Architectural Synthesis and Optimization of Digital Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof. dr. J.H. van Lint, voor een commissie aangewezen door het College van Deken en het openbaar te verdedigen op woensdag 10 juli 1991 om 16.00 uur.

door

Leon Stok

geboren te Veldhoven
Dit proefschrift is goedgekeurd door de promotoren:
Prof. dr.-ing. J.A.G. Jess
en
Prof. dr. H. De Man

Copromotor: dr. ir. J.T.J. van Eijndhoven

© Copyright 1991 Leon Stok
All rights reserved. No part of this publication may be reproduced, stored in a retrieval
system, or transmitted, in any form or by any means, electronic, mechanical, photo-
copying, recording, or otherwise, without the prior written permission from the copyright
owner.
Druk: Wibro Dissertatiedrukkerij, Helmond

CIP-GEGEVENS KONINKLIJE BIBLIOTHEEK, DEN HAAG
Stok, Leon
Architectural Synthesis and Optimization of Digital Systems / Leon Stok. – [s.l. : s.n.]
ISBN 90-6003966-X
SISO 663.43 UDC 621.382 049.771.14:581.3:06(043.3) NUGI 832
Trefw.: digitale schakelingen; computer aided design.
Contents

Contents ........................................ 5
Summary ........................................ 9
Samenvatting ................................... 11
Preface ......................................... 15

1 Architectural Synthesis ......................... 17
   1.1. Historical Overview ..................... 18
   1.2. EASY Philosophy ....................... 20
   1.3. EASY System Description ............... 21
   1.4. Demonstrator Design .................... 24

2 System Representation .......................... 31
   2.1. Data Flow Graph ....................... 32
   2.2. Data Flow Graph Semantics .......... 33
   2.3. Representation of a Behavior Description Language in the DFG ...... 36
      2.3.1. Expression ......................... 36
      2.3.2. Conditional Statements .......... 37
      2.3.3. Repetition ......................... 38
      2.3.4. Procedures ......................... 40
      2.3.5. Arrays ................................ 41
      2.3.6. Input and Output .................. 42
   2.4. Queue Size ................................ 42
   2.5. State Transition Graph ............... 43
   2.6. Data Path Graph ....................... 43
   2.7. An Example ............................. 44
   2.8. Data Flow Graph Optimization ........ 46
   2.9. Memory Access Optimization .......... 47
   2.10. Mutual Exclusion ..................... 52
   2.11. System Representation Overview ..... 58

3 Scheduling ................................... 63
   3.1. The Scheduling Problem ............... 63
3.2. The Scheduling Problem in Architectural Synthesis .............................................. 64
3.3. Scheduling Schemes ......................................................................................... 69
3.4. Overview of Scheduling Algorithms .................................................................... 70
3.4.1. Transformational Scheduling ........................................................................ 70
3.4.2. ILP-Scheduling ......................................................................................... 71
3.4.3. Neural Network Scheduling ......................................................................... 71
3.4.4. Iterative Scheduling .................................................................................. 71
3.4.5. Force Directed Scheduling ........................................................................ 74
3.6. Extensions to Force Directed Scheduling ......................................................... 76
3.7. Creation of Storage Values ............................................................................. 79
3.8. Review of Scheduling Techniques ...................................................................... 81

4 Data Path Allocation .............................................................................................. 83
4.1. Definitions ....................................................................................................... 83
4.2. Overview of Allocation Schemes ....................................................................... 85
4.2.1. Rule Based Schemes ................................................................................ 86
4.2.2. Greedy / Iterative Schemes ...................................................................... 87
4.2.3. Branch & Bound Schemes ........................................................................ 88
4.2.4. Linear programming .................................................................................. 88
4.2.5. Allocation Schemes based on Logic Synthesis ............................................ 89
4.3. Global Allocation Schemes ............................................................................. 89
4.3.1. Tseng's Formulation ................................................................................ 89
4.3.2. Left Edge Algorithm ................................................................................ 91
4.3.3. The early EASY Allocation Scheme ......................................................... 91
4.3.4. The CADDY Allocation Scheme ............................................................... 93
4.3.5. The HAL Allocation Scheme ................................................................... 93
4.3.6. An Allocation Scheme based on Bipartite Matching .................................. 94
4.3.7. Allocation of Operation Sequences ........................................................... 94
4.4. Discussion of the Allocation Schemes ............................................................... 95

5 Register Allocation in Cyclic Data Flow Graphs ................................................... 97
5.1. Cyclic Register Allocation ............................................................................... 97
5.1.1. Arc Splitting ............................................................................................ 99
5.1.2. Initial Coloring ......................................................................................... 99
5.1.3. Boundary Condition ............................................................................... 100
5.1.4. Flow Network Construction .................................................................... 101
5.1.5. Arc Rerouting ......................................................................................... 102
5.2. The Multicommodity Flow Problem ............................................................... 103
5.3. Improvement to the Basic Algorithm .............................................................. 104
5.4. Experiments and Results .............................................................................. 104
5.5. Conclusions .................................................................................................... 106

6 An Exact Algorithm for Module Allocation ......................................................... 107
6.1. Definitions and Notations .............................................................................. 108
6.2. Module Allocation Graphs ............................................................................ 109
6.2.1. Module Allocation Graphs and Libraries ................................................. 109
6.2.2. Module Allocation Graphs and Mutual Exclusion .................................... 111
6.3. Weighted Module Allocation ...................................................................... 113
6.4. Conclusions ................................................................................................... 115

7 Interconnect Minimization during Data Path Synthesis ........................................ 117
7.1. Storage Value Grouping ............................................................................... 118
7.1.1. General Edge Coloring Algorithm ............................................................ 120
7.2. Bipartite Edge Coloring ............................................................................... 122
7.3. Multiple Read Values .................................................................................... 125
Summary

High level synthesis means going from an functional specification of a digital system at an algorithmic level to a register transfer level structure. Different applications will ask for different design styles. Despite this diversity in design styles many tasks in the synthesis will be similar. There is no need to write a new synthesis system for each design style.

The best way to go seems a decomposition of the high level synthesis problems in several well defined subproblems. How the problem is decomposed depends heavily on a) the type of network architecture chosen, b) the constraints applied to the design and c) on the functional description itself. From this architecture style, the constraints and the functional description a synthesis scheme can be derived. Once this scheme is fixed, algorithms can be chosen and fit into this scheme and solve the subproblems in a fast and, when possible, optimal way.

To support such a synthesis philosophy, a framework is needed in which all design information can be stored in a unique way during the various phases of the design process. This asks for a design data base capable of handling all design information with a formally defined interface to all design tools. This thesis gives a formal way to describe both the functional representation, the register transfer level structure and the controller and the relations between all three of them. Special attention has been paid to the efficient representation of mutual exclusive operations and array accesses. The scheduling and allocation problems are defined as mappings between these formal representations. Both the existing synthesis algorithms and the new algorithms described in this thesis fit into this framework.

Three new allocation algorithms are presented in this thesis: an algorithm for optimal register allocation in cyclic data flow graphs, an exact polynomial algorithm to do the module allocation and a new scheme to minimize the number of interconnections during all stages of the data path allocation.

Cyclic data flow graphs result from high level behavioral descriptions that contain loops. Algorithms for register allocation in high level synthesis published up
till now, only considered loop free data flow graphs. When these algorithms are applied to data flow graphs with loops, unnecessary register transfer operations are introduced. A new algorithm is presented that performs a minimal register allocation and eliminates all superfluous register transfer operations. The problem is reformulated as a multicommodity network flow problem for which very efficient solutions exist. Experiments on a benchmark set have shown that in all test cases all register transfers could be eliminated at no increase in register cost.

Only heuristic algorithms appeared in literature to solve the module allocation problem. The module allocation problem is usually defined as a clique cover problem on a so-called module allocation graph. It is shown that, under certain conditions, the module allocation graph belongs to the special class of comparability graphs. A polynomial time algorithm can optimally find a clique cover of such a graph. Even when interconnect weights are taken into account, this can be solved exactly. This problem can be transformed into a maximal cost network flow problem, which can be solved exactly in polynomial time. An algorithm is described which solves the module allocation problem with interconnect weights exactly, with a complexity $O(kn^2)$, where $n$ is the number of operations and $k$ the number of resulting modules.

In previous research, interconnection was optimized when the module allocation for the operations and the register allocation for the variables already had been done. However, the amount of multiplexing and interconnect are crucial factors to both the delay and the area of a circuit. A new scheme is presented to minimize the number of interconnections during the data path allocation. This scheme first groups all values based on their read and write times. Values belonging to the same group can share a register file. This minimizes the number of data transfers with different sources and destinations. Secondly, registers are allocated for each group separately. Finally the interconnect allocation is done. During the interconnect allocation, the module allocation is determined. The value grouping is based on edge coloring algorithms providing a sharp upper bound on the number of colors needed. Two techniques: splitting read and write phases of values and introducing serial (re-)write operations for the same value, make that even more efficient exact edge coloring algorithms can be used. It is shown that when variables are grouped into register files and operations are assigned to modules during the interconnection minimization, significant savings (20%) can be obtained in the number of local interconnections and the amount of global interconnect, at the expense of only slightly more register area.
Samenvatting

Hoog niveau synthese is het vertalen van een functionele specificatie van een digitaal systeem op een algoritmeisch niveau naar een netwerk op register transfer niveau. Verschillende applicaties vereisen een verschillende ontwerpstijl. Ondanks deze verschillendheid in ontwerpstijlen zullen veel taken in een synthese proces gelijksoortig zijn. Het is niet nodig om voor iedere ontwerpstijl een compleet nieuw synthese systeem te bouwen.

De beste te volgen weg lijkt het decomponeren van de hoog niveau synthese problematiek in een aantal goed gedefinieerde doelproblemen. Hoe deze opdeling geschiedt hangt enerzijds af van het gekozen type netwerk architectuur en de eisen die aan het ontwerp gesteld worden en anderzijds van de functionele beschrijving. Als deze drie zijn vastgelegd kan een synthese schema worden vastgesteld. Snelle en optimale algoritmes die in dit schema passen kunnen dan worden gekozen.

Om een dergelijke synthese filosofie te ondersteunen is een geraamte nodig waarin alle ontwerpinformatie op een unieke manier kan worden opgeslagen gedurende alle fasen van het ontwerpproces. Dit vraagt om een data base die in staat is om alle ontwerp informatie op te slaan met een formele interface naar alle ontwerp gereedschappen. Dit proefschrift geeft een formele beschrijving van zowel de functionele specificatie, het netwerk op register transfer niveau en het bijbehorende controle circuit, als de relaties tussen deze drie. Speciale aandacht wordt besteed aan het efficiënt representeren van wederzijds exclusieve operaties en de toegang tot informatie die in arrays is opgeslagen. De tijdsplanning - en toewijzing problematiek zijn gedefinieerd als afbeeldingen tussen deze formele beschrijvingen. Door te laten zien hoe verschillende bestaande algoritmes in dit geraamte passen wordt de geschiktheid van deze benadering aangetoond.

Drie nieuwe toevoegingen algoritmes worden in dit proefschrift gepresenteerd: een algoritme voor de optimale toevoeging van registers in cyclische data stroom grafen, een exact polynomials algoritme om de toevoeging van modules te doen en
een nieuw toewijzingschema om het aantal verbindingen te minimaliseren gedurende alle stappen van de data pad synthese.

Cyclische data stroom grafen ontstaan als de functionele beschrijving tussen bevat. Register toewijzings algoritmes die tot dusver in de hoog niveau synthese gebruikt werden hielden alleen rekening met data stroom grafen die niet cyclisch zijn. Wanneer deze algoritmes worden toegepast op cyclische data stroom grafen worden overbodige verplaatsingen van data tussen de registers gegenereerd. Het nieuwe algoritme doet een minimale toewijzing van de waarden aan registers en verwijdert alle overbodige data verplaatsingen. Het probleem is geformuleerd als het zoeken naar een transportweg voor meerdere producten in hetzelfde netwerk. Voor dit probleem bestaan zeer efficiënte oplossingen. Experimenten op een verzameling van referentie voorbeelden hebben aangetoond, dat in al deze test gevallen de overbodige data verplaatsingen konden worden verwijderd, zonder dat meer registers nodig waren.

In het verleden zijn een aantal heuristische algoritmes gepubliceerd om de toewijzing van modules te doen. Gewoonlijk wordt het module toewijzings probleem geformuleerd als het zoeken naar een bedekking met maximaal verbonden deelgrafen van een zogenaamde module toewijzingsgraaf. In veel gevallen behoort de module toewijzingsgraaf tot de speciale klasse van comparabiliteitsgrafen. Een polynomaal algoritme kan een optimale bedekking van een dergelijke graaf vinden. Als ook gewichten voor de verbindingen in beschouwing worden genomen kan dit probleem worden getransformeerd naar een maximale kosten netwerk transport probleem. Dit probleem kan exact opgelost worden in polynomaal tijd. Hieruit kunnen we concluderen dat het module toewijzingsprobleem met gewichten voor de verbindingen exact opgelost kan worden met behulp van een algoritme met complexiteit $O(kn^2)$, waarin $n$ staat voor het aantal operaties en $k$ voor het aantal resulterende modules.

Tot nu toe werd het aantal verbindingen pas geoptimaliseerd als de toewijzing van de operaties aan de modules en de waarden aan de registers was gedaan. Echter, zowel het aantal multiplexers als de hoeveelheid verbindingen zijn beide cruciale factoren voor de vertraging en de oppervlakte van het circuit. Daarom is een nieuw toewijzings schema nodig dat het aantal verbindingen minimaliseert gedurende alle fases van de data pad synthese. Dit schema groepeert eerst alle waarden gebaseerd op de tijden waarop ze worden gelezen en geschreven. Waarden die in dezelfde groep terechtkomen kunnen een register bank delen. Deze groepering minimaliseert dus het aantal data verplaatsingen met verschillende begin- en eindpunten. Vervolgens worden, voor iedere groep apart, de waarden aan registers toegewezen. Ten slotte worden de verbindingen vastgelegd. Gedurende het vastleggen van de verbindingen wordt de toewijzing van de modules bepaald. Het groeperen van de waarden is gedaan met behulp van een algoritme voer een kleuring van de takken van een graaf. Deze algoritmes maken gebruik van de scherpe boven-
Samenvatting

grens die bestaat voor het aantal benodigde kleuren. Twee technieken: het splitsen van de lees- en schrijffases en de introductie van seriële herschrijf operaties zorgen ervoor dat nog efficiëntere exacte takkleringssalgoritmes kunnen worden gebruikt. Er wordt aangetoond dat wanneer een dergelijk schema wordt gebruikt, belangrijke besparingen (20%) in het aantal lokale verbindingen en de hoeveelheid globale verbindingen verkregen kunnen worden, ten koste van een kleine toename in het aantal registers.
This thesis is the result of the work I did on high level and architectural synthesis. The work started when I joined the Design Automation Section of the Faculty of Electrical Engineering at the Eindhoven University of Technology in August 1986.

The book consists of eight chapters. Chapter one gives an introduction to the architectural synthesis problem. The architectural synthesis problem can be broken down in several subproblems: representation, scheduling and allocation. Chapter two deals with the representation of the system under design. This chapter presents a formal way to represent a system. Chapter three discusses the scheduling problem and several solutions to this problem. Chapter four to seven describe various strategies and algorithms for the data path allocation.

In chapter four the data path allocation is subdivided in several subproblems. The chapter describes various approaches which have appeared in literature to tackle this problem. Chapter five and six give two new algorithms which fit in this traditional allocation scheme.
When the functional specification contains loops, this can result in cyclic lifetimes for some of the values, i.e. the values are produced in one iteration and needed in a subsequent iteration. Chapter five presents a new algorithm for register allocation which can deal with this sort of cyclic constraints.

In chapter six a set of conditions are derived under which module allocation can be performed optimally in polynomial time. An optimal algorithm for the weighted module allocation problem is presented. This weighted approach takes into account costs which can be based on interconnect considerations.

Chapter seven describes a new strategy to solve the allocation problem. Since in many applications the interconnection constraints are the most important, the order in which the allocation problems were traditionally solved is changed. A new scheme and algorithms, minimizing the amount of interconnect in the data path, are discussed in chapter seven.

Chapter eight shows how the presented algorithms can be fit together to build a synthesis system. This book ends with a discussion of several problems that need more research.

This book is completed with a list of notations, a reference to the places where they are defined and an index. I hope these will improve the usefulness of this book.

Finally, I would like to mention those who contributed to this work. I am grateful to Jochen Jess for the opportunity to work in his research group. He initiated the research presented in this book and encouraged the exploration of new areas by his critical questions. He stimulated a precise formulation of the problems and their solutions when proofreading the various papers and articles that are written during this project.

I would like to thank Jos van Eijndhoven and Gjalt de Jong for their discussions on data flow graphs and Wim Philipsen, who was a soundboard for my ideas on colors and graphs. I would also like to mention the students who took part in developing parts of the system: Willem Crucet, Marcel de Leeuw, Wigert Vollebergh and Frans Vos. Special thanks to Adwin Timmer and Marc Heijligers for proofreading (parts of) the manuscript.

I am especially indebted to Reiner van den Born, who was there when he was needed most: at the beginning of the project, when he introduced me in the field of high level synthesis and data flow analysis, and at the end when he did very carefully proofread the manuscript for this book.

Finally, I would like to thank all other members of the Design Automation Section of the Eindhoven University of Technology, without which stimulating environment none of the results would have been possible.

Leon Stok
Chapter 1

Architectural Synthesis

Recent studies have shown that the design time becomes more and more a bottleneck in the whole process of producing an integrated circuit. In the last decade the development of the layout tools [Lengauer90] increased the productivity of the designer. Initially, the use of layout editors, later the usage of placement and routing tools and nowadays the wide application of module generators increased the productivity with a factor of 10. This increase is not enough to keep up with the increasing complexity of the integrated VLSI designs. Gordon Moore gave the following relation for the number of transistors on a chip in a given year.

\[ \text{transistors per chip} = 2^{(52 + 1956)} x 2/5 \]

In 1990 this results in roughly 6.5 million transistors per chip.

To keep the design time within acceptable limits the logic design tools [Bryanton84] became more widely used in recent years. The logic design tools enable the designer to focus on the functions which specify the design on an abstract boolean level. A sequence of logic synthesis programs optimises and transforms such a specification into a description which can be handled by the aforementioned layout generation tools.

The variety of integrated circuits also asks for many different designs to be made. While a decade ago only a few different IC’s were on the market, it is expected that the ASIC (Application Specific IC) market will consist of 20% of the IC-market in 1994 [Bosch80]. Due to the still decreasing life time of products these numbers will grow explosively in the next decade. There are two mechanisms that cause this. On one side, the electronics give unknown possibilities for product innovation. On the other side, more electronics will be needed per application.

The still increasing complexity and versatility of the integrated circuits ask for an even higher level of specification: the functional level. The functional specification is a formal description at an early stage of the design process. A simulation of such a specification allows the designer to evaluate the intended behavior of the de-
sign. The results of these simulations can be a reference during all subsequent synthesis steps.

The higher level specification becomes really useful if tools are available to synthesize circuits from such a specification. The realization of a register transfer level description from the functional specification is called the high level or architectural synthesis. The use of high level synthesis tools can result in a shorter design cycle. It enables the system designer to explore the design space rapidly and will result in designs that contain fewer errors. The formal functional description forces the designer to document his design very precisely. Together with the records kept by the automated tools, the result will be well documented. Since less knowledge of circuit and layout design will be needed, the IC technology will come within reach for more designers. Ultimately the synthesis system should offer the designer the same advantages as do software compilers to the programmers.

Figure 1.1 Steps in synthesis of VLSI systems
1.1 Historical Overview

Combining the architectural synthesis with the above mentioned logic and layout synthesis tools can result in a powerful system to aid the designer in the VLSI design process. The different levels of documentation and the phases of the synthesis forming the transitions between the levels are shown in figure 1.1.

The high level synthesis means going from a functional specification at the algorithmic level of a digital system to a register transfer level structure. The logic synthesis optimizes the logic functions of several register transfer modules and maps them into a given technology. Some other modules will be directly available from a module library or generated by special module generators. The layout synthesis produces a complete description of a mask set for the production of the circuit in a given technology. The layout synthesis phase consists of a sequence of placement and router steps.

This book will focus on the high level synthesis. A framework for a high level synthesis system is described and various synthesis schemes and algorithms, which can be applied under different conditions, are explained.

1.1. Historical Overview

One of the first high level synthesis systems was the CMUDA system developed by researchers at Carnegie-Mellon University in the late seventies. Since then many research groups started exploring the field of the high level synthesis. Apart from the work done at Eindhoven University, intensive research in high level synthesis is going on at the following European universities and research institutes: the Cathedral Projects at IMEC [DeMan90], the CADDY project at Karlsruhe University [Kramer88] and the MIMOLA project at the University of Kiel [Marwedel88].

In the United States the following projects are going on in the academic world, the System Architect's Workbench at CMU [Thomas89], the HSS system at the University of California at Irvine [Lisk88], the ADAM project at the University of Southern California [Park88] and the HYPER system at the University of California at Berkeley [Chua89]. In Canada research is going on with the HAL system [Paulin86].

Also the high level synthesis is gaining some interest in the industrial environment. In Europe we find the Pyramid, PHIDEO systems at Philips Research Laboratories [Huiskens88][Lippen89], the CALLAS system at Siemens [Wecker89]. In the United States the industrial high level synthesis systems under development are the YSC and V-compiler at the IBM T.J. Watson Research Center [Brayton88][Bersits89], the various systems (Bridge, SAM) under development at AT&T Bell Laboratories [Tseng88][Woo90a], and the Parsifal system at General Electric [Casas89]. In Japan the work done at NEC [Wakaba88] and NTT [Takaoka88] illustrates the industrial interest in high level synthesis.
1.2. EASY Philosophy

Different applications use different design styles. Despite this diversity in design styles, many tasks in the synthesis will be similar. It is not needed to write a new synthesis system for each design style.

On the other hand, programs that solve the whole high level synthesis problem at once do not seem the appropriate strategy either. In practical designs the architectural synthesis problem is too large to be solved in a single step. The algorithms in these approaches are usually too general (Devadas89). It is very difficult to guide the algorithms through the cost functions and the results are very unpredictable. Finding a real optimum even in small designs is already very time consuming.

The best way to go seems a decomposition of the architectural synthesis problem in several well defined subproblems. How the problem is decomposed depends heavily on a) the type of network architecture chosen, b) the constraints applied to the design and c) on the functional description itself. In a normal design flow the designer starts by writing the functional specification. He chooses a type of architecture and the constraints that have to be applied to the system. From this architecture style, the constraints and the functional description a synthesis scheme can be derived. The synthesis scheme describes which design decisions should be taken in which order. A scheme describes which information has to be available at the beginning of a synthesis step and which information is added by this specific step. Once this scheme is fixed, algorithms can be chosen which fit into this scheme and solve the subproblems in a fast and optimal way.

The decomposition will lead to mutual dependent subproblems. Solving these subproblems optimally does not guarantee an optimal solution to the global problem. However, when an appropriate order is chosen in which the problems are solved, the results can be very good.

In general, two things have to be considered when a scheme is determined for a certain application. Each step in the synthesis scheme not only restricts the following steps, but also provides more information to the following steps. For example, when certain operations are defined to be executed on specific modules as a first synthesis step, this can prevent the scheduler from finding an optimal schedule. On the other hand since detailed timing information for these operations is available, a much more precise schedule can be made.

Answers to questions which can be derived from the constraints and the type of application are very important during the problem decomposition. Answers to questions like:

- Which constraints in terms of area, timing and power dissipation are the most important?
- Which are the expensive modules in the network?
- Do the modules have a large difference in cost and/or execution time?
- Are registers expensive compared to the other modules?
1.3 EASY System Description

- Will this be a data transfer intensive or computational intensive design or both?
- Are interconnections expensive?
- Is this a control or data dominated design?

are very helpful in setting up a synthesis scheme. Most of the answers can simply be found by inspecting the data flow graph derived from the functional description, the library, the data path architecture and the constraints carefully.

To support such a synthesis philosophy, a framework is needed in which all design information can be stored during the various phases of the design process. This asks for a design data base capable of handling all design information with a formally defined interface to all design tools. In the EASY system a piece of design data is accessed in a unique way during all phases of the synthesis.

Experimenting with different functional descriptions and architectures and applying different constraints can lead to schemes which produce good results for a wide range of applications.

1.3. EASY System Description

An overview of the EASY system is depicted in figure 1.2. The input of the system is a functional description in a high level language. Many software languages have been adopted for high level functional specifications. Examples are: ADA [Circzy84], OCCAM [Monto85], C as HardwareC [DaMich88], Fortran [Tanaka89] and Prolog [Suzuki88]. Also numerous languages have been specially designed to write functional specifications like ISPS [Barbac81], SLIDE [Parker81], Conlan [Pilaty85], ZEUS [German85], DSL [Campos89], MIMOLA [Marwed86] and Silage [Hilfin86] and more recently ELLA [Morrison87] and VHDL [VHDL87].

Since language issues were not of primary concern in this research project, a powerful subset of Pascal [Jensen78] was chosen to enter our functional descriptions. Up till now we did not encounter any problems with this subset, when describing the various high level synthesis benchmarks. The functional specification is parsed and a data flow graph is build by the DFG-constructor. This constructor performs a full data flow analysis of the functional specification. The data flow graph is stored in the design data base. All subsequent tools will work from this single data base.

The DFG-optimizer transforms a data flow graph in an equivalent data flow graph. Equivalent means that both data flow graphs describe the same behavior with respect to the inputs and outputs. One sort of transformations eliminates superfluous constructs used by the designer. Another category transforms to data flow graphs that can be processed better by the subsequent tools.

The main synthesis is done by the schedulers and allocators. They will be discussed in more detail in the following chapters. The designer can choose from different libraries by loading the appropriate library into the design data base.
Figure 1.2 The EASY system

The extractor is an interface to extract the network and controller data from the database. Access functions are defined for all sorts of network related data. They can be combined with print functions to generate networks and controllers in the various standards like EDIF [EDIF88].

Much attention has been paid to provide the designer with information about the status of the design. Since a lot of data is involved in the sequence of design steps, a graphical representation of the data is indispensable. The feedback tools can be used to directly evaluate the results of the schedulers and allocators. The statistics can be kept over a series of design iterations and displayed to find the most optimal configuration. Examples of the feedback data will be shown in the next section.

Certainly a very important feedback aspect in the EASY system is the ability to display graphically the generated network and controller. The automatically generated schematics enable the designer to display the results produced by the synthesis system. The schematics are also very useful during the development of the various synthesis programs. A series of placement and routing programs have been devel-
oped during this project [Koster89]. The initial placement program (Poblo) [Stok89a] placed the network modules based on their connectivity. The functional partitioning part of the program could be guided by providing information on modules that belong to each other from a functional point of view. Later a placement special for logic circuits, circuits with only a few feedback loops, (Logpoblo) and a special placer for bus dominated data paths (Buspoblo) were added [Stok91c]. If no graphical information is provided for a certain module, this module can be generated on the fly by the module generator Quirilo. The result of all these placers can be routed by a new type of line router (Eureko) which was specially designed to produce a good looking routing instead of a minimal one. The resulting nets are easy traceable since the number of crossings and bends in the nets is minimized. The network pictures shown throughout this book are generated with the programs described above.

Another program that produces feedback is the data flow graph drawing program. A special placement program which has knowledge of the special structure of the data flow graphs places all nodes. Dummy nodes are added to reserve room for the edges. During the node placement it is tried to minimize the number of crossing edges by placing the dummy nodes on appropriate places while maintaining the data flow graph structure. Thanks to the dummy nodes the edges can be added as simple straight line segments. The hierarchy of the graphs is maintained. Various attributes of the nodes and edges can be displayed in the pictures. The drawing of the graphs proved to be very useful in the development of the other tools. The designer will be less interested in the graph, he mainly deals with the design on the level of the functional description. However, drawing a graph can be very useful when evaluating the results of the data flow graph transformations.

---

Procedure syndrome_generator;

const beta = 0.12, gamma = 0.23;
var x1, x2, x3, sym1, sym2, sym3: data_type;
import p1, p2;
output p3, p4, p5;

get (p1, x1);
get (p2, x2);
get (p1, x3);
sym1 = x1 + x2 + x3;
sym2 = beta * x1 + beta * x2 + beta * x3;
sym3 = gamma * x1 + gamma * x2 + gamma * x3;
put (p3, sym1);
put (p4, sym2);
put (p5, sym3);

Figure 1.3 Functional Description for the Syndrome Generator
1.4. Demonstrator Design

In this section a typical design iteration with the EASY synthesis system will be described. As a test vehicle, a design will be made for a syndrome generator [Depuydt80].

The syndrome generator, as described in figure 1.3, reads three values from the input ports, does some data processing and writes three values to the output ports. The data flow graph constructor generates the data flow graph displayed in figure 1.4.

![Data Flow Graph for the Syndrome Generator](image)

Figure 1.4 Data Flow Graph for the Syndrome Generator

The graph consists of three get nodes for the input operations and three put nodes for the outputs. Furthermore, the graph contains nodes for the 6 multiplications and 6 additions and constants.

Data flow graph optimization will result in the data flow graph displayed in figure 1.5. The common subexpression elimination will remove four additions and multiplications. The optimized data flow graph would have been generated immediately if the initial specification had been written as:
1.4 Demonstrator Design

\[ \text{syn1} = x_1 + x_2 + x_3; \]
\[ \text{syn2} = \beta \cdot \text{syn1}; \]
\[ \text{syn3} = \gamma \cdot \text{syn1}; \]

In the remainder of this example we will continue with the unoptimized version of the data flow graph.

![Diagram](image)

**Figure 1.5 Optimized Data Flow Graph for the Syndrome Generator**

After the graph is constructed a schedule can be made. The designer specifies a cycle time of 58ns and allows at most 8 cycle steps. In the library chosen the additions can be executed in 60ns by a ripple adder and the multiplications in 100ns by an array multiplier. Input and output operations take each 20ns. Two input multiplexers take 6ns and four input multiplexers take 8ns. Since a cycle time of 58ns was specified the multiplications have to be executed in two cycle steps. All other operations take one cycle step. The time constrained scheduler produces a result which uses three multipliers and two adders. The feedback programs display the distribution functions to the designer. It can be seen that no better results are possible with
the given library and timing constraints. This schedule forms a starting point for the allocation phases.

