1)*

Die Bauteilliste stellt das interne Format dar, die mit '--' abgesetzten Kommentare sind intern nicht vorhanden sondern nur für eine anschaulichere Darstellung eingefügt.
Das Assignment als die Zuweisung der Befehle zu Bauteilen geschieht im nächsten Schritt. In den schraffierten Klammern unter jedem Bauteil sind die Nummern der Befehle dargestellt, die in diesem Bauteil zugewiesen sind.

*Add2*
*1*
*Funktion: +(1,1,1)*

*{ 2 , 3 , 14 , 24 , 30 , 32 }*

*IN*
*1*
*Funktion: in(1,1,1)*

*{ 0 }*

*Add1*
*1*
*Funktion: +(1,1,1)*

*{ 1 , 7 , 16 , 21 , 25 }*

*MU1*
*4*
*Funktion: *(2,2,2)*

*{ 6 , 11 , 13 , 20 , 27 , 31 }*

*Add2*
*1*
*Funktion: +(1,1,1)*

*{ 4 , 12 , 17 , 26 , 28 }*

*Add2*
*1*
*Funktion: +(1,1,1)*

*{ 8 , 15 , 19 , 22 , 33 }*

*Add1*
*1*
*Funktion: +(1,1,1)*

*{ 9 }*

*MU3*
*4*
*Funktion: *(2,2,2)*

*{ 23 }*

*OUT*
*1*
*Funktion: out(1,1,1)*

*{ 34 }*

*Add2*
*1*
*Funktion: +(1,1,1)*

*{ 5 , 10 , 18 , 29 }*

Nach dem Assignment wird das Scheduling durchgeführt, welches jeden Befehl einem Kontrollschritt zuweist.

*STEP[0] : 0 ..*
*STEP[1] : 1 ..*
*STEP[2] : 2 ..*
*STEP[3] : 3 .. 4 ..*
*STEP[4] : 5 ..*
*STEP[5] : 6 ..*
*STEP[6] :*
*STEP[7] : 7 .. 8 ..*
*STEP[8] : 9 .. 10 .. 12 ..*
*STEP[9] : 11 ..*
*STEP[10] :*
*STEP[11] : 13 .. 14 ..*
*STEP[12] : 15 .. 18 ..*
*STEP[13] : 16 .. 20 ..*
*STEP[14] : 17 .. 19 .. 21 ..*
*STEP[15] : 22 .. 23 .. 25 .. 27 ..*
*STEP[16] : 26 ..*
*STEP[17] : 24 .. 28 .. 31 ..*
*STEP[18] : 29 .. 30 ..*
*STEP[19] : 32 ..*
*STEP[20] : 33 ..*
*STEP[21] : 34 ..*

Mit diesen Berechnungen kann eine VHDL Strukturbeschreibung erzeugt werden.

```
ENTITY Elliptical IS
PORT (IN_V0,IN_V2,IN_V3,IN_V7,IN_V8,IN_V9,IN_V10,IN_V14 : in INTEGER;
OUT_V0,OUT_V2,OUT_V3,OUT_V4,OUT_V5,OUT_V8,OUT_V9 : out INTEGER;
 C : in Bit;
D_0,D_1,D_2,D_3,D_4,D_5,D_6,D_7,D_8,D_9,D_10,D_11,D_12,D_13,D_14 : in Bit;
SV_0 : in Bit_Vector(0 to 2);
SV_2 : in Bit_Vector(0 to 2);
SV_3 : in Bit_Vector(0 to 1);
SV_4 : in Bit_Vector(0 to 0);
SV_5 : in Bit_Vector(0 to 0);
```

```
SV_7 : in Bit_Vector(0 to 0);
SV_8 : in Bit_Vector(0 to 0);
SV_9 : in Bit_Vector(0 to 0);
SA_2 : in Bit_Vector(0 to 2);
SA_4 : in Bit_Vector(0 to 2);
SA_5 : in Bit_Vector(0 to 2);
SA_7 : in Bit_Vector(0 to 2);
SA_8 : in Bit_Vector(0 to 2);
SA_12 : in Bit_Vector(0 to 1));
END Elliptical;
ARCHITECTURE Arch OF Elliptical IS

COMPONENT REG
PORT (I: INTEGER; D,C : in Bit; O: out INTEGER);
END COMPONENT;
for all: REG
        use entity work.REG(R);
COMPONENT MULTI_2
PORT(I0,I1 : in INTEGER;
 S : in Bit_Vector(0 to 0);  O: out INTEGER);
END COMPONENT;
for all: MULTI_2
        use entity work.MULTI_2(M2);

COMPONENT MULTI_4
PORT(I0,I1,I2,I3 : in INTEGER;
 S : in Bit_Vector(0 to 1);  O: out INTEGER);
END COMPONENT;
for all: MULTI_4
        use entity work.MULTI_4(M4);

COMPONENT MULTI_5
PORT(I0,I1,I2,I3,I4 : in INTEGER;
 S : in Bit_Vector(0 to 2);  O: out INTEGER);
END COMPONENT;
for all: MULTI_5
        use entity work.MULTI_5(M5);

COMPONENT MULTI_6
PORT(I0,I1,I2,I3,I4,I5 : in INTEGER;
 S : in Bit_Vector(0 to 2);  O: out INTEGER);
END COMPONENT;
for all: MULTI_6
        use entity work.MULTI_6(M6);

COMPONENT MULTI_7
PORT(I0,I1,I2,I3,I4,I5,I6 : in INTEGER;
 S : in Bit_Vector(0 to 2);  O: out INTEGER);
END COMPONENT;
for all: MULTI_7
```

```
        use entity work.MULTI_7(M7);

COMPONENT Add2
PORT(I0,I1 : in INTEGER;
O0 : out INTEGER );
END COMPONENT;
for all: Add2
        use entity work.Add2(ARCH_Add2);

COMPONENT Add1
PORT(I0,I1 : in INTEGER;
O0 : out INTEGER );
END COMPONENT;
for all: Add1
        use entity work.Add1(ARCH_Add1);

COMPONENT MU1
PORT(I0,I1 : in INTEGER;
O0 : out INTEGER );
END COMPONENT;
for all: MU1
        use entity work.MU1(ARCH_MU1);

COMPONENT MU3
PORT(I0,I1 : in INTEGER;
O0 : out INTEGER );
END COMPONENT;
for all: MU3
        use entity work.MU3(ARCH_MU3);

SIGNAL OVAR_0_O : INTEGER;
SIGNAL M_REG_O_0 : INTEGER;
SIGNAL OVAR_1_O : INTEGER;
SIGNAL OVAR_2_O : INTEGER;
SIGNAL M_REG_O_2 : INTEGER;
SIGNAL OVAR_3_O : INTEGER;
SIGNAL M_REG_O_3 : INTEGER;
SIGNAL OVAR_4_O : INTEGER;
SIGNAL M_REG_O_4 : INTEGER;
SIGNAL OVAR_5_O : INTEGER;
SIGNAL M_REG_O_5 : INTEGER;
SIGNAL OVAR_6_O : INTEGER;
SIGNAL OVAR_7_O : INTEGER;
SIGNAL M_REG_O_7 : INTEGER;
SIGNAL OVAR_8_O : INTEGER;
SIGNAL M_REG_O_8 : INTEGER;
SIGNAL OVAR_9_O : INTEGER;
SIGNAL M_REG_O_9 : INTEGER;
SIGNAL OVAR_10_O : INTEGER;
SIGNAL OVAR_11_O : INTEGER;
```

```
SIGNAL OVAR_12_O : INTEGER;
SIGNAL OVAR_13_O : INTEGER;
SIGNAL OVAR_14_O : INTEGER;
SIGNAL M_ALU2_I0_O : INTEGER;
SIGNAL M_ALU2_I1_O : INTEGER;
SIGNAL OALU_2_O0 : INTEGER;
SIGNAL OALU_3_O0 : INTEGER;
SIGNAL M_ALU4_I0_O : INTEGER;
SIGNAL M_ALU4_I1_O : INTEGER;
SIGNAL OALU_4_O0 : INTEGER;
SIGNAL M_ALU5_I0_O : INTEGER;
SIGNAL M_ALU5_I1_O : INTEGER;
SIGNAL OALU_5_O0 : INTEGER;
SIGNAL M_ALU7_I0_O : INTEGER;
SIGNAL M_ALU7_I1_O : INTEGER;
SIGNAL OALU_7_O0 : INTEGER;
SIGNAL M_ALU8_I0_O : INTEGER;
SIGNAL M_ALU8_I1_O : INTEGER;
SIGNAL OALU_8_O0 : INTEGER;
SIGNAL OALU_9_O0 : INTEGER;
SIGNAL OALU_10_O0 : INTEGER;
SIGNAL OALU_11_O0 : INTEGER;
SIGNAL M_ALU12_I0_O : INTEGER;
SIGNAL M_ALU12_I1_O : INTEGER;
SIGNAL OALU_12_O0 : INTEGER;
BEGIN

MULTI_V0: MULTI_7
PORT MAP(OALU_2_O0, IN_V0, OALU_4_O0, OALU_5_O0, OALU_7_O0, OALU_9_O0,
OALU_10_O0, SV_0, M_REG_O_0);

REG_0: REG
PORT MAP(M_REG_O_0, D_0, C, OVAR_0_O);

REG_1: REG
PORT MAP(OALU_11_O0, D_1, C, OVAR_1_O);

MULTI_V2: MULTI_5
PORT MAP(IN_V2, OALU_5_O0, OALU_7_O0, OALU_8_O0, OALU_12_O0, SV_2,
M_REG_O_2);

REG_2: REG
PORT MAP(M_REG_O_2, D_2, C, OVAR_2_O);

MULTI_V3: MULTI_4
PORT MAP(OALU_2_O0, IN_V3, OALU_8_O0, OALU_12_O0, SV_3, M_REG_O_3);

REG_3: REG
PORT MAP(M_REG_O_3, D_3, C, OVAR_3_O);
```

*MULTI_V4: MULTI_2*
*PORT MAP(OALU_2_O0, OALU_7_O0, SV_4, M_REG_O_4);*

*REG_4: REG*
*PORT MAP(M_REG_O_4, D_4, C, OVAR_4_O);*

*MULTI_V5: MULTI_2*
*PORT MAP(OALU_4_O0, OALU_8_O0, SV_5, M_REG_O_5);*

*REG_5: REG*
*PORT MAP(M_REG_O_5, D_5, C, OVAR_5_O);*

*REG_6: REG*
*PORT MAP(OALU_4_O0, D_6, C, OVAR_6_O);*

*MULTI_V7: MULTI_2*
*PORT MAP(IN_V7, OALU_8_O0, SV_7, M_REG_O_7);*

*REG_7: REG*
*PORT MAP(M_REG_O_7, D_7, C, OVAR_7_O);*

*MULTI_V8: MULTI_2*
*PORT MAP(OALU_2_O0, IN_V8, SV_8, M_REG_O_8);*

*REG_8: REG*
*PORT MAP(M_REG_O_8, D_8, C, OVAR_8_O);*

*MULTI_V9: MULTI_2*
*PORT MAP(IN_V9, OALU_7_O0, SV_9, M_REG_O_9);*

*REG_9: REG*
*PORT MAP(M_REG_O_9, D_9, C, OVAR_9_O);*

*REG_10: REG*
*PORT MAP(IN_V10, D_10, C, OVAR_10_O);*

*REG_11: REG*
*PORT MAP(OALU_12_O0, D_11, C, OVAR_11_O);*

*REG_12: REG*
*PORT MAP(OALU_8_O0, D_12, C, OVAR_12_O);*

*REG_13: REG*
*PORT MAP(, D_13, C, OVAR_13_O);*

*REG_14: REG*
*PORT MAP(IN_V14, D_14, C, OVAR_14_O);*

*MULTI_A2_I0 : MULTI_6*
*PORT MAP(OVAR_6_O, OVAR_8_O, OVAR_6_O, OVAR_5_O, OVAR_11_O, OVAR_0_O,*

*SA_2, M_ALU2_I0_O);*

*MULTI_A2_I1 : MULTI_6*
*PORT MAP(OVAR_2_O, OVAR_7_O, OVAR_0_O, OVAR_0_O, OVAR_12_O, OVAR_14_O, SA_2, M_ALU2_I1_O);*

*ALU2: Add2*
*PORT MAP(M_ALU2_I0_O,M_ALU2_I1_O, OALU_2_O0);*

*MULTI_A4_I0 : MULTI_5*
*PORT MAP(OVAR_9_O, OVAR_0_O, OVAR_0_O, OVAR_10_O, OVAR_2_O, SA_4, M_ALU4_I0_O);*

*MULTI_A4_I1 : MULTI_5*
*PORT MAP(OVAR_0_O, OVAR_4_O, OVAR_7_O, OVAR_2_O, OVAR_9_O, SA_4, M_ALU4_I1_O);*

*ALU4: Add1*
*PORT MAP(M_ALU4_I0_O,M_ALU4_I1_O, OALU_4_O0);*

*MULTI_A5_I0 : MULTI_6*
*PORT MAP(OVAR_13_O, OVAR_0_O, OVAR_2_O, OVAR_13_O, OVAR_13_O, OVAR_13_O, SA_5, M_ALU5_I0_O);*

*MULTI_A5_I1 : MULTI_6*
*PORT MAP(OVAR_2_O, OVAR_13_O, OVAR_13_O, OVAR_3_O, OVAR_6_O, OVAR_7_O, SA_5, M_ALU5_I1_O);*

*ALU5: MU1*
*PORT MAP(M_ALU5_I0_O,M_ALU5_I1_O, OALU_5_O0);*

*MULTI_A7_I0 : MULTI_5*
*PORT MAP(OVAR_4_O, OVAR_3_O, OVAR_5_O, OVAR_0_O, OVAR_2_O, SA_7, M_ALU7_I0_O);*

*MULTI_A7_I1 : MULTI_5*
*PORT MAP(OVAR_3_O, OVAR_12_O, OVAR_12_O, OVAR_4_O, OVAR_10_O, SA_7, M_ALU7_I1_O);*

*ALU7: Add2*
*PORT MAP(M_ALU7_I0_O,M_ALU7_I1_O, OALU_7_O0);*

*MULTI_A8_I0 : MULTI_5*
*PORT MAP(OVAR_0_O, OVAR_4_O, OVAR_5_O, OVAR_8_O, OVAR_7_O, SA_8, M_ALU8_I0_O);*

*MULTI_A8_I1 : MULTI_5*
*PORT MAP(OVAR_3_O, OVAR_5_O, OVAR_7_O, OVAR_0_O, OVAR_0_O, SA_8, M_ALU8_I1_O);*

```
ALU8: Add2
PORT MAP(M_ALU8_I0_O,M_ALU8_I1_O, OALU_8_O0);

ALU9: Add1
PORT MAP(OVAR_4_O, OVAR_5_O, OALU_9_O0);

ALU10: MU3
PORT MAP(OVAR_13_O, OVAR_3_O, OALU_10_O0);

OUT_V4 <= OVAR_4_O;
OUT_V2 <= OVAR_2_O;
OUT_V9 <= OVAR_9_O;
OUT_V3 <= OVAR_3_O;
OUT_V5 <= OVAR_5_O;
OUT_V0 <= OVAR_0_O;
OUT_V8 <= OVAR_8_O;

MULTI_A12_I0 : MULTI_4
PORT MAP(OVAR_0_O, OVAR_2_O, OVAR_6_O, OVAR_9_O, SA_12, M_ALU12_I0_O);

MULTI_A12_I1 : MULTI_4
PORT MAP(OVAR_3_O, OVAR_5_O, OVAR_4_O, OVAR_6_O, SA_12, M_ALU12_I1_O);

ALU12: Add2
PORT MAP(M_ALU12_I0_O,M_ALU12_I1_O, OALU_12_O0);

END Arch;

package ROMs is
constant ROM_WIDTH: INTEGER := 45;
subtype ROM_WORD is BIT_VECTOR (1 TO ROM_WIDTH);
subtype ROM_RANGE is INTEGER range 0 to 21;
type ROM_TABLE is array (0 to 21) of ROM_WORD;
 constant ROM: ROM_TABLE := ROM_TABLE'(
ROM_WORD'("101100011110001001000010001000000000000000000"),
ROM_WORD'("000000100000000000000000000000000000000000000"),
ROM_WORD'("000010000000000000000000000000000000000000000"),
ROM_WORD'("100100000000001000000000000001000000000000000"),
ROM_WORD'("001000000000000001000000000000000000000000000"),
ROM_WORD'("000000000000000000000000000000000000000000000"),
ROM_WORD'("100000000000000110000000000000000000000000000"),
ROM_WORD'("000001000000100000000000000000000100000000000"),
ROM_WORD'("101000000010001010100000000000000000100001"),
ROM_WORD'("000000000000000000000000000000000000100000000"),
ROM_WORD'("100000000000000110000000000000000000100000000"),
ROM_WORD'("000010000000000000000000000001000001000000000"),
ROM_WORD'("101100000000000011011110000000000001000000110"),
ROM_WORD'("000001000000000000000000000000001001100000000"),
ROM_WORD'("101100100000000100001100000000001101101001000"),
```

```
ROM_WORD'("10000001000000001000000001000001001000000001100"),
ROM_WORD'("10101000000000011000100100000000001000110000"),
ROM_WORD'("00000000110000000000000000001011000101100000000"),
ROM_WORD'("10110000000000001110000000001000001010000000011"),
ROM_WORD'("10000000000000000000000000001010000000000000000"),
ROM_WORD'("00000100000000000000000001000000000000000010000"),
ROM_WORD'("01000000000000000000000000000000000000000000000"));
end ROMs;
use work.ROMs.all;
entity CO_E is
 port (
  STEP_I : in INTEGER;
  CLOCK: in BIT;
  RESET: in Bit;
  WOUT : out Bit_Vector(1 to 45);
  STEP_O : out INTEGER);
end CO_E;

ARCHITECTURE CO_A OF CO_E IS
SIGNAL STEP: ROM_RANGE;
BEGIN
 TIMESTEP_COUNTER: process
 BEGIN
  wait until CLOCK'event and CLOCK = '1';
  if RESET = '1' then
   STEP <= ROM_RANGE'low;
  elsif STEP = ROM_RANGE'high then
   STEP <= 0;
  else
   STEP <= STEP_I;
  END if;
 END process TIMESTEP_COUNTER;
STEP_O <= STEP + 1;
WOUT <= ROM(STEP);
END CO_A;

ENTITY Gesamt IS
PORT (
CLOCK : in Bit;
RESET : in Bit;
I_V0,I_V2,I_V3,I_V7,I_V8,I_V9,I_V10,I_V14 : in INTEGER;
O_V0,O_V2,O_V3,O_V4,O_V5,O_V8,O_V9 : out INTEGER
);
END Gesamt;


ARCHITECTURE Archi OF Gesamt IS

COMPONENT Elliptical
 PORT(IN_V0,IN_V2,IN_V3,IN_V7,IN_V8,IN_V9,IN_V10,IN_V14 : in INTEGER;
```

```
OUT_V0,OUT_V2,OUT_V3,OUT_V4,OUT_V5,OUT_V8,OUT_V9 : out INTEGER;
 C : in Bit;
D_0,D_1,D_2,D_3,D_4,D_5,D_6,D_7,D_8,D_9,D_10,D_11,D_12,D_13,D_14 : in Bit;
SV_0 : in Bit_Vector(0 to 2);
SV_2 : in Bit_Vector(0 to 2);
SV_3 : in Bit_Vector(0 to 1);
SV_4 : in Bit_Vector(0 to 0);
SV_5 : in Bit_Vector(0 to 0);
SV_7 : in Bit_Vector(0 to 0);
SV_8 : in Bit_Vector(0 to 0);
SV_9 : in Bit_Vector(0 to 0);
SA_2 : in Bit_Vector(0 to 2);
SA_4 : in Bit_Vector(0 to 2);
SA_5 : in Bit_Vector(0 to 2);
SA_7 : in Bit_Vector(0 to 2);
SA_8 : in Bit_Vector(0 to 2);
SA_12 : in Bit_Vector(0 to 1));
END COMPONENT;
for all: Elliptical
  use entity work.Elliptical(Arch);

COMPONENT CO_E
 PORT (
  STEP_I : in INTEGER;
  CLOCK: in BIT;
  RESET: in Bit;
  WOUT : out Bit_Vector(1 to 45);
  STEP_O : out INTEGER);
end COMPONENT;

for all: CO_E
  use entity work.CO_E(CO_A);

SIGNAL ST : Bit_Vector(1 to 45);
SIGNAL S_IN : INTEGER;
SIGNAL S_OUT : INTEGER;
BEGIN
Elliptical_I : Elliptical
PORT MAP
(
I_V0,I_V2,I_V3,I_V7,I_V8,I_V9,I_V10,I_V14
,O_V0,O_V2,O_V3,O_V4,O_V5,O_V8,O_V9
, CLOCK
,ST(1),ST(2),ST(3),ST(4),ST(5),ST(6),ST(7),ST(8),ST(9),ST(10),ST(11),ST(12),ST(13),ST(14),
ST(15)
,ST(16 TO 18),ST(19 TO 21),ST(22 TO 23),ST(24 TO 24),ST(25 TO 25),ST(26 TO 26),ST(27
TO 27),ST(28 TO 28)
,ST(29 TO 31),ST(32 TO 34),ST(35 TO 37),ST(38 TO 40),ST(41 TO 43),ST(44 TO 45)
);
```

*CO_E_I : CO_E*
*PORT MAP*
*(*
*S_IN,*
*CLOCK, RESET,*
*ST,*
*S_OUT);*
*END Archi;*

Die erzeugte Schaltung besteht aus einer übergeordneten Entity 'Gesamt', diese wiederum enthält als Komponenten die ebenfalls erzeugte Entity 'CO_E' welche den Kontroller darstellt und die Entity 'Elliptical' welche die Recheneinheit darstellt. In jedem Schritt wird für eine so erzeugte Schaltung die Bewertung berechnet. Dazu wird der Registerbedarf, der Multiplexerbedarf, der Zeitbedarf und der Ressourcenbedarf aus dem erzeugten Ergebnis gewonnen. Anschließend kann dann entsprechend des jeweiligen Optimierungsalgorithmus eine Auswahl des besten Ergebnis durchgeführt werden. Im Rahmen dieser Arbeit wurde keine Optimierung des Kontrollers betrachtet, deshalb ist dieser als einfaches ROM dargestellt. Sicherlich sind hier PLA-Lösungen oft sinnvoller und Platzsparender, wobei bekannte Optimierungsalgorithmen für diesen Bereich eingesetzt werden können.

## Anhang B : Der Quelltext des Synthesesystems

In diesem Anhang ist der Quellcode des entwickelten Synthesewerkzeugs dargestellt. Es handelt sich im wesentlichen um ein experimentelles System. Es wurden nur die Merkmale des Synthesesystems implementiert, die für eine Abgrenzung gegenüber anderen Systemen von Bedeutung sind. Daher folgt, daß nicht alle beschriebenen Optimierungsmöglichkeiten implementiert wurden. In den wenigen Fällen in denen dies geschehen ist, ist es aber relativ einfach entsprechende Erweiterungen durchzuführen. Die Struktur des Synthesesystemes und die Modulbeschreibung wurde schon in Kapitel 12 geliefert, so daß in diesem Anhang im wesentlichen nur die Quellcodes abgebildet sind, also die *.h Datei und die *.cc Dateien.

