Complexity Analysis and Algorithms for Data Path Synthesis

by

Chittaranjan Mandal

A thesis submitted
in partial fulfilment of the requirementss
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
West Bengal 721302, INDIA

September 1995

Certificate

This is to certify that the thesis entitled **Complexity Analysis and Algorithms for Data Path Synthesis** being submitted by Chittaranjan A. Mandal, an external research scholar in the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, for the award of the degree of **Doctor of Philosophy** is an original research work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Institute for award of any degree or diploma.

Dated : Sept. 1995
Kharagpur 721 302

P. P. Chakrabarti, Assistant Professor
Department of Computer Sc. & Engg.
Indian Institute of Technology, Kharagpur
W. B. 721 302, INDIA.

Acknowledgment

First and foremost I wish to express my sincere gratitude to my supervisor Prof. P. P. Chakrabarti for sharing with me his invaluable technical skills and offering me encouragement throughout the duration of this research work. Working with him was a pleasure and a learning experience. I wish to give him my special thanks for patiently going through my thesis and offering constructive criticisms for its improvement.

I also wish to thank Prof. S. Ghose for taking keen interest in my work and offering deep insights into some aspects of the synthesis problem. He has always been a source of inspiration and encouragement.

Thanks are due to Prof. P. Pal Chaudhuri, Prof. S. C. De Sarkar, Prof. A. K. Majumdar and Prof. A. Pal for their direct and indirect support. I am thankful to Prof. M. K. Roy, Prof. D. Ghosh Dastidar and Prof. R. Datta Gupta for their help and co-operation at Jadavpur University.

I fondly re-collect the kind hospitality extended to me on several occasions by Prof. P. P. Chakrabarti and his family. I have enjoyed the friendly company of Dr. Apurba Banerjee, Mr. Gautam Biswas, Indrajit Chakrabarti, Santanu Chatterjee, Dibyendu Das, Sukumar Nandi, Sudeshna Sarkar and others.

I am grateful to my parents, Paramita and Subrota for their affection and support. Without the blessings of my parents this work would not have been possible.

Chittaranjan A. Mandal

Abstract

We are now witnessing a proliferation of integrated circuit solutions to several signal processing and related problems. This has motivated the evolution of High Level Synthesis (HLS) where the objective is to obtain an efficient register-transfer level (RTL) realization of a target system from its behavioural specification. An important aspect of HLS is the Data Path Synthesis (DPS) problem, which concerns the construction of optimal data paths of the target system. In this work we examine the complexity of several DPS problems and propose solutions to them. The overall objectives of this work are characterization of the complexity of the synthesis problem and proposing solutions to major tasks involved in DPS. We have, in general, considered the hardness of the problem in question and also the hardness of its approximation schemes. We have found that several interconnect optimization problems, like port assignment of multi-port memories, register-interconnect optimization and allocation and binding, are NP-hard and their relative approximations are also NP-hard. A few scheduling problems, on the other hand, can be solved optimally in polynomial time. List scheduling sometimes guarantees a 2-approximate solution under some conditions. A number of scheduling problems and their constant approximations are also NP-hard. To the best of our knowledge the complexity of relative approximation of many scheduling problems, of interest in DPS, is open.

We have proposed solutions to some individual interconnection oriented sub-problems of data path synthesis and solution to the entire DPS problem. We have made use of heuristic techniques, controlled search and genetic algorithms as problem solving tools. The individual sub-problems are register-interconnect optimization (RIO), memory-interconnect optimization (MIO) and port assignment (PA) of dual and triple port memories. For RIO and MIO we use heuristic techniques, whereas the PA problems are solved using genetic algorithms. For the entire DPS problem we have adopted an approach based on a design space exploration. Instead of finding a single design from a given behavioural specification, we seek to find a set of competitive designs which satisfy the specification. We have proposed a two phase solution to the entire synthesis problem. In the first phase we find a set of competitive schedules through design space exploration, based on controlled search and heuristic methods. We have also developed a genetic list scheduling technique to work with our DSE technique. In the second phase we find the data paths for each of these schedules through allocation and binding using a genetic algorithm approach.

Keywords: VLSI Design, Data Path Synthesis, Algorithms, Complexity Analysis, Genetic Algorithms, Design Space Exploration, Interconnect Optimization, Memory, Port Assignment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We have witnessed the progress of I.C. fabrication technology in leaps and bounds. In the SSI era only tens of transistors could be fabricated on a single chip. In the late sixties, the MSI era, the number of transistors per chip was between a hundred and a thousand. The range moved up to about twenty thousand in the seventies, the LSI era. The eighties saw the ushering of the VLSI era when the number of devices per chip went up to fifty thousand. Nowadays chips housing over a million transistors are common place.

While in the SSI era it was quite natural to craft each transistor and wire on the chip individually, at present it would be impossible to design a complex chip involving over a million transistors without sophisticated design tools.

In the early stages, when only a few transistors could be fabricated on the chip the circuits designed were very simple and it was mainly necessary to check that the transistor and wire layouts were correct. For this, design rule checking tools were developed. When it became possible to fabricate some more transistors, tools and languages were developed to facilitate layout, routing and other aspects of physical design. An important example is the Caltech Intermediate Form representation [1]. Logic design tools and FSM synthesis tools also started finding extensive application in chip design. Soon the use of module libraries became common. Thus two levels of design became apparent, the low level design or physical design and the intermediate level design or logic design. The increasing complexity of the systems necessitated such a distinction between design stages for proper design management.

Testing has always been an important design step for which additional structures may have to be introduced in the design to facilitate observability and controllability of the relevant points in the circuit. The excitation that should be applied and the response that should be observed are computed in this step. Testing is intended to detect faults introduced during fabrication. It is assumed that the design is correct prior to testability synthesis.

As one moves from gates and flip-flops, to small modules like counters, shift registers and functional units (f.u.) like adders, subtracters and arithmetic logic units (ALU), to digital systems like microprocessors, digital signal processing (DSP) systems and application specific integrated circuits (ASIC) the complexity of the products keep on increasing. In the design of complex digital systems architectural design has been found to play an important role.

The design of present day digital systems involves a variety of design architectures and design automation tools. The architecture could be a simple pipeline architecture, systolic array architecture, a super-scaler architecture, an architecture supporting parallel execution of operations and so on. Each architecture would require a different design technique. In attempting to develop a tool to support a set of design techniques many factors need to be taken into consideration. The degree of automation provided by such a tool will depend on the maturity of the automation techniques available, vis-a-vis the demands at the market place. For example, there would be considerable man-machine interaction in the design of the architecture of a contemporary microprocessor to achieve the highest levels of throughput, considering the market requirements. However, for the design of some types of ASIC's, to be designed and supplied in low volumes but in a short time, it may be very desirable to use tools to design their detailed architecture to achieve correct designs having reasonably good performance. A mature tool may be expected to outperform the human designer for a reasonably small design cycle time.

It may be noted that for complex digital systems there are some design steps that precede logic design and physical design. These are collectively called High Level Synthesis (HLS), which are explained in the next section. The present work addresses an aspect of HLS called Data Path Synthesis.

1.1 High Level Synthesis

High level Synthesis seeks to construct a circuit structure from a behavioural specification. While considering such a scheme three issues immediately become apparent; these are:

- The kind of language that should be used for the specification.
- The general properties of the designed circuit structure.
- The kind of optimizations that should be performed while mapping from the behaviour to the structure.

In principle each issue could be addressed in a variety of ways. The language could be designed in the lines of conventional programming languages or it could be specifically designed for digital signal processing or be richly equipped with simulation constructs. The target structure could consist of simple RTL components or it could be a specialized structure. The types of optimization used will be partially governed by the way the input is expressed and the type of circuit structure that is to be used. On the whole the optimizations have to be tailored to find a balance between the performance of the target design and its cost. At present no strict formalism for the design prevails. This can be attributed to the inherent complexity of the problem, arising partially out of a large number of different aspects involved and to the wide variety for each aspect.

1.1.1 Hardware Description Language

A hardware description language (HDL) typically has several programming language features such as data typing, operations, assignment statements and control constructs. Data typing is an important aid in the development of behavioural specifications, because of its use in making some consistency checks. For the purpose of synthesis some hardware specific features are also included. These include interface declarations, structural declarations and interprocess communication. Structural declarations are useful for incorporating pre-designed modules which can be used directly for the current design. HDL's also permit the specification of design constraints to guide the synthesis tool. These come in the form of design parameters, performance, costs, testability, reliability and other physical restrictions. Some of these constraints are usually specified separately from the behaviour. Timing constraints are typically interleaved with the behaviour. Typical HDL's are VHDL [2], Verilog [3], Mimola [4] and Hardware-C [5].

1.1.2 Target Architecture

The target system is expected to accept inputs and produce outputs. If the outputs happen to be independent of past inputs then the system could simply be a combinational logic network. Otherwise, it becomes necessary to incorporate memory into the system. From a theoretical angle, the target machine could be a finite state machine (FSM) or a push down automata (PDA), depending on the kind of computation power required. Practically speaking, the system will always have finite memory, and is therefore, a FSM. Present design techniques for FSM's can handle thousands of states, but it is easy to encounter design problems where millions of states are involved. A class of these complex FSM's can be described and designed in a structured manner using techniques of HLS. The realization consists of two major components. One is a conventional FSM and the second is an associated data path. We calls such machines a *FSM with Data Path* (FSMD), as proposed in [6]. This is an appropriate nomenclature for the target architecture used by many of the current synthesis tools. The data path of an FSMD could use bus based or point to point interconnection topology. It may be biased towards using individual hardware operators like adder, subtracter, etc, or towards multi-function functional units like arithmetic logic units, etc. to implement the operations used in the behavioural specification. The target architecture could also use pipelined components.

1.1.3 Design Steps

The algorithmic description has to be compiled and converted into an intermediate representation, without losing any information present in the original specification. This representation could be a flow graph of basic blocks (b.b.) [7] or a control data flow graph (CDFG) [6]. There are many different types of CDFG's in HLS. An overview of these is available in [8]. Some transformations can be carried out on this intermediate representation. These could be dead code elimination, loop optimization, extraction of common sub-expressions and so on, which are commonly employed in compilers. The next step is scheduling where operations are assigned to control steps. An optimization called data flow graph restructuring which is relevant for synthesis, is sometimes performed either before scheduling or simultaneously with it. This is followed by the design of data path and the control part of the FSMD. The control part is essentially an FSM which may be designed using FSM synthesis tools [9, 10, 11]. Scheduling and construction of the data paths of the FSMD usually come under Data Path Synthesis (DPS). The back end of HLS consists of data path and control part synthesis. The optimized FSM is essentially the control part.

1.2 Data Path Synthesis

Data Path Synthesis (DPS) starts with the optimized control and data flow information of the target design along with the design parameters and proceeds towards the construction of the data paths of the target system. The parameters could be the number of input/output *interface* ports to be used, the maximum number of operations that may be executed in a time step, the number of buses to be used in the system and so on. The design parameters arise out of physical constraints or the desire of the designer to guide the design in a particular direction. DPS requires several sub-tasks to be handled. These include i) scheduling of operations ii) formation of functional units (f.u.) to execute operations iii) formation of a set of registers and memories (storage configuration) to store values and iv) interconnect allocation.

Scheduling determines the number of time steps required to implement the behaviour. Indirectly it also affects the number and configuration of f.u.'s to be allocated, the storage and the bus requirement. While scheduling it may be necessary to accommodate wide differences in the speed of the hardware implementation of various operations. After scheduling it is necessary to find the data path structure to implement the schedule of operations. The data paths are made up of functional units, storage units in the form of individual registers and memories, interconnection elements in the form of buses and switches, and interface ports. It is a natural design objective to find a data path of minimum cost. However, minimization of the cost of the data path components even for a particular category of components has been a difficult problem. On the other hand a good design requires a globally optimized data path.

A lot of work has already been done in DPS, leading to the development of individual techniques, systems and analytical results. A recent survey of data path synthesis appears in [12] and a tutorial may be found in [13]. Some of the prevailing HLS/DPS systems are: the ADAM System [14], HAL [15], Chippe [16], Facet [17], Mimola [4], Hercules [18], Gauss [19] and VITAL [20]. Most of these systems include tools for scheduling and allocation and binding and address some if not all the sub-problems that we have addressed. A variety of scheduling techniques are used. HAL introduces force-directed scheduling. Chippe's scheduler Slicer uses a list scheduling algorithm. Multi-port memories are increasingly being used as an RTL component. A k-port memory permits k simultaneous (consistent) accesses to its cells. Gauss is one the tools that make use of multi-port memories in the data path. A number of individual techniques have been developed for register allocation with interconnect optimization when the variable conflict graph happens to be an interval graph [21]. This situation is interesting because for this case efficient optimal algorithms, like the left edge algorithm [21], are available for pure register allocation.

Individual scheduling methods include ILP scheduling [22], lower bound resource estimate based list scheduling [23], and zone scheduling [24]. STAR [25] and Rim's method [26] are individual methods for allocation and binding. Methods for integrated allocation and binding include SAM [27], a method proposed by Balakrishnan and Marwedel [28] and Devadas' method [29]. Most of the work on DPS is implementation oriented, however, some theoretical studies have also been done. An analytical method for estimating wire lengths has been proposed in Plest [30]. Pangrle analyzes the complexity of connectivity binding in [31]. Some classical results on scheduling complexity [32] carry over to DPS. A recent tutorial on genetic algorithms appears in [33].

1.3 Major Objectives

This work address the problem of data path synthesis (DPS). From a theoretical point of view it is understood that the complete problem of optimization of the data path is NP-hard. This is because some sub-problems like scheduling of a general data flow graph or optimization of registers are known to be NP-complete. The practical implication of this is that we do not expect a polynomial time algorithm which will solve the DPS problem optimally. The only known optimal approaches are exponential in complexity. The size of the DPS problem is by itself quite large. A high order polynomial time algorithm is sometimes unworkable in practice. Therefore, exponential time algorithms are nearly ruled out for large scale DPS problems. However, DPS is an important problem which demands reasonably fast and good solutions. Therefore, the DPS designer is placed in the difficult situation of finding a good and reasonably fast solution to a seemingly intractable problem. In such a scenario a logical approach to be followed is to *critically understand the complexity of various aspects of the synthesis problem* (with respect to exactly how difficult it is to obtain optimal and sub-optimal solutions of different sub-problems) and *appropriately develop a framework for solving these problems*, making the best use of available problem solving methodologies.

In view of the above, the present work aims at studying the data path synthesis problem both from a theoretical and practical standpoint. The overall objectives of this work are:

1. Characterization of the complexity of the synthesis problem with special emphasis on demarcating the complexities of individual sub-tasks.

2. Proposing solutions to major tasks involved in DPS.

While it might not be possible to address each and every aspect of DPS, the aim has been to consider the complexity of some of the sub-tasks which have not attracted the interest of researchers greatly and to use problem solving methodologies which have not been widely explored for DPS. In the next two sections we highlight the issues related to the above mentioned objectives and summarize the results obtained through this work.

1.4 Complexity Studies

The two major tasks of DPS are generally accepted to be *scheduling* and *module allocation and binding*. Both these sub-problems have several aspects. In this work we examine the complexity issues of several variations of both scheduling and allocation problems. We examine *scheduling with time constraints* and *resource constraints*. As scheduling problems in DPS almost always involve precedence constraints, we consider mostly scheduling problems belonging to the precedence constrained scheduling category. Both for scheduling and allocation we have considered a few new problems also. One such problem is the *scheduling of variable transfers to take place under the constraints imposed by the available hardware*. We call it the variable assignment problem. For allocation and binding we consider the complexity of the *port assignment problem for dual and triple port memories*. We have examined the complexity of the general allocation and binding problem with special emphasis on *interconnect optimization*. In particular we have examined the complexity of *interconnect optimization for straight line code*. This problem is specially interesting because the register optimization problem has an efficient polynomial time solution. We have also examined the complexity of constructing *functional units of minimum cost*.

Our style of performing the complexity analysis of a problem is to try to identify the simplest case of that problem that is NP-hard. We also look for the most general case of that problem that is solvable in polynomial time. These two help to characterize the complexity of a problem and indicate when the problem becomes intractable. This analysis also helps in suggesting what type of solution may be used in different situations. The more accurate this characterization is, the better is the understanding of the complexity of the problem. For example the problem of scheduling when the precedence constraints are a tree (or forest) where each node performs the same type of operation, can be solved in polynomial time [32]. However, if the precedences are in the form of a directed acyclic graph (DAG) the problem becomes NP-hard [32]. On the other hand if there are only two functional units, then the problem of scheduling a DAG can be done in linear time [32].

While examining the complexity of a problem we first check whether it has a polynomial time solution or whether it is NP-hard. Most of the problems that we have considered belong to the latter category. There is another major issue related to complexity analysis of practical problems like DPS. We may relax the requirement of optimality provided we get some guarantees regarding the maximum degradation in the solution quality and hope for a polynomial time solution. Therefore, with a problem shown to be NP-hard we examine the complexity of finding a feasible solution whose cost is bounded in some respect to the cost of the optimal solution. Such a solution is called an approximate solution and the problem of finding such a solution is called an *approximation* version of the original problem. One such is the absolute approximation scheme where the cost of the approximate solution differs from the optimal by a fixed constant. The other scheme is that of relative approximation where the cost of the approximate solution is related to the cost of the optimal solution by a fixed constant factor. For example though scheduling of a single operation DAG is NP-hard, list scheduling guarantees that the schedule obtained will not be more than twice the optimal schedule length [34]. However, some problems may be so complex that the problem of approximation is itself NP-hard. This gives us another criterion to characterize the complexity of a problem.

With these aspects in view we have considered the complexity of DPS problems in an attempt to find out the nature of complexity of sub-problems mentioned earlier. We now summarize the results obtained in scheduling and allocation.

1.4.1 Summary of Scheduling Complexity Results

In VLSI scheduling we usually have more that one type of operation in the DAG and the functional units (f.u.) are typically heterogeneous. There are precedence constraints between the operations. These precedences, in general, take the form of a directed acyclic graph. The operations may execute in either in a single time step or in multiple time steps. In some situations scheduling has to be done in the presence of shared resources. For example, if several operations have to read data form a ROM having only one port we have a case of resource constraint.

Keeping DPS applications in mind we have first attempted to analyze the complexity of DAG's having more than one type of operation. A very simple type of DAG is a set of chains. We have considered the problem of scheduling chains having two types of operation, on two functional units (one of each type). We also consider the special case of scheduling only two chains.

- The problem of scheduling a set of chains having only two types of operations (unit execution times), on two f.u.'s (one for each type of operation), given a deadline D is NP-complete.

- The problem of finding a minimum length schedule of two chains of two types of operations using two functional units, one of each type, is solvable in polynomial time.

There is an earlier result that the problem is NP-hard for DAG's [32]. It is also known that single operation DAG's can be scheduled in polynomial time, using two functional units, to minimize schedule length [32]. The problem of scheduling single operation type trees is known to be polynomially solvable for an arbitrary number of f.u.'s. Chains are a special kind of trees.

We have the following result for resource constrained scheduling. It may be noted that a previous result states that a similar problem is NP-hard for DAG's [32].

- Scheduling m chains having only one type of operation with two f.u.'s, unit execution times, and one resource with limit 1 is NP-complete.

We also analyze the complexity of approximations of scheduling problems.

- Absolute approximation of scheduling DAG's is NP-hard for the problem of minimization of schedule length.
- Absolute approximation of scheduling DAG's with multiple operation types, given a deadline D, is NP-hard for the problem of minimization of the number of f.u.'s (where each f.u. implements only one type of operation).

The variable assignment problem is a new problem that we have considered, and for it we have the following two results.

- The problem of scheduling variable assignments in minimum number of time steps, subject to the availability of a fixed number of points from where these variables can be accessed is NP-hard.
- The absolute approximation of scheduling variable assignments in minimum number of time steps, subject to the availability of a fixed number of points from where these variables can be accessed is NP-hard.

1.4.2 Summary of Allocation and Binding Complexity Results

The NP-hardness of register optimization (RO) is a well known result since clique partition can be reduced to this problem. Pure register optimization without considerations for interconnect optimization usually leads to poor designs. Some new results have appeared on the complexity of connectivity binding, subject to certain assumptions [31]. We have derived a number of results regarding the hardness of register-interconnect optimization (RIO) and the complexity of the corresponding approximation schemes.

The first set of results that we have derived are on the complexity of the port assignment (PA) of dual and triple port memories. The PA problem is to assign the memory accesses to its ports so that the cost of interconnection required is minimized. The PA problems arise when such memories are used as storage elements in the data path. Complexity analysis of this problem has not yet attracted much attention. For port assignment we have the following results.

- Port assignment for dual port memories is NP-complete.
- Port assignment for triple port memories is NP-complete.
- Relative approximation of port assignment for triple port memories is NP-complete.

The following two results concern the complexity of interconnect optimization.

- Register-interconnect optimization for straight-line code (SRIO) is NP-hard.
- Relative approximation of register-interconnect optimization for straight-line code is NP-hard.

- Relative approximation of general register-interconnect optimization is NP-hard.

A functional unit is a circuit capable of implementing a set of behavioural operations, such as $< +, -, * >$. An interesting problem that we have considered is the problem of forming functional units at minimal cost. Here we have the following result.

- The problem of determining the assignment of operations to a fixed number (exceeding one) of functional units so as to minimize their cost is NP-hard.

As mentioned earlier we have tried to identify the simplest case of a problem while analyzing its complexity. For example for scheduling we have considered the case of chains. As indicated already a slightly simpler version of the problem is solvable in polynomial time. Similarly, the port assignment of single port memories is trivial. While RO for straight line code has an efficient polynomial time solution (the left-edge algorithm) [35], SRIO as well as its relative approximation are NP-hard. Even for the problem of scheduling variable to variable data transfers in minimum number of time steps the only constraint we have considered is the number of available storage access ports.

From the above results we note that while both scheduling and binding problems are mostly NP-hard, we have been able to prove that for many of the allocation and binding problems even the constant bounded relative approximation is NP-hard. Albeit, the complexity of approximation of many of the scheduling problems is still open. However, list scheduling does guarantee schedules for single operation DAG's whose length is no more that twice the optimal schedule length. This suggests that allocation and binding appear to be difficult than scheduling in many situations.

1.5 Implementation for DPS Sub-Problems

The second part of the work involves development of algorithms for solving sub-tasks for data path synthesis. Here we have two major objectives.

1. To solve some individual sub-tasks related to interconnect optimization.
2. To solve the entire DPS problem so as to generate the optimized data path from a set of data flow graphs.

1.5.1 Solutions to Some Individual Sub-Problems

Interconnect optimization has been receiving attention of researchers due to its impact on the area of the final design. An excessive number of switching elements in the interconnection aspect of the design also inflates the complexity of the FSM that will implement the controller. We have first considered the problem of optimizing both the register and interconnect cost, during register optimization. We call this problem *register-interconnect optimization* (RIO). We then consider the more general problem where the storage is implemented using not only registers but also multi-port memories.

This problem is called *memory interconnect optimization* (MIO). The use of multi-port memories gives rise to another interesting sub-problem, which is that of assigning the accesses to the memories to its ports. This is the *port assignment problem* (PA). We present solutions to the port assignment problem for *dual and triple port memories.* The register and memory interconnect optimization problems are solved using heuristic algorithms, while the two PA problems are solved using a genetic algorithm approach. The techniques used are now briefly described.

Work on Register and Memory Interconnect Optimization

Register-interconnect optimization (RIO) is one of the individual DPS sub-tasks that we have considered. The input is assumed to be a partial design where scheduling, functional unit formation and operation binding has been completed but storage formation is yet to be done. Storage formation has an impact not only on the number of registers and memory cells that will be required but also on the interconnection structure (and its cost) that will be needed to satisfy the transfers between the functional units and the storage. For RIO we use the multiplexers needed in the circuit to derive a measure of the interconnection cost. A multiplexer reflects the additional connectivity of the points where it is used. This combined optimization would result not only in a reduction of the number of registers and multiplexers in the design but also in the physical interconnection which is coupled with multiplexer usage.

For RIO we have developed a heuristic procedure to achieve a joint optimization of the register and multiplexer cost, based on a clique partitioning formulation. This formulation generates a state space where the basic transformation operation is register merging. The heuristic looks for a pair of compatible registers for merging which correspond to a pair of vertices connected by an edge in the graph. The choice is made on the basis of a measure which we call the clique factor. This is a quickly computable global measure based on the number of common vertices, the deletable edges, the total number of edges having at least a certain number of common vertices and the saving in the hardware cost as a result of merging two compatible vertices or registers. The idea is to choose the edge which will help to maximize future mergings while maximizing the current hardware saving.

In memory-interconnect optimization (MIO) we have then considered the more general problem of using multi-port memories instead of individual registers only. A greedy approach has been used to determine which variables should be packed into the current memory. The introduction of a variables to a memory is subject to two constraints. One is that the maximum number of ports required to satisfy all the accesses to the memory should not exceed a pre-defined limit. The second constraint is that the maximum number of memory cells should not exceed a pre-defined limit. The latter constraint is to ensure that the access time of the memory does not become too large. The maximum number of ports has been limited to three. For MIO, when multi-port memories are used, the port assignment is formulated in terms of RIO and solved using the RIO algorithm.

Techniques for both RIO and MIO have been tested on individual designs. The technique for RIO has been tested on a large number of automatically generated examples as well. In this experimentation, for each design we computed two results, one was for pure RO and the other was for RIO, using the same algorithm. This has helped us not only to evaluate the robustness of the algorithm but also to make a study of the kind of trade-off involved to incorporate interconnect optimization with register optimization. The results indicate that with a little increase in register cost there can be considerable saving in the interconnect cost.

Work on Port Assignment

The other individual sub-problem that we have worked upon is that of port assignment (PA) of multi-port memories. We have developed genetic algorithms for the PA of dual and triple port memories. The basic requirement of a PA is that inputs or outputs of registers or f.u.'s, (in general points in the circuit) that access the memory in the same time step should be able to satisfy their access to the memory through distinct ports. The accesses to the memory can be obtained by examining the schedule and the data path where operation, variable, transfer bindings have been completed. For a dual port memory the access conflicts are conveniently represented by a graph. However, for triple port memory a graph representation turns out to be inadequate and so a hyper-graph is used instead. For both the cases separate genetic algorithms (GA) have been developed.

The formulation for the dual port memory PA is based on node deletion to make a graph colourable using two colours. The GA for this problem uses a special crossover which makes use of a graph colouring algorithm. We have been able to show that the solution generated using the crossover will be an optimal one, with high probability. This property of the crossover makes it less susceptible to the deceptability problem [36].