![Graphs showing distribution functions for the Syndrome Generator](image)

**Figure 1.6 Distribution Functions for the Syndrome Generator**

The values that have to be stored are assigned to register files and when possible, they share a register. A module allocator assigns the operations to the operational modules. An interconnection structure is built for all data transfers that have to take place. The network that is stored in the design database can be extracted and a schematic can be generated by the placement and routing tools for schematics. The result is shown in figure 1.7.
Simultaneous with the data path allocation a controller is constructed which controls the function of all operational modules, the cycle steps in which the registers have to read new data, and the position of the multiplexers in each state. Since the functional specification contains no special constructs like procedure calls or loops, the controller can traverse the cycle steps in a straight line and has no inputs. One state is generated for each control step. The controller for the circuit of figure 1.7 is shown in figure 1.8. For each state the output signals and the next state are specified. Only the output signals for the operational modules are shown.
st Syndrome_0
(IN_PORT_1...C1->#(0))
st Syndrome_1
(ARRAY_MULT_5...C1->#(1) ARRAY_MULT_4...C1->#(1)
 RISE_ADD_2...C1->#(1) IN_PORT_0...C1->#(0))
tr Syndrome_2
(ARRAY_ADD_6...C1->#(1) ARRAY_MULT_5...C1->#(1)
 ARRAY_MULT_4...C1->#(1) ARRAY_MULT_6...C1->#(1))
tr Syndrome_3
(ARRAY_MULT_4...5C1->#(1) ARRAY_MULT_5...C1->#(1))
tr Syndrome_4
(ARRAY_ADD_5...C1->#(1) ARRAY_MULT_5...C1->#(1)
 ARRAY_MULT_4...C1->#(1) ARRAY_MULT_6...C1->#(1))
tr Syndrome_5
(ARRAY_ADD_6...C1->#(1) ARRAY_MULT_5...C1->#(1)
 ARRAY_MULT_6...C1->#(1))
tr Syndrome_6
(ARRAY_ADD_6...C1->#(1) RISE_ADD_2...C1->#(1))
tr Syndrome_7
(OUT_PORT_9...C1->#(1) OUT_PORT_10...C1->#(1)
 OUT_PORT_11...C1->#(1))
tr Syndrome_0

Figure 1.8 Controller for the Syndrome Generator

Assume that in the next phase the designer wants to see the influence of more tight or relaxed timing constraints on the design. The number of cycle steps is varied between 5 and 10. For all schedules a complete network is generated. During this design exploration all design statistics are kept. When the exploration is finished the important design data can be displayed.

Figure 1.9 shows the statistics that were kept for the syndrome generator. The number of multipliers, adders, registers, register files, multiplexers, multiplexer inputs, the number of connections and the area are plotted against the variation in the number of allowed cycle steps. The fastest implementation needs four multipliers and three adders. Relaxing the timing constraint with one cycle step decreases the number of multipliers and adders to two.

The number of registers seems insensitive to the schedule length. This has been noticed in many applications. If a fast schedule is made, much data is generated in each cycle step and a lot of registers will be needed. On the other side, if a slow schedule is made, a lot of data has to be kept for many cycles. This also requires a certain
number of registers. It seems that these effects counterbalance which makes the number of registers insensitive to the schedule length.

The number of register files increases with the number of cycle steps. Since less data transfers will take place simultaneously more of the registers can share a register file. The number of multiplexers and nets form a measure for complexity of the routing that may be expected. The main statistic is the area required by the design. The area is calculated by adding the area for the modules, registers and multiplexers and an estimate of the interconnect area.

![Graphs showing relationships between time, multipliers, adders, registers, register files, multiplexers, mux inputs, nets, and area](image)

**Figure 1.9** Design Space Exploration for the Syndrome Generator

This walk-through design example illustrated some of the design steps, that will be discussed in much more detail in the forthcoming chapters.
Chapter 2

System Representation

A uniform system representation plays a key role in a high level synthesis system. A good system representation should:

- support design partitioning in a well defined manner.
- provide a uniform basis for tool interfacing and user interaction.
- make incremental design and backtracking possible.
- allow the design effort to be partitioned in a well-structured way.
- minimize the redundancy of design data. This minimization makes it also much easier to keep the data consistent and to know in which place what information is stored.

In this chapter a single model is defined to represent a system in all phases during the high level synthesis. The model consists basically of two levels: the behavioral level and the register transfer level. At the behavioral level the system is represented as a Data Flow Graph (DFG). At the register transfer level the system is represented by two graphs: a Data Path Graph (DPG) and a State Transition Graph (STG). The data path graph is analogous to a schematic diagram: it describes the design in terms of modules and their interconnections. The state transition graph is used to model a finite state controller, that controls the various modules in the data path.

The various representations that have appeared in literature, are very similar with respect to their representation on the register transfer level. Our register transfer level representation is described in the sections 2.5 and 2.6. However, major differences exist in the behavioral level representation. The main feature of the DFG is that it not only expresses the maximal parallelism between operations in the presence of conditionals but also in the presence of loops and procedures. Therefore, a formal model of the DFG is introduced in section 2.2. In section 2.3 is shown how the constructs from most behavioral description languages can be represented in this DFG. Another advantage is that not only the parallelism based on data dependencies but also the parallelism in the access of memory is represented clearly.
Section 2.9 gives an algorithm which detects the possible concurrency between the memory accesses in the DFG.

Not only the exposure of the concurrency between the operations is important, but also the detection of operations that seem to be concurrent, but can never be executed simultaneously due to conditional data flow, has to be done. Two new notions of mutual exclusion are discussed in section 2.10.

To be able to give a classification of the known representations and to clearly show the advantages of the DFG, the discussion on the earlier representations is postponed till the final section. Criteria to classify system representations are given and a classification of most of the representations that have appeared in the literature is made.

2.1. Data Flow Graph

The data flow graph, as defined in this section, closely resembles the flow graphs used in compilers for data flow machines. Our DFG is very similar to the demand graph as defined in [Veen86]. In the DFG all control flow constructs in the behavioral description language are represented by data flow operators. During the data dependency analysis, the effects of the control constructs in the behavioral description are interpreted and expressed in special nodes. In this section, first a formal definition of the data flow graph is given. The definition of the semantics not only provides a uniform entry to the synthesis tools but also allows the DFG itself to be executed and to simulate its own behavior. A classification of the DFG nodes is made based on their semantics. After the description of the semantics of each of these classes, the representation of the various components of a behavioral description language in terms of the DFG is given. An algorithm how to construct the DFG from the input language will not be given here. A method that transforms a program into a data flow graph similar to the DFG is described by Veen in [Veen86]. The implementation of this method to create the DFG is discussed in [Stok86].

**Definition 2.1.** A data flow graph (DFG) is a graph \( G (V, E_{d}, E_{s}) \) where:

1. \( V = \{v_1, v_2, \ldots, v_n\} \) is a finite set whose elements are nodes, and
2. \( E_{d} \subseteq V \times V \) is an asymmetric data flow relation, whose elements are directed data and control flow edges, and
3. \( E_{s} \subseteq V \times V \) is an asymmetric sequencing relation, whose elements are directed sequence edges.

The directed data and control edges represent the flow of data and control signals between the nodes. The nodes represent operations that are carried out on the data and control signals that reach the node on its input edges. These operations can be arithmetic operations (like +, −, *, /) or logical operations (like and, or, xor, not) or operations that influence the flow of the data depending on a condition on one of
2.2 Data Flow Graph Semantics

The input edges. Sequence edges are used to describe a precedence relation between nodes when there is no direct data flow relation between them.

2.2. Data Flow Graph Semantics

The DFG nodes can be distinguished by their semantics into five classes: the source nodes, the operation nodes, the branch nodes, the merge nodes and the multiple output nodes. The following definitions are needed to give a description of the five classes based on their semantics. The description of the semantics is loosely based on the definition in [Kavi85].

The set of data input edges to a node $v$, and data output edges from a node $v$ are denoted by $I_d(v)$ and $O_d(v)$.

$$I_d(v) = \{ (x, v) \in E_d \}$$

$$O_d(v) = \{ (v, x) \in E_d \}$$

Similarly, the set of sequence input edges to a node $v$, and sequence output edges from a node $v$ are defined by:

$$I_s(v) = \{ (x, v) \in E_s \}$$

$$O_s(v) = \{ (v, x) \in E_s \}$$

The set of input edges to a node $v$, and output edges from a node $v$ can be defined as:

$$I(v) = I_d(v) \cup I_s(v)$$

$$O(v) = O_d(v) \cup O_s(v)$$

As far as the semantics of the data flow model are concerned, sequence edges are the same as the data flow edges. The motivation for special sequence edges is explained from a synthesis point of view, and is not in the data flow model, because here they have the same meaning as data flow edges. The data flow edges represent a transfer of real data and therefore also a precedence relation, because data can not be transferred before it is produced. The sequence edges only describe a precedence relation. No real data transfer is modelled. However, to fit these sequence edges into our model, these edges will contain symbolic data, which represents the precedences, during the execution of the model.

**Definition 2.2:** A marking is a mapping

$$M : E_d \rightarrow D$$

where $D$ is a set of queues of data values, which are denoted by $<d_1, d_2, \ldots, d_n>$. This queue may also be empty. The function $\cup (M(e))$ returns true if the queue $M(e)$ is empty. The function $\cap (M(e))$ returns true if the queue $M(e)$ is full. However, in the general model, queues of unbounded length will be assumed. This means that
in this case the function $\cap (M(e))$ will always return false. The function $\lor <d_1, d_2, \ldots, d_n>$ returns the first element of the queue i.e. $d_1$.

In the following paragraphs a model which defines the execution of the nodes will be described. Some nodes will execute if all their input edges have data values, some others if only a subset contains data. The queues on one or more of the output edges need to have some room available to store the newly generated values. To classify the nodes, based on this criterion, the concept is formalized by the introduction of the execution sets.

An edge $e$ is said to contain data in a marking $M$ if $M(e)$ not empty. An initial marking $M_0$ is a marking in which a specific subset of the edges contain data.

**Definition 2.3:** An execution of a node $v$ is a mapping from markings to markings.

A node is executable at a marking $M$ if the following conditions hold:

\[
\bigcup (M(e_i)) = \text{false} \quad \text{for all } e_i \in E_1(v, M)
\]

\[
\bigcap (M(e_i)) = \text{false} \quad \text{for all } e_i \in E_2(v, M)
\]

where $E_1(v, M) \subseteq I(v)$ and $E_2(v, M) \subseteq O(v)$ are the so called execution sets.

The nodes will be divided into semantical classes depending on the definition of the execution sets. When a node $v$ is executed, data is consumed from the edges which belong to $E_1(v, M)$ and data is placed on edges in the set $E_2(v, M)$. Thus, a new mapping $M'$ from the execution of a node $v$ at marking $M$ can be derived in the following way:

\[
M'(e_i) = \begin{cases} 
<d_1, \ldots, d_n> & \text{if } e_i \in E_1(v, M) \\
<d_1, \ldots, d_n, d> & \text{if } e_i \in E_2(v, M) \\
M(e_i) & \text{otherwise}
\end{cases}
\]

where $d$ is the result of applying the operation of the node to the data values on the edges in $E_1(v, M)$.

Execution rules can be defined depending on whether the execution sets $E_1(v, M)$ and $E_2(v, M)$ select one, a subset or the entire set of the input and output edges. In the DFG the following five classes of nodes can be distinguished based on the execution rules they follow:

1) **Source class**

A node belonging to the source class has no incoming edges. Such a node executes once at the beginning of the execution of the DFG and places a data value on each output edge. That is:

\[E_1(v, M) = \emptyset \quad \text{and} \quad E_2(v, M) = O(v)\]

2) **Operation class**

Operation nodes represent the arithmetic and logical operations. Such a node may execute if all incoming edges have data available on them. The execution of an oper-
2.3 Data Flow Graph Semantics

A station node will remove data from each incoming edge and put data on each outgoing edge. That is:

\[ E_1(v, M) = I(v) \text{ and } E_2(v, M) = O(v) \]

The value of the output data is calculated by applying the operation to the data on the head of the queues of all input edges, that is:

\[ d = op \left( + (M(e_1)), \ldots, + (M(e_n)) \right) \text{ with } e_1, \ldots, e_n \in E_1(v, M) \]

3) Branch class

A branch node has two incoming edges: a control edge and a data edge, and one or more outgoing edges. A branch node passes data from the data input edge to exactly one outgoing edge. Which outgoing edge, is determined by the value taken from the queue on the control edge. That is:

\[ E_1(v, M) = I(v) \text{ and } E_2(v, M) = \{ \}
\]

\[ E_3(v, M) = \{ o_i \} \text{ if } i = + (M(e_i)), 1 \leq i \leq m \]

\[ d = + (M(e_d)), \text{ where } e_d \text{ is the incoming data edge and } e_c \text{ is the control edge.} \]

Branch nodes are drawn as displayed in figure 2.1.

![Figure 2.1](image)

**Figure 2.1** Representation of a) a Branch Node  b) a Merge Node c) Groups of Branch and Merge nodes which have the same control signal may be drawn connected by the control signal to avoid clutter in the drawings.

4) Merge class

Nodes in the merge class are dual to branch nodes. They have only one outgoing data edge and a number of incoming data edges. A merge node also has an incoming control edge. Data is passed from one of the incoming edges to the outgoing edge. The value of the data on the control edge determines from which input edge the data is taken. That is:

\[ I(v) = \{ e_1, e_2, \ldots, e_n \} \]

\[ E_1(v, M) = \{ e_i \} \text{ if } i = + (M(e_i)), 1 \leq i \leq n \]

\[ E_2(v, M) = O(v) \text{ and } d = + (M(e_d)) \]

5) Multiple Output class

Nodes which belong to the multiple output class are almost similar to the nodes in the operation class. The difference lies in the fact that the multiple output class
nodes can have different data on the different output edges, while operation class nodes always have the same data on all output edges.

\[ I(v) = \{ e_1, e_2, \ldots, e_n \}, \quad O(v) = \{ e_1, e_2, \ldots, e_m \} \]

\[ E_1(v, M) = I(v) \quad \text{and} \quad E_2(v, M) = O(v) \]

\[ d_i = \varphi; ( \rightarrow (M(e_1)), \ldots, \rightarrow (M(e_n)) ), \quad 1 \leq i \leq m \]

### 2.3. Representation of a Behavior Description Language in the DFG.

Behavioral description languages usually contain the following basic constructs: expressions and assignment statements, conditionals, repetition and a procedure call mechanism. In this section we will describe how these basic language constructs can be modelled in the DFG using nodes with the semantics as defined in the previous section. Each node is given a type to distinguish between nodes with the same semantics but resulting from different language constructs. Each type belongs to one and only one of the semantical classes. A function \( r : V \rightarrow O \) is defined which gives the type of each node \( v \in V \).

The following drawing conventions will be used in subsequent figures (see for example figure 2.2): a node labelled \( D_i \) is a node which defines (assigns a value to) the variable \( x \). Similarly, a node labelled \( U_i \) is a node which uses (uses the value of) the variable \( x \). Solid arrows represent data flow edges, dashed arrows represent sequence edges. Sometimes edges with the same source and different destinations will share parts of the arrows to avoid clutter in the drawings, but they are separate edges in the DFG model.

#### 2.3.1. Expressions

All node types that are used to model arithmetic and logical expressions belong to the operations class. A distinction is made between monadic operation nodes, which have one input, dyadic operation nodes which have two inputs and polyadic nodes which have an arbitrary number of inputs. A constant in an expression is represented by a constant node. A constant node will always deliver the same constant value. A node of the type constant has one input. This input connects the constant node to a special node: the source node. The source node belongs to the source class. In the initial marking \( M_0 \) a value is placed on all outgoing edges of the source node.

Formally, constant nodes and monadic nodes have \( |E_i(v, M)| = 1 \), dyadic nodes have \( |E_i(v, M)| = 2 \) and the polyadic nodes have \( |E_i(v, M)| > 2 \). The nodes of each of these types can have an arbitrary number of outputs.
2.3.2. Conditional Statements

Pairs of branch and merge nodes are used to represent the conditional statements of the language. Case-statements can have an arbitrary number of branches. If-statements have two branches called the then-branch and the else-branch. In figure 2.3 a) a template for the representation of a conditional statement is given. Figure 2.3 b) contains an example for the following if-expression:

\[
\begin{align*}
&\text{if test} \\
&\quad \text{then then-branch} \quad \text{then } a = a + 1; \\
&\quad \text{else else-branch} \quad \text{else } a = a - 1; \\
&\quad \text{endif}
\end{align*}
\]

Note that in the figure the edges from the source node to the constant nodes are not drawn. The test part defines a boolean control signal c which is used as the control input for all branch and merge nodes (node Dc). For each variable that is used in the then- or else-branch a branch node is made. This branch node connects the definition of a variable (node Da) to its respective uses in the then- or else-branch (node Du). A merge node is made for each variable which is defined in at least one of the branches. The inputs of the merge nodes originate from the definitions in the then- and else-branch. If a variable is not defined in one of the branches, the merge node will be connected directly to the link node in that branch. In the template in figure 2.3 it is assumed that variable b is not defined in the else-branch and the merge node is directly connected to the branch node.

Branch nodes only have a single output for each branch in the conditional. A link node is connected to each of these outputs. This link node makes it possible that
Figure 2.3 a) Template for conditional statement.
   b) Data flow graph for if-example.

More edges can be connected to a branch node in a certain branch. Thus, a link node has one incoming edge and an arbitrary number of output edges. The link node belongs to the operation class. When a link node is executed the value from the input edge is copied to all output edges. A link node belongs to the operation class.

In the case of an if-statement, the control signal $D_1$ can have the value 0 or 1, which determines what branch is chosen. In case-statements, the conditional can have an arbitrary number of branches. The control signal has a value from the ordered set $\{c_0, c_1, \ldots, c_{n-1}\}$ where $n$ is the number of branches of the conditional and $\forall j \neq j', 0 \leq i, j \leq n-1, c_i \neq c_j$. If the control signal $D_1$ equals $c_i$, branch $i$ is chosen.

2.3.3. Repetition

The nodes from the branch and the merge class can also be used to represent the repetition statements. To distinguish between branch and merge nodes resulting from conditionals and repetition, they are called entry and exit nodes when they deal with repetition. The nodes of the entry type belong to the merge class, the nodes of the exit type belong to the branch class. In Figure 2.4 a) the template for a repetition is shown. An entry–exit pair is made for each variable that is used or defined in the
test or body of the loop. Data is entered in the loop through the entry node. The exit node sends the data into the body of the loop or out of the loop depending on the value of control signal $D_1$, as defined in the test.

![Diagram](image)

Figure 2.4 a) Template for repetition b) Data flow graph for while-example.

In figure 2.4 b) a part of the DFG for the following while-loop is given:

```plaintext
a := 0;
while TEST do
    while a < b do
        BODY
        a := a + 1
    endwhile
endwhile
```

For-loops can be modelled in the same way as while-loops. Repeat-until are modelled slightly different; all the nodes that belong to the body, are placed in between the entry and exit nodes, directly before the test.

The control input edges of the entry node have as their origin a node that is within the loop. But since the entry node will not execute as long as its control edge is undefined, none of the nodes within the test or body will execute. Therefore, the control edges of all entry nodes in the DFG are defined once in the initial marking $M_0$, to make sure that the entry nodes execute at least once.
2.3.4. Procedures

Procedures can be used to describe a behavior hierarchically, or to break down a description into several smaller pieces. For each procedure a separate DFG is made. These DFGs are surrounded by special nodes which define the connections to the calling graph. For all variables that are used in the procedure body and that are not defined locally first, a param node is created. The param node can be seen as the first definition of a variable in the procedure. If a behavioral description language is used with a scoping of variables similar to the Pascal language (Jensen78), param nodes are created for all value and reference parameters of the procedure and for the global variables that are used in the body of the procedure. Result nodes are added for variables that are defined in the procedure body and whose values have to be exported to the calling environment. In the Pascal like languages these are the reference parameters and the globals that are defined within the body.

---

![Flow graph for a procedure call](image)

**Figure 2.5 a) Flow graph for a procedure call. b) Flow graph for a procedure definition.**

In the DFG from where a procedure is called (see figure 2.5) a node with the name of the procedure is inserted. A call-in node is made for each param node and connected to the procedure node and the last definition of the parameter. For each procedure call a call relation is defined. This one-to-one mapping maps a call-in node to the corresponding param node:
2.3 Language Representation in DFG

\[
\text{Call} : V_{\text{call-in}} \rightarrow V_{\text{param}}, \quad \text{where} \quad V_{\text{call-in}} = \{ v \mid v \in V_1 \text{ and } r(v) = \text{call-in} \} \text{ and } \\
V_{\text{param}} = \{ v \mid v \in V_2 \text{ and } r(v) = \text{param} \}, \quad \text{and } V_1 \text{ are the nodes of the calling DFG and } \\
V_2 \text{ are the nodes of the called procedure.}
\]

The \textit{call} relation determines at which call\textsuperscript{-in} nodes values are imported in the procedure body, when the procedure is called.

Similarly \textit{call-out} nodes are made for all result nodes, and a \textit{return} relation is defined as well.

\[
\text{Return} : V_{\text{call-out}} \rightarrow V_{\text{result}}, \quad \text{where} \quad V_{\text{call-out}} = \{ v \mid v \in V_1 \text{ and } r(v) = \text{call-out} \} \text{ and } \\
V_{\text{result}} = \{ v \mid v \in V_2 \text{ and } r(v) = \text{result} \}
\]

The \textit{call-in} and \textit{call-out} nodes belong to the operation class. \textit{Call-in} nodes have a single input and a single output. \textit{Call-out} nodes have a single input and an arbitrary number of outputs. The operation of both node types is the same: copying the data on the input edge to all output edges upon execution of the node.

The \textit{procedure} node is a multiple output operation class node. It executes when all input edges are defined and starts the execution of the procedure body by placing the data in the \textit{param} nodes. When the procedure body is finished the data from the result nodes is placed on the corresponding output edges of the \textit{procedure} node.

The nodes with type \textit{param} belong to the input class. The \textit{result} nodes are a member of the output class. Functions are represented in the same way as procedures.

2.3.5. Arrays

The definition of an array is represented in the DFG by a node of type \textit{array} (see figure 2.6.a). This node belongs to the operation class. The incoming edges of this node come from constant nodes which are the initial values of the array. The outgoing edges represent the collection of all array values. An array access will appear in the data flow graph either as an \textit{update} or a \textit{retrieve} node. The retrieve node is used in the value context (when the value of an array element is used). The update node is used in the address context (when an array element is assigned a new value). The inputs to the \textit{update} node are the index and the new data value. A \textit{retrieve} node has an index input and a data output. The \textit{retrieve} and \textit{update} types belong both to the operation class.

The array accesses which can influence each other have to be left in order. Therefore, edges are added between all access nodes of the same array to assure that they are executed in the same sequence as in the initial description. In this way a chain of edges is formed for each array, starting at the \textit{array} node of the corresponding array. The order in which the accesses are written in a sequential specification does not have to be the optimal order in the implementation. Section 2.9 describes an algorithm that detects dependencies between the array accesses and rearranges the edges to reflect parallelism between the array accesses.
2.3.6. Input and Output

To exchange data with the environment, input and output ports are needed. These ports are accessed by the input and output operations. The put operation writes a value to a port, and a get operation reads a value from a port. It is important that the order in which the data is written to and read from one of the ports is maintained during the synthesis. Therefore, all put and get nodes that belong to the same port are connected through sequence edges. This ensures that all data is read and written to a port in the order as in the initial behavioral specification. For each port a chain of sequence edges is formed which connects all put and get operations that are done using this port. Figure 2.6.b) shows a chain of input–output operations. The get and put nodes belong both to the operation class.

2.4. Queue Size

In the general model an unbounded size for the queues on the edges is assumed. However, in an implementation of the DFG the queue sizes have to be limited. One of the first steps in the synthesis process is therefore, to determine the size of each of these queues. The simplest case is to assume that all queues have size not greater than one. A queue size of one determines that a node \( v \) can only be executed if the queues of edges in \( F_G(v, M) \) are empty (definition 2.3). A queue size of 1 also assures that all repetitions are executed completely synchronously. When only while-
2.6 Data Path Graph

loops are considered, the execution of all operations in one iteration of the repetition is completed before operations of the next iteration are executed. If queues of a size greater than 1 are allowed, operations in the various repetitions are not synchronized anymore. For example, the iteration variable can be calculated for several iterations and stored before other operations in the repetition body are executed. The determination of the queue sizes also influences how operations are shifted to other iterations of a loop. It can be shown that the various loop folding techniques, see for example [Goosse89], can be reduced to the problem of the determination of the queue sizes.

2.5. State Transition Graph

The control of synchronous digital hardware can be described by a set of finite state machines. A state machine will be described by a state transition graph.

Definition 2.4: The state transition graph (STG) is a graph \( G(, T, X, Y) \) where:

(i) a sequence \( S = <x_1, ..., x_m> \) of cycle steps

(ii) a set \( X = \{x_1, ..., x_m\} \) of input signals

(iii) a set \( Y = \{y_1, ..., y_l\} \) of output signals

(iv) a transition function \( \tau : S \times X \rightarrow S \), represented by the edges in \( T \)

(v) an output function \( g : S \times X \rightarrow Y \)

The finite state machine acts by reading a string of input symbols and writing a string of output symbols. If the machine is currently in cycle step \( s \in S \) and it is presented an input symbol \( x \in X \), then it will change its cycle step to \( \tau(s, x) \), and write the output symbol \( g(s, x) \).

Definition 2.5: If \( A \) is a set, then the power set \( \Pi(A) \) of \( A \) is the set of all subsets of \( A \). Thus: \( \Pi(A) = \{X | X \subseteq A\} \).

If \( A \) is an ordered set (sequence) the subsets \( X \) will be assumed to be ordered as well.

A scheduling function \( \sigma : V \rightarrow \Pi(S) \), where \( \Pi(S) \) is the power set of \( S \) as defined in definition 2.5, assigns to each DFG node a sequence of cycle steps in which the node is scheduled.

2.6. Data Path Graph

The data path graph describes a register transfer level design analogous to a schematic diagram. The design is described in terms of modules and their interconnections.

Definition 2.6: The data path graph (DPG) is a graph \( G(, M, I) \) where:

(i) a set \( M = \{m_1, ..., m_k\} \), whose elements are called modules, are the nodes of the graph. Modules can be operational modules like adders, multipli-
ers and ALUs or storage modules, like registers and register files or interconnection modules like multiplexers, demultiplexers and bus drivers.

(ii) an interconnection relation \( i \subseteq M \times M \), whose elements are busses and other interconnection links. These are the edges of the data path graph.

The set \( L = \{l_1, \ldots, l_j\} \) is defined as the set of library components. Each module \( m \in M \) specifies:

(i) the library component of which this module is an instance, and

(ii) the pins \( P = \{p_1, \ldots, p_l\} \) of the module.

For each interconnection it is defined which pins of the module it is connected to. The assignment function is a mapping from the data flow graph nodes to the library components. The assignment function determines the type of module an operation is mapped onto. The allocation function is a mapping from the data flow graph nodes to the modules in the network. The allocation function describes the mapping to an actual module instance.

### 2.7. An Example

A simple program, taken from [McFar88] which computes the square root of \( X \) using Newton’s method will be used as an illustration. The number of iterations is usually very small. In our example the number of iterations is four:

```plaintext
get(p1, x);
\n\t y := 0.22 + 0.89 * x;
\n\t i := 0;
\nwhile i < 4 do
\t\t y := 0.5 * ( y + x / y ) ;
\t\t i := i + 1;
\nendwhile
\nput(p2, y);
```

In figure 2.7 the DFG for this example is shown. The solid edges describe the dataflow relation. For each value that circulates in the loop, a separate entry-exit pair is made. The nodes in between the right pair execute the increment and test of the iteration variable. The middle right pair represent the update of the value carried by variable \( Y \). In the middle left pair the value of variable \( X \) circles around such that it can be used in each update of \( Y \).

Assume that the whole program is scheduled in three cycle steps, and that the nodes are distributed over the cycle steps as follows:

\[
\begin{align*}
\text{s}_0 & \quad \text{get } X \\
\text{s}_1 & \quad + \quad \leftarrow \text{put} \\
\text{s}_2 & \quad \times
\end{align*}
\]
In the graphical representation of the STG, the nodes represent the cycle steps and the edges the cycle step transitions. The scheduling function $\Theta$ is shown by the dotted lines. In this example, each node is scheduled in only one cycle step. For example, $\Theta(\text{get}) = s_0$. The loop in the STG is caused by the repetition in the DFG. The result of the $\texttt{<-}$ operation is an input variable of the STG.
The data path for this example consists of three operation modules, five multiplexers (shown by thick lines) and several registers to hold constant and data values. The allocation function is displayed by the dashed lines. The multiply / divide module performs three operations during the execution of the algorithm. This is possible because all three operations are assigned to different cycle steps as shown by the scheduling function. The outputs of the STG are not shown in the figure. They will be the signals that are needed to control the operations of the modules in the DPG. For example: the left most module has to be controlled to execute either a multiplication or division depending on the cycle step of the STG.

2.8. Data Flow Graph Optimization

When the behavioral description is written down several inefficiencies may occur in the description. Some are introduced because of facilities provided in the behavioral description language. Examples are the use of symbolic constants or the use of many procedures to structure the description. Other inefficiencies are introduced because the designer does not write the description carefully. Some of these inefficiencies can be removed by various DFG optimization algorithms. Many of these are similar to those optimizations used in software compilers (Kennedy81). The DFG has proved to be a very suitable representation to perform all kinds of optimizations. Only a subset of the optimizations will be discussed in this chapter.