**Befehl**

```
#include<iostream.h>

class Befehl  // Dies ist ein Befehl
 {
 private:
  char* f;     // Mit der Funktion f
  int anzi;
  char* vi[17];
  int anzo;
  char* vo[17];
  int s;     // Steuerschritt in dem dieser Befehl ausgefuehrt wird
  int la;
  char* label;
  int m;   // Marke

 public:
 Befehl();

 void put_in_anzahl(int);  // Anzahl Eingaenge
 int get_in_anzahl();
 void put_out_anzahl(int);  // Anzahl Ausgaenge
 int get_out_anzahl();

 void put_input(char*);     // Eingangsvariable eingeben
 char* get_input(int);

 void put_output(char*);    // Ausgangsvariable
 char* get_output();         // erste Ausgangsvariable
 char* get_output(int);     // ite.
 void put_function(char*);   // Funktion eingeben
 char* get_function();

 void put_steuer(int);
 int get_steuer();

 void put_label(char *);
 int is_label();
 char* get_label();
 void kill_label();

 void put_marke(int);
 int get_marke();

 void input();
 void output();
```

```
 friend istream& operator>>(istream& is, Befehl &b)
  {
   b.input();
   return is;
  };
 friend ostream& operator<<(ostream& os, Befehl &b)
  {
   b.output();
   return os;
  };
 }; // end Befehl.h

Befehl::Befehl()
 {
  anzi = 0;
  anzo = 0;
  s = 0;
  la = 0;
  m = 0;
 };

void Befehl::put_in_anzahl(int i)
   {
    anzi = i;
   };

int Befehl::get_in_anzahl()
   {
    return anzi;
   };


void Befehl::put_out_anzahl(int i)
   {
    anzo = i;
   };

int Befehl::get_out_anzahl()
   {
    return anzo;
   };

void Befehl::put_input(char* v)
   {
    vi[anzi] = (char *) malloc(27);
    sprintf(vi[anzi],"%s",v);
    anzi++;
   };

char* Befehl::get_input(int i)
   {
    return vi[i];
   };

void Befehl::put_output(char* v)
   {
    vo[anzo] = (char *) malloc(27);
    sprintf(vo[anzo],"%s",v);
```

```
   anzo++;
  };

char* Befehl::get_output()
  {
   return vo[0];
  };

char* Befehl::get_output(int i)
 {
  return vo[i];
 };

void Befehl::put_function(char* h)
  {
   f = (char *) malloc(27);
   sprintf(f,"%s",h);
  };

char* Befehl::get_function()
  {
   return f;
  };

void Befehl::put_steuer(int i)
 {
  s = i;
 };

int Befehl::get_steuer()
 {
  return s;
 };

void Befehl::put_label(char* lab)
 {
  la = 1;
  label = (char*) malloc(7);
  sprintf(label,"%s",lab);
 };

int Befehl::is_label()
 {
  return la;
 };

char* Befehl::get_label()
 {
  return label;
 };

void Befehl::kill_label()
 {
  la = 0;
 };

void Befehl::put_marke(int i)
 {
```

```
 m = i;
};
int Befehl::get_marke()
{
 return m;
};
void Befehl::input()
 {
  char st[99];  // = (char*) malloc(35);
  char* sh = (char*) malloc(99);
  cin >> st;
  cout << st << endl;
  int i;
  i=0;
  put_in_anzahl(0);
  put_out_anzahl(0);
  kill_label();
  sprintf(sh,"");

//    while (st[i] == ' ') i++;

    if ((st[i] != '(') && (i<strlen(st)) )   // dann Label ..
     {
      while( (st[i] != ':') && (i<strlen(st)) )
       {
        sprintf(sh,"%s%c",sh,st[i]);
        i++;
       };
      put_label(sh);  // Das Label
      sprintf(sh,"");
     }
    else kill_label();

    if (st[i] == ':') i++;
    if (st[i] == '(') i++;
    while( (st[i] != ',') && (i<strlen(st)) )
     {
      sprintf(sh,"%s%c",sh,st[i]);
      i++;
     };
    put_function(sh);  // Die Funktion

    sprintf(sh,"");

    i++;   // Das Komma
   if (st[i+1] != '}')
   do{
    i++;
    while ((st[i] != '}') && (st[i] != ',') && (i<strlen(st)) )
     {
      sprintf(sh,"%s%c",sh,st[i]);
      i++;
     };
     put_input(sh);  // Die Funktion

     sprintf(sh,"");
    } while ((st[i] != '}') && (i<strlen(st)));  //Die Inputs
   else i++;
```

```
  i++; // Komma
  i++; // Klammer auf
   if (st[i+1] != '}')
   do{
    i++;
    while ((st[i] != '}') && (st[i] != ',') && (i<strlen(st)) )
     {
      sprintf(sh,"%s%c",sh,st[i]);
      i++;
     };
    put_output(sh);  // Die Funktion
     sprintf(sh,"");
    } while ((st[i] != '}') && (i<strlen(st)));  //Die Outputs
   else i++;

 };

void Befehl::output()
  {   if (is_label())
       cout << get_label() << ":";
      if (get_out_anzahl() > 0)
       {
        for (int o=0; o < get_out_anzahl() ; o++)
          cout << get_output(o) << " ";
       };
      cout << " = " ;
      if (get_in_anzahl() > 0)
       {
        for (int i=0; i< get_in_anzahl() - 1; i++)
         {
          cout << get_input(i) << " ";
          cout << get_function() << " ";
         }

        cout << get_input(get_in_anzahl() - 1) << " ";
       };
      cout << " ..... " << get_marke();
      cout << endl;
  };
```

**Befehls_Liste**

```
#include <fstream.h>
#include <Befehl.h>
#include <iostream.h>
class Befehls_Liste
 {
  private:
  int anz;
  int DA_Matrix[99][99];
  public:
  Befehl Liste[99];
  Befehls_Liste();
  void put_Anzahl(int);  // Anzahl Befehle eingeben
  int get_Anzahl();      // Anzahl Befehle ausgeben
  void put_Befehl(Befehl);   // Befehl zufuegen (anz wird automatisch erhoeht)
  Befehl get_Befehl(int);    // den n ten Befehl rausholen
  void put_steuer(int,int); //  Befehl an Steuer (b,s)
  int get_steuer(int);     // Steuerschritt von BEfehl
```

```
 int all_da(int, int);     //
 int trans_all(int, int);
 void init_all();   // Init matrix
 void init_Tda();  // nur fuer Datenab initialisieren !!
 int aab(int, int);
 int ada(int, int);
 int da(int, int);
 int abh(int);
 void input();
 void output();
 friend istream& operator>>(istream& is, Befehls_Liste &r)
  {
   r.input();
   return is;
  };

 friend ostream& operator<<(ostream& os, Befehls_Liste &z)
  {
   z.output();
   return os;
  };
 int file_input(char*);
 int file_output(char*);
};  // end Befehls_Liste.h

Befehls_Liste::Befehls_Liste()
 {
  anz = 0;
 };

void Befehls_Liste::put_Anzahl(int i)
   {
    anz = i;
   };

int Befehls_Liste::get_Anzahl()
   {
    return anz;
   };


void Befehls_Liste::put_Befehl(Befehl b)
   {
    Liste[anz] = b;
    anz++;
   };

Befehl Befehls_Liste::get_Befehl(int i)
   {
    return Liste[i];
   };

void Befehls_Liste::put_steuer(int b,int s)
 {
  Liste[b].put_steuer(s);
 };

int Befehls_Liste::get_steuer(int b)
```

```
{
 return Liste[b].get_steuer();
};

int Befehls_Liste::all_da(int x, int y)   // x vor y !!!! nicht rekursiv !!!
  {
   if (y <= x) return 0;
   if (x>=anz) return 0;
   if (y>=anz) return 0;

   if (aab(x,y) == 1)
    return 1;    // xo = yo  aab

   if (ada(x,y) == 1)
    return 1;

   if (da(x,y) ==1)
    return 1;

   return 0;
  };

int Befehls_Liste::trans_all(int x,int y)
{
 if ((x< anz) && (y<anz))
  if (DA_Matrix[x][y] == 1)
   return 1;
 return 0;
};

void Befehls_Liste::init_all()
{

 for(int b= 0;b<anz;b++)
  Liste[b].put_steuer(0);

 for( int x = 0; x < anz; x++)
  for( int y = 0; y < anz; y++)
   DA_Matrix[x][y] = all_da(x,y);

for( int t = 0; t < get_Anzahl(); t++)
 for( int x = 0; x < anz; x++)
  {
   for( int y = 0; y < anz; y++)
    {
     for( int k=0; k<anz; k++)
      if ((DA_Matrix[x][k] == 1) && (DA_Matrix[k][y] == 1))
       {
        DA_Matrix[x][y] = 1;
        k = anz;
       };
    };  // for y
  }; // for x
}; // init_Matrix

void Befehls_Liste::init_Tda()
{
 for( int x = 0; x < anz; x++)
```

```
  for( int y = 0; y < anz; y++)
   DA_Matrix[x][y] = da(x,y);

 for( int t = 0; t < get_Anzahl(); t++)
  for( int x = 0; x < anz; x++)
   {
    for( int y = x+1; y < anz; y++)
     {
      for( int k=x+1; k<anz; k++)
       if ((DA_Matrix[x][k] == 1) && (DA_Matrix[k][y] == 1))
        {
         DA_Matrix[x][y] = 1;
         k = anz;
        };
     };  // for y
   }; // for x
 }; // init_Matrix nur fuer Datenabhaengigkeiten

int Befehls_Liste::aab(int x, int y)
   {
    if (y <= x) return 0;
    if (x>=anz) return 0;
    if (y>=anz) return 0;
    if (get_Befehl(x).get_out_anzahl() == 0) return 0;
    if (get_Befehl(y).get_out_anzahl() == 0) return 0;

    for(int i= 0; i< get_Befehl(x).get_out_anzahl();i++)
     for(int j=0; j< get_Befehl(y).get_out_anzahl();j++)
      if (strcmp(get_Befehl(x).get_output(i),get_Befehl(y).get_output(j)) == 0)
       return 1;    // xo = yo  aab
    return 0;
   };


int Befehls_Liste::ada(int x, int y)
   {
    if (y <= x) return 0;
    if (x>=anz) return 0;
    if (y>=anz) return 0;
    if (get_Befehl(x).get_in_anzahl() == 0) return 0;
    if (get_Befehl(y).get_out_anzahl() == 0) return 0;

    for(int i=0; i < get_Befehl(x).get_in_anzahl(); i++)
     for(int j=0 ; j < get_Befehl(y).get_out_anzahl();j++)
      if (strcmp(get_Befehl(x).get_input(i),get_Befehl(y).get_output(j)) == 0)
       return 1;      // xi = yo   ada
    return 0;
   };


int Befehls_Liste::da(int x, int y)
   {
    if (y <= x) return 0;
    if (x>=anz) return 0;
    if (y>=anz) return 0;
    if (get_Befehl(x).get_out_anzahl() == 0) return 0;
    if (get_Befehl(y).get_in_anzahl() == 0) return 0;
```

```
   for(int i=0; i < get_Befehl(y).get_in_anzahl(); i++)
    for(int j=0 ; j < get_Befehl(x).get_out_anzahl();j++)
     if (strcmp(get_Befehl(y).get_input(i),get_Befehl(x).get_output(j)) == 0)
      return 1;      // xo = yi    da

   return 0;
  };

int Befehls_Liste::abh(int x)   // gibt an ob ein Befehl von sich selbst abhaengig ist.
  {
   if (get_Befehl(x).get_in_anzahl() == 0) return 0;
   if (get_Befehl(x).get_out_anzahl() == 0) return 0;

   for(int i=0; i < get_Befehl(x).get_in_anzahl(); i++)
    for(int j=0 ; j < get_Befehl(x).get_out_anzahl();j++)
     if (strcmp(get_Befehl(x).get_input(i),get_Befehl(x).get_output(j)) == 0)
      return 1;      // xo = xi    -> da ada
  };


void Befehls_Liste::input()
 {
  Befehl b;

  int i;
  int j;
  char f;
  char l;

  put_Anzahl(0);
  cout << " Anzahl Befehle ?";
  cin >> i;
  for(int z = 0; z < i; z ++)
   {
    cin >> b;
    put_Befehl(b);   // zur Liste hinzufügen
   };
 };


void Befehls_Liste::output()
   {
    cout << "Die Befehlsliste" << endl;

    for(int i=0;i < get_Anzahl(); i++)
     {
      cout << "Befehl[" << i << "] :: ";
      cout << get_Befehl(i);
      cout << endl;
     };
   };

int Befehls_Liste::file_input(char* name)
 {
  char buf[99];
  int a;
  a = 0;
  for(int b= 0;b<anz;b++)
```

```
 Liste[b].put_steuer(0);

ifstream my_file(name);
if(!my_file)
 return 0;
put_Anzahl(0);

my_file.getline(buf,99, '\n');
for(int i=0; i< strlen(buf);i++)
 a= (int)buf[i] - 48 + (a*10);
cout << a << endl;
for(i=0; i< a;i++)
 {
  Befehl b;
  int j;
  j = 0;
  char* sh = (char*) malloc(99);
  my_file.getline(buf,99, '\n');
  cout << buf << "...." << endl;
  b.put_in_anzahl(0);
  b.put_out_anzahl(0);
  b.kill_label();
  sprintf(sh,"");

   if ((buf[j] != '(') && (j<strlen(buf)) )   // dann Label ..
    {
     while( (buf[j] != ':') && (j<strlen(buf)) )
      {
       sprintf(sh,"%s%c",sh,buf[j]);
       j++;
      };
     b.put_label(sh);  // Das Label
     sprintf(sh,"");
    }
   else b.kill_label();

   if (buf[j] == ':') j++;
   if (buf[j] == '(') j++;
   while( (buf[j] != ',') && (j<strlen(buf)) )
    {
     sprintf(sh,"%s%c",sh,buf[j]);
     j++;
    };
   b.put_function(sh);  // Die Funktion

   sprintf(sh,"");

   j++;   // Das Komma
  if (buf[j+1] != '}')
  do{
   j++;
   while ((buf[j] != '}') && (buf[j] != ',') && (j<strlen(buf)) )
    {
     sprintf(sh,"%s%c",sh,buf[j]);
     j++;
    };
    b.put_input(sh);  // Die Funktion
```

```
     sprintf(sh,"");
    } while ((buf[j] != '}') && (j<strlen(buf)));  //Die Inputs
   else j++;

  j++; // Komma
  j++; // Klammer auf

   if (buf[j+1] != '}')
   do{
    j++;
    while ((buf[j] != '}') && (buf[j] != ',') && (j<strlen(buf)) )
     {
      sprintf(sh,"%s%c",sh,buf[j]);
      j++;
     };
    b.put_output(sh);  // Die Funktion

     sprintf(sh,"");
    } while ((buf[j] != '}') && (j<strlen(buf)));  //Die Outputs
   else j++;

   put_Befehl(b);
  };

 return 1;
};

int Befehls_Liste::file_output(char * name)
 { // file schreiben
  ofstream my_file(name);
  if(!my_file) return 0;
  my_file << get_Anzahl() << "\n";
   for(int i=0;i < get_Anzahl(); i++)
    {

     if (get_Befehl(i).is_label())
       my_file << get_Befehl(i).get_label() << ":";
     my_file << "(" << get_Befehl(i).get_function() << ",";
     my_file << "{";

     if (get_Befehl(i).get_in_anzahl() > 0)   // input
       {
        for (int j=0; j< get_Befehl(i).get_in_anzahl() - 1; j++)
         {
          my_file << get_Befehl(i).get_input(j) << ",";
         };

        my_file << get_Befehl(i).get_input(get_Befehl(i).get_in_anzahl() - 1);
       };

     my_file << "},{";


     if (get_Befehl(i).get_out_anzahl() > 0)
       {
        for (int o=0; o < get_Befehl(i).get_out_anzahl() - 1 ; o++)
         my_file << get_Befehl(i).get_output(o) << ",";
```

```
        my_file << get_Befehl(i).get_output(get_Befehl(i).get_out_anzahl()- 1);

       }; // if
     my_file << "})\n";

    };  // for

 my_file.close();
 return 1;
};
```

**Bauteil**

```
#include<iostream.h>
#define MAX2 30

class Bauteil
 {

  private:
   char* name;
   int anz_func;
   int anz_eing;
   int anz_ausg;
   float Space;
   int Time1[MAX2];  // bzgl anz_func ..
   int Time2[MAX2];
   int Time3[MAX2];
   char* Func[MAX2];

  public:

   Bauteil();
   void put_name(char*);
   char* get_name();
   void put_space(float);
   float get_space();
   void put_anz_eing(int);
   int get_anz_eing();
   void put_anz_ausg(int);
   int get_anz_ausg();
   void add_function(char*,int,int,int);
   void put_anz_func(int);
   int get_anz_func();
   int get_Time1(char *);  // Zeit bzgl. Funktion
   int get_Time2(char *);
   int get_Time3(char *);
   int get_func_nr(char *);
   char* get_func(int);

 /* get_Time(int) == get_Time(get_func())  */

  // Ab hier Assignment ...

   int befehl_geht(Befehl);
   int befehl_geht(Befehls_Liste,int);  //Nr des Befehls in Liste
   int befehl_geht(char *);  // funktion
```

```
  void input();
  void output();
  friend ostream& operator<<(ostream& os, Bauteil &b)
   {
    b.output();
    return os;
   };
  friend istream& operator>>(istream& is, Bauteil &b)
   {
    b.input();
    return is;
   };
 // Assignment machen wir woanders
};  // end Bauteil.h

Bauteil::Bauteil()
 {
  anz_func = 0;
  anz_eing = 0;
  Space = 0;
  anz_ausg = 1;
 };

void Bauteil::put_name(char* n)
 {
  name = (char *) malloc(25);
  strcpy(name,n);
 };

char* Bauteil::get_name()
 {
  return name;
 };

void Bauteil::put_space(float s)
 {
  Space =  s;
 };

float Bauteil::get_space()
 {
  return Space;
 };

void Bauteil::put_anz_eing(int a)
 {
  anz_eing = a;
 };

int Bauteil::get_anz_eing()
 {
  return anz_eing;
 };

void Bauteil::put_anz_ausg(int a)
 {
  anz_ausg = a;
 };
```

```
int Bauteil::get_anz_ausg()
 {
  return anz_ausg;
 };

void Bauteil::add_function(char * f, int t1, int t2, int t3)
 {
  Func[anz_func] = (char *) malloc(20);
  strcpy(Func[anz_func],f);
  Time1[anz_func] = t1;
  Time2[anz_func] = t2;
  Time3[anz_func] = t3;
  anz_func++;
 };

void Bauteil::put_anz_func(int a)
 {
  anz_func = 0;
 };

int Bauteil::get_anz_func()
 {
  return anz_func;
 };

int Bauteil::get_Time1(char* f)
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(f,Func[i]) == 0)
    return Time1[i];
 };

int Bauteil::get_Time2(char* f)
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(f,Func[i]) == 0)
    return Time2[i];
 };

int Bauteil::get_Time3(char* f)
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(f,Func[i]) == 0)
    return Time3[i];
 };

int Bauteil::get_func_nr(char* f)  // nur nach befehl_geht
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(f,Func[i]) == 0)
    return i;
 };

char* Bauteil::get_func(int i)
 {
  return Func[i];
 };
```

```
int Bauteil::befehl_geht(Befehl b)
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(b.get_function(),Func[i]) == 0)
    return 1;
  return 0;
 };

int Bauteil::befehl_geht(Befehls_Liste BL,int j)
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(BL.get_Befehl(j).get_function(),Func[i]) == 0)
    return 1;
  return 0;
 };

int Bauteil::befehl_geht(char* f)
 {
  for(int i=0; i< anz_func; i++)
   if (strcmp(f,Func[i]) == 0)
    return 1;
  return 0;
 };

void Bauteil::input()
 {
  int a;
  int t1;
  int t2;
  int t3;
  char f[10];
  char n[25];
  float ar;
  int ai;
  cout << "Name" ;
  cin >> n;
  cout << "Area " ;
  cin >> ar;
  cout << "ANzahl Eingaenge ?";
  cin >> ai;
  put_name(n);
  put_space(ar);
  put_anz_eing(ai);
  put_anz_ausg(1);
  cout << "Anzahl der Funktionen ? ";
  cin >> a;

  anz_func = 0;
  for (int i= 0; i< a ; i++)
   {
    cout << " Funktion: ";
    cin >> f;
    cout << " Time 1: ";
    cin >> t1;
    cout << " Time 2: ";
    cin >> t2;
    cout << " Time 3: ";
```

```
   cin >> t3;
   add_function(f,t1,t2,t3);
  };
 };


void Bauteil::output()
 {
  cout << get_name() << endl;
  cout << get_space() << endl;
  for (int i= 0; i< get_anz_func() ; i++)
   {
    cout << " Funktion: " << get_func(i) << "(";
    cout << get_Time1(get_func(i)) << ",";
    cout << get_Time2(get_func(i)) << ",";
    cout << get_Time3(get_func(i)) << ")" << endl;
   };
 };
```