We have proposed a graph theoretic formulation for the triple port memory PA problem as well. Here we not only need to represent binary relations to indicate two simultaneous accesses (as for dual port memory PA) but also ternary relations to indicate three simultaneous accesses. Conventional graphs do not offer a simple solution to this requirement of representation and so we use hyper-graphs, instead. The edges may include either two or three vertices. The vertices may take one, two or three colours to abstract connection to the ports of the triple port memory. A valid colouring is one where the colour sets of vertices connected by a hyper-edge include separate colours. The objective is to minimize the total number of instances where additional colours are used. A special algorithmic crossover has been developed for this problem too.

PA is a computational intensive task. So it not always feasible to use it as a lookahead, especially where a number of optimizations are being done at the same time. For dual port memory PA we have developed a probabilistic estimator to estimate the interconnect cost arising out of the assignment. Extensive experimentation has shown that the estimates provided are in close conformity with the solutions obtained by the dual port memory PA genetic algorithm. The techniques for both the PA's have been tested on a large number of automatically generated examples. In all the cases the results obtained have indicated that our methods yield optimal or near optimal results.

1.5.2 Solution to the Entire Data Path Synthesis Problem

We now outline our methodology and implementation for the entire DPS problem. The input is a set of optimized data flow graphs which are connected to form the flow graph [7] which indicates the flow of control. The data flow graphs contain the operations which appear within a basic block, and the dependencies between these operations. The operations could be single time step or multi-cycle operations. We are required to find a data path which is "optimized". However, the implication of optimization in this situation is slightly involved. We seek to optimize not only the area cost of the data path but also the performance of the implementation measured as a function of the number of time steps involved in the final design. This makes the synthesis problem one of multi-criteria optimization. There are other criteria like power dissipation and testability which designers may also seek to optimize. The difficulty with multi-criteria optimization is that the criteria are usually non-commensurate and sometimes conflicting. It is, therefore, difficult to combine the criteria into a single optimization function. Following recent ideas in [37], we take the approach of representing the design cost as a tuple of the costs of the individual objectives. A design is said to dominate another distinct design if the cost of the first is no worse than the cost of the second with respect to each criteria. The global problem of optimization is to find the set of designs which are not dominated by any other designs. Systematic exploration of the design space to find all or some of the set of non-dominated design points is usually termed Design Space Exploration (DSE). Therefore, any solution to the synthesis problem which considers multiple optimization criteria must not only consider the aspect of scheduling and allocation but also have scope for DSE. Some work on design space exploration for signal processing applications has been done by Kurdahi et al. [38, 39]. Our work is concerned with tradeoffs relating to schedule length and f.u. cost.

In this work we present a two fold solution to the synthesis problem. In the first phase we do DSE with scheduling to obtain a set of non-dominated design points. Each design point corresponds to a cost related to area and a schedule. The number of time steps in the schedule gives an estimate of the performance. In the second phase we perform allocation and binding. A bus based interconnection structure is used for constructing the data path. Allocation and binding consists of functional unit formation (f.u.), storage formation, binding of operations to f.u.'s, binding of variables to memories, binding of transfers to buses and interconnect allocation. More accurate area estimates of the data path is available after allocation and binding. Design space exploration and scheduling is done using a combination of systematic search of the design space and a variety of scheduling techniques. These include heuristic scheduling algorithms and genetic list scheduling. The complete allocation and binding is done using a genetic algorithm approach. The genetic approach has the advantage of simultaneously looking at a set of competitive solutions. It, therefore, provides a controllable method of exploring design alternatives. It also has the natural capacity of allowing the use of multiple heuristics. This makes it quite an attractive scheme to solve a very complex problem like DPS. A further motivation of using the GA technique is due to the encouraging results obtained through extensive experimentation on the PA problem. Since scheduling appears to be less complex than allocation we have used a time controlled branch and bound technique at the heart of DSE. In order to contain the combinatorial explosion and yet allow some exploration of possibilities we have used the genetic approach for allocation and binding. The details of the two phases are briefly described next.

Design Space Exploration and Scheduling

Conventional scheduling algorithms require a time constraint or a specification of the available f.u.'s. In a practical DPS situation neither the appropriate time constraint nor the appropriate f.u. requirement will be known in advance. If DSE is to be integrated with scheduling it is necessary to systematically explore several combinations of time constraints and hardware configurations that are feasible. We use the concept of multi-criteria optimization and arrive at several configurations with different performance and f.u. requirement estimates. We employ a multi-objective search approach to perform design space exploration and scheduling. In our scheme we have a state space generation mechanism coupled with an estimator for obtaining various <hardware cost, time> estimates. A depth first branch and bound is used to search the state space. In order to contain the combinatorial explosion, the computational effort to be spent on DSE can be controlled by certain parameters. While designing these tools we also permit the designer to impose design parameters and then examine the design space for possible designs which satisfy these parameters. We have chosen these parameters to reflect some important architectural aspects, such as the number of buses, the number of f.u. sites, the number of system ports, etc. over which the designer may wish to have some control.

The state space generation for DSE has been done in a special manner which is different from previous approaches [37, 40]. Due to the time constrained nature of DSE it would not always be possible to generate the complete state space. We then have to settle for lower bound (l.b.) estimates of the f.u. requirement. We took the approach of successively partitioning the input DAG's into smaller ones to have better estimates of the hardware operator requirement and also to get partial schedules. Thus after the first phase of DSE we have a set of design points. With each design point we also have the set of partitioned DAG's which had lead to its f.u. estimate component. At this juncture we complete the schedules to these partitioned DAG's using standard algorithms like FDLS [41] or the scheduling method proposed in [23]. The solutions obtained from this completion gives us upper bound (u.b.) estimates. If these match with the lower bound estimates obtained through DSE, we can terminate with accurate design points and schedules. On the other hand, if the u.b.'s and the l.b.'s differ, we explore around the estimated design point for feasible schedules leading to non-dominated <hardware cost, performance> design points. That is, we make limited search (in polynomial time) around the estimated design points obtained earlier. Our study of some list scheduling algorithms shows that these algorithms usually terminated with optimal solutions for small DAG's. Therefore, in our state space generation we performed decomposition in a balanced manner to ensure that the sub-problems generated after DSE are small and more suitable for existing scheduling algorithms.

We have employed the genetic paradigm to develop a genetic list scheduling scheme. This algorithm has been developed to accept the partitioned DAG's generated by the

DSE tool. As usual a population of feasible schedules is maintained. As in list scheduling a ready list of operations is maintained. While constructing a new solution, instead of immediately employing a priority function to select the operations to be scheduled, the first round of selection is done by inspecting the operations that have been scheduled in the appropriate time step of the two parent solutions. If there are still some idle f.u.'s and the ready list is not empty then the remaining operations to be scheduled in the current time step are selected using an elementary priority function.

The DSE scheme has been applied to some prevailing designs like the Facet example [17], elliptic wave filter [42] and differential equation solver [43]. The genetic list scheduling scheme has been extensively tested using randomly generated schedules.

Allocation and Binding

A tool for allocation and binding has been developed based on the genetic approach. The input is a set of scheduled data flow graphs and some design parameters and the output is the optimized RTL data path. The interconnection style used is bus based. Support for multi-cycle operations and use of memories and pipelined functional units in the data path is available. The objective is to minimize the overall hardware cost of the data path measured as the sum of costs of the f.u.'s, storage units, interface elements and switching elements. The advantages of the genetic method are: i) an implicit parallel searching of several building blocks that go into the making of solutions (the building block hypothesis [44]) and ii) at termination of a large number of distinct solutions of the best cost are often obtained, unlike most other methods which terminate with only one solution.

In the design of the GA for this task a number of factors had to be taken into account. In view of the complex nature of the problem a structured representation clearly indicating the various bindings has been used. The conventional bit string representation was not found convenient. An intelligent crossover has been used, for two main reasons. First, simple splicing and juxtapositioning almost invariably lead to infeasible solutions. Secondly, ensuring feasibility is also not enough to obtain good quality solutions in the population. This is mainly due to the *deceptability* inherent in the problem. A special population control mechanism had to be used to sustain maximum diversity in the population, while at the same time retaining solutions with good overall and also partial fitness. It was not sufficient to simply give preference to better cost solutions in the genetic pool. We used the cost of the memory elements of the data paths as a secondary cost in deciding whether to retain the solution or to replace it with a newly generated solution. This was guided by our observation of the importance of an appropriate memory configuration in the solution. We also had to exercise special care in implementing the parent selection strategy. As a solution to the above two problems we classified solutions having high fitness value (low cost) or having a low cost memory configuration, with respect to their memory configuration. While removing a solution we avoid removing a solution which will lead the number of distinct memory configurations to fall below a certain threshold. It may also be noted that solutions of a class are genetically "close". This property is helpful when we like to have a crossover between similar parents.

While designing the crossover it may be noted that operation to functional unit mapping and transfer to bus mapping need to proceed time step by time step, whereas the memory formation is based on the simultaneous consideration of the lifetime and access information gathered form all the time steps. The basic steps in the formation of a new solution through crossover of two parent solutions, in our method, are briefly listed below. First construct the memories of the new solution through a sub-crossover of the memories of the parent solutions. It is then necessary to proceed time step by time step to obtain the complete solution. In each time step the following need to be done. i) Complete the essential transfer and operation bindings to satisfy the data transfer and execution requirements of multi-cycle and pipelined operations. ii) Perform operation to f.u. bindings for the pending operations for this time step. iii) Perform transfer to bus bindings for the pending transfers for this time step. For the last two steps a force directed heuristic algorithm has been incorporated in the crossover to improve the chances of a better offspring when two solutions are crossed. This algorithm has been designed to make use of the existing partial structure to satisfy the pending bindings of operations and transfers. Thus while binding transfers to buses preference is given to a binding that not only can be satisfied using existing links and switches connected to the bus but also leaves open a similar opportunity for the maximum number of pending transfers. A similar criterion is used for computing the forces for the operation to f.u. bindings. In any time step the binding of pending operations precedes the binding of pending transfers. A limited lookahead for transfer forces has been incorporated in the computation of the operation forces. The technique for allocation and binding has been tested out on several prevailing examples like the Facet example [17], elliptic wave filter [42] and differential equation solver [43]. In some cases pipelined multipliers have also been used. In most cases we have obtained slightly better results than what has been reported in the literature. The results also indicate that the use of memories in the the data path construction lead to compact designs, which we believe is a useful feature.

Though we have analyzed the complexity of several sub-tasks of synthesis and developed algorithms for the entire synthesis problem, there are aspects of DPS which have been kept out of the scope of this work. These include testability, power dissipation and clock cycle synthesis. Our major objective was to critically analyze the complexity of some of the basic synthesis sub-tasks and propose effective solutions based on our understanding of the complexity. That is why we solved some problems using heuristic algorithms, some using controlled branch and bound and made use of GA for more complex tasks like allocation and binding. The work has indicated that our methods perform well.

1.6 Thesis Organization

In the next chapter a brief survey of related work has been made. The subsequent two chapters of this thesis are devoted to the complexity studies. In chapter 3 we examine the complexity of scheduling and in chapter 4 we examine the complexity of allocation and binding in DPS. The register and memory interconnect problem was the first problem that we solved and we present the work on RIO and MIO in chapter 5. In this chapter we have introduced the use of memories in RTL design. Here we also present a general formulation of port assignment for multi-port memories. We have found this formulation more useful for the complexity analysis of several synthesis sub-tasks than for developing large scale synthesis systems. The refined techniques for port assignment are presented in the next chapter (chapter 6), where we present solutions for the port assignment of dual and triple port memories. Chapter 7 is on design space exploration and scheduling where we cover exact and approximated design space exploration, and genetic list scheduling. Chapter 8 is on allocation and binding. Wherever it has been practicable we have tested out the algorithms statistically on several automatically generated problem instances. This method of testing has been applied to RIO. A more refined method of statistical testing has been used for dual and triple port PA and genetic list scheduling. The other techniques have been tested with prevailing examples. Our conclusions and some directions of future work are presented in the last chapter. We have included three appendixes. In the first appendix we give a brief introduction to genetic algorithms which we have employed several times to solve some DPS sub-problems. In the second appendix we have included schedules used for the experimentation on allocation and binding. In the last appendix we have tabulated the results of experimentation on DSE with randomly generated schedules.

Chapter 2

Related Work

Various research groups around the world have been working on the problem of *Data Path Synthesis* in the context of *High Level Synthesis*, well over a decade. It will not be possible for us to survey all the work that has been done in this span of time. We only present a brief description of a subset of systems and individual techniques which are closely related to our work, and which have helped us to develop an initial understanding of the problem and the related issues.

The survey has been done in the following order. First we take a look at some synthesis systems. Then we examine individual techniques which have not already been covered under these systems. There we also examine approaches to scheduling, methods for binding and allocation, integrated scheduling and allocation and design space exploration. We also briefly mention some results related to the complexity of data path synthesis sub-problems.

2.1 Some Synthesis Systems

We have presented a description of the systems for two reasons. One reason is to study the data path synthesis techniques that they have adopted. The other reason is to give a more global perspective of both the problem and the solution approaches. The systems that we take a look at are: The ADAM System, HAL, Chippe, Facet [17], Mimola [4], Hercules [18], Gauss [19] and VITAL [20]. All these systems have modules to handle some, if not all, the sub-problems that we have addressed. We also describe other systems like STAR [25], SAM [27], etc when we discuss individual techniques.

2.1.1 The ADAM System

The ADAM system [14] consists of a number of tools for design automation. Some of the tools are as follows:

CSSP to generate a clocking scheme.

MAHA to perform scheduling and functional unit allocation.

REAL to perform register allocation.

SEHWA to perform pipelined synthesis.

The clock scheme synthesis package (CSSP) is first invoked to determine the number of clock cycles for the design. The critical path is divided into the corresponding number of time steps. This module also determines the period of the clock cycle.

Maha is the scheduler. It first allocates processor units –referred to a functional units in the actual work– to operations on the critical path. The operations off the critical path are then scheduled. These operations are scheduled in the increasing order of their freedoms. An operation is scheduled in the earliest time step where it can be bound to a processor unit that has already been allocated. If no resource sharing is possible then it is arbitrarily scheduled in the earliest permissible time step and an additional processing unit is allocated.

If at some point of time the hardware cost is exceeded then the process is repeated all over again, but with a more relaxed time constraint. The relaxation is brought about by partitioning the critical path into more number of time steps.

After scheduling register allocation is performed using the well known left-edge algorithm. This algorithm is optimal when the register life times is representable as an interval graph (as in case of a single DAG).

Sehwa is a program for the synthesis of pipelined data paths. The subtasks identified are:

1. Scheduling - assignment of operations time steps or clock cycles.

2. Resource allocation - allocation of specific number of modules of various types, and

3. Register-transfer synthesis - detailed assignment of operations to operators, and placement and interconnection of storage elements and multiplexers.

Sehwa can find minimum cost designs, highest performance designs and other designs in between the two in the design space. Sehwa makes use of a resource allocation table for resource conflict checking. An urgency measure based on the critical paths starting from each node, considering actual delay times of operators assigned to operations is used for scheduling the operations. Sehwa also has a module for predicting area time tradeoffs for pipelined data paths.

2.1.2 HAL and Related Techniques

HAL [43] is a complete synthesis system based on multiple programming paradigm. At the heart of this system is the force directed scheduling (FDS) algorithm. FDS relies heavily on the computation of a global measure is called *force*, analogous to the force exerted by a spring obeying Hooke's law. The details of force computation is given in the following subsection on *force computation*. The operations are scheduled one by one in specific time steps. The choice of the time step in which an operation is to be scheduled is done in a best first manner, using the force as the heuristic measure.

The concept of the force arising out of the distribution of operations has been generalized to storage elements as well. From the partial schedule, where all operations have not yet been scheduled to unique time steps, a heuristic estimate is made of the distribution of the life time of each storage operation. This is integrated with the scheme to perform operation balancing to obtain a schedule in which both operations and storage elements are well balanced.

A separate module performs allocation of data paths. First arithmetic and logic operations are bound to specific functional units so as to minimize the number of distinct inputs to each one. For the purpose of storage a special facility called local storage is used. Local storage permits a value to be stored in multiple locations. This is sometimes useful in reducing the interconnection requirements for the data path to be constructed. Register minimization is done using *weight directed clique partitioning*. In this type of partitioning, the compatibility graph is reduced by considering only those registers which have interconnection affinity, over some threshold. By making the threshold high at the beginning the large number of edges in the compatibility graph is considerably reduced. As the number of candidate registers for merging goes down the threshold is lowered to permit greater merging. This helps to reduce the problem size of clique partitioning. A similar method is used for merging multiplexers to form buses.

FDLS

A related technique is Force Directed List Scheduling (FDLS) [41]. The basic scheduling technique is that of list scheduling. The priority function is based on force computation. As in list scheduling if sufficient operators are available the operations in the ready list are scheduled in the current time step. Otherwise operations in the list are scanned one by one to check which one is to be deferred to subsequent time steps. The cost of a tentative deferment is taken as the resulting force value. The deferment that leads to the minimum force is chosen. In case the resources get exhausted before all the critical operations in the ready list are scheduled, then the length of the schedule is incremented.

Force computation

The time frame, T, of each operation is computed from the ASAP and the ALAP schedules. It is assumed that the operations are uniformly distributed in the time steps of their time frames. Therefore the operation is assumed to be scheduled in any slot of its time frame with probability $\frac{1}{|T|}$. The probability that an operation o is scheduled in time step t may be represented P_o^t. The Distribution Graph of operations of type A in time step t may be represented as DG_A^t. It may be computed as $DG_A^t = \sum_{\{o\,:\,type(o)\,=\,A\}} P_o^t$. The force measure tries to quantify the effect of restricting (scheduling) an operation in any slot(s) within its time frame. This results in a new time frame of that operation, and in general, will also cause changes in the time frames of other operations. This in turn affects the probability of occurrence of those operations in their original time frames.

The change in probability of an operation o occurring in a time step t may be represented as $\delta(P_o^t)$. The effect of tentatively restricting an operation is studied by computing the *force*, defined as $F = \sum_{\{\forall o, \forall t_f\}}(DG_A^t \cdot \delta(P_o^{t_f}))$, where t_f represents the time steps in the time frame of the operation o. As it can be seen the summation is carried out over all the operations. This imparts a global character to the force definition. Use of the force heuristic requires that the operator set should partition the set of operations used in the specifications.

2.1.3 Chippe

Chippe [16] uses an expert system driven by global constraints and the present state of the design to make design trade-offs. Chippe's design process is sub-divided into four coupled tasks. These are: i) allocating the control unit and the data path, ii) scheduling the operations, iii) building the interconnect and iv) evaluating the design. In Chippe allocation is performed iteratively by an expert system.

Chippe's scheduler, Slicer, uses a list scheduling type algorithm which calculates the critical path and then heuristically schedules operations in each time step in a particular order. The operations are ordered according to their freedoms. Ties are resolved in favour of operations with greater number number of successors. Operations which are not schedulable in their ASAP time are deferred to the next time step. Slicer supports both multi-cycling and chaining.

Chippe's interconnect optimizer is Splicer. It uses a bus based interconnectivity model. Splicer uses a branch and bound algorithm, using heuristic estimates, which performs operation to functional unit (f.u.) binding, f.u. to bus binding and register to bus binding. Splicer performs connectivity binding through four major operations:

1. connect-register-to-bus,
2. connect-bus-to-unit,
3. connect-unit-to-bus and
4. connect-register-to-bus.

This system permits the user to control the search for optimal connectivity by sending only portions of the state graph at a time for binding. This is referred to as lookahead.

Chippe iterates the above stages in a closed loop under the guidance of an expert system. To support this strategy it is necessary to evaluate the design. The evaluations include design performance, function unit usage, interconnection usage and power consumption.

2.1.4 Facet

Facet [17] starts with an *asap* schedule of the given DAG and then proceeds to do allocation and binding. Facet presents a graph theoretic formulation of the binding task.

The three important aspects of binding, viz. variable to register binding, operation to functional unit binding and transfer to bus binding, have been addressed. A uniform approach has been adopted all the three problems. Each sub-task has been presented as a clique partitioning problem and a heuristic procedure has been used to solve this NP-Complete problem. In Facet the scheduling and binding tasks are well segregated, and any scheduling technique that does not perform binding may be used.

Facet designers have expressed the importance of interconnect optimization while doing register, functional unit and bus minimization. However, in the system implementation this concern has been addressed only in a small way.

2.1.5 MIMOLA

MIMOLA [4] is a tool to facilitate the design of digital processor, following a top-down method. It accepts the input specification, which may be structural or behavioural through a HDL also known as MIMOLA. MIMOLA requires the behavioural specification to be coded in the form of micro statements. If the input is a behavioural description the output contains a listing of the hardware which will be necessary. In addition statistical information of the hardware utilization is also generated. It is possible to declare the amount of createable hardware and the available hardware.

MIMOLA, the tool, has a number of modules. Among them are the compiler, hardware allocator and the statistical analyzer. All the information of the hardware is maintained in a hardware data structure (HDS). The hardware allocator of MIMOLA looks for correspondence between the language expressions and the hardware structures represented in the HDS. If sufficient hardware is not available the allocator determines what additional hardware may be defined, subject to restrictions already declared. When there is ultimately insufficient hardware the hardware allocator gives a *pseudo error* indication to the compiler which then splits the microinstruction.

2.1.6 HERCULES

Hercules [18] is a system for high level synthesis developed at the Stanford University as part of the Olympus synthesis project, involving high level, logic and physical synthesis. Some of the important components of Hercules are modeling hardware behaviour, transformation that preserve functionality without structural implications and the mapping of behaviour into structure. Hercules uses *Hardware C*, a derivative of the *C* language, and having among other features, process and interprocess communication description features, for describing the hardware.

Behavioural optimizations are performed on the parse tree derived from the input specifications. A special data structure called the *Reference Stack* is used extensively for performing behavioural optimizations, which are mainly compiler style optimizations, but also include procedure in line expansion, loop unrolling, etc. Structural synthesis maps the optimized parse tree into a sequencing graph model called the sequencing intermediate form (SIF).

After deciding on the modules to be used and linking them with the logic equations that describe the data path, a combined logic optimization of the controller and derived data path is performed to generate the final design. This is similar to the approach taken in the Yorktown Silicon Compiler [45]. Iterative refinement of the structure is guided by the estimates on area and timing provided by logic synthesis.

2.1.7 GAUSS

Gauss [19] is an complete synthesis system. It permits micro architecture synthesis of digital systems from a behavioural description in C. It has several modules, some of which are: translator from behavioural description to intermediate flow graphs, module identifier, flow graph optimizer, controller synthesizer and data path synthesizer. Gauss uses a straight forward method to construct the initial data path of the design. It then performs interconnect optimization by packing the registers into multi port memories using GREGMAP, described below. In addition to the basic tools for synthesis Gauss also incorporates some features for efficient design management.

GREGMAP

These techniques have been developed to perform interconnect allocation using multi port memories. This method primarily attempts to find a packing of registers into multi port memories. The number and type of multi port memories to be used is determined heuristically by one module and the actual packing of registers into the memories is performed by another module. Initial assignment of the registers to the memories is performed without directly taking interconnect minimization into account. The assignment of a register to a particular memory is guided by the *affinity* of unbound registers to the various memory modules. Subsequent refinements to the initial allocation through greedy register interchanges between memory modules. In case, at some point of time, it is not possible to assign a register to any memory module then the earlier decisions are altered in a search framework through backtracking. In case a memory configuration is found to be incapable of accommodating all the registers then a new configuration is generated, either automatically or manually, in the designer mode. The built-in functions for altering a memory configuration are:

ADDPORT Add a port to a memory,

DOUBLE Double the size of a memory,

SPLIT Split the memory into two and

COALESCE Combine two memories into one.

2.1.8 VITAL

VITAL [20] is an integrated approach to data path synthesis. This actually consists of three techniques: VITAL-EX (for synthesis with exhaustive search), VITAL-SS (for synthesis with selective search) and VITAL-NS (for synthesis without any search). The last of these is essentially a heuristic optimization technique. The tool performs scheduling, allocation and binding sub-tasks of DPS. F.u.'s, registers and buses are partially allocated during the scheduling stage. It can produce optimal solutions using an exhaustive search scheme or it approximate solutions using heuristic techniques. It is also capable of making a trade-off between the solution quality and computation time by controlling the search space and accuracy of the cost model. It can synthesize using pipelined operation units. It also supports the use of multi-function and multi-cycle functional units (f.u.). An interesting feature that it supports is the use of implementations of operations differing in speed. It supports both time and cost constrained scheduling. Data paths with bus based and point to point interconnection can be designed.

Various ILP formulations have also been developed for the above problem, but the heuristic method has been generally preferred because of its computational time advantage.

2.2 Approaches to Scheduling

It is evident that *list scheduling* is a widely used scheme. This is basically a resource constrained scheduling (RCS) technique. However, most systems use priority functions that depend on the knowledge of the task completion time. For this reason, very often a relaxable time constraint is imposed. In this section we discuss some scheduling techniques used for DPS not discussed earlier. FDS and MAHA are time constrained scheduling (TCS) algorithms, designed to minimize the resource cost. The ILP scheme presented in section 2.2.2 is an optimal formulation for TCS. Zone scheduling (section 2.2.3) is a parameterized RCS technique.

2.2.1 List Scheduling using a Lower Bound Measure

A list scheduling algorithm similar to FDLS, but using a different heuristic measure to determine which operations of the ready list have to be deferred in the absence of sufficient resources for executing these operations, has been reported in [23]. The measure used is a lower bound (l.b.) of the operator cost. The operation whose deferment leads to a minimum increase in the lower bound of the resource requirements is chosen; ties are resolved in favour of operations with lower freedom. For this method to work the operators in the operator set, as in FDS, must partition the set of the types of operations present in the DAG. It is assumed that the number of time steps n, available for scheduling is known before hand. The basic method for lower bound determination is described in subsection on *lower bound definition*.

Lower Bound Definition

A good measure that may be used on a DAG is the lower bound on the cost of operators that will be eventually required. The total number of time steps within which scheduling must be done, say n, is known at the time of scheduling. The time frame of each operation is determined from the ASAP and ALAP times. The l.b. of each type of operation is determined by examining each of the $\frac{n(n+1)}{2}$ windows in the DAG. An operation definitely occurs in a particular window if its own time frame lies completely within that window. If p operations of a particular type x are present in a window w of size l, then at least $\lceil \frac{p}{l} \rceil$ operations of type x must be present in some time step of w. This, in fact, is a lower bound on the number of operations of type x that will be finally required. Let this value be $lb_{x,w}$. For all windows w_y defined over the DAG, the maximum value of lb_{x,w_y} is a lower bound for operations of type x. Let this be lb_x. If it is assumed that operations of different types are realized on distinct operators, then the lower bound on the operator cost may be defined as $\sum_{x_i}(lb_{x_i} \cdot C_{x_i})$, for all operations x_i, C_x being the cost of an operator for an operation of type x.