Dead Code Elimination
Operations that do not in any way contribute to the output values can be removed from the DFG. From the definition of the DFG it follows that any node that is not a predecessor of a put node can be removed from the DFG. A simple marking algorithm, starting from the put nodes marks recursively all its predecessors. When the marking process is finished all unmarked nodes can be removed.

Constant propagation
Due to the use of (symbolic) constants, operations that only have constants as their operands can be present in the DFG. The results of these operations can be calculated at compile time, and the operation nodes can be replaced by the constant value. If the test of a conditional evaluates to a constant all branches except for one can be removed. Similarly, the body of a repetition has to be executed never, once or endlessly when the test reduces to a constant value. The constant propagation is also very useful to determine the value of indices in array accesses.

Common Subexpression Elimination
If two nodes that both compute the same expression \( A \circ B \), where \( \circ \) can be any operation, are separated by code which contains no definition of either \( A \) or \( B \), then the second node can be eliminated and the result of the first used instead.
2.9 Memory Access Optimization

Detection of Loop Invariant Computations
Operations that depend only upon variables whose values do not change in a loop, may be moved out of the loop. This improves the performance by reducing the operations' frequency of execution. The implementation of a combined common subexpression elimination and loop invariant code motion algorithm within the EASY environment, is discussed in [deLeeuw87].

Tree height reduction
The associativity and commutative properties of operations can be used to reduce the length of the critical path of an expression. When an expression is represented as a tree one speaks about tree height reduction. [Baer69] gives an algorithm that produces a minimum height tree, for trees containing multiplications, additions, subtractions, divisions, exponentiation and unary minus operations. The Baer (and Bovet) algorithm assumes unit delay operations. An extension to this algorithm is made in [Beatty72] which takes also operation delays into account. A new tree height reduction algorithm has been developed which works on the DFG. The algorithm [Heijlig89] breaks the expressions in the DFG up in trees, minimizes the height of each tree separately and recombines the trees. A special algorithm for the three height minimization in pipelined architectures is given in [Hartley89].

Procedure expansion, elimination
Two other manipulations on data flow graphs: procedure expansion and procedure elimination can be carried out very easily using this graph structure. Procedure expansion is performed by copying the procedure declaration and changing the name of the calling procedure node. Procedures that are called only once (or fully expanded procedures) can be eliminated by connecting the nodes as described in the call and return relation and removing the call-in, call-out, param, result and procedure nodes.

2.9. Memory Access Optimization

Arrays are used to model either memories or register files. Two types of arrays can be distinguished by the type of indexing used to access the array elements.

(a) In all array access operations, the value of all indices are constants or can be determined at compile time using constant propagation.

(b) There is at least one access of which the value of the index is data dependent and therefore can not be determined at compile time.

Arrays of type (a) can be implemented as a collection of single registers, as on chip register files or as memory. The decision how to implement these arrays will usually depend on the size of the array and number of values that have to be accessed simultaneously. Arrays of type (b) will always be mapped onto memory. When arrays of type (a) are implemented as a collection of single registers, they will be modelled in the DFG as a collection of separate variables. When the arrays are im-
implemented as memory, additional analysis can determine the potential parallelism in the array access operations.

Memory operations are expensive and detecting and eliminating unnecessary memory references may improve the performance and decrease the cost of the design. Doing a more detailed analysis of the effects of array access operations makes it possible to make their control dependencies explicit in the data flow graph. This has the following advantages:

- The length of the critical path of control dependencies can be reduced. This result is achieved by removing redundant array accesses. Furthermore nodes that represent array access operations that are unrelated to each other are not chained by sequence edges.
- The parallelism inherent in the array access operations is clearly exposed. The synthesis programs can benefit from the knowledge of the control dependencies when choosing a suitable memory structure. Possible choices are the number of read and/or write ports and the partitioning of memory into smaller parts.

In the initial DFG generation all access operations on an array are chained by sequence edges. However, this prevents that any optimization can be done; accesses cannot be done concurrently and it is not possible to change the order of the access operations. The following algorithm allows a chain of sequence edges and adds and removes sequence edges in such a way that the maximal parallelism between the accesses is obtained, while maintaining the data integrity.

In the algorithm the following notations are used. An address node, which determines the position in the array that is accessed, is connected to each access node . The address function \( \text{AD}(v_i) : V \rightarrow NU(\perp) \), returns the value of the address node connected to an access node \( v_i \), if the address node is a constant. Otherwise it returns \( \perp \). The function \( \text{AR}(v_i) : V \rightarrow V \) returns the array node, that is accessed by \( v_i \). Let \( X = \{ v_1, \ldots, v_n \} \) be a set of access nodes. The function \( \Phi(X, \kappa) : X \times N \rightarrow \Pi(X) \) returns the subset of \( X \) containing all nodes in \( X \) having an index node with value \( \kappa \), that is:

\[
\Phi(X, \kappa) = \{ v_i \mid v_i \in X \land \text{AD}(v_i) = \kappa \}
\]

In figure 2.8 an update operation on array \( A \) is shown. The access node is \( a2 \). The address node for node \( a2 \) is node \( a3 \), i.e. \( \text{AD}(a2) = 4 \). The array node for node \( a2 \) is node \( a1 \).

To determine the control dependencies for the array access nodes, the nodes are divided into sets, according to the operation they perform: retrieve or update and the index: constant or variable. This results in the following four sets: retrieve constant (RC), retrieve variable (RV), update constant (UC) and update variable (UV).
The retrieve constant set for an array \( x \) is denoted as \( RC_x \), a similar definition holds for the other sets.

The procedure in algorithm 2.1 is called once for each update and retrieve node, while traversing the chain of sequence edges for an array from the beginning to the end. Initially, the set \( UV_x \) will contain the array definition node. The procedure Connect_Array_Access first determines if the node \( v_i \) is an update or a retrieve node. Next the procedure determines whether the address is indexed with a constant or a variable. In the situation that \( v_i \) is a constant update access of array \( x \), \( v_i \) is connected to all nodes in \( UV_x \), \( RV_x \) and the nodes in \( UC_x \) and \( RC_x \) with the same constant address. Finally, the set \( UC_x \) is updated by removing the previous constant update node and adding \( v_i \).

In the other cases, depending on the node type, sequence edges are added to all nodes in (or subsets of) the sets given above. Then the sets are updated to reflect the new status after dealing with node \( v_i \).

The procedure shows the sequence edges that are added. It is assumed that all sequence edges that formed the original chain are deleted.
Algorithm 2.1: Connect array access

1. procedure Connect_Array_Access \((v_i)\)

2. \(x := \text{AR}(v_i);\)

3. If \(x(v_i) = \text{update} \) then

4. If \(\text{AD}(v_i) \neq \bot\) then

5. foreach \(v_j \in UV_x \cup RV_x \cup \Phi(UC_x, AD(v_i)) \cup \Phi(UC_x, AD(v_i)) \cup RC_x, AD(v_i))\) do

6. \(E_s := E_s \cup \{(v_j, v_i)\};\)

7. endforeach

8. \(UC_x := UC_x \setminus \Phi(UC_x, AD(v_i));\)

9. \(UC_x := UC_x \cup \{v_i\};\)

10. \(RC_x := RC_x \setminus \Phi(RC_x, AD(v_i));\)

11. else /* variable update */

12. foreach \(v_j \in UV_x \cup RV_x \cup UC_x \cup RC_x\) do

13. \(E_s := E_s \cup \{(v_j, v_i)\};\)

14. endforeach

15. \(UV_x := \{v_i\};\)

16. \(RV_x := UC_x := RC_x := \emptyset;\)

17. endif

18. else /* retrieve */

19. If \(\text{AD}(v_i) = \bot\) then

20. foreach \(v_j \in UV_x \cup \Phi(UC_x, AD(v_i))\) do

21. \(E_s := E_s \cup \{(v_j, v_i)\};\)

22. endforeach

23. \(RC_x := RC_x \cup \{v_i\};\)

24. else /* variable retrieve */

25. foreach \(v_j \in UV_x \cup UC_x\) do

26. \(E_s := E_s \cup \{(v_j, v_i)\};\)

27. endforeach

28. \(RV_x := RV_x \cup \{v_i\};\)

29. endif

30. endif

end

Algorithm 2.1 can be updated in such a way that the number of array accesses is reduced when possible. For instance, if a retrieve follows a retrieve with the same index, the second retrieve can be removed. Similarly, if a retrieve follows an update with the same index, the data that is written in the array can be used directly without reading it from the array again. The code fragment for the constant retrieve (lines 19 till 24) can therefore be replaced by the following code to perform these
optimizations. If there exists a previous update with the same index (say \( v_i \)) the source \( v_k \) of the incoming data to \( v_i \) is searched for. All nodes that use the data from \( v_i \) can now be connected to \( v_k \).

---

**Algorithm 2.2**: Connect array access (II)

```plaintext
1 (19) if \( \text{AD}(v_i) \neq \bot \) then
   if \( \Phi(UC_2, \text{AD}(v_i)) \neq \emptyset \) then
     let \( v_l \in \Phi(UC_2, \text{AD}(v_i)) \)
     \( v_k \in \{ v \in V | (v_k, v_i) \in E_{df} \land \pi((v_k, v_i)) = \text{data} \} \)
     foreach \((v_i, v_j) \in E_{df}\) do
       \( E_{df} := E_{df} \setminus \{(v_i, v_j)\} \);
       \( E_{df} := E_{df} \cup \{(v_k, v_j)\} \);
     endforeach
     \( V := V \setminus \{v_i\} \);
   else
     if \( \Phi(RC_2, \text{AD}(v_i)) \neq \emptyset \) then
       let \( v_k \in \Phi(RC_2, \text{AD}(v_i)) \)
       foreach \((v_i, v_j) \in E_{df}\) do
         \( E_{df} := E_{df} \setminus \{(v_i, v_j)\} \);
         \( E_{df} := E_{df} \cup \{(v_k, v_j)\} \);
       endforeach
       \( V := V \setminus \{v_i\} \);
     else
       foreach \( v_j \in UV_k \) do
         \( E_{df} := E_{df} \cup \{(v_j, v_i)\} \);
       endforeach
       \( RC_2 := RC_2 \cup \{v_j\} \);
     endif
   endif
```

Note that the modified algorithm effectively removes all array accesses from the data flow graph if there are only accesses with constant indices. All retrieve nodes will be connected directly to the node that produced the data and all update nodes become superfluous and can be removed by a dead-code elimination procedure. An example of the sequence edges that are added by the algorithm is shown in figure 2.9. The last retrieve of \( A[1] \) is connected directly to the data used in the last update.
of \( A[i] \), i.e. the constant node with value 3. The retrieve node and the index can be removed.

Two subsequent retrieves with the same variable index or an update and a retrieve can be optimized similarly as constant index accesses. A simple search for the origins of the indices can find whether the two indices have the same operation node as an immediate predecessor. If so, this sequence of variable accesses can be optimized as well.

Figure 2.9 Sequencing of array access operations.

2.10. Mutual Exclusion

In this section, three types of mutual exclusion will be discussed: the traditional mutual exclusion, data dependent mutual exclusion and dynamic mutual exclusion.

**Definition 2.7:** Two operations are mutually exclusive if they are executed in different branches of a conditional.

If an operation node \( v \) produces a value which is only used in a subset of the branches of a conditional, \( v \) does not have to produce a value under the other conditions. In the example of figure 2.10, the division and subtraction can be mutually exclusive. If the value of \( b \) is not used anymore after the if-statement, the division \( x / 2 \) has to be executed only when the test of the if-statement is true.

**Definition 2.8:** Two operations \( a \) and \( b \) are data dependent mutually exclusive if operation \( a \) produces data that is not used in the branch in which operation \( b \) occurs and nowhere after the conditional statement in which \( b \) occurs.
\begin{verbatim}
a := 1; b := x / 2;
if a < x then
    a := a + b;
else
    a := a - b;
endif
/* b is not used anymore */
\end{verbatim}

Figure 2.10 Mutual Exclusion Example.

When there is only one level of conditional statements, testing mutual exclusion involves only the check whether two nodes are in different branches of the conditional. However, when the DFG contains multiple levels of if- and case- statements, detection is more complicated. But several design steps in the synthesis process require the testing of mutual exclusion between each pair of operations. Time can be saved in the various synthesis steps if the calculation of the mutual exclusion is done in a preprocessing phase. In this phase, a vector can be assigned to each node which uniquely identifies under what conditions the operation is executed. Detection of mutual exclusion between a pair of nodes can be done by simply comparing the vectors assigned to both nodes.

An algorithm to assign code vectors to the nodes in various branches to detect mutual exclusion is given in [Park88]. A drawback of this marking scheme is that mutual exclusions which result from data dependencies are not detected.

One solution to this problem of data dependent mutual exclusion is to detect these operations and move them into the appropriate branches. Another approach, which is followed here, incorporates this information in the mutual exclusion coding. A coding similar to the one described in [Wakaba89] is used.

The following definitions are used to describe the mutual exclusion coding scheme. A conditional tree \( T = (B, U) \) represents the nesting structure of the conditional statements. A node \( b_i, (b_i \in B) \) is made for each branch of a conditional. There is an edge \((b_i, b_j) \in U\) if \( b_j \) is directly contained in \( b_i \), i.e., if the if-statement that contains \( b_j \) is directly within branch \( b_i \). The branches are encoded in such a way that each leaf of the conditional tree has a unique vector in a one-hot encoding. The length of this one-hot encoded vector is the \((\#if-statements + 1) + \#case_branches - (\#case-statements - 1)\). Figure 2.11 gives an example of a condition tree.

Using this one-hot encoded vector, a condition vector \( c \) can be defined for each node in the DFG:

\[ c(v_i) = \begin{cases} 
\text{one hot encoded vector for } b & \text{if } v_i \text{ is a node in leaf } b \\
\bigvee (c(v_1) \lor c(v_2) \lor \ldots \lor c(v_n)) & \text{if } v_i \text{ is not a node in a leaf branch}
\end{cases} \]
Figure 2.11  a) A skeleton for a set of nested if-statements and their corresponding basic blocks. b) The conditional tree. The condition vector is listed below each node.

where : \( v_1, v_2, \ldots, v_n \) are the successors of \( v_i \) and \( \lor \) is the bitwise logical or operation.

The mutual exclusion vector \( m(v_i) \) is defined for each node \( v_i \) in the data flow graph as:

\[
m(v_i) = \begin{cases} 
1 & \text{if } v_i = \text{put} \\
\tau(v_i) = \text{merge} & (v_i, v_k) \in E_{df}, \tau(v_k) = \text{merge} \\
m(v_1) \lor \ldots \lor m(v_n) & \text{otherwise}
\end{cases}
\]

where \( v_1, v_2, \ldots, v_n \) are the successors of \( v_i \) and \( \{1\} \) is a vector with all ones. Algorithm 2.3 calculates the mutual exclusion vector for all nodes in data flow graph \( G \). The procedure does a breadth-first search starting from the put nodes and updates all mutual exclusion vectors. When a node is a predecessor of a merge node the mutual exclusion vector is set equal to the condition vector.

The definition of mutual exclusion can be made more explicit.

**Definition 2.9.** Two nodes \( v_i \) and \( v_j \) are (data dependent) mutually exclusive if

\[
m(v_i) \land m(v_j) = \{0\},
\]

where \( \land \) is the bitwise logical and operation on two vectors.
Algorithm 2.3: Calculate Mutual Exclusion Vector.

1. procedure Calculate_Mutual_Exclusion_Vector (G):
   \[ L := \emptyset; \]
   \[ \text{foreach} \ v_i \in V \ do \]
   \[ m(v_i) := [1]; \]
   \[ L := L \cup \{v_i\}; \]
   \[ \text{endforeach;} \]
   \[ \text{while} \ L \neq \emptyset \ do \]
   \[ NL := \emptyset; \]
   \[ \text{foreach} \ v_i \in L \ do \]
   \[ \text{foreach} \ (v_j, v_i) \in E_{df} \ do \]
   \[ \text{if} \ \tau(v_i) = \text{merge} \ then \]
   \[ m(v_j) := c'(v_j); \]
   \[ \text{else} \]
   \[ m(v_j) := m(v_j) \lor m(v_i); \]
   \[ \text{endif} \]
   \[ NL := NL \cup \{v_j\}; \]
   \[ \text{endforeach} \]
   \[ L := NL; \]
   \[ \text{ endwhile} \]
   \[ end \]

When this algorithm is applied to the example of figure 2.10 the flow graph in figure 2.12 is obtained. All nodes are labelled with their mutual exclusion vector. The division and subtract node are mutual exclusive according to definition 2.9, i.e. \([10] \land [01] = [00]\). Another form of mutual exclusion appears in repetition. The exit nodes, used in the representation of the repetition, act similar to the branch nodes in the conditionals. For example, in a while loop, depending on the result of the test, the nodes in the body are executed again or the nodes after the loop are executed. The nodes in the body and the nodes after the loop can be considered mutual exclusive. However, this form of mutual exclusion is not determined statically but depends on the place where the repetition is broken, i.e. the place where a register is positioned. Therefore this is called dynamic mutual exclusion. All nodes in the loop body after the exit node and before the break in the loop are mutual exclusive with the nodes outside the loop after the exit node.

An example how the notion of dynamic mutual exclusion can lead to very efficient implementations is given in figure 2.13. In this figure the DFG for the following example is shown:
get ($p_1$, $y$);
$y := y + c_1$;
while ($y < c_2 < c_6$) do
$y := c_3 + y + c_4$;
endwhile
$y := y + c_5$;
put ($p_1$, $y$);

Note: all operations in this example are performed with constants as one operand, to keep the example simple. Of course some of these operations can be folded during constant propagation.

Because of the while loop at least two cycle steps are required. In figure 2.13 the loop is broken between the two additions in the loop body. This results in two cycle steps in the STG. The get node and the addition with $c_1$ are scheduled only in $s_0$, the addition with $c_4$ is scheduled only in $s_1$ and all other nodes are scheduled in both cycle steps. This break also results in a dynamic mutual exclusion between the addition with $c_3$ in the body and the addition with $c_5$ after the body. In the construction of the DFG, the mutual exclusiveness of these two nodes can be used by allocating
the same adder for both additions. The result of the loop test determines, by controlling a multiplexer, if the addition is done with c3 or c5.

The discussion about the dynamic mutual exclusion also showed some other advantages of the representation of repetition in the DFG:

• The nodes before the loop, in the test part of the loop, and the nodes after the loop can be assigned to the same state. If the test is immediately false only one state is required. In the example the whole path between the get and put nodes is executed in state s1, if the loop condition is immediately false.

• The implementation of the repetition itself forces the creation of only one additional state.

• The place where the repetition is broken (where a register is placed) can be determined dynamically and can be different for each value that circles in the loop. This allows a better distribution of operations over the available cycle steps. In the example breaking the loop between the addition with c3 and c4 makes the allocation of the same adder for the addition with c3 and c5 possi-
ble. If the loop was broken between the addition with $c_4$ and the entry node one more adder would have been needed.

These advantages are especially important in applications with very tight timing constraints and when the DFG has to be implemented in as few cycle steps as possible. Many of these situations will arise in applications where the delay of the operations is short compared to the length of a cycle step. In this situation, a sequence of operations can be assigned to a cycle step without violating constraints on the time a cycle step may take.

2.11. System Representation Overview

The system representations on the register transfer level are very similar in most high level synthesis environments. They are much alike the DFG and STG as described in the previous sections. The design representations at the behavioral level, that have appeared in the literature, show a much larger variety. Several criteria will be defined to classify the various data and control flow models.

- The design is represented by a tree or a graph.
- The design is represented by a true data flow graph or a so-called semi data flow graph. In the true data flow graph $G(V, E)$ the nodes $V$ represent only operations and the edges $E$ describe the data flow between these operations, independent from variable names. The semi data flow graph $G(V, W, E)$ is usually represented as a bipartite graph in which the node set $V$ represents the operations and the node set $W$ the variables. The edges $E$ represent a mapping from variables to operation inputs and from operation outputs to variables. Note that in the true data flow graph there is only a notion of values, while in the semi data flow graph variables are considered. Building a true data flow graph asks for a complete use-definition analysis of the input description, while the semi data flow graph can be directly derived from the input description.
- The design is represented by a separate data flow and control flow graph or a combined data and control flow graph.

Using the above criteria the following four classes are distinguished:

1) Tree representations

One of the ways to represent a design internally is by using a parse tree which is generated directly from the input description. IMEC's Cathedral system uses a tree directly derived from the Silage description [Hilfin84]. The Silage language itself is in fact a textual description of a signal flow graph. It is assumed that each signal flow graph is implicitly embedded in an outer loop. This notion is very useful in many signal processing applications in which inputs arrive, and outputs have to be produced at regular time intervals. A special operator in Silage is the delay operator.
2.11 System Representation Overview

This delay operator allows one to refer to the value of a variable in the previous iteration of the main loop.

The MIMOLA design system (Marwed96) also uses a tree format. The intermediate tree language, similar to the parse trees used in software compilers, is generated directly from the MIMOLA (Machine Independent Microprogramming Language) input language.

2) Semi data flow graph representations

The ADAM (Advanced Design Automation System) from the University of Southern California uses the DDS (Design Data Structure) as representation model [Knapp85]. The DDS uses four separate models for the data flow, timing and control, logical structure and physical structure. The data flow model consists of a bipartite directed acyclic graph where the two node types represent operations and variables.

To account for repetition a subscript is affixed to variable names and operations. At the top level of the behavioral hierarchy a single node represents the whole behavior; it is recursively subdivided, until the nodes represent primitive functions.

The control and timing model is a directed acyclic graph, whose nodes represent events, such as the initiation and termination of an operation and whose edges represent relations between events, such as operation durations and causality. In [Hayashi88] the model has been extended to incorporate timing constraints and both synchronous and asynchronous behavior.

The University of Karlsruhe's synthesis system uses three graphs that share the same nodes [Campos85][Campos89a]. The nodes in the graphs are the operations and the variables in the DSL input program. The first graph describes the predecessor relationship between the nodes, as derived from the functional specification. The second graph describes the data flow relation and the third graph the timing constraints between the nodes. The data flow relation describes the mapping from the operation nodes to the variable nodes. From these graphs a true data flow graph is constructed by introducing temporary variables for each multiple assignment during the synthesis. In [Campos86] details are given on how to include the timing information in this representation.

The last example of a semi data flow graph is the YIF format used in IBM's Yorktown Silicon Compiler [Brayton88]. The nodes represent the operations. The edges represent a precedence relation which is directly derived from the input specification. Furthermore, an input and an output relation are defined for each operation node, which describe the mapping between the variables and the nodes.

3) Separate true data and control flow graph representations

Two other systems use a separate data flow graph and control flow graph. The first example is the design representation for the synthesis of behavioral VHDL models as defined in [Campos89b]. Each operation node is present in both graphs. Initially, the ordering of the nodes in the control flow graph (CFG) is identical to the sequence
of operations in the input description. The edges at the end of the loop bodies are removed to make the graph acyclic. The edges in the data flow graph (DFG) show the data dependencies between the operation nodes.

At the University of California at Irvine a system based upon the VHDL language is under development. Their VHDL Synthesis System (VSS) [Lia88] also uses a separate control and data flow graph. A sequence of operations in which no control flow is present (called straight line code) is defined as a basic block. A data flow graph is made for each basic block. The sequence in which the basic blocks are executed is defined in the control flow graph. The nodes in the control flow graph are the basic blocks and the edges shows the sequence of executions of the different basic blocks. Repetition and procedure calls are always represented in the control flow graph, i.e. new basic blocks are made for each repetition and procedure call. Conditions are represented either in the control flow graph as separate basic blocks or in the data flow graph. If a conditional contains an iteration or a procedure call then it is always represented in the control flow graph.

The ASM (Algorithmic State Machine) charts as used by the University of Illinois' synthesis system [Pang87a] represent each basic block of straight-line with a data flow graph and adds control points to each block to represent the control flow between the blocks.

The main difference between the first and the other two is that in the first format the control flow graph contains the same nodes as the data flow graph, while in the second and third the control flow graph contains only a node for each basic block.

4) Combined data and control flow graph representations

The System Architect's Workbench (and its predecessors, the CMU-DA systems) use a combined data and control flow graph description: the Value Trace (VT) [Walk87] [McFarl78]. The VT is a directed acyclic graph. Procedures and labeled blocks are mapped onto subgraphs called vtbodyes. Operations are mapped onto nodes in the VT and values are mapped into edges. The conditional operations are represented by the select nodes. Originally select nodes were simply data multiplexers. Later this definition has been changed to select nodes, that choose from a set of alternative branches and execute only the operation nodes in that branch. Vtbodyes are called via call nodes, and the values that are inputs to the call nodes are passed to the vtbody. Repetition constructs are modeled by conditional calls to labeled blocks. [Nestor86] describes extensions to facilitate interface descriptions in the VT.

5) Other formats

Other design representation formats, about which not enough information is available to classify them but which are listed for completeness are: SIF (Sequencing Intermediate Form) [DeMich88] as used in the Olympus project at Stanford. In [Bhaskar88a] and [Bhaskar88b] an approach to represent VHDL processes can be
found. The author uses conventional compiler techniques, as described in [Aho86] to build a basic block representation and performs a use-definition analysis in each of these basic blocks.

In general, the basic block representations make it very difficult for the synthesis algorithms to look and optimize across the basic block borders. Most systems model the conditionals within the basic blocks, but the repetition is represented outside the basic blocks. The only two representations that do not have this problem are the DFG/CFG as defined in the IBM VHDL synthesis system and the DFG as defined in this chapter. They both allow synthesis across loop boundaries. However, the DFG/CFG forces a break in the repetition at the end of the loop body while the DFG allows to break a loop anywhere and separately for each data value, as long as there is at least one cut in the iteration of each data value.

A unique advantage of the DFG representation is that only a few nodes with different semantics have to be used to describe both the data and the control flow. The modelling of the control in the data flow graph makes it possible and straightforward to make a parallel representation of both the data and the control flow, even in the presence of repetition and array accesses. All synthesis algorithms can benefit from the explicit representation of the parallelism in the functional description.
3

Scheduling

This chapter deals with the scheduling problem in architectural synthesis. Informally, this problem can be described as determining in which cycle step each operation is going to take place. It defines also how many cycle steps will be needed to execute a whole data flow graph and how many modules are minimally required.

3.1. The Scheduling Problem

Two types of scheduling problems can be formulated with respect to architectural synthesis. The first type of problems assumes a set of modules and a set of operations which is to be serviced by the modules. Based on prespecified properties and constraints on the operations and resources, the problem is to find an efficient algorithm for sequencing the operations to optimize some desired performance. The measure that can be considered in this context is the schedule length, i.e. the time needed to execute all operations. This type of scheduling problems will be called: scheduling under hardware constraints. The second type assumes a set of operations and a given schedule length. The measure that can be considered in this context is the number of modules needed to execute the operations in the given time. This is called scheduling under timing constraints.

The system of operations \( v \in V \) is given in the data flow graph \( G(V,E_G,E_{\Delta}) \). The data flow graph represents a partial order \( \prec \) on all operations \( v \in V \), which specifies operational precedence constraints. In particular \( v_i \prec v_j \) means that operation \( v_i \) has to be fully executed before \( v_j \) can start. For any \( v_i \in V \), a set of predecessors is defined as: \( \prec_i (v_i) = \{ v' \mid v' \in V \land v' \prec v_i \} \). The set of immediate predecessors of \( v_i \) is denoted as \( \prec_i (v_i) \). Similar, for any node \( v_i \in V \) a set of successors is defined as: \( \succ_i (v_i) = \{ v' \mid v' \in V \land v_i \prec v' \} \). \( \succ_i (v_i) \) is the set of immediate successors of \( v_i \).

A schedule of the data flow graph is a mapping \( \Theta : V \rightarrow S \) associating with each operation a cycle step \( \Theta (v_i) \) such that,
v_i < v_j \Leftrightarrow \Theta (v_i) < \Theta (v_j) \tag{3.1}

With any schedule two parameters can be associated:

\begin{enumerate}
\item its schedule length $F$
\item its width $W(\Theta) = \max \{ \lvert \Theta (v_i) \rvert \mid \Theta (v_i) = \Theta \}
\end{enumerate}

The schedule length $F$ is the cycle step in which the last operation of the data flow graph finishes its execution. The width is a direct measure for the minimum number of modules needed in an implementation, if each module can execute all operations.

The two scheduler types, introduced informally above, can be described as follows: The scheduling under hardware constraints tries to minimize the schedule length $F$ given a width $W(\Theta)$ and a data flow graph $G(V, E, E_T)$. The schedule under timing constraints minimizes the width, given a schedule length $F$.

Given a schedule length $F$, two values can be defined for each node $v_i \in V$. The early execution time $e(v_i)$ is the earliest time the execution of $v_i$ can start considering that all predecessors have been executed earlier and assuming that the number of available processors is arbitrarily large.

**Definition 3.1:** The early execution time $e(v)$ of an operation $v$ is defined as:

\[ e(v) = \max_{v_i \in \mathbb{C}_O(v)} \{ e(v_i) + 1 \} \tag{3.3} \]

\[ e(v) = 0 \quad \text{if} \quad \mathbb{C}_O(v) = \emptyset \]

Assume that the execution of the data flow graph has to be finished at time $F$, i.e., the schedule length is $F$. The late execution time $l_F(v)$ is the time at which operation $v_i$ has to be finished to ensure the completion of the computation of the operations of the data flow graph at the time $F$ and assuming that the number of available processors is arbitrarily large.

**Definition 3.2:** The late execution time $l_F(v)$ of an operation $v$ is defined as:

\[ l_F(v) = \min_{v_i \in \mathbb{C}_O(v)} \{ l_F(v_i) - 1 \} \tag{3.4} \]

\[ l_F(v) = F \quad \text{if} \quad \mathbb{C}_O(v) = \emptyset \]

With each operation an execution interval $[e(v_i), l_F(v_i)]$ can be associated. Given a schedule length $F$ equation (3.1) can be replaced by the following restriction on the schedule:

\[ \Theta (v_i) \subseteq l_F(v_i) \tag{3.5} \]

### 3.2. The Scheduling Problem in Architectural Synthesis

To cope with the scheduling problems in the architectural synthesis area the general scheduling problem has to be refined. Several features with respect to this type of
scheduling and their impact on the formulation of the scheduling problem will be discussed.