**Bauteil_Liste**

```
#define MAX 65

class Bauteil_Liste
 {
  protected:
   Bauteil Bau[MAX];
   int anz_bau;

  public:
   Bauteil_Liste();
   void add_Bauteil(Bauteil);
   int sub_Bauteil(int);
   Bauteil get_Bauteil(int);
   int get_Anzahl();
   void put_Anzahl(int);
   float distance(int,int);
   void sort();
   void input();
   void output();

   friend istream& operator>>(istream& is, Bauteil_Liste &l)
    {
     l.input();
     return is;
    };

   friend ostream& operator<<(ostream& os, Bauteil_Liste &l)
    {
     l.output();
     return os;
    };

   int file_input(char*);
   int file_output(char*);
 }; // end Bauteil_Liste.h

Bauteil_Liste::Bauteil_Liste()
```

```
{
 anz_bau = 0;
};

void Bauteil_Liste::add_Bauteil(Bauteil B)
{
 Bau[anz_bau] = B;
 anz_bau++;
};

int Bauteil_Liste::sub_Bauteil(int i)
{
 if (i < get_Anzahl())
  {
   for (int j=0; j< get_Anzahl(); j++)
    if (j > i)
     Bau[j-1] = Bau[j];
   anz_bau = anz_bau - 1;
   return 1;
  };
 return 0;
};

float Bauteil_Liste::distance(int b1, int b2)
   {
    float c1;
    float c2;
    c1 = 0;
    c2 = 0;
    for(int i= 0; i < get_Bauteil(b1).get_anz_func(); i++)
     for (int j= 0; j < get_Bauteil(b2).get_anz_func(); j++)
      {
       if (strcmp(get_Bauteil(b1).get_func(i), get_Bauteil(b2).get_func(j)) == 0)
        {
         c1++;
         j = get_Bauteil(b2).get_anz_func();
        };
      };

    c2 = get_Bauteil(b1).get_anz_func() + get_Bauteil(b2).get_anz_func() - c1;

    return ((c2 - c1) / c2);
   };

void Bauteil_Liste::sort()    // natürlich kein richtiger !!
   {
    Bauteil h;
    for(int i= 0; i < get_Anzahl(); i++)
     for (int j= i+2; j < get_Anzahl(); j++)
      if (distance(i,j) < distance(i,i+1))
       {
        h = Bau[j];
        Bau[j] = Bau[i+1];
        Bau[i+1] = h;
       }
   };
```

```
Bauteil Bauteil_Liste::get_Bauteil(int i)
 {
  return Bau[i];
 };

int Bauteil_Liste::get_Anzahl()
 {
  return anz_bau;
 };

void Bauteil_Liste::put_Anzahl(int a)
 {
  anz_bau = a;
 };

void Bauteil_Liste::input()
 {
  int an;
  Bauteil b;
  cout << "Anzahl der Bauteile eingeben ";
  cin >> an;
  for(int i= 0; i< an; i++)
   {
    cin >> b;
    add_Bauteil(b);
   };
 };

void Bauteil_Liste::output()
 {
  cout << "*****" << endl;
  for(int i= 0; i< anz_bau; i++)
   cout << Bau[i] << endl;
 };

int Bauteil_Liste::file_input(char* name)
 {
  anz_bau = 0;
  char buf[40];
  ifstream my_file(name);
  if(!my_file)
   return 0;

  my_file.getline(buf,40, '\n');  // Anzahl Bauteile;
  int an;
  an = strtoint(buf);
  cout << "anzahl " << an << endl;
  for(int i= 0; i< an; i++)
   {
    Bauteil b;
    float s;
    my_file.getline(buf,40, '\n');  // name
    b.put_name(buf);
    my_file.getline(buf,40, '\n');  // Space;
    s = strtofloat(buf);
    b.put_space(s);
    cout << "space " << s << endl;
    my_file.getline(buf,40, '\n');  // Anz eingaenge;
```

```
   int ae;
   ae = strtoint(buf);
   b.put_anz_eing(ae);
   cout << "anz ein " << ae << endl;
   my_file.getline(buf,40, '\n');  // Anz Funktionen;
   int af;
   af = strtoint(buf);
   cout << "anz fun " << af << endl;
   for (int j=0; j< af; j++)
    {
     my_file.getline(buf,40, '\n');  // func;
     char* func;
     func = (char*) malloc(40);
     strcpy(func,buf);
     my_file.getline(buf,40, '\n');  // t1;
     int t1;
     t1 = strtoint(buf);
     my_file.getline(buf,40, '\n');  // t2;
     int t2;
     t2 = strtoint(buf);
     my_file.getline(buf,40, '\n');  // t3;
     int t3;
     t3 = strtoint(buf);
     b.add_function(func,t1,t2,t3);
    }; // for j;
   add_Bauteil(b);
  }; // fori

 return 1;
};

int Bauteil_Liste::file_output(char* name)
 {
  ofstream my_file(name);
  if(!my_file) return 0;
  my_file << get_Anzahl() << "\n";  // Anzahl ausgeben !
  for (int a=0; a< get_Anzahl(); a++)
   {
    my_file << get_Bauteil(a).get_name() << "\n"; // Name
    my_file << get_Bauteil(a).get_space() << "\n";  //Area
    my_file << get_Bauteil(a).get_anz_eing() << "\n";  // Anzahl Eingaenge
    my_file << get_Bauteil(a).get_anz_func() << "\n";
    for(int i=0;i<get_Bauteil(a).get_anz_func(); i++)
     {
      my_file << get_Bauteil(a).get_func(i) << "\n";
      my_file << get_Bauteil(a).get_Time1(get_Bauteil(a).get_func(i)) << "\n";
      my_file << get_Bauteil(a).get_Time2(get_Bauteil(a).get_func(i)) << "\n";
      my_file << get_Bauteil(a).get_Time3(get_Bauteil(a).get_func(i)) << "\n";
     };  // for i
   };  // for a
  my_file.close();
  return 1;
 };
```

**Allocation**

```
class Allocation : public Bauteil_Liste
 {
```

```
 private:
  Befehls_Liste BEF;
  float oberG1;
  float oberG2;
  float unterG1;
  float unterG2;
  float ges1;
  float ges2;

 public:
 Allocation();
 Allocation(Befehls_Liste);
 Allocation(Befehls_Liste,Bauteil_Liste);
 int Bewertung();
 int Bewertung(Befehls_Liste);
 float get_ober_G1();
 float get_ober_G2();
 float get_unter_G1();
 float get_unter_G2();
 float get_bewertung_T1();
 float get_bewertung_T2();
 float get_space();
 mutation();
 }; // end Allocation.h

Allocation::Allocation()
 {
  anz_bau = 0;
 };

Allocation::Allocation(Befehls_Liste BE)
 {
  BEF = BE;
  anz_bau = 0;
  oberG1 = 0;
  oberG2 = 0;
  unterG1 = 0;
  unterG2 = 0;
  ges1 = 0;
  ges2 = 0;
 };

Allocation::Allocation(Befehls_Liste BE,Bauteil_Liste BA)
 {
  BEF = BE;
  anz_bau = 0;
  oberG1 = 0;
  oberG2 = 0;
  unterG1 = 0;
  unterG2 = 0;
  ges1 = 0;
  ges2 = 0;
  for (int b=0; b< BA.get_Anzahl(); b++)
   {
    add_Bauteil(BA.get_Bauteil(b));
   };
 };
```

```
int Allocation::Bewertung(Befehls_Liste BE)  // kommt erst ma weg
 {
 float bew1;   // help
 float bew2;   // help
 float nb;
 int n;
 n = 0;
 oberG1 = 0;
 unterG1 = 0;
 oberG2 = 0;
 unterG2 = 0;
 ges1 = 0;
 ges2 = 0;
 float og1;
 float ug1;
 float og2;
 float ug2;

 for (int b= 0;b < BE.get_Anzahl(); b++)   // alle Befehle
  {
   n = 0;
   og1 = 0;
   og2 = 0;
   ug1 = 99999;
   ug2 = 99999;
   bew1 = 0;
   bew2 = 0;
   for (int a = 0; a<get_Anzahl();a++)  // Bauteile
    if(get_Bauteil(a).befehl_geht(BE,b) == 1)
     {
      bew1 = bew1 + (get_Bauteil(a).get_Time1(BE.get_Befehl(b).get_function())
                * get_Bauteil(a).get_anz_func());
      bew2 = bew2 + (get_Bauteil(a).get_Time2(BE.get_Befehl(b).get_function())
                * get_Bauteil(a).get_anz_func());
      n = n+1;   // Anzahl Bauteile
      og1 = max(og1,(float)get_Bauteil(a).get_Time1(BE.get_Befehl(b).get_function()));
      og2 = max(og2,(float)get_Bauteil(a).get_Time2(BE.get_Befehl(b).get_function()));
      ug1 = min(ug1,(float)get_Bauteil(a).get_Time1(BE.get_Befehl(b).get_function()));
      ug2 = min(ug2,(float)get_Bauteil(a).get_Time2(BE.get_Befehl(b).get_function()));
     };  // if for a
   if (n>0)
    {
     ges1 = ges1 + (bew1/(n*n));  // Durchschnitt durch Anzahl Bauteile die Bef machen
     ges2 = ges2 + (bew2/(n*n));
    }
   else
    {
     ges1 = 99999;
     ges2 = 99999;
    };
   oberG1 = oberG1 + og1;
   oberG2 = oberG2 + og2;
   unterG1 = unterG1 + ug1;
   unterG2 = unterG2 + ug2;
  }; // for b
 if ((oberG1 < 99999) && (oberG2 < 99999) &&
    (unterG1 < 99999) && (unterG2 < 99999)) return 1;
 return 0;
```

```
 }; // bewerten

int Allocation::Bewertung()
 {
  return Bewertung(BEF);
 };

float Allocation::get_ober_G1()
 {
  return oberG1;
 };

float Allocation::get_ober_G2()
 {
  return oberG2;
 };

float Allocation::get_unter_G1()
 {
  return unterG1;
 };

float Allocation::get_unter_G2()
 {
  return unterG2;
 };

float Allocation::get_bewertung_T1()
 {
  return ges1;
 };

float Allocation::get_bewertung_T2()
 {
  return ges2;
 };

float Allocation::get_space()
 {
  float s;
  s = 0;
  for (int i = 0; i<get_Anzahl(); i++)
   {
    s = s + get_Bauteil(i).get_space();
   };
  return s;
 };
```

**Allocation_Liste**

```
class Allocation_Liste
 {
  private:
   Bauteil_Liste BAU;
   Allocation Allo[30];  // Die Generation
   Befehls_Liste BEF;
   int anz;
```

```
 public:
  double b_function(double,
                    double,
                    double,
                    double,double,
                    double,double); // Bewertung
  Allocation_Liste(Befehls_Liste);
  Allocation_Liste(Befehls_Liste,Bauteil_Liste);
  Allocation_Liste(Befehls_Liste,Bauteil_Liste,float);
  void init(Bauteil_Liste,float);
  void init(float);
  void put_Anzahl(int);
  int get_Anzahl();
  int geht(int);
  float distance(Bauteil,Bauteil);
  void mutation(int,int);   // Alloc a Bauteil b
  void crossover(int,int,int,int);
  void sort();
  void output();
  void output(int);
  int file_output(char*,int);
  int file_input(char*,int);
 }; // end Allocation_Liste.h

Allocation_Liste::Allocation_Liste(Befehls_Liste BE)
 {
  BEF = BE;
  for(int a= 0; a < NR_GEN; a++)  //alle Allocationen
   {
    Allo[a] = Allocation(BE);
   };
  anz = NR_GEN;
 };

Allocation_Liste::Allocation_Liste(Befehls_Liste BE,Bauteil_Liste BA)
 {
  anz = 0;
  BEF = BE;
  BAU = BA;
  BAU.sort();
 };

Allocation_Liste::Allocation_Liste(Befehls_Liste BE,Bauteil_Liste BA,float s)
 {
  anz = NR_GEN;
  BEF = BE;
  BAU = BA;
  BAU.sort();
  init(s);
 };

void Allocation_Liste::init(Bauteil_Liste BA, float space)
 {
  BAU = BA;
  BAU.sort();
  for(int a= 0; a < get_Anzahl(); a++)  //alle Allocationen
   {
    Allo[a] = Allocation(BEF);
```

```
  Allo[a].Bewertung();
  int end;
  end = 0;
  while ((Allo[a].get_space() < space) && (end == 0)
         && (Allo[a].get_Anzahl() < BEF.get_Anzahl()))
   {
    int l;
    l = rand()%BAU.get_Anzahl();
    cout << l << endl;

    if( (Allo[a].get_space() + BAU.get_Bauteil(l).get_space()) < space)
     Allo[a].add_Bauteil(BAU.get_Bauteil(l));
    else
     end = 1;
   };  // while

  // ueberpruefung obs geht
  float sphelp;
  sphelp = space;
  int geht;
  geht = 1;

  do
   {
    geht = 1;
    for (int b=0; b< BEF.get_Anzahl(); b++)  // alle Befehle
     {
      int t = 0;
      for (int c =0; c< Allo[a].get_Anzahl(); c++)  // alle Bauteile
       {
        if (Allo[a].get_Bauteil(c).befehl_geht(BEF,b) == 1)
         t = 1;
        };

      if (t == 0)   // Befehl geht nicht ...
       {
        geht = 0;
        int l;
        l = rand()%BA.get_Anzahl();
        if( (Allo[a].get_space() + BA.get_Bauteil(l).get_space()) < sphelp)
         Allo[a].add_Bauteil(BA.get_Bauteil(l));  // noch ein Bauteil
        else
         {
          l = rand()%Allo[a].get_Anzahl();
          Allo[a].sub_Bauteil(l);  // Bauteil tauschen
          l = rand()%BA.get_Anzahl();
          Allo[a].add_Bauteil(BA.get_Bauteil(l));
         };
       };
     };  // for b
    sphelp++;
   }
   while ((geht == 0) && (sphelp < (2*space)));

  }; // for a
 };  // init

int Allocation_Liste::geht(int a)
```

```
{
 return Allo[a].Bewertung();
};

void Allocation_Liste::init(float space)
{

 for(int a= 0; a < get_Anzahl(); a++)  //alle Allocationen
  {
   Allo[a] = Allocation(BEF);
   Allo[a].put_Anzahl(0);
   int end;
   end = 0;

   while ((Allo[a].get_space() < space) && (end == 0)
          && (Allo[a].get_Anzahl() < BEF.get_Anzahl()))
    {
     int l;
     l = rand()%BAU.get_Anzahl();
     cout << l << endl;
     if( (Allo[a].get_space() + BAU.get_Bauteil(l).get_space()) < space)
      Allo[a].add_Bauteil(BAU.get_Bauteil(l));
     else
      end = 1;
    };  // while

  // ueberpruefung obs geht
   float sphelp;
   sphelp = space;
   int geht;
   geht = 1;

   do
    {
     geht = 1;
     for (int b=0; b< BEF.get_Anzahl(); b++)  // alle Befehle
      {
       int t = 0;
       for (int c =0; c< Allo[a].get_Anzahl(); c++)  // alle Bauteile
        {
         if (Allo[a].get_Bauteil(c).befehl_geht(BEF,b) == 1)
          t = 1;
        };

       if (t == 0)   // Befehl geht nicht ...
        {
         geht = 0;
         int l;
         l = rand()%BAU.get_Anzahl();
         if( (Allo[a].get_space() + BAU.get_Bauteil(l).get_space()) < sphelp)
          Allo[a].add_Bauteil(BAU.get_Bauteil(l));  // noch ein Bauteil
         else
          {
           l = rand()%Allo[a].get_Anzahl();
           Allo[a].sub_Bauteil(l);  // Bauteil tauschen
           l = rand()%BAU.get_Anzahl();
           Allo[a].add_Bauteil(BAU.get_Bauteil(l));
          };
```

```
        };
       };  // for b
      sphelp++;
     }
    while ((geht == 0) && (sphelp < (2*space)));


  }; // for a
 };  // init


void Allocation_Liste::put_Anzahl(int a)
 {
  anz =a;
 };

int Allocation_Liste::get_Anzahl()
 {
   return anz;
 };

float Allocation_Liste::distance(Bauteil A, Bauteil B)
 {
    float c1;
    float c2;
    c1 = 0;
    c2 = 0;
    for(int i= 0; i < A.get_anz_func(); i++)
     for (int j= 0; j < B.get_anz_func(); j++)
      {
       if (strcmp(A.get_func(i), B.get_func(j)) == 0)
        {
         c1++;
         j = B.get_anz_func();
        };
       };

    c2 = A.get_anz_func() + B.get_anz_func() - c1;

    return ((c2 - c1) / c2);
 };

void Allocation_Liste::mutation(int a,int b)  // Allo, Bauteil
 {
  float min;
  min = 999;
  int n;
  if (b >= Allo[a].get_Anzahl())
   b = rand()%Allo[a].get_Anzahl();
  for (int ba=0 ; ba < BAU.get_Anzahl(); ba++)  // alle Bauteile
   {
    if (distance(Allo[a].get_Bauteil(b), BAU.get_Bauteil(ba)) < min)
     {
      n = ba;
      min = distance(Allo[a].get_Bauteil(b), BAU.get_Bauteil(ba));
     };
   };
```

```
 if ((n < BAU.get_Anzahl() - 2) && (n > 1))
  {
   int i;
   i = rand()%2;
   if (i == 0)
    n = n - (rand()%3 + 1);
   else
    n = n + (rand()%3 + 1);
  }

 Allo[a].sub_Bauteil(b);
 Allo[a].add_Bauteil(BAU.get_Bauteil(n));
};

void Allocation_Liste::crossover(int x, int y, int z, int p)
{
 if (p > Allo[x].get_Anzahl())
   p = Allo[x].get_Anzahl();

 Allo[z].put_Anzahl(0);
 for(int i=0; i< Allo[y].get_Anzahl(); i++)
  {
   if (i < p)
    Allo[z].add_Bauteil(Allo[x].get_Bauteil(i));
   else
    Allo[z].add_Bauteil(Allo[y].get_Bauteil(i));
  };
};

// Die Funktion setzt die Bewertungen zusammen ...

double Allocation_Liste::b_function(double og1,
                double og2,
                double ug1,
                double ug2,
                double T1,
                double T2,
                double s)
{
 double ret;
 ret = (0.9*og1)
     + (og2*0.9)
     + (ug1*0.9)
     + (ug2*0.9)
     + (T1 * 5)
     + (T2 * 5)
     + (s * 2.5);
 return ret;
};

void Allocation_Liste::sort()
{
 for (int a=0; a< get_Anzahl(); a++)
  {
   Allo[a].Bewertung();
  };

 for(int x=0; x<get_Anzahl();x++)
```

```
  for(int y= x+1; y<get_Anzahl();y++)
   {

    if (b_function((double)Allo[x].get_ober_G1(),
                   (double)Allo[x].get_ober_G2(),
                   (double)Allo[x].get_unter_G1(),
                   (double)Allo[x].get_unter_G2(),
                   (double)Allo[x].get_bewertung_T1(),
                   (double)Allo[x].get_bewertung_T2(),
                   (double)Allo[x].get_space())
            >
       b_function((double)Allo[y].get_ober_G1(),
                   (double)Allo[y].get_ober_G2(),
                   (double)Allo[y].get_unter_G1(),
                   (double)Allo[y].get_unter_G2(),
                   (double)Allo[y].get_bewertung_T1(),
                   (double)Allo[y].get_bewertung_T2(),
                   (double)Allo[y].get_space())
      )  // hier andere Bed rein ? -> wird in b_function festgelegt.
     {
      // tauschen
      Allocation L;
      L = Allo[x];
      Allo[x] = Allo[y];
      Allo[y] = L;
     };

     if (Allo[x].get_bewertung_T1()
       == Allo[y].get_bewertung_T1())  // hier andere Bed rein ?
      if (Allo[x].get_unter_G1() > Allo[y].get_unter_G1())
      {
       // tauschen
       Allocation L;
       L = Allo[x];
       Allo[x] = Allo[y];
       Allo[y] = L;
      };



    };
};


void Allocation_Liste::output()
{
 for (int a = 0; a < get_Anzahl() ; a++)
  {
   output(a);
   cout << endl;
  };
};

void Allocation_Liste::output(int i)
{
 cout << "Platz : " << Allo[i].get_space() << endl;
 cout << "Obere Grenze T1 : " << Allo[i].get_ober_G1() << endl;
 cout << "Obere Grenze T2 : " << Allo[i].get_ober_G2() << endl;
```

```
 cout << "Untere Grenze T1 : " << Allo[i].get_unter_G1() << endl;
 cout << "Untere Grenze T2 : " << Allo[i].get_unter_G2() << endl;
 cout << "Bewertung T1 : " << Allo[i].get_bewertung_T1() << endl;
 cout << "Bewertung T2 : " << Allo[i].get_bewertung_T2() << endl;
 cout << (Bauteil_Liste)Allo[i] << endl;
};

int Allocation_Liste::file_output(char* name,int i)
{
 return Allo[i].file_output(name);
};

int Allocation_Liste::file_input(char* name, int i)
{
 return Allo[i].file_input(name);
};
```

**Allocation_Generation**

```
class Allocation_Generation
{
 private:
  Bauteil_Liste BAU;
  Allocation_Code Allo[20];
  Befehls_Liste BEF;
  int anz;

 public:
  Allocation_Generation();
  Allocation_Generation(Befehls_Liste,Bauteil_Liste);
  void init();

  void mutation(int,int);      // Alloc_Nr , Bauteil_Nr
  void crossover(int,int,int,int);  // x,y,z, p
  void sort();
  void put_Anzahl(int);
  int get_Anzahl();
  Allocation_Code get_Allocation_Code(int);
  int file_get_bewertung(char*,int);
  void put_bewertung(int,float);
  void put_time(int,float);
  void put_space(int,float);
};   // Allocation_Generation.h

Allocation_Generation::Allocation_Generation()
{
 put_Anzahl(20);
};

Allocation_Generation::Allocation_Generation(Befehls_Liste BL,Bauteil_Liste BA)
{
 BAU = BA;
 BEF = BL;
 put_Anzahl(20);
 for(int i=0; i< get_Anzahl(); i++)
  {
   Allo[i].init(BEF,BAU);
  };
```

```
};


void Allocation_Generation::init()
{
 for(int i=0; i< get_Anzahl(); i++)
  {
   Allo[i].init(BEF,BAU);
  };
};


void Allocation_Generation::mutation(int m,int a)
{
 Allo[m].mutation(a);
};


void Allocation_Generation::crossover(int x,int y,int z, int p)
{
 for(int i=0; i < BAU.get_Anzahl(); i++)
  {
   if (i < p)
    Allo[z].put_anzahl_inst(i,Allo[x].get_anzahl_inst(i));
   else
    Allo[z].put_anzahl_inst(i,Allo[y].get_anzahl_inst(i));
  };
};

void Allocation_Generation::put_Anzahl(int a)
{
 anz = a;
};

int Allocation_Generation::get_Anzahl()
{
 return anz;
};

void Allocation_Generation::sort()
{
 for(int x=0; x < get_Anzahl();x++)
  for(int y= x+1; y < get_Anzahl();y++)
   {
    if (Allo[x].get_bewertung() > Allo[y].get_bewertung())
     {
      // tauschen
      Allocation_Code L;
      L = Allo[x];
      Allo[x] = Allo[y];
      Allo[y] = L;
     };
   };
};


Allocation_Code Allocation_Generation::get_Allocation_Code(int i)
{
```

```
 return Allo[i];
};


int Allocation_Generation::file_get_bewertung(char* name, int nr)
{
 return Allo[nr].file_get_bewertung(name);
};

void Allocation_Generation::put_bewertung(int n,float bew)
{
 Allo[n].put_bewertung(bew);
};

void Allocation_Generation::put_time(int n,float bew)
{
 Allo[n].put_time(bew);
};

void Allocation_Generation::put_space(int n,float bew)
{
 Allo[n].put_space(bew);
};
```

**Assignment**

```
class Assignment : public Bauteil , public Menge
{
 public:
 Assignment(Bauteil,Menge);
 Assignment();
 void sort(Befehls_Liste);
}; // Assignment.h

Assignment::Assignment(Bauteil B, Menge M)
{
 put_name(B.get_name());
 put_anz_func(0);
 put_anz_eing(B.get_anz_eing());
 put_space(B.get_space());

 for (int a=0;a<B.get_anz_func();a++)
  {

   add_function(B.get_func(a),B.get_Time1(B.get_func(a)),
              B.get_Time2(B.get_func(a)),
              B.get_Time3(B.get_func(a)));
  };
 put_anzahl(0);

 for(int i=0; i< M.get_anzahl(); i++)
  {
   add_element(M.get_element(i));  // vorher testen ob Befehl geht !!
  };
};

Assignment::Assignment()
{
```

```
 put_anz_func(0);
 put_anzahl(0);
 put_anz_eing(0);
 put_space(0);
 };


void Assignment::sort(Befehls_Liste BL)
 {
  BL.init_all();
  for(int i=0; i<get_anzahl(); i++)
   for(int j = i+1; j< get_anzahl(); j++)
    {
     if (BL.trans_all(get_element(j),get_element(i)) == 1)
      {
       int h= get_element(j);
       put_element(get_element(i),j);
       put_element(h,i);
      };
    };
 };
```