A weaker l.b. can be computed by considering the n windows, instead of $O(n^2)$ windows as required in the above scheme. These windows are formed by keeping the end boundary of the windows fixed at the last time step and varying the start boundary, as proposed in [46].

2.2.2 ILP scheduling

The ILP method [22] finds an optimal schedule using a branch and bound algorithm that involves backtracking. The key ingredient is casting the scheduling problem in an integer linear form, using 0/1 integer variables, so that an ILP package can be used to obtain the solution.

In general, to improve the time required to solve the problem, the formulation can be streamlined to reduce the number of variables involved. Also some additional constraints may be carefully added. A standard method is to first determine the time frames of the operations from the *asap* and *alap* schedules. This helps to restrict the number of 0/1 integer variables required to indicate the time step where the operation is scheduled.

The main difficulties with the above method are: (i) ILP is among the hardest NP-Complete problems, and (ii) the derived ILP problem grows rapidly with the size of the original scheduling problem. However, there are situations where ILP may still be attractive.

2.2.3 Zone Scheduling

Zone scheduling is a relatively new technique for the resource constrained scheduling problem [24]. The approach works by successively dividing the control steps into zones and solving each of them by a 0/1 ILP technique. By varying size of the zone the formulation can serve as an optimal one for list scheduling or something in between. While scheduling a zone a decision has to be made regarding which operations are going to lie in that zone and in which time steps of the zone these operations are going to be scheduled. The operations whose time frames are completely within the zone are necessarily scheduled within the current zone while the *cross* zone operations, whose time frames go beyond the lower boundary of the zone, may or may not lie in the current zone. The choice in made by optimizing an objective function defined in the lines of the *force* [43], so that the more difficult to schedule operations are included into the current zone, as far as possible.

2.2.4 Design Space Exploration

Design space exploration (DSE) is a relatively new area where the goal is to find a set of competitive designs for a given behavioural specification. We are more interested in DSE at the time of scheduling, but DSE is applicable at almost any level of design. A hierarchical design space exploration technique for signal and image processing applications has been presented in [38]. This approach is based on the identification of regularly occurring templates in the data flow graph. A design space exploration technique has been suggested in [40], to evaluate components in a library for use in a design by considering the weighted sum of the parameters (area, delay, power, etc.) of each component. Integrated scheduling, allocation and binding schemes are also categorized as DSE techniques by some researchers [28].

2.3 Methods for Allocation and Binding

Binding is computationally a difficult task. Simple heuristic methods do not work well, while optimal methods take up a lot of time. The general approach has been to solve the problem in parts using an optimal method. This method has been used in STAR and Splicer. Some systems/techniques, such as the ILP method discussed in section 2.3.2 and SAM (section 2.4.1) solve a restricted version of the problem. SAM uses a heuristic method and yet reports good results.

2.3.1 STAR

It is a package for data path allocation and binding [25]. It divides the problem into three sub-tasks: i) pre-processing, ii) data path construction (DPC) and iii) data path refinement (DPR). STAR treats three important important aspects of the binding task: i) data transfer binding, ii) operation assignment and iii) variable binding. STAR binds registers to register files rather than to individual registers. In the DPC phase first the data transfer bindings are performed. Subsequent to this, the register binding and operation binding may be performed independently. In each case a restricted branch and bound algorithm is used to obtain the assignments. For the DPR phase, a graph is constructed so that vertices corresponding to objects whose bindings are well co-related are connected by a edge of a relatively high weight. Then the data path is evaluated globally by evaluating the binding quality of each object, and probabilistically selecting a cluster of well co-related objects to be ripped up on the co-relation graph. The selected objects are re-allocated to form a better design or it is determined that there can be no more cost improvement.

2.3.2 Binding Using ILP

In [26] an ILP formulation has been proposed for constructing the data paths of a system. The system is capable of handling pipelining, multi-cycling, operation chaining and operation commutation. A minimum number of modules of each type and registers required by the implementation of a design are determined from the schedule and the max-cut of the input description, respectively. Mapping of behavioural entities, such as operations and variables, is then done to minimize the interconnection cost, measured either as the multiplexer requirement or the wire requirement. A point to point interconnection style is assumed.

2.4 Integrated Scheduling and Binding

Scheduling, allocation and binding are inter-related tasks, and in principle they should be solve simultaneously. This makes the problem even more difficult, but efficient formulations which have come up gradually have made this a feasible approach now. Some methods which do this are briefly outlined in this section.

2.4.1 SAM

This tool combines scheduling allocation and mapping in a single algorithm. The SAM algorithm is based on the scheduling ideas developed for force directed scheduling [27]. The algorithm uses the notion of force to measure the effect that a tentative scheduling of an operation would have on the resource requirements. The concept has been extended by adding terms to the force equation that represent the compatibility of the operation to the individual instances. The compatibility is based on the match between the connections that exist to the instance. This allows the force calculation to be used to select an operation for scheduling that is best for instance mapping as well as for resource utilization.

The outline of the algorithm is as follows. While there are still operations to be scheduled and mapped, the following things are done in a loop. All the unmapped and unscheduled operations are considered. First the time frames of the operation are calculated. Then each operation is tentatively scheduled in each of the possible control steps and a force is calculated. This force includes factors related to operator concurrency and connection compatibility. Once all forces have been calculated then the tentative scheduling with the most negative force is selected. This defines the operation and the control step where it will be scheduled. The next step determines the mapping for this operation. Using the information already calculated about the compatibility between operation and the instances, the best mapping is selected. The final step updates the data structures by marking the operation as scheduled, allocating new hardware if necessary, and adding any required connections to the representation of the hardware structure.

2.4.2 Technique Proposed by Balakrishnan and Marwedel

A method for integrated scheduling and binding has been presented in [28]. The basic scheme is to schedule an operation and then bind it along with its associated source and destination operands. The operands are bound to storage elements while the operation is bound to a functional unit. This loop is carried on till all the operations are scheduled. Three types of scheduling schemes are supported:

1. Forward scheduling (based on as soon as possible).
2. Backward scheduling (based on as late as possible).
3. Double headed scheduling (Based on alternating between the two).

The binding decisions are made via a 0/1 ILP problem. A rich mix of operators, including operators of various speeds for a particular operation, is permitted.

2.4.3 Devadas' Method

This method [29] performs scheduling and allocation simultaneously. The design is represented in a two dimensional grid for various time and hardware operator combinations. The total cost of the design is taken as

$$C \;=\; p1 \cdot (f.u.cost) \;+\; p2 \cdot (exec\_time) \;+\; p3 \cdot (\#register) \;+\; p4 \cdot (\#bus)$$

$p1, p3$ and $p4$ are area parameters while $p3$ is the execution time parameter. The parameter $p1$ is a function of the processor cost. The processor cost is determined by examining the operations that must be performed by a particular processor. This is done by examining the processor-time grid. The parameter $p2$ indicates the relative importance attached to the execution time as against the area parameters. The execution time refers to the time steps used in a tentative schedule. The parameter $p3$ is a function of the cost of a register and the number of registers currently needed in the design. The the number of registers is estimated from the density of variable life times in a tentative schedule. The parameter $p4$ is derived from the number of components currently used in the design and from the skew of the value transfer nets in the data flow graph (DFG), in a partially bound tentative design. The optimization is performed in a stochastic framework using the method of simulated annealing.

2.5 Complexity Studies Related to DPS

2.5.1 Complexity of Scheduling

A lot of work has been done on job shop scheduling [32]. There the term "processor" usually refers to functional units in normal DPS terminology. However, these results are mostly for single operation DAG's. An important result in this category is the *unit-execution time (UET) scheduling problem*, in yes-no form. The aim is to determine whether it is possible to schedule n tasks, each requiring a single time step and having precedence relationships which form a partial order, on m processors within a deadline D. This problem is known to be NP-complete [47]. The problem of one and two step multi-cycling on two homogeneous to minimize schedule length is known to be NP-Complete [47]. The problem of scheduling in the presence of resource constraints is another class of scheduling problems bearing relevance to DPS. It is known that scheduling with three processors, unit execution times, one resource and empty precedence constraints is NP-complete [48]. It is also known that a similar problem for DAG's where the precedence constraints form a partial order and there are two processors in NP-complete [48].

2.5.2 Complexity of Allocation and Connectivity Binding

Register minimization and some other problems bear a direct correspondence with the clique partitioning problem (CPP) [17]. CPP is a well known NP-complete problem [49]. Thus allocation and binding in general is understood to be NP-complete. However, except for some studies in [31] not much work in complexity characterization of DPS has come to our notice. We mention some results related to connectivity binding obtained by Pangrle in [31] below.

A study of the connectivity binding problem had been done in [31]. Results on two restricted cases have been derived. The first case is as follows: First we assume that the following are present: i) a set of registers $\mathbf{R}$ where $|\mathbf{R}| = \rho$, ii) a set of operation types $\mathbf{\Phi}$, iii) a set of units $\mathbf{U}$ where $|\mathbf{U}| = \mu$ and each unit performs a subset of the operations in $\mathbf{\Phi}$, iv) a state graph $\mathbf{G} = (\mathbf{V}, \mathbf{A})$ where $|\mathbf{V}| = \nu$ and $|\mathbf{A}| = \alpha$ and v) a state assignment $f_s : v \rightarrow s$ of vertices to states in $\mathbf{S}$ where $|\mathbf{S}| = \sigma$, vi) a binding of arcs in $\mathbf{A}$ that extend across state boundaries to registers in $\mathbf{R}$, $f_r : a \rightarrow r$, and vii) an integer N. The problem is stated as follows: Is there a mapping of the vertices in the state graph $\mathbf{G}$ to the units in $\mathbf{U}$, $f_u : v \rightarrow u$, such that each unit is used at most once in any state $s \in \mathbf{S}$ and that the number of connections between registers and units is less than or equal to N? The second problem instance analyzed is as follows: Given a state graph $\mathbf{G} = (\mathbf{V}, \mathbf{A})$ and a set of registers $\mathbf{R}$, a set of function units $\mathbf{U}$, $f_u : v \rightarrow u$, and a positive integer N, is there a binding $f_r : a \rightarrow r$, of arcs in $\mathbf{A}$ to registers in $\mathbf{R}$ such that the number of connections between registers and units is less than of equal to N? Both the cases have been shown to be NP-Complete. The general decision problem has also been proved to be NP-Complete.

Chapter 3

Complexity of Scheduling in Data Path Synthesis

3.1 Introduction

The problem of scheduling is an important one in Data Path Synthesis (DPS). It is a primary problem in high-level synthesis of VLSI systems [13]. The scheduling problem surfaces soon after the behavioural specifications have been converted to the intermediate form which, as mentioned in chapter 1, is usually in the form of a set of data flow graphs. These may be in the form of directed acyclic graphs (DAG) which contain the dependencies between the operations. A DAG contains nodes which denote the type of operation to be performed (like +, - , etc.) and the precedence constraints on the operations. The DAG's are themselves connected by directed edges to form a flow graph, which depicts the flow of control between them. Scheduling is applied to each DAG in the flow graph. The purpose of scheduling is to assign the operations to time steps, satisfying the precedence constraints. In DPS, scheduling may precede or follow the allocation of hardware. Classical treatment of scheduling recognizes a two fold classification of the hardware. One class is referred to as *functional units* (f.u.) which directly implement the operations (in the DAG's). The other class is usually referred to as *resources* which are required to perform additional functions with the operations. In DPS such f.u.'s may be individual operators or in general ALU's and resources may be memories, system ports, buses and so on.

The scheduling problem has several variations. One important class of scheduling problems involves minimization of the number (or cost) of hardware operators given a deadline for finishing all the tasks. Another class concerns the minimization of the schedule length given a limit on the number and type of f.u.'s. There are some situations where scheduling must take into account resource constraints as well, like the presence of a limited number of memories, buses, system ports, etc. There are other formulations of scheduling where both f.u.'s and resources may be minimized. Several heuristic algorithms have been proposed for such types of problem formulations [50, 51] in VLSI design. There are also a few approximate algorithms with non-trivial error bounds. Scheduling problems, in general, have been hard to solve optimally in the sense that many of them have been proved to be NP-complete. This means that there is little or no chance that a polynomial time algorithm exists [49] to obtain the optimal cost solution.

The complexity of a hard problem is usually characterized in the following way. On the one hand we attempt to find out the simplest subproblem which is NP-hard, and on the other we search for polynomial time solutions to the most complicated subproblem possible. Finding out both of these is quite difficult and the closer one comes to it, the better is the characterization of the problem. The present study of complexity of the scheduling problem of DPS in this work has been in the above mentioned direction with special emphasis at differentiating it from the classical problem of scheduling single operation DAG's. This chapter addresses the issue of complexity of the scheduling problem in Data Path Synthesis.

Extensive studies have been made on the complexity and algorithms for problems in job shop scheduling [52, 32]. The scheduling problems in DPS share some common features. A common feature of scheduling for DPS is the presence of precedence constraints and the non-preemptive character of the operations. Also the execution times of the operations are sometimes integral, rather than unit. Operations may be of one type or of multiple types. The functional units (f.u.), may be simple, i.e., capable of realizing only one type of operation or complex, i.e., capable of realizing multiple types of operations. They may also be heterogeneous, i.e., two f.u.'s having non-identical capabilities.

There are several complexity results in scheduling theory which concern scheduling with precedence constraints [32, 49, 53]. They are, however, mostly related to DAG's which have only one type of operation. It is known that for DAG's with only one operation type the problem of minimum length scheduling with one or two f.u.'s can be solved in polynomial time; in fact it can be done in linear time [53]. But the problem for arbitrary m $(m > 0)$ f.u.'s is NP-complete [47, 32, 49]. For a fixed number of f.u.'s the complexity issue is still open. However, if the DAG is a forest (a set of disjoint trees) then the problem can be solved in polynomial time [49].

On the other hand, in VLSI scheduling we usually have more than one type of operation in the DAG's and the operators (or functional units) may either perform a particular operation or a set of them. If all operators are capable of performing all types of operations then the problem reduces to the original scheduling problem mentioned above. However, this is usually never the case. Moreover, in many situations where shared resources exist, resource constrained scheduling is an important problem to be tackled. For example, if several operations have to read data form a read only memory (ROM) having only one port we have a case of resource constraint.

In this chapter the complexity issue of scheduling multiple operation DAG's on heterogeneous f.u.'s has been considered. Specifically, we have considered the simple case where the DAG is a collection of chains (which are DAG's where every node has at most one immediate predecessor and at most one immediate successor), there are two types of operations and two f.u.'s, one for each type of operation. This problem has been shown to be NP-complete for the general case of m $(m > 2)$ chains. For $m = 1$, the problem is trivial, for $m = 2$, a polynomial time solution has been shown to exist. It may be noted that the problem of scheduling DAG's on homogeneous f.u.'s can be solved in polynomial time for up to two f.u.'s.

The above problem has been identified as a special case of a host of other scheduling problems, all of which have thus been shown to be NP-hard. In section 3.3, the proof technique used to derive the previous result has been used to derive the complexity of two other problems which are: i) Schedule length minimization of rooted binary trees of two operations using two f.u.'s, one of each type; and ii) Schedule length minimization of single operation chains in the presence of a single resource which an operation either uses wholly or not at all, during the period of its execution. A previous result for the second problem (in [32]) states that this problem is NP-hard for a DAG.

Subsequent to this, the complexity of approximation of various scheduling problems concerning schedule length minimization as well as f.u. minimization have been considered. A few open problems have also been discussed.

While working with designs involving multiple basic blocks (b.b.) another problem that often needs to be solved is that of scheduling variable to variable data transfers. Multiple b.b.'s will result when the behavioural description includes branching and looping constructs. Variable to variable data transfers may be required for straight line code behavioural specifications too. It has been shown that the problem of scheduling such transfers is NP-complete. An absolute approximation scheme for this problem has also been shown to be NP-complete.

The organization of the chapter is as follows. In section 3.2 we prove the main result on the NP-hardness of scheduling two operation chains on two f.u.'s. In the next section we examine some results that are derivable from the main result. These include the scheduling of rooted binary trees, resource constrained scheduling and scheduling only two chains involving two operations on two f.u.'s. In section 3.4 we examine the complexity of approximation schemes for scheduling DAG's to minimize the schedule length as well the number of f.u.'s (separately). In section 3.5 we examine the complexity of performing variable assignments subject to constraints imposed by the available hardware.

3.2 The Complexity of Scheduling Two Operation Chains

3.2.1 The Problem

We are given a set of tasks $t_1, t_2, \ldots, t_n$ each of which takes one time unit. The tasks are of two types, 1 and 2. There are two f.u.'s, one of each type. An operation can execute only on the f.u. of its type. There is a precedence constraint on the tasks which is restricted to be a collection of chains. There is a deadline D for completion of all the tasks. We wish to solve the decision problem of whether all the tasks can be scheduled on appropriate f.u.'s satisfying the precedence constraints and meeting the deadline D.

Figure 3.1: A Ring chain and a Ring Slot chain.

3.2.2 The Reduction

The reduction is from Exact Cover by three-sets [49]. Given a set A=$\{a_1, a_2, \ldots, a_n\}$, n is divisible by 3, and a collection of sets $B_1, B_2, \ldots, B_n$ where each B_i is a subset of A and has three elements, we construct a graph G consisting of a set of $(n+1)$ chains of nodes of two types 1 and 2. The total number of nodes is $6n^2 + 42n$. We then show that G has a schedule of length $3n^2 + 21n$ if and only if there are $n/3$ sets in $B_1, B_2, \ldots, B_n$ whose union is A.

We use a reduction technique which is similar to the one proposed by Berger and Lenore [53] for proving the NP completeness of scheduling with $<$ and $=$ constraints for DAG's. A chain of the graph is defined by a sequence of the type 1-1-2-1 which means a type 1 node followed by a type 1 node followed by a type 2 node followed by type 1 node. We define the following types of chain structures. These structures are illustrated in figures 3.1 and 3.2. In these figures a type 1 operation is represented by a single circle, while a type 2 operation is represented by two concentric circles.

Ring This consists of three nodes of the type 1-1-2.

Ring Slot This consists of three nodes of the type 2-2-1. (Note that Ring Slot is the dual of a Ring in the sense that they can be scheduled in tandem.)

Key We shall have n different types of Keys, one for each element a_i. Each Key consists of $n+6$ nodes. The first five nodes of a Key are of the pattern 1-1-1-2-2. The next $n+1$ nodes are all of type 2 except the i-th one which is 1 for the i-th Key.

Key Slot This is the dual of the Key. There are again n different such types. Each consists of $n+6$ nodes. The first five are of the type 2-2-2-1-1. The next $n+1$ nodes are all of type 1 except the i-th one which is 2 for the i-th Key.

We now describe the set of chains formed by the reduction. The constraint forest consists of a set of $(n+1)$ chains. For each set $B_i = \{a_{k_i}, a_{l_i}, a_{m_i}\}$, $k_i < l_i < m_i$, in the set cover problem we have a chain C_i which consists of a Ring (which we call the header of a chain) followed by three Keys of the types k_i, l_i and m_i respectively. This construction ensures that the Key chains corresponding to a_{k_i}, a_{l_i}, and a_{m_i} of B_i must be

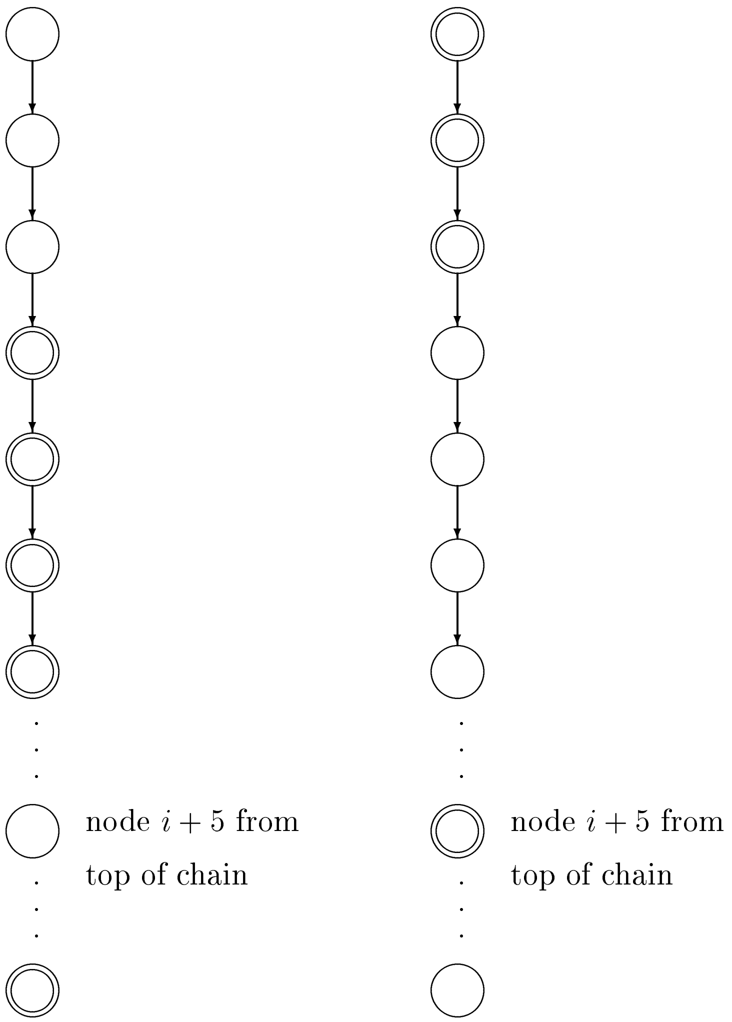


Figure 3.2: A Key chain and a Key Slot chain for a_i.

scheduled in that order. This gives us n chains. The final structure, i.e., the $(n+1)$-th chain, is the special chain called *Time Line* consisting of $3n^2 + 21n$ nodes having the following four conceptual stages:

Stage 1:[READY] It consists of a chain of n/3 Ring Slots. This is followed by

Stage 2:[FIT] It consists of a Key Slot for each $a_1, a_2, \ldots, a_n$ in that order. This is followed by

Stage 3:[RELEASE] It consists of a chain of 2n/3 Ring Slots. This is followed by

Stage 4:[PACK] This consists of a set of 2n Key Slots. For each a_i (in the order $a_1, a_2, \ldots, a_n$) there are $d_i - 1$ Key Slots of type i where d_i is the number of sets among $B_1, B_2, \ldots, B_n$ where a_i occurs.

The length of the Time Line is $3n^2 + 21n$. We define $D = 3n^2 + 21n$. We may note here that by the above construction, we are asking for an "exact schedule" which implies that at no time step will we have any f.u. idle. Based on the above construction we now prove that the chain scheduling problem is NP-complete.

Lemma 3.1 *If the set A has an exact cover by 3 sets, then we can schedule the chains within the deadline D.*

Proof: We perform the scheduling in the following manner. We have to schedule the Time Line as it has been given. We first schedule the headers (the Rings) of those chains C_i which correspond to the sets B_i occurring in the exact cover in stage 1. These chains are then scheduled perfectly in stage 2 of the Time Line. The remaining chain headers are scheduled in stage 3 and the rest of the chains are scheduled in stage 4. □

Lemma 3.2 *If we can schedule in the given deadline D then A has an exact cover by 3 sets.*

Proof: We shall prove this by showing that the only way in which we can obtain a schedule of length D (if it exists) is the manner in which it has been described in the previous lemma. We shall show that if there is a deviation from this then we cannot get a schedule of length D in the sense that we will never obtain a perfect schedule. We shall prove it case by case for every stage considering the first place where deviation occurs.

Consider stage 1. This is a sequence of n/3 2-2-1 chains (Ring Slots). Suppose it is correctly scheduled up to the $(i-1)$-th such Ring Slot. Thus, till the end of the $(i-1)$-th Ring Slot, where a type 1 node occurs in the Time Line, we must have exactly $(i-1)$ Rings from the headers of the chains $C_1, C_2, \ldots, C_n$ scheduled. Now consider the i-th Ring Slot (2-2-1) at this stage of the Time Line. If for the 2-2 portion of this chain we have a deviation then there may occur four cases. The *first* is where two type 1 nodes from headers of two different C_i's are scheduled. The *second* is one where two type 1 nodes are scheduled one from a header and another from the first 1 of a ready chain. (A ready chain is one whose header Ring is already scheduled). The *third* case is one where two type 1 nodes from the same ready chain is scheduled and the *fourth* case is one where two type 1 nodes from two different ready chains are scheduled. In each of the cases we can show that the type 2 f.u. corresponding to the Slot adjacent to the 1 in the 2-2-1 sequence of the i-th Ring Slot in the Time Line (in stage 1) will go empty (that is no task will be schedulable here). Thus we will not get a perfect schedule.

If there is no deviation for the 2-2 portion of the 2-2-1 chain then the 1 portion must also be free of deviation.

Consider stage 2. Similarly, here we will again assume that the schedule is correct (that is as mentioned in the proof of Lemma 3.1) up to the Key Slot corresponding to a_{i-1}. Let it deviate at the Key Slot corresponding to a_i. Now consider the 2-2-2 part of the Key Slot of this stage, in the Time Line. The following deviations might occur.

1. Only type 1 nodes from more than one Key are scheduled on the idle type 1 f.u. in these three time steps. In this case the type 2 f.u. in the first time step of the succeeding 1-1... part of the Time Line, will go idle.

2. Exactly one type 1 node from a Ring of one of the remaining 2n/3 chains is scheduled and type 1 nodes from one or two ready chains are scheduled. The type 2 f.u. in the next time step goes empty.

3. Two type 1 nodes from a Ring of one of the remaining 2n/3 chains and one type 1 node from a ready chain Key are scheduled. Only the type 2 node of the Ring is available for scheduling on a type 2 f.u. Depending on whether it is scheduled the next time step or the next to next time step, the type 2 f.u. goes empty in one of the two time steps.

4. Three type 1 nodes, one each from the Rings of three of the remaining 2n/3 chains are scheduled. The type 2 f.u. goes empty in the next time step.

If there is no deviation in the 2-2-2 part of the Time Line then there is no scope for deviation of the subsequent 1-1-...-1 part till the appearance of the first 2. It is clear that any deviation will cause the type 1 f.u. to go empty when the 2 appears on the Time Line. Again there is no scope of deviation in the remaining 1-1-...-1 part of this Key Slot. As explained while considering stage 1, if a f.u. goes empty then the schedule cannot be perfect.

In the absence of any deviation in the READY and FIT parts of the Time Line, the exact cover may be easily extracted. The arguments for the other two stages follow along similar lines. □

We are therefore ready to state the resultant theorem.

Theorem 3.3 *The problem of scheduling a set of chains corresponding to two different types of operations in two f.u.'s (one for each type of operation) given a deadline D is NP-complete.*

Proof: That this problem is in NP can easily be shown. That it is NP-hard follows from the construction and lemmas 3.1 and 3.2. □

We now present a small example to illustrate the generation of an instance of a scheduling problem from an instance of an exact problem.