**Library**

Up till now it was assumed that the modules could execute all operations. In architectural synthesis it is common to use a set of modules each of which can only execute a limited set of operations. A way to describe the relation between the types of operations and the modules is a library.

**Definition 3.3:** The library \( \Lambda (O, L) \) is defined by the set of operation types \( O \) and the set of library components \( L \). The library function \( \lambda : O \rightarrow \Pi (L) \), determines for each operation type \( o \in O \) by which library components it can be performed.

For example: let the library \( L \) contain the following components:

\[ L = \{ \text{ripple_carry_adder, carry_look_ahead_adder, ALU, array_multiplier} \}. \]

The library function for the plus-operation:

\[ \lambda(+) = \{ \text{ripple_carry_adder, carry_look_ahead_adder, ALU} \} \]

**Definition 3.4:** The cost of a component is a function \( c : L \rightarrow N \), where \( N \) the set of natural numbers, which specifies for each library component the cost. The term cost is used to describe the amount of area and power resources that are needed by a component.

**Definition 3.5:** The operation assignment function \( \alpha : V \rightarrow L \), maps each operation node of the graph on a library component.

The following diagram gives the relation between the library function and the operation assignment function. The type of the dataflow graph node determines on which set of library components the node can be mapped. The operation assignment chooses one of the library components from this set.

Since we do not have a universal set of components that can perform all operations, the width of the schedule is replaced by the cost. The cost of a schedule is defined as:

\[
C(\Theta) = \sum_{i \in L} c(i) \max_{s \in S} \{ |\Theta(v)| \mid \Theta(v) = s \land \alpha(v) = i \}
\]  

(3.6)

The cost of the components is used in this formula to account for the different costs of the different components. The scheduling algorithm not only deals with how many components have to be used, but also with how many of each type. Formula (3.6) reflects only the cost of the operational component area. To have a better estimate of the cost of the final implementation, also terms for register and interconnect area have to be added.
Time delay of Operations

It is very uncommon that all operations have equal delays. A simple and-operation will usually be much faster than a multiplication.

**Definition 3.6** The execution time \( ET(v) \) of an operation \( v \in V \) is defined as the time needed to execute the operation \( v \).

The execution time can be extracted from the library. When the component \( l = a(v) \), assigned to an operation \( v \), is known the time component \( t \) needs to execute an operation of type \( t(v) \), can be found in the library. The actual number of cycle steps an operation needs also depends on the cycle time. The cycle time \( T_{cycle} \) is defined as the time which is reserved for one cycle step. The cycle delay of an operation \( v, \in V \) is defined as:

\[
CD(v) = \frac{ET(v)}{T_{cycle}}
\]  

(3.7)

This definition of the cycle delay automatically leads to the definition of multicycling. An operation is called a multicycle operation if \( CD(v) > 1 \). More than one cycle is needed to execute this operation. Allowing multicycling enables the choice of a cycle time shorter than the delay of the slowest component. Thus, faster components can execute several operations during the cycle steps it takes for the slower components to finish their computation. Note that if multicycling is allowed an operation can be scheduled in more than one cycle step. In this situation the scheduling function \( \Theta \) will be a mapping \( \Theta: V \rightarrow \Pi(S) \).

**Chaining** of operations means the execution of several operations in a sequence within a cycle step. The operations \( v_1, \ldots, v_k \) can be chained if:

\[
\sum_{i=1}^{k} CD(v_i) \leq 1
\]  

(3.8)

An example of multicycling and chaining is given in figure 3.1. The execution time of the addition and subtraction are for example 40 ns and for the multiplication 80 ns. If the cycle time \( T_{cycle} \) is set to 60 ns the multiplication needs two cycle steps to finish its execution. If the cycle time is set to 100 ns the subtraction and addition can be chained.

The simple summation in equation (3.8) is only valid when chaining of operations does not have an impact on the speed of their combined implementation. However, in many applications chaining two operations will enable a faster implementation of the pair of operations. For example, when two carry look ahead additions are chained the carry chains can be connected. The result will be faster than the sum of two adder delays. These effects can be taken into account during synthesis, if the library specifies delay functions for combinations of components. These functions can be used instead of the simple summation in equation (3.8).
3.2 Scheduling in Architectural Synthesis

Incorporating the cycle delay changes the definitions of the early and late execution times.

**Definition 3.7:** The early execution time \( e(v) \) of an operation \( v \) is defined as:

\[
e(v) = \max_{v_i \in E_0 (v)} \{ CD(v_i) + e(v_i) \}
\]

\[
e(v) = 0 \quad \text{if} \quad E_0(v) = \emptyset
\]

**Definition 3.8:** The late execution time \( l_r(v) \) of an operation \( v \) is defined as:

\[
l_r(v) = \min_{v_i \in E_0 (v)} \{ l_r(v_i) - CD(v_i) \}
\]

\[
l_r(v) = F \quad \text{if} \quad E_0(v) = \emptyset
\]

In most scheduling algorithms both the cycle time and the execution time of each operation type have to be fixed before the scheduling algorithm starts. Evaluating the influence of different cycle times is usually done by iterating over the main scheduling algorithm.

There are two other issues that make the determination of the cost of a schedule slightly more complicated: **mutual exclusion** and **pipelining**.

**Mutual Exclusion**

When two similar operations are **mutual exclusive** (section 2.10.), they only need one module despite the fact that they are scheduled in the same cycle step. Mutual exclusive operations should not be counted multiple times in the sets in formula (3.6). For each branch, the number of operations of each type for that particular operation branch are calculated. Among all the mutual exclusive branches the maximal is taken and this is the value used in formula (3.6).
Pipelining

If the input to the system operates on samples of input data that appear at regular time intervals, the system can operate as a pipeline. If operations which work upon different instances of the input data are executed simultaneously, this is called pipelining. The latency is defined as the number of clock cycles between two consecutive executions of the algorithm. Two cycle steps are called pipelined if they are exactly one latency period apart.

\[
T_{\text{lat}} = 100\text{ns}
\]

Input rate = 100 ns

\[
\frac{\text{Total Execution time}}{T_{\text{cycle}}} = \frac{200}{50} = 2
\]

\[\chi = \frac{\text{input rate}}{T_{\text{cycle}}} = \frac{100}{50} = 2\]

\[i = \{0, 1, 2, 3\}\]

\[s_1 = \{1, 2, 3\}\]

\[s_2 = \{4, 5, 6\}\]

\[s_3 = \{7, 8, 9\}\]

\[s_4 = \{10\}\]

\[\frac{\text{Input rate}}{T_{\text{cycle}}} = \frac{100}{50} = 2\]

\[\chi = \frac{\text{Total Execution time}}{T_{\text{cycle}}} = \frac{200}{50} = 2\]

\[\chi = \frac{\text{Input rate}}{T_{\text{cycle}}} = \frac{100}{50} = 2\]

\[\text{Total Execution time} = 200\text{ns}\]

\[T_{\text{cycle}} = 50\text{ns}\]

\[\chi = \frac{\text{Input rate}}{T_{\text{cycle}}} = \frac{100}{50} = 2\]

\[\text{Input rate} = 100\text{ns}\]

Figure 3.2 Pipelining

Given the latency \( \chi \), the sets of pipelined cycle steps \( \{ PCS_1, ..., PCS_\chi \} \) can be calculated as follows:

\[ PCS_i = \{ s_i, s_{i+1}, ..., s_{i+\chi-1} \}, \quad 1 \leq i \leq \chi \]  

For example, for \( \chi = 3 \) and \( \chi = 10 \), the following \( PCS \) 's are obtained:

\[ PCS_1 = \{ s_1, s_4, s_7, s_10 \} \]
\[ PCS_2 = \{ s_2, s_5, s_8 \} \]
\[ PCS_3 = \{ s_3, s_6, s_9 \} \]

Taking into account the pipelining changes the original cost of a schedule to:

\[ C(\Theta) = \sum_{i=1}^{\chi} \max_{t \in \Theta_i} \{ \lambda(\Theta) \cdot \Theta(\eta_j) \in PCS_i \wedge (\eta_j) = i \} \]  

In the example of figure 3.2, the pipeline consists of two stages. The first stage consists of \( s_1 \) and \( s_2 \), the second of \( s_3 \) and \( s_4 \). Two sets of pipelined cycle steps are formed, \( PCS_1 = \{ s_1, s_3 \} \) and \( PCS_2 = \{ s_2, s_4 \} \). All cycle steps \( s_1 \) and \( s_4 \) in \( PCS_1 \) together contain two additions and two multiplications. Since a similar argument holds for \( PCS_2 \), the width for this example is:
3.3 Scheduling Schemes

\[ W(\Theta) = c(\text{adder}) \times 2 + c(\text{multiplier}) \times 2 \]

When both multicycling and pipelining are used, none of the operations may have a cycle delay larger than the latency $\gamma$.

Pipelining can also have a large impact on the number of values to be stored. An efficient pipeline scheme can be constructed, in which many values have to be stored for only a few cycle step transitions. In (Goosse89a) many of these aspects concerning pipelined implementations are discussed.

All the features discussed in the previous sections can be incorporated one way or another in the various known scheduling algorithms. In some algorithms it may be easier to include a certain feature than in another, but there are no insurmountable problems.

3.3. Scheduling Schemes

Most existing scheduling algorithms can not handle data flow graphs that contain loops and procedure calls. Generally the data flow graph is partitioned into various subgraphs, which are handled by the scheduler one at a time. The order in which the data flow nodes are visited during scheduling is called a scheduling scheme. The design representation has a large impact on the scheduling scheme used. Before going into detail on scheduling algorithms the various schemes will be discussed.

A scheduling scheme determines in which order the data flow nodes are scheduled. Two different schemes are discussed here: a block-by-block scheme, which is extended in a hierarchical scheme, and a path-based scheme.

Almost all scheduling methods are applied on a block-by-block basis. In general these methods do not attempt to schedule across the block boundaries. This is suggested by a design representation, which strictly divides the data flow graph into basic blocks. A drawback of such scheduling schemes is that a cycle step transition is forced at each basic block boundary. This is not a real problem when the basic blocks itself are scheduled in a large number of cycle steps. But in applications with a lot of control flow, relatively to the amount of data processing, i.e. a large number of basic blocks each containing only a small number of operations, this can become a problem.

The first scheme that does some sort of scheduling across block boundaries is described in (Potkon90). The hierarchical scheme is an extension of the block-by-block schemes. The hierarchical scheduling method first schedules each basic block using a given set of modules. From the schedule the global timing is calculated. If there is some time left, an "expensive and not so much used" module is removed from the set of available modules. All blocks affected by the removal of the module are scheduled again. If the new schedule does not violate the timing constraints it is accepted. The process continues until no module can be found whose removal will not violate a timing constraint.
Another scheme is the path-based scheme [Bergam90]. This scheme relies on the representation using a separate control flow graph. A path in the control flow graph results from the conditional data flow. Loops are broken at the end and considered as two way conditionals, the loop body is executed or is not executed. In the path-based scheme all possible paths in the control flow graph are scheduled separately. Note that the path-based scheme maintains the order of the nodes as specified in the control flow graph. In [Bergam90] this order is directly derived from the sequential input description. The scheme only determines, for each path separately, between which operations the cycle step transitions are placed. As a result an operation that occurs in different paths can be scheduled in different cycle steps when the boundaries are chosen differently for the paths. A path-based scheme is useful when a very fast schedule is required which contains a lot of chained operations. This situation occurs when the cycle time is much larger than the execution time of most operations. Applications for a path-based scheme can mainly be found in the processor area. A processor description consists of a large number of parallel paths, one for each instruction, which in turn contain very simple operations. In the extreme case of a RISC processor each path has to be executed in a single cycle step and all operations in a path will be chained.

3.4. Overview of Scheduling Algorithms

The scheduling algorithms can be divided in four categories. The transformational scheduling algorithms try to improve an existing schedule by applying transformations to it. The algorithms based on integer linear programming formulate the scheduling problem as an ILP problem. The algorithms based on neural networks try to find a solution using rules for self organization. The iterative algorithms schedule the operations one at a time. Different strategies are developed which operation to pick first.

3.4.1. Transformational Scheduling

Transformational scheduling algorithms start from either a maximal parallel schedule or a full serial schedule. In the first case the schedule is serialized by applying transformations, the second case works the other way around and applies parallelizing transformations. The scheduling in the YSC [Brayton86] is an example of the first type. The scheduling algorithms starts with the least control steps possible. Initially, only control steps for loops and procedure calls are present. Furthermore, some cycle steps due to constraints and data flow restrictions are added. Then, the cycle steps in which the combinatorial part is too slow or that require too much hardware are split. During this process the critical paths in each control step are balanced. The CAMAD system [Peng86] uses the second type of transformational scheduling.
3.4.2. ILP-Scheduling

Two similar approaches [Lee89] and [Papach90] formulate the scheduling as an integer programming (ILP) problem. A decision variable \( x_{ij} \) is defined as one if operation \( i \) is assigned to cycle step \( j \) or zero otherwise. Another set of variables \( M_k \) describes how many modules of type \( k \) are allowed. Constraints are added such that (a) in no cycle step the schedule contains more than \( M_k \) operations of the type \( k \), (b) an operation is scheduled somewhere in its execution interval and (c) all predecessors of an operation are scheduled before the operation itself. The objective function states that the cost of the modules should be minimized. Solving this linear program results in an assignment of operations to cycle steps. This ILP formulation describes a scheduling under timing constraints. A slightly different formulation [Huang90a] allows the scheduling under hardware constraints. However, solving this general ILP-problem for middle size data flow graphs becomes already very time consuming. Some techniques [Huang90a] look only at a few cycle steps at a time in order to reduce the ILP-problem size.

3.4.3. Neural Network Scheduling

In [Hemani90] an algorithm for scheduling data flow graph operations using a self-organizing algorithm is described. This scheduler is of the time constrained type. In the neural network model all operations compete for a cycle step that falls within their execution interval. The model takes into account the effect of scheduling an operation on the schedule of its neighbors. The size of the neighborhood that is considered is gradually reduced until only primary moves (only a move of the operation itself) are possible. The net effect is a uniform distribution of the operations over the schedule space. A drawback of this type of scheduling methods is that they are very slow. An advantage is that it is easy to adapt this sort of algorithms to run on massively parallel machines.

3.4.4. Iterative Scheduling

The iterative scheduling algorithms are often called constructive algorithms. The iterative algorithms schedule one operation at a time. Two different iteration schemes can be distinguished. One scheme visits the cycle steps in order. The current cycle step is completely filled with operations that seem most appropriate before proceeding to the next cycle step. The other scheme works operation by operation. An operation is selected according to certain criteria and that operation is scheduled in the cycle step that seems the best at that moment. The as soon as possible and the list schedulers belong to the first type. The critical path scheduler and the distribution based schedulers are of the second type.
Algorithm 3.1: ASAP scheduling

procedure ASAP_scheduling (G);
list := initial_nodes of G.
i := 0; new_list := Ø ;
while list ≠ Ø do
 foreach v ∈ list do
 Ω (v) = i ;
 foreach v' ∈ △i (v) do
 if (∀p ∈ △i (v') | p is scheduled) then
 newlist = newlist U {v'} ;
 endif
endfor
endfor
list := new_list;
n newList := Ø ;
i := i + 1;
endwhile
end

As Soon As Possible

The simplest form of an iterative scheduling algorithm is an as soon as possible (ASAP) scheduling as described in algorithm 3.1. This type of scheduling was used in the first CMU/DA system [Teeng83]. The ASAP-scheduling (when applied without hardware constraints) always guarantees the fastest possible execution of the data flow graph, however a surplus of hardware can be needed.

Note that this algorithm can be used to determine the early execution times of all nodes. An algorithm similar to algorithm 3.1 starting from the terminal nodes can be used to determine all late execution times. An algorithm like this is called an As Late As Possible (ALAP) scheduling.

The description of an ASAP scheduling, given in algorithm 3.1, automatically leads to the list scheduling algorithm.

List Scheduling

Instead of scheduling all nodes from the list in algorithm 3.1 immediately, some criteria can be used to delay the scheduling of certain operations. Usually an upper-bound on the number of modules that are available limits the number of operations that can be scheduled in a certain cycle step. A new cycle step is forced before all operations from the list are scheduled. A criterion has to be found which operations have to be postponed to the new cycle step.

In ELF [Giroux85] an urgency weight is used as a criterion. The urgency weight for an operation is defined to be the minimum number of cycle steps required to ex-
3.4 Overview of Scheduling Algorithms

Execute the portion of the data flow graph from the operation to the nearest enclosing timing constraint. Essentially, a partial ALAP scheduling is done. Operations with the highest urgency are selected first from the list. A similar approach is followed in the CSTEPE [Thomas98] scheduler.

Splitter [Fangrie87a] initially performs both ASAP and an ALAP scheduling. The difference between the ALAP and ASAP schedule of a node is called the mobility. The mobility is similar to the execution interval. The list scheduling is performed with the mobility as a criterion. Since all nodes on the critical path have mobility zero, priority is given to these nodes. To break a tie between two nodes with the same mobility, the node with the highest number of successors is chosen first.

SEHWA [Park98] uses a combination of list schedulers. First a maximal scheduling (a list scheduling based on urgency without hardware constraints) and a feasible scheduling (a list scheduling based on urgency with hardware constraints) are done. If the maximal scheduling is better than the feasible one there is a possibility of existence of a shorter feasible schedule. In this case an exhaustive scheduler (complexity \(O(n^3)\)) is used to find an optimal schedule.

The HAL system [Paulin89a] uses the forces (which will be discussed later) as a criterion to select nodes from the list to be scheduled.

Critical Path Scheduling

Another category of iterative algorithms schedules all operations on the critical path first [Parker86]. For all remaining operations the freedom is calculated. (Note that the the freedom is similar to the term execution interval before). The nodes are assigned to a cycle step in the order of increasing freedom.

In ATOMICS [Goosse87] the operations on the longest critical path are scheduled first as well.

Distribution Based Schedulers

Other iterative schedulers are based on the distribution of the operations among the for each operation available available cycle steps. In these approaches the operations are assumed to have a uniform probability of being scheduled somewhere in the execution interval. Based on these distributions several strategies can be defined which operation to schedule next in what cycle step. The first scheduler of this type is the force directed scheduler [Paulin87]. This scheduler will be discussed in more detail in the next section since it is applied in EASY.

Another one is the CASCH scheduler as applied in CADDY [Kräuser90]. In this scheduler a mean value of the number of operations of a certain type that is executed in a certain cycle step is calculated. The delay introduced by constraints on the number of modules is calculated by adding up the difference in the mean value and the available number of modules for each operation type over the remaining cycle steps. Because several modules can perform more types of operations, the influence of a certain distribution on the others is taken into account. This is done by assigning
the types for which the least modules are available first, and correcting the mean values for the already assigned types.

### 3.5. Force Directed Scheduling

In this section the basic force directed scheduling according to [Paulin87] is described. The following section contains some enhancements made to force directed scheduling in the EASY system. The force directed scheduler uses the schedule length $F$ to calculate the initial distribution of operations over the available time. The operations for which their execution interval $I(v)$ is equal to their cycle delay $CD(v)$ are on the critical path. These operations are fixed in the cycle steps between their $e(v)$ and $i(v)$. All other operations (free operations) can be shifted in their execution intervals. During an iterative process in which a free operation is scheduled in each operation an attempt is made to obtain a distribution that minimizes the width of the schedule. It is tried to obtain a distribution of operations over the cycle steps such that operations which can be implemented by the same module are assigned to distinct cycle steps. A load balancing technique is used to distribute the operations evenly over the cycle steps.

#### Probabilities

The probability that an operation is scheduled somewhere in its execution interval is assumed to be uniformly distributed over each cycle step in this interval. $P(v, i)$ is defined as the probability that an operation $v$ will be scheduled in cycle step $i$. When all operations have a cycle delay equal to 1, i.e. no modulo cycling, the probability is defined as:

$$
P(v, i) = \begin{cases} 
\frac{1}{I(v)} & e(v) \leq i < I(v) \\
0 & \text{otherwise} 
\end{cases} \quad (3.13)
$$

#### Distribution Functions

For each operation type set $\eta$, a function $DF_\eta(s)$ is defined which gives the density of the operations with a type belonging to $\eta$ in cycle step $s$. In [Paulin88] this is called a distribution function. This term will be used here as well, despite the fact that it is not a true distribution function in the sense as used in statistics. The distribution function for an operation type set is defined as the sum over the probabilities of all operations that belong to the operation type set:

$$
DF_\eta(s) = \sum_{v, i \in \eta} P(v, s) \quad (3.14)
$$

In cycle steps where the distribution function has a large value for a given operation type set the concurrency of this kind of operations is high. If the concurrency of operations which belong to the same operation type set has to be reduced, operations should be removed from cycle steps with large values for their distribution function.
3.5 Force Directed Scheduling

Forces

Assume that an operation $v$ is fixed in a particular cycle step $k$. A force $SF(v,k)$ reflects the influence of this assignment on the distributions.

$$SF(v,k) = \sum_{s \in S} DF_y(s) \cdot \Delta P_{v,s}(v,s)$$  \hspace{1cm} (3.15)$$

where: $\Delta P_{v,s}(v,s)$ is the change of the probability that operation $v_j$ is scheduled in cycle step $s$, when operation $v_i$ is scheduled in cycle step $k$. When $v_i = v_j$ this reduces to:

$$\Delta P_{v,s}(v,s) = \begin{cases} 
- \frac{P(v,s)}{1 - P(v,s)} & s \neq k \\
\frac{1}{1 - P(v,s)} & s = k 
\end{cases}$$  \hspace{1cm} (3.16)$$

Formula (3.15) reflects the change in the distribution of the operation itself when scheduled in a cycle step. This assignment may also affect the execution intervals of the predecessors and successors of the operation. This can be taken into account by adding these forces to the original self-force defined in (3.15). The predecessor forces are defined by:

$$PredF(v,k) = \sum_{v_i \in D(v)} \sum_{s \in S} DF_y(s) \cdot \Delta P_{v,s}(v_i,s) \hspace{0.5cm} \text{where: } \tau(v_i) \in \eta_i \hspace{1cm} (3.17)$$

Similarly, the successor forces can be defined as:

$$SuccF(v,k) = \sum_{v_i \in C(v)} \sum_{s \in S} DF_y(s) \cdot \Delta P_{v,s}(v_i,s) \hspace{0.5cm} \text{where: } \tau(v_i) \in \eta_i \hspace{1cm} (3.18)$$

The influence of scheduling operation $v$ in cycle step $k$ on the overall distribution of the operations is now reflected by the (total) force:

$$F(v,k) = SF(v,k) + PredF(v,k) + SuccF(v,k)$$  \hspace{1cm} (3.19)$$

This force will be used in the following scheduling algorithm.

Scheduling Algorithm

First the execution intervals are determined by an ASAP and ALAP schedule and the distribution functions are calculated. In each iteration of the load balancing algorithm one operation is selected and assigned to a cycle step (or range of cycle steps in case of a multicycle operation). For all operation–range pairs the force is calculated. The operation–range pair with the lowest force is selected. Then the execution intervals and the distribution functions are updated and the next iteration is started.
Algorithm 3.2: Force Directed Scheduling

ASAP-scheduling:
ALAP-scheduling:
calculate Distribution Functions;
while (there exist operations that are not fixed) do

calculate all forces;
select and assign operation and range with lowest force;
update Execution Intervals;
update Distribution functions;
endwhile;

In the update of the late execution times only the nodes of which the selected operation is a predecessor have to be updated. When in turn for a node its execution interval does not change, the successors of this last node have not to be updated. The same holds for predecessor nodes of the selected node when the early execution times are updated. Distribution functions only have to be updated if the execution interval of one of the members of the operation type set is changed. For these functions, the values in the affected intervals are calculated again.

3.6. Extensions to Force Directed Scheduling

Multi Cycling

Initially, it is assumed that the execution time of the operations is an integer multiple of the cycle time, i.e. the \( CD(v) \) is an integer. If multicycling is assumed the probability, as defined in (3.13), has to be replaced by equation (3.20).

\[
P(v, i) = \frac{N(v, i)}{(1/l(v) - CD(v) + 1)}
\]

(3.20)

where \( N(v, i) \) gives the number of assignments in which cycle step \( i \) is occupied by node \( v \) and \( l(v) \) is the length of the interval \( l(v) \), i.e. \( l(v) = v - e(v) \), i.e.

\[
N(v, i) = \min\left\{ \begin{array}{ll}
i - e(v) + 1 & \text{if } e(v) \leq i < l(v) \\
l(v) - i & \text{if } i < e(v) \\
CD(v) & \text{if } i \geq l(v) \\
\end{array} \right.
\]

These equations are illustrated in figure 3.3. In this example we schedule an operation with cycle delay 2 in an execution interval with length 4. The operation can be placed in three different positions. Cycle step \( s_1 \) is occupied in one of these three placements thus \( P(v, s_1) = 1/3 \). Similar \( P(v, s_2) = 2/3 \) etc.
3.6 Extensions to Force Directed Scheduling

![Diagram](image.png)

**Figure 3.3** Cycle step occupation with multicycling.

**Pipelining, Mutual Exclusion, Chaining**

Pipelining can be added to the force directed scheduler as described in [Stok88a]. The method as described in section 3.2. can be applied directly to the force directed scheduler. Instead of calculating a distribution function $DF_s(x)$ for each cycle step, a distribution function $DF_{PCS_i}$ is calculated for each set of pipelined cycle steps $PCS_i, 1 \leq i \leq X$. This distribution function is defined as:

$$DF_{PCS_i} = \sum_{s \in PCS_i} DF_s(s)$$

If two operations are mutual exclusive they should not both contribute to the distribution function. A way to decide which one has to be included in the distribution function is the following. If, for each path in the data flow graph a probability is known how often it is executed, the path probability can be multiplied to the probability defined in equation (3.20), for each node in the path. The modified probabilities will be used in the calculation of the distribution functions. If the path probabilities are not known, only the operation with the largest probability among the mutual exclusive ones contributes to the distribution function.

Chaining is implemented in the EASY force directed scheduler by maintaining the early execution times $e(v)$ and late execution times $l(v)$ in fractions instead of in integers. When the distribution functions are calculated, the early execution time is rounded to the nearest smaller integer and the late execution time to the nearest greater integer. The update of the early and late execution times is done in the fractions. As soon as the execution interval is equal to the cycle delay of an operation this operation is considered fixed. This update scheme automatically leads to the possibility of chaining of operations within a cycle step, when their summed cycle delays do not exceed the cycle time.
Chapter 3. Scheduling

Operation Type Sets

To describe which operation types can be combined the concept of operation type sets is introduced. An operation type set describes the set of operations that can be implemented by a single module. Most libraries result in operation type sets that are disjoint. If the operation type sets are not disjoint a preference factor as described in [Stok86a] can be used to choose between the possible implementations of an operation.

Complexity

The complexity of determining all forces can be calculated as follows. Suppose, there are \( n \) nodes, each of which can be assigned to \( |S| \) cycle steps. Each of these assignments can effect the probabilities of \( n \) (including itself) other operations, which in turn have at most \( |S| \) possible assignments. Thus calculating all forces is of complexity \( O( |S|^2 n^2 ) \). Since in each iteration of the scheduling algorithm 3.2 one operation is scheduled, \( n \) iterations can be possible. The complexity of the force directed scheduling is therefore \( O( |S|^2 n^2 ) \).

In [Paulin89] it is claimed that an efficient update scheme of the predecessor and successor forces reduces the complexity to \( O( cn^2 ) \). The update scheme is (cited from [Paulin89]): “The first phase consists of calculating and storing the self-force of all operations. In the second phase, the graph is traversed from bottom to top and each operation "queries" its immediate successors for their stored force. In turn, their stored force is made equal to the sum of their own self force and the stored force of their immediate successors. In the third and final phase, the process is repeated by traversing the graph from top to bottom and performing a running sum of the predecessor forces. This second method reduces the complexity to \( O( c n^2 ) \).”

The \( c \) replaces the \( |S|^2 \), since he does not include the schedule length in its complexity analysis. However, when the update scheme is applied to graphs with convergent paths, as all data flow graphs are, this will not result in the correct calculation of the predecessor and successor forces. The problem is that some forces will be counted multiple times.

Applying this scheme to the example of figure 3.4 results in the forces displayed next to each node. Since there exist three paths from \( E \) to \( A \) the force from \( E \) will be counted three times in the force of \( A \). We conclude that the suggested update scheme does not calculate the forces as defined before and therefore the complexity of the force directed scheduling is \( O( |S|^2 n^3 ) \).

A typical run with the force directed scheduler in EASY is shown in figure 3.5. The initial, final and two intermediate distributions are shown. The first intermediate result is plotted after 6 iterations and the second after 10 iterations of the main algorithm. As an example a finite impulse response filter is used. It is scheduled in
3.7 Creation of Storage Nodes

13 cycle steps. The first and the last cycle step are used for input and output. A library is chosen in which an array multiplier is four times as expensive as an adder. It can be seen from the figure that in the first iterations a lot of effort is done to equalize the distribution of the multipliers. In the later iterations also the distribution of the adders is made more smoothly. The final schedule uses two adders and two multipliers.