**Assignment_Liste**

```
class Assignment_Liste
 {
  private:

   int anz_assi;
   int DA_Matrix[99][99];
   int abh(int,int);
   void init_matrix();
   void update_matrix(int);

  protected:
   Assignment Assi[99];
   Befehls_Liste BEF;
   Schleifen_Liste SCH;

  public:
   Assignment_Liste();
   Assignment_Liste(Befehls_Liste);
   Assignment_Liste(Bauteil_Liste,Befehls_Liste);
   void add_Assignment(Assignment);
   Assignment get_Assignment(int);
   int get_Anzahl();

   int get_used_Anzahl();

   void put_Befehls_Liste(Befehls_Liste);
   Befehls_Liste get_Befehls_Liste();
   void put_Schleifen_Liste(Schleifen_Liste);
   void sort(); // Sortiert alle abhaengig voneinander ...
   void sort(Befehls_Liste);

   void input();
   void output();
```

```
  friend istream& operator>>(istream& is, Assignment_Liste &l)
   {
    l.input();
    return is;
   };

  friend ostream& operator<<(ostream& os, Assignment_Liste &l)
   {
    l.output();
    return os;
   };
};// end Assignment_Liste

int Assignment_Liste::abh(int x, int y)
 {
  if (x == y) return 0;

  if ((x < BEF.get_Anzahl()) && (y < BEF.get_Anzahl()))
   {
    if (DA_Matrix[x][y] == 1) return 1;
   };
  return 0;
 };

void Assignment_Liste::init_matrix()
 {
  int u;
  BEF.init_all();
  for( int x = 0; x < BEF.get_Anzahl(); x++)
   {
   for( int y = 0; y < BEF.get_Anzahl(); y++)
    {
     if ( (BEF.trans_all(x,y) == 1) || (SCH.nach(x,y) == 1) )
      DA_Matrix[x][y] = 1;
     else
      DA_Matrix[x][y] = 0;
    };
   };
 cout << " 11 14 " << DA_Matrix[11][14] <<  DA_Matrix[14][11]<< endl;

 u = 1;
 for( int t = 0; ((t < (BEF.get_Anzahl() + 3)) && (u==1)); t++)
  {
  u = 0;
  for( int x = 0; x < BEF.get_Anzahl(); x++)
   {
   for( int k=0; k<BEF.get_Anzahl(); k++)
    if (DA_Matrix[x][k] == 1)  // || (SCH.nach(x,k) == 1)

     for( int y = 0; y < BEF.get_Anzahl(); y++)
      {
       if (DA_Matrix[k][y] == 1) // || (SCH.nach(k,y) == 1))
        {
         u = 1;
         DA_Matrix[x][y] = 1;
        };
      };  // for y   for k
  }; // for x
```

```
 }; // for t;
}; // init_Matrix


void Assignment_Liste::update_matrix(int n)  // bzgl. schon sortierter Dinger
{
int u;
u = 1;
for(int b=0; b< get_Assignment(n).get_anzahl() -1; b++)
 {
  DA_Matrix[get_Assignment(n).get_element(b)]
         [get_Assignment(n).get_element(b+1)] = 1;
 };

for( int t = 0; ((t < BEF.get_Anzahl()) && (u = 1)); t++)
 {
  u = 0;
 for( int x = 0; x < BEF.get_Anzahl(); x++)
  {
   for( int k=0; k<BEF.get_Anzahl(); k++)
    if (DA_Matrix[x][k] == 1) //  || (get_Assignment(n).nach(x,k) == 1))
     for( int y = 0; y < BEF.get_Anzahl(); y++)
      {
       if (DA_Matrix[k][y] == 1)  // || (get_Assignment(n).nach(k,y) == 1))
        {
         if (DA_Matrix[x][y] == 0) u = 1;
         DA_Matrix[x][y] = 1;
        };  // if for
      };  // for y if for k
   }; // for x
 };  // for t
};

Assignment_Liste::Assignment_Liste()
{
 anz_assi = 0;
};

Assignment_Liste::Assignment_Liste(Befehls_Liste BL)
{
 anz_assi = 0;
 BEF = BL;
 // cout << BEF;
};

Assignment_Liste::Assignment_Liste(Bauteil_Liste BA, Befehls_Liste BL)
{
 for(int i=0; i< BA.get_Anzahl(); i++)
  {
    Menge M;
    Assignment A(BA.get_Bauteil(i),M);
    add_Assignment(A);
  };
 BEF = BL;
};

void Assignment_Liste::add_Assignment(Assignment A)
{
```

```
 Assi[anz_assi] = A;
 anz_assi++;
};

Assignment Assignment_Liste::get_Assignment(int i)
{
 return Assi[i];
};

int Assignment_Liste::get_Anzahl()
{
 return anz_assi;
};

int Assignment_Liste::get_used_Anzahl()
{
 int r;
 r = 0;
 for(int i= 0; i< anz_assi; i++)
  {
   if (get_Assignment(i).get_anzahl() > 0)
   r = r+1;
  };
 return r;
};

void Assignment_Liste::put_Befehls_Liste(Befehls_Liste BL)
{
 BEF = BL;
};

Befehls_Liste Assignment_Liste::get_Befehls_Liste()
{
 return BEF;
};

void Assignment_Liste::put_Schleifen_Liste(Schleifen_Liste SL)
{
 SCH = SL;
};

void Assignment_Liste::sort()
{
 BEF.init_all();
 init_matrix();
 cout << "nach init_matrix" << endl;
 cout << "a 11 14 " << abh(11,14) << abh(14,11) << endl;
 for(int a=0; a<get_Anzahl();a++)
  {
   for(int i=0; i<get_Assignment(a).get_anzahl(); i++)
    {
    for(int j = i+1; j< get_Assignment(a).get_anzahl(); j++)
     {
      if (abh(get_Assignment(a).get_element(j),
            get_Assignment(a).get_element(i)) == 1)
       {
        int h= get_Assignment(a).get_element(j);
        Assi[a].put_element(get_Assignment(a).get_element(i),j);
```

```
        Assi[a].put_element(h,i);
       }; //if
      };  // j
     };   // i
    cout << (Menge)get_Assignment(a);
    if (get_Assignment(a).get_anzahl() > 1)
     update_matrix(a);
   }; // a
  cout << "fertig " << endl;
 }; // sort

void Assignment_Liste::sort(Befehls_Liste BL)
 {
  BEF = BL;
  sort();
 };

void Assignment_Liste::input()
 {
  int an;
  Bauteil B;
  Menge M;
  cout << "Anzahl der Assignments eingeben ";
  cin >> an;
  for(int i= 0; i< an; i++)
   {
    cin >> B;
    M.put_anzahl(0);
    cin >> M;
    Assignment A = Assignment(B,M);
    add_Assignment(A);
   };
 };

void Assignment_Liste::output()
 {
  cout << "*****" << endl;
  cout << BEF << endl;
  for(int i= 0; i< anz_assi; i++)
   {
    if (get_Assignment(i).get_anzahl() > 0)
     {
      cout << (Bauteil)get_Assignment(i) << endl;
      cout << (Menge)get_Assignment(i) << endl << endl;
     };
   };
  cout << "***"  << endl;
 };
```

**Schleife**

```
class Schleife
 {

  private:
  int sa;   // Anfang und Ende der Schleife !!
  int se;
  Menge SAnf;
```

```
 Menge SEnde;
 Menge SIn;
 Menge SOut;

 public:
 Schleife();
 Schleife(int, int, Menge, Menge, Menge, Menge);
 int get_anfang();
 int get_ende();
 Menge get_SAnfang();
 Menge get_SEnde();
 Menge get_SIn();
 Menge get_SOut();
 int nach(int i, int j);
 int Move_Schleife(int i);  // Hier kommt was anderes hin !
 void output();

 friend ostream& operator<<(ostream& os, Schleife &s)
   {
    s.output();
    return os;
   };
}; // end Schleife.h

Schleife::Schleife()
 {
 };

Schleife::Schleife(int a, int e, Menge S_A, Menge S_E, Menge S_I, Menge S_O)
  {
   sa = a;
   se = e;
   SAnf.put_anzahl(0);
   SEnde.put_anzahl(0);
   SIn.put_anzahl(0);
   SOut.put_anzahl(0);
   SAnf = S_A;
   SEnde = S_E;
   SIn = S_I;
   SOut = S_O;
  };

int Schleife::get_anfang()
  {
   return sa;
  };

int Schleife::get_ende()
  {
   return se;
  };

Menge Schleife::get_SAnfang()
  {
   return SAnf;
  };

Menge Schleife::get_SEnde()
```

```
  {
   return SEnde;
  };

Menge Schleife::get_SIn()
  {
   return SIn;
  };

Menge Schleife::get_SOut()
 {
  return SOut;
 };

int Schleife::nach(int i, int j)
  {
   if ((i == sa) && (SEnde.ist_in(j) == 1) )
    return 1;
   if ((i == se) && (SEnde.ist_in(j) == 1) )
    return 1;
   if ( (SAnf.ist_in(i) == 1) && (j == sa) )
    return 1;
   if ((SAnf.ist_in(i) == 1) && (j == se) )
    return 1;
   if ((i == sa) && (SIn.ist_in(j) == 1))
    return 1;
   if ((SIn.ist_in(i) == 1) && (j == se))
    return 1;
   return 0;
  };

int Schleife::Move_Schleife(int i)
  {
    if (SAnf.sub_element(i) == 1)
     return SEnde.add_element(i);
    else
     if (SEnde.sub_element(i) == 1)
      return SAnf.add_element(i);
    return 0;
  };

void Schleife::output()
 {
    cout << get_anfang() << " : ";
    cout << SAnf << endl;
    cout << SIn << endl;
    cout << get_ende() << " : ";
    cout << SEnde << endl;
 };
```

**Schleifen_Liste**

```
class Schleifen_Liste
 {
  private:
  Schleife S[15];
  int anz_sch;
```

```
 public:
 Schleifen_Liste();
 void add_Schleife(int,int,Befehls_Liste);  // hier soll schon der Algo laufen
 int get_anzahl();
 void extract_Schleifen(Befehls_Liste);
 void optimize_Schleifen(Befehls_Liste);
 int nach(int,int); // Das nach fuer alle Schleifen ...

 void output();
 friend ostream& operator<<(ostream& os, Schleifen_Liste &s)
   {
    s.output();
    return os;
   };
}; // end Schleifen_Liste.h

Schleifen_Liste::Schleifen_Liste()
 {
 anz_sch =0;
 };


void Schleifen_Liste::add_Schleife(int sa,int se, Befehls_Liste BL)
 {
 // Aller einfachste Version .....
 Menge S_A;
 Menge S_I;
 Menge S_E;
 Menge S_O;

 for(int c =0; c< sa;c++)
  S_A.add_element(c);
 for(c=sa+1; c<se;c++)
  S_I.add_element(c);
 for(c=se+1; c<BL.get_Anzahl();c++)
  S_E.add_element(c);
 S[anz_sch] = Schleife(sa,se,S_A,S_E,S_I,S_O);
 anz_sch++;
 };

int Schleifen_Liste::get_anzahl()
 {
 return anz_sch;
 };

void Schleifen_Liste::extract_Schleifen(Befehls_Liste BL)
 {
 int sa;
 int se;
 for(int c=0;c<BL.get_Anzahl();c++)
  {
   if (BL.get_Befehl(c).is_label() == 1)
    {
     char* anf = BL.get_Befehl(c).get_label();
     cout << "la:" << anf << endl;
     if (anf[1] == 'A')
      {
       sa = c;
```

```
      for (int d= c+1;d< BL.get_Anzahl();d++)
       {
        if (BL.get_Befehl(d).is_label() == 1)
         {
          char* end = BL.get_Befehl(d).get_label();
          if (strcmp(&anf[2],&end[2]) == 0)
           {
            cout << "le " << end << endl;
            se = d;
           } // if
         }; // if
       };  // for
      add_Schleife(sa,se,BL);
     };   // if
    }; // if
   }; // for
 }; // extract

void Schleifen_Liste::optimize_Schleifen(Befehls_Liste BL)
 {
  Menge Sha;
  Menge She;
  Menge Shin;
  Menge Shelp;
  Menge Shout;

  for (int c= 0; c< get_anzahl(); c++)

  {
   Shin.put_anzahl(0);
   Sha = S[c].get_SAnfang();
   She = S[c].get_SEnde();
   Shelp = S[c].get_SIn();  // alle Befehle die dazwischen stehen
   Shelp.add_element(S[c].get_anfang());  // Anfangs und Endbefehl
   Shelp.add_element(S[c].get_ende());

   for (int b = S[c].get_anfang(); b<= S[c].get_ende(); b++)
    {
     if (BL.abh(b) == 1)   // falls selbstabhaengig
      Shin.add_element(b);
    };

   Shin.add_element(S[c].get_anfang());  // Anfangs und Endbefehl
   Shin.add_element(S[c].get_ende());

   /* ------ */

   BL.init_Tda();  // Matrix wird nur fuer Datenabhaengigkeiten init !!!!

for (int a= 0; a< Shelp.get_anzahl(); a++)
 {
    for(int x=0; x< Shelp.get_anzahl(); x++)
     for(int y = 0; y < Shelp.get_anzahl(); y++)
      {
       if (BL.ada(Shelp.get_element(x),Shelp.get_element(y)) == 1)
        {
         Shin.add_element(Shelp.get_element(x));
         Shin.add_element(Shelp.get_element(y));
```

```
      };
     };

   for (x = 0; x < Shelp.get_anzahl(); x++)
    for (int y = 0; y < Shin.get_anzahl(); y++)
     for (int z = 0; z < Shin.get_anzahl(); z++)
      if (BL.da(Shelp.get_element(x),Shin.get_element(y)) == 1)
       if (BL.aab(Shelp.get_element(x),Shin.get_element(z)) == 1)
        Shin.add_element(Shelp.get_element(x));


   for (x = 0; x < Shin.get_anzahl(); x++)
    for (int y = 0; y < Shelp.get_anzahl(); y++)
     for (int z = 0; z < Shin.get_anzahl(); z++)
      if (BL.da(Shin.get_element(x),Shelp.get_element(y)) == 1)
       if (BL.trans_all(Shelp.get_element(y),Shin.get_element(z)) == 1) // Tda
        Shin.add_element(Shelp.get_element(y));

 };   // for a

cout << Shin << endl;
Shelp.sub_element(S[c].get_anfang());
Shelp.sub_element(S[c].get_ende());

  //  BL.init_all();
   for (int x= 0;x < Shelp.get_anzahl(); x++)
    if (Shin.ist_in(Shelp.get_element(x)) == 0)  // alle Knoten die ueber sind
     {
  /*
      int t;
      t =0;
      for(int y = 0; y < Shin.get_anzahl(); y++)
       if (BL.trans_all(Shelp.get_element(x), Shin.get_element(y)) == 1)
        {
         t = 1;
         y = Shin.get_anzahl();
        };
      if (t == 1) Shout.add_element(Shelp.get_element(x));
       else
  */
      Shout.add_element(Shelp.get_element(x));
     };  // if  for

   Shin.sub_element(S[c].get_anfang());
   Shin.sub_element(S[c].get_ende());

   int sa = S[c].get_anfang();
   int se = S[c].get_ende();

   S[c] = Schleife(sa,se,Sha,She,Shin,Shout);
  }; // for c
};


int Schleifen_Liste::nach(int x, int y)
 {
  if (get_anzahl() == 0) return 0;
  for (int c=0; c< get_anzahl(); c++)
```

```
  {
   if (S[c].nach(x,y) == 1)
    return 1;
  };
 return 0;
}; // nicht rekursiv ....

void Schleifen_Liste::output()
 {
 for (int c=0; c< anz_sch;c++)
  cout << S[c];
 };
```

**Lösung**

```
class Loesung : public Assignment_Liste
 {
  int my_time;
  float my_space;
  int my_var;

  public:
  Loesung();
  Loesung(Bauteil_Liste, Befehls_Liste);
  Befehls_Liste get_Befehls_Liste();
  int init(); // Bilden irgendeiner Loesung
  int Befehl_auf_Bauteil(int);  // Auf welchem Bauteil laeuft b ?
  int move_Befehl_Bauteil(int,int);  // Befehl nach Bauteil immer ...
  int move_Befehl_Bauteil(int,int,int);  //Bewegt Befehl von Bauteil1 nach B2
  int move_Befehl_Befehl(int,int);  // Vertauscht zwei Befehle
                          //  (im gleichen Bauteil oder in verschiedenen)
  void Bewertung();       // == Scheduling
  int get_steuer(int);   // Schritt von Befehl holen
  int get_Bewertung();   // Anzahl der Steuerschritte
  void Markierung();     // den laengsten Weg markieren
  float get_Space();     // Gesamtplatzbedarf
  void output();
  friend ostream& operator<<(ostream& os, Loesung &l)
    {
     cout << (Assignment_Liste)l;
     l.output();
     return os;
    };

 int file_input(char*);
 int file_output(char*);
 int file_put_bewertung(char*);
 int file_get_bewertung(char*);
 int get_my_time();
 float get_my_space();
 void put_my_var(int);
 int get_my_var();
 }; // end Loesung.h

Loesung::Loesung()
 {
 };
```

```
Loesung::Loesung(Bauteil_Liste BA, Befehls_Liste BL)
 {
  for(int i=0; i< BA.get_Anzahl(); i++)
   {
     Menge M;
     Assignment A(BA.get_Bauteil(i),M);
     add_Assignment(A);
   };
  BEF = BL;
 };

Befehls_Liste Loesung::get_Befehls_Liste()
 {
  return BEF;
 };

int Loesung::init()
 {
  for (int b=0;b< BEF.get_Anzahl();b++)
   {
    int c =0;
    int i;
    i = 0;
    int a;
    do
     {
      a = (rand()*101)%get_Anzahl();
      if( (i = Assi[a].befehl_geht(BEF,b)) == 1)
       {
        Assi[a].add_element(b);
       };
     }
    while((c<100) && (i == 0));  // 100 zufaellige versuche
    if (i ==0)
     for( a = 0;a<get_Anzahl();a++)  // sonst den ersten der geht
      if (Assi[a].befehl_geht(BEF,b) == 1)
       {
        Assi[a].add_element(b);
        a = get_Anzahl();
        i = 1;
       }; //if for if
    if (i == 0) return 0;     // sonst 0 zurueck
   }; // for
  return 1;
 };       // init

int Loesung::Befehl_auf_Bauteil(int b)
 {
  for(int a= 0;a < get_Anzahl();a++)
   if (Assi[a].ist_in(b))
    return a;
 };



int Loesung::move_Befehl_Bauteil(int b,int ba, int bb)  // von ba nach bb
 {
  if (Assi[ba].ist_in(b))
```

```
  if (Assi[bb].befehl_geht(BEF,b))
   {
    Assi[ba].sub_element(b);
    Assi[bb].add_element(b);
    return 1;
   };
 return 0;
};

int Loesung::move_Befehl_Bauteil(int b, int ba)    // immer nach ba
{
 if (Assi[ba].befehl_geht(BEF,b))
  {
   Assi[Befehl_auf_Bauteil(b)].sub_element(b);
   Assi[ba].add_element(b);
   return 1;
  };
 return 0;
};


int Loesung::move_Befehl_Befehl(int x,int y)  // zwei Befehle (+Stellen ) vert.
{
 int ax;
 int sx;
 int ay;
 int sy;
 int z;
 z=0;
 for(int a= 0;a < get_Anzahl();a++)
  {
   if (Assi[a].ist_in(x))
    {
     ax = a;
     sx = Assi[a].get_referenz(x);
     z++;
    };
   if (Assi[a].ist_in(y))
    {
     ay = a;
     sy = Assi[a].get_referenz(y);
     z++;
    };
  }; // for

 if (z==2)
  {
   if ((Assi[ax].befehl_geht(BEF,y)) && (Assi[ay].befehl_geht(BEF,x)))
    {
     Assi[ax].put_element(y,sx);
     Assi[ay].put_element(x,sy);
     return 1;
    };
  };
 return 0;
};

void Loesung::Bewertung()   // Modifizierter ASAP
```

```
{
 int ax;
 int ay;
 int t;
 t = 1;
 sort();   // initialisieren;
 for(int b=0; b< BEF.get_Anzahl(); b++)
  {
   BEF.put_steuer(b,0);
  };

 for (int z=0; ((z < BEF.get_Anzahl()) && (t == 1)); z++)
  {   // alle Befehlspaare ...
   t = 0;
  for(int x=0; x<BEF.get_Anzahl(); x++)
   for(int y=0; y<BEF.get_Anzahl(); y++)
    {
     ax = Befehl_auf_Bauteil(x);
     ay = Befehl_auf_Bauteil(y);


     if (BEF.da(x,y) == 1)
      {
       if (BEF.get_steuer(y)
         < (BEF.get_steuer(x)
          + Assi[ax].get_Time1(BEF.get_Befehl(x).get_function())))
        {
         t = 1;
         BEF.put_steuer(y,BEF.get_steuer(x)
                   + Assi[ax].get_Time1(BEF.get_Befehl(x).get_function()));
        };
      };


     if (BEF.aab(x,y) == 1)
      {
       if (BEF.get_steuer(y)
         < (BEF.get_steuer(x)
          + Assi[ax].get_Time1(BEF.get_Befehl(x).get_function())   // T1(x)
          - Assi[ay].get_Time1(BEF.get_Befehl(y).get_function())   // T1(y)
          + 1 ))
        {
         t = 1;
         BEF.put_steuer(y,BEF.get_steuer(x)
                   + Assi[ax].get_Time1(BEF.get_Befehl(x).get_function())
                   - Assi[ay].get_Time1(BEF.get_Befehl(y).get_function())
                   + 1 );
        };
      };


     if ((ax == ay) && (Assi[ax].nach(x,y) == 1))
      {
       if (BEF.get_steuer(y)
         < (BEF.get_steuer(x)
          + Assi[ax].get_Time2(BEF.get_Befehl(x).get_function())))
        {
         t = 1;
```

```
        BEF.put_steuer(y,BEF.get_steuer(x)
                + Assi[ax].get_Time2(BEF.get_Befehl(x).get_function()));
       };
      };


     if (BEF.ada(x,y) == 1)
      {
       if (BEF.get_steuer(y)
        < (BEF.get_steuer(x)
         + Assi[ax].get_Time3(BEF.get_Befehl(x).get_function())   // T3(x)
         - Assi[ay].get_Time1(BEF.get_Befehl(y).get_function())))
        {
         t = 1;
         BEF.put_steuer(y,BEF.get_steuer(x)
                + Assi[ax].get_Time3(BEF.get_Befehl(x).get_function())
                - Assi[ay].get_Time1(BEF.get_Befehl(y).get_function()));
        }; // if
      };  // if


     if (SCH.nach(x,y) == 1)  // bei Schleife +T1
      {
       if (BEF.get_steuer(y)
        < (BEF.get_steuer(x)
         + Assi[ax].get_Time1(BEF.get_Befehl(x).get_function())))
        {
         t = 1;
         BEF.put_steuer(y,BEF.get_steuer(x)
                + Assi[ax].get_Time1(BEF.get_Befehl(x).get_function()));
        };
      };

     };  // for for

   }; // do

};  // Bewertung


int Loesung::get_steuer(int b)
{
 return BEF.get_steuer(b);
};

int Loesung::get_Bewertung()
{
 int max;
 max = 0;
 for(int b=0;b < BEF.get_Anzahl(); b++)
  {
   if (max < (BEF.get_steuer(b) +
    Assi[Befehl_auf_Bauteil(b)].get_Time1(BEF.get_Befehl(b).get_function()) ))
     max = BEF.get_steuer(b) +
    Assi[Befehl_auf_Bauteil(b)].get_Time1(BEF.get_Befehl(b).get_function());
  };
 my_time = max;
 return max;
```

```
 };

float Loesung::get_Space()
 {
  float h;
  h = 0;
  for(int i= 0; i< get_Anzahl(); i++)
   {
    if (get_Assignment(i).get_anzahl() > 0)
     {
      h = h + get_Assignment(i).get_space();
     };
   };
  my_space = h;
  return h;
 };

void Loesung::Markierung()
 {
  int t;
  t = 1;
  int last;
  last = 0;
  int bew;
  bew = get_Bewertung();

  for (int b=0;b < BEF.get_Anzahl(); b++)  // letzter Befehl
   if (bew == (BEF.get_steuer(b) +
     Assi[Befehl_auf_Bauteil(b)].get_Time1(BEF.get_Befehl(b).get_function())))
    {
     BEF.Liste[b].put_marke(2);
     cout << "erste Marke " << b << endl;
    };

  do
   {
    t = 0;
   for (int b = 0; b < BEF.get_Anzahl();b++)
    {
    if (BEF.get_Befehl(b).get_marke() == 2)
     {
      t = 1;
      last = b;
      BEF.Liste[b].put_marke(0);
      for(int c = 0; c< BEF.get_Anzahl();c++)
       if(b != c)
       {
        if ( (BEF.da(c,b) == 1)
           && (BEF.get_steuer(c) +
         Assi[Befehl_auf_Bauteil(c)].get_Time1(BEF.get_Befehl(c).get_function()))
              ==  BEF.get_steuer(b))
         {
          BEF.Liste[c].put_marke(2);   // if1
          BEF.Liste[b].put_marke(1);
         };

        if ( (BEF.ada(c,b) == 1)
           && ( ( BEF.get_steuer(c)
```

```
            +
            Assi[Befehl_auf_Bauteil(c)].get_Time3(BEF.get_Befehl(c).get_function())
            -
            Assi[Befehl_auf_Bauteil(b)].get_Time1(BEF.get_Befehl(b).get_function() ) )
            ==  BEF.get_steuer(b) ))
       {
        BEF.Liste[c].put_marke(2);   // if2
        BEF.Liste[b].put_marke(1);

       };

      if (BEF.aab(c,b) == 1)
       {
        if (BEF.get_steuer(b)
          == (BEF.get_steuer(c)
           + Assi[Befehl_auf_Bauteil(c)].get_Time1(BEF.get_Befehl(c).get_function())   // T1(c)
           - Assi[Befehl_auf_Bauteil(b)].get_Time1(BEF.get_Befehl(b).get_function())   // T1(b)
           + 1 ))
         {
          BEF.Liste[c].put_marke(2);   // if3
          BEF.Liste[b].put_marke(1);
         };
       };



      if ((Befehl_auf_Bauteil(c) == Befehl_auf_Bauteil(b)) && (Assi[Befehl_auf_Bauteil(c)].nach(c,b) == 1))
       {
        if (BEF.get_steuer(b)
          == (BEF.get_steuer(c)
            + Assi[Befehl_auf_Bauteil(c)].get_Time2(BEF.get_Befehl(c).get_function())))
         {
          BEF.Liste[c].put_marke(2);  // if4
          BEF.Liste[b].put_marke(1);
         };
       };


      if (SCH.nach(c,b) == 1)  // bei Schleife +T1
       {
        if (BEF.get_steuer(b)
          == (BEF.get_steuer(c)
           + Assi[Befehl_auf_Bauteil(c)].get_Time1(BEF.get_Befehl(c).get_function())))
         {
          BEF.Liste[c].put_marke(2);  // if5
          BEF.Liste[b].put_marke(1);
         };
       };
     }; // for c
    };  // if
   }; // for b
  }
  while(t == 1);
  BEF.Liste[last].put_marke(1);
 };  // Markierung


void Loesung::output()
```

```
{
 cout << "Platzbedarf : " << get_Space() << endl;
 cout << "Laengster : " << get_Bewertung() << endl;
 for (int s=0; s < get_Bewertung(); s++)
  {
   cout << "STEP[" << s << "] : ";
   for (int b =0; b< BEF.get_Anzahl(); b++)
    {
     if (s == BEF.get_steuer(b))
      cout << b << " .. ";
    };
   cout << endl;
  };
};

int Loesung::file_input(char* name)
{
 char buf[40];
 ifstream my_file(name);
 if(!my_file)
  return 0;
 my_file.getline(buf,40, '\n');  // Anzahl Bauteile;
 int an;
 an = strtoint(buf);
 if(an != get_Anzahl())
  return 0;
 for (int a= 0; a < get_Anzahl(); a++)
   {
    my_file.getline(buf,40, '\n');  // Anzahl Befehle;
    int abef;
    abef = strtoint(buf);
    for (int b=0; b< abef; b++)
     {
      my_file.getline(buf,40, '\n');  // Assignment == a;
      int assa;
      assa = strtoint(buf);
      my_file.getline(buf,40, '\n');  // Element (Befehl)
      int bef;
      bef = strtoint(buf);
      move_Befehl_Bauteil(bef,assa);
     };
   };
 return 1;
};


int Loesung::file_output(char* name)
{
 ofstream my_file(name);
 if(!my_file) return 0;
 my_file << get_Anzahl() << "\n";  // Anzahl Assignments
  for (int a= 0; a < get_Anzahl(); a++)
   {
    my_file << get_Assignment(a).get_anzahl() << "\n";  // Anzahl Befehle
    for (int b=0; b< get_Assignment(a).get_anzahl(); b++)
     {
      my_file << a << "\n";
      my_file << get_Assignment(a).get_element(b) << "\n";
```

```
    };
   };
 my_file.close();
 return 1;
};

int Loesung::file_put_bewertung(char* name)  // time und space setzen
 {
  ofstream my_file(name);
  if(!my_file) return 0;
  my_file << get_Bewertung() << "\n";
  my_file << get_Space() << "\n";
  my_file << get_my_var() << "\n";
  my_file.close();
  return 1;
 };

int Loesung::file_get_bewertung(char* name)  // my_time  und my_space setzen
 {
  char buf[20];
  ifstream my_file(name);
  if(!my_file)
   return 0;
  my_file.getline(buf,20, '\n');  // Anzahl bewertung;
  my_time = strtoint(buf);
  my_file.getline(buf,20, '\n');  // Anzahl space;
  my_space = strtofloat(buf);
  my_file.getline(buf,20, '\n');  // Anzahl var;
  my_var = strtoint(buf);
  return 1;
 };

int Loesung::get_my_time()
 {
  return my_time;
 };

float Loesung::get_my_space()
 {
  return my_space;
 };

void Loesung::put_my_var(int v)
 {
  my_var = v;
 };

int Loesung::get_my_var()
 {
  return my_var;
 };
```