Example 3.1 Consider the set $A = \{abcdef\}$, and the sets $B_1 = \{abc\}, B_2 = \{acf\}, B_3 = \{bcd\}, B_4 = \{cdf\}, B_5 = \{ace\}$ and $B_6 = \{def\}$.
For each B_i we construct a chain C_i as follows:
$C_1 =$ 1.1.2. 1.1.1.2.2. 1.2.2.2.2.2.2. 1.1.1.2.2. 2.1.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2,
$C_2 =$ 1.1.2. 1.1.1.2.2. 1.2.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2. 1.1.1.2.2. 2.2.2.2.2.1.2,
$C_3 =$ 1.1.2. 1.1.1.2.2. 2.1.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2. 1.1.1.2.2. 2.2.2.1.2.2.2,
$C_4 =$ 1.1.2. 1.1.1.2.2. 2.2.1.2.2.2.2. 1.1.1.2.2. 2.2.2.1.2.2.2. 1.1.1.2.2. 2.2.2.2.2.1.2,
$C_5 =$ 1.1.2. 1.1.1.2.2. 1.2.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2. 1.1.1.2.2. 2.2.2.2.1.2.2 and
$C_6 =$ 1.1.2. 1.1.1.2.2. 2.2.2.1.2.2.2. 1.1.1.2.2. 2.2.2.2.1.2.2. 1.1.1.2.2. 2.2.2.2.2.1.2,

The Time Line T is as follows:
$T =$ 2.2.1. 2.2.1.
2.2.2.1.1. 2.1.1.1.1.1.1. 2.2.2.1.1. 1.2.1.1.1.1.1.
2.2.2.1.1. 1.1.2.1.1.1.1. 2.2.2.1.1. 1.1.1.2.1.1.1.
2.2.2.1.1. 1.1.1.1.2.1.1. 2.2.2.1.1. 1.1.1.1.1.2.1.
2.2.1. 2.2.1. 2.2.1. 2.2.1.
2.2.2.1.1. 2.1.1.1.1.1.1. 2.2.2.1.1. 2.1.1.1.1.1.1.
2.2.2.1.1. 1.2.1.1.1.1.1. 2.2.2.1.1. 1.1.2.1.1.1.1.
2.2.2.1.1. 1.1.2.1.1.1.1. 2.2.2.1.1. 1.1.2.1.1.1.1.
2.2.2.1.1. 1.1.2.1.1.1.1. 2.2.2.1.1. 1.1.1.2.1.1.1.
2.2.2.1.1. 1.1.1.2.1.1.1. 2.2.2.1.1. 1.1.1.1.2.1.1.
2.2.2.1.1. 1.1.1.1.1.2.1. 2.2.2.1.1. 1.1.1.1.1.2.1.

The first line of T is the *ready* part, the next three lines correspond to the *fit* part, the fifth line is the *release* part, while the remaining lines correspond to the *pack* part. In this example B_1 and B_6 exactly cover A, this is also reflected in the existence of an exact schedule where C_1 and C_2 are scheduled in the *ready* and *fit* part of the Time Line, while the remaining chains are scheduled in the *release* and *pack* part of the Time Line. □

3.3 Related Results

Theorem 3.3 implies several other results. It follows from the theorem that scheduling chains of k types of operations (k-operation chains), $k \geq 3$, on k f.u.'s, one of each type, is NP-hard. This can be proved by simply augmenting the construction used for the proof with $k-2$ f.u.'s for the new types of operations. Now only the first two f.u.'s are required not to go idle in any time step. We do not elaborate here. Also if jobs have un-equal processing times, the problem is also NP-hard. However, since results similar to these generalizations (but not to the original 2-operation chain scheduling problem) are already available in scheduling theory [52, 32, 49], we will not discuss them here. We will discuss some other results which can be derived from the construction used and the result proved in theorem 3.3.

The corresponding optimization problem of finding the smallest length schedule is, therefore, also NP-hard. Since this problem is hard it becomes obvious that even when the f.u.'s are mixed (in the sense they can perform a set of operations) the problem is NP-hard. It is, therefore, a direct corollary that the problem of optimally scheduling a single two operation tree (or DAG) using two f.u.'s, one of each type, is NP-hard. This can be constructed from the previous problem of chains by constructing a single tree or a DAG from the chains.

Scheduling rooted binary trees of two operations on two f.u.'s

Since most operations are binary operations, we consider the special case when the DAG is a binary tree. We consider dependencies which satisfies the following properties. A node can have up to two successors and all but the a designated node called the root node will have exactly one predecessor. The root node does not have any predecessor. The corresponding graph of such dependencies will be referred to as a rooted binary tree. For the case of scheduling a rooted binary tree the following result may be obtained.

Theorem 3.4 *The problem of scheduling a single rooted binary tree of two operation types on two f.u.'s, one of each type, is NP-complete.*

Proof: A rooted binary tree may, in fact, be constructed from the chains that have been described above, as follows. Let the root node of the tree be R. R has two successors T_1 and C_1. A node T_i has one successor T_{i+1}, if $i < n$. The successor of T_n is the first node of the Time Line. A node C_i has two successors, C_{i+1} and the first node of the i-th chain, if $i < n$. The successor of C_n is the first node of the n-th chain. The nodes T_i, $1 \leq i \leq n$ are of type 1. The nodes C_i, $1 \leq i \leq n$ are of type 2. The root node is arbitrarily chosen to be of type 1. The deadline is now taken as $D' = 3n^2 + 22n + 1$. As before a schedule is attempted on two f.u.'s. In all schedules of this rooted binary tree, the type 2 f.u. in the first time step, where the root node is scheduled, will remain idle. In the lines of lemma 3.1 and lemma 3.2, the hardness result can be proved by considering a perfect schedule in the remaining $D' - 1$ time steps of the deadline. □

In practice we are likely to encounter such dependency structures where the dependency relation is just the reverse of a rooted binary tree. Such a dependency structure may be referred to an inverted binary tree.

Corollary 3.5 *The problem of scheduling an inverted rooted binary tree of two operation types on two f.u.'s, one of each type, is NP-complete.*

Proof: Follows along similar lines as theorem 3.4. □

It may be noted that single operation DAG's can be scheduled on up to two f.u.'s in polynomial time optimally.

Resource constrained scheduling

Next we consider a resource constrained scheduling problem (RCS). As mentioned earlier, the operators themselves are generally not considered to be resources. An important sub-problem in this category is that of *scheduling with two f.u.'s, unit execution times, and one resource with limit 1* [32]. In this problem some of the operations will use the resource while the others will not. As before the operations are considered to have unit execution times. A previous result for this problem (in [32]) states that this problem is NP-hard for a DAG. The method used to prove theorem 3.3 may be used to prove the hardness of this problem, for the case where the scheduling constraints are a set of chains.

Theorem 3.6 *Scheduling m chains having only one type of operation with two f.u.'s, unit execution times, and one resource with limit 1, when the precedence constraints are a set of chains, is NP-complete.*

Proof: The NP-hardness can be proved using a construction similar to the one used to prove lemma 3.2. While constructing the chains, now, a type 1 operation indicates that it does not need to use the resource, while a type 2 operation indicates that resource is used by this operation. The f.u.'s are identical because both type 1 and type 2 nodes perform the same operation. In the earlier problem only a type 1 and a type 2 node could be scheduled simultaneously. In the present problem, however, the only restriction is that two type 2 operations cannot be scheduled together, there being only a single resource. This leads to a slightly different proof. The cases considered here are similar, only the arguments used are different. We illustrate the treatment for stage 1 of the Time line, the other stages are handled in a similar manner.

The first stage is a sequence of $n/3$ chains of type 2-2-1. In line of the proof for lemma 3.2 we assume correct scheduling up to the $(i-1)$-th Ring Slot of this stage. If for the 2-2 portion there is a deviation then the same four cases considered in the proof of lemma 3.2 can occur. In the first case two type 1 nodes from the headers of two different chains are scheduled here. In the proof of lemma 3.2 it was shown that the type 2 f.u. (in that proof) would go idle because only a type 1 node would be ready for scheduling. Here too only a type 1 node will be available for scheduling and it can be scheduled with the type 1 node scheduled from the Time Line. Nevertheless, this situation is not tenable for the following reason. For every type 1 node in the Time Line there is a type 2 node for all the other chains considered together. Similarly, for a type 2 node in the Time Line there is a type 1 node in all the other chains taken together. We note that two type 2 operation cannot be scheduled in the same time step. *This requirement precludes the scheduling of two type 1 operation in the same time step.* Thus the proof can be easily completed in lines of the proof of lemma 3.2.

It is easy to see that the problem is in NP. Hence we conclude that the problem is NP-complete. □

A practical version of RCS comes up in DPS scheduling of DAG's where input/output needs to be performed. Due to pin limitations it becomes necessary to restrict the number of ports in the system. The port, now, is a system resource, some of the operations that need to do input/output will use these resources while the others will not. If there is only one port then we are directly faced with the 0/1 RCS problem described above.

Scheduling only two 2-operation chains on two f.u.'s

We now consider the problem of scheduling 2-operation chains using one f.u. of each type where we have exactly two such chains. We show that this problem can be solved optimally using an $O(n^2)$ algorithm.

Let the two chains be $X = \ < x_1, x_2, \ldots, x_m >$ and $Y = \ < y_1, y_2, \ldots, y_{m'} >$ where $m + m' = n$ and $t(x_i)$ or $t(y_i)$ denotes the operation type associated with the respective node.

Observation 3.7 *The problem of finding a minimum length schedule of two chains using two f.u.'s, one of each type, is equivalent to finding the maximum compatible subsequences of the two chains. Two nodes are said to be compatible if $t(x_i)$ is not equal to $t(y_j)$.*

The observation is due to the fact that we map the nodes on the maximum compatible subsequences to the same time steps. Then the two chains can always be scheduled in $m + m' - C$ steps where C is the total length of the maximum compatible subsequences of X and Y. In fact, for any schedule, in any time step where both the f.u.'s are busy, the pair of nodes of the two chains which are mapped on the f.u.'s are of different types and can participate in the formation of a compatible subsequence. Therefore, no schedule is possible where both the f.u.'s are busy for C' steps, where $C' > C$, because this would violate the premise that the length of the maximum compatible subsequences is C.

The compatibility problem may be solved using the following recursive decomposition, similar to that used in [54] for the Longest Common Sequence (LCS) problem:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } t(x_i) <> t(y_j) \\ max\ (c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } t(x_i) = t(y_j) \end{cases}$$

where c[i,j] is the length of the maximum compatible subsequences of $< x_1, x_2, \ldots, x_i >$ and $< y_1, y_2, \ldots, y_j >$.

This recursion equation can be solved by dynamic programming in O(mm') time using the algorithm given in figure 3.3 in pseudo code form. This shows that the two chain problem is solvable in polynomial time. However, the problem of r-chains for a fixed r $(r > 2)$ still remains open.

```
procedure find_max_compact(X, Y)
{
    m:= length(X)
    m':= length(Y)
    for i:= 1 to m do c[i,0]:=0
    for j:= 0 to m' do c[0,j]:=0
    for i:= 1 to m do
        for j:= 1 to m' do
            if x_i <> y_j then c[i, j] := c[i-1,j-1] +1
            else
            if c[i-1,j] > c[i,j-1] then
                c[i, j] := c[i-1,j]
            else   c[i, j] := c[i, j-1]
}
```

Figure 3.3: Algorithm for finding the maximum compatible subsequences.

3.4 The Question of Approximation and Other Open Problems

The result proved in Theorem 3.3 makes it quite apparent that in practice optimality has to be discarded if we are looking for polynomial time solutions. We therefore have to rest content with getting sub-optimal solutions in polynomial time. But here again we wish to seek guarantees in the solution. Such algorithms are known as approximation algorithms, where we usually seek a solution whose cost bears a definite relation to the cost of the optimal solution. For example, the cost of the approximate may not exceed the cost of the optimal solution multiplied by a constant value.

We know that list scheduling provides a very good bound on the quality of solutions in the sense that the schedule it provides for single operation type DAG's, given p f.u.'s, never takes more than twice the number of time steps of the optimal [34]. In the case of DAG's with k types of operations, however, with p f.u.'s for each operation type, if we use list scheduling then we can do the following:

1. We may convert the DAG to a single operation type one.
2. Schedule the DAG by list scheduling with kp f.u.'s to obtain a list schedule of no more than twice the optimal length.
3. Consider those time steps where more than p nodes of the same type have been scheduled in the same time step.
4. Sequence these nodes in different time steps using no more than p f.u.'s of a particular type at one time step.

This way we can schedule a k operation type DAG using p f.u.'s of each type in no more than $2k$ time steps of the optimal schedule. However, can we get a bound which is better than the above obvious generalization of list scheduling, especially a bound which is independent of k? This remains an open problem.

The other important problem is in the area of minimization of the number (or cost) of f.u.'s. In such cases we wish to find algorithms which can schedule in a given number of time steps with minimum number of f.u.'s. Can we obtain an approximate scheme which requires no more than a small constant times the minimum number required? Very interestingly, even in the case of single f.u. DAG's such results have not come to our notice. Even for popular algorithms like FDS [50] there are no known theoretical bounds on performance.

Consider DAG's with two types of operation for which an optimal schedule, exists in D time steps using two f.u.'s, one for each type of operation. We can obtain a schedule in D time steps with four operators, two of each type, as follows. We can convert the DAG to a single operation type one. Then we can optimally schedule this in linear time using the HLS algorithm [53] of Gabow. This will give a schedule in time $T \leq D$. Using two more f.u.'s, one for each operation type, we can easily find a schedule within D steps as required. However, the general case remains quite open if really strong bounds are to be obtained.

There is another type of approximation which is known as absolute approximation. In such cases we expect to obtain solutions which are bounded by opt + k where opt is the optimal and k is a constant. These are more difficult to obtain and in this particular case it is not difficult to show that they are NP-hard for nearly all types of scheduling problems.

Theorem 3.8 *Absolute approximation of scheduling DAG's is NP-hard for the problem of minimization of schedule length.*

Proof: Suppose a k approximation exists for the schedule length minimization problem. Take any DAG D. Make $k+1$ copies of it. Chain them up so that they must be scheduled one after another. Set the number of f.u.'s to m > 2. If this gives a k absolute approximate schedule then at least one of the $k+1$ copies of D must have been scheduled optimally for m f.u.'s. This proof is valid even for single operation type DAG's. □

Theorem 3.9 *Absolute approximation of scheduling DAG's with multiple operation types, given a deadline, is NP-hard for the problem of minimization of the number of f.u.'s.*

Proof: We take any DAG D, of a single operation type. Assume a k absolute approximate algorithm exists. Make $k+1$ copies of D resulting in a set of disjoint DAG's. Give each DAG a different operation type; all nodes of a particular DAG have the same operation type. Try to find a schedule satisfying the deadline. If a k absolute approximate solution exists then one of the DAG's must have been scheduled with the minimum number of f.u.'s. □

However, the complexity issue of absolute approximation of single operation DAG's where we wish to minimize the number of f.u.'s remains another open problem.

3.5 Complexity of Variable Assignment

3.5.1 The Problem

The behavioural specification for the high level synthesis of a digital system often consists of looping and branching constructs. These constructs give rise to numerous basic blocks (b.b.s) in the intermediate representation of the behaviour. It is seen that the first few statements of a basic block are often variable to variable assignments, which assign variables defined in other basic blocks to variables in the current basic block. Variable to variable transfers are also used to assign values that have been defined internally in the current b.b. There is a difference between the two which will become apparent as we explain the construction of the intermediate representation in the presence of variable assignments from externally defined values.

In the intermediate representation for each operation there is a node. The node contains the type of the operation (+, -, etc.), the sources and the destination. In the textual specification both the sources and destination are expressed as variable names. The destination variable name is annotated as a label in the node of the operation. While constructing such a node if this variable is already present as a label in one of the nodes constructed earlier then it is deleted from that earlier node. The labels indicate the specific variables that need to be assigned the value of that node. As the labels can get deleted, it is possible that during the construction of the intermediate representation, a node may be left with no label at all. The absence of a label simply means that there is no specifically designated variable to which the value of that node is to be assigned. In such a situation a new variable called a *temporary variable* [7] is put into the label field.

In order to identify the source of an operation or a pure variable assignment, it is necessary to identify the node which has the label corresponding to the source variable annotated to it. If this variable has been defined by an earlier operation in the current b.b. then we may be sure that a node carrying such a label will be found. However, for the first use of an externally defined variable such a label will not be found. In such a case a special node called an entry node is created. The entry node can be annotated with labels like operation nodes. It has a special field, the entry field, to indicate the variable which brings in a value into the current b.b. through this node.

An assignment "$a \leftarrow b$" is handled as follows. First a check is made if a happens to be in the set of labels of any node in the current basic block. It is deleted from that set if such a node is found. The label a is now augmented to the set of labels of the node that carries the label b.

We shall now restrict our attention to variable assignment statements which lead

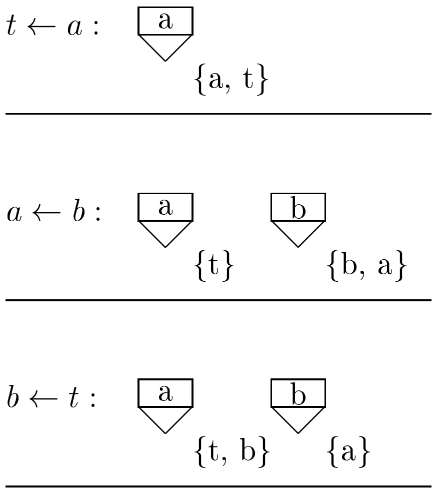


Figure 3.4: Development of entry nodes.

to the augmentation of the labels of the entry nodes only. Example 3.2 depicts the development of the entry nodes and their set of labels.

Example 3.2 Assume that a and b were defined outside the current basic block. Consider the following transfers corresponding to the interchange to the variables a and b within the current basic block.

1. $t \leftarrow a$
2. $a \leftarrow b$
3. $b \leftarrow t$

The developments in the entry nodes for a and b as these statements are processed is shown figure 3.4. In the figure an entry node is represented with a rectangle and a downward pointing triangle fixed to the base of the rectangle. The variable in the entry field is written inside the rectangle. □

For convenience the transfers implied by the variable entry node and its labels will be represented in a more explicit form as a directed graph as follows. Let S be the set of all the variables in the entry field and labels of each entry node. Construct a graph G where there is a node for each variable in S. Construct a directed edge from a node x to node y, in G, if y appears in the set of labels of the entry node for x. This edge represents a transfer from x to y and it is different from the precedence constraints discussed earlier in this chapter.

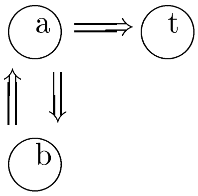


Figure 3.5: A transfer graph.

Each node with a successor in the transfer graph corresponds to the assignment of the value of the variable of that node to the variables corresponding to its successors. Actually a single node in the transfer graph could be associated with several transfers in the specification, as indicated in example 3.3.

Example 3.3 The following transfers could be represented in the node of figure 3.7.

1. $b \leftarrow a$
2. $c \leftarrow a$
3. $d \leftarrow a$
4. $e \leftarrow a$

□

The representation for the transfers in example 3.2 is indicated in figure 3.5. This example also serves to illustrate the formation of cyclic dependencies. A transfer to a variable, as indicated in the graph, cannot be scheduled before the transfers originating from the variable has been scheduled.

The cycles in the transfer graph pose a difficulty in scheduling these transfers. However, these cycles can be broken, along with the introduction of some additional transfers to consistently represent the original transfers. We do not explain this technique here but figure 3.6 illustrates the application of this technique to remove the cycles arising in example 3.2. This graph indicates the following sequence of transfers: $t \leftarrow b; b \leftarrow a; a \leftarrow t$. Though this sequence is not exactly the same as the original code sequence it is still guaranteed to correctly transfer the values.

Having indicated that the cycles can be broken, we now look into the details of scheduling the transfers. The variable to variables transfers are scheduled over the available system buses, to take place between the storage access points in the data paths. The number of buses and storage access points to be present in the data path are fairly important design parameters that may be specified by the design engineer. It

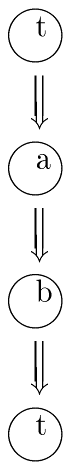


Figure 3.6: Cycle free transfer graph.

is, therefore, necessary to schedule the transfers on the specified number of buses using the specified number of storage access points. We shall now consider the complexity of scheduling such transfers using the available hardware configuration.

3.5.2 Complexity of Variable Assignment

To infer the complexity of scheduling the transfers in minimum number of time steps, we now examine a simple case when the transfer graph is a forest of trees of height one, corresponding to independent transfers. Consider a node in a transfer graph with

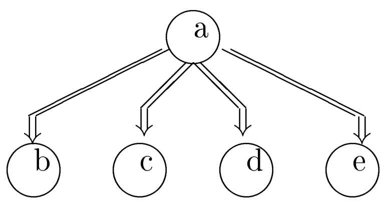


Figure 3.7: Transfer graph for transfer to multiple destinations.

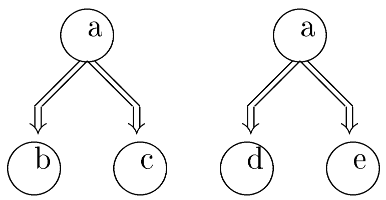


Figure 3.8: Split transfer graph.

k successors. This transfer could be carried out in a single time step over one bus and using $k+1$ storage access points. One storage access point would be required for the source and k others for the destinations denoted by each successor node. However, if sufficient number of storage access points are not available then the the transfer would have to be split over several time steps. For example, if only three storage access points are available then it would be necessary to split up the transfer and schedule them over two time steps, as indicated in figure 3.8. We shall now formulate the problem of scheduling these independent transfers as kind of bin packing problem which is equivalent to optimally splitting the transfer graph.

Consider bins of capacity M, M being equal to the number of storage access points. Our transfer graph consists of, say, N trees of depth one. Let the number of nodes with in-degree greater than zero (non-root nodes) in each tree be t_i. Consider the problem of packing N objects each of size t_i, $i = 1..N$, using the minimum number of bins of size M. While packing objects it is permissible to fragment the objects into integral units as desired. However, each object of size t, whether whole or after fragmentation, consumes a capacity of $t+1$ of the bin into which it is packed. The extra unit capacity is consumed because, as we have already explained earlier, a node with k successors takes up $k+1$ storage access points. The number of bins used corresponds to the minimum number of time steps in which the transfers can be scheduled. We now define the *fragmentable object bin packing* problem.

Definition 3.1 *Fragmentable object bin packing (FOBP) is the decision problem of packing N objects each of size t_i, $i = 1..N$, into m bins each of size M and the capacity used up while packing each object (whether whole or after fragmentation) of size k is $k+1$.*

Lemma 3.10 *The decision problem for fragmentable object bin packing for two bins (FOBP2) is NP-hard.*

Proof: We prove the lemma by reducing the partition problem to FOBP2 in two steps.

Step 1: Consider the partition problem where given N objects with integer weights w_i $(w_i > 0)$, it is necessary to determine whether it is possible to partition these into two sets whose total weights are the same. Given an instance P of the partition problem we construct another instance $P2$ of partition where the weight of each object in $P2$ is twice the weight of the corresponding object in P. Clearly P has a partition if and only if $P2$ has a partition.

Step 2: Given a problem $P2$ we construct an instance of FOBP2 as follows. For an object of $P2$ of weight w_i there is an object in the instance of FOBP2 of weight $w_i - 1$. There are two bins for packing the objects. Let $W = \sum w_i$, where w_i is the weight of each object of $P2$. Choose the size of each bin of FOBP2 as $\frac{W}{2}$.

Clearly $P2$ has a partition if and only if the constructed instance of FOBP2 can be packed into the two bins. □

We can now easily prove the following result.

Corollary 3.11 *FOBP is NP-hard.*

FOBP is a special case of the variable assignment problem, and this leads to the following result.

Theorem 3.12 *The problem of scheduling variable assignments in minimum number of steps is NP-complete.*

Proof: It is easy to see that this problem is in NP. That it is NP-hard follows from corollary 3.11. □

The bin packing problem posed here is interesting because if the restriction of consuming a unit additional capacity for packing each fragment is nullified then the packing may be done optimally. On the other hand if fragmentation is not permitted then we get the conventional bin packing problem, for which an approximate algorithm exists with the relative error being bounded by a constant and for which an approximation scheme with the absolute error being bounded by a constant is NP-hard.

We now show that an approximation scheme for the fragmentable object bin packing problem with the absolute error being bounded by a constant is NP-hard. Let OPT be be number of bins required to solve the problem optimally. Now consider a problem instance when each of these OPT bins would be fully packed without any fragmentation of the objects. Let the approximate algorithm solve the problem using $OPT + k$ bins. The additional capacity introduced by the k bins is Mk units of bin capacity. Any fragmentation made by the approximate algorithm would consume at least two units of this additional space.

Now consider another problem instance derived by replicating the above problem instance $\lceil \frac{MK}{2} \rceil + 1$ times. If this problem is to be solved using k bins in addition to $(\lceil \frac{MK}{2} \rceil + 1)OPT$ bins, then at least one of these instances would have to be solved optimally. This leads to the following theorem.

Theorem 3.13 *An absolute approximation scheme with the error being bounded by a constant, for the fragmentable object bin packing problem is NP-hard.*

Corollary 3.14 *The absolute approximation scheme for the problem of scheduling variable assignments in minimum time is NP-complete.*

Proof: It may be shown that this problem is in NP. Theorem 3.13 indicates that this problem is NP-hard. □

3.6 Conclusion

In this chapter we raised the issue of the complexity of scheduling in high-level synthesis and the approximation algorithms. The problem is different from normal scheduling problems in the sense that there are different types of operations which cannot be assigned to all f.u.'s. We have proved a result which states that even in a simple of chains and two f.u.'s, the problem is NP-hard. A dynamic programming algorithm is provided for the two chain scheduling problem. The problem of 0/1 resource constrained scheduling on single operation chains has also been shown to be NP-hard. We have also shown that absolute approximation of scheduling is NP-hard for both schedule length minimization or f.u. minimization. In the case of approximations we give a simple relative approximation using the well known list scheduling technique and an even better bound for two f.u.'s (section 3.4). However, we acknowledge that these algorithms are rudimentary, if not obvious, and stress the need for having much more improved bounds and deeper results. In the case of f.u. minimization no proper bounds have come to our notice. Though algorithms which appear to perform well have been proposed [50, 23] there has been no proper theoretical analysis of their performance. This remains a very important area of future work with several open questions remaining, some of which have been listed in this chapter. In addition to the conventional scheduling problem we have also studied the variable assignment problem which is bound to come up in practical situations where high level synthesis techniques are applied. We have shown that this problem and its absolute approximation scheme are both NP-hard.