3.7. Creation of Storage Values

When the scheduling is finished each node \( v_i \in V \) has been assigned a sequence \( \Theta (v_i) \) of cycle steps in which it is executed. The following two functions are defined on the sequence \( \Theta (v_i) \). The function \( \vdash: \Pi(S) \rightarrow S \) gives the first cycle step in the sequence \( \Theta (v_i) \), i.e., \( \vdash (\Theta (v_i)) \) is the cycle step in which the execution of \( v_i \) is started. The function \( \dashv: \Pi(S) \rightarrow S \) returns the last cycle step in a sequence. From this schedule it can be determined which values have to be stored in registers, i.e., which edges cross cycle step boundaries. These values are called storage values. Algorithm 3.3 inserts nodes in the data flow graph for each storage value.
Figure 3.5 Equalization of distributions by force directed scheduling.
**Algorithm 3.4**: Insertion of Storage Values

```plaintext
procedure Insert_Storage_Nodes ( G):
  foreach $v_i \in V_{gf}$ do
    if $\exists v_j \mid (v_i, v_j) \in E_{gf} \land \mathcal{H}(v_i) \not\models \mathcal{H}(v_j)$ then
      let $v_i$ a new node with $\pi(v_i) =$ register;
      $V := V \cup \{v_i\}$;
      foreach $(v_i, v_j) \in E_{gf}$ do
        if $\mathcal{H}(v_i) \not\models \mathcal{H}(v_j)$ then
          $E_{gf} := E_{gf} \setminus \{(v_i, v_j)\}$;
          $E_{gf} := E_{gf} \cup \{(v_i, v_j)\}$;
        endif
      endfor
    endif
  endfor
  $V := V \cup V_i$;
end
```

For each node $v \in V_i$, the following functions are defined:

**Definition 3.9**: The function $\omega : V_i \rightarrow S$ determines for each storage value when it is written.

**Definition 3.10**: The function $g : V_i \rightarrow \Pi (S)$, determines the set of cycle steps in which the storage value is read. The function $P : V_i \rightarrow S$, determines the last cycle step in which the storage value $v \in V_i$ is read. Thus, $P(v) = \max (g(v))$.

A storage value $v$ that is written in state $\omega(v)$ and read for the last time in the state $P(v)$ is called life in the interval $\mathcal{R} \omega(v) \cdot P(v)$ . The interval will be called the lifetime interval of a storage value $v$. For the the value $v_3$ in figure 3.6 this results in: $\omega(v_3) = s_1$, $g(v_3) = \{s_2, s_3\}$ and $P(v_3) = s_3$. The value $v_3$ is life in : $\mathcal{R} s_1 \cdot s_3$.

For any multicycle operation, there is a certain interval in time where the input values are required to be stable. This time interval has to be specified in the library. This time interval can be used to determine the last cycle step in which a value is read.

### 3.8. Review of Scheduling Techniques

Recent studies [Verhaegh91] have shown that force directed scheduling can produce high quality results. The improvements include a modification in which operations are not immediately fixed in their intervals but where the execution intervals are gradually reduced. The application of dynamic weight factors (spring constants) to the distribution functions did improve the results further. However, when a large
number of operations are involved (> 100) and the schedule length $F$ is somewhat larger (> 2 times) than the critical path, force directed scheduling becomes very time consuming. Experiments in speeding up the algorithm [Heijlig91] by neglecting or approximating some factors in the algorithm did all deteriorate the final results.

Other approaches [Guter91] [Heijlig91] try to make hardware constrained scheduling algorithms more effective. They incorporate more global analysis in the list scheduling algorithms. The run times of these algorithms are usually considerably faster, but they still produce somewhat lesser quality results.

The best way to go seems to develop algorithms, that can deal with both strong hardware bounds and strong time constraints. They can produce good results taking advantage of the reduction in search space provided by these hard bounds.
Chapter 4

Data Path Allocation

The allocation of a datapath in a high level synthesis system consists basically of three mutual dependent subproblems. The register allocation assigns registers to all values that have to be stored. The module allocation assigns modules to all operations. Finally, the interconnect allocation connects all modules that are allocated in the previous steps. The approaches which have been taken to solve the data path allocation problem can be divided in two categories: strategies that solve the three problems simultaneously or strategies that solve them sequentially.

Approaches that follow the first strategy immediately face the problem that finding an optimal solution by an exact algorithm becomes intractable even for small data flow graphs. This leads to various sorts of algorithms with a heuristic nature. The sequential approaches have to find a way to decouple the three problems. As will be shown in the sequel, the order in which the problems are solved is extremely important and has a large impact on the final result. The sequential approach has the advantage that some of the subproblems can be solved optimally. This type of algorithms usually solves a subproblem for the whole data flow graph at once and will therefore be called global algorithms.

This chapter starts with a description of the terms that will be used in the formal model of the data path allocation. In section 4.2, an overview of the allocations schemes that have been published is given. Emphasis is put on the schemes with a global nature. Among others the allocation strategy of the early EASY (Eindhoven Architectural Synthesis System) is described.

4.1. Definitions

First, the definitions of the data flow graph (DFG) and data path graph (DPG) as given in chapter 2 are refined. The module set \( M \) is refined as \( M = M_c \cup M_l \cup M_1 \), where \( M_c \) are the modules used to execute the operations, like adders, ALU's and multipliers, \( M_l \) are the modules used to store values, like registers and register files.
and \( M \) are modules used for interconnection like, multiplexers, bus drivers and tri-state buffers. Similarly, the set of nodes in the data flow graph (DFG) is partitioned into \( V = V_o \cup V_i \), where \( V_o \) are the operation nodes as defined in chapter 2 and \( V_i \) are the values that have to be stored in registers. The storage values are added to the DFG during or after the scheduling by a procedure as described in section 3.7. The sets, thus defined are used in the following definitions.

When data path allocation is done a collection of modules is needed to realize the various operations. The modules and operations are described by a library as defined in section 3.2. In most applications not the full library is used in the final realization. The module selection function specifies which modules from a library are used in a data path.

Given a datapath \( P \):

**Definition 4.1:** The module selection function \( \sigma : L \rightarrow N^L \), where \( N \) is the set of natural numbers, determines how many library components of each type are instantiated from the library \( L \) in the data path \( P \).

The module selection function \( \sigma(L) = \{1, 0, 2, 0\} \) specifies that, given the example library \( L \) as defined in section 3.2, the data path will contain one ripple_carry_adder and two ALUs. The module selection function determines which components from \( L \) are used to form the operational modules \( M_o \) of the datapath \( P \). In the example:

\[ M_o = \{ \text{ripple_carry_adder}_0, \ ALU_0, \ ALU_1 \} \]

Given a DFG and a module library:

**Definition 4.2:** The module allocation function \( \mu : V_o \rightarrow M_o \), determines which module performs a given operation.

Note that a module allocation \( \mu(v) = m \), \( m \in M_o \), \( v \in V_o \) can only be a valid allocation if \( m \in \lambda(\tau(v)) \).

**Definition 4.3:** The register allocation function \( \psi : V_s \rightarrow M_s \), identifies the storage module holding a value from the set \( V_s \).

**Definition 4.4:** The interconnect allocation function \( \iota : E_y \rightarrow M_i \), describes how the modules and registers are connected and which interconnection is assigned for which data transfer.

The mappings as defined above describe the most general relations between the DFG and DPG. The scheduling determines if these mappings are many-to-many or many-to-one mappings. Two situations can occur which lead to the fact that an operation is scheduled in more than one cycle step:
4.2 Overview of Allocation Schemes

1) If the cycle time is smaller than the time the module needs to execute an operation, a sequence of cycle steps can be assigned. This is what is called multitasking.

2) If an operation is executed more than once under different conditions, the operation can be scheduled in different cycle steps under the different conditions.

In the first case only a single module will be assigned to the operation. In the second case the operations can be mapped under the various conditions to different modules as well. In this situation the module allocation is a many-to-many mapping. A similar reasoning can be made for the register allocation. If situations like 2) do not occur in the schedule the allocations can be described by many-to-one mappings.

In all literature published on data path allocation, as known to us, the various data path allocations are treated as many-to-one mappings. Therefore, the terms module allocation, register allocation and interconnect allocation will be used in their many-to-one meaning in the following discussion about the various allocation algorithms.

4.2. Overview of Allocation Schemes

In this section an overview of the allocation schemes as they have been published thus far will be given. An allocation scheme consists of a set of algorithms which together form the whole data path allocation. A scheme is characterized by the type of algorithms that are used and by the order in which these algorithms are applied.

The following types of algorithms will be distinguished: 1) Rule based / expert systems 2) Greedy / Iterative algorithms 3) Branch & Bound algorithms 4) Linear Programming Algorithms 5) Allocation by logic synthesis 6) Global Algorithms.

The order in which the allocations are done is especially important in the global algorithms. In establishing an order two things have to be taken into account. On one side, what information from a previous step can be used advantageously in the later steps. On the other side, attention has to be paid to the fact that a previous step may prevent a later step from finding an optimal solution to its subproblem. If, for example, the allocation of interconnect takes place when the allocation of the values and operations are already fixed, the impact of sharing a connection by two data transfers can be evaluated exactly. However, the interconnect allocation can not profit anymore from the fact that a different module allocation can result in a much better interconnection pattern. In fact, it is very important to look at the elements that have the largest impact on the data path and find an order which optimally allocates these elements.

Many systems use a mixture of the algorithm types during the various phases of the data path allocation. In the next overview the systems are organized by the
type of the algorithm that constitutes the main part of the data path allocation in this system.

4.2.1. Rule Based Schemes

Here, two systems that mainly rely on rule-based expert systems are described. DAA (Design Automation Assistant) [Kowal85a] [Kowal85b] uses a knowledge-based expert system approach. First the DAA allocates memories, global registers and constants using the expert system. Second, it partitions the design and allocates cycle steps, operators, registers and control logic on a block by block basis. Third, local optimizations are applied to remove or combine registers, combine modules etc. Finally, global optimizations are applied to remove unreferenced modules, unneeded registers and to allocate bus structures. The DAA uses the BUD (Bottom-Up Design) system [McFar86b] [McFar86a] to provide the expert system with information with regard to partitioning. BUD takes the dataflow operations and values in the dataflow graph and partitions them into clusters that have geometrical as well as logical meaning. Clusters are made based on a distance metric. This distance metric considers the amount of common functionality, the degree of interconnect and the potential parallelism between operations. Each cluster will get its own modules and multiplexing to perform the operations assigned to it. Thus, in the DAA the module selection and partitioning of the design are based on the clustering algorithm, while all other allocation decisions are taken by the expert system.

In the Cathedral-II system [Rabaey88], the main allocation tasks are performed by the rule-based Jack-the-mapper [Goosse88]. In Jack-the-mapper addressing schemes for arrays (memories) are generated and a bus structure is defined with a maximal number of busses to avoid conflicts. Examples of rules in the rule base are: - rules how to generate addressing schemes for background memory or - rules how to expand multiplications and divisions on an ALU in a sequence of shifts and additions or - rules how to generate loop counters when iterations are implemented. The module selection and module allocation can be influenced by the designer by writing so-called pragmas in the input language which prescribe the selection of certain types of modules (called EXUs) or the allocation of certain operations. The final allocation of modules and registers is done by the Atomics tool during the microcode scheduling, which takes place after the main allocation phase. Later an iterative Busmerger is used, which merges the excess of busses generated by Jack-the-mapper. The busmerger uses the following iterative algorithm: Let B be the number of busses. The busmerger selects the N most intensively used busses and tries to merge the B - N busses into them. If this does not work it tries it again for N + 1. This is done for N = 1 to N equal to the number of busses - 1 or until a merging succeeds. To summarize, in Cathedral-II the selection of operation modules is done by hand while the selection of storage and interconnect structures is done rule-based. The allocation of operations can be partly prescribed by the designer and will be completed to-
4.2 Overview of Allocation Schemes

gather with the register allocation during the list-scheduling based micro-code scheduling.

4.2.2. Greedy / Iterative Schemes

The program EMUCS [Thomas'90] maps operations onto modules in a step by step fashion. In each step tables of cost which reflect the feasibility of allocating each operation onto each module are generated. For each unallocated operation, EMUCS calculates the difference of the two lowest binding costs. The operation with the highest difference is bound to the module with the lowest cost. This approach attempts to minimize the additional cost that would be incurred if that element would have been bound in a later step.

In the CHARM, previously called SAM, system [Woo90a], operations are iteratively merged into sets of compatible operations and these sets are merged further into larger sets. Two operations are compatible if a module is available in the library that can execute both operations. At each merge the cost of a separate allocation of both sets is compared to the cost of merging both sets. At each iteration of the algorithm, all possible merges are considered and the possible savings of all these merges calculated. The merge with the largest saving in cost is done. If no merges result in a saving in the cost, the algorithms stops and a module is allocated for each of the merged sets and one for the remaining (not merged) operation. During the cost evaluation the register and interconnect allocation is done. All registers already in the datapath are listed [Woo90b] and a selection which registers to use is made based on interconnect minimization, when operations sets are merged.

Another greedy algorithm is used by MABAL [Köpük'89], however some decisions can be reversed later. This makes a limited amount of backtracking possible. No difference is made between module and register allocation. All operations and values are treated as operations and allocated in the order in which they are scheduled. During the incremental allocation decisions, MABAL calculates the cost of the possible allocations for an operation and chooses the best allocation with respect to the partial design and the tentative interconnect, or adds a new module if that is cheaper. In a post-processing phase busses are merged.

An iteration similar to the one in MABAL was done in the ADA to standard cell compiler as presented in [Girczyk'84]. The allocation selects all operations in a cycle step and allocates hardware for them, one at a time. The operation assignments are formulated as graph production rules, and during the module allocation all possible production rules are evaluated and the cheapest one is chosen. If more than one module is added during the allocation of all operations in a single cycle step, a total relocation for all nodes allocated thus far is done.

All three iterative schemes maintain and update a cost function in which they reflect the cost of potential bindings. The differences between the three are in the ways the iteration is done. In MABAL (and the ADA to standard cell compiler) the
iteration is done by visiting the operations in the scheduled order and binding each operation as it is visited. Thus, in each iteration all possible bindings for a single operation are considered. EMUCS, on the other hand, considers the binding of all operations to modules (even the bindings to empty modules which represent the addition of a new module) in each iteration, and binds the most promising operation. CHARM iteratively merges the operations into larger sets, where each set is bound to the same module.

4.2.3. Branch & Bound Schemes
Splicer [Pangrie88] uses a combined allocation scheme which does a dynamic allocation of registers and modules during the interconnect allocation step. The algorithm works cycle by cycle step, starting with the registers that contain the initial values. First, the algorithm connects input busses to the registers. An attempt is made to use busses that are already there. Then, the input busses are connected to functional units. A functional unit, that can perform the operation and is already connected to the bus is searched. This implicitly determines the module allocation. Then, output busses are added at the module its outputs and the output busses are connected to registers. Again it is tried to use a register that is already connected to the bus. Thus, also the register allocation is done implicitly. When all connections for a cycle step are made the algorithm proceeds to the next cycle step. The Splicer algorithm is based upon branch & bound search. An initial solution is obtained by a depth first search which is improved upon by backtracking. The search can be limited by passing only a subsection of the control/data flow graph to Splicer at a time.

In the Mimola system [Marwedel86] register allocation is done first, however only for straight line code. Module selection is done by an integer linear programming technique, which minimizes the overall functional module cost. Module allocation is done during interconnect allocation by a branch & bound algorithm. However, executing the full branch & bound algorithm becomes too complex in general, and therefore this is done one cycle step at a time. The algorithm starts the allocation with the cycle step in which the most operations are scheduled.

4.2.4. Linear programming
Hafer and Parker [Hafer83] used a linear program model for the whole data path synthesis but this was only applicable to very small design examples. Linear programming has been applied more successfully in the module selection phase. One example is the MIMOLA system discussed in the previous section, which uses an integer linear program formulation for module selection after scheduling. A recent update of the MIMOLA system [Balakrishnan89] describes also the register and module allocation as a zero–one integer programming model. This model is solved in each step of a list scheduling to find the optimal allocations for the operations in this step. The ADPS [Papachristou89][Papachristou90] also does the module selection during the scheduling by a linear programming.
4.2.5. Allocation Schemes based on Logic Synthesis

Two systems are reported which heavily depend on logic synthesis to finish the data path allocation. Typical for these systems is that they optimize the controller and the data path simultaneously. Furthermore, still some inefficiencies are allowed in the data path, because they will be removed by the logic synthesis. For example, if an adder can be allocated for an increment operation the logic synthesizer will propagate the constant and remove the unnecessary gates.

The Yorktown Silicon Compiler [Brayton88] allocates for each operation a separate functional module. During the operation folding modules that execute in different control steps will be merged when significant savings are obtained. Register folding, based on lifetime analysis, reduces the number of registers needed. The controller and the data path are combined together and prepared together for a logic synthesis system. Before running the logic synthesis the combined design is partitioned [Campos87], mainly because the logic synthesis cannot handle the very large circuits. Adding semantic information (which operations are similar with respect to their logic implementation) to the partitioning, which was initially based on area and interconnections only, improved the results significantly. The partitioned design is further optimized by the logic synthesis system.

In the Hercules system [DeMich88] the operations are also mapped directly to Boolean equations. First a maximal parallel implementation is made and only variables that are declared as architectural registers and variables in loops are implemented as registers. Logic synthesis is performed for the whole data path and controller at once. The results of the logic synthesis can be fed back to guide a next iteration in the synthesis process.

4.3. Global Allocation Schemes

Another category of data path allocation schemes performs a global allocation. Most of these algorithms are based on a clique covering of some sort of undirected graph $G(X,Y)$. Given a subset $A \subseteq X$ of the vertices, a subgraph induced by $A$ is defined to be $G(A,Y_A)$, where:

$$Y_A = \{ (x_i,x_j) \in Y \mid x_i \in A \land x_j \in A \}$$

A graph is complete if every pair of distinct nodes is adjacent, i.e. connected by an edge. A clique is a subset $A \subseteq X$ inducing a complete subgraph. A clique $A$ is maximum if there is no clique of $G$ which properly contains $A$. A clique cover of size $k$ is a partition of the nodes $X = A_1 \cup A_2 \cup \cdots \cup A_k$, such that each $A_i$ is a clique.

4.3.1. Tseng's Formulation

Tseng [Tseng86] was the first who applied the clique partitioning to the the data path allocation problem. His approach will be discussed here in more detail because most of the global allocation algorithms are variations of his approach.
The three allocation phases are done in the following order: first the register allocation
then the module allocation and finally the interconnect allocation. To do the reg-
ister allocation a graph $G_d(V_s, W)$ is defined, where :

$$ W = \{ (u, v) \mid \ll o(u), P(v) \gg \ll o(v), P(v) \gg = 0 \}, $$

where $o(v)$ and $P(v)$ are the write and last read cycle steps as defined in definition 3.10 and
where the operator $\ll$ returns $true$ if two intervals overlap and returns $false$
otherwise:

$$ \ll x_1, y_1 \gg \ll x_2, y_2 \gg = \begin{cases} false & \text{if } y_1 < x_2 \lor y_2 < x_1 \\ true & \text{otherwise} \end{cases} $$

This means that the storage values that are adjacent in $G_d(V_s, W)$ can be stored in
the same register without overwriting each others values. A clique covering of this
graph groups the values in such a way that all values that belong to the same clique
can be stored in the same register. The clique covering is done using a heuristic that
first combines those nodes having the most neighbors in common.

A similar technique can be applied to perform the module allocation. A graph
$G_d(V_s, Y)$ is defined where :

$$ Y = \{ (v, u) \mid (\Theta (v)) \cap (\Theta (u)) = \emptyset ) \lor (m(v) \land m(u) = \emptyset ) \lor \lambda (o(v)) \land \lambda (o(u)) = \emptyset \} $$

An edge is added between two nodes if the two operations they represent can be
combined in a single module, i.e. the operations do not have to be executed concurrently
or are mutually exclusive and there exists a module that can perform both oper-
ations.

The same clique partitioning as used in the register allocation can be used here
again. However, since the information about the register allocation is not yet available,
this can be used to try to reduce the number of interconnections. In the search for
cliques, priority is given to combine operations that have data coming from (or to be
stored in) the same registers. Therefore, each edge $y \in Y$ is given a weight as
follows (assuming two input, single output modules):

**Weight 4:** The two pairs of input values have the same origin and the output pair
has the same destination.

**Weight 3:** One pair of input values has the same origin and the output pair has the
same destination, or the two pairs of input values have the same origin but
the output pair has different destinations.

**Weight 2:** None of the pairs of input values has the same origin and the output pair
has the same destination, or one pair of input values has the same origin and
the output pair has a different destination.

**Weight 1:** All three pairs of values have a different origin or destination.

The edge weight is a measure for the number of common inputs and outputs two op-
erations have. Since the operation nodes are not yet allocated only the inputs and
outputs that have the same operation as origin or destination can be counted. For the storage values the actual allocation can be taken into account. The heuristic search for cliques, starts with the edges with the highest weight. When a clique cover is found, one module is allocated for each clique in the cover of \( G_d(V, E) \).

Interconnect allocation is done in the following way in [Tseng86]: A graph \( G_f(X, Z) \) is defined where a node \( x_i \in X \) is made for each edge \((v_i, u_i) \in E_f \). The elements \( x_i \) are called data transfers. The function \( \Theta : X \rightarrow S \) describes for each data transfer in which cycle step it takes place. This function can be derived from the cycle step in which the source of the data transfer produces its result. \( Z \) is defined as:

\[
Z = \{ (x_i, x_j) | \Theta (x_i) = \Theta (x_j) \}
\]

thus, two nodes in \( G_f(X, Z) \) are connected if their data transfers do not take place simultaneously.

For each edge \( z \in Z \) a weight can be defined which implies the similarity of this pair of data transfers. The weights of 2, 1 and 0 can be assigned if the data transfers respectively have the source and destination, the source or destination or none of them in common.

Again edges with higher weight are given preference in the search for a clique cover. This completes a description of the data path allocation scheme as was informally described in [Tseng86].

4.3.2. Left Edge Algorithm

In REAL [Kurth87] it is shown that the register allocation, as defined by Tseng, can be modeled as a channel routing problem. The lifetimes of values in a DFG can be represented by intervals. These intervals can be seen as wires which have to be assigned to tracks (the registers). The problem can then be solved by a left edge algorithm [Hash171].

In [Stoke88b] it is shown that the register allocation graph, as defined above, is the complement of an interval graph. Note that finding a clique covering in a graph \( G \) is a similar problem as finding a coloring in the complement graph \( \overline{G} \). (The complement of a graph \( G(V, E) \) is defined as the graph \( \overline{G}(V, \overline{E}) \) where \( \overline{E} = \{(x, y) \in V \times V | x \neq y \land (x, y) \notin E \} \). Based on the notion that the register allocation is a special case of coloring in interval graphs, it was shown in [Stoke88b] that the left-edge algorithm will only lead to optimal results when the data flow graph is free of loops.

4.3.3. The early EASY Allocation Scheme

This section describes the first data path allocation scheme that was implemented in EASY [Stoke88a]. In this scheme the module allocation is done first. The module allocation is defined as a maximal weight clique cover problem. This approach uses
a heuristic to find a maximal weight clique cover. The graph $G(V_o, Y)$ is defined as describe above, but the weights of the edges are calculated differently. For each combination of operation nodes the advantage is calculated when they are combined in a single module. The advantage is defined by the difference in cost for the combined implementation and the sum of the cost for the separate implementations. Thus the advantage of combining two operations $v_1$ and $v_2$ is:

$$Advantage(v_1, v_2) = Cost(v_1) + Cost(v_2) - Cost(v_1 + v_2 - max \_ cost(v_1 + v_2))$$

If the types of $v_1$ and $v_2$ are the same both could be implemented by the same module and the equation reduces to the difference in cost for an additional module and the multiplexing cost. Multiplexers are only added to the input ports of a module. The number of inputs to a multiplexer connected to a port should be equal to the number of different origins of the values which are input to this port. The cost of such a multiplexer can be obtained from the library. The advantage is used as the weight in the graph $G(V_o, Y)$. Note that all edges have a positive weight, because edges with a negative weight are left out because combining these operations does not bring any advantage. These weights are updated during the search for a maximal weight clique cover, to reflect the merging of an operation in an already existing clique of operations.

The register allocation is done by a left edge algorithm on the graph $G(V, W)$. In the left edge algorithm one interval is randomly chosen when more values have the same definition time. The algorithm can however be extended to use the information available from the previous assignment of modules. Obviously, it is advantageous to combine values that are inputs or outputs of the same module in a single register. A connection set is defined as the set of values which are connected to a single port of a module. Thus, a module with two inputs and one output port has three connection sets attached to it. The number of connection sets in which $v_1$ and $v_2$ appear together is defined as $N_{ij}$.

The left edge algorithm is updated by adding the following:

If there are more values with the same definition time we choose one according to the advantage it delivers to the implementation. The advantage of the assignment of value $v_i$ to register $m_o$ is defined as:

$$Advantage(v_i, m_o) = \sum_{y(v_i) = m_o} N_{ij}$$

The advantage is calculated for each register $m_o$ to which the value may be assigned and for all values with the same definition time. The clique value pair with the highest advantage is selected and the value is assigned to this clique. This algorithm still delivers a minimal set of registers, but it decreases the amount of interconnect needed to connect all registers to the operation modules.
4.3 Global Allocation Schemes

In [Stok88a] also the register merging is introduced. This last step in the allocation scheme can be seen as an optimization step. A valid design is already obtained but an attempt is made to reduce the number of busses by merging registers into register files. A register file is a set of registers such that not more than one register can be read and one alternate register can be written each cycle. Thus merging is possible if two registers are not written or read in the same cycle. Merging of two registers is only advantageous if the values written to the registers have at least once the same origin during the execution of the algorithm (or the values read have the same destination). The register preference graph $G_{\text{reg}}(M, Z)$ is built. Two registers are connected by an edge $z \in Z$, if they may be combined in a register file. To each edge a weight is attached which is defined as:

"the number of times during the execution of the algorithm that the values to be stored in the registers have a common origin or destination."

The algorithm to determine a maximal weight clique cover can be used here again. All registers which will be collected in the same clique are merged into a single register file. Especially when a large number of registers are not read or written each cycle step but are used to store the values for a longer time, the merging of the registers can result in a substantial saving of the area needed for interconnections.

4.3.4. The CADDY Allocation Scheme

The order of the allocations in the scheme of CADDY [Kraemer90] is similar to Tseeng's approach. After scheduling, the design is partitioned manually into several processors and into several pipeline stages. Then the actual data path construction starts. Two graphs are build for the register allocation phase, a restriction graph and a preference graph. The restriction graph is the complement of $G(V_r, W)$. The second graph represents the preferences of combining several values into the same registers. The edges are weighted based on similarity in interconnections. Since the complement of the graph is defined, a coloring of the graph will result in a register allocation, when a register is created for each color. The minimum number of colors needed is estimated by coloring the first graph using a general node coloring heuristic based on the sorting of the nodes according to their degree. Then, the preferences are taken into account in a second coloring process. If the number of colors needed in the second phase is close to the estimated minimum the coloring is accepted else the edges with minimum weight are removed from the preference graph and the process is started again. The allocation of operations and busses are also described in terms of coloring restriction and preference graphs.

4.3.5. The HAL Allocation Scheme

In HAL [Paulin89b] module allocation is done first, using a functional partitioning method. Then, register allocation is done by a weighted clique partitioning on a graph $G(V_r, W)$. The weights are not based on a count of individual interconnec-
tions, but are derived from interconnection patterns. Four different interconnection patterns are distinguished, which divide the edges of \( G(V, W) \) into four groups. Clique partitioning starts with the group of the pattern that represents the highest sharing of interconnections. Paulin remarks that, since the clique search is done groupwise and the groups are usually small, exhaustive search can be used. The sharing of multiplexers in the interconnect allocation phase is modeled similarly, but the groups are formed depending on the number of multiplexer inputs that are the same.

### 4.3.6. An Allocation Scheme based on Bipartite Matching

A data path allocation scheme based on bipartite matching is described in [Huang90b]. This algorithm does the same as the register allocation described in [Stok88a], in the sense that all possible register assignments are considered during a step of the left edge algorithm instead of choosing the first free register. At each step during the left edge algorithm, the register allocation is formulated as a weighted bipartite matching problem on a graph \( G_{bp}(V \cup M, E) \). There is an edge \( e \in E \) between a value \( v \) and a register \( m \) if no value has been assigned to \( m \) which lifetime conflicts with \( v \). Weights are obtained from an estimate of similar sources and destinations since the allocation of modules has not been done yet.

The module selection and module assignment are done before the module allocation. Then, the module allocation can be formulated as a weighted bipartite matching on a graph \( G_{bp}(V \cup M, E) \). This matching is performed for each control step at a time. The weights can be obtained from the allocation of the registers.

The interconnect allocation iteratively adds multiplexers to register outputs, starting from the register outputs that are most heavily used, and trying to merge subsequent multiplexers into them. A similar process adds multiplexers in between the operational modules and the registers.

### 4.3.7. Allocation of Operation Sequences

Another approach which treats the wiring effect as a first order effect during the data path synthesis is described in [Park89]. Instead of trying to combine single operations it is attempted to map sequences of operations with the same interconnection pattern to a group of modules. In this way various sequences can share the same data interconnections. All sequences of a certain length (typically two and three) are searched for and a graph is established whose nodes represent the sequences. Two nodes are connected by an edge if the sequences can be implemented on the same group of modules. A clique partitioning of this graph gives an allocation of the paths to groups of modules. This technique is applied to pipelined designs where advantage can be taken of the fact that modules can be shared at most as many times as the latency of the design. Therefore a heuristic partitioning algorithm, that
searches for cliques of size up to the latency can be used. However, paths in some cliques may be in conflict with paths in other cliques. The selection, deciding about which cliques to implement, is not described in their paper. The module allocation also determines the register allocation in the sequences. The remaining registers are allocated as in [Kurdah97]. The data path is completed by allocating the remaining interconnections by hand.

4.4. Discussion of the Allocation Schemes

In this section some advantages and disadvantages of the various allocation schemes will be discussed.

A major drawback of the rule based systems is that they are usually very slow and that it is difficult to add rules, that do not disturb the already present ruleset. However, results from Jack-the-mapper have shown when the design space is narrowed by giving the appropriate pragmas and the application domain is restricted, more domain specific knowledge can be used and good results can be obtained.

A problem with the iterative algorithms is that the solutions heavily depend on the order in which the decisions are taken. A lot of 'global' analysis is needed to pick the right order in which the allocations are fixed.

The branch & bound algorithms are usually too expensive to be applicable for larger designs. Of course, heuristics can be incorporated which delimit the search space further, but this will degrade the quality of the solution.