**Variable**

```
class Variable
{
 private:
  int von;
```

```
  int bis;
  char* name;
  int ist_ein_aus;  // Eingang == 1 Ausgang == 2 inout = 3 sonst 0;
  int use;          // wie oft wird v beschrieben ?

 public:
  Variable();
  Variable(char*);
  void set_eingang();
  void set_ausgang();
  void set_ein_aus();
  void set_normal();
  int ist_eingang();
  int ist_ausgang();
  int ist_normal();
  void set_von(int);
  void set_bis(int);
  void set_name(char*);  // Zu
  void set_use(int);

  int get_von();
  int get_bis();
  char* get_name();
  int get_use();  // Anzahl M-Eingaenge
};  // end Variable.h

Variable::Variable()
 {
  von = 999;
  bis = 1;
  ist_ein_aus = 0;
  name = (char*) malloc(13);
  use = 0;
 };

Variable::Variable(char* c)
 {
  von = 999;
  bis = 1;
  ist_ein_aus = 0;
  name = (char*) malloc(13);
  sprintf(name,"%s",c);
 };

void Variable::set_eingang()
 {
  ist_ein_aus = 1;
 };

void Variable::set_ausgang()
 {
  ist_ein_aus = 2;
 };

void Variable::set_ein_aus()
 {
  ist_ein_aus = 3;
 };
```

```
void Variable::set_normal()
 {
  ist_ein_aus = 0;
 };

int Variable::ist_eingang()
 {
  if ((ist_ein_aus == 1) || (ist_ein_aus == 3))
    return 1;
  return 0;
 };

int Variable::ist_ausgang()
 {
  if ((ist_ein_aus == 2) || (ist_ein_aus == 3))
    return 1;
  return 0;
 };

int Variable::ist_normal()
 {
  if (ist_ein_aus == 0)
    return 1;
  return 0;
 };

void Variable::set_von(int v)
 {
  von = v;
 };

void Variable::set_bis(int b)
 {
  bis = b;
 };

void Variable::set_name(char* c)
 {
  sprintf(name,"%s",c);
 };

void Variable::set_use(int u)
 {
  use = u;
 };

int Variable::get_von()
 {
  return von;
 };

int Variable::get_bis()
 {
  return bis;
 };

char* Variable::get_name()
```

```
{
 return name;
};

int Variable::get_use()
{
 return use;
};
```

**Variablen_Liste**

```
class Variablen_Liste
{
 private:
  int anz;
  Variable Liste[100];
  int Name[100];  // der zugewiesene Name (v1 .. v199)
  Loesung L;
  Befehls_Liste BEF;
  Bauteil_Liste BAU;
  Schleifen_Liste SCH;

 public:
  Variablen_Liste(Befehls_Liste);
  Variablen_Liste(Befehls_Liste,Bauteil_Liste);
  Variablen_Liste(Befehls_Liste,Bauteil_Liste,Schleifen_Liste);

  Variablen_Liste();
  Variablen_Liste(Loesung);
  void init(Loesung);
  void extract();   // Variablen extrahieren ...
  void put_anzahl(int);
  int get_anzahl();
  void put_Variable(Variable);
  Variable get_Variable(int);
  int add_Variable(Variable);

  int get_Name(int);
  int ist_eingang_name(int);  // bzgl. Name
  int ist_ausgang_name(int);  // bzgl. Name
  int get_use_name(int);     // bzgl. Name
  int get_bau_use_name(int, Loesung); // Von wievielen Bauteilen wird v besch.
  int ist_in(char*);
  int ist_in(Variable);
  int var_nr(char*);   // Var Nr nicht bzgl. Name
  int sub_Variable(Variable);
  int sub_Variable(char *);
  void sort_unten();
  void sort_oben();
  void zusammenfassen();
  int join(int,int);
  int rename();  // gibt die Anzahl der ben. Var zurueck
  void output();
  friend ostream& operator<<(ostream& os, Variablen_Liste &v)
   {
    v.output();
    return os;
   };
```

```
}; // end Variablen_Liste.h

Variablen_Liste::Variablen_Liste()
{
 anz = 0;
};

Variablen_Liste::Variablen_Liste(Loesung LO)
{
// extrahieren aller Variablen mit anfang und Ende aus der Loesung
 anz = 0;
 L = Loesung(BAU,BEF);
 L = LO;
 extract();
};



Variablen_Liste::Variablen_Liste(Befehls_Liste B)
{
 anz = 0;
 BEF = B;
} ;

Variablen_Liste::Variablen_Liste(Befehls_Liste B,Bauteil_Liste BA)
{
 anz =0;
 BEF = B;
 BAU = BA;
};

Variablen_Liste::Variablen_Liste(Befehls_Liste B,Bauteil_Liste BA, Schleifen_Liste SC)
{
 anz =0;
 BEF = B;
 BAU = BA;
 SCH = SC;
 L = Loesung(BAU,BEF);
 L.put_Schleifen_Liste(SCH);

};

void Variablen_Liste::init(Loesung L1)
{
 L = L1;
 L.Bewertung();
 L.get_Bewertung();
 L.Markierung();
};


Variablen_Liste::var_nr(char* vn)
{
 for (int n=0; n< get_anzahl(); n++)
  if (strcmp(Liste[n].get_name(),vn) == 0)
   return n;
};
```

```
void Variablen_Liste::extract()
 {
  anz = 0;
  cout << BEF.get_Anzahl() << endl;

  for(int b = 0; b< L.get_Befehls_Liste().get_Anzahl(); b++)
   {

    for(int i = 0; i< L.get_Befehls_Liste().get_Befehl(b).get_in_anzahl(); i++)
     {
      Variable v;
      v.set_name(L.get_Befehls_Liste().get_Befehl(b).get_input(i));
      add_Variable(v);
     };

    for(i = 0; i< L.get_Befehls_Liste().get_Befehl(b).get_out_anzahl(); i++)
     {
      Variable v;
      v.set_name(L.get_Befehls_Liste().get_Befehl(b).get_output(i));
      add_Variable(v);                  // set_use = 0 falls gibts noch nicht
      Liste[var_nr(v.get_name())].set_use(Liste[var_nr(v.get_name())].get_use() +1);
     };
   };



  for(b = 0; b< L.get_Befehls_Liste().get_Anzahl(); b++)
   {
    if (strcmp(L.get_Befehls_Liste().get_Befehl(b).get_function(),"out") == 0)
     for(int i = 0; i< L.get_Befehls_Liste().get_Befehl(b).get_in_anzahl(); i++)
      {
       if(Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_input(i))].ist_eingang() == 1)
        Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_input(i))].set_ein_aus();
       else
        Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_input(i))].set_ausgang();
      };

    if (strcmp(L.get_Befehls_Liste().get_Befehl(b).get_function(),"in") == 0)
     for(int i = 0; i< L.get_Befehls_Liste().get_Befehl(b).get_out_anzahl(); i++)
      {
       if(Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_output(i))].ist_ausgang() == 1)
        Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_output(i))].set_ein_aus();
       else
        Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_output(i))].set_eingang();
      };
   };


/*  Hier werden die Intervalle gebildet */

  for(b = 0; b< L.get_Befehls_Liste().get_Anzahl(); b++)
   {
```

```
   for(int i = 0; i< L.get_Befehls_Liste().get_Befehl(b).get_in_anzahl(); i++) // Eingangsv
    {
     if (Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_input(i))].get_bis() <
        (L.get_steuer(b)
       + L.get_Assignment(L.Befehl_auf_Bauteil(b)).get_Time3(L.get_Befehls_Liste().
              get_Befehl(b).get_function()) - 1))
      Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_input(i))].set_bis(L.get_steuer(b)
       + L.get_Assignment(L.Befehl_auf_Bauteil(b)).get_Time3(L.get_Befehls_Liste().
              get_Befehl(b).get_function()) -1);
    };  // for i

   for(int o = 0; o< L.get_Befehls_Liste().get_Befehl(b).get_out_anzahl(); o++) // Ausgangsv
    {
     if (Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_output(o))].get_von() >
        (L.get_steuer(b)
       + L.get_Assignment(L.Befehl_auf_Bauteil(b)).get_Time1(L.get_Befehls_Liste().
              get_Befehl(b).get_function())  ))
      Liste[var_nr(L.get_Befehls_Liste().
              get_Befehl(b).get_output(o))].set_von(L.get_steuer(b)
       + L.get_Assignment(L.Befehl_auf_Bauteil(b)).get_Time1(L.get_Befehls_Liste().
              get_Befehl(b).get_function())  );
    };  // for o
  };  // for b

 };  // extract

void Variablen_Liste::put_anzahl(int a)
 {
  anz = a;
 };

int Variablen_Liste::get_anzahl()
 {
  return anz;
 };

void Variablen_Liste::put_Variable(Variable v)
 {
    Liste[anz] = v;
    Name[anz] = anz;
    anz++;
 };

Variable Variablen_Liste::get_Variable(int i)
 {
  return Liste[i];
 };

int Variablen_Liste::add_Variable(Variable v)
   {
    if (anz == 0)
     {
      Name[anz] = 0;
      Liste[anz] = v;
      Liste[anz].set_use(0);  // erste benutzung
```

```
      if(v.ist_eingang() == 1)
       Liste[anz].set_eingang();
      if (v.ist_ausgang() == 1)
       Liste[anz].set_ausgang();
      if ((v.ist_eingang() == 1) && (v.ist_ausgang() == 1))
       Liste[anz].set_ein_aus();
      anz = 1;
      return 1;
     }
    else
     {
     for (int j=0;j < anz; j++)
      if (strcmp(Liste[j].get_name(),v.get_name()) == 0)
       return 0;  // gibts schon
     Name[anz] = anz;
     Liste[anz] = v;
     Liste[anz].set_use(0);
     if(v.ist_eingang() == 1)
      Liste[anz].set_eingang();
     if (v.ist_ausgang() == 1)
      Liste[anz].set_ausgang();
     if ((v.ist_eingang() == 1) && (v.ist_ausgang() == 1))
      Liste[anz].set_ein_aus();
     anz++;
     return 1;
     };
   };

int Variablen_Liste::get_Name(int i)
 {
  return Name[i];
 };

int Variablen_Liste::ist_eingang_name(int v)
 {
  for(int i=0; i < get_anzahl(); i++)
   {
    if (v == get_Name(i))
     {
      if (get_Variable(i).ist_eingang() == 1)
       return 1;
     };
   };
  return 0;
 }; // ist_ausgang

int Variablen_Liste::ist_ausgang_name(int v)
 {
  for(int i=0; i < get_anzahl(); i++)
   {
    if (v == get_Name(i))
     {
      if (get_Variable(i).ist_ausgang() == 1)
       return 1;
     };
   };
  return 0;
 };  // ist_ausgang
```

```
int Variablen_Liste::get_use_name(int v)
 {
  int r;
  r = 0;
  for (int i = 0; i < get_anzahl(); i++)
   if (v == get_Name(i))
    {
     r = r + get_Variable(i).get_use();
    };
  return r;
 };  // get_use_name

int Variablen_Liste::get_bau_use_name(int v, Loesung L1)
 {
  int ret;
  ret = 0;
   for (int a = 0; a < L.get_Anzahl(); a++)  // alle assignments
    {
     for (int b =0; b < L.get_Assignment(a).get_anzahl(); b++)  // alle bef
      for (int o =0; o < L.get_Befehls_Liste().
                    get_Befehl(
       L.get_Assignment(a).get_element(b)).get_out_anzahl(); o++)  // alle ausg
      for(int j=0; j<get_anzahl(); j++)
       if (get_Name(j) == v)    // Variablen raussuchen
        if(strcmp(get_Variable(j).get_name()   // Falls AUsgang == VAR
              ,L.get_Befehls_Liste().get_Befehl(
              L.get_Assignment(a).get_element(b)).get_output(o)) == 0)
         {
          ret++;
          b = L.get_Assignment(a).get_anzahl();
          j = get_anzahl();
          o = L.get_Befehls_Liste().
                    get_Befehl(L.get_Assignment(a).get_element(b))
                     .get_out_anzahl();
         }; // if if for for
    };  // for;
  return ret;
 };

int Variablen_Liste::ist_in(char* c)   // Variable gibts schon
 {
  for (int j=0;j < anz; j++)
   if (strcmp(Liste[j].get_name(),c) == 0)
    return 1;
  return 0;
 };

int Variablen_Liste::ist_in(Variable v)
 {
  for (int j=0;j < anz; j++)
   if (strcmp(Liste[j].get_name(),v.get_name()) == 0)
    return 1;
  return 0;
 };

int Variablen_Liste::sub_Variable(Variable v)
   {
```

```
   for (int j=0; j<anz; j++)
     if (strcmp(Liste[j].get_name(),v.get_name()) == 0)
      {
       for(int k=j; k<anz; k++)
         Liste[k] = Liste[k+1];
       anz = anz-1;
       return 1;
      };
   return 0;
  };

int Variablen_Liste::sub_Variable(char* c)
  {
   for (int j=0; j<anz; j++)
     if (strcmp(Liste[j].get_name(),c) == 0)
      {
       for(int k=j; k<anz; k++)
         Liste[k] = Liste[k+1];
       anz = anz-1;
       return 1;
      };
   return 0;
  };

void Variablen_Liste::sort_unten()
 {
  // Liste sortieren nach unterer Grenze
  for (int i=0; i< get_anzahl(); i++)
   for (int j= i+1; j < get_anzahl(); j++)
    {
     if (Liste[i].get_von() > Liste[j].get_von())
      {
       Variable V;
       V = Liste[i];
       Liste[i] = Liste[j];
       Liste[j] = V;
       int w;
       w = Name[i];
       Name[i] = Name[j];
       Name[j] = w;
      };
    };
 };

void Variablen_Liste::sort_oben()
 {
  // Liste nach oberer Grenze sortieren
  for (int i=0; i< get_anzahl(); i++)
   for (int j= i+1; j < get_anzahl(); j++)
    {
     if (Liste[i].get_bis() > Liste[j].get_bis())
      {
       Variable V;
       V = Liste[i];
       Liste[i] = Liste[j];
       Liste[j] = V;
       int w;
       w = Name[i];
```

```
      Name[i] = Name[j];
      Name[j] = w;
     };
   };
};

void Variablen_Liste::zusammenfassen()
{
 sort_oben();
 for (int i=0; i< get_anzahl(); i++)  // init
  Name[i] = i;
 for (int x=0; x<get_anzahl(); x++)
  for (int y=0; y < get_anzahl(); y++)
   {
    if (Liste[x].get_bis() < Liste[y].get_von())
     join(x,y);
   };
};

int Variablen_Liste::join(int x,int y)
{
 if (Liste[x].get_von() > Liste[x].get_bis()) return 0;
 if (Liste[y].get_von() > Liste[y].get_bis()) return 0;
 if (Liste[x].get_bis() < Liste[y].get_von())
  {
   int n;
   n = Name[y];
   for(int i= 0; i< get_anzahl() ; i++)
    {
     if (Name[i] == Name[x])
       Liste[i].set_bis(Liste[y].get_bis());
     if (Name[i] == Name[y])
       Liste[i].set_von(Liste[x].get_von());
    }; // for i
   for(i= 0; i< get_anzahl() ; i++)
     if (Name[i] == n)
       Name[i] = Name[x];
   return 1;
  };  // if
 return 0;
};

int Variablen_Liste::rename()
{
 int max;
 max = 0;
 int t;
 do {
 max = 0;
 t = 0;
 for( int count =0; count < get_anzahl(); count++)
  {
   int c;
   c = 0;
    for (int i = 0; i< get_anzahl(); i++)
     {
      if (Name[i] == count)
       {
```

```
        c = 1;
        if (max < count)
         max = count;
       };
      };
     if (c == 0)
     {
      for(int i = 0; i< get_anzahl(); i++)
       {
        if (Name[i] > count)
         {
          t = 1;
          Name[i] = Name[i] - 1;
         };
       }; // for
     };  // if
  };   // for
 }
 while(t == 1);
 return (max + 1);
};   // rename

void Variablen_Liste::output()
{
 for (int i =0; i< get_anzahl(); i++)
  {
   cout << "V_" << Name[i] << " : ";
   cout << Liste[i].get_name();
   cout << "[" << Liste[i].get_von() << "," << Liste[i].get_bis() << "] :";
   if (Liste[i].ist_eingang() == 1)
    cout << "in";
   if (get_Variable(i).ist_ausgang() == 1)
    cout << "out";
   cout << " -> " << get_Variable(i).get_use();
   cout << endl;
  };
};
```