Chapter 4

Complexity of Allocation and Binding

4.1 Introduction

The complexity of scheduling has been examined in the previous chapter. Scheduling is one of the many problems which have to be solved in order to get an optimized data path for a target system. The other problems which have to be addressed for data path synthesis are: *a*) functional unit (f.u.) formation, *b*) interconnect formation, *c*) memory allocation, and *d*) register optimization. The port assignment problem is an important sub–problem of the memory allocation task. Together these make up a major part of the allocation and binding problem whose complexity we examine in this chapter.

The overall problem of allocation and binding is NP-hard because some of its sub-problems are already known to be NP-hard. The NP-hardness of the register minimization problem is one of the oldest complexity results in this area. The register minimization problem has been formulated as a clique partitioning problem in [17] which is a standard NP-complete problem. The complexity of connectivity binding has been examined in [31].

In our study of the complexity of allocation and binding we consider several new problems which are as follows.

- The port assignment of dual and triple port memories (PA).
- Register–interconnect optimization (RIO).
- The problem of functional unit formation (FUF).

We have shown that these problems are NP-complete. In our study we not only examine the complexity of optimally solving the above mentioned problems, but also seek to establish the complexity of finding an approximate solution wherever possible.

In general for a NP-hard optimization problem we do not expect to find a polynomial time algorithm to solve that problem optimally. The only known methods of obtaining optimum solutions (like branch an bound techniques) are exponential in time complexity. DPS problems which occur in practice are usually so complex that enumerative approaches like branch and bound are ruled out in many situations. The alternative approach is to relax the requirement of finding optimal cost solutions. Designers are often satisfied with a fast (polynomial time) algorithm provided it guarantees some error bounds on the cost of the solution. Such algorithms are known as approximation algorithms. There are two well known types of approximation algorithms, viz. the absolute approximation algorithm (which guarantees that the solution obtained will not differ from the optimal by more than a fixed constant) and the relative approximation algorithm (which guarantees that cost of the solution obtained by the algorithm will not exceed, in the case of minimization, the cost of the optimal by a constant factor). Absolute approximation algorithms are usually difficult to find for most NP-complete problems. We have already established the hardness of absolute approximation of several scheduling problems in the previous chapter. The second type of approximation algorithms, namely those of relative approximation, are obtainable for many problems where an absolute approximation algorithm is not obtainable. Relative approximation algorithms are available for many scheduling problems. For example, list scheduling guarantees that the solution obtained in the case of a single operation DAG is never more than twice the optimal schedule length.

In this chapter we shall show that while scheduling and allocation are both NP-complete, allocation appears to be more difficult than scheduling. This is because we shall establish that for nearly all the allocation problems that we consider the problem of finding a relative approximation itself is NP-complete. Throughout this chapter we shall use the following notation: for any problem X, (for which we require an optimal solution) X–R shall denote the problem of finding a solution whose relative error is bounded by a constant. X–R will also be referred to as a constant bounded relative approximation for X. When we say that X–R is NP-complete we mean that if there is a polynomial time approximate algorithm which guarantees a constant relative error bound for X, then $P{=}NP$. In this chapter we show that not only are most allocation problems NP-complete but also that constant bounded relative approximation of several versions of the PA and RIO are NP-complete.

The organization of the chapter is as follows. In section 4.2 we introduce the general node deletion problem. We show that this problem as well as its relative approximations are NP-hard. These results form the basis of most of the results derived later in this chapter. In section 4.3 we examine the port assignment problem for dual and triple port memories. It becomes necessary to solve these problems when multi-port memories are used as building blocks in data path synthesis. For the dual port case we show that the problem is NP-complete. For triple port memories we have proved that the port assignment problem and its absolute and relative approximations are all NP-hard. In section 4.4 we review the register optimization problem. In this section we introduce the register optimization problem for straight line code, which is known to be solvable in polynomial time [21]. In section 4.5 we generalize this problem to the register–interconnect optimization problem for straight line code (SRIO). This problem is naturally encountered for simple instances for DPS problems. Through this problem we prove that the interconnect optimization as well as its constant bounded relative approximation are NP-hard. The NP-hardness of SRIO is a significant result because the problem of register optimization for straight line code is solvable in polynomial time and the problem of general register–interconnect optimization is already known to be NP-hard.

In section 4.6 we examine a different problem, namely that of functional unit formation (f.u.f.). The f.u.f. problem comes up when the schedule of operations does not explicitly indicate the f.u. to which an operation should be mapped. This is often the case when specific f.u.'s are not available at the time of scheduling. During allocation and binding it is then necessary to find a suitable mapping of the operations to the f.u.'s. This mapping determines the capabilities and therefore the cost of the f.u.'s. We prove in particular that the problem of minimizing the cost of the f.u.'s as well as its absolute approximation are NP-hard.

4.2 General Node Deletion

The node deletion problems have been used, in this chapter, to establish the complexity of many of the sub–problems encountered during allocation and binding. The node deletion problem (ND) [55] is as follows. *Given a graph $G(V, E)$, V being the set of vertices and E the set of edges, identify a set of nodes N_d for deletion such that the graph, after deletion of nodes in N_d, is bipartite.* A graph is said to be bipartite if it can be partitioned into two independent sets. A graph is bipartite if and only if it can be coloured using two colours, i.e. it is 2-colourable. For the node deletion problem N_d should be the smallest possible set of such vertices. The node deletion decision problem (NDD) answers the question whether the given graph can be rendered bipartite by the deletion of m vertices, $0 \leq m \leq |V| - 2$. It has been shown that both ND and NDD are NP-complete [55].

The general node deletion problem (GND) is formulated as follows. Given a graph $G(V, E)$, identify a set of nodes N_d for deletion such that the graph, after deletion of nodes in N_d, is k–colourable. N_d should be the smallest possible set of such vertices. The corresponding decision problem (GNDD) is to answer the question whether the given graph can be rendered k–colourable by the deletion of m vertices, $0 \leq m \leq |V| - k$. Such a decision problem will be represented as GNDD(m, k).

Theorem 4.1 *The general node deletion decision problem (GNDD) is NP-complete.*

Proof: For 2–colourability GNDD is NP–Hard because NDD is NP–Hard. For k–colourability, $k \geq 3$, GNDD is NP–Hard because the corresponding chromatic decision problem is NP–Hard. It is easy to write a polynomial time non–deterministic algorithm [55] to solve GNDD. □

The following corollary follows easily.

Corollary 4.2 *GND is NP-complete.*

The question of polynomial time approximate algorithms for GND is now examined. An approximation algorithm for GND with constant relative error bound would be one which guarantees $d_r \leq (1 + \epsilon)d^\star$, for any instance of GND, where $d^\star$ is the number of vertices deleted by the optimal algorithm, d_r is the number of vertices deleted by the approximation algorithm for this approximation and $\epsilon > 0$ is a constant. We denote the problem of obtaining an approximation algorithm for GND whose relative error is bounded by a constant as GND–R. This is also referred to as the constant bounded relative approximation for GND. The following important result can be proved.

Theorem 4.3 *A constant bounded relative approximation for GND (GND–R) is NP-complete.*

Proof: Suppose that a polynomial time algorithm exists which guarantees that the relative error in the approximate solutions to GND is bounded by a constant. Let it be A_r. Let $G(V, E)$ be an instance of GND, such that the graph is k–colourable. Therefore $d^\star = 0$. This requires that A_r should report $d_r = 0$. If this is so, then A_r could be used to solve the chromatic decision problem (CDP) [55] in polynomial time. CDP being NP-complete, A_r will be such a polynomial time approximate algorithm for GND only if it is also an optimal algorithm for CDP. Thus GND–R must be NP-complete. □

The particular case for GND to ensure that a given graph will be made three colourable will be referred to as GN3D. The following result regarding a constant bounded relative approximation for GN3D follows from theorem 4.3, since the three colour decision problem is NP-hard. We shall use this result several times in the rest of this chapter.

Corollary 4.4 *GND–R for* 3*–colourability, GN3D–R, is NP-complete.*

4.3 Port Assignment

4.3.1 Prologue

The variables that are used to specify a behaviour need to be implemented as storage elements. These may be clustered into memory modules of one, two or three ports. Some work on port assignment has been reported in [56]. At this level of abstraction it is permissible to view inputs and outputs of components, as well as ports of memories as single points in the circuit. A point in a circuit is said to access a memory if it transfers data to or from a cell of the memory. Given a set of registers being placed in the memory along with the permissible number of ports and their capabilities, and the set of accesses to these registers; it is necessary to assign the accessing points (in the circuit) to the memory ports so that all the access will be satisfied. The assignment should be made in such a way that the resulting interconnect overhead is minimized. The assignment process is explained with a simple example.

Example 4.1 Consider the transfers given below.

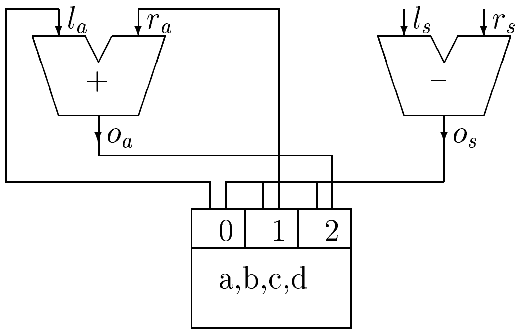


Figure 4.1: Connections to a three port memory.

1. $a = b + c$;
2. $q = c + d, \quad b = p - q$;
3. $d = a + r, \quad c = p - r$;
4. $a = p + c, \quad b = q - r$;

Suppose a, b, c, d, p, q, r and s are registers, of which only a, b, c and d are to be placed in the same memory. Assume that three ports are permitted, and the ports are labeled $0, 1$ and 2 respectively. It will be noted that at most three accesses are made to the memory in any time step. Suppose that an adder and a subtracter are used. Let the adder inputs be labeled l_a and r_a while the adder and subtracter outputs be respectively labeled o_a and o_s. It will be observed that l_a, r_a, o_a and o_s are the only four points accessing the memory in the various control steps. They need to be assigned to the ports suitably. Consider the assignment where l_a, r_a and o_a are mapped on ports $0, 1$ and 2 respectively, and o_s is mapped to all the ports $0, 1$ and 2. All the transfers can be satisfied using this assignment. The connections are illustrated in figure 4.1. It will be noted that a total of six switches will be required at the ports of the memory shown. □

It will be noted that a point in the circuit which must access a member of the memory under consideration, will be connected to at least one of its ports. To reduce interconnect overhead port assignment (PA) should be done so that the minimum number of points is connected to more than one port. When a point that reads from the memory is connected to $k \quad (k > 0)$ ports of the memory, each connection has to be switched through a multiplexer channel. Similar multiplexing is required when multiple sources are connected to a write port of a memory. For the port assignment suggested in example 4.1, o_s is mapped to all the three ports and so its connectivity is three, as against the desired level of one.

An algorithm that finds an optimal PA should lead to an interconnection that requires the least number of switches for multiplexing. In this section we shall deal with the PA

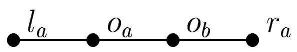


Figure 4.2: A conflict graph.

problem for dual and triple port memories. In each case we shall first introduce the problem. Following the strategy of chapter 3, we shall identify a special case of the corresponding PA problem to derive the complexity results.

The PA problem may have several variations, depending on the number and type of the ports. The ports may be uniform , being, read/write (rw) or purely read (r); or may have arbitrary capability, i.e. rw, r or w. PA for a single port is trivial. The cases for two and three ports, with uniform capability are of interest and are considered here.

4.3.2 Memories with Two Uniform Ports

A formulation is presented for this case assuming that the ports of the memory are uniform. Since only two ports are permitted, at most two accesses to the memory may be made in any time step. If the ports are only of type r then the accesses must be all reads. If the ports are rw then arbitrary combinations of accesses may be made. However, since both the ports are uniform, an access may be satisfied using any one of the two ports. It is necessary to perform an assignment of points that access the memory, to the ports so that all the accesses may be satisfied and the switching requirement is minimum.

The formulation of PA relates it to the node deletion problem in the following manner. If two points in the circuit access the memory in a particular time step, they must do so through different ports. Two such points are said to be in *conflict*. A conflict graph of points accessing the memory is defined as a graph where the points correspond to the vertices and an edge is present between a pair of vertices if their corresponding points are in conflict. An attempt is made to two colour this graph with colours, {0,1}, say. If the colouring is successful then vertices coloured 0 are connected to one port, and the vertices coloured 1 to the other. In general the graph may not be 2–colourable and it will be necessary to transform the graph in order to render it 2–colourable.

Example 4.2 Consider the transfers of example 4.1. The variables a, b and d may be packed into a 2-port memory. For such a packing the conflict graph is shown in figure 4.2. The nodes are labeled in the lines of example 4.1 and figure 4.1. Evidently the conflict graph for this packing is very simple. □

Now the close relationship between port assignment and the node deletion problem (ND) will be established. The necessary and sufficient conditions for a graph to be 2–colourable is that it should be free of cycles of odd length, that is, it should be bipartite [55]. If a set of vertices $v_1, v_2, ..., v_l$ form an odd cycle, then one of the points v_i corresponding to these vertices must be connected to both the ports of the memory.

It may also be noted that if v_a and v_b are points accessing the memory in the same time step and if v_b is connected to both the ports and v_a to any one port then the access conflict of v_a and v_b can always be resolved. This is because, no matter from which port v_a accesses the memory, v_b can always access the memory through the other port. Since a point connected to both the ports ceases to be in conflict with any other point, its corresponding vertex in the conflict graph may be deleted. Vertex deletion is the operation that will be used to render a graph bipartite.

The problem of finding the set of vertices to be deleted to have a bipartite graph as well as have minimum number of switches for the interconnection will be referred to as *p*ort *a*ssignment for *two u*niform ports (PA2U). In order to prove the complexity results of this problem we consider a special case of it which is as follows. *The accesses to the memory in question are only read accesses and the points that read from the memory do not read from or write to any other place in the circuit.* The first condition implies that multiplexers are required only at the circuit points and not at the ports. The second assumption ensures that the only lines incident at these circuit points are those coming from the memory ports. The advantage of considering this special case is that, now, the interconnection cost is highly simplified. The costs may be computed as follows. If a point is connected to a single point then no multiplexing is needed and the interconnect cost is zero. If, on the other hand, the point is connected to both the ports then a 2-to-1 multiplexer, will be needed. Such a multiplexer is essentially a set of 2 switches which operate in a mutually exclusive manner. The interconnect cost for a 2-to-1 multiplexer is then taken as 2. The objective is to minimize the total cost accrued in interconnection as a result of the port assignment. When we consider the node deletion formulation, the interconnection cost is twice the number of nodes deleted. It is quite obvious that the general case of PA2U is more complicated than this special case. This special case of PA2U will be referred to as PA2U1.

Theorem 4.5 *PA2U1 is NP-complete.*

Proof: In order to prove the hardness of PA2U1, a polynomial time reduction will be obtained from ND to PA2U1. Given any graph $G(V, E)$ which is a problem instance of the node deletion problem, transform it to PA2U1 as follows. Construct a circuit where there are points corresponding to the vertices in the graph. These points are chosen as inputs of hardware elements in a circuit. Corresponding to each edge (v_1, v_2), form a control step including two transfers, one each, between the memory and v_1 and v_2. This generates an instance of PA2U1. An optimal solution to this instance of PA2U1 can be used to obtain an optimal solution to the node deletion problem. It is necessary to delete those vertices from the graph whose corresponding point in the circuit are connected to both the ports of the memory. □

Since PA2U is a special case of PA2U1 the following result follows.

Corollary 4.6 *PA2U is NP-complete.*

The problem of solving ND, and therefore PA2U1, so that the relative error in the solution is bounded by a constant is an open problem.

4.3.3 Memories with Three Uniform Ports

Prologue

Given a set of registers which have been placed in a particular memory with three uniform ports, it is necessary to assign the circuit points accessing the memory such that *a*) all the accesses in each control step are satisfied and *b*) the cost of switches for interconnection to the ports is minimized. This problem will be referred to as PA3U. First a relaxed formulation PA3UA1 based on GND is presented in section 4.3.3. PA3UA1 will be proved to be as hard as GND. PA3UA1 will then be used to derive complexity results for PA3U.

PA3UA1 and Its Relationship with GN3D

We now consider a relaxed formulation of PA3U, to be referred to as PA3UA1, which is as follows. *The accesses to the memory in question are only read accesses, the points that read from the memory do not read from or write to any other place in the circuit and these points are connected to exactly one port or to all the three ports of the memory.* The first condition implies that multiplexers are required only at the circuit points and not at the ports. The second assumption ensures that the only lines incident at these circuit points are those coming from the memory ports. If a point is connected to a single point then no multiplexing is needed. If, on the other hand, the point is connected to k $(k > 1)$ ports then a k-to-1 multiplexer, will be needed. Such a multiplexer is essentially a set of k switches which operate in a mutually exclusive manner. The interconnect cost for a k-to-1 multiplexer is then taken as k. The third assumption ensures that either $k = 1$ or $k = 3$. If $k = 1$ then interconnect cost is zero and if $k = 3$ then cost is three. This special case, to be referred to as PA3UA1, will help us to use the complexity results of GN3D.

Given an instance of PA3UA1 we make the following construction. A graph is constructed from the set of transfers for the various control steps. If points p_1 and p_2 access the memory in the same control step then they must access the memory through distinct ports. In the graph an edge is introduced between the vertices corresponding to p_1 and p_2. If this graph is 3-colourable then a feasible assignment of the the points to the ports can be directly obtained. Otherwise, it will be necessary to connect some of the points to more than one port, to satisfy the memory accesses. Since we are working on an instance of PA3UA1, we shall connect such a point to all the three ports of the memory. All the conflicts for such a point can always be resolved, and the vertex corresponding to this point may be deleted from the conflict graph. To minimize the interconnect cost the number of such points should be minimized. Clearly this corresponds to the general node deletion problem to achieve three colourability (GN3D).

Complexity of PA3UA1

It will be shown in this section that PA3UA1 is just as hard as GN3D. Thus an approximation for PA3UA1 whose relative error is bounded by a fixed constant is NP-complete.

The transformation described previously leads to the following theorem.

Theorem 4.7 *The relaxed formulation, PA3UA1, of the port assignment problem of three port memories is NP-complete.*

Proof: Given a graph $G(V, E)$ for GN3D, an instance of PA3UA1 will be constructed as follows. Let r_1, r_2 and r_3 be registers packed into the given memory. For each vertex $v_i \in V$ let p_i be a point in the circuit accessing the memory. For each edge $(v_{i_1}, v_{i_2}) \in E$, find a vertex $v_{i_3} \in V$, if such a vertex exists, such that $(v_{i_1}, v_{i_3}) \in E$ and $(v_{i_2}, v_{i_3}) \in E$. If v_{i_3} exists construct the control step
$\mathbf{p_{i_1} \leftarrow r_1}, \quad \mathbf{p_{i_2} \leftarrow r_2}, \quad \mathbf{p_{i_3} \leftarrow r_3},$
otherwise construct the control step
$\mathbf{p_{i_1} \leftarrow r_1}, \quad \mathbf{p_{i_2} \leftarrow r_2}.$

This creates an instance of the port assignment problem with three ports. It is easy to see that the deletion of the vertices whose corresponding points are chosen for connection to all the three ports for conflict resolution will be a feasible solution to GND. This is because the remaining points are connected to single ports and their corresponding vertices may therefore be assigned three distinct colours.

It is easy to see that PA3UA1 is in NP. □

Now, a relative approximate algorithm for PA3UA1 may also be used as an approximate algorithm for GN3D with the same error. Such an approximation for PA3UA1 will be referred to a PA3UA1–R. Therefore from corollary 4.4 the following corollary follows.

Corollary 4.8 *An approximation for PA3UA1, PA3UA1–R, whose relative error is bounded by a fixed constant is NP-complete.*

Now the complexity of the original three port problem, viz. PA3U, will be analyzed and it will be shown that it is also as difficult as PA3UA1.

Complexity of PA3U

We first consider a special case of PA3U, PA3U1 which is as follows. *The accesses to the memory in question are only read accesses, the points that read from the memory do not read from or write to any other place in the circuit.* We note that PA3U1 and PA3UA1 differ only in the way the points may be connected to the ports of the memory. In the case of PA3UA1 a point may be connected to either one or all of the ports. Thus either no multiplexer is needed or a 3-to-1 multiplexer is needed. However, for PA3U1 the point may be connected to one, two or all three of the ports. In this case either no multiplexer is required or a 2-to-1 or a 3-to-1 multiplexer is required at that point. We prove the following theorems for PA3U1. These results directly carry over to PA3U, it being a generalization of PA3U1.

Theorem 4.9 *PA3U1 is NP-complete.*

Proof: Proved by reducing PA3UA1–R to PA3U1. Let $\mathbf{Y}^\star$ be the cost of the optimal solution to PA3U1 and $\mathbf{X}^\star$, the cost of the optimal solution to PA3UA1. Clearly $\mathbf{Y}^\star \leq \mathbf{X}^\star$. Let an optimal solution to PA3U1 consist of $p_2^\star$ connected to two ports and $p_3^\star$ connected to three ports.
Thus, $\mathbf{Y}^\star = 2p_2^\star + 3p_3^\star \geq 2p^\star$, where $p^\star = p_2^\star + p_3^\star$. Thus, $p^\star \leq \frac{1}{2}\mathbf{Y}^\star$.

The $p^\star$ points of an optimal solution to PA3U1 may be connected to all the three ports, so that this modified solution serves as an approximate solution to PA3UA1. Let $\mathbf{X}$, be the cost of this approximate solution to PA3UA1.
$\mathbf{X} = 3p^\star \leq \frac{3}{2}\mathbf{Y}^\star \leq \frac{3}{2}\mathbf{X}^\star$.

Thus the optimal algorithm for PA3U1 could be used to solve PA3UA1–R, proving that PA3U1 is NP–Hard. It is easy to show that PA3U1 is in NP. □

Corollary 4.10 *PA3U is NP-complete.*

Proof: NP-hardness of PA3U follows from lemma 4.9 because PA3U1 is a special case of PA3U. That it is in NP can be shown easily. □

Theorem 4.11 *An approximation for PA3U1, PA3U1–R, whose relative error is bounded by a fixed constant is NP-complete.*

Proof: Let $\mathbf{Y}$ be the cost obtained by an algorithm for PA3U1–R. Let $\mathbf{Y}^\star$ be the cost of the optimal solution to PA3U1. $\mathbf{Y} \leq \mathbf{k}\mathbf{Y}^\star$. Let the sub–optimal solution to PA3U1 consist of p_2 connected to two ports and p_3 connected to three ports.
Thus, $\mathbf{Y} = 2p_2 + 3p_3 \geq 2p$, where $p = p_2 + p_3$. Thus, $p \leq \frac{1}{2}\mathbf{Y}$.
In the lines of lemma 4.9 this solution could also be used as a solution for PA3UA1–R.
Let $\mathbf{X}$, be the cost of this solution to PA3UA1–R.
$\mathbf{X} = 3p \leq \frac{3}{2}\mathbf{Y} \leq \frac{3k}{2}\mathbf{Y}^\star \leq \frac{3k}{2}\mathbf{X}^\star$, where $\mathbf{X}^\star$ is the cost of the optimal solution to PA3UA1.
□

Corollary 4.12 *An approximation for PA3U, PA3U–R, whose relative error is bounded by a fixed constant is NP-complete.*

Proof: Follows from theorem 4.11, as PA3U1–R is a special case of PA3U–R. □

This completes our treatment of the port assignment problem for dual and triple port memories. We shall use the results derived here in the subsequent sections of this chapter. We now go over to register optimization and register–interconnect optimization problems.

4.4 Register Optimization

4.4.1 Prologue

The aim of register optimization (RO) is to minimize the number of registers needed in the design [21]. Registers need to be used to store values between control steps. In the context of data path synthesis registers are needed to implement the variables used to describe the behaviour of the target system. In addition to the variables declared by the designer, some variables may be used at the time of generating intermediate code. All variables in the final implementation need to be mapped onto registers. It may be possible to map some of these registers onto on-chip memories. Registers and variables will be used in an interchangeable manner.

A variable is live from the time when it is first defined till the time that value is last used. A variable may become live several times during the execution of the program. Two variables that are never live at the same time may be merged and placed on the same register without affecting the logical input/output behaviour of the program. It is necessary to determine the life times of each variable in a program. This is called live variable analysis [7]. Once the life times of the variables are known it is necessary to represent their sharability and perform register minimization.

4.4.2 RO for Straight Line Code – A Solved Problem

We call a patch of code that contains neither branching instructions nor targets of branching instructions, a *straight line code*. Such code may be represented by a single DAG. The register optimization for this case will be referred to as SRO. In practice some behaviours encountered are operation intensive and do not contain any decision making branches or looping constructs at all. Sometimes loops with fixed number of iterations may be unrolled to remove the iteration. The intermediate code of such behaviours take a particularly simple form, consisting of only arithmetic or logical operations. For such kind of code it turns out that the complement of the vertex compatibility graph of the variables in the DAG is an interval graph. The complemented vertex compatibility graph (CVCG) in this case being an interval graph may be optimally coloured, using the *left edge algorithm* [21], in polynomial time, V being the set of vertices in the CVCG.

4.4.3 General RO

In general, however, the sharability will not be as simple as that for SRO and may be represented by a graph where there is a vertex for each variable. Two vertices are joined by an edge if the lifetimes of the corresponding variables are disjoint. This graph will be called the vertex compatibility graph (VCG). The problem of register minimization may now be mapped to the clique partitioning problem (CP), which is to find the minimum number of disjoint cliques that cover a graph. Each clique in the VCG corresponds to a set of variables to be mapped on a single register. This general case of RO will be called GRO.

Theorem 4.13 *GRO is NP-complete.*

Proof: CP may be reduced to GRO in polynomial time. Given any graph, an instance of GRO will be constructed. Let the graph be $G(V, E)$. Consider a behaviour which has for each vertex v_i in V of G, a variable x_i. In addition there is a special variable t. Let (v_1, v_2) be any edge in E. The life times of the variables corresponding to v_1 and v_2 should overlap. This is ensured by writing the behaviour as follows.

1. The first line of the behaviour is $\mathbf{read(t)}$;

2. For every edge (v_i, v_j) in E the following lines of code are generated.

   $\mathbf{x_i = f_1(t)}$;

   $\mathbf{x_j = f_2(t)}$;

   $\mathbf{t = f_3(x_i, x_j, t)}$;

   where f_1, f_2 and f_3 are suitable functions.

3. The last line of the behaviour is $\mathbf{write(t)}$.

It will be noticed that the life time of t overlaps with the lifetimes of all other variables in the behaviour and must therefore be implemented on a separate register. The other variables may be grouped depending on their life times. These variable groups will be called *x–clusters*. The life times of any pair of variable x_i and x_j, on the other hand, overlap if and only if $(v_i, v_j) \in E$. Thus each *x–cluster* in the optimal solution to the GRO of this behaviour will correspond to a clique in minimum clique partition of G.