Linear programming is also very expensive and cannot be used to solve the general allocation problem, except for very small designs. However, the subproblem of module selection has been solved adequately using a linear programming formulation.

The logic synthesis based algorithms are not applicable to data path dominated designs. The logic synthesis backends do not work very well for the arithmetic components in the data path. However, this may be a feasible approach for the synthesis of control dominated designs.

Most of the global allocation schemes have the advantage that subproblems can be formulated on a solid graph theoretical basis and that efficient algorithms can be found to solve these subproblems optimally. Several examples of such formulations have already been given in the previous sections. Two new approaches are discussed in the following chapters. Chapter 5 shows how to optimally solve the register allocation problem, in data flow graphs that contain loops. Chapter 6 gives a new formulation of the module allocation problem, which can be solved optimally in polynomial time.

However, the division into subproblems and the choice of the order in which to solve them is a crucial task. The architecture of the data path should play a key role in this decision. The scheme chosen should take care that expensive elements in the architecture are used optimally.
Chapter 5

Register Allocation in Cyclic Data Flow Graphs

It was shown in [Stok88b] that the left–edge algorithm used for register allocation will only lead to optimal results when the data flow graph is free of loops.

However, in many applications data flow graphs do contain loops. Especially in signal processing applications the DFGs contain at least one outer loop and mostly several inner loops as well. One straightforward way [Stok88a] to deal with the cyclic flow of data in the DFG is to break the cycles at the loop boundaries. Values whose lifetimes cross the loop boundary are split and treated as two separate values. In [Goosse89b] a heuristic is used, which tries to fill the gaps between the two intervals resulting from the same value first. After that a normal left–edge procedure is used. A drawback of both these methods is, that if the two values are assigned to different registers, an additional register transfer operation is needed.

This paragraph discusses an algorithm for optimal register allocation in the presence of loops that eliminates superfluous register transfers. In section 5.1. the problem is reformulated as a multicommodity network flow problem. An efficient technique to solve such a flow problem is described in section 5.2. Section 5.3. gives some improvements to the basic register allocation algorithm. In section 5.4. some experiments are described and the results of applying this algorithm to a set of benchmark DFGs are presented.

The graph $G(V, W)$, which was defined in the section on register allocation in section 4.3., will be called a register allocation graph. Figure 5.1b) shows the register allocation graph derived from a scheduled data flow graph in figure 5.1a). An edge between two nodes in the register allocation graph means that the two corresponding storage values have non–overlapping lifetimes and can therefore share a register.

5.1. Cyclic Register Allocation

When loops occur in the DFG the lifetimes can not be represented anymore by intervals on a straight line. In this situation the life times can be represented by arcs
around a circle. This circle will be called the lifetime circle. The family of arcs $A$ contains an arc for each storage value from the DFG. Figure 5.2 shows the family of arcs for the data flow graph of figure 5.1. A graph $G(A)$ can be associated with the family of arcs, having a node for each arc and having an edge between two nodes when the corresponding arcs overlap. The graph $G(A)$ is called a circular arc graph [Tucker71]. The graph $G(A)$ is the complement of the register allocation graph as defined in the previous section. Finding a coloring of the graph $G(A)$ will give an assignment of values to registers where each color corresponds to a register.

The basic algorithm for cyclic register allocation is shown below. The problem is represented as a problem of producing a $q$-coloring of the family of arcs $A$, which in turn is converted into a multicommodity flow problem. The following paragraphs describe the steps of the algorithm in more detail.

To explain the various phases of the algorithm the following definitions are used. The boundaries between the cycle steps will be called cycle step transitions or short transitions. A transition has the same index as the cycle step it precedes. The transition between the cycle steps $s_{i-1}$ and $s_i$ ($s_{i-1}, s_i \in S$) will be called $t_i$. For each transition $t_i$, $1 \leq i < n$ an overlap set $O_i$ is defined, which contains all arcs that pass $t_i$. Note that two arcs overlap if and only if they both are simultaneously element in the same overlap set $O_i$.
5.1 Cyclic Register Allocation

![Diagram of a cyclic register allocation]

**Figure 5.2**: Lifetime Circle

### 5.1.1. Arc Splitting

The first step in the algorithm forms a new family of arcs $A'$ from $A$ by arc splitting. A new transition $t_n$ is created. All arcs that cross the initial cycle step transition of the loop, $t_1$, are split. Let $a_i$ remain the name of the arc that crosses $t_1$, and let $a'_i$ be the name of the new arc that crosses $t_n$. The arcs $a'_1$ and $a_1$ are called companion arcs of each other. The overlap set $O_1$ contains all arcs $a_i$, while the new overlap set $O_n$ contains all $a'_i$. In Figure 5.2 the arc splitting creates a new transition $t_3$. In this situation $O_1 = \{ u, x, y \}$ and $O_3 = \{ u', x', y' \}$.

### 5.1.2. Initial Coloring

An initial coloring of the family of arcs $A'$ can be made with $q$ colors where:

$$ q = \max \{ |O_i| \} $$

To obtain such an initial coloring first for any $k, 1 \leq k \leq |O_i|$ we assign color $k$ to arc $a_k \in O_i$. Next, a color, that does not conflict with the previously colored arcs, is assigned to each arc $a_i \in O_i, 1 < i \leq n$, that has not yet been colored. This is essentially a left edge algorithm.

Figure 5.3 shows an initial coloring for the example of figure 5.2. The arcs for the value $u$, $x$, and $y$ are split at transition $t_1$, called the boundary. Five registers (colors) are needed. The initial coloring implies three register transfer operations: one from register 1 to register 3 for $y'$ and $y$, one from 2 to 1 for $u'$ and $u$, and one from 5 to 2 for $x'$ and $x$. In the following steps this initial coloring will be changed such that the register transfers are eliminated.
5.1.3. Boundary Condition

The following condition with respect to the initial coloring is defined at the boundary:

\[ \forall a_i \in O_i : C(a_i) = C(a'_i) \]

i.e. for each arc \( a'_i \) in \( O_a \) that is split, \( a'_i \) should obtain the same color as the companion arc \( a_i \) in \( O_i \).

Some arcs will be excluded from the boundary condition. These are the arcs that correspond to a value that always generates a register transfer. Such a situation occurs when a new instantiation of a value is created before the last consumption of the previous instantiation. An example is given in figure 5.4 where the value \( x \) always needs a transfer.

![Diagram](image)

**Figure 5.4**: DFG that always generates a register transfer.

Our goal is to find a recoloring of all arcs in \( A' \) such that the boundary condition is fulfilled. For each transition \( i, 1 < i < n \), a free color set \( F_i \) is defined by the set
of colors not used by any arcs in \( O_1 \cap O_{k-1} \). Initially all colors are assumed to be free, i.e., \( F_1 = \{1, \ldots, q\} \). The colors in \( F_1 \) are exactly those colors which can be used to recolor an arc that starts at \( i \). If the color of an arc \( a \in O_i \) is changed from color \( j \) to color \( j' \), then in the overlap sets \( O_i \), \( i < i < n \), some of the arcs with color \( j \) have to be recolored with color \( j' \) and some arcs with color \( j' \) have to be recolored with color \( j \). Note that, if the colors of all arcs with colors \( j \) and \( j' \) are exchanged in all \( O_i \), \( i < i < n \) always a legal coloring is obtained. A sequence of color permutations has to be found such that the arcs in \( O_n \) are assigned the same colors as their companion arcs. A useful way to represent all the possible permutations is by a flow network.

### 5.1.4. Flow Network Construction

Consider a network \( N = (U, E, C, K) \) consisting of:

- A finite set of vertices \( U \);
- A collection of directed edges \( E \);
- A cost function \( C : E \rightarrow N \) from the set of edges into non-negative integers;
- A capacity function \( K : E \rightarrow N \) from the set of edges into non-negative integers;

The recoloring problem can be transformed to a multicommodity network flow problem as follows. A vertex \( u_{i,k} \), \( 1 \leq i \leq n \), \( 1 \leq j \leq q \) is made for each color \( j \) for all cycle step transitions. The vertices \( u_{i,k} \), \( 1 \leq k \leq q \), are called the sources of the network. The vertex \( u_{i,k} \) is the source of commodity \( k \). The vertices \( u_{k,j} \), \( 1 \leq j \leq q \), are called the sinks of the network. Vertex \( u_{k,j} \) is the sink of commodity \( k \), if arc \( a_i \) has color \( j \) and the companion arc \( a_i \) has color \( k \). For each transition \( 1 \leq i < n \) and every color \( 1 \leq j \leq q \), there is an edge from \( u_{i,j} \) to \( u_{i,j} \). For each \( f \in F_1 \), \( 1 \leq f \leq n \), there are edges from \( u_{i,f} \) to \( u_{j} \) for all \( j \in F_1 \). In our network the costs of all arcs are one. Also each edge has a capacity of one.

Figure 5.5 shows the network derived from the initial register assignment in figure 5.3 following the rules defined above. It can be seen that \( u_{i,j} \) is the source of commodity \( j \). Since \( u_j \) has color 2 in the initial coloring, vertex \( u_{i,2} \) is the sink of commodity 1.

A flow of commodity \( k \) from its source \( u_{i,k} \) to its sink \( u_{k,j} \) gives a recoloring scheme such that arc \( a'_i \) obtains the same color as arc \( a_i \). If commodity \( k \) flows through a vertex \( u_{i,j} \), this means that at transition \( i_j \) color \( k \) replaces the role of color \( j \). The arcs in \( O_i \) with color \( j \) will now get color \( k \). A simultaneous flow for all commodities, \( 1 \leq k \leq q \), in which no two flows share a vertex, gives a recoloring scheme such that the boundary condition is fulfilled.
5.1.5. Arc Recoloring

The flows, resulting from a solution of the multicommodity flow problem, can be used to recolor all the arcs. The result of applying the flows found in figure 5.5 to the initial coloring obtained earlier is shown in figure 5.6. When we follow the flow of commodity 1 (\(u_{i,1}, v_{i,1}, w_{i,1}, u_{i,2}, v_{i,2}\)), we see that at transitions \(t_4\) and \(t_5\) the values \(v_6\) and \(u'\) are moved from register 2 to register 1, which fulfills the boundary condition for \(u\).

![Figure 5.6: Final Register Assignment without Register Transfers.](image)

It is possible that the multicommodity flow problem has no solution using only \(q\) colors. If no \(q\)-coloring can be produced, but a \(q+1\)-coloring is possible, this \(q+1\)-coloring is clearly a minimal one. This leads to the following algorithm for a transfer-free register allocation:
Algorithm 5.1: Cyclic register allocation

Split arcs at loop boundary;
Obtain initial coloring (in \( q \) colors) by a left edge algorithm;
Formulate boundary condition;
while (boundary condition not fulfilled) do
  construct flow network;
  solve multicommodity flow problem;
  \( q = q + 1 \);
endwhile
recolor arcs from the flows;

The \( q + 1 \)–coloring will need one additional register to eliminate all register transfers. However, we will see in the result section that the main loop of the program is usually traversed only once and an immediate \( q \)–coloring is obtained.

5.2. The Multicommodity Flow Problem

The structure of the flow network derived in the previous paragraph can be described by an \( l \times m \) matrix \( A \), defined as follows:

\[
A_{ij} = \begin{cases} 
+1 & \text{if node } i \text{ is the source of edge } j \\
-1 & \text{if node } i \text{ is the destination of edge } j \\
0 & \text{otherwise}
\end{cases}
\]

where, \( l \) is the number of nodes in the network (which equals \( n \times q \)) and \( m \) is the number of edges in the network. The matrix \( A \) is called a node–arc incidence matrix. The decision variable \( x_{ij} \) denotes the amount of flow of commodity \( k \) through edge \( j \) and the \( m \)–component vector of flow \( k \) is denoted by \( x^{k} \). The unit cost of flow through all edges is one in our network. The requirement of flow of commodity \( k \) at node \( i \) is denoted as \( r_{i}^{k} \). The requirements are defined as follows:

\[
r_{i}^{k} = \begin{cases} 
+1 & \text{if node } i \text{ is the source of flow } k \\
-1 & \text{if node } i \text{ is the destination of flow } k \\
0 & \text{otherwise}
\end{cases}
\]

Mathematically, the multicommodity network flow problem may now be stated as follows:

\[
\min \sum_{i=1}^{l} \sum_{k=1}^{q} x_{ij}^{k} \tag{1}
\]

subject to

\[
Ax^{k} = r^{k}, \quad k = 1, \ldots, q \tag{2.a}
\]
\[ \sum_{i=1}^{m} \sum_{j \in \text{out}(n_i)} x_{ij} = 1, \quad j = 1, \ldots, l \quad (2. b) \]

\[ 0 \leq x_{ij} \leq 1, \quad k = 1, \ldots, q, \quad i = 1, \ldots, m \quad (2. c) \]

Condition (2.a) assures a unit flow from each source to each destination. Condition (2.b) specifies that at most one unit of flow passes a node \( n_i \), by limiting the sum of the flow of all from \( n_i \) outgoing edges to one.

The multicommodity flow problem can be solved by a primal algorithm that exploits the network structure [Kemeny80]. The algorithm uses the Dantzig-Wolfe decomposition. This technique decomposes the problem into a master program and \( q \) minimal cost network flow problems. The master program is solved by the primal simplex method. The minimal cost network flow problems [Hu82] can be solved by a shortest path method.

Applying these techniques to the problem of figure 5.5, resulted in the flows indicated by the solid lines in the figure. These flows were used to construct the coloring which satisfied the boundary conditions.

5.3. Improvement to the Basic Algorithm.

If no arc in the family \( A \) contains another arc, \( A \) is called a proper family of arcs, and \( G(A) \) is a proper circular arc graph. In [Tucker74] some special properties of circular arc graphs can be found. It is shown in [Orlin81] that a coloring of this type of graph can be found in \( O(n^2) \). Thus before the recoloring procedure is started it can be checked if the graph is of the proper circular arc type. In this case a more efficient algorithm can be used.

5.4. Experiments and Results

The first experiments were done using the wave digital elliptic filter example from [Dewilde85]. Since all operation were assumed to have unit delay the filter was scheduled in 14 cycle steps. Figure 5.7a shows the initial register allocation. The values with the name \( T_2 \), T13, T18, T26, T33, T38 and T39 are used in the following iteration. These values are used to formulate the boundary condition. The final register allocation is plotted in figure 5.7b). The 11 registers in the initial coloring did not have to be expanded to eliminate all register transfers. All split values end up in the same registers.

Similar experiments were carried out on the following set of benchmark graphs:
- \textit{diffeq}, differential equation from [Paulin87]
- \textit{udwf}, fifth order wave digital filter from [Dewilde85]
- \textit{tempcon}, temperature controller from [Girczyo84]
- \textit{frct}, fast discrete cosine transform from [Denyer90]
- \textit{cwdf}, circular wave digital filter from [Haroun88]
Figure 5.7: a) Initial and b) Final Register Assignment for Wave Digital Filter Example.

real, first example from [Kurdahi87]
pipe, pipelining example from [Devada89]

The results are collected in Table 5.1. The table shows the number of cycle steps in which the design was scheduled, the number of storage values in the DFG, the
number of values involved in the boundary condition, the number of registers needed in the initial and final allocation and the number of register transfer operations in the final allocation. The run times an Apollo DN2500 workstation varied between 0.5 second for the smaller examples and 30 seconds for \textit{fdeft}. In all the test cases no additional registers were needed and the multicommodity flow immediately generated a solution. All register transfers could be eliminated except in the \textit{cudef}. The various schedules for this data flow graph did contain situations similar to the one depicted in figure 5.4 and did force some register transfers.

Table 5.1. Cyclic Register Allocation Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># cycles</th>
<th># storage values</th>
<th># boundary values</th>
<th># initial registers</th>
<th># final registers</th>
<th># register transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff-eq</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>wdf</td>
<td>14</td>
<td>15</td>
<td>7</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>wdf</td>
<td>16</td>
<td>42</td>
<td>7</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>tempcon</td>
<td>3</td>
<td>15</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>tempcon</td>
<td>4</td>
<td>15</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>syncon</td>
<td>5</td>
<td>15</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>real</td>
<td>11</td>
<td>14</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>pipe</td>
<td>5</td>
<td>26</td>
<td>4</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>pipe</td>
<td>7</td>
<td>28</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>pipe</td>
<td>16</td>
<td>28</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>fdeft</td>
<td>6</td>
<td>50</td>
<td>7</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>fact</td>
<td>10</td>
<td>50</td>
<td>7</td>
<td>13</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>fact</td>
<td>14</td>
<td>50</td>
<td>7</td>
<td>14</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>cudef</td>
<td>8</td>
<td>31</td>
<td>8</td>
<td>13</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>cudef</td>
<td>10</td>
<td>31</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>cudef</td>
<td>12</td>
<td>31</td>
<td>8</td>
<td>12</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>cudef</td>
<td>16</td>
<td>31</td>
<td>8</td>
<td>13</td>
<td>13</td>
<td>4</td>
</tr>
</tbody>
</table>

5.5. Conclusions

An algorithm for register allocation in data flow graphs with loops is presented. The algorithm generates an initial register assignment using the minimum number of registers. This initial assignment is improved to eliminate all register transfer operations. Experiments with the algorithm have shown that in most cases no additional registers were introduced during the elimination of the register transfers.
Chapter 6

An Exact Algorithm for Module Allocation

This chapter describes an exact algorithm for module allocation. As has been shown in section 4.3., a graph $G_d(V_o, T)$ can be directly derived from a scheduled data flow graph. The graph $G_d(V_o, T)$ will be called a module allocation graph (MAG). In figure 6.1 a data flow graph and its corresponding module allocation graph are displayed. If the schedule does not contain any operations that are scheduled over multiple cycles, this will be called a simple schedule. A DFG that contains no conditional branches will be called an unconditional DFG.

![Diagram](image)

**Figure 6.1**: a) Scheduled Data Flow Graph b) Module Allocation Graph.

Under certain conditions, module allocation graphs are shown to be comparability graphs [Golumbic80]. A clique covering of such graphs can be found in polynomial time. Section 6.1 gives some definitions needed later in this chapter. Section 6.2 shows how the comparability property of module allocation graphs relates to the
components, available in the library, and the presence of mutual exclusion in the
DFG. When interconnection weights are taken into account, the problem can be
transformed into the maximum cost network flow problem described in section 6.3.

6.1. Definitions and Notations

**Definition 6.1:** An undirected graph \( G(V, E) \) is called a comparability graph if each
eedge can be assigned a one way direction, in such a way that the resulting
oriented graph \( G(V, T) \) satisfies the following condition:
\[
\forall v, v' \in V \quad (v \leq v') \implies (v, v') \in E
\]
Comparability graphs are often called *transitively orientable* graphs. A relation is
called a *partial ordering* of a set \( A \), when it is reflexive, anti-symmetric and transitive.
The relation \( T \) is a partial ordering defined on the set \( V \), called the *transitive orientation* of \( G \).

Let \( H_0 \) be a graph with \( n \) vertices \( v_1, v_2, \ldots, v_n \) and let \( H_1, H_2, \ldots, H_n \) be \( n \) disjoint
graphs. The *composition graph* \( H = H_0 \cup \{ H_1, H_2, \ldots, H_n \} \) is formed as follows: For all
\( 1 \leq i, j \leq n \), replace vertex \( v_i \) in \( H_0 \) with the graph \( H_i \) and make each vertex of \( H_i \)
dependent to each vertex of \( H_j \), whenever \( v_i \) is adjacent to \( v_j \) in \( H_0 \).

**Definition 6.2:** Given \( H_0 \), the composition graph \( H = H_0 \cup \{ H_1, H_2, \ldots, H_n \} \) of a set
of graphs \( H_i = (V_i, E_i), (0 \leq i \leq n) \) is a graph \( H(V, E) \), where:
\[
V = \bigcup_{i=1}^{n} V_i \quad \text{and} \quad E = \bigcup_{i=1}^{n} E_i \cup \{(v, y) \mid x \in V_i \land y \in V_k \land (v_j, y) \in E_k\}
\]

**Theorem 6.1:** ([Golumbic80]) Let \( H = H_0 \cup \{ H_1, H_2, \ldots, H_n \} \), where the \( H_i \) are disjoint
undirected graphs. Then \( H \) is a comparability graph if and only if each \( H_i \)
\( (0 \leq i \leq n) \) is a comparability graph.

**Theorem 6.2:** A complete graph is a comparability graph.

**Proof:** Let \( K_1 \) be a graph which consists of a single vertex. Since \( K_1 \) has no
edges, this graph is a comparability graph. Let \( K_2 \) be a graph which consists of two
adjacent vertices. The edge can be given any direction to form a transitive orientation.
Therefore, \( K_2 \) is a comparability graph. Every complete graph can be con-
structed by establishing composition graphs from \( K_1 \) and \( K_2 \). Following theorem
6.1 all complete graphs are comparability graphs. \[ \]

**Theorem 6.3:** ([Golumbic80]) Every comparability graph is a perfect graph.

**Theorem 6.4:** ([Golumbic80]) A coloring for a comparability graph \( G(V, E) \) can be
found in \( O(d\epsilon + n) \) steps, where \( \epsilon \) is the number of edges, \( n \) is the number of
vertices and \( \delta = \max \{ \deg(v) \mid v \in V \} \).
6.2. Module Allocation Graphs

In this section the influence of the library components and the influence of the conditional constructs in the DFG on the module allocation graph will be studied.

6.2.1. Module Allocation Graphs and Libraries

Let \( A \) be any set. The set inclusion relation \( \subset \) on \( \Pi(A) \) is a partial ordering of \( \Pi(A) \), since: (i) \( X \subset X \), for all \( X \in \Pi(A) \), (ii) if \( X \subset Y \) and \( Y \subset X \), then \( X = Y \), and (iii) if \( X \subset Y \) and \( Y \subset Z \), then \( X \subset Z \). If \( X \subset Y \), \( Y \) is superior or equal to \( X \). An element \( m \) is a maximal element of a set \( A \), if there does not exist any element in \( A \) which is strictly superior to \( m \).

Given a library \( \Lambda(O, L) \) as defined in definition 3.3:

**Definition 6.3:** The operation type set function \( \Omega : L \rightarrow \Pi(O) \), gives for each library component \( l \in L \), which operation types it can execute. The sets \( \Omega(l), l \in L \) will be called the operation type sets of the library \( \Lambda(O, L) \).

**Definition 6.4:** A library \( \Lambda(O, L) \) is called an ordered library, if the operation type sets of \( \Lambda(O, L) \) form a partial ordering by the set inclusion relation, where all maximal sets are disjoint.

An ordered set can be depicted conveniently by a Hasse diagram, in which each element is represented by a point, so placed that if \( X \subset Y \), the point representing \( X \) lies below the point representing \( Y \). Lines are drawn to connect sets \( X \) and \( Y \) such that \( Y \) covers \( X \), i.e., \( X \subset Y \) and there is no set \( Z \) such that \( X \subset Z \subset Y \).

Figure 6.2a) shows a Hasse diagram for an ordered library which contains 9 modules. The library has two disjoint maximal operation type sets; \(+, -, *, /, \&\&\), and \(|, \|, .|\).

**Definition 6.5:** A library is called complete if the library is ordered and has only one maximal set.

If the library of figure 6.2a) contains also a component with operation type set \(+, -, *, /, \&\&\), the complete library as depicted in figure 6.2b) is obtained. Note that the library in [Tse86], contains an ALU, which can execute all operation types, and is a complete library.

A library is called ordered with respect to a DFG, if all the operation types in the DFG are an element of one of the maximal operation type sets. Similarly, a library is called complete with respect to a DFG, if the maximal operation type set contains all operation types in the DFG. In the remainder of this chapter we will use the terms complete and ordered libraries always with respect to a DFG.

First module allocation graphs, which result from complete libraries, are considered.
**Theorem 6.5**: A module allocation graph derived from an unconditional DFG with a simple schedule and a complete library is a comparability graph.

*Proof*: Assume that the scheduled data flow graph \( G \) consists of a cycle ι steps. For each cycle step ι, \( ι \) operations are scheduled. A module allocation graph can be constructed as follows. Construct a complete graph \( H_0 \) which consists of ι vertices. A series of disjoint graphs \( H_1, H_2, ..., H_ι \) are constructed where each \( H_i \) consists of ι unconnected vertices. Since a complete library is used, the composition graph \( H = H_0 \cup H_1, H_2, ..., H_ι \) is the module allocation graph for \( G \). Since the graph \( H_0 \) and the graphs \( H_1, H_2, ..., H_ι \) are comparability graphs, the module allocation graph is a comparability graph.

An illustration of the composition of a module allocation graph from a complete graph and several disconnected graphs is given in figure 6.3.

**Theorem 6.6**: A module allocation graph derived from an unconditional DFG with a simple schedule and an ordered library is a comparability graph.

*Proof*: The proof goes along the same lines as the proof of theorem 6.5. Since the library is ordered, the composition will result in a module allocation graph consisting of m disjoint comparability graphs, where m is the number of maximal operation type sets in the library of which elements are present in the data flow graph.

An illustration of a module allocation graph consisting of two disjoint subgraphs is given in figure 6.6a.)
6.2 Module Allocation Graphs

6.2.2. Module Allocation Graphs and Mutual Exclusion

In definition 2.9, a definition of the mutual exclusion of two operations is given. Since mutual exclusion operations never have to be executed simultaneously they can share a module. Edges can appear in the module allocation graph between nodes that are scheduled in the same cycle steps, since the nodes are mutual exclusive.

Theorem 6.7: A module allocation graph derived from a DFG with a simple schedule with the property that in each cycle step all operations are mutually exclusive and an ordered library, is a comparability graph.

Proof: The proof goes along the same lines as the proof of theorem 6.5. However the graphs $H_1, H_2, \ldots, H_n$ are complete graphs, when all operations scheduled in a cycle step are mutual exclusive. Since the complete graphs are comparability graphs the composition module allocation graph is a comparability graph.

The branches of a conditional program can be represented by a conditional tree as described in section 2.10.

Theorem 6.8: A module allocation graph derived from a DFG with a schedule with the property that when operations from a branch are scheduled in the same
cycle steps with operations from at most one of its successor branches, and an ordered library, is a comparability graph.

**Proof:** Assume a single branch $B$ and its successors $B_1$ and $B_2$. Let the operations in $B$ before the conditional be in set $A$ and after the conditional in set $C$. Since $A$ does not contain any conditionals the MAGs are a comparability graph. The same holds for $MAG_{B_1}$, $MAG_{B_2}$ and $MAG_C$. Since the operations of $B_1$ are unconditional with respect to the operations in $A$ and $C$, the combined $MAG_{A,B_1,C}$ will be a comparability graph. The operations from $B_2$ are not scheduled in the same cycle steps as $A$ and $C$, since at most one branch is allowed to overlap, which was $B_1$ in this case. The operations in $B_2$ are fully mutual exclusive from the operations in $B_1$ and not overlapping with the operations in $A$ and $C$. The nodes in $MAG_{B_2}$ will be connected to all nodes in $MAG_{A,B_1,C}$. The resulting composed graph (where $K_2$ is the complete graph with two nodes) $MAG_{A,B_1,C,B_2} = K_2 \cup MAG_{A,B_1,C} \cup MAG_{B_2}$ is a comparability graph. If the DFG contains more branches the proof can be applied recursively for $MAG_{A,B_1,C}$ and $MAG_{B_2}$. \[\square\]

If operations of a branch are scheduled in the same cycle steps as two or more of its successor branches non comparability MAGs may be the result. To study this situation the following definitions are needed.

**Definition 6.6** A graph is called *asteroidal* if it contains three distinct nodes $a_1, a_2, a_3$ and three paths $W_i, W_2, W_3$ such that for $i = 1, 2, 3$:

(i) $W_i$ connect the two nodes $a_i, a_k$. ($i, k \neq i$)

(ii) the path $W_i$ contains no nodes from the adjacency of $a_i$.

An example of an asteroidal graph is depicted in figure 8.5, graph I. The asteroidal triple in this graph is formed by nodes $a, b$ and $c$. There exist a path $a, b, c, d, e$ which does not contain the adjacency of $g$. Similar paths exist for $a, \ldots, c, d$ and for $c, \ldots, g$.

A graph $G$ is a comparability graph if the complement $\overline{G}$ of graph $G$ is not asteroidal [Lecker82]. Following theorem 4 in [Lecker82], a graph $G$ is a comparability graph if $\overline{G}$ does not contain one of the subgraphs I, I, II, IV, V. The simplest versions of the subgraphs are depicted in figure 6.5. The first two graphs (I and II) causing asteroidal triples are also recognized in [Springer90], and called *branch* and *skip* asteroidal triples. On the right side some schedules and mutual exclusion conditions are sketched which give rise to non comparability module allocation graphs.

In [Lecker82] an algorithm of order $3m^2 + O(m^2)$, where $m$ is the number of nodes in the module allocation graph, is described, which checks if the graph contains asteroidal triples. If a schedule is generated not conform theorem 6.8, and
6.3 Weighted Module Allocation

---

**Figure 6.5**: Graphs whose complement are no comparability graphs.

Could contain one of the situations sketched in figure 6.5, it has to be checked whether the module allocation graph is a comparability graph.

### 6.3. Weighted Module Allocation

The amount of interconnections needed to connect all modules can be reduced by combining operations that have the same inputs and/or outputs. In [Tseng86] the pairs of operations are classified into categories depending on the input and output connections. Here, the weight categories, applied to the edges of the module allocation graph are as defined in section 4.3. The weight of an edge from \( v_i \) to \( v_j \), \( v_i, v_j \in V_c \) is denoted as \( w(v_i, v_j) \).
The module allocation graph $G'_d(V_o, Y)$ in figure 6.6a) is constructed using a library with the maximal operations sets: $\{+, -, \&\}$ and $\{x, .\}$. The weights are obtained from the register assignment in figure 6.1 (thick lines labelled $a..e$). For each pair of operations the input and output registers are listed in the form: \text{operation} = (\text{input}1, \text{input}2 \rightarrow \text{output}).