**Generation**

```
class Generation
 {
  private:
   int Time[20];
   float Space[20];
   int Var[20];
   int anz;
   Bauteil_Liste BAU;
   Befehls_Liste BEF;
   Schleifen_Liste SCH;
  public:
   Loesung *GEN[20];
   float w1;  // Time
   float w2;  // Space
   float w3;  // Variablen
   Generation();
   Generation(int,Befehls_Liste,Bauteil_Liste,Schleifen_Liste);
   Generation(int,Befehls_Liste,Bauteil_Liste);
```

```
  void put_anzahl(int);
  int get_anzahl();
  void init();   // Zufaellige Generation  generieren
  void sort();   // Sortieren nach bestem  (init)
  void sort_verteilt();   // Sortiert spaeter
  int vergleich(int,int);  // x < y ?
  void crossover(int,int,int,int); // (x,y) -> z und Punkt p bzgl. BEF
  void output();
  void output(int);
  Loesung get_Loesung(int);  // gibt Loesung mit Nr int aus
  friend ostream& operator<<(ostream& os, Generation &l)
   {
    l.output();
    return os;
   };
  int file_output(char*,int);
  int file_input(char*,int);
  int file_put_bewertung(char*,int);
  int file_get_bewertung(char*,int);
}; // Generation.h

Generation::Generation()
 {
 BEF = Befehls_Liste();
 BAU = Bauteil_Liste();
 SCH = Schleifen_Liste();
  anz = 0;
 w1 = 1;
 w2 = 1;
 w3 = 1;
 };

Generation::Generation(int a,Befehls_Liste BL,Bauteil_Liste BA, Schleifen_Liste SC)
 {
  w1 = 1;
  w2 = 1;
  w3 = 1;
  BEF = BL;
  BAU = BA;
  SCH = SC;
  put_anzahl(a);
  init();
 };

Generation::Generation(int a,Befehls_Liste BL,Bauteil_Liste BA)
 {
  w1 = 1;
  w2 = 1;
  w3 = 1;
  BEF = BL;
  BAU = BA;
  SCH.extract_Schleifen(BL);
  SCH.optimize_Schleifen(BL);
  put_anzahl(a);
  init();
 };

void Generation::put_anzahl(int a)
```

```
{
 anz = a;
};

int Generation::get_anzahl()
{
 return anz;
};

Loesung Generation::get_Loesung(int i)
{
 return *GEN[i];
};

void Generation::init()
{
 for (int a=0; a< anz; a++)
  {
   GEN[a] = new Loesung;
   *GEN[a] = Loesung(BAU,BEF);
   GEN[a]->put_Schleifen_Liste(SCH);
   GEN[a]->init();
  };
};

void Generation::sort()
{
 for (int a=0; a< anz; a++)
  {
   GEN[a]->Bewertung();
   Time[a] = GEN[a]->get_Bewertung();
   Space[a] = GEN[a]->get_Space();
   GEN[a]->Markierung();
  };

 for(int x=0; x<anz;x++)
  for(int y= x+1; y<anz;y++)
   {
    if (vergleich(y,x) == 1)
     {
      // tauschen

      Loesung *L;
      L = GEN[x];
      GEN[x] = GEN[y];
      GEN[y] = L;
      int t;
      t = Time[x];
      Time[x] = Time[y];
      Time[y] = t;
      float s;
      s = Space[x];
      Space[x] = Space[y];
      Space[y] = s;
     };
   };
};
```

```
void Generation::sort_verteilt()
 {
  for (int a=0; a< anz; a++)
   {
    Time[a] = GEN[a]->get_my_time();
    Space[a] = GEN[a]->get_my_space();
    Var[a] = GEN[a]->get_my_var();
   };

  for(int x=0; x<anz;x++)
   for(int y= x+1; y<anz;y++)
    {
     if (vergleich(y,x) == 1)
      {
       // tauschen
       Loesung *L;
       L = GEN[x];
       GEN[x] = GEN[y];
       GEN[y] = L;
       int t;
       t = Time[x];
       Time[x] = Time[y];
       Time[y] = t;
       float s;
       s = Space[x];
       Space[x] = Space[y];
       Space[y] = s;
       int v;
       v = Var[x];
       Var[x] = Var[y];
       Var[y] = v;
      };
    };
 };


int Generation::vergleich(int x, int y)  // Bewertungsfunktion nur Time !!
 {
  if ((Time[x]*w1 + Space[x]*w2 + Var[x]*w3) < (Time[y]*w1 + Space[y]*w2 + Var[y]*w3))
   return 1;
  return 0;
 };

void Generation::crossover(int x,int y, int z, int p)
 {
  if (BEF.get_Anzahl() < p)
   p = BEF.get_Anzahl();
  for (int b=0; b< p; b++)
   {
    int a;
    a = GEN[x]->Befehl_auf_Bauteil(b);
    GEN[z]->move_Befehl_Bauteil(b,a);
   };
  for(b=p; b < BEF.get_Anzahl(); b++)
   {
    int a;
    a = GEN[y]->Befehl_auf_Bauteil(b);
    GEN[z]->move_Befehl_Bauteil(b,a);
```

```
  }
};

void Generation::output()
{
 for (int i=0; i<anz;i++)
  {
   cout << "Echte Bewertung: " << endl;
   cout << " TIME: " << GEN[i]->get_my_time() << endl;
   cout << " SPACE: " << GEN[i]->get_my_space() << endl;
   cout << " VAR:   " << GEN[i]->get_my_var() << endl;
   cout << "Loesung " << i << "::" << *GEN[i] << endl;
  };
};

void Generation::output(int i)
{
   cout << "Echte Bewertung: " << endl;
   cout << " TIME: " << GEN[i]->get_my_time() << endl;
   cout << " SPACE: " << GEN[i]->get_my_space() << endl;
   cout << " VAR:   " << GEN[i]->get_my_var() << endl;
   GEN[i]->Bewertung();
   GEN[i]->get_Bewertung();
   GEN[i]->Markierung();
   cout << "Loesung " << i << "::" << *GEN[i] << endl;
};

int Generation::file_output(char* name, int i)
{
 return GEN[i]->file_output(name);
};

int Generation::file_input(char* name, int i)
{
 return GEN[i]->file_input(name);
};

int Generation::file_put_bewertung(char* name, int i)
{
 return GEN[i]->file_put_bewertung(name);
};

int Generation::file_get_bewertung(char* name, int i)
{
 return GEN[i]->file_get_bewertung(name);
};
```

**Netzliste**

```
class Netzliste
{
 private:
  Loesung LSG;
  Variablen_Liste VAR;

 public:
 Netzliste(Loesung,Variablen_Liste);
 Netzliste();
```

```
  int entity(char*);  // Anzahl Steuer zurueckgeben
  void mult_component(int);
  void alu_component(int);
  void register_component();
  void var_multiplexer(int);
  void var2_multiplexer(int);
  void alu_multiplexer(int);
  void variable(int);
  void alu(int);
  void out_slow(int);
  void out_fast(int);
  void signale_var();
  void signale_alu();
  void alle_variablen();
  void alle_alus();
  void alle_componenten();
  void architecture(char*, char*);
 }; // Netzliste.h

Netzliste::Netzliste(Loesung L, Variablen_Liste VL)
 {
  LSG = L;
  VAR = VL;
 };

Netzliste::Netzliste()
 {};

int Netzliste::entity(char* name)  // gibt anzahl Steuerleitungen zurueck
 {
      // entity erzeugen
  int ret;
  ret = 0;

     cout << "ENTITY " << name << " IS" << endl;

     cout << "PORT (";
     int k;

// Eingaenge

     k = 0;
     for (int i= 0; i<VAR.rename(); i++)
      {
       if (VAR.ist_eingang_name(i) == 1)
        {
         if (k == 1)
          {
           cout << ",";
          };
         k = 1;
         cout << "IN_V" << i;
        }; // if
      };  // for
     cout << " : in INTEGER; " << endl;

// Ausgaenge
```

```
    k = 0;
    for (i= 0; i<VAR.rename(); i++)
     {
      if (VAR.ist_ausgang_name(i) == 1)
       {
        if (k == 1)
         {
          cout << ",";
         };
        k = 1;
        cout << "OUT_V" << i;
       }; // if
     };  // for
    cout << " : out INTEGER; " << endl;

/* Inouts

    k = 0;
    for (i= 0; i<VAR.rename(); i++)
     {
      if ((VAR.ist_eingang_name(i) == 1)
       && (VAR.ist_ausgang_name(i) == 1))
       {
        if (k == 1)
         {
          cout << ",";
         };
        k = 1;
        cout << "INOUT_V" << i;
       }; // if
     };  // for
    cout << " : inout INTEGER; " << endl;
*/

// Clock
    cout << " C : in Bit; " << endl;

// D ..

    k = 0;
    for (i= 0; i<VAR.rename(); i++)
     {
      if (k == 1)
       {
        cout << ",";
       };
      k = 1;
      cout << "D_" << i;
     };
    cout << " : in Bit; " << endl;

ret = VAR.rename();

// Variablen Multiplexer
    int semi;

    semi = 0;
    for (i= 0; i<VAR.rename(); i++)
```

```
      {
       if (VAR.get_bau_use_name(i,LSG) > 1)
        {
         if (semi == 1) cout << ";" << endl;
         semi = 1;
         cout << "SV_" << i;
         cout << " : in Bit_Vector(0 to ";
         cout << (ceil(log10(VAR.get_bau_use_name(i,LSG))/log10(2)) - 1);
         cout << ")";
         ret = ret + (ceil(log10(VAR.get_bau_use_name(i,LSG))/log10(2)));
        };
      };

// ALU Multiplexer

     for (i= 0; i<LSG.get_Anzahl(); i++)
      {
       if (LSG.get_Assignment(i).get_anzahl() > 1)
        {
         if (semi == 1) cout << ";" << endl;
         semi = 1;
         cout << "SA_" << i;
         cout << " : in Bit_Vector(0 to ";
         cout << (ceil(log10(LSG.get_Assignment(i).get_anzahl())/log10(2)) - 1);
         cout << ")";
         ret = ret + (ceil(log10(LSG.get_Assignment(i).get_anzahl())/log10(2)));
        };
      };

// ALU Funktionen

     for (i= 0; i<LSG.get_Anzahl(); i++)
      {
       if (LSG.get_Assignment(i).get_anzahl() > 0)   // Falls benutzt
        if (LSG.get_Assignment(i).get_anz_func() > 1)  // Falls mehr als eine F
         {
          if (semi == 1) cout << ";" << endl;
          semi = 1;
          cout << "F_" << i;
          cout << " : in Bit_Vector(0 to ";
          cout << (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2)) - 1);
          cout << ")";
          ret = ret + (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2)));
         };  // if if
      };  // for

     cout << ");" << endl;
     cout << "END " << name << ";" << endl;
 return ret;  // Anzahl der Steuerleitungen
};  // end entity

void Netzliste::mult_component(int i)
 {
  cout << "COMPONENT MULTI_" << i << endl;
  cout << "PORT(";
  int k;
  k = 0;
  for (int j=0; j< i; j++)
```

```
  {
   if (k == 1)
    {
     cout << ",";
    };
   k = 1;
   cout << "I" << j;
  };  // for
  cout << " : in INTEGER; " << endl;
  cout << " S : in Bit_Vector(0 to ";
  cout << (ceil(log10(i)/log10(2)) - 1);
  cout << "); ";
  cout << " O: out INTEGER); " << endl;
  cout << "END COMPONENT; " << endl;
  cout << "for all: MULTI_" << i << endl;
  cout << "         use entity work.MULTI_" << i << "(M" << i << ");" << endl;
 };

 void Netzliste::alu_component(int i)
 {
  if ((strcmp(LSG.get_Assignment(i).get_func(0),
                    "out") != 0) && (strcmp(LSG.get_Assignment(i).get_func(0),
                    "in") != 0))
 {

 //  cout << "COMPONENT ALU_" << i << endl;
 // neu :
  cout << "COMPONENT " << LSG.get_Assignment(i).get_name() << endl;

  cout << "PORT(";
  int k;
  k = 0;
  if (LSG.get_Assignment(i).get_anz_eing() > 0)
   {
    for (int j=0; j< LSG.get_Assignment(i).get_anz_eing(); j++)
     {
      if (k == 1)
       {
        cout << ",";
       };
      k = 1;
      cout << "I" << j;
     };  // for
    cout << " : in INTEGER; " << endl;
   }; // if

  if (LSG.get_Assignment(i).get_anz_func() > 1)
   {
    cout << " F : in Bit_Vector(0 to ";
    cout << (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2)) - 1);
    cout << "); ";
   };

  k = 0;
  if (LSG.get_Assignment(i).get_anz_ausg() > 0)
   {
    for (int j=0; j< LSG.get_Assignment(i).get_anz_ausg(); j++)
     {
```

```
      if (k == 1)
       {
        cout << ",";
       };
      k = 1;
      cout << "O" << j;
     };  // for
    cout << " : out INTEGER ";
   };  // if

  cout << ");" << endl;
  cout << "END COMPONENT;  " << endl;

//  cout << "for all: ALU_" << i << endl;
//  cout << "         use entity work.ALU_" << i << "(A" << i << ");" << endl;
// neu:
  cout << "for all: " << LSG.get_Assignment(i).get_name() << endl;
  cout << "         use entity work." << LSG.get_Assignment(i).get_name();
  cout << "(ARCH_" << LSG.get_Assignment(i).get_name() << ");" << endl;


 }; // if
 };

void Netzliste::register_component()
 {
  cout << "COMPONENT REG" << endl;
  cout << "PORT (I: INTEGER; D,C : in Bit; O: out INTEGER);" << endl;
  cout << "END COMPONENT;  " << endl;
  cout << "for all: REG" << endl;
  cout << "         use entity work.REG(R);" << endl;
 };  // reg_compo

void Netzliste::variable(int i)
 {
  cout << "REG_" << i << ": REG" << endl;
  cout << "PORT MAP(";
  if (VAR.get_bau_use_name(i,LSG) > 1) // Mplex am Eingang
   cout << "M_REG_O_" << i;
  else
   if (VAR.ist_eingang_name(i) == 1)
    cout << "IN_V" << i;
   else
    {
     for(int j=0; j<VAR.get_anzahl(); j++)
      if (VAR.get_Name(j) == i)  // Register Allocation
       for(int l=0; l< LSG.get_Befehls_Liste().get_Anzahl(); l++)  // alle BEf
        for(int m=0; m< LSG.get_Befehls_Liste().get_Befehl(l).get_out_anzahl(); m++)
         if(strcmp(VAR.get_Variable(j).get_name()   // Falls Ausgang == VAR
                ,LSG.get_Befehls_Liste().get_Befehl(l).get_output(m)) == 0)
          {
           cout << "OALU_" << LSG.Befehl_auf_Bauteil(l) << "_O" << m;
           j = VAR.get_anzahl();
           l = LSG.get_Befehls_Liste().get_Anzahl();
          };
    };

  cout << ", " << "D_" << i << ", C, OVAR_" << i << "_O" << ");" << endl << endl;
```

```
};

void Netzliste::out_slow(int i)
{
   int b;
   b = LSG.get_Assignment(i).get_element(0);
   for(int l=0; l< LSG.get_Befehls_Liste().
                    get_Befehl(b).get_in_anzahl(); l++)
    for(int j=0; j<VAR.get_anzahl(); j++)
     if(strcmp(VAR.get_Variable(j).get_name()   // Falls Eingang == VAR
              ,LSG.get_Befehls_Liste().get_Befehl(b).get_input(l)) == 0)
      {
       cout << "OUT_V" << VAR.get_Name(j) << " <= ";
       cout << "OVAR_" << VAR.get_Name(j) << "_O" << ";" << endl;
      };
};

void Netzliste::out_fast(int i)
{
   int b;
   b = LSG.get_Assignment(i).get_element(0);
   for(int l=0; l< LSG.get_Befehls_Liste().
                    get_Befehl(b).get_in_anzahl(); l++)  // Anz eing = l
    for(int j=0; j<VAR.get_anzahl(); j++)
     if(strcmp(VAR.get_Variable(j).get_name()   // Falls Ausgang == VAR
              ,LSG.get_Befehls_Liste().get_Befehl(b).get_input(l)) == 0)
      {
// fast
       if (VAR.get_use_name(j) > 1)
        {
         cout << "OUT_V" << VAR.get_Name(j) << " <= ";
         cout << "M_REG_O_" << j << ";" << endl;
        }
       else
        if (VAR.ist_eingang_name(j) == 1)
         {
          cout << "OUT_V" << VAR.get_Name(j) << " <= ";
          cout << "IN_V" << j << ";" << endl;
         }
        else
         {
          for(int k=0; k<VAR.get_anzahl(); k++)
           if (VAR.get_Name(k) == j)  // Register Allocation
            for(int l1=0; l1< LSG.get_Befehls_Liste().get_Anzahl(); l1++)  // alle BEf
             for(int m=0; m< LSG.get_Befehls_Liste().get_Befehl(l1).get_out_anzahl(); m++)
              if(strcmp(VAR.get_Variable(k).get_name()   // Falls Ausgang == VAR
                       ,LSG.get_Befehls_Liste().get_Befehl(l1).get_output(m)) == 0)
               {
                cout << "OUT_V" << VAR.get_Name(j) << " <= ";
                cout << "OALU_" << LSG.Befehl_auf_Bauteil(l1) << "_O" << m << ";" << endl;
               };
         }; // else




      };  // if for for
};  // out fast
```

```
void Netzliste::alu(int i)
 {
if (LSG.get_Assignment(i).get_anzahl() > 0)  // ALU wird benutzt
 {
 if (strcmp(LSG.get_Assignment(i).get_func(0),
                  "out") == 0)   // out gibts immer nur einmal !!!
  {
   out_slow(i);  // out fast geht nur bei einem Ausgang !!
  }
 else
  if (strcmp(LSG.get_Assignment(i).get_func(0),
                  "in") != 0)
 {

//  cout << "ALU" << i << ": ALU_" << i << endl;
// neu:
  cout << "ALU" << i << ": " << LSG.get_Assignment(i).get_name() << endl;
  cout << "PORT MAP(";
  if (LSG.get_Assignment(i).get_anzahl() > 1)  // Mplex am Eingang
   {
    int k;
    k = 0;
    for(int in = 0; in < LSG.get_Assignment(i).get_anz_eing(); in++)
     {
      if (k==1)
       cout << ",";
      k = 1;
      cout << "M_ALU" << i << "_I" << in << "_O";
     };
   }
  else    // sonst kein Mplex -> nur ein Befehl
   {    // alle eingangsvariabeln durchgehen
    int b;
    int k = 0;
    b = LSG.get_Assignment(i).get_element(0);
    for(int l=0; l< LSG.get_Befehls_Liste().
                    get_Befehl(b).get_in_anzahl(); l++)
     for(int j=0; j<VAR.get_anzahl(); j++)
      if(strcmp(VAR.get_Variable(j).get_name()   // Falls Ausgang == VAR
               ,LSG.get_Befehls_Liste().get_Befehl(b).get_input(l)) == 0)
       {
        if (k == 1) cout << ", ";
        cout << "OVAR_" << VAR.get_Name(j) << "_O";
        k = 1;
       };
   };

  if (LSG.get_Assignment(i).get_anz_func() > 1)
   cout << ", F_" << i;

  for (int j = 0; j<LSG.get_Assignment(i).get_anz_ausg();j++)
   {
    cout << ", OALU_" << i << "_O" << j;
   };
```

```
 cout << ");" << endl;
};  // alles andere als out
}; // if    alu wird benutzt
};  // alu;

void Netzliste::var_multiplexer(int v)
{
 if (VAR.get_use_name(v) > 1)
  {
   cout << "MULTI_V" << v << ": ";
   cout << "MULTI_" << VAR.get_use_name(v) << endl;
   cout << "PORT MAP(";
   int k;
   k = 0;
   for (int b = 0; b < LSG.get_Befehls_Liste().get_Anzahl(); b++)
    {
     for (int n=0;n< LSG.get_Befehls_Liste().get_Befehl(b).get_out_anzahl() ; n++)
      for (int vn=0; vn< VAR.get_anzahl(); vn++)
       {
        if (VAR.get_Name(vn) == v)
         if(strcmp(LSG.get_Befehls_Liste().get_Befehl(b).get_output(n),
                VAR.get_Variable(vn).get_name()) == 0)
          {
           if (strcmp(LSG.get_Befehls_Liste().get_Befehl(b).get_function(),
                  "in") == 0)
            {
             if (k==1) cout << ", ";
             cout << "IN_V" << v;
             k = 1;
            }
           else
            {
             if (k==1) cout << ", ";
             cout << "OALU_" << LSG.Befehl_auf_Bauteil(b) << "_O" << n;
             k = 1;
            };
          }; // if if
       }; // for for
    }; // for
   cout << ", " << "SV_" << v << ", M_REG_O_" << v;
   cout << ");" << endl << endl;
  }; // if
};


void Netzliste::var2_multiplexer(int v)
{
 if (VAR.get_bau_use_name(v,LSG) > 1)
  {
   cout << "MULTI_V" << v << ": ";
   cout << "MULTI_" << VAR.get_bau_use_name(v,LSG) << endl;
   cout << "PORT MAP(";
   int k;
   k = 0;
   for (int a = 0; a < LSG.get_Anzahl(); a++)  // alle assignments
    {
    for (int b =0; b < LSG.get_Assignment(a).get_anzahl(); b++)  // alle bef
     for (int o =0; o < LSG.get_Befehls_Liste().
```

```
                    get_Befehl(LSG.get_Assignment(a).get_element(b)).get_out_anzahl();
        o++)  // alle ausg
     for(int j=0; j< VAR.get_anzahl(); j++)
      if (VAR.get_Name(j) == v)    // Variablen raussuchen
       if(strcmp(VAR.get_Variable(j).get_name()   // Falls AUsgang == VAR
             ,LSG.get_Befehls_Liste().
               get_Befehl(LSG.get_Assignment(a).get_element(b)).get_output(o)) == 0)
        {
         if (strcmp(LSG.get_Befehls_Liste().get_Befehl(
              LSG.get_Assignment(a).get_element(b)).get_function(),
                  "in") == 0)
          {
           if (k==1) cout << ", ";
           cout << "IN_V" << v;
           k = 1;
          }
         else
          {
           if (k==1) cout << ", ";
           cout << "OALU_"
                << LSG.Befehl_auf_Bauteil(LSG.get_Assignment(a).get_element(b))
                << "_O" << o;
           k = 1;
          };
         b = LSG.get_Assignment(a).get_anzahl();
         o = LSG.get_Befehls_Liste().
             get_Befehl(LSG.get_Assignment(a).get_element(b)).get_out_anzahl();
        j = VAR.get_anzahl();

        }; // if if for for
    };  // for;

   cout << ", " << "SV_" << v << ", M_REG_O_" << v;
   cout << ");" << endl << endl;

  };  // if

};


void Netzliste::alu_multiplexer(int a)
{
 if(LSG.get_Assignment(a).get_anzahl() > 1)
  {
   for(int i=0; i < LSG.get_Assignment(a).get_anz_eing(); i++)
    {
     cout << "MULTI_A" << a << "_I" << i << " : ";
     cout << "MULTI_" << LSG.get_Assignment(a).get_anzahl() << endl;
     cout << "PORT MAP(";
     int k;
     k = 0;
     for (int b = 0; b < LSG.get_Befehls_Liste().get_Anzahl(); b++)
      if (LSG.Befehl_auf_Bauteil(b) == a)
       // immer den iten Befehl
       {
        if (k==1) cout << ", ";
        cout << "OVAR_";
        cout << VAR.get_Name(VAR.var_nr
```

```
                    (LSG.get_Befehls_Liste().get_Befehl(b).get_input(i)));
       cout << "_O";
       k = 1;
       }; // if
      cout << ", SA_" << a << ", M_ALU" << a << "_I" << i<< "_O";
      cout << ");" << endl << endl;
    };  // for
  }; // if
};  // alu_multi

void Netzliste::signale_var()
{
 for(int v=0; v < VAR.rename(); v++)
  {
   cout << "SIGNAL OVAR_" << v << "_O" << " : INTEGER;" << endl;
         // Ausgaenge der Regs
   if (VAR.get_bau_use_name(v,LSG) > 1)
    cout << "SIGNAL M_REG_O_" << v << " : INTEGER;" << endl;
       // Ausgaenge der Mplexer
  };
};

void Netzliste::signale_alu()
{
 for (int a=0; a < LSG.get_Anzahl(); a++)
  {
   if(LSG.get_Assignment(a).get_anzahl() > 1)   // es gibt Mplex
    for (int i=0; i< LSG.get_Assignment(a).get_anz_eing(); i++)
     {
      cout << "SIGNAL M_ALU" << a << "_I" << i << "_O";
      cout << " : INTEGER;" << endl;
     };
   if(LSG.get_Assignment(a).get_anzahl() > 0)  // ALU wird benutzt
    {
     for (int n=0; n< LSG.get_Assignment(a).get_anz_ausg(); n++)
      {
       cout << "SIGNAL OALU_" << a << "_O" << n;
       cout << " : INTEGER;" << endl;   // jeder Ausgang von den ALUs
      };
    };
  };  // for
};

void Netzliste::alle_componenten()
{

  register_component();
 for (int m=2; m < 23; m++)
  {
   int l;
   l = 0;
   for (int a=0; a < LSG.get_Anzahl(); a++)
    if(LSG.get_Assignment(a).get_anzahl() == m)
     {
      mult_component(m);
      cout << endl;
      a = LSG.get_Anzahl();
      l = 1;
```

```
      };

    if (l == 0)
     for (int v=0; v < VAR.rename(); v++)
      if(VAR.get_bau_use_name(v,LSG) == m)
       {
        mult_component(m);
        cout << endl;
        v = VAR.rename();
       };

   };  // for m

/*

  for (int a= 0 ; a< LSG.get_Anzahl(); a++)
   {
    if(LSG.get_Assignment(a).get_anzahl() > 0)
     {
      alu_component(a);
      cout << endl;
     };
   };

*/

//neu:

  for (int a= 0 ; a< LSG.get_Anzahl(); a++)
   {
    int ti;
    ti = 0;
    if (LSG.get_Assignment(a).get_anzahl() > 0)
     {
      for(int c = 0; c < a; c++)
       if ((strcmp(LSG.get_Assignment(a).get_name(),
                LSG.get_Assignment(c).get_name()) == 0) &&
           (LSG.get_Assignment(c).get_anzahl() > 0))
        ti = 1;
      if (ti == 0)
       {
        alu_component(a);
        cout << endl;
       };
     };  // if
   };  // for a


 };

void Netzliste::alle_variablen()
 {
  for(int v=0; v < VAR.rename(); v++)
   {
    var2_multiplexer(v);
    cout << endl;
    variable(v);
    cout << endl;
```

```
  };
 };

void Netzliste::alle_alus()
 {
  for (int a=0; a < LSG.get_Anzahl(); a++)
   {
    if(LSG.get_Assignment(a).get_anzahl() > 0)
     {
      alu_multiplexer(a);
      cout << endl;
      alu(a);
      cout << endl;
     };
   };
 };

void Netzliste::architecture(char* ent, char* arch)
 {
  cout << "ARCHITECTURE " << arch << " OF " << ent << " IS " << endl << endl;
  alle_componenten();
  signale_var();
  signale_alu();

  cout << "BEGIN" << endl << endl;
  alle_variablen();
  alle_alus();
  cout << "END " << arch << ";" << endl;
 };
```