It may be shown that GRO is in NP. □

A comparative study of the complexity of the Traveling Salesperson Problem, Clique, Colouring and Bin Packing has been made in [57]. It has been shown in [58] that no polynomial time approximate algorithm is currently known for graph colouring for which the bound on the relative error is even close to ∞. A very recent result in [59] states that, for the colouring problem there is constant $\epsilon > 0$ such that no polynomial time approximation algorithm can achieve a ratio of n^ϵ (to the optimal) unless P=NP. This leads to the following result.

Theorem 4.14 *The relative approximation of GRO, GRO–R, is NP-hard.*

4.4.4 A More Flexible RO

So far it has been assumed that a variable is mapped onto a single register. This restriction is not essential as a variable can be mapped to multiple registers. We explain this using an example. Referring to figure 4.3, normally, a would be mapped to r_a, b to r_b and c to r_c. However, by the time a becomes live for the second time, b is no longer required. Taking advantage of this, a dynamic mapping could be as follows. First map

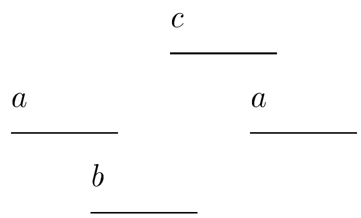


Figure 4.3: Three variables with non-disjoint life times.

a to r_1 and b to r_2. Then map c to r_1 and finally a to r_2. This way only two registers are required.

In practice, however, if a variable has a non-contiguous life time, then the assumption is that due to inherent design constraints (possibly embedded in the behaviour), all the intervals of that variable must be mapped onto the same register. Normally, during the translation process all variables used to store intermediate values are identified through live variable analysis [7] and DAG generation of basic blocks, and named distinctly, thus permitting maximal scope for variable merging. Then the situation explained in the example will not arise.

In the next section we consider the problem of simultaneously optimizing the register and the multiplexer cost. We call this register–interconnect optimization (RIO). RIO based on flexible variable binding could be applied to *intermediate variables*, by breaking up a single contiguous life time. This would not lead to a reduction in the number of variables, but could help to reduce interconnection overhead. It can be easily shown that this is an even harder problem than interconnect optimization with fixed variable bindings.

4.5 Register–Interconnect Optimization

4.5.1 Prologue

Pure RO permits the minimization of registers. However, it has been seen in several design examples that pure RO leads to inferior designs, with excessively high interconnect cost. The interconnect cost may be estimated by counting the total number of multiplexer channels needed at the inputs of the hardware elements used in the circuit. RO performed along with interconnect optimization is called register–interconnect optimization (RIO). The formulation of RIO is similar to RO and is briefly presented below.

The register sharability information is identical to that for RO. In addition a description of the logical netlist is also presented to evaluate the effect of merging two registers on the interconnect cost. For a particular merger the change in multiplexer cost may be computed. The net list needs to be updated at each step to indicate the merger of the two registers. The net effect of the register mergers at a certain time may be computed from the updated netlist of the design. The RIO problem may be formulated as that of finding a set of mergers of registers such that a weighted sum of the register and the multiplexer cost is minimized. Thus the objective function to be minimized may be taken as

$$C = w_1 n_r C_R + w_2 n_m C_M, \tag{4.1}$$

where C_R is the register cost and C_M is the unit multiplexer switch cost; n_r and n_m are the number of registers and the multiplexer channels, respectively. The cost of a register is proportional to the number of bits in the register. The cost of a multiplexer is proportional to the number of lines being multiplexed and the width of its output. If w_1 and w_2 are both taken as 1, in equation 4.1 then the total cost for the registers and the multiplexer channels will be minimized.

If we consider the RIO problem for a special class of circuits where n_m in equation 4.1 will be zero, we immediately get a reduction of RIO to RO. This directly leads to the following result.

Theorem 4.15 *RIO is NP-complete.*

As for RO it is possible to distinguish between general RIO (GRIO) and RIO for straight line code (SRIO). RIO being NP-complete GRIO is also NP-complete. It will be shown in section 4.5.2 that SRIO is also NP-complete. Thus, even though SRO is solvable optimally in polynomial time, using the *left edge* algorithm, SRIO is not. In fact, even approximation of SRIO is hard.

Reduction of PA3U to SRIO

This section presents a reduction of the port assignment problem for three ports, PA3U, to RIO for straight line code, SRIO, and, therefore, to RIO also. The PA3U case consists of a set of transfers between the memory and the points accessing the memory in the various control steps. Each such transfer must take place through a port of the memory. Since only three ports are permitted, in any control step no more than three simultaneous accesses to the memory may be allowed. Since all the ports are uniform, it is permissible to map a transfer on any of the three ports. The mapping is done in two phases. First a mapping is made to SRO. This mapping does not ensure optimality. It is only a prelude to a mapping to SRIO which does ensure obtaining an optimal solution.

The mapping is explained with the help of example 4.3. In this example each access to the memory is denoted as a_{it}, for the i-th access in time step t. While constructing an instance of SRO for the PA problem each such access plays the role of a register. Thus each a_{it} is mapped to a register r_l, such that the mapping $< i, t > \rightarrow l$ is unique. It is evident from the construction that the lifetime of the register r_l is the time step t in which the access a_{it} takes place. The register lifetimes form and interval graph, as expected. This is how the mapping from an instance of triple port memory PA to SRO is achieved.

The total number of registers live in any time step will not exceed three, which is the maximum number of accesses to the memory in any time step. It is, therefore, ensured that there exists an non-empty set of solutions to the SRO problem requiring not more that three registers. These are the solutions that we shall consider. Such a solution will always be obtained by running the left edge algorithm. Each register of the solution to the SRO problem instance actually corresponds to a grouping of memory accesses in different time steps. This can, indeed, be considered an assignment of this transfers to one of the ports of the memory. Since the solutions considered contain no more than three ports, each solution may be used as a feasible solution to the PA problem.

Example 4.3 The creation of the problem instance for SRO and SRIO from a triple port memory PA problem of example 4.1 is now explained. Shown below are the transfers, the corresponding accesses and the variables for the SRO and SRIO instance. The variables a, b, c and d are packed into a memory for which the PA needs to be done. The points $l_a, l_b, \ldots$ are as in example 4.1. (refer to figure 4.1).

	Transfers		Accesses	Registers
1.	$a = b + c;$		a_{11}, a_{21}, a_{31}	r_1, r_2, r_3
2.	$q = c + d,$	$b = p - q;$	a_{12}, a_{22}, a_{32}	r_4, r_5, r_6
3.	$d = a + r,$	$c = p - r;$	a_{13}, a_{23}, a_{33}	r_7, r_8, r_9
4.	$a = p + c,$	$b = q - r;$	a_{14}, a_{24}, a_{34}	r_{10}, r_{11}, r_{12}

The transfers for the SRIO instance which are needed to construct the netlist are shown below.

1.	$r_1 \leftarrow o_a$	$l_a \leftarrow r_2$	$r_a \leftarrow r_3$
2.	$l_a \leftarrow r_4$	$r_a \leftarrow r_5$	$r_6 \leftarrow o_s$
3.	$r_7 \leftarrow o_a$	$l_a \leftarrow r_8$	$r_9 \leftarrow o_s$
4.	$r_{10} \leftarrow o_a$	$r_{11} \leftarrow o_s$	$r_{12} \leftarrow o_s$

□

The above mapping to SRO, does not attempt to restrict the multiplexer usage and so the interconnect cost may be expected to be high. An optimal assignment may be obtained by mapping the problem to SRIO. The register lifetimes are same as for SRO. Only the interconnect information needs to be incorporated. The method of extraction of interconnection information from the transfers has already been illustrated in example 4.3, is as follows. Each point accessing the memory now plays the role of a point in a circuit in the SRIO instance. For a transfer between a point p and the memory in access a_{it}, in the PA example, a transfer between register r_l and point p is created in the SRIO instance.

We now complete the mapping to SRIO. In order to ensure that the solution will not require more than three ports, we fix the weights in equation 4.1 suitably. Suppose that the total number of circuit points accessing the memory is m. In the worst case each point will be connected to all the ports and the number of multiplexer lines needed will be $3m$. This is a upper bound on the number of multiplexer lines needed for the port assignment. In equation 4.1, let w_1 be set to $(3m+2)$ and w_2 be set to 1. Any feasible solution will have cost $c \leq 12m+6$. It is assumed that C_R and C_M have each been set to 1.

4.5.2 Complexity of RIO

It was shown in section 4.5 that RIO is NP-complete. It will now be shown that an approximation for RIO, RIO–R, whose relative error is bounded by a fixed constant is NP-complete. This will be proved through a sequence of reductions. First PA3UA1–R is reduced to SRIO–R. SRIO–R is directly reducible to RIO–R and SRIO. This way SRIO–R and RIO–R will be proved to be NP-hard.

Consider an approximation for SRIO, SRIO–R, whose relative error is bounded by a constant. It has been shown that PA3UA1–R is NP-complete. It will be shown that the approximation PA3UA1–R is reducible to the approximation SRIO–R.

Theorem 4.16 *SRIO–R is NP-complete.*

Proof: Given an instance of PA3UA1–R, which is essentially a PA problem for triple port memory, consider the following. Let $\mathbf{A}^\star$ be an optimal algorithm for PA3UA1. Let $\mathbf{X}^\star$ be the cost of the solution obtained by $\mathbf{A}^\star$, i.e. the multiplexer lines needed as a result of connecting some of the circuit points to all the three ports.

Let $\mathbf{B}^\star$ be an optimal algorithm for SRIO. Let $\mathbf{B}$ be an algorithm for SRIO–R. Let $\mathbf{Y}^\star$ be the effective cost of the solution obtained by $\mathbf{B}^\star$, i.e. total number of multiplexer lines used for the port assignment. Let $\mathbf{Y}$ be the cost of the solution obtained by $\mathbf{B}$. $\mathbf{Y} \leq k_2\mathbf{Y}^\star$.

$\mathbf{Y}^\star \leq \mathbf{X}^\star$, since the mapping of section 4.5.1 ensures that $\mathbf{B}^\star$ yields an optimal solution to PA3U1 and because PA3UA1 is a sub–optimal formulation for PA3U1.

Suppose that the solution obtained be $\mathbf{B}$ requires p_2 two–to–one multiplexers and p_3 three–to–one multiplexers.
Then, $\mathbf{Y} = 2p_2 + 3p_3 \Rightarrow p_2 + p_3 \leq \frac{1}{2}\mathbf{Y} \Rightarrow p_2 + p_3 \leq \frac{k_2}{2}\mathbf{Y}^\star \leq \frac{k_2}{2}\mathbf{X}^\star$

The sum $(p_2 + p_3)$ is the total number of points which are connected to more than one point by $\mathbf{B}$. A sub–optimal solution to PA3UA1 may be obtained by connecting these points to all the three ports of the memory. The cost of this solution would be $3(p_1 + p_2) = \mathbf{X}$.
Thus, $\mathbf{X} = 3(p_1 + p_2) \leq \frac{3k_2}{2}\mathbf{X}^\star \leq k_1^{'}\mathbf{X}^\star$

But the error in the sub–optimal solution obtained by the above method for PA3UA is bounded by a fixed constant. The above method may therefore be used as an algorithm for PA3UA1–R. Since PA3UA1–R is NP-complete, it follows that SRIO–R is NP–Hard.

It may be easily proved that SRIO–R is in NP. □

The following three corollaries follow from theorem 4.16.

Corollary 4.17 *SRIO is NP-complete.*

Corollary 4.18 *An approximation for RIO, RIO–R, for which the relative error in the solution is bounded by a fixed constant is NP-complete.*

From corollary 4.18 also it follows that RIO is NP-hard.

4.6 The Problem of Forming Functional Units

We have explained earlier that the design of the data paths starts with the scheduling of the DAG's and is followed by allocation and binding. We consider a practical situation where the design has to be performed subject to a constraint that a specified number of functional units has to be used. This constraint leads to the requirement that no more than this number of operations should be scheduled in any time step. This a small departure from a prevalent approach to the scheduling problem where the sum total of the cost of hardware operators is to be minimized without consideration for the number of units where these operators are housed.

With both the approaches the objective is to minimize the cost of hardware operators. Scheduling algorithms for the prevailing approach seek to minimize the maximum number of operations of each kind that will be performed in any time step. The actual formation of the f.u.'s is usually done by binding the operations to the functional units (f.u.) during the allocation and binding step. This process is illustrated and clarified in example 4.4.

Example 4.4 Consider the schedule of operations in example 4.1. The scheduling algorithm would report that a maximum of one addition and one subtraction are performed in each time step. Thus a minimum of one adder and one subtracter needs to be allocated. The scheduling has been done so that a maximum of two operations are scheduled per time step, so that two f.u.'s would suffice to actually accommodate each operation in each time step. All the additions could be mapped onto one of the f.u.'s and all the subtractions could be mapped onto the other. The resulting f.u.'s would be as shown in figure 4.1. □

Example 4.5 Now consider the following transfers.

1. $q = c + d, \quad b = p - q;$
2. $d = a + r, \quad c = p \,\&\, r;$
3. $a = p \,\&\, c, \quad b = q - r;$

As in the case of example 4.4 a maximum of two operations are scheduled per time step and so two f.u.'s would suffice to actually accommodate the operations in each time step. The scheduling algorithm would report a maximum of one adder, one subtracter and one *word–and gate*. However, an inspection will reveal that an assignment of these operations to the f.u.'s such that at most one f.u. implements one operation is not possible. At least one of the operations would have to be assigned to two f.u.'s in order to satisfy the specified schedule. Thus we see that the allocation of one operation of each kind which is implied by the schedule, cannot be satisfied in this case. A set of f.u.'s resulting from such an assignment would be $< +, - >$, and $< +, \& >$. □

Example 4.5 actually reveals a severe difficulty faced while performing scheduling. While scheduling it is desirable to find a schedule which is realizable using a set of f.u.'s of minimum cost. The usual method of estimating the cost of the operator set that would be required to actually realize the operations in the schedule is to sum up the costs corresponding to the maximum number of operations of each kind. As it has been indicated in example 4.5 this method is inadequate, as in final assignment of operations to the f.u.'s some more operations of each kind could be required, thus violating the allocation implied by the schedule.

The decision problem of forming f.u.'s from the allocation implied by the schedule is formally defined as follows. We are given a schedule of operations, each of a single time step. The schedule consists of $p, p > 0$, types of single time cycle operations. At most $n, n \geq 3$, operations are scheduled in each time step. The maximum number of operations of type i in any time step of the schedule is $m_i, m_i \leq n$. The problem is to determine whether an assignment exists of the operations in each time step to n f.u.'s, such that no two operations in the same time step are mapped to the same f.u. and no more than m_i of the f.u.'s implement operation of type i. We shall refer to this problem as FUFD.

We would like to determine the complexity of the above problem. To do this we consider the problem special case when there is at most one operation of a particular kind scheduled in any time step, i.e. $m_i = 1$ and a fixed number k of f.u.'s are to be used. This special case will be referred to as FUFD1(k), for which we have the following result.

Theorem 4.19 *For a fixed integer k, FUFD1(k) is NP-hard.*

Proof: Given a graph $G(V, E)$ for GND, an instance of FUFD1(k) will be constructed as follows. For each vertex $v_i \in V$ let O_i be a type of operation in the schedule. For each clique of size l, $l \leq k$, construct a time step where the operations scheduled are as follows:

$\mathbf{o_{i_1}}, \quad \mathbf{o_{i_2}}, \ldots \quad \mathbf{o_{i_l}},$

where o_j is an operation of type O_j.

This creates an instance of FUFD1(k) in polynomial time when k is fixed. The graph will be k colourable if and only if an assignment of the operations to the k f.u.'s exist such that no operation is implemented by more than a single f.u. □

We need not be restricted to designing f.u.'s where the usage of individual operations does not violate the initial allocation. Our main concern is to get f.u.'s of minimum cost so that the schedule that has been obtained can be realized. However, as the decision problem of designing the f.u.'s with a fixed number of f.u.'s is hard, as shown in theorem 4.19, the aforesaid minimization problem is also hard.

Theorem 4.19 establishes the NP-hardness for this problem only when three or more f.u.'s are to be used. If only one f.u. is to be used then the problem is trivial. We shall now examine the problem when only two f.u.'s are to be used.

When only two f.u.'s are to be used it follows that at most two operations may be scheduled in any time step. Suppose that a maximum of one operation of any kind is present in a time step. It is now possible to find out in polynomial time whether the required assignment of operations will exist or not. This is because the problem of finding the bipartite partition of a graph, if one exists, can be done in polynomial time. However, for our purpose we need the actual assignment of operations to the f.u.'s so that the cost of the units is minimized. We now show that this problem is NP-hard.

Theorem 4.20 *The problem of determining the actual assignment of single cycle operations to f.u.'s when exactly two f.u.'s are to be used, so as to minimize the cost of the f.u.'s is NP-hard.*

Proof: Consider the special case when exactly one operation of each type is present and an operator for each operation has the same cost. Construct a conflict graph G as explained the proof of theorem 4.19. It would now be necessary to two colour the graph with just two colours. If the colouring succeeds then an assignment exists that does not violate the default allocation. However, if the colouring fails, then it becomes necessary which of these operations should be assigned to both the f.u.'s. In order to minimize the cost of the formation of the f.u.'s we would like to assign as few of these operations, as possible, to both the f.u.'s. This corresponds exactly to performing minimum node deletion on the graph G. The operations corresponding to the nodes that have been deleted would have to be assigned to both the f.u.'s. It is well known that minimum node deletion is NP-complete and this proves the theorem. □

Theorem 4.21 *The problem of determining the assignment of operations to a fixed number k, $k > 1$ of f.u.'s, so as to minimize the cost of the f.u.'s is NP-hard.*

Proof: It is easy to show that this problem is in NP. The NP-hardness of this problem follows from theorem 4.19 and theorem 4.20. □

4.7 Conclusions

In this chapter we have examined the complexity of several synthesis tasks that are encountered during allocation and binding. The problems that we have considered are the port assignment problem for dual and triple port memories, the register-interconnect optimization problem and the problem of formation of functional units. We have shown that all these problems in general are NP-complete.

The port assignment problem has recently acquired significance because dual and triple port memories are now being used on-chip. Moreover, we have used the complexity results of PA to derive the complexity results for RIO.

We have also examined the complexity of the problem of minimizing the cost of functional units, when it is necessary to make a design where the total number of f.u.'s is specified before hand. We have shown that this problem is also NP-hard.

We have examined several versions of the RIO problem. The simplest case that we have considered is RO for straight line code which is solvable in polynomial time. The next more general case that we have considered is RIO for straight-line code, which we have shown to be NP-hard. All the results of SRIO carry over to RIO because SRIO is a special case of RIO. Some results on connectivity binding were already available [31]. However, our results have been derived using a different approach which produces newer insights into the hardness of the interconnect optimization problem.

The most significant contribution of our work on the complexity of allocation and binding are the hardness results of the relative approximation of several sub-problems in this area. We have shown that the constant bounded relative approximation of PA for triple port memories and SRIO are both NP-hard. These hardness results suggest that the use of deterministic approaches to very global formulation of allocation and binding may not prove fruitful, in general. We have developed genetic algorithms (GA) for the dual and triple port memory PA problems and the allocation and binding problems. The GA is essentially a stochastic optimization technique. We have obtained favourable results in all the cases. These are discussed in subsequent chapters.

We restate here that while approximation of most of the allocation and binding problems are NP-hard, there are polynomial time approximate algorithms for scheduling. It is known that list scheduling can guarantee schedules within twice the optimal schedule length.

Chapter 5

Register and Memory Interconnect Optimization

5.1 Introduction

In chapters 3 and 4 we have examined the complexity of several synthesis sub-problems. We have noted that these problems are in general hard. However, these problems are of practical interest which demand solutions. In this work we have proposed solutions to some individual sub-tasks and to the entire DPS problem to generate an optimized data path. The individual sub-problems that we have considered are related to interconnect optimization. These have been receiving considerable attention because of the impact of interconnect area on the final design. The particular individual sub-problems that we have considered are as follows.

1. *Register-interconnect optimization* (RIO) assuming that operation to functional unit binding has been completed. The interconnection considered here is point to point.

2. *Memory interconnect optimization* (MIO) in the lines of RIO. The difference is that the modules for storage which may be used are registers, single port or multiple port memories.

3. *Port assignment* (PA) for dual port and triple port memories.

While developing a solution to the entire DPS problem we present a two-phase strategy with the following two phases;

1. *Design space exploration (DSE) and Scheduling*, where the input is a set of optimized data flow graphs along with some design parameters. The output is a set of scheduled data flow graphs.

2. *Allocation and binding*, which accepts the output of the DSE module and produces the data paths of the target system. This module assumes a bus based interconnection structure.

The global problem of allocation and binding, as we have considered it, involves binding of operations to f.u. sites, binding of transfers to buses and memory formation. All these tasks have to be done to minimize the overall cost of the data path, which is the sum of the cost of the f.u.'s, the storage elements and the switches required for interconnection. Our solution to DSE and scheduling is based upon combination of systematic search of the design space and a variety of scheduling techniques, including heuristic scheduling and genetic list scheduling algorithms.

In this chapter we present techniques that we have developed for RIO and MIO. Solutions to the other problems have been discussed in the subsequent chapters. Here we have considered two sub-problems of data path optimization where it is assumed that operation to functional unit (f.u.) binding has already been completed. Both register and memory interconnect optimization work under this assumption. We have chosen a point to point interconnect style as against the bus based interconnection structure used later for global allocation and binding. Through this work we have also been able to make a study of the kind of trade-off that exists for RIO against pure register optimization (RO). We have then generalized this approach to use multi-port memories instead of single registers only. For this formulation we have been able to integrate port assignment with interconnect optimization.

Considerable work has been done for pure RO. In general, representation of conflict of variables takes the form of an arbitrary graph, for which general graph colouring algorithms [60] need to be used. The conflict graphs are simplest for single basic block (b.b.) designs, (when they happen to be interval graphs) and can be coloured efficiently [35]. Even when multiple b.b.'s are present there are some special cases when the conflict graph is asteroidal, for which efficient colouring algorithms are available [61]. Paulin has proposed a method for heuristic weighted register allocation [62] to combine interconnect optimization with RO. A comprehensive study of register allocation for DPS is available in [12].

In section 5.2 we present the formulation for register-interconnect optimization. The algorithm for RIO is explained in the next section, which is followed by the experimental results for RIO. In section 5.5 we explain the work on memory-interconnect optimization (MIO). The use of the RIO technique for doing port assignment in MIO is explained in section 5.5.1. Sections 5.5.2 and 5.5.3 present the formulation and algorithm, respectively, for memory allocation. Experimental results for memory-interconnect optimization are given in section 5.6. The conclusions are given in section 5.7.

5.2 Problem Formulation for RIO

5.2.1 Prologue

Register-interconnect optimization (RIO) is an intermediate stage in the synthesis process. In DPS, first the initial description is transformed into one or more directed acyclic graphs (DAG) [7] to represent the data flow and the control flow information. Next, the DAG's are scheduled, possibly using a given set of functional units. Alternatively, the f.u.'s may be fixed after scheduling is over. Then the operations in the DAG's are bound to specific f.u.'s. These scheduled DAG's with operations bound to specific f.u.'s serve as the input to the RIO algorithm. The RIO proposed here proceeds by merging several variables into registers in such a way that the total register area and interconnect area is reduced.

In order to perform merging of several variables into a single register it is necessary to extract the compatibility information of the variables from the scheduled DAG's. This is done in as step called live variable analysis (LVA) [7]. By using LVA we detect the conflicts not only of variables used within a basic block but also of variables which only store a value in a particular basic block for use in a different one. The output of LVA is the compatibility information of all the variables used in the intermediate representation. For practical implementation this information will be stored in a suitable data structure. However, for describing our techniques in this chapter we will assume that it has been stored in a matrix.

To compute the interconnection cost it is necessary to extract the interconnection information from the scheduled DAG's. A statement of the form $a = b$ **op** c, where operation **op** is bound to hardware module M, gives rise to the following interconnection. The output of M feeds the input of register containing variable a, output of registers containing variables b and c feed the left and right inputs of M, respectively. Thus for each operation as well as transfers to or from system interface ports the interconnection is appropriately formed. Switches or multiplexers are placed so that proper transfer of data can take place.

5.2.2 Formulation

We now present the condition of variable/register mergers and how it affects the sharability and interconnect information. It is necessary to partition the variables into sets. The variables in each set which are to be implemented on a single register must be mutually compatible. The grouping of the variables into registers not only affects the number of registers that need to be used but also has an impact on the interconnection structure of the target system. The initial mapping of each variable to a single register can be considered to be a trivial design which we call d_{R_0}. In d_{R_O} each register is associated with a set that contains only one variable.

Subsequently, the sets of variables are merged to reduce the number of registers in the design. Two sets of variables can be merged only if all the variables in the union of these two sets are mutually compatible. The merger of two sets of variables is equivalent to the merger of their corresponding registers, so that register merging and variable set merging can be used interchangeably. A merger of two registers corresponds to a 'move' (or state transformation) which results in a new design. It may be noted that from a design d_{R_i}, we can have several new designs depending on the type of merging we do in d_{R_i}. We shall designate the collection of all possible designs (starting from d_{R_0} and using various mergings) as the register design space D_R. We also designate a function f_{C_R}, defined over D_R, which returns the cost of a design d_{R_i}. It is the sum of the cost of registers and multiplexers used in d_{R_i}. The cost of the functional units and other design components is not taken because they remain invariant during RIO.

With the merger of a pair of registers it becomes necessary to update the compatibility information of the variables as well as the interconnection information of the design. We associate with each design d_{R_i} a compatibility matrix s^i. (The initial compatibility matrix is called s^0.) The compatibility matrix s is symmetric and $s_{ij} = 1$ only if register r_i is compatible with register r_j. While moving from design d_l to d_{l+1} as a result of merging registers r_i to r_j, $i < j$, s^{l+1} is obtained from s^l as follows. We assume that after merging, the merged register is termed r_i. Matrix s^{l+1} is obtained by deleting the j-th row and column from s^l. Element $s^{l+1}_{ik} = 1$ only if $s^l_{ik} = s^l_{jk} = 1,\ k < j$ or $s^l_{ik+1} = s^l_{jk+1} = 1,\ k \geq j$. We denote the function which maps s^l to s^{l+1} as a result of merging registers r_i and r_j by f_{T_S},

The updation of the interconnect information is as follows. All the sources that were initially feeding to r_i and r_j now feed only to r_i, all the points that were receiving from r_j now receive from r_i, instead. Register r_j is no longer required. All registers $r_l,\ l > j$ are now renamed r_{l-1}. We have a function f_{T_R} which updates the interconnect information as a result of register merging in d_R.