Consider the weighted module allocation graph $G'_d(V_o, Y)$. To transform this graph to a network, all edges are given a direction according to a transitive orientation $G'_d(V_o, Y)$. Furthermore, a source node $a$, a sink node $z$, edges from the source node to all nodes, edges from all nodes to the sink node and an edge from the sink to the source node are added. The edge from the sink to the source is called the return edge. The weight of all these additional edges is zero.

Formally:

A network $N_G = (V_o, E_o, a, z, C, K)$ corresponding to the module allocation graph is constructed, where:

- $V_o = V_o \cup \{a, z\}$
- $E_o = T \cup \{(a, v), (v, z) | v \in V_o \cup \{1, 0\}\}$
- $w(a, v) = w(v, z) = 0$ for all $v \in V_o$, $w(z, a) = 0$

For each edge $e \in E_o$, a cost function $C : E_o \rightarrow N$ is defined, which assigns to each edge a non-negative integer. The cost will be equal to the weight of the edges.
6.3 Weighted Module Allocation

- $C((u, v)) = w(u, v)$ for all $(u, v) \in E_a$

For each edge $e \in E_a$, a capacity function $K : E_a \rightarrow \mathbb{N}$ is defined, which assigns to each edge a non-negative integer. The capacity of all edges is one, except for the return edge which has capacity $k$:

- $K((z, a)) = k$, $K((u, v)) = 1$ for all $(u, v) \in E_a \setminus \{(z, a)\}$

A flow in the network $N_G$ is a function $f : E_a \rightarrow \mathbb{N}$, which assigns to each edge a non-negative integer, such that for each $e \in E_a$, $0 \leq f(e) \leq K(e)$ and for any node $u \in V_a$,

$$\sum_{(u, v) \in E_a} f(u, v) - \sum_{(v, u) \in E_a} f(v, u) = 0.$$ The amount of flow in the return edge $(z, a)$ is denoted as $f^r = f((z, a))$. The total cost of the flow is:

$$\kappa(f) = \sum_{e \in E_a} C(e) \cdot f(e).$$

An example of a network, which is obtained from the module allocation graph in figure 6.6(a), is shown in figure 6.6(b).

**Theorem 6.9**: A flow $f : E_a \rightarrow \mathbb{N}$ in the network $N_a$ corresponds to a set of cliques $X_1, ..., X_k$ in $G'_a$ where $k = f^r$.

**Proof**: Consider a flow $f$ with $f^r = 1$. Since graph $G'_a$ is acyclic, the flow $f$ will follow a directed path $P = \{x, y_1, ..., y_{f^r}\}$ in $N_a$. The graph $G'_a$ is a comparability graph, and has a transitive orientation, according to definition 6.1. Thus, if the edges $\{y_i, y_{i+1}\}$ and $\{y_{i+1}, y_{i+2}\}$ exist, also the edge $\{y_i, y_{i+2}\}$ will exist. The path $P$ in $N_a$ corresponds to a clique $X$ in $G'_a$. Since the capacity of all edges is one, each unit of flow will follow a separate path. A flow with $f^r = k$, will follow $k$ different paths, $P_1, ..., P_k$, which correspond to the cliques $X_1, ..., X_k$ in $G'_a$.

The paths $P_1, ..., P_k$ will be edge disjoint, but do not necessarily go through different nodes. Thus, the sets $X_1, ..., X_k$ are not necessarily node disjoint. To enforce node disjoint paths, a node separation technique (Sarrail90) can be used. In the node separation, all nodes $v \in V_0$ are duplicated. The duplicate of a node $v$ is called $v'$. All edges outgoing from $v$, obtain the node $v'$ as their origin. The node $v$ and its duplicate are connected by an edge with capacity $K((v, v')) = 1$ and a cost $C((v, v')) = 0$.

This node separation results in a network $N'_G = (V'_a, E'_a, a, z, C', K')$, where:

- $V'_a = V_a \cup V'_0$
- there is a vertex $v' = f(v)$, $v' \in V'_0$ corresponding to each vertex $v \in V_0$
- $E'_a = T' \cup \{(v, z) \mid v \in V_a\} \cup \{(z, a)\} \cup \{(v, f(v)) \mid v \in V_a\}$
- $T' = \{(f(v), u) \mid (v, u) \in T\}$
- $C'((z, a)) = C((v, f(v))) = 0$ for all $v \in V_0$
- $C'((v', u)) = C((v, u))$ for all $[v', u] \in T' \cup \{(v, z) \mid v \in V_a\}$
- \( k'(\{z,a\}) = k \), \( k'(\{u,v\}) = 1 \) for all \( u \neq z \) and \( v \neq a \)

Since the capacity of \( K(\{v,v'\}) = 1 \) at most one unit of flow can go through the edge \([v,v']\). This leads to the following theorem.

**Theorem 6.10:** A flow \( f : E_{a} \rightarrow N_{a} \) in the network \( N_{a} \) corresponds to a set of node disjoint cliques \( X_{1}, \ldots, X_{k} \) in \( G'_{a} \) where \( k = F \).

The separated network, which is obtained from the network in figure 6.6a) is depicted in figure 6.7. The maximal cost network flow problem finds among all possible flows with amount \( k \) the one with maximum cost. In [Edmonds72] it is shown that this problem can be solved in \( O(km^{3}) \), where \( m = |V_{ij}| \times 2 + 2 \).

![Maximum cost network flow diagram](image)

**Figure 6.7:** Separated Flow Network

When the maximal cost network flow is solved for the network \( N'_{a} \), a solution to the module allocation problem is found, which takes into account the first order effects of the interconnection weights. These first order effect describe the result of combining each pair of operations. Taking into account the influence of this combination on the weights with a third operation, would require a dynamic update of the edge weights.

### 6.4. Conclusions

This chapter describes some constraints on the library and the schedule, for the module allocation graphs to be comparability graphs. An exact algorithm for module allocation on module allocation graphs that are comparability graphs is presented, which takes into account the first order effects of the interconnection weights. The algorithm has a complexity \( O(km^{3}) \), where \( k \) is the number of resulting modules and \( m = |V_{ij}| \times 2 + 2 \), with \(|V_{ij}|\) the number of operation nodes in the DFG.
Chapter

7

Interconnect Minimization during Data Path Synthesis

As McFarland stated [McFar67], the interconnections constitute a substantial part of the register transfer level design cost. Omitting these costs results in implementations that are far from optimal. The multiplexers and busses that have to be added have a large impact on both the area and the delay of the implementation. Several allocation schemes have appeared in the literature that realize the importance of the interconnections and try to minimize the amount of interconnect by taking into account the interconnect in their cost functions and weights. Some authors try to take interconnections into account by inserting appropriate penalties into the optimization strategy. However, when the interconnection costs become the primary objective of the minimization, the result may be even more efficient implementations. Therefore, the main emphasis throughout all allocation phases, as described in this chapter, is to minimize the cost of the required interconnections.

In the new data path allocation scheme presented here the interconnection allocation is split in a global and a local process. First the global interconnect allocation is done, which minimizes the number of global interconnections (busses) needed. Furthermore, the number of local connection links from the operation module to the various busses is minimized as well. Because the scheduler only determines the types of modules as well as the multiplicity of each type of module, the actual assignment of operations to modules can be delayed until the interconnect is allocated. This freedom in assigning operations is used to reduce the number of required interconnection links. The data path allocation scheme consists of the following sequence of steps: storage value grouping, register allocation, and module allocation. Each of these subproblems will be discussed in the following sections.

Section 7.1 gives a mathematical formulation and an algorithm to solve the general storage value grouping problem. Section 7.2 presents an algorithm which solves an instance of the storage grouping problem in which the read and write cycle steps are separated. Both parallel storage and serial (re-)writes of values that have to be read several times are considered. Section 7.4 describes the register allo-
tion. In section 7.5, the various register merging algorithms are compared using a set of experiments. The algorithm which minimizes the number of local connection links and multiplexer inputs is described in section 7.6. Finally, in section 7.7, the results of our data path allocation scheme are compared to the results of other systems.

7.1. Storage Value Grouping

In most previous approaches [Kurdahi87], [Tseng86], [Stok88a] values are mapped onto registers according to their lifetimes. In our approach, values are merged into register groups depending on their read and write times. A single register file with a single read/write port can be reserved for values that are never pairwise read or written simultaneously. The following definitions are needed to define an algorithm that merges values according to this condition.

**Definition 7.1:** For any pair \( u, v \in V \), the disjoint access relation \( \mathcal{R} \) is defined:

\[
\mathcal{R}(u, v) = \{ (u, v) \in \mathcal{G} : u \in \mathcal{G} \land v \in \mathcal{G} \land (u \neq v) \lor (u \cap v) = \emptyset \land (u \cap v) \notin \mathcal{G} \}
\]

Using this relation, groups of storage values can be defined.

**Definition 7.2:** A storage group \( \mathcal{G} \subseteq V \), is a set of storage values such that for any pair the disjoint access relation holds, i.e., \( \forall u, v \in \mathcal{G} \implies \mathcal{R}(u, v) \).

Thus all storage values in a group have disjoint read and write times. This means that they are never accessed simultaneously, and that they can share interconnections and ports needed to access these values.

**Definition 7.3:** The storage group function \( \gamma : V \rightarrow \mathcal{G} \), determines the group a storage value is assigned to.

The following definitions hold for a general graph \( G(V, E) \). In this section a different notation for the edges of a graph \( G(V, E) \) will be used. As will be shown, multi-graphs are created, thus each edge has to be labeled. Two nodes \( x \) and \( y \) connected by a directed edge with label \( l \) are written as \( e_{x,y,l} \).

The chromatic index \( \chi(G) \) defines the smallest number of colors needed to color the edges of \( G \) so that no two adjacent edges have the same color. Such a coloring is called a proper edge coloring of graph \( G \). The degree \( \delta(v) \) of a node \( v \) is the number of edges incident to \( v \). The degree of a graph \( G(V, E) \) denoted by \( \delta \) is defined as:

\[
\delta = \max_{v \in V} \delta(v)
\]

The multiplicity \( m(x, y) \) of a pair \( x, y \in V \) is defined to be the cardinality of the set of edges \( F \) where

\[
F(x, y) = \{ e \in E : (v_1, v_2) \in E \land (v_1 = x \land v_2 = y) \lor (v_1 = y \land v_2 = x) \}
\]

The multiplicity \( M \) of graph \( G(V, E) \) is:
Assume a colored graph \( G \) and assume \( SC \) to be the set of colors. The notation \( \text{color}(e) \) is used to denote color of an edge \( e \). For \( v \in V \) we define the set of colors incident to \( v \) by

\[
IC(v) = \{ \text{color}(e) \mid e \in E \land \{x,v\} \subseteq E \}
\]

That is the set of colors assigned to the edges incident to \( v \). The set of missing colors of \( v \) denoted by \( MC(v) \) is defined by: \( MC(v) = SC - IC(v) \). Given the node \( v \) and two colors \( A, B \) the Kmpesubgraph \( KS(A, B) \) is a connected component of graph \( G \) such that the vertex set is equal to the union of \( v \) and all other vertices that are reachable from \( v \) by paths in the graph only through edges alternately colored \( A \) and \( B \). The following notation for sequences are used: \( <> \) is defined as the empty sequence. The dot operator \( . \) concatenates two sequences. The elements of a sequence \( q \) are denoted as \( q_i \) with \( 0 \leq i \leq \text{length}(q) \).

![Figure 7.1](image)

**Figure 7.1** a) Data Flow Graph Section with read and write cycle steps for the storage values. b) State Graph corresponding to Data Flow Graph of a)

**Definition 7.4:** A state graph \( G(S,E) \) is defined as a directed graph where:

\[
E = \{ s_i \mid \{s_1, s_2\} \in E \land s_1 = s(v) \land s_2 \in g(v) \}
\]

Figure 7.1 shows a section of a DFG and its corresponding state graph. On the left a section of a data flow graph with storage values is shown. The numbers above (below) the storage values indicate the read (write) cycle steps. In the state graph a node is created for each cycle step and an edge is drawn for each storage value be-
between its write cycle step and its read cycle step. This graph has a degree $\delta$ of two and a multiplicity $M$ of one.

### 7.1.1. General Edge Coloring Algorithm

Vizing [Vizing64] gives the following bounds for the chromatic index of a graph $G$:

$$\delta \leq \chi'(G) \leq \delta + M$$

Therefore it will always be possible to assign all storage values to register files using at most $\delta + M$ register files.

---

**Algorithm 7.2: General Edge Coloring Algorithm**

```plaintext
procedure General_Edge_Coloring ( G(V,E) );
repeat
    let (v, w) ∈ E a not colored;
    mc_v := MC(v); mc_w := MC(w);
    if (mc_v ∩ mc_w = 0) then
        color[(v, w)] := element ( mc_v ∩ mc_w );
    else let C_0 ∈ mc_v; C_i ∈ mc_w; C := <>; V := <>; j := 1;
        while ((∃ v ∈ V ∧ color(e) = C_j))
            and (C_i ⊆ C) do
            let C := C ∪ C_i;
            V := V ∪ {v}; C := C ∩ C_i;
            j := j + 1;
        endwhile;
        if not((∃ v ∈ V ∧ color(e) = C_j)) then
            for i=1 to j do
                color[(v, w)] := C_i;
            else (C_i is a member of C)
                let v_0 ∈ V with color(e) = C_i;
                construct_kempe_subgraph ( K3_2(C_i, C_j) );
                v := end_of_chain ( K3_2(C_i, C_j) );
                exchange_colors ( K3_2(C_i, C_j) );
                if v ≠ v_0 then
                    for k=1 to j do
                        color[(v, w)] := C_i;
                else (v_0 ≠ v)
                    for i=k+1 to j do
                        color[(v, w)] := C_i;
                color[(v, w)] := C_i;
                for i=1 to j do
                    color[(v, w)] := C_i;
            endif
        endif
    endif
until all edges of G are colored end.
```
Theorem 7.1: Algorithm 7.2 finds an edge coloring of a graph $G$ using at most $\delta - M$ colors.

Proof: We prove this by induction on the number of edges. For one edge the theorem is trivially true. We suppose that all edges have been colored using at most $\delta - M$ colors except for the edge $e(v, v')$. Since there are $\delta - M$ colors available there is always at least one missing color at $v$ and one missing color at $v'$. If the same color is missing at both nodes we can color $e(v, v')$ with this color. We assume that color $C_0$ is missing from $v$ and present at $v'$, and that color $C_1$ is missing from $v'$ and present at $v$, and $C_0$ and $C_1$ are not equal. We construct a sequence of edges $e(v, v_1), e(v_1, v_2), e(v_2, v_3), \ldots$ and a sequence of colors $C_0, C_1, C_2, \ldots$ such that $C_i$ is missing at node $v_i$ and such that $e(v_i, v_{i+1})$ is colored $C_{i+1}$. The construction of the sequences stops when:

1. no edge $e(v_i, v_{i+1})$ which is colored $C_{i+1}$ can be found, or
2. when color $C_j$ is already a member of the sequence $C_0, \ldots, C_{j-1}$.

- In case 1) we obtain a proper coloring by coloring each edge $e(v_i, v_{i+1})$ for $i \leq j$ with color $C_j$. Now color $C_j$ is missing at both $v_i$ and $v_{i+1}$ and thus edge $e(v_i, v_{i+1})$ can be colored $C_{i+1}$.

- In case 2) there exists some $k < j - 1$ for which the edge $e(v_k, v_{k+1})$ has color $C_j$ that is equal to color $C_{i+1}$. Now we can construct the connected subgraph $K_{C_0, C_1, \ldots, C_j}$ which contains $v_i$. This is an alternating chain of colors $C_0$ and $C_1$, and one of its endpoints is $v_i$, because $C_0$ is not present at $v_i$. Let $v'_i$ be the
other endpoint of this chain. If \( v_i \) or \( v_j \) lies on this chain it is an endpoint. Now one of the following two situations arises:

**Case 1** \( v_i \neq v_j \)

We obtain a valid coloring by interchanging the colors in \( KS_{a/C_o,C_j} \) and recoloring each edge of \( \{v_i,n\} \) for \( 1 \leq i \leq 4 \) with color \( C_i \).

**Case 2** \( v_i = v_j \)

We obtain a valid coloring by interchanging the colors in \( KS_{a/C_o,C_j} \) and recoloring the edge \( \{v_i,n\} \) with color \( C_i \). Furthermore, each edge \( \{v_i,v_j\} \), \( 1 \leq i \neq j \leq 4 \), can be colored with \( C_i \).

This concludes the proof of theorem 7.1. \( \Box \)

### 7.2. Bipartite Edge Coloring

The general result obtained in the previous paragraph can be improved further. If a two-phase clocking scheme is used, data transfer from the register files to the module is executed in the first phase and a transfer from the operation module back to the registers takes place in the second phase. All cycle steps can be split in a separate write part and read part. All nodes in the state graph are duplicated and the problem can be reformulated as a bipartite edge coloring problem.

**Definition 7.5:** A bipartite state graph \( GS_{a} \) is defined as a directed graph

where: \( E = \{ e_{\{v_i,v_j\}} : v_i \in V, v_j \in V \} \)

An edge \( e_{\{v_i,v_j\}} \) is present between a node \( v_i \) and a node \( v_j \), if a value is written in cycle step \( i \) and read in cycle step \( j \). Figure 7.3 shows a section of a DFG and the corresponding bipartite state graph.

In this bipartite case the number of register files needed can be bound further using the following theorem [König16].

**Theorem 7.2:** If \( BG \) is a bipartite graph an edge coloring can be made for \( BG \) using at most \( \delta \) colors.

**Proof:** We prove this by induction on the number of edges. The theorem is trivially true for one edge. We suppose that all edges have been colored using \( \delta \) colors except for edge \( e_{\{v_i,v_j\}} \). Since \( \delta \) colors are available at least one color is missing from \( v_i \) (say \( C_i \)) and one is missing from \( v_j \) (say \( C_j \)). If these colors are the same we can color \( e_{\{v_i,v_j\}} \) with this color. Otherwise we construct the two-colored subgraph \( KS_{a/C_i,C_j} \) which contains \( v_i \). Now any path in \( KS_{a/C_i,C_j} \) from \( v_i \) to \( v_j \) must have a final edge colored \( C_i \), but \( C_i \) is missing from \( v_j \). Thus we can interchange the colors along \( KS_{a/C_i,C_j} \) so that \( C_i \) is absent from both \( v_i \) and \( v_j \) and we can color the edge \( e_{\{v_i,v_j\}} \) with \( C_i \). This concludes the proof of theorem 7.2. \( \Box \)

A **Bipartite Edge Coloring Algorithm** to color a bipartite graph can directly be derived from the proof of theorem 7.2.
Algorithm 7.3: Bipartite Edge Coloring Algorithm.

procedure Bipartite_Edge_Coloring (G(\mathcal{V}, \mathcal{E}) \cup S, S);
repeat
  let \( e, v \in \mathcal{E} \setminus e \) not colored;
  \( m_{c_1}, m_{c_2} := MC(v) \);
  if \( m_{c_1} = m_{c_2} = \emptyset \) then
    color \( \{e, v\} \) := element \( m_{c_1} \cap m_{c_2} \);
  else
    let \( C_1 \in m_{c_1} ; C_2 \in m_{c_2} \);
    construct_bipartite_subgraph \( K_{3,3}/C_1, C_2 \);
    exchange_colors \( K_{3,3}/C_1, C_2 \);
    color \( \{e, v\} \) := C_1;
  endif
until all edges of \( G \) are colored.
end.

7.3 Multiple Read Values

As can be seen from the definition of the read function \( \phi(v) \), a value can be read several times.

Definition 7.6: A storage value \( v \in V_f \) is called a multiple read value if \( \phi(v) > 1 \).
Multiple read values are values that are written once and read several times. The writes and reads of these values can be modelled in two ways.

Figure 7.4 Multiple reads: a) Part of Data Flow Graph. b) Parallel State Graph. c) Serial State Graph.

Figure 7.4 a) gives a part of a data flow graph in which value \( x \) is written by a node in cycle step 1. This value gives rise to three edges in the state graph because it is read three times, in the states \( r_2 \), \( r_3 \) and \( r_4 \). We denote these edges by indexing with the value they result from i.e. \( e_{1\{w_1, v_r\}} \). In figure 7.4 b) the write cycle step is connected to all its read cycle steps. This is called the parallel case. The serial approach simulates a rewrite of the value in the companion write cycle step of each (except for the last) read cycle step. In figure 7.4 c) the value is read for the first time in cycle step \( r_2 \) and rewritten in cycle step \( w_2 \).

The parallel reads imply the simultaneous presence of the value in multiple register files. The serial reads imply the introduction of register transfers. A register transfer requires a link between the two busses to which the register files are connected, yet no additional busses are needed. When the edges \( e_{1\{w_1, v_r\}} \) and \( e_{1\{w_2, v_r\}} \) (figure 7.4 c) obtain a different color this implies a register transfer in cycle step 2 from the register file represented by the color of \( e_{1\{w_1, v_r\}} \) to the register file represented by the color of \( e_{1\{w_2, v_r\}} \).

We define a cluster as the set of edges originating from the same value, i.e. edges with the same label. When the reads are connected in parallel a copy of the value has to be made for each read because the coloring algorithm is prohibited to use the same colors for edges from the same cluster. The edges in figure 7.4 b) will always be colored differently, because these edges are all incident to the same node \( v_r \). However in the serial model two edges belonging to the same cluster can still be colored with the same color, making a register transfer superfluous.
The **Augmenting Edge coloring** (algorithm 7.4) minimizes the number of register files and tries to minimize the number of register transfers. All edges are sorted according to their write time in the clusters. All clusters are sorted in decreasing size (the size of a cluster is the number of edges in that cluster) and placed in the cluster_list. The largest cluster is colored first. It is tried to color all edges of a cluster with the same color. When a cluster can not be colored with a single color the cluster is broken into two parts such that only a single register transfer is introduced. These new clusters are inserted in the appropriate place in the cluster_list. If a size of the cluster decreases below the threshold (typically 1) the edges are colored individually.

**Algorithm 7.4 : Augmenting Edge coloring Algorithm.**

```plaintext
procedure Augmenting_Edge_coloring (G(S, \cup S, E));
    sort(cluster_list);
    foreach cluster E cluster_list do
        if size(cluster) > threshold then
            foreach C E \[ C \in\] do
                if (all edges in cluster can be colored C) then
                    foreach e(xy) E cluster do
                        color(e(xy)) := C;
                else
                    ng1, ng2 := split_cluster(cluster);
                    remove_cluster(cluster, cluster_list);
                    insert_cluster(ng1, cluster_list);
                    insert_cluster(ng2, cluster_list);
                endif
            endforeach
        else (edges in cluster uncolored )
            foreach e(xy) E cluster do
                let mC1 := MCG(e); mC2 := MCG(e);
                if (mC1 \& mC2 = \[ \emptyset \] ) then
                    color(e(xy)) := element ( mC1 \& mC2 );
                else
                    let C, C \[ \in\] mC1; C \[ \in\] mC2;
                    construct_hampe_subgraph ( K3\[ \cup\]C, C);
                    exchange_colors( K3\[ \cup\]C, C );
                    color(e(xy)) := C;
                endif
            endforeach
        endif
    endforeach
end;
```
7.4. Register Allocation

The allocation of registers can be done with similar algorithms as the ones described in the previous chapter. The number of registers in a register file can easily be determined by a life time analysis of all values to be stored in this register file. As depicted in figure 7.5 the left edge algorithm can be used. The storage values (with their life times) which are assigned to this register file are shown on the left. A register file consisting of three registers is enough to store all the values in this case. If the data flow graphs contains loops the algorithm from chapter 5 can be used.

![Figure 7.5 Left Edge algorithm to allocate registers.](image)

7.5. Specific Experiments and Results.

The example chosen is the elliptic wave digital filter from [Dewilde85]. A number of different schedules have been made. For each of these schedules a large number of allocations have been run with different options. The results are discussed here to illustrate several properties of the algorithms described above.

In the first graph (figure 7.6) the bipartite coloring scheme is compared to the augmenting algorithm. The augmenting algorithm not only reduces the number of register transfers but also the number of registers. The register nodes having overlapping life times are scattered more effectively over the available register files, which makes a better allocation within each register file possible.

In figure 7.7 the parallel existence of values is compared to the serial rewrite of the values. The parallel graphs are computed with the bipartite algorithm while the serial graphs are computed using the augmenting algorithm. The serial case uses on the average 7 registers and 1 register file less than the parallel case at the cost of at most 2 register transfers in some designs (see figure 7.6).

Finally, figure 7.8 shows the results of from the designs with combined read/write cycle steps (using the general edge coloring) compared to the results of the designs with split read/write cycle steps (using the augmenting algorithm).
Figure 7.6 Bipartite Edge coloring (x), Augmenting Edge coloring (+).

Figure 7.7 Parallel existence of values (x) versus serial re-writes (+).

Figure 7.8 Single read/write cycle steps (x) versus separated read/write cycle steps (+).
7.6. Local Interconnect Optimization

As stated before, the module allocation phase can be used to optimize the local connection links. The assignment of operations to modules can be exploited to minimize the number of connections between the module and the various register files. Also, the commutativity of certain operations (e.g., +, * etc.) can be used to optimize the number of local connections. Two inputs can be exchanged to make use of an operation's commutativity.

In figures 7.9 and 7.10 two versions of a data path for the same scheduled data flow graph are shown. Both contain 3 ALU's to execute the operations and 5 register files to store the values. The different assignment of the operations to the ALU's and the use of the commutativity of operations resulted in a reduction in the number of multiplexer inputs and bus drivers.

![Diagram](image)

**Figure 7.9** Unoptimized data path.

The data path of figure 7.9 contains 12 multiplexer inputs divided among 5 multiplexers and 11 tri-state outputs. Figure 7.10 needs only one two input multiplexer and 8 tri-state outputs. Note that when the number of different inputs and outputs reduce also interconnections to less busses are needed. In this section an algorithm is described which transforms a data path like the one in figure 7.9 to an optimized data path like the one in figure 7.10.

7.6.1. Data Path Model

A convenient way to represent a data path at this stage of the allocation is by a connection cube.

A connection cube is build from $|S|$ connection planes, where $|S|$ is the number of cycle stops assigned by the scheduler to execute the algorithm. The rows of a con-
7.6 Local Interconnect Optimization

![Diagram](image)

**Figure 7.10** Optimised data path for figure 7.9.

connection plane are formed by the inputs and the outputs of the modules. The number of rows in a connection plane can be calculated by the following equation:

\[ \text{rows} = \sum_{i=1}^{M_t} \text{inputs}(m_i) + \text{outputs}(m_i) \]

where \( M_t \) is the number of operation modules. The columns of the plane are formed by the register files. It is assumed that each register file is connected to a single bus, thus the number of register files \( | M_s | \) is equal to the number of global busses.

There is a 1 at position \([i, j]\) in the connection plane \( s \) if the input (output) which corresponds to row \( i \) reads a value from (writes a value to) register file \( j \) in cycle step \( s \). A connection plane is built for each cycle step indicating which connections have to be made in this cycle step. There are as many planes in the cube as there are cycle steps in the schedule. The *result plane* is defined as:

\[ \text{result plane}(i, j) = \bigvee_{1 \leq s \leq |S|} \text{plane}_s(i, j) \]

with:

\[ 1 \leq i \leq |\text{rows}|, \quad 1 \leq j \leq |M_t| \]

The result plane is a projection of the cube to a plane in the direction of the cycle steps. A one at position \([i, j]\) of the result plane means that least in one of the connectivity planes there is a one in position \([i, j]\). Somewhere in the execution of the algorithm a transfer is made from the register file (column) to the processor (row) if row is an input. The transfer direction is reverse when row is an output. The number of ones in a single row show how many busses an input/output is connected.
The total number of ones in the result plane equals the total number of multiplexer inputs and demultiplexer (tristate) outputs in the network. The number of ones in the result plane can be changed by moving operations across different processors and by storing values at different locations. The goal is to minimize the number of ones in the result plane.

7.6.2. The Annealing Algorithm

To minimize the number of ones in the result plane of a cube a simulated annealing algorithm is used. Three types of moves are used.

1. **Row moves**: Two rows in plane $s$ may be exchanged when they are both inputs to the same processor and a commutative operation is executed on this processor in cycle step $s$.

2. **Operation moves**: An operation can be executed on another processor if this processor is of the appropriate type to perform the operation. An operation can be moved to a processor that is free in cycle step $s$, or two operations can be exchanged. An operation move in cycle step $s$ consists of a series of row moves for the input/output rows of the processor to which the operation was assigned in connection plane $s$.

3. **Column moves**: A value $x$ can be moved to another register file if this register file contains no value with the same read and write times as $x$, or two values with the same read and write times can be exchanged.
Obviously the cost function is defined as the number of ones in the result plane. The cooling schedule and the stop criteria are derived according to the annealing algorithm described in [Otten85].

Table 7.1 illustrates the effects of the various move sets on the final results. The operation moves have the largest impact on the final result. This emphasizes that fixing the module allocation in earlier stages of the synthesis process can result in suboptimal designs. Allowing all kinds of moves simultaneously results in all cases in the most optimal designs. When the various move sets were applied sequentially, i.e. three annealing runs are done, each on the results of the previous run where in each run only one type of move is allowed, results were on the average 5% worse than in the simultaneous case.