**Controller**

```
class Controller
 {
  private:
   Loesung LSG;
   Variablen_Liste VAR;
   void Umrechnen(int,int);
  public:
   Controller();
   Controller(Loesung, Variablen_Liste);
   void step(int);
   int variable(int,int);   // UEbergabe von step, v
   int var_multiplexer(int,int);  // Steuerung step, v
   int var2_multiplexer(int,int);
   int alu_multiplexer(int,int);  // Steuerung step, a
   int alu(int,int);              // Steuerung step, a -> F
   void alles();
   void ROM_package(char*, int);
   void make_ROM(char*, char*, char*, int);
 }; // Controller.h

Controller::Controller(Loesung L, Variablen_Liste VL)
 {
  LSG = L;
  VAR = VL;
 };
```

```
Controller::Controller()
 {
 };

void Controller::step(int i)
 {

  for(int v = 0; v<VAR.rename(); v++)  // Alle Variablen
       {
        variable(i,v);
       };

//  cout << "|";

  for(v = 0; v<VAR.rename(); v++)  // Alle Variablen
       {
        if (VAR.get_bau_use_name(v,LSG) > 1)
         var2_multiplexer(i,v);
       };

//  cout << "|";


  for (int a= 0; a<LSG.get_Anzahl(); a++)  // Alle ALUs
       {
        if (LSG.get_Assignment(a).get_anzahl() > 1)
         alu_multiplexer(i,a);
       };

//  cout << "|";


  for (a= 0; a<LSG.get_Anzahl(); a++)
       {
        if (LSG.get_Assignment(a).get_anzahl() > 0)   // Falls benutzt
         if (LSG.get_Assignment(a).get_anz_func() > 1)  // Falls mehr als eine F
          {
           alu(i,a);
          };
       };


 };  // step

int Controller::variable(int i,int v)
 {
  if(v>=VAR.rename())
   return 0;

  for (int j= 0;j<VAR.get_anzahl(); j++)
   if ((strcmp(VAR.get_Variable(j).get_name(),"STEP") == 0)  &&
       (VAR.get_Name(j) == v))
    {
     cout << "1";
     return 1;
    };

  if (i==0)
```

```
  {
   if (VAR.ist_eingang_name(v) == 1)
    cout << "1";
   else cout << "0";
   }
  else
   {
   int o;
   o = 0;
   for(int va=0; va < VAR.get_anzahl(); va++)
    if(VAR.get_Name(va) == v)   // alle Variabeln mit Nr == v
     for (int b=0; b < LSG.get_Befehls_Liste().get_Anzahl(); b++)  //alle Befehle
      if((LSG.get_steuer(b) +
         LSG.get_Assignment(LSG.Befehl_auf_Bauteil(b)).
            get_Time1(LSG.get_Befehls_Liste().
                     get_Befehl(b).get_function()) - 1) == i)
      for(int au=0; au < LSG.get_Befehls_Liste().get_Befehl(b).get_out_anzahl(); au++)
       if(strcmp(LSG.get_Befehls_Liste().get_Befehl(b).get_output(au),
              VAR.get_Variable(va).get_name()) == 0)
        {
         if (strcmp(LSG.get_Befehls_Liste().get_Befehl(b).get_function(),"in") != 0)
          {
           cout << "1";
           o = 1;
          };
        };
   if (o == 0) cout << "0";
  }; // else;
 return 1;
}; // variable;

void Controller::Umrechnen(int x,int n)  // n ist max -> bin darstellung von x !!
   {
    for(int i = ceil(log10(n+1)/log10(2)) ; i > 0; i--)
     {
      if (x/pow(2,i-1) >= 1)
       {
        x = x - pow(2,i-1);
        cout << "1";
       }
      else cout << "0";   // auch bei -1 werden 0 ausgegeben !
     };
   };

int Controller::var_multiplexer(int i,int v)
{
 if (VAR.get_use_name(v) <= 1)
  return 0;
 if(v>=VAR.rename())
  return 0;
 if (i==0)
  {
   if (VAR.ist_eingang_name(v) == 1)
    Umrechnen(0,VAR.get_use_name(v)-1);
   else
    Umrechnen(0,VAR.get_use_name(v)-1);
  }
 else
```

```
  {
   int o;
   o = 0;
   int count;
   count = 0;

    for (int b=0; b < LSG.get_Befehls_Liste().get_Anzahl(); b++)  //alle Befehle
     {
      for(int va=0; va < VAR.get_anzahl(); va++)
       if(VAR.get_Name(va) == v)   // alle Variabeln mit Nr == v
        for(int au=0; au < LSG.get_Befehls_Liste().get_Befehl(b).get_out_anzahl(); au++)
         if(strcmp(LSG.get_Befehls_Liste().get_Befehl(b).get_output(au),
                VAR.get_Variable(va).get_name()) == 0)
          {
           count++;
            if((LSG.get_steuer(b) +
             LSG.get_Assignment(LSG.Befehl_auf_Bauteil(b)).
                get_Time1(LSG.get_Befehls_Liste().
                        get_Befehl(b).get_function()) - 1) == i)
             {
              if (strcmp(LSG.get_Befehls_Liste().get_Befehl(b).get_function(),"in") != 0)
               {
                Umrechnen(count-1,VAR.get_use_name(v)-1);
                o = 1;
               };
             };
            };
     }; // for b

   if (o == 0)
    Umrechnen(0,VAR.get_use_name(v)-1);

  }; // else;
 return 1;
};


int Controller::var2_multiplexer(int i, int v)
 {
  if (VAR.get_bau_use_name(v,LSG) <= 1)
   return 0;
  if(v>=VAR.rename())
   return 0;
/*
  if (i==0)
   {
    if (VAR.ist_eingang_name(v) == 1)
     Umrechnen(0,VAR.get_bau_use_name(v,LSG)-1);
    else
     Umrechnen(0,VAR.get_bau_use_name(v,LSG)-1);
   }
  else
   {
*/
    int ou;
    ou = 0;
    int count;
    count = 0;
```

```
   int c_ja;
   c_ja = 0;

   for (int a = 0; a < LSG.get_Anzahl(); a++)  // alle assignments
    {
    for (int b =0; b < LSG.get_Assignment(a).get_anzahl(); b++)  // alle bef bzgl Assi
     for (int o =0; o < LSG.get_Befehls_Liste().
          get_Befehl(LSG.get_Assignment(a).get_element(b)).get_out_anzahl(); o++)  // alle ausg
      for(int j=0; j< VAR.get_anzahl(); j++)
       if (VAR.get_Name(j) == v)    // Variablen raussuchen
        if(strcmp(VAR.get_Variable(j).get_name()   // Falls AUsgang == VAR
              ,LSG.get_Befehls_Liste().get_Befehl(
               LSG.get_Assignment(a).get_element(b)).get_output(o)) == 0)
         {

           c_ja = 1;
            if((LSG.get_steuer(LSG.get_Assignment(a).get_element(b)) +
             LSG.get_Assignment(LSG.Befehl_auf_Bauteil(LSG.get_Assignment(a).get_element(b))).
                get_Time1(LSG.get_Befehls_Liste().
                        get_Befehl(
                     LSG.get_Assignment(a).get_element(b)).get_function()) - 1) == i)
            {
            //  if (strcmp(LSG.get_Befehls_Liste().get_Befehl(
            //      LSG.get_Assignment(a).get_element(b)).get_function(),"in") != 0)
            //  {
               Umrechnen(count,VAR.get_bau_use_name(v,LSG)-1);
               ou = 1;
               b = LSG.get_Assignment(a).get_anzahl();
               o = LSG.get_Befehls_Liste().
                 get_Befehl(LSG.get_Assignment(a).get_element(b)).get_out_anzahl();
               j = VAR.get_anzahl();

            //  };
            };


          }; // if if for for for
        if (c_ja == 1) count++;
        c_ja = 0;

   }; // for a

   if (ou == 0)
    Umrechnen(0,VAR.get_bau_use_name(v,LSG)-1);

//  };
// else;
  return 1;
 };


int Controller::alu_multiplexer(int i,int a)
 {
  if (LSG.get_Assignment(a).get_anzahl() < 2)
   return 0;
  int o;
  o = 0;
  int count;
```

```
 count = 0;
 for(int b=0; b < LSG.get_Befehls_Liste().get_Anzahl(); b++)  //alle Befehle
  if(LSG.Befehl_auf_Bauteil(b) == a)
   {
    count++;
    if((LSG.get_steuer(b) <= i) &&
      ((LSG.get_steuer(b) + LSG.get_Assignment(a).
           get_Time3(LSG.get_Befehls_Liste().
                     get_Befehl(b).get_function()
                ) - 1) >= i))
     {
      Umrechnen(count-1,LSG.get_Assignment(a).get_anzahl() -1);
      o = 1;
     };
   };
 if (o == 0)
  Umrechnen(0,LSG.get_Assignment(a).get_anzahl() -1);
 return 1;
};

int Controller::alu(int i ,int a)
 {
 // if (LSG.get_Assignment(a).get_anzahl() < 2)
 //  return 0;
 int o;
 o = 0;
 int count;
 count = 0;
 for(int b=0; b < LSG.get_Befehls_Liste().get_Anzahl(); b++)  //alle Befehle
  if(LSG.Befehl_auf_Bauteil(b) == a)
   {
    if((LSG.get_steuer(b) <= i) &&
      ((LSG.get_steuer(b) + LSG.get_Assignment(a).
           get_Time2(LSG.get_Befehls_Liste().
                     get_Befehl(b).get_function()
                ) - 1) >= i))
     {
      Umrechnen(LSG.get_Assignment(a).get_func_nr(LSG.get_Befehls_Liste().
                                          get_Befehl(b).get_function())
           ,LSG.get_Assignment(a).get_anz_func() -1);
      o = 1;
     };
   };
 if (o == 0)
  Umrechnen(0,LSG.get_Assignment(a).get_anz_func() -1);
 return 1;
};

void Controller::alles()
 {
 cout << "ROM_WORD'(" << '\"';
 step(0);
 for(int i=1; i < LSG.get_Bewertung(); i++)
  {
   cout << '\"' << ")," << endl;
   cout << "ROM_WORD'(" << '\"';
   step(i);
  };
```

```
   cout << '\"' << "));" << endl;
 };

void Controller::ROM_package(char* name, int anz)  // anzahl steuerleitungen
 {
 cout << "package " << name << " is" << endl;
 cout << "constant ROM_WIDTH: INTEGER := ";
 cout << anz << ";" << endl;
 cout << "subtype ROM_WORD is BIT_VECTOR (1 TO ROM_WIDTH); " << endl;
 cout << "subtype ROM_RANGE is INTEGER range 0 to " << (LSG.get_Bewertung() -1);
 cout << ";" <<endl;
 cout << "type ROM_TABLE is array (0 to " << (LSG.get_Bewertung() -1);
 cout << ") of ROM_WORD; " << endl;
 cout << " constant ROM: ROM_TABLE := ROM_TABLE'(" << endl;
 alles();
 cout << "end " << name << ";" << endl;
 };


void Controller::make_ROM(char* ent, char* arch, char* pack, int width)
 {
 // entity
 cout << "use work." << pack << ".all;" << endl;
 cout << "entity " << ent << " is" << endl;
 cout << " port (" << endl;
 cout << "  STEP_I : in INTEGER;" << endl;
 cout << "  CLOCK: in BIT;" << endl;
 cout << "  RESET: in Bit;" << endl;
 cout << "  WOUT : out Bit_Vector(1 to " << width << ");" << endl;
 cout << "  STEP_O : out INTEGER);" << endl;
 cout << "end " << ent << ";" << endl << endl;

 // architecture
 cout << "ARCHITECTURE " << arch << " OF " << ent << " IS" << endl;
 cout << "SIGNAL STEP: ROM_RANGE;" << endl;
 cout << "BEGIN" << endl;
 cout << " TIMESTEP_COUNTER: process " << endl;
 cout << " BEGIN" << endl;
 cout << "  wait until CLOCK'event and CLOCK = '1'; " << endl;
 cout << "  if RESET = '1' then" << endl;
 cout << "   STEP <= ROM_RANGE'low; " << endl;
 cout << "  elsif STEP = ROM_RANGE'high then " << endl;
 cout << "   STEP <= 0; " << endl;
 cout << "  else" << endl;
 cout << "   STEP <= STEP_I;" << endl;
 cout << "  END if;  " << endl;
 cout << " END process TIMESTEP_COUNTER;" << endl;
 cout << "STEP_O <= STEP + 1;" << endl;
 cout << "WOUT <= ROM(STEP); " << endl;
 cout << "END " << arch << ";" << endl;
 };
```

**Gesamtnetz**

```
class Gesamtnetz
 {
  private:
   Loesung LSG;
```

```
  Variablen_Liste VAR;
  Netzliste NETZ;
  Controller CONT;
 public:
  Gesamtnetz(Loesung, Variablen_Liste);
  void make_componenten(char*, char*, char*, char*, char*);
  void entity(char*);  // Keine Steuerleitung !!
  int netz_component(char*, char*);
  void cont_component(char*, char*, int);
  void map_netz();
  void map_controller();
  void architecture(char*, char*, char*, char*, char*, char*);
 }; // Gesamtnetz.h

Gesamtnetz::Gesamtnetz(Loesung L, Variablen_Liste VL)
 {
 LSG = L;
 VAR = VL;
 NETZ = Netzliste(L,VL);
 CONT = Controller(L,VL);
 };

void Gesamtnetz::make_componenten(char* netz_e,
                        char* netz_a,
                        char* cont_e,
                        char* cont_a,
                        char* pack)
 {
 int anzst;
 anzst = NETZ.entity(netz_e);
 NETZ.architecture(netz_e,netz_a);
 cout << endl;

 CONT.ROM_package(pack,anzst);
 CONT.make_ROM(cont_e,cont_a,pack,anzst);
 };

void Gesamtnetz::entity(char* name)
 {
    cout << "ENTITY " << name << " IS" << endl;

    cout << "PORT (" << endl;
    int k;
    cout << "CLOCK : in Bit;" << endl;
    cout << "RESET : in Bit;" << endl;

// Eingaenge

    k = 0;
    for (int i= 0; i<VAR.rename(); i++)
     {
      if (VAR.ist_eingang_name(i) == 1)
       {
        if (k == 1)
         {
          cout << ",";
         };
        k = 1;
```

```
        cout << "I_V" << i;
       }; // if
      };  // for
     cout << " : in INTEGER; " << endl;

// Ausgaenge

     k = 0;
     for (i= 0; i<VAR.rename(); i++)
      {
       if (VAR.ist_ausgang_name(i) == 1)
        {
         if (k == 1)
          {
           cout << ",";
          };
         k = 1;
         cout << "O_V" << i;
        }; // if
      };  // for
     cout << " : out INTEGER " << endl;

     cout << ");" << endl;
     cout << "END " << name << ";" << endl;
};


int Gesamtnetz::netz_component(char* name, char* arch)
 {
  cout << "COMPONENT " << name << endl;
  cout << " PORT(";
        // entity erzeugen
  int ret;
  ret = 0;

  int k;

// Eingaenge

     k = 0;
     for (int i= 0; i<VAR.rename(); i++)
      {
       if (VAR.ist_eingang_name(i) == 1)
        {
         if (k == 1)
          {
           cout << ",";
          };
         k = 1;
         cout << "IN_V" << i;
        }; // if
      };  // for
     cout << " : in INTEGER; " << endl;

// Ausgaenge

     k = 0;
     for (i= 0; i<VAR.rename(); i++)
```

```
      {
       if (VAR.ist_ausgang_name(i) == 1)
        {
         if (k == 1)
          {
           cout << ",";
          };
         k = 1;
         cout << "OUT_V" << i;
        }; // if
      };  // for
     cout << " : out INTEGER; " << endl;

/* Inouts

     k = 0;
     for (i= 0; i<VAR.rename(); i++)
      {
       if ((VAR.ist_eingang_name(i) == 1)
        && (VAR.ist_ausgang_name(i) == 1))
        {
         if (k == 1)
          {
           cout << ",";
          };
         k = 1;
         cout << "INOUT_V" << i;
        }; // if
      };  // for
     cout << " : inout INTEGER; " << endl;
*/

// Clock
     cout << " C : in Bit; " << endl;

// D ..

     k = 0;
     for (i= 0; i<VAR.rename(); i++)
      {
       if (k == 1)
        {
         cout << ",";
        };
       k = 1;
       cout << "D_" << i;
      };
     cout << " : in Bit; " << endl;

ret = VAR.rename();

// Variablen Multiplexer
     int semi;

     semi = 0;
     for (i= 0; i<VAR.rename(); i++)
      {
       if (VAR.get_bau_use_name(i,LSG) > 1)
```

```
        {
         if (semi == 1) cout << ";" << endl;
         semi = 1;
         cout << "SV_" << i;
         cout << " : in Bit_Vector(0 to ";
         cout << (ceil(log10(VAR.get_bau_use_name(i,LSG))/log10(2)) - 1);
         cout << ")";
         ret = ret + (ceil(log10(VAR.get_bau_use_name(i,LSG))/log10(2)));
        };
       };

// ALU Multiplexer

     for (i= 0; i<LSG.get_Anzahl(); i++)
      {
       if (LSG.get_Assignment(i).get_anzahl() > 1)
        {
         if (semi == 1) cout << ";" << endl;
         semi = 1;
         cout << "SA_" << i;
         cout << " : in Bit_Vector(0 to ";
         cout << (ceil(log10(LSG.get_Assignment(i).get_anzahl())/log10(2)) - 1);
         cout << ")";
         ret = ret + (ceil(log10(LSG.get_Assignment(i).get_anzahl())/log10(2)));
        };
       };

// ALU Funktionen

     for (i= 0; i<LSG.get_Anzahl(); i++)
      {
       if (LSG.get_Assignment(i).get_anzahl() > 0)   // Falls benutzt
        if (LSG.get_Assignment(i).get_anz_func() > 1)  // Falls mehr als eine F
         {
          if (semi == 1) cout << ";" << endl;
          semi = 1;
          cout << "F_" << i;
          cout << " : in Bit_Vector(0 to ";
          cout << (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2)) - 1);
          cout << ")";
          ret = ret + (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2)));
         };  // if if
       };  // for

     cout << ");" << endl;
     cout << "END COMPONENT;" << endl;

     cout << "for all: " << name << endl;
     cout << "  use entity work." << name << "(" << arch << ");" << endl;

 return ret;  // Anzahl der Steuerleitungen
};  // end entity

void Gesamtnetz::cont_component(char* name, char* arch, int width)
 {
  cout << "COMPONENT " << name << endl;
  cout << " PORT (" << endl;
  cout << "  STEP_I : in INTEGER;" << endl;
```

```
 cout << "  CLOCK: in BIT;" << endl;
 cout << "  RESET: in Bit;" << endl;
 cout << "  WOUT : out Bit_Vector(1 to " << width << ");" << endl;
 cout << "  STEP_O : out INTEGER);" << endl;
 cout << "end " << "COMPONENT" << ";" << endl << endl;
 cout << "for all: " << name << endl;
 cout << "  use entity work." << name << "(" << arch << ");" << endl;
};

void Gesamtnetz::map_netz()
{
// Eingaenge
int ret;
ret = 0;
int k;

cout << "(" << endl;

    k = 0;
    for (int i= 0; i<VAR.rename(); i++)
     {
      if (VAR.ist_eingang_name(i) == 1)
       {
        int w;
        w = 0;
        for (int j= 0;j<VAR.get_anzahl(); j++)
         {
          if ((strcmp(VAR.get_Variable(j).get_name(),"STEP") == 0) &&
              (VAR.get_Name(j) == i))
           {
            if (k == 1)
             {
              cout << ",";
             };
            k = 1;
            cout << "S_OUT";
            w = 1;
            j = VAR.get_anzahl();
           }; // if
         };  // for

        if (w == 0)
         {
           if (k == 1)
            {
             cout << ",";
            };
            k = 1;
            cout << "I_V" << i;
         };  // if w
       }; // if
     };  // for

cout << endl;
// Ausgaenge

    for (i= 0; i<VAR.rename(); i++)
     {
```

```
        if (VAR.ist_ausgang_name(i) == 1)
         {
          int w;
          w = 0;
          for (int j= 0;j<VAR.get_anzahl(); j++)
           {
            if ((strcmp(VAR.get_Variable(j).get_name(),"STEP") == 0) &&
                (VAR.get_Name(j) == i))
             {
              if (k == 1)
               {
                cout << ",";
               };
              k = 1;
              cout << "S_IN";
              w = 1;
              j = VAR.get_anzahl();
             }; // if
           };  // for

          if (w == 0)
           {
            if (k == 1)
             {
              cout << ",";
             };
            k = 1;
            cout << "O_V" << i;

           }; // if w

         }; // if
        };  // for

cout << endl;

/* Inouts

      for (i= 0; i<VAR.rename(); i++)
       {
        if ((VAR.ist_eingang_name(i) == 1)
         && (VAR.ist_ausgang_name(i) == 1))
         {
          if (k == 1)
           {
            cout << ",";
           };
          k = 1;
          cout << "INOUT_V" << i;
         }; // if
       };  // for
      cout << " : inout INTEGER; " << endl;
cout << endl;

*/

// Clock
          if (k == 1)
```

```
        {
         cout << ",";
        };

        cout << " CLOCK " << endl;
        k = 1;

// D ..

    for (i= 0; i<VAR.rename(); i++)
     {
      if (k == 1)
       {
        cout << ",";
       };
      k = 1;
      cout << "ST(" << i+1 << ")";
     };

cout << endl;
ret = VAR.rename();

// Variablen Multiplexer

    for (i= 0; i<VAR.rename(); i++)
     {
      if (VAR.get_bau_use_name(i,LSG) > 1)
       {
        if (k == 1) cout << ",";
        k = 1;
        cout << "ST(" << ret + 1 << " TO ";
        cout << (ret + (ceil(log10(VAR.get_bau_use_name(i,LSG))/log10(2))));
        cout << ")";
        ret = ret + (ceil(log10(VAR.get_bau_use_name(i,LSG))/log10(2)));
       };
     };

cout << endl;

// ALU Multiplexer

    for (i= 0; i<LSG.get_Anzahl(); i++)
     {
      if (LSG.get_Assignment(i).get_anzahl() > 1)
       {
        if (k == 1) cout << ",";
        k = 1;
        cout << "ST(" << ret + 1 << " TO ";
        cout << (ret + (ceil(log10(LSG.get_Assignment(i).get_anzahl())/log10(2))));
        cout << ")";
        ret = ret + (ceil(log10(LSG.get_Assignment(i).get_anzahl())/log10(2)));
       };
     };

cout << endl;

// ALU Funktionen
```

```
    for (i= 0; i<LSG.get_Anzahl(); i++)
     {
      if (LSG.get_Assignment(i).get_anzahl() > 0)   // Falls benutzt
       if (LSG.get_Assignment(i).get_anz_func() > 1)  // Falls mehr als eine F
        {
         if (k == 1) cout << ",";
         k = 1;
         cout << "ST(" << ret + 1 << " TO ";
         cout << (ret + (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2))));
         cout << ")";
         ret = ret + (ceil(log10(LSG.get_Assignment(i).get_anz_func())/log10(2)));
        };  // if if
     };  // for

    cout << ");" << endl;
};

void Gesamtnetz::map_controller()
 {
  cout << "(" << endl;
  cout << "S_IN, " << endl;
  cout << "CLOCK, RESET, " << endl;
  cout << "ST," << endl;
  cout << "S_OUT);" << endl;
 };

void Gesamtnetz::architecture(char* ent, char* arch,
                              char* netz_e, char* netz_a,
                              char* co_e, char* co_a)
 {
  cout << endl;
  cout << "ARCHITECTURE " << arch << " OF " << ent << " IS " << endl << endl;
  int a;
  a = netz_component(netz_e,netz_a);
  cout << endl;
  cont_component(co_e,co_a,a);
  cout << endl;

 // geht noch weiter ....

  cout << "SIGNAL ST : Bit_Vector(1 to " << a << ");" << endl;
  cout << "SIGNAL S_IN : INTEGER;" << endl;    // N -> C
  cout << "SIGNAL S_OUT : INTEGER;" << endl;  // C -> N

  cout << "BEGIN" << endl;
// Die beiden Componenten
  cout << netz_e << "_I" << " : " << netz_e << endl;
  cout << "PORT MAP " << endl;
  map_netz();
  cout << endl;
  cout << co_e << "_I : " << co_e << endl;
  cout << "PORT MAP " << endl;
  map_controller();
  cout << endl;
  cout << "END " << arch << ";" << endl;
  };
```