It is clear that f_{T_R} and f_{T_S} have to be used together on the current design and the current compatibility matrix, respectively. Their application actually constitute a *move* from one design to another. Each application of f_{T_R} corresponds to the updation of the current interconnect information and each application of f_{T_S} corresponds to the updation of the current compatibility matrix as a result of merging the two registers selected for merging in the current move. The set of designs reachable from the starting state by a sequence of moves constitute the state space. Each design or state is a feasible design having a certain cost, as indicated by the cost function f_{C_R}. The problem is to find a sequence of moves, starting from the initial design d_{R_0} to the design of minimum cost. This formulation suggests the use of a branch and bound strategy for obtaining the optimal solution with respect to the f_{C_R}. However, in view of the large search space and the storage requirements for each partial solution we choose a heuristic method, instead. Such a heuristic method, in general, will not locate the design of minimum cost, but we expect that it will locate a design whose cost is "close" to this cost. Since both absolute and relative approximation of RIO have been proved to be NP-hard in chapter 4, we will study the performance of our scheme by comparing known results as well as by extensive testing using appropriate randomly generated inputs.

The cost function f_{C_R} is a measure of the area occupied by the registers and the interconnect elements. This function should be easily computable as the time complexity of the algorithm would be unacceptably high if the time complexity of computing f_{C_R} is high. A measure derived by actually performing the routing and layout is very costly in terms of computation time. Even area estimators such as Plest [30] that avoid routing are not fast enough for our purpose.

The cost function we use here is based on the register and multiplexer usage. The register cost is proportional to the width of the register and the multiplexer cost is proportional to the number of input channels and the width of the output. A linear combination of the register and multiplexer cost is taken as the total cost. This cost function is as follows:

$$f_{C_R} = w_1 C_R + w_2 C_M, \tag{5.1}$$

where C_R and C_M are the register and multiplexer costs, respectively. This function will be referred to as RMC. Each register or functional unit port itself takes into account one bus at every input point. Multiple connections which enter a point require a multiplexer whose size varies depending on the number of lines it must multiplex. So a multiplexer accounts for the additional connectivity at a point. The minimization of this function would result in a reduction of the number of registers and multiplexers in the design. The physical interconnection is also coupled with this cost function in some sense since it is normally reduced along with the registers and the multiplexers.

5.3 Algorithm for RIO

5.3.1 Motivation

Several factors have been kept in mind while designing the heuristic algorithm for RIO. Merging of two registers in general produces a saving in area. Therefore the greedy algorithm must attempt to find register pairs that yield maximum reduction in the interconnect complexity. However, this approach could choose pairs that rapidly deplete the edges in the compatibility graph. This will reduce chances of future mergings and generate a design with too many registers. By concentrating on register minimization alone a design with high interconnection complexity could result. It is therefore necessary to have a tradeoff between the two optimization criteria.

The following definitions enable a concise presentation of the algorithm.

Common vertex A vertex v is said to be common to vertices v_1 and v_2 if $s_{vv_1} = 1$ and $s_{vv_2} = 1$. Here s is the compatibility matrix.

Deletable edge An edge (v, v_1) is said to be deletable on merging v_1 and v_2 if $s_{vv_1} = 1$ and $s_{vv_2} = 0$. The function $de(v_1, v_2)$ returns the number of edges deletable on merging v_1 and v_2.

Clique factor Let P_i be the vertex pairs in s having precisely i common vertices. Let $Q_i = \bigcup_{j=i}^{j=\infty} P_j$. The clique factor associated with Q_i is $cf(Q, i) = \dfrac{2|Q_i|}{(i+2)(i+1)}$.

The above definition of clique factor is motivated by the fact that in a clique of $n+2$ vertices there are $\dfrac{(n+2)(n+1)}{2}$ edges, and every edge has n common vertices. Thus for the presence of a $n+2$ member clique a necessary condition is that $cf(Q, n) \geq 1$. It is quite clear that this is not a sufficient condition. We shall use the clique factor to quickly determine possibly large cliques.

5.3.2 Pseudocode

The algorithm may now be outlined as follows.
procedure reg_min

1. $d_R = d_{R_0}$ /* initial interconnect information */
2. $s = s^0$ /* the initial compatibility matrix */
3. while ($d \neq \mathbf{0}$) do
4. { Determine the maximum x and the corresponding Q such that $cf(Q, x) \geq 1$; in the absence of such an x, let Q include all the compatible variable pairs in s.
5. Determine the member (v_1, v_2) of Q which has the minimum value of $\dfrac{de(v_1, v_2)}{f_{C_R}(d_R) - f_{C_R}(f_{T_R}(d_R, v_i, v_j))}$
6. $d_R = f_{T_R}(d_R, v_1, v_2)$
7. $s = f_{T_S}(s, v_1, v_2)$
8. }
9. $d_{R_f} = d_R$

5.3.3 Example

A small example is now presented to illustrate the operation of the given algorithm.

Example 5.1 Consider the code sequence given below.

$v3 = v1 + v2 \quad v12 = v1$
$v5 = v3 - v4$
$v6 = v3 + v5 \quad v7 = v8/v5$
$op = v4$ & $v7$

The intermediate code for this behaviour is a single DAG and only the life times of the variables are given for convenience. The two f.u.'s used for this design are ⟨+, -, &⟩ and ⟨/⟩. Thus the operations '+', '-' and 'and' are realized on the same f.u., while '/' is realized on the other f.u.. The final assignment is made to **op** which is an output port. The life times are as follows. **L** means that the variable is live. **D** means that the variable is dead.

time step	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
1	L	L	D	L	D	D	D	L
2	D	D	L	L	D	D	D	L
3	D	D	L	D	L	D	D	L
4	D	D	D	D	D	L	L	D

In the first iteration *reg_min* finds the following variable pairs satisfying the clique factor criterion for a four member clique: $\langle v_1, v_3\rangle$, $\langle v_1, v_6\rangle$, $\langle v_1, v_7\rangle$, $\langle v_3, v_6\rangle$, $\langle v_3, v_7\rangle$, $\langle v_1, v_5\rangle$, $\langle v_5, v_6\rangle$, $\langle v_5, v_7\rangle$, $\langle v_2, v_3\rangle$, $\langle v_2, v_6\rangle$, $\langle v_2, v_7\rangle$, $\langle v_2, v_5\rangle$, $\langle v_4, v_5\rangle$. Actually a four member clique cannot be formed, but this is obscured by the large number of edges present.

Technique	No. of registers	No. of mux. channels	Cost
RIO	4	4	64
REAL	4	5	72

Table 5.1: Results for example 5.1.

Among these edges $\langle v_1, v_3\rangle$ has the maximum multiplexer saving and one of the least deletable edge count. Thus this edge is selected for merging. The merged variable takes the name v_1.

In the second iteration the following edges are found satisfying the clique factor for a four member clique: $\langle v_5, v_6\rangle$, $\langle v_5, v_7\rangle$, $\langle v_2, v_3\rangle$, $\langle v_2, v_6\rangle$, $\langle v_2, v_7\rangle$, $\langle v_2, v_5\rangle$, $\langle v_4, v_5\rangle$. Among these edges $\langle v_2, v_5\rangle$ has the one of the maximum multiplexer saving and one of the least deletable edge count. Thus this edge is selected for merging. The merged variable takes the name v_2.

In the third iteration the following edges are found satisfying the clique factor for a three member clique: $\langle v_1, v_6\rangle$, $\langle v_1, v_7\rangle$, $\langle v_2, v_6\rangle$, $\langle v_2, v_7\rangle$, $\langle v_4, v_6\rangle$, $\langle v_4, v_7\rangle$. We select edge $\langle v_1, v_6\rangle$ which has the one of the maximum multiplexer saving and one of the least deletable edge count. The merged variable takes the name v_1.

Finally the edges $\langle v_2, v_7\rangle$, $\langle v_4, v_7\rangle$ and $\langle v_8, v_7\rangle$ satisfy the clique factor for a two member clique. The edge $\langle v_2, v_7\rangle$ is selected. The merged variable takes the name v_2. The final groupings turn out to be $\langle v_1, v_3, v_6\rangle$, $\langle v_2, v_5, v_7\rangle$, $\langle v_4\rangle$ and $\langle v_8\rangle$. For this grouping four multiplexer channels are needed.

REAL [21] might have found the groupings $\langle v_1\rangle$, $\langle v_2, v_3\rangle$, $\langle v_4, v_5, v_6\rangle$ and $\langle v_8, v_7\rangle$, for which five multiplexer channels are needed. The results are tabulated in table 5.1. □

5.3.4 Analysis

The complexity of the algorithm is dominated by the time taken to execute the inner loop. Most of the operations are sensitive to the data structures used to represent the information. Essentially, a trade–off needs to be made in the choice of the data structure used for representation. In order to make searching efficient, a hash table may be used. In order to find the edge with maximum number of adjacent vertices, a heap may be used. For overall speed of the algorithm a redundant representation scheme is useful, involving both hashing and heap representation schemes. Now the upper bound on the time complexity of the algorithm can be easily found. Let e be the number of edges in the graph. The hash table is formed in $O(e)$ time. The heap is formed in $O(e\ \log(e))$ time. The edge with maximum common vertices is found in constant time. As a result of merging two vertices at most e edges are affected. Thus e entries in the heap may have to be repositioned. This can be done in $O(e\ \log(e))$ time. The updation of the interconnect cost can be done in $O(en)$ time, where n is the number of fixed connection points. The computation inside the loop takes $O(\max(\log(e), n))$ time. The outer loop can iterate at most $v - 1$ times, where v is the number of vertices in the original graph. The processing time dominates over the pre–processing time. The overall time complexity of the algorithm is $O(ve \max(\log(e), n))$.

The algorithm *reg_min* performs a local optimization of the register and the multiplexer cost. Normally, the register cost is considerably more than the multiplexer cost. If the unit register cost is taken to be no less than twice the unit multiplexer cost, it is seen that each register merger monotonically decreases RMC of equation 5.1. However, *reg_min* does not go by the local minima of RMC. It goes by the heuristic represented by the clique factor. This is done to get the maximum number of register mergers. The performance has been checked by a statistical testing. Details of performance of the above algorithm are presented in section 5.4.

5.4 Experimentation for RIO

The above algorithm has been tested extensively on randomly generated behaviours. The behaviours are generated on the basis of two parameters, namely. the number of variables and the number of functional units (f.u.). Here f.u.'s are used to denote hardware f.u.'s capable of performing some predetermined functions. It had been assumed that all the elements of the data path are eight bits wide. Each randomly generated behaviour is compiled and the corresponding data flow graph, in the form of a DAG is constructed. This DAG is scheduled on the basis of a specified set of f.u.'s, which was generated along with the behaviour. A live variable analysis is performed on the DAG and the life times of the variables is determined. From this information the initial compatibility graph is constructed. A trivial allocation of registers for the variables is made on a one-one basis. The initial interconnection is now implied. The initial allocation and the interconnection along with the compatibility matrix for register-interconnect optimization is generated. The above preprocessing steps are done by ABS [63], a synthesis tool developed earlier.

Using the initial interconnection and compatibility information two designs are synthesized. The cost per bit of a register has been taken as two, while the cost per bit of a multiplexer output line has been taken as one. Both the methods use the above algorithm but use different linear combinations of the register and multiplexer costs. In one case the weight of the multiplexer cost is zero while the weight of the register cost is one. In the second case both the costs have the same weights. In the first case only register optimization takes place, whereas combined optimization takes place in the later case. The two cases are sometimes referred to as 'Z' and 'NZ' respectively.

The testing has been done for 1, 2 and 3 f.u.'s in association with 5, 10, .., 25 variables. The programs had been generated to avoid dead code formation. For each (f.u., variable) pair 30 examples had been generated and tested. The results have been summarized in the figures.

The DAG representation of the given behaviour is similar to the single assignment transformation. This leads to an inflation in the actual number of registers to be mini-

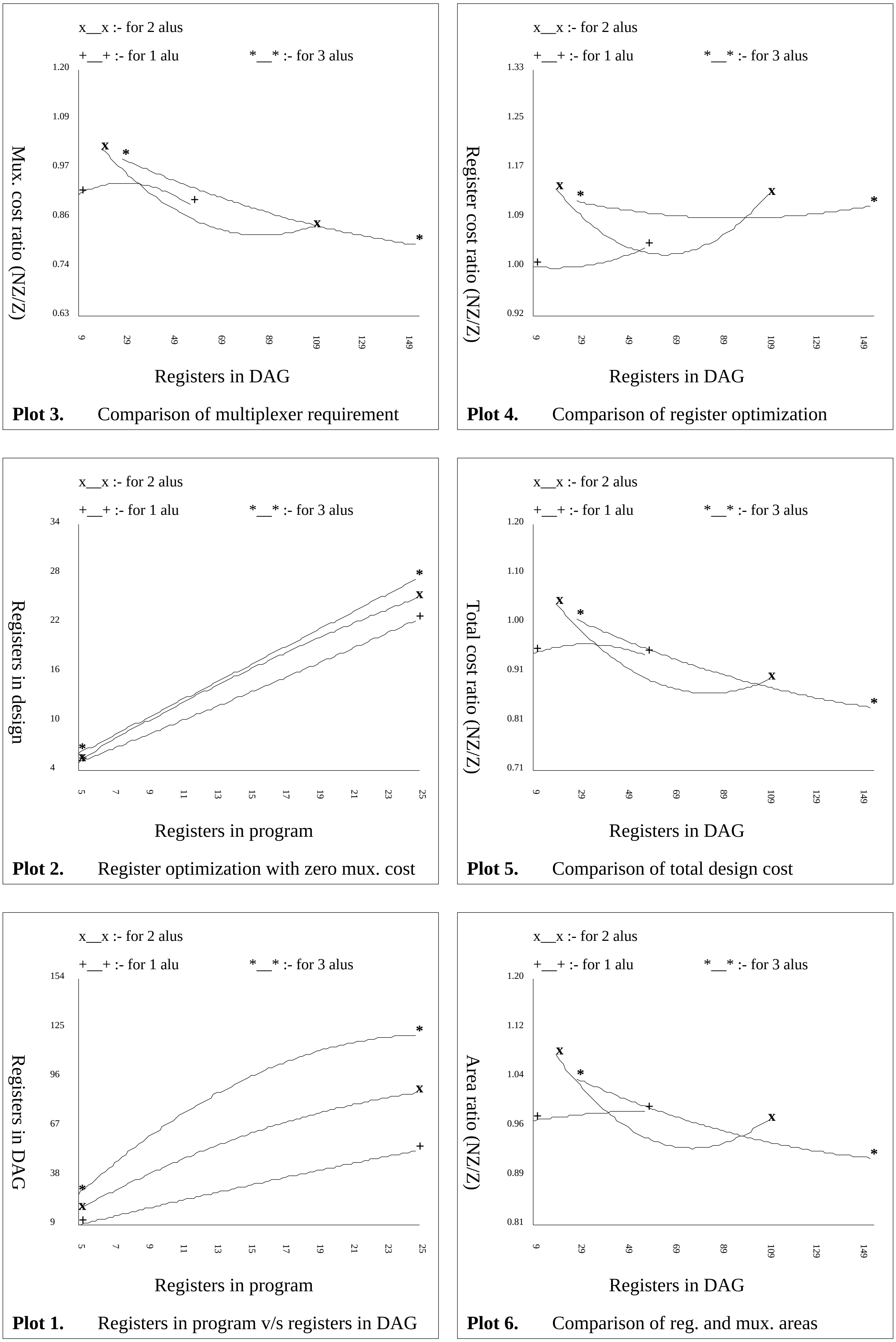


Figure 5.1: Graphical results of statistical testing for RIO.

mized. Plot 1 of figure 5.1 shows that the registers in the DAG grow linearly with the registers used in the corresponding behaviour (expressed as a program segment). All the statistics to study the performance of the algorithm are measured with respect to the registers in the DAG because this is what is minimized, along with the multiplexers.

Plot 2 of figure 5.1 shows the final register count attained when the multiplexer cost is zero (Z). Here the algorithm performs pure register minimization. When a single f.u. is used, it is always possible to synthesize a design with precisely the number of variables that occur in the program. The graph indicates that in the case of a single f.u. the algorithm does achieve this. For more number of f.u.'s more registers are required as expected. But the final register count is still close to the number of variables used in the behaviour. Thus the algorithm performs register reduction reasonably.

Plot 3 of figure 5.1 shows the ratio of the multiplexer cost of the design when the multiplexer *cost* (in f_{C_R}) is taken to be zero to than when the *cost* is 1. We notice the there is always an improvement. The improvement is even better when the complexity of the design goes up.

Plot 4 of figure 5.1 shows the register cost ratio computed as above. As expected the register usage is less (or equal) when the multiplexer cost is zero. It has a tendency to go up slowly with the number of f.u.'s used. However, the ratio does not depend strongly on the number of variables used in the program. The above factors make the technique attractive. This is because, even when the multiplexer cost is non-zero (NZ) the algorithm is able to perform register minimization effectively. Moreover, multiplexer cost minimization also takes place. As stated before this directly reduces the number of buses in the system.

Plot 5 of figure 5.1 shows $\frac{f_{C_R}\ {}_{mux.cost=1}}{f_{C_R}\ {}_{mux.cost=0}}$, indicating the ratio of the total cost of the components used in the design, as a result of optimizing with and without the cost of multiplexing switches. Plot 6 of figure 5.1 shows the ratio of the register and the multiplexer areas. The register and multiplexer costs have been taken to be proportional to the number of transistors required to fabricate them. This ratio also shows the same trend as that of plot 5. Both these plots are similar, though plot 6 is more accentuated, as expected.

In addition to the statistical testing described above and example 5.1 mentioned earlier, the method was applied to an example reported in Facet [17]. The example is reproduced below (in example 5.2) for convenience. The resulting design was compared against the reported design. A marginal saving was obtained. Register allocation for the same design was also done using REAL [21]. The designs are compared in table 5.2. It is seen that the technique presented in this chapter requires fewer multiplexer channels.

Example 5.2 The code sequence is as follows.

$v3 = v1 + v2$	$v12 = v1$	
$v5 = v3 - v4$	$v7 = v3 * v6$	$v13 = v3$
$v8 = v3 + v5$	$v9 = v1 + v7$	$v11 = v10/v5$
$v14 = v11\ \&\ v8$	$v15 = v12\ or\ v9$	
$v1 = v14$	$v2 = v15$	

Comparison of results				
Technique	No. of registers	No. of mux.	No. of mux. channels	Cost
Facet	8	4	9	136
RIO	8	4	8	128
REAL	8	5	12	160

Table 5.2: Results for example 5.2.

The operations supported by the three f.u.'s used by Facet are $\langle +, \star, \text{or} \rangle$, $\langle +,-,\& \rangle$ and $\langle / \rangle$. □

It may now be concluded that register–interconnect optimization is an important problem. It finds application for independent optimization of the data path, after scheduling and f.u. allocation have been done. It also finds application for performing lookahead to estimate the RI cost in global data path optimization schemes. An useful technique has been developed to perform register–interconnect optimization which is more effective than pure register minimization.

5.5 Memory–Interconnect Optimization

We have seen that RIO is capable of producing interconnect optimized designs using about the same number of registers that would be used with pure RO. It is possible to achieve some further reduction in cost, in terms of real estate, by grouping registers into memories. It may be noted that small sized memories, say up to eight, behave nearly as fast individual registers [64]. Thus the performance degradation is not significant. The reduction in multiplexers and buses arises due to the following facts.

- Two registers with overlapping life-times but disjoint access times can be merged into the same memory. The merging is more beneficial if they have common sources and targets.
- Two registers having overlapping access times can only be merged using a multi-port memory. Here we have the added flexibility of distributing the memory references over the ports so that multiplexer cost is reduced.

Allocation of registers to memories has two important subproblems.

1. Which register should be placed in a particular memory?
2. How should the references to the constituent registers be distributed over the ports?

For the first sub-problem, called *memory allocation*, we have devised a greedy algorithm to select the variables which are to be packed into a particular memory. We form the memories one by one, till no more variables may be profitably grouped to form a memory. The algorithm works with constraints on the maximum number of cells and ports permissible for a memory. Thus a few variables may be mapped to individual registers instead of being packed into memories. The selection of variables to be placed in a particular memory is done to minimize the cost of the interconnection of the memory with other elements in the circuit. When a set of variables is placed in a memory the cost of interconnection is determined by performing an actual *port assignment*. This is the second problem mentioned above. The formulation and technique for port assignment (PA) for MIO is different from the PA techniques discussed in chapter 6. In this case the PA problem has been formulated as a register-interconnect optimization problem and solved using the *reg_min* procedure described earlier in this chapter. We explain the PA formulation for MIO and its solution in section 5.5.1. The technique for memory formation is explained in section 5.5.2.

5.5.1 Using RIO as a PA tool for MIO

We have already given a basic introduction to the port assignment problem in section 4.3. Given a set of variables to be placed in a memory, it is necessary to distribute accesses to these variables over the ports of the memory. A single port memory might suffice, or a multi-port memory might be called for, depending on the maximum number of accesses made to these variables in any time step. The port assignment should be such that cost of the resulting interconnection is minimal. It is sometimes necessary to do the PA when the number of ports to be used is specified in advance. This is the approach taken in chapter 6, where we solve the problem for dual and triple port memories. In this case we are given the set of variables to be placed in the memory and the maximum number of ports that may be used. Our objective is to determine the actual number of ports to be used and the corresponding port assignment so as to minimize the cost of the interconnection involved. We have formulated the solution to this problem in terms of register-interconnect optimization problem. Essentially we transform an instance of the PA problem to an instance of RIO, solve the RIO problem and then use this solution to derive the solution to the PA problem. We use a transformation similar to the one which has been used in section 4.5.1. We present the transformation below.

We are given a set of variables to be placed in a memory and the schedule of operations. This leads to an instance of the PA problem. Now with the help of example 5.3 we explain in example 5.4 the construction of the instance of RIO. In example 5.4 each access to the memory is denoted as a_{it}, for the i-th access in time step t. Each such access plays the role of a variable in the RIO instance. Each access a_{it} is mapped to a register r_l initially, such that the mapping $< i, t > \rightarrow l$ is unique. It is evident from the construction that the lifetime of the register r_l is the time step t in which the access a_{it} takes place. A valid grouping of variables to form a memory will satisfy the requirement that the total number of registers live in any time step will not exceed the maximum number of ports permitted. Each register of the solution to the RIO instance

instance actually corresponds to a grouping of memory accesses in different time steps and becomes a memory port in the solution to the PA problem. The interconnection information in the RIO instance is gathered by examining the accesses a_{ij}.

Example 5.3 Here we explain the port assignment problem. Consider the transfers given below.

1. $a = b + c$;
2. $q = c + d, \quad b = p - q$;
3. $d = a + r, \quad c = p - r$;
4. $a = p + c, \quad b = q - r$;

Suppose a, b, c, d, p, q, r and s are registers, of which only a, b, c and d are to be placed in the same memory. Assume that at most three ports are permitted, and the ports are labeled $0, 1$ and 2 respectively. It will be noted that at most three accesses are made to the memory in any time step. Suppose that an adder and a subtracter are used. Let the adder inputs be labeled l_a and r_a while the adder and subtracter outputs be respectively labeled o_a and o_s. It will be observed that l_a, r_a, o_a and o_s are the only four points accessing the memory in the various control steps. They need to be assigned to the ports suitably. Consider the assignment where l_a, r_a and o_a are mapped on ports $0, 1$ and 2 respectively, and o_s is mapped to all the ports $0, 1$ and 2. All the transfers can be satisfied using this assignment. The connections are illustrated in figure 4.1. It will be noted that a total of six switches will be required at the ports of the memory shown.

(adapted from chapter 4) □

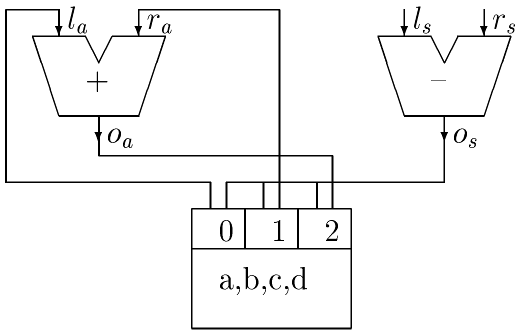


Figure 4.1: Connections to a three port memory. *(reproduced from chapter 4)*

Example 5.4 The creation of the problem instance for RIO from the PA problem of example 5.3 is now explained. Shown below are the transfers, the corresponding accesses and the variables for the RIO instance. The variables a, b, c and d are packed into a memory for which the PA needs to be done. The points $l_a, l_b, \ldots$ are as in example 5.3, (refer to figure 4.1 above).

	Transfers	Accesses	Registers
1.	$a = b + c;$	a_{11}, a_{21}, a_{31}	r_1, r_2, r_3
2.	$q = c + d, \quad b = p - q;$	a_{12}, a_{22}, a_{32}	r_4, r_5, r_6
3.	$d = a + r, \quad c = p - r;$	a_{13}, a_{23}, a_{33}	r_7, r_8, r_9
4.	$a = p + c, \quad b = q - r;$	a_{14}, a_{24}, a_{34}	r_{10}, r_{11}, r_{12}

The transfers for the RIO instance which are needed to construct the netlist are shown below.

1.	$r_1 \leftarrow o_a$	$l_a \leftarrow r_2$	$r_a \leftarrow r_3$
2.	$l_a \leftarrow r_4$	$r_a \leftarrow r_5$	$r_6 \leftarrow o_s$
3.	$r_7 \leftarrow o_a$	$l_a \leftarrow r_8$	$r_9 \leftarrow o_s$
4.	$r_{10} \leftarrow o_a$	$r_{11} \leftarrow o_s$	$r_{12} \leftarrow o_s$

□

The actual port assignment is constructed from the solution to the RIO instance. The set of accesses grouped together in a register are assigned to a port. Suppose that access a_{ij} corresponding to the transfer $r_l \leftarrow u_k$ is mapped to register r in the solution to the RIO instance. If this "register" r is mapped to port p then the point u_k will be connected to port p of the memory. This is how the port assignments are extracted from solution to the RIO instance. The kind of assignment will be determined by the register and switch costs chosen in equation 5.1, while solving the RIO instance. If all the accesses are either *read* or *write* then it is sufficient for the port to be either *read* or *write*, respectively. Otherwise the port has to have both *read* and *write* capability. Based on this formulation, the reg_min algorithm for RIO can be adapted for the port assignment aspect of MIO.