<table>
<thead>
<tr>
<th>Design</th>
<th>wd17</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>best</td>
<td>%impr.</td>
<td>best</td>
<td>%impr.</td>
<td>best</td>
</tr>
<tr>
<td>set</td>
<td>score</td>
<td></td>
<td>score</td>
<td></td>
<td>score</td>
</tr>
<tr>
<td>col/opr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>row</td>
<td>55</td>
<td>26.57</td>
<td>49</td>
<td>18.33</td>
<td>42</td>
</tr>
<tr>
<td>col/row</td>
<td>62</td>
<td>19.48</td>
<td>55</td>
<td>8.33</td>
<td>49</td>
</tr>
<tr>
<td>col/opr</td>
<td>59</td>
<td>23.38</td>
<td>52</td>
<td>13.33</td>
<td>48</td>
</tr>
<tr>
<td>opr/row</td>
<td>60</td>
<td>22.06</td>
<td>51</td>
<td>15.00</td>
<td>45</td>
</tr>
<tr>
<td>col</td>
<td>68</td>
<td>11.69</td>
<td>58</td>
<td>3.33</td>
<td>53</td>
</tr>
<tr>
<td>opr</td>
<td>66</td>
<td>14.29</td>
<td>53</td>
<td>11.67</td>
<td>46</td>
</tr>
<tr>
<td>row</td>
<td>73</td>
<td>5.19</td>
<td>57</td>
<td>5.00</td>
<td>52</td>
</tr>
</tbody>
</table>

The connection planes can be implemented as sparse matrices. The cost function can be evaluated very fast when running sums over the rows and columns are saved and updated incrementally. This results in a very fast implementation of the annealing algorithm. On HP9000/835 workstations an average of 4000 moves per second was measured almost independently of the cube size. All results in table 7.1 were generated within one minute.

The annealing algorithm seems very suitable for this problem. The cost function gives a very accurate representation of the objective. The annealing behaves very smoothly thanks to the fact that the moves do not introduce very dramatic changes in the cost objective. The largest change in the cost function is equal to the number of rows that are swapped during an operation move. If an operation move is done which swaps two operations with two inputs and one output, which were unable to share any connections with operations in other cycle steps but can share all connections now, a decrease of 6 in the cost function is the result. The cost function versus the annealing steps during some typical annealing sessions is shown in figure 7.12. The number of moves per temperature step increases typically from around 85 at the higher temperatures to around 100 at the lower temperatures. We did also com-
Figure 7.12 Annealing cost schedules for wd17, wd21 and wd28.

Compare the annealing algorithm to an iterative improvement algorithm. This algorithm performs the same set of moves as defined above; however it only accepts moves if they improve the cost function. The results of the iterative improvement and the annealing are compared in Table 7.2. The table shows that also the iterative improvement comes within a few percent from the optimum found by the annealing. This indicates that the solution space for the interconnect problem as defined above is rather smooth, which speeds up the annealing process and increases the probability that it finds the optimal solution.

Table 7.2. Simulated Annealing compared to Iterative Improvement

<table>
<thead>
<tr>
<th>Cost</th>
<th>Annealing</th>
<th>Iterative improvement</th>
<th>Optimization by annealing</th>
<th>Optimization by iterative improvement</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>45</td>
<td>48</td>
<td>21%</td>
<td>15%</td>
<td>6%</td>
</tr>
<tr>
<td>55</td>
<td>42</td>
<td>44</td>
<td>23%</td>
<td>20%</td>
<td>3%</td>
</tr>
<tr>
<td>47</td>
<td>36</td>
<td>36</td>
<td>23%</td>
<td>23%</td>
<td>0%</td>
</tr>
<tr>
<td>48</td>
<td>37</td>
<td>39</td>
<td>19%</td>
<td>19%</td>
<td>4%</td>
</tr>
<tr>
<td>40</td>
<td>31</td>
<td>33</td>
<td>22%</td>
<td>17%</td>
<td>5%</td>
</tr>
</tbody>
</table>

7.7. Benchmark Results

In this section the results of the allocation phase as a whole are compared with results from different allocation schemes found in literature. A comparison is made between HAL [Paulin89], Spicer [Pangre88], SAW [Thomas88] and EASY. In previous approaches less emphasis was put on interconnect minimization, so it is difficult to make comparisons. However, the number of multiplexer inputs are a good
measure for the number of local interconnections. The local interconnect can be compared using the number of multiplexer inputs that appeared in literature. EASY needs in most cases less local interconnections (up to 50% less multiplexer inputs) than the other systems. On the average the datapaths of EASY contain 20% less local interconnections. This improvement in interconnect has to be paid for by a slight increase in the number of registers used. Note that the number of register files remains constant. When an area efficient implementation for a register file is made, significant savings in the final circuits can be obtained. In general the EASY datapath minimizes also the global communication. To make comparisons on the global interconnections is more difficult. Their influence can only be seen when actual layouts for the register transfer descriptions are generated. This will result in less and shorter buses and thus in a smaller layout.

Table 7.3. Allocation results for Fifth Order Wave Digital Filter.

<table>
<thead>
<tr>
<th>System</th>
<th>Cycles</th>
<th>Pipe</th>
<th>Mult.</th>
<th>Adders</th>
<th>Registers</th>
<th>Reg. Files</th>
<th>Local Interconn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EASY</td>
<td>17</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>20</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>EASY</td>
<td>17</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>18</td>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>HAL</td>
<td>17</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>12</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>EASY</td>
<td>19</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>17</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>EASY</td>
<td>19</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>16</td>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>HAL</td>
<td>19</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>12</td>
<td>-</td>
<td>28</td>
</tr>
<tr>
<td>EASY</td>
<td>19</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>17</td>
<td>8</td>
<td>19</td>
</tr>
<tr>
<td>EASY</td>
<td>19</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>HAL</td>
<td>19</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>-</td>
<td>29</td>
</tr>
<tr>
<td>SAW</td>
<td>19</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>-</td>
<td>34</td>
</tr>
<tr>
<td>EASY</td>
<td>21</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>20</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>EASY</td>
<td>21</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>15</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>HAL</td>
<td>21</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>SPLICER</td>
<td>21</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>?</td>
<td>-</td>
<td>30</td>
</tr>
</tbody>
</table>

7.8. Conclusions

In many design styles the minimization of the interconnections is very important. The amount of global interconnection can be optimized during several phases of the data path allocation using efficient edge coloring algorithms. Various edge colorings can be used when different types of data paths have to be allocated. The number of multiplexers and the local interconnections can be minimized by choosing a suitable module allocation after the global communication scheme has been set up. The postponement of the operation and value assignment to the interconnect allocation phase, results in significant reduction in the interconnections required. The results of this data path allocation scheme compare favorably to the results of other systems.
In the previous chapters, various problems occurring in architectural synthesis are described. An assortment of solutions that have appeared in the literature are summarized and quite a few new solution strategies are presented. To make this collection of algorithms practically useful, they have to be clustered in a single system. In the EASY system, the cluster is called the architectural synthesis toolbox.

8.1. The EASY Architectural Synthesis Toolbox

The architectural synthesis tools in EASY are grouped around the design data base. The design data base stores all the objects as described in the notations chapter in this book. It also provides the design data base access functions. All tools use these functions to access the design data. Initially, the data base contains the DFG created by the data flow graph constructor. The data flow graph optimization contains the optimizations described in section 2.8.

8.1.1. Scheduling

The EASY system contains basically two schedulers. When the application has very tight area constraints and the designer knows quite well which modules, and how many of them, he wants to use, a list scheduler can be used. The list scheduler in EASY can handle various urgency measures, such as critical path length, length of the execution interval and the number of successors. When very tight timing constraints are applicable to a design, a scheduler of the force directed type can be used. Force directed schedulers with different object functions and selection criteria are available.

The object functions used in the schedulers mostly take into account the cost of the operation modules, registers and an estimation of the interconnect area. The cost of the registers does not vary that much with the schedule length. If a schedule with many cycle steps is generated, a lot of values have to be stored for many cycle
steps. Therefore, a lot of registers have to keep the values for many cycles and cannot be reused. If a parallel schedule with only a few cycle steps is generated, many values will be generated in each cycle step, which have to be stored for only a few cycle steps. Both these effects seem to compensate each other for most data flow graphs. This indicates that the minimum number of registers does not vary that much with the schedule length. Experiments with a large set of benchmark data flow graphs have shown the same results.

8.1.2. Allocation

The EASY system provides many alternatives for the various allocation phases. In the traditional scheme of register allocation, module allocation and interconnect allocation, the designer can use the original clique search heuristics as defined in [Theng86].

However, we have noticed that the register allocation can be done much better when the module allocation has been done. Since the sources and destinations of all values are known, a register assignment can be done which will result in a reduction in the interconnect. A scheme which does the module allocation before the register allocation is described in [Stok88a] and summarized in chapter 4. If the module allocation can be represented by a comparability graph, the algorithms described in chapter 6 can be used. Otherwise, a heuristic weighted clique covering algorithm has to be used. The register allocation can be handled by a left edge algorithm [Stok88a], which also takes the module allocation into account. The algorithm will deliver a solution with the absolute minimum number of registers and tries to minimize the number of interconnections. When the DFG contains loops, the algorithm
8.1 The EASY Architectural Synthesis Toolbox

described in chapter 5 can be used for the register allocation. When the DFG contains complex branching [Springer90][Stok91b] structures, a general clique search algorithm is used.

The schemes described above work reasonably well in situations where only a limited amount of data flows through the algorithm, which is scheduled using relatively many modules. Thus, almost all values are accessed simultaneously in each cycle step. However, in situations where more values are stored for longer periods of time, and relatively more values are present in the algorithm compared to the number of modules, the schemes sketched above will result in too many interconnections.

When interconnect plays a very important role in the design, an allocation scheme as described in chapter 7 can be applied. In this scheme the global interconnections are allocated before the register and module allocation. Depending on the type of clocking scheme that is used, one of the edge coloring algorithms can be applied to group all values into register groups. The same register allocation algorithms as described in the previous paragraph can be applied for each group separately. The allocation scheme is completed by a simulated annealing algorithm to do the module allocation. The number of local interconnections is the major constituent of the annealing cost function.

8.1.3. Data Path and Controller Generation

When all allocation steps are finished, enough information is added to the database to make a straightforward generation of the data path possible. The correct components have to be instantiated from the library or the appropriate module generators have to be called, and all modules have to be connected as specified in the interconnect allocation function.

The generation of the controller involves more steps. A controller has to be built according to the specification as given in the state transition graph as defined in section 2.5. The binary encoding for the inputs and outputs are already determined by the specification of the modules. The encoding for the states can be freely chosen to optimize the implementation of the state machine. Several general algorithms Kiss [DeMich88], Mustang [Devadas88] have been published to perform this state encoding task. However, due to the special structure of our state transition graph, a more specific algorithm can be used. Several implementations of state machines use counters [Amann87] to count sequences of states. The basic idea behind this approach is that some state transitions can be replaced by counting transitions. The area of the remainder of the controller will be reduced because for the states in the chain, no new state information has to be generated. Therefore, the state encoding starts by trying to find the minimum number of counting chains with maximum length. However, in the type of state transition graphs that are generated by the high level synthesis, it is known where these chains are. This information can be
directly applied in the state encoding. An implementation of such an encoding algorithm within the EASY environment is discussed in [Pernot89]. The encoded state machine is processed further with the EUCLID logic synthesis package [Berkel90].

EUCLID delivers a single module or set of modules for the controller. These control modules will be connected to the data path in the layout generation phase. This phase places and routes all modules and will produce the complete layout for the design.

8.2. Future Work

Architectural synthesis systems targeted to specific applications and used under the appropriate constraints, can already be very helpful to designers. Examples of systems, which have proven to be useful in practice, are given in chapter 7. Where applied, these high level synthesis tools have shown a reduction in design time.

However, there are still many problems, which have to be solved for architectural synthesis systems to become widely used and to be applicable to a large range of designs. A few topics will be discussed in this section.

8.2.1. Operation Assignment

The operation assignment as defined in chapter three determines on what type of library component an operation will be executed. Most schedulers need to have an estimation of the execution time of all operations and require that the operation assignment has been done before the scheduling. A way to incorporate the operation assignment in the scheduling is described in [Stok88a]. However, this will only work if the execution times of the various possible assignments for an operation do not differ widely. The choice is based on area cost considerations. Usually, the operation assignment in EASY is done manually before the scheduling.

In [Rain88] and [McNall90] two heuristic algorithms are described, which give a limited solution to the operation assignment and the module selection problems for strictly pipelined designs. Future research is needed to automate the operation assignment for more general cases and when possible, to include it in the scheduling.

8.2.2. Interaction with Logic / Layout Synthesis

A crucial factor in the high level synthesis is the ability to take into account the results of the lower level design tools. In this book the whole physical design process was represented by a library and a number of parameters for each module. A few constraints which can be put on the library to ease the allocation are described in chapter 6. In reality, the modules are realized by a collection of module generators for structured modules like adders, multipliers, registers and register files and a whole logic synthesis package for the synthesis of more irregular modules, like the controller.
8.2 Future Work

On one side the characteristics of the generators have to be adequately represented in the high level synthesis. On the other side, the generators can also be used to build modules according to specifications defined by the high level synthesis. If a very tight timing constraint is prescribed by the high level synthesis, a module generator can produce a slightly larger module, whose delay fits the specifications. The interaction between the high and low level design tools should be a major topic in future research. Part of the work will consist of providing tools which predict the behavior of the module generators. The architectural and logic / layout synthesis should interact according to a predictor / adapter paradigm as been presented in [Ginneken89] for layout synthesis tools.

8.2.3. Multiplexer and Interconnect Delays

During the scheduling no delays according to multiplexers and interconnections are taken into account. It is assumed that the difference between the cycle time and the critical path in a cycle step is large enough to fit these delays. If these interconnect and multiplexer delays constitute a significant part of the delay, they have to be taken into account in an earlier stage of the architectural synthesis. This requires good estimations of these types of delays and techniques to incorporate them in the scheduling and allocation phases. In the present system, minimization of the number of interconnections is a major objective, but no attention has been paid to the influence of the interconnection delays on the overall delays in the design.

8.2.4. Scheduling of Loops

Yet no adequate solutions exist to the problem of scheduling nested loops. When the number of iterations is fixed techniques like loop folding [Goosse89b] can be applied or the data can be represented by streams [Lippens91], on which many useful operations can be defined. However, when the number of iterations is data dependent, very little is known how to effectively handle these types of data flow graphs.

8.3. Conclusions

This book describes a way to model the architectural synthesis process. Within this framework several new efficient algorithms are presented, which can be applied in the synthesis of certain types of architectures for certain applications. How to effectively use this set of algorithms is illustrated in this chapter. However, still a number of problems have to be solved for architectural synthesis to become generally applicable and widely accepted.
References


References


References


References


References


Notation

- \( V = \{ v_1, v_2, \ldots \} \) \quad \text{set of elements} \quad v_1, v_2, \ldots
- |V| \quad \text{cardinality of} \quad V
- V \cup W \quad \text{set union}
- V \cap W \quad \text{set intersection}
- V \setminus W \quad \text{set difference}
- V \subseteq W \quad \text{\( V \) is a subset of} \quad W
- v \subseteq V \quad \text{element of}
- \emptyset \quad \text{empty set}
- \Pi(V) \quad \text{power set of} \quad V
- < \quad \text{partial order}
- < > \quad \text{sequence, queue}
- \uparrow \quad \text{queue empty}
- \downarrow \quad \text{queue full}
- \text{first element of sequence}
- \text{last element of sequence}
- a . b \quad \text{concatenation of sequences} \quad a \quad \text{and} \quad b
- (x, y) \quad \text{undirected edge}
- [x, y] \quad \text{directed edge}
- \langle a, b \rangle \quad \text{interval}
- l \langle a, b \rangle \quad \text{length of interval}
- \parallel \quad \text{overlapping intervals}
- \chi(G) \quad \text{chromatic index of a graph} \quad G
- \delta(x) \quad \text{degree of a vertex} \quad x
- m(x, y) \quad \text{multiplicity of the edges between} \quad x \quad \text{and} \quad y
- IC(x) \quad \text{colors incident to vertex} \quad x
$MC(x)$: missing colors of vertex $x$

$KS(A, B)$: Kempe subgraph

**Design Data Base Objects**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G(V, E_g, E_d)$</td>
<td>data flow graph</td>
<td>32</td>
</tr>
<tr>
<td>$V$</td>
<td>set of data flow graph nodes</td>
<td>32</td>
</tr>
<tr>
<td>$V_o$</td>
<td>operation nodes</td>
<td>84</td>
</tr>
<tr>
<td>$V_s$</td>
<td>storage nodes</td>
<td>84</td>
</tr>
<tr>
<td>$E_g$</td>
<td>set of data flow edges</td>
<td>32</td>
</tr>
<tr>
<td>$E_d$</td>
<td>sequence edges</td>
<td>32</td>
</tr>
<tr>
<td>$D$</td>
<td>queue of data values</td>
<td>33</td>
</tr>
<tr>
<td>$O$</td>
<td>operation types</td>
<td>36</td>
</tr>
<tr>
<td>$G(S, T, X, Y)$</td>
<td>cycle step transition graph</td>
<td>43</td>
</tr>
<tr>
<td>$S$</td>
<td>cycle steps</td>
<td>43</td>
</tr>
<tr>
<td>$T$</td>
<td>cycle step transitions</td>
<td>43</td>
</tr>
<tr>
<td>$F$</td>
<td>schedule length</td>
<td>54</td>
</tr>
<tr>
<td>$T_{cycle}$</td>
<td>cycle time</td>
<td>66</td>
</tr>
<tr>
<td>$I$</td>
<td>latency</td>
<td>68</td>
</tr>
<tr>
<td>$G(L, M, P, I)$</td>
<td>data path graph</td>
<td>43</td>
</tr>
<tr>
<td>$M$</td>
<td>set of modules</td>
<td>43</td>
</tr>
<tr>
<td>$M_o$</td>
<td>set of operational modules</td>
<td>83</td>
</tr>
<tr>
<td>$M_s$</td>
<td>set of storage modules</td>
<td>83</td>
</tr>
<tr>
<td>$M_l$</td>
<td>set of interconnection modules</td>
<td>83</td>
</tr>
<tr>
<td>$L$</td>
<td>set of library components</td>
<td>43</td>
</tr>
<tr>
<td>$P$</td>
<td>set of pins</td>
<td>43</td>
</tr>
<tr>
<td>$G$</td>
<td>storage group</td>
<td>118</td>
</tr>
<tr>
<td>$G(S, E)$</td>
<td>state graph</td>
<td>119</td>
</tr>
</tbody>
</table>

**Design Data Base Functions**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I(v)$</td>
<td>input edges of a node $v$</td>
<td>33</td>
</tr>
<tr>
<td>$O(v)$</td>
<td>output edges of a node $v$</td>
<td>33</td>
</tr>
<tr>
<td>$I_o(v)$</td>
<td>input sequence edges of a node $v$</td>
<td>33</td>
</tr>
<tr>
<td>$O_o(v)$</td>
<td>output sequence edges of a node $v$</td>
<td>33</td>
</tr>
<tr>
<td>$M: E_g \rightarrow D$</td>
<td>marking</td>
<td>33</td>
</tr>
<tr>
<td>$r: V \rightarrow O$</td>
<td>type of operation</td>
<td>36</td>
</tr>
<tr>
<td>$AD(v)$</td>
<td>address node of array access node $v$</td>
<td>48</td>
</tr>
<tr>
<td>$AR(v)$</td>
<td>array node accessed by access node $v$</td>
<td>48</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>( \Phi(x, n) )</td>
<td>access nodes from set ( X ) with index ( n )</td>
<td></td>
</tr>
<tr>
<td>( t : S \times X \to S )</td>
<td>transition function</td>
<td></td>
</tr>
<tr>
<td>( g : S \times X \to Y )</td>
<td>output function</td>
<td></td>
</tr>
<tr>
<td>( \Theta : V \to \Pi(S) )</td>
<td>scheduling function</td>
<td></td>
</tr>
<tr>
<td>( I : P \to P )</td>
<td>interconnect relation</td>
<td></td>
</tr>
<tr>
<td>( c(v) )</td>
<td>condition vector of node ( v )</td>
<td></td>
</tr>
<tr>
<td>( m(v) )</td>
<td>mutual exclusion vector of node ( v )</td>
<td></td>
</tr>
<tr>
<td>( e(v) )</td>
<td>early execution time</td>
<td></td>
</tr>
<tr>
<td>( l(e) )</td>
<td>late execution time</td>
<td></td>
</tr>
<tr>
<td>( d(v) )</td>
<td>execution interval</td>
<td></td>
</tr>
<tr>
<td>( \Box(v) )</td>
<td>predecessors of ( v )</td>
<td></td>
</tr>
<tr>
<td>( \Box(v) )</td>
<td>successors of ( v )</td>
<td></td>
</tr>
<tr>
<td>( E(v) )</td>
<td>execution time of ( v )</td>
<td></td>
</tr>
<tr>
<td>( CD(v) )</td>
<td>cycle delay of ( v )</td>
<td></td>
</tr>
<tr>
<td>( W(\Theta) )</td>
<td>width of schedule ( \Theta )</td>
<td></td>
</tr>
<tr>
<td>( P(v, \xi) )</td>
<td>probability that node ( v ) is scheduled in cycle step ( \xi )</td>
<td></td>
</tr>
<tr>
<td>( DF(\xi) )</td>
<td>distribution function of operation set ( \xi ) in cycle step ( \xi )</td>
<td></td>
</tr>
<tr>
<td>( SF(v, \xi) )</td>
<td>self force of node ( v ) when scheduled in cycle step ( \xi )</td>
<td></td>
</tr>
<tr>
<td>( g(v, s) )</td>
<td>read steps of value ( v_s )</td>
<td></td>
</tr>
<tr>
<td>( p(v, s) )</td>
<td>last read step of value ( v_s )</td>
<td></td>
</tr>
<tr>
<td>( o(v, s) )</td>
<td>write step of value ( v_s )</td>
<td></td>
</tr>
<tr>
<td>( \lambda : O \to \Pi(L) )</td>
<td>library function</td>
<td></td>
</tr>
<tr>
<td>( c : L \to N )</td>
<td>cost of a library module</td>
<td></td>
</tr>
<tr>
<td>( \sigma : L \to N )</td>
<td>module selection function</td>
<td></td>
</tr>
<tr>
<td>( \sigma : V \to L )</td>
<td>operation assignment function</td>
<td></td>
</tr>
<tr>
<td>( \mu : V \to \Pi(M) )</td>
<td>module allocation function</td>
<td></td>
</tr>
<tr>
<td>( \psi : V \to \Pi(M) )</td>
<td>register allocation function</td>
<td></td>
</tr>
<tr>
<td>( \iota : E \to \Pi(\xi) )</td>
<td>interconnect allocation function</td>
<td></td>
</tr>
<tr>
<td>( \gamma : V \to \mathcal{G} )</td>
<td>storage group function</td>
<td></td>
</tr>
</tbody>
</table>
Index

A
address node, 48
allocation, 138
bipartite matching, 94
branch & bound, 86
function, 44
global, 89
iterative, 87
linear programming, 88
logic synthesis, 89
rule based, 85
arc, splitting, 99
array, 47
array node, 48
assignment, function, 44
astroidal, triple, 112

clique covering, 90
clique partitioning, weighted, 93
cluster, 124
coloring, 91
commodity, 101
common subexpression elimination, 46
companion, arc, 99
conditional, tree, 111
connection cube, 128
connection plane, 128
connection set, 92
constant, propagation, 46
teacher, 27
generation, 127
cost, function, 101, 114, 131
cycle
delay, 66
step, 43
time, 96
cycle step, transition, 98

D
data dependency analysis, 32
data flow graph, 32
semi, 59
true
combined, 60
separate, 69
data path, generation, 137
dead code elimination, 46
Index

degree, 118
design
  exploration, 28
  statistics, 28
disjoint access, relation, 118
distribution function, 74
drawing, data flow graph, 23

E
EASY, 155
  overview, 21
edge coloring
  augmenting, 155
  bipartite, 122
  general, 120
BEF, 22
EUCLID, 138
Kureka, 23
execution interval, 64
execution rules, 34
execution set, 34
execution time, 66
  early, 64, 67
  late, 64, 67
expressions, 36
extractor, design data, 22

F
feedback, design information, 22
finite state machine, 43
flow, 115
  microcommodity, 102
force, 75
functional specification, 19

G
graph
  asteroidal, 112
  circular arc, 98
  comparability, 107, 108
  complement, 112
  composition, 108
  data path, 43
  interval, 91
  module allocation, 107
  proper circular arc, 104
  register allocation, 97
  register preference, 93
  state, 119
  state transition, 43
  transitively orientable, 106

H
hierarchy, 40

I
interconnect, delay, 139
iterative improvement, 132

K
Kampe subgraph, 119
Kiss, 157

L
language, behavioral description, 21
latency, 68
layout, synthesis, 138
layout synthesis, 17, 138
left edge, 91
library, 65
  complete, 109
  function, 65
  ordered, 109
lifetime
  circle, 96
  interval, 81
line router, 23
load balancing, 74
local interconnect optimization, 128
logic synthesis, 17, 138
loop invariant computations, 47
loops, 139

M
marking, 33
maximal element, 109
memory, 47
missing colors, 119
module, 43
allocation, weighted, 113
operational, 43
netlist
column, 130
operation, 130
row, 130
multi-cycling, 66
multiple read value, 128
multiplexer, delay, 139
multiplicity, 118
Mustang, 137
mutual exclusion, 62, 67, 77
data dependent, 54
dynamic, 55

N
network, 114
flow, 101
network flow
maximal cost, 116
multi-commodity, 108
node separation, 115

O
operation
assignment, 153
assignment function, 65
operation type, 36
operation type set, function, 109
optimization, data flow, 46
output function, 43
overlap set, 98

P
Pablo, 23
partial ordering, 108
pipelined cycle steps, 68
pipelining, 68, 77
placement, schematics, 23
power set, 43
procedure
elimination, 47

Q
expansion, 47
queue, 33
unbounded, 33
Quinto, 23

R
register
allocation, 126
cyclic, 97
transfers, 124
register file, 47
register merging, 93
return, edge, 115
routing, schematics, 33

S
schedule length, 64
scheduling, 135
algorithms, 70
as late as possible, 72
cost driven, 71
integer linear programming, 71
iterative, 71
as soon as possible, 72
critical path, 73
force directed, 74
list, 72
loop, 139
neural network, 75
transformational, 70
scheduling scheme, 69
block-by-block, 69
elevation, 69
path based, 70
scheme
allocation, 85
data path allocation, 117
synthesis, 20
sensitive classes, 34
sensitivity, data flow graph, 33
sequence edges, 33
simple schedule, 107
simulated annealing, 130
state
machine, 43
transition graph, 43
Index

state graph, 119
bipartite, 122
statement
conditional, 37
procedure, 40
repetition, 38
storage
group, 118
value, grouping, 118
storage group function, 118
superior, 109
syntheses, scheme, 20

T
toolbox, 135

transition, 43, 98
transition function, 43
transitive orientation, 108
tree, 58
conditional, 53
expression, 47
tree height reduction, 47
type, 36

U
unconditional, DFG, 107
use-definition, analysis, 58
Leon Stok was born on January 12, 1964 in Veldhoven, The Netherlands. He studied electrical engineering at Eindhoven University of Technology, the Netherlands, from which he graduated with honors on August 28, 1986. Since then he has been working on a Ph.D. degree in the Design Automation Section of the Eindhoven University. He hopes to graduate on the work in this thesis on July 10, 1991.

Mr. Stok did work in the "Unternehmensbereich Kommunikations- und Datentechnik" of Siemens AG in München during the summer of 1985. He was given the SIPP (Siemens International Practical Programme) award for this work. He has worked in the Mathematical Sciences Department of IBM's Thomas J. Watson Research Center, Yorktown Heights, New York, during the second half of 1989 and the first half of 1990. In July 1991, he will start working in the Computer Science Department of the Thomas J. Watson Research Center.

Mr. Stok has published several papers on the various aspects of high level and architectural synthesis and on the automatic placement and routing for schematic diagrams.
Stellingen

behorende bij het proefschrift van
Leon Stok

1. De toewijzing van de operaties aan modules is een van de stappen in de data pad synthese die zeer grote invloed heeft op de hoeveelheid benodigde verbindingen. (Dit proefschrift, hoofdstuk 7)

2. Het minimum aantal registers benodigd voor de realisering van een "geschedule-de" data flow graph is nagenoeg onafhankelijk van de lengte van het "schedule". (Dit proefschrift, hoofdstuk 8)

3. Een groepering van variabelen in een minimaal aantal groepen, zodanig dat in een groep geen twee variabelen tegelijkertijd geschreven of gelezen worden, kan in polynomiale tijd opgelost worden. (Dit proefschrift, hoofdstuk 7)

4. Indien de juiste randvoorwaarden aan de module toewijzing worden opgelegd, kan de module toewijzing gerepresenteerd worden door een compatibiliteits graaf, hetgeen het mogelijk maakt dat een aantal problemen met algoritmes van polynomiale orde, opgelost kunnen worden. (Dit proefschrift, hoofdstuk 6)

6. De eenvoudigste manier om eerdere benchmark resultaten te verbeteren, is een deel van de benchmark weg te laten.


7. Een geïntegreerde, geïnterpreteerde en geëxecuteerde werkomgeving maakt een zeer snelle ontwikkeling van prototype programmatuur mogelijk zonder al te veel aan executietijd in te boven leveren en is derhalve te prefereven voor prototype implementaties gedurende wetenschappelijk onderzoek in CAD.

8. Het aanleggen van allerlei voorzieningen voor de gehandicapten maakt de voetpaden ook heel wat toegankelijker voor fietsers, hetgeen de verkeersveiligheid op de voetpaden niet ten goede komt.

9. In een multi-disciplinaire IC-productie omgeving zijn de diverse ingenieurs te herkennen aan de eenden waarin zij rekenen: de werktuigbouwkundige rekent in μ's, de IC ontwerper rekent in vierkante μ's, de fysicus in kubieke μ's en de bedrijfskundige in Mensuren.

10. Het doen van onderzoek is een tijdconcurrent proces, en derhalve zou iedere zichzelf respecterende onderzoeksinstelling 24 uur per dag toegankelijk moeten zijn.

11. De kennis van de literatuur is van essentieel belang voor het doen van goed onderzoek. Derhalve is het opzetten van een goed literatuurzoeksysteem van vitaal belang voor een onderzoeksinstelling, en dienen daartoe genoeg financiële middelen ter beschikking gesteld te worden.


12. Als planologen zouden ze afdelen mijden hebben als IC-ontwerpers, zouden zowel het file als het woningprobleem met de nu ter beschikking staande middelen direct oplosbaar zijn.