## Literatur


[1] Aarts, E. H. L. ; Simulated Annealing and Boltzmann Machines, Wiley, Chichester, UK, 1989.

[2] Achatz H. ; Extended 0/1 LP formulation for the scheduling problem in high-level synthesis ; EURO-DAC ’93 ; 1993.

[3] Aiken A. , Nicolau A. ; Optimal loop parallelisation ; Proc. SIGPLAN ’88 Conf. on Programming, Design and Implementation ; Atlanta 1988.

[4] Airiau R. ; Berge J.-M. ; Olive V. ; Circuit Synthesis with VHDL; Kluwer Academic Publishers 1994.

[5] Albert J. , Hinker S. , Schoof J. ; Parallele Evolutionäre Algorithmen auf einem heterogenen Workstation-Cluster ; SIWORK 1996.

[6] Albertz J. (Hrsg.) Evolution und Evolutionsstrategien in Biologie, Technik und Gesellschaft, 2. Aufl., Wiesbaden: Freie Akademie 1990.

[7] Amann R. , Neher M. , Rietsche G. ; Rosenstiel W. ;CASTOR: FSM Synthesis in a Digital Curcuit Synthesis System ; Proc. ESSCIRC 1989.

[8] Armstrong J.-R. ; Chip-Level Modeling with VHDL; Prentice Hall 1989.

[9] Armstrong J.-R. ; Gray F. G. ; Structured Logic Design with VHDL ; Prentce Hall 1993.

[10] Aßmann C. ; FAST - Eine Prozessor-Architektur mit einem neuartigen Stacksystem ; Dissertationsschrift ; Cristian-Albrechts-Universität Kiel 1992

[11] Bähring, H. ; Mikrorechner-Systeme, Springer Lehrbuch, 2. Auflage, Springer Verlag, Berlin 1994.

[12] Banerjee U. ; Loop Transformations for Restructuring Compilers: The Foundations; Kluwer Academic; Boston 1993.

[13] Barros E. , Rosenstiel W. ; A Method for Hardware/Software Partitioning; Proc. Compeuro 1992.

[14] Beckmann R. et al. ; The TREEMOLA Language Reference Manual Version 4.0 ; Report Nr. 391 ; Universität Dortmund 1991.

[15] Bhasker J. ; A VHDL Primer; Prentice Hall; 1993.

[16] Biesenack J. et al. ; The Siemens High-Level Synthesis System CALLAS ; 4th High-Level Synthesis Workshop 1992.

[17] Bolc L. , Borowik P. ; Many Valued Logics 1 Theoretical Foundations ; Springer Verlag 1992.

[18] Brandstädt A. ; Graphen und Algorithmen; B.G. Teubner Stuttgart; 1994.

[19] Brayton R., Camposano R., DeMicheli G. Otten R.H.J.M., van Eindhoven J. ; The Yorktown Silicon Compiler System ; IBM Research Report RC 12500 ; Mathematics 1986.

[20] O’ Brien K. , Rahmouni M. Jerraya A. ; DLS : A Scheduling Algorithm For High-Level Synthesis in VHDL ; Proc. European Conference on Design Automation (EDAC) 1993.

[21] Brück, R. ; Entwurfswerkzeuge für VLSI-Layout ; Methoden und Algorithmen für en

Rechnergestützten Entwurf von VLSI-Layout.

[22] F. Burchert, A. Falkenberg, M. Koch, D. Tavangarian: "Ein Funktionsgenerator zur Synthese inhaltsorientierter Speicherarchitekturen", Workshop Custom Computing, Schloß Dagstuhl Juni 1996.

[23] Burkard, R. E. ; Fincke, U. : The Asymptotic Probabilistic Behaviour of Quadratic Sum Assignment Problems, in: Zeitschrift für Operations Research 27 (1983), S. 73-81.

[24] Burkard, R. E. ; Fincke, U. : Probabilistic Asymptotic Properties of Some Combinational Optimization Problems, in : Discrete Applied Mathematics 12 (1985), S. 21 - 29.

[25] Camposano R. ; Path-Based Scheduling for Synthesis, IEEE Transactions on CAD, Vol. 10, No. 1, 1991.

[26] Camposano R. ; Bergamaschi R.A. ; Synthesis using Path-Based Scheduling: Algorithms and Exercises ; 4th High Level Synthesis Workshop; Kennebunkport, 1989.

[27] Camposano R. ; Rosenstiel W. ; Synthesizing Circuits from Behavioural Descriptions ; IEEE Transactions on CAD, Vol. 8, 1989.

[28] Cohen B. ; VHDL Coding Styles and Methodologies - an in Depth Tutorial ; Kluwer Academic Publishers 1995.

[29] Darwin C. ; On the Origin of Species ; Eile ; 1859.

[30] Deo N. ; Graph Theorie with applications to engeneering and computer science ; Prentice Hall 1974.

[31] Devadas S. ; Newton R. ; Algorithms for Hardware Allocation in Data Path Synthesis ; IEEE Transactions on CAD, Vol 8, No. 7, Juli 1989.

[32] Dörfler W. , Mühlbacher J. ; Graphentheorie für Informatiker ; de Gruyter 1973.

[33] Dueck G. ; Scheuer T. ; Wallmeier A.-M. ; Toleranzschwelle und Sintflut: Neue Ideen zur Optimierung ; Spektrum der Wissenschaft März 1993.

[34] H.-J. Eikerling, M. Schmidt, W. Rosenstiel ; Anwendung symbolischer Techniken zur flächenminimierenden HW-Resynthese. In 7. E.I.S.-Workshop, Chemnitz, Germany, November 1995.

[35] Ernst R. et al. ; The COSYMA Environment for Hardware/Software Cosynthesis of Small Embedded Systems ; Microprocessors and Microsystems, Elsevier, Vol. 20, No. 3, pp. 159-166, May 1996.

[36] Even S. , Litman A. ; On the Capabilities of Systolic Systems ; 3rd ACM Symposium on Parallel Algorithms and Architectures ; July 1991.

[37] Fähnrich R. ; Rechnergestützter Entwurf und Simulation zweistufiger Schaltnetze und mikroprogrammierter Steuerwerke ; Dissertation ; Universität Dortmund 1983.

[38] Falkenberg A. ; BiCMOS Technologie ; Seminarvortrag bei der Projektgruppe : BiCMOS Schaltungsextraktor ; Universität Dortmund 1993.

[39] Falkenberg A. ; Entwurf eines Reduktionsmaschienen-Chips ; Diplomarbeit ; Universität Dortmund 1994.

[40] A. Falkenberg, F. Burchert, D. Tavangarian: "A New Approach Towards Accelerating VLSI-Synthesis", The Second World Conference on Integrated Design & Process

Technology (Society for Design and Process Science), Austin/Texas, 1996.

[41] A. Falkenberg; Multiple Criteria Optimization in High Level Synthesis using Genetic Algorithms; 1997 International Symposium on Nonlinear Theory and its Applications (NOLTA'97); Hawaii 1997.

[42] A.Falkenberg; Accelerating High Level Synthesis using fast Allocation; International Conference on Computational Intelligence and Multimedia Applications (ICCIMA '98) ; Churchill Australia.

[43] A. Falkenberg; Multi-criteria Optimization of VLSI-Synthesis using Genetic Algorithms; 1st International Symposium on Communication Systems and Digital Signal Processing, Sheffield, UK, 1998.

[44] A. Falkenberg; High Level Synthesis with Genetic Algorithms - A Theoretical Approach; Computational Engineering in Systems Applications, CESA '98 IMACS-IEEE Multiconference; Hammamet, Tunesia, 1998.

[45] A. Falkenberg; High Level Synthesis with Genetic Algorithms using Multi-Criteria Optimization; The Fourth International Conference on Optimization Techniques and Applications (ICOTA '98); Perth, Australia; 1998.

[46] A. Falkenberg; Parallel Synthesis using Genetic Algorithms in a parallel Workstation Cluster; 2nd International Symposium Tools and Methods for Concurrent Engineering (TMCE '98); Manchester, UK; 1998.

[47] Fengler W. , Philippow I. ; Entwurf industrieller Microcomputersysteme ; Carl Hanser Verlag 1991.

[48] Fernández, Koch, Madrid, Kloos, Rosenstiel ; Hardware-Software Prototyping from LOTOSDAES '97

[49] Fischberg C.J.; Rhie C.M.; Zacharias R.M.; Bradley P.C.; DesSureault T.M.; Using Hundreds of Workstations for Production Running of Parallel CFD Applications; Proceedings of Parallel CFD 1995 Pasadena, CA, USA.

[50] Gajski D.; Dutt N.; Wu A.; Lin S., 1992, "High-Level Synthesis - Introduction to Chip and System Design", Kluwer Academic Publishers, Norwell, Massachusets

[51] Goldberg, D. E. ; Genetic Algorithms in Search, Optimization and Machine Learning, reading/MA ; Addison -Weseley ; 1989.

[52] Goosens G. , Vandewalle J. , De Man H. ; Loop Optimization in Register-Transfer Scheduling for DSP-Systems ; Proceedings of the 26th Design Automation Conference, 1986.

[53] Gould R. Graph Theorie ; Benjamin / Cummings 1988.

[54] Gupta R. ; DeMicheli G. ; Hardware-Software Cosynthesis for Digital Systems, IEEE Design & Test of Computers, Sept. 1993.

[55] Haroun B.S. ; Elmasrey M.I. ; Architectural Synthesis for DSP Silicon Compilers ; IEEE Trans. on CAD ; Vol. 8, No. 4, April 1989.

[56] Heiliger Geist ; Bibel ; 3500 v.Chr - ca. 300 n Chr.

[57] Heistermann J.; Genetische Algorithmen Teubner Texte zu Informatik Band 9; B.G. Teubner 1994

[58] Henkel J. , Ernst R. ; COSYMA Ein System zur Hardware/Software Co-Synthese, GME Fachnericht Nr. 15 Mikroelektronik, S. 167-172, 1995.

[59] Hipper G. , Tavangarian D. ; Ein Konzept für kosteneffiziente Workstation-Cluster mit nebenläufiger Netzarchitektur basierend auf FastEthernet ; SIWORK 1996.

[60] Holland, J. H. ; Adaption in Natural and Artifical Systems ; Ann Arbor ; The University of Michigan Press ; 1975.

[61] Holtmann U. ; Synthese von Co-Prozessoren mit spekulativer Ausführung ; Dissertation; TU Braunschweig ; 1995.

[62] De Jong, K. ; An Analysis of the Behaviour of a Class of Genetic Adaptive Systems; Dissertation; Universitiy of Michigan ; Ann Arbor ; 1975.

[63] De Jong, K. ; Spears W.M. ; Using Genetic Algorithms to Solve NP-Complete Problems, in Proceedings of the third International Conference on Genetic Algorithms; Morgan Kaufmann Publishers; Los Altos CA, 1989.

[64] Kaderali F. , Poguntke W. ; Graphen Algorithmen Netze - Grundlagen und Anwendungen in der Nachrichtentechnik ; Vieweg 1995.

[65] Koch M. ; Verfahren zur Akzeleration der Simulation von VHDL-Modellen ; Fern-Universität Hagen 1995.

[66] Koch M. ; Dicken H. ; Burchert F. VHDL-Modellierung, Simulation, Synthese; Teil 1 bis 8; ELRAD 97; Heise Verlag.

[67] Koopmans, T. C. ; Beckmann, M. J. ; Assignment Problems and the Location of Economic Activities, in: Econometrica 25 (1957), S. 53-76.

[68] Krämer H. ; Automatische Synthese von parallelen Prozessor-Strukturen für VLSI-Schaltungen ; Dissertation ; Universität Karlsruhe ; 1993.

[69] Ku D. ; DeMicheli G. ; High-Level Synthesis and Optimization Strategies in Hercules and Hebe ; Proc. IFIP-Conference on Logic and Architectural Synthesis; 1990.

[70] Kurdahi F.J. ; Parker A.C. ; REAL : A Program for Register Allocation ; Proc. of 24th Design Automation Conference , 1987.

[71] Ladage L. ; Kompaktierung heterogener Maskenlayouts mit komplexen Restriktionen ; Berichte aus der Informatik ; Shaker 1995.

[72] Landwehr B. , Marwedel P., Doemer R. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. Universität Dortmund, Internal report no. 484, 1994.

[73] De Man H. , Rabaey J. , Six P. ; Cathedral II : A Synthesis and Module Generation System for Multiprocessor Systems on a Chip ; Design Systems for Logic Synthesis ; Martinus Nijhoff Publishers 1987.

[74] Marwedel P. et al. ; The Mimola Language Version 4.1 Universität Dortmund 1994.

[75] Marwedel P. ; Synthese und Simulation von VLSI-Systemen ; Hanser Verlag ; München 1993.

[76] Marwedel P. ; Introducing Complex Components into Architectural Synthesis ;Asia South Pacific Design Automation Conference (ASP-DAC), Chiba, Japan, 1997.

[77] Mehlhorn K. Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness; Springer-Verlag 1984.

[78] Metropolis N. ; Rosenbluth; Teller; Equation of State Calcululations by fast Computing Machine; Journal of Chemical Physics Vol. 21 1953.

[79] Michalewicz Z. ; Genetic Algorithms + Data Structures = Evolution Programs ; Springer-Verlag 1992.

[80] De Michelli G. , Ku D. HERKULES - A System for High-Level Synthesis ; Proc. of 25th Design Automation Conference 1988.

[81] Nicolau A. ; Percolation Scheduling : A parallel compilation technique, Technical Report TR 85-678 ; Cornell University; Ithaca USA 1985.

[82] Nicolau A. ; Potasman R. ; Incremental Tree Height Reduction for High-Level Synthesis; Proc. Design Automation Conference (DAC) ,1991.

[83] Nissen V. Evolutionäre Algorithmen; Darstellung, Beispiele, betriebswirtschaftliche Anwendungsmöglichkeiten; DUV, Wiesbaden 1994.

[84] Note S. ; Catthoor F. ; DeMan H. ; Hardwired data Path Synthesis for High Speed DSP Systems with the Cathedral III Compilation Environment ; Proc. of Intl. Workshop on Logic and Architectural Synthesis for Silicon Compilers, 1988.

[85] Ott D. E. ; Wilderotter T. J. ; A Designer's Guide to VHDL Synthesis ; Kluwer Academic Publishers 1994.

[86] Ouweneel W.J. ; Evolution in der Zeitenwende ; Biologie und Evolutionslehre - Die Folgen des Evolutionismus ; Christliche Schriftenverbreitung ; Hückeswagen.

[87] Pangrle B.M. ; Gajski D. ; Design Tools for Intelligent Silicon Compilation ; IEEE Trans. on CAD ; 1987.

[88] Pangrle B.M. ; Splicer: A heuristic Approach to Connectivity Binding; Proc. of 25th Design Automation Conference; 1988.

[89] Parker A.C. ; Pizarro J. ; Mlinar M. ; MAHA: A Program for Datapath Synthesis ; Proc. of 23rd Design Automation Conference , 1986.

[90] Paulin P.G., Knight J.P. ; Girczyc E.F. ; HAL: A Multi-Paradigm Approach to Automated Data Path Synthesis ; Proc. of 23rd Design Automation Conference ; 1986.

[91] Paulin P.G., Knight J.P., 1989, Force-Directed Scheduling for the Behavioural Synthesis of ASIC's, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE-Press.

[92] Perry D.-L. ; VHDL Second Edition ; McGraw Hill Series on Computer Engineering 1993.

[93] Rankin J.R. ; Computer Graphics Software Construction; Prentice Hall; 1989.

[94] Rechenberg, I ; Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution ; in [6].

[95] Reichl, H.-D.: Untersuchung von Modellierung- und Syntheseverfahren assoziativer Architekturen mit Hilfe von VHDL, Diplomarbeit Technische Informatik II, Fern Universität Hagen, 1994.

[96] Schwefel, H.-P. ; Numerical Optimization of Computer Models; Chichester; John Wiley & Sons 1981.

[97] Shahid A. ; Sadiq M. S. ; Benten M. S.T. ; GSA: Scheduling and Allocation using Genetic Algorithm ; EURO-DAC '94 Grenoble France 1994.

[98] Sharma A. ; Rajiv J. ; InSyn : Integrated Scheduling for DSP Applications ; 30th Design Automation Conference Dallas, Texas 1993.

[99] Sjoholm S. ; Lindh Lennart ; VHDL for Designers ; Prentice Hall Europe ; 1997.

[100] Stok L. ; Interconnect Optimisation during Data Path Allocation ; Proc. of 1st European Design Automation Conference ; Glasgow 1990.

[101] Stok L. ; Data path synthesis; Elsevier ; INTEGRATION, the VLSI journal 18 (1994).

[102] Suplick Jean; An analysis of Load Balancing Technologie; Comparing LSF with other Load Balancing Software Packages; Texas 1994.

[103] van Swaaji M. ; Franssen F. ; Catthor F. ; DeMan H. ; Modeling data flow and control flow for DSP system synthesis in : Bayoumi M. (Hrsg.) ; VLSI Design Methodologies for DSP Systems; Kluwer; 1993.

[104] Tavangarian D. ; VLSI-Entwurfsalgorithmen; Fernkurs 1721 der Fernuniversität Hagen; 1992.

[105] Tavangarian D.; Einsatz von VHDL für die Schaltkreissimulation; it+ti 6/93; pp. 16-24.

[106] Tavangarian D. ; Koch M. ; Klein M. ; Hipper G. ; Hochleistungs-Datenverarbeitung in Workstation-Clustern, it+ti 1/95.

[107] Tavangarian D. ; Hipper G. ; Parallele Datenverarbeitung in Workstation-Clustern: Entwicklungsstand und architektonische Konzepte; 1. G-IIA-Symposium Stand und Anwendung digitaler Kommunikationsnetze; Bochum 1996.

[108] Trickey H. ; Compiling Pascal Programs into Silicon ; PhD Thesis; Stanford University, Report No STAN-CS-85-1059, 1985.

[109] Tseng C.-J. ; Wei R.-S. ; Rothweiler S. ; Tong M. ; Bridge: a versatile Behavioural Synthesis System ; Proc. of 25th Design Automation Conference, 1988.

[110] Walker R.A. ; Thomas D.E. ; Behavioural Transformations for Algorithmic Level IC Design ; IEEE Trans. on CAD; Vol. 8, No. 10, 1989.

[111] Wegener I. ; Effiziente Algorithmen für grundlegende Funktionen; Teubner 1989.

[112] Wegener I. ; The Complexity of Boolean Functions ; Wiley-Teubner Series in Computer Science 1987.

[113] Wegener I. ; Schaltwerktheorie ; Vorlesungsskript SS1990 ; Universität Dortmund.

[114] Wegener I. ; Theoretische Informatik ; B.G. Teubner Stuttgart ; 1993.

[115] When N. ; Glesner M. ; Held M. Anovel scheduling/allocation approach for datapath synthesis based on genetic paradigms. IFIP Working Conference on Logic and Architecture Synthesis, Paris 1990.

[116] Yeung P. , Rees D. ; Resource Restricted Aggressive Scheduling ; Proc. European Conference on Design Automation (EDAC) , 1992.