5.5.2 Memory Allocation for MIO

We have presented a technique based on RIO for port assignment. We now propose a scheme for memory allocation, that makes use of this PA technique to evaluate the cost of a memory configuration. As for RIO the initial trivial allocation and any feasible memory configuration reachable from the initial state constitute the design space. We call the space of feasible memory configurations D_M. A design in D_M will be represented a d_M. The initial trivial design is called d_{M_0}. Considering the nature of the initial allocation, it is evident that $d_{M_0} \equiv d_{R_0}$. The cost of a design in D_M may be obtained using a function f_{C_M}. This function measures the cost of the memories and the switching elements required for interconnect. The cost of a memory depends of the number of cells and the number of ports. A precise formula for the cost of a multi-port memory is given in section 8.2.1. Given a set of n variables, one can certainly construct a memory of n cells. However, the number of cells may be reduced by performing a localized merging of the variables grouped to form the memory. It may be noted that this variable merging will not change the port assignment. While computing f_{C_M} we compute the cost of each memory after performing such a variable merging.

The basic move in RIO was the merger of two registers. For MIO the basic move will be the introduction of a register into a memory, or to start building a new memory by grouping together two registers. A move results in a transformation of one design to another and is denoted by f_{T_M}. The objective is to find the a sequence of moves that lead to the design of minimum cost. As in case of the register-interconnect minimization problem, it is possible to formulate a branch and bound algorithm for performing register allocation. However, such an algorithm would take an unreasonably long time to do the design, and so we have developed a heuristic procedure for memory allocation, which we describe in the next section. It may be noted that memory interconnect optimization may also be preceded by a register-interconnect optimization, if desired. At each step a port assignment is performed to determine the quality of the assignment and the best candidate is chosen. This approach is made feasible by the polynomial time characteristics of the reg_min algorithm.

5.5.3 Algorithm for Memory Allocation

The algorithm described below takes the initial design d_{M0} and produces d_{Mf} as a result of grouping registers into memories, when possible. The actual number of memory cells required when a set of variables is packed into a memory is obtained through a pure register minimization applied on these variables. M records the final register groups.

procedure mem_alloc
$d_M = d_{M0}$
$R = \{ r \mid r$ is a register in $d_{M0} \}$
$M = \phi$
loop
{ if $R == \phi$ then break
  $r =$ any register of R
  $R = R - \{ r \}$
  $R_0 = \{ r \}$
  loop
  { if $R == \phi$ or $|R_0| \geq$ mem_size then break
    Find $r \in R$ that maximizes $p == f_{CM}(d_M)$
      $- f_{CM}(f_{TM}(d_M, R_0 \cup \{r\}))$ and leads to
      the usage of no more than *max_ports* ports;
    /* this is essentially application of the RIO formulation */
    Let p_{max} be this value of p
    if $p_{max} \geq 0$
    { $R_0 = R_0 \cup \{ r \}$
      $R = R - \{ r \}$
    } else
      break
  }
  if $| R_0 | > 1$
  { $M = M \cup R_0$

fig. ref.	max. cells perm-itted	max. port perm-itted	memory usage	num. of mem. used	multiplexer usage	num. of mux. used	num. of lines multi-plexed
5.2	6	3	3 port mem; 1 no. 2 port mem; 2 nos.	3	3 inp. mux.; 2 nos. 2 inp. mux.; 1 no.	3	8
5.3	8	3	3 port mem; 2 nos.	2	3 inp. mux.; 1 no. 4 inp. mux.; 1 no.	2	7
5.4	8	2	2 port mem; 3 nos.	3	3 inp. mux.; 2 nos. 2 inp. mux.; 1 no.	3	8
5.5	6	2	2 port mem; 3 nos.	3	3 inp. mux.; 1 no. 2 inp. mux.; 2 nos.	3	7
-	-	-	2 port mem [a]; 3 nos.	3	2 inp. mux.; 1 no. 3 inp. mux.; 3 nos.	4	11
-	-	-	3 port mem[a]; 2 nos.	2	2 inp. mux.; 4 nos.	4	8

[a] Quoted from result of designs in [64]

Table 5.3: Results for example 5.5.

$d_M \ = \ f_{TM}(d_M, \ R_0)$
}
}
$d_{Mf} \ = \ d_M$

5.6 Experimental Results for MIO

This scheme has been tried on the following example of [64].

Example 5.5

Step 1: $r_3 = r_1 + r_2$; $r_{12} = r_1 * r_7$
Step 2: $r_5 = r_3 - r_4$; $r_7 = r_3 * r_6$
Step 3: $r_8 = r_3 + r_5$; $r_{11} = r_{10}/r_5$
Step 4: $r_{14} = r_{11} + r_8$; $r_{15} = r_{12} * r_9$
Step 5: $r_1 = r_{14} - r_{13}$; $r_2 = r_{11} * r_{15}$

□

The maximum number of ports allowed per memory is varied between 2 and 3. The maximum capacity of a memory is varied between 6 and 8. For the case where only two port memories are allowed our designs multiplex between eight and seven lines, as against eleven lines for the corresponding design in [64]. When three port memories are allowed with a maximum capacity of eight registers, our design multiplexes seven lines, as against eight lines for the corresponding design in [64]. However, for the two port case we require on memory more. The design generated for a port constraint of two and a maximum memory size of six fares better than both the designs generated with corresponding constraints of (2,8) and (3,6). Thus, the technique does not exhibit monotonicity of performance, although it produces competitive results.

5.7 Conclusion

In this chapter we have considered the problem of jointly minimizing the register and multiplexer cost. We have handled this problem for the situation where operation to f.u. binding has been completed and the interconnection style is point to point. We have shown the interconnect optimization problem to be NP-hard in chapter 4. The RIO problem that we have considered here is possibly the simplest practical optimization problem in this category. For this problem we have proposed a heuristic algorithm and the results that we have obtained through our experimentation, we believe, are quite satisfactory. We have also considered the little more general problem of memory-interconnect optimization. For this problem we have developed an interesting formulation of the port assignment problem in terms of RIO. This was actually the key construction that helped us to derive the complexity of RIO using the complexity results of PA. We have noted that results for MIO are not nearly as encouraging as that of RIO. This is expected, considering the increase in complexity of the problem and the limitation of heuristic methods. For the larger allocation and binding problem which we have considered in chapter 8 we have, therefore, resorted to a method which is, in principle, more powerful than a purely heuristic technique. In chapter 6 we present compact graph based formulations for dual and triple port PA.

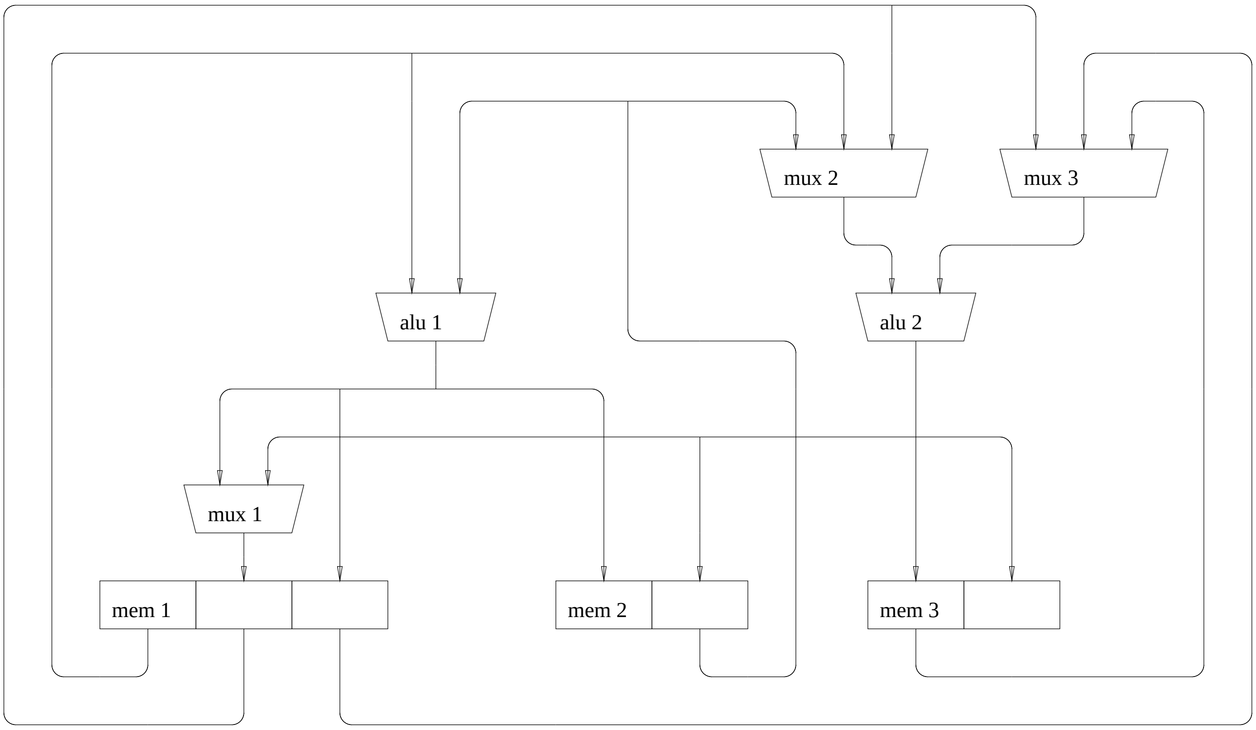


Figure 5.2: Data paths for memory size of six and three ports.

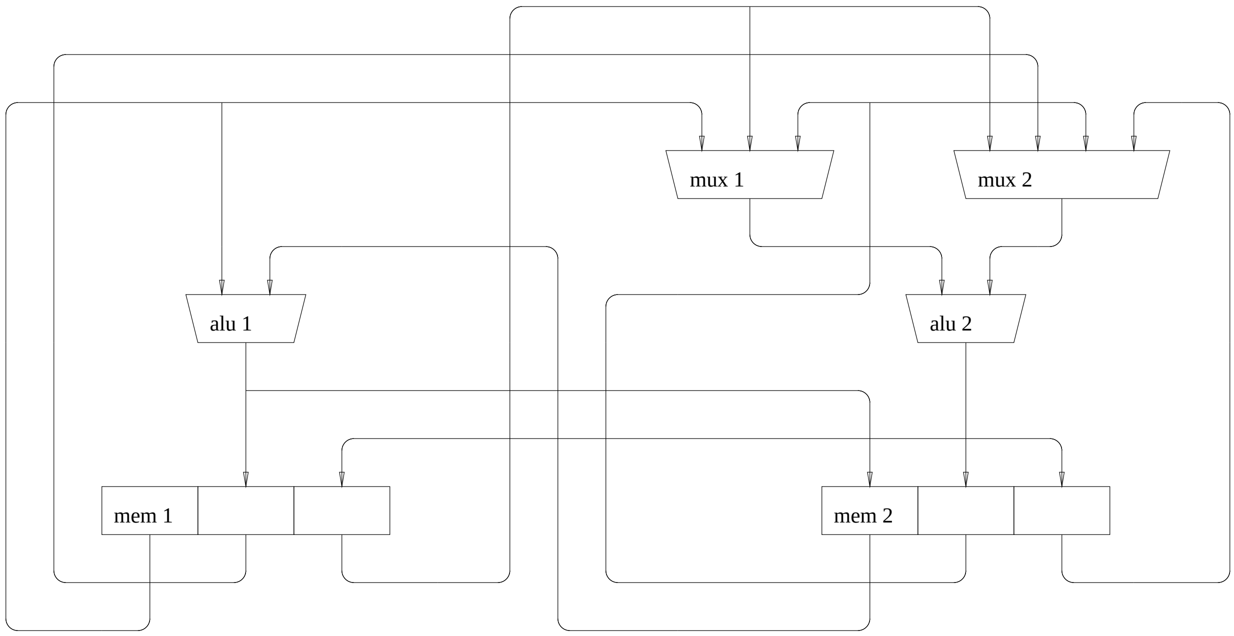


Figure 5.3: Data paths for memory size of eight and three ports.

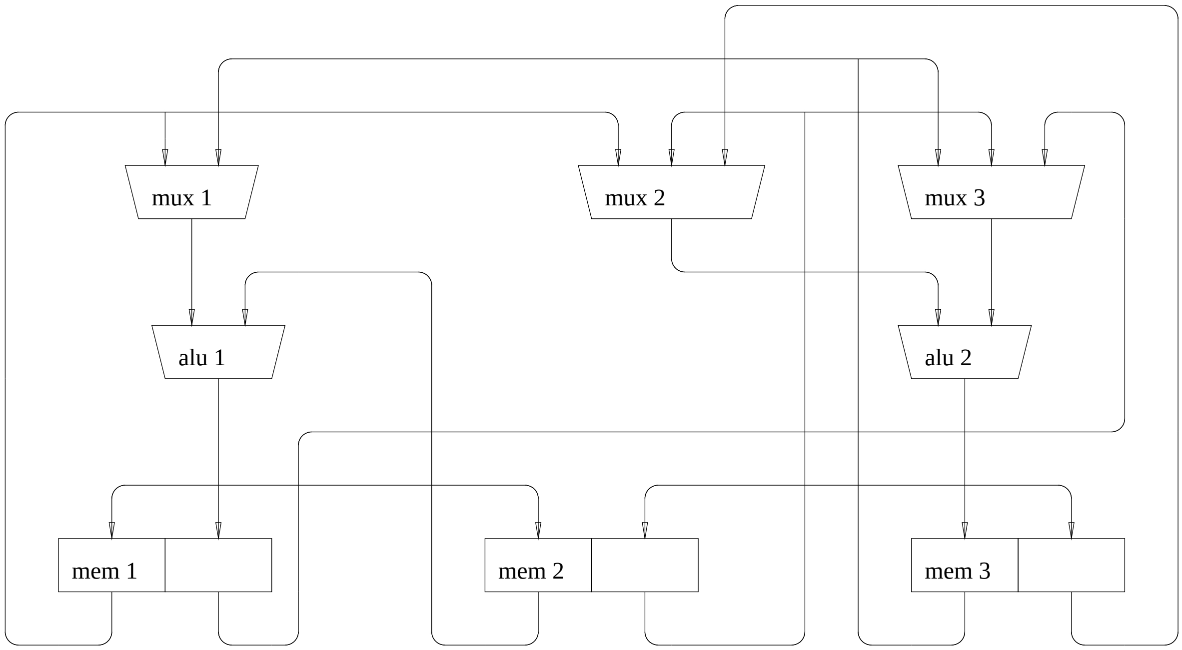


Figure 5.4: Data paths for memory size of eight and two ports.

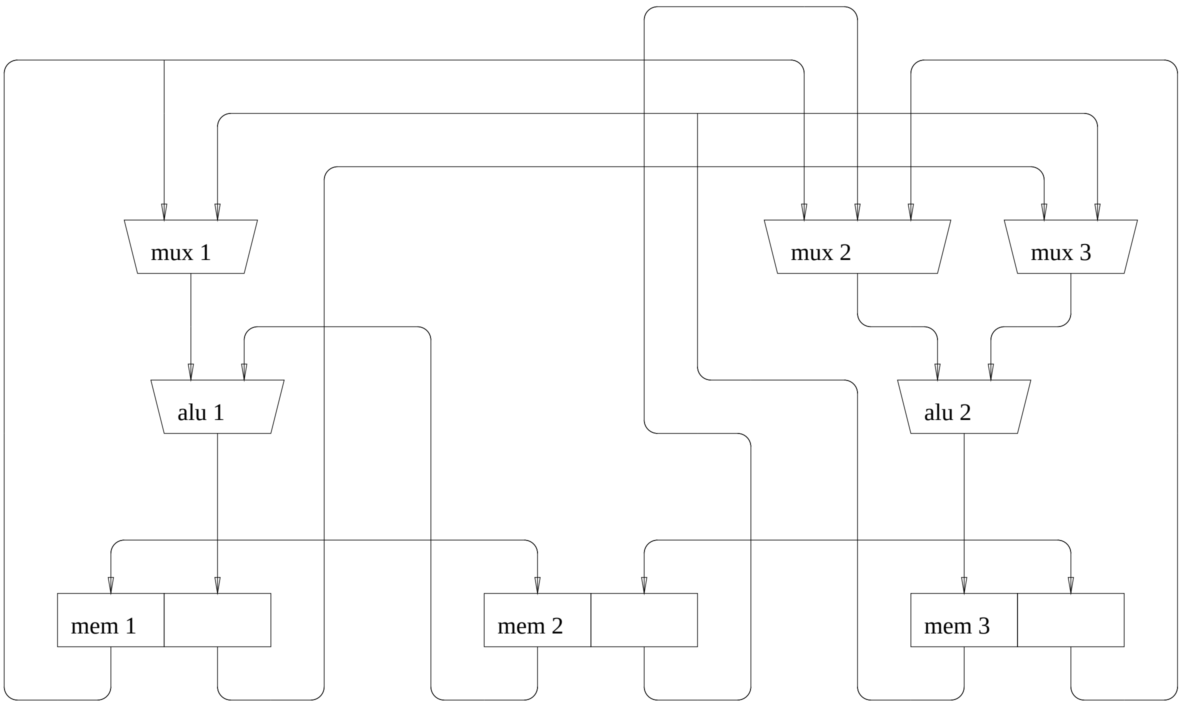


Figure 5.5: Data paths for memory size of six and two ports.

Chapter 6

Port Assignment of Dual and Triple Port Memories

6.1 Introduction

On-chip dual and triple port memories are increasingly being used as architectural elements. When multi-port memories are used it becomes necessary to carefully assign the accesses to its cells over its ports so as to minimize the cost of interconnecting the memory with other circuit elements. This is the origin of the port assignment problem. We have already introduced the port assignment (PA) problem in section 4.3, where we have examined the complexity of this problem. We shall now examine some implementational aspects of the PA problem for dual and triple port memories, and propose solutions to the PA problem for dual and triple port memories. In chapter 5 we have developed a method for PA in the context of memory-interconnect optimization. However, there are important differences between the two approaches. In this chapter we specifically consider PA problems for dual and triple port memories using a genetic approach, whereas in chapter 5 we determine the appropriate number of ports and the corresponding PA together using a heuristic algorithm. While the formulation of chapter 5 is a more general one, the formulations proposed here are more specific and efficient. The genetic algorithms developed for these formulations also perform better than the heuristic algorithm.

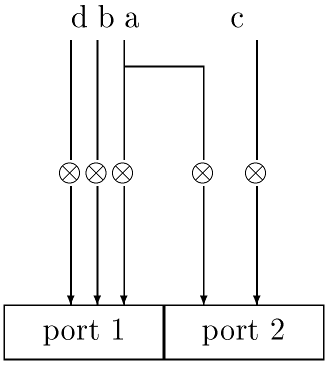
Some work on memory allocation and port assignment has been reported in [65], [66], [64] and [56]. Gregmap [66] is a comprehensive memory allocation and port assignment package. It uses an integer programming formulation. In this tool, the memory allocation and the port assignment tasks are handled separately. During memory allocation the number of registers packed in a single memory is maximized. The port assignment is performed optimally for each control step, separately. This method does not ensure a globally optimal port assignment. A different approach is presented in [56], where the port assignment problem is solved using a graph theoretic approach. The formulation is based on graph colouring, graph reduction and annotation of the graph to indicate the introduction of multiplexers for resolving access conflicts. This method has been shown to work well on individual test examples.

In chapter 4 we have derived several results on the complexity of both the PA problems. We have shown that both the problems are NP-complete. The relative approximation for triple port memory PA has also been shown to be NP-hard. These results have been derived by reducing NP-hard graph colouring type problems to special cases of these problems. For the dual port memory case we had obtained a correspondence with the Minimum Node Deletion (MND) problem [55]. This serves both as reduction of MND, an NP-hard problem, to this problem and also as its formulation. We have used this formulation for dual port memory PA in this chapter, and developed a GA to solve it. In proving the hardness of the port assignment problem for triple port memories (PA3U) in chapter 4, we did not require a detailed formulation of PA3U. We showed in section 4.3.3 that a special case of PA3UA is hard. We then used this to transform the result from PA3UA to PA3U. However, in this chapter we need to solve this problem and, therefore, a detailed formulation is required.

In section 6.2 we explain the implementational aspects of the port assignment problem. Graph theoretic formulations for both the dual and triple port memory PA have been developed in sections 6.3 and 6.7, respectively. The first problem has been formulated as the node deletion problem, a standard graph problem, whose solution using a genetic algorithm has been explained in section 6.4. A hyper graph based formulation has been used for triple port memory PA for which a GA has been developed and explained in section 6.8. PA is a computational intensive task and it is sometimes desirable to estimate the interconnect cost arising out of the assignment. For dual and triple port memory PA we have developed a probabilistic estimators based on a random graph model. These have been explained in section 6.5 and section 6.9, respectively. Section 6.6 explains the experimentation for the GA developed. The hyper graph based formulation for the triple port memory PA has been given in section 6.7. A GA has been developed for this problem too and is described in the succeeding section. The experimentation for this method is given in section 6.10. We have finally considered port assignment involving multi-cycle accesses and working with exact switching requirements. We present the formulation and the GA for this problem in section 6.11.

6.2 The Port Assignment Problem

A k-port memory has k sets of address and data access ports. The address port is used by the controller to determine the cell in the memory that to be accessed through the corresponding data port. It is assumed that the specification language does not permit concurrent writing to any memory cell from distinct sources. It is also assumed that the correctness of the other design stages will ensure that the same cell of the memory will not be attempted to be written through the multiple ports, simultaneously. Consider a set of registers $r_1, r_2, \ldots, r_n$, which have been mapped on a dual or a triple port memory M. During the different time steps some of these registers receive data from various points in the circuit, like outputs of the arithmetic logic units or alus, while other registers of the memory will transfer their data to various other points in the circuit, such as alu inputs, as part of the data transfers in each time step. For data



node a is deleted
node b and d are coloured 0, say, and
node c is coloured 1
port 1 corresponds to colour 0, while
port 2 corresponds to colour 1

Figure 6.1: Connections when a is deleted

transfers to take place, the relevant points of the circuit must be connected to one or both the ports of the memory. In general, the connection of a circuit point to a port of the memory will have a switch to either enable or disable the link between the port and that point. Links and switches form an important part of the interconnection elements present in the system. The switches (and links) add directly to the interconnect cost of the system. The set of switches that control the links which feed data to a specific point in the circuit very often, though not always, appear as a multiplexer at that point.

The following example illustrates how the various points may be connected to the ports to satisfy access of points in the circuit communicating with the memory during various time steps.

Example 6.1 Let r_1, r_2, r_3 and r_4 be registers packed into a dual port memory. Let a, b, c and d be some points in the circuit. Let us consider the following pairs of concurrent transfers of five time steps.

$$a \leftarrow r_1,\, b \leftarrow r_2; \quad b \leftarrow r_3,\, c \leftarrow r_4; \quad c \leftarrow r_2,\, d \leftarrow r_4;$$
$$a \leftarrow r_2,\, d \leftarrow r_3; \quad a \leftarrow r_3,\, c \leftarrow r_1;$$

If a is connected to both the ports, b and d are connected to the first port (say) and c is connected to the second port (figure 6.1), then it may be verified that all the transfers may be satisfied. For such a connection scheme five switches would be required. The transfers will also be satisfied if b and d are connected to both the ports and a and c are connected to separate ports (figure 6.2). Six switches are required in this scheme of connections. □

It is evident, from the example, that numerous connection schemes are possible, which differ in the number of links and switches that are used. We would, therefore, like to obtain an assignment which will lead to a connection scheme that requires the least number of switches (and links).

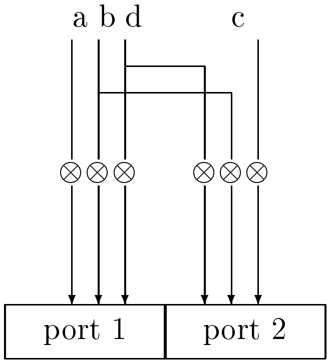


Figure 6.2: Connections when c and d are deleted

While solving either of the two PA problems our motivation will be two fold. Firstly, we would like to develop a robust algorithm to solve the problem. Secondly, we would also like to develop an estimator to only estimate the cost of the solution without actually solving the problem. This is because during the allocation and binding phase of high level synthesis, where PA comes as a sub-problem, many strategies (like branch-and-bound, greedy search, simulated annealing, genetic algorithms, etc.) require an estimate of the cost of a port assignment, before finding the actual solution, in order to explore the design space efficiently.

6.3 Formulation for Dual Port Memory PA

A dual port memory is equipped with two sets of address and data access ports. We assume that some registers have been packed into a dual port memory and a set of the circuit points have data transfers with the memory. It follows from the discussion in section 6.2 that the key to reducing interconnect complexity during port assignment is to have the least number of points in the circuit connected to both the ports. Since only two ports are involved, a point accessing the memory is connected to either one of the ports or to both the ports. It would be desirable to connect each point of the circuit communicating with the memory to only one of its ports because this would help to reduce the number of links and switches associated with the ports of the memory. But in some cases this might not be possible, and some of these points may have to be connected to both the ports. An inspection of the accesses of the memory by the circuit points in example 6.1 will reveal that it is not possible to satisfy the accesses, unless one or more of the points are connected to both the ports. We would, therefore, like to do the port assignment in such a way that a minimum of these points are connected to both the ports. This will lead to the minimization of the interconnect cost arising out of the port assignment. In this wsection we have formulated the problem of port assignment as a graph theoretic problem. The formulation has been done so that as a result of the assignment a minimum number of points will be connected to both the ports. This is similar to the formulation of PA2U in section 4.3.2. We repeat some aspects so that the chapter is self contained.

Let u and z be points accessing the memory in the same time step. Clearly u and z cannot access the memory through the same memory port in this time step. Such two points are said to have an *access conflict*. Two points having an access conflict must have connections to separate ports of the memory. Now suppose that z is connected to both the ports. Let u be connected to any one of the ports. It is easy to see that now the access conflict of u and z can always be resolved. This is because, no matter from which port u accesses M, z can always access M through the other port. On the basis of this argument, we also note that the access conflict of z with any other point accessing the memory will also be resolved if z is connected to both ports. This observation is fundamental to the formulation of the problem. Our formulation is as follows.

If two points in the circuit access the memory in a particular time step, this needs to be accomplished through different ports. We say that two such points are in conflict. A conflict graph of points accessing the memory is defined to be the graph where each circuit point corresponds to a vertex and an edge is present between a pair of vertices if their corresponding points are in conflict. An attempt is made to colour this graph with just two colours, $\{0,1\}$, say, with the constraint that no two vertices connected by an edge (that are in conflict) have the same colour. This will be referred to as the two colouring of the graph. A graph is said to be bipartite if its vertices can be partitioned into two sets such that no two vertices in the same set are connected by an edge. It is a well known result in graph theory that a graph is two colourable if and only if it is bipartite [67]. Also bipartiteness can be easily tested by checking up whether there are any *odd cycles* in the graph [67]. If the colouring is successful then vertices coloured as 0 are connected to one port, and the vertices coloured as 1 to the other. Such an assignment of ports will clearly satisfy all the data transfers to and from the memory. In general the graph will not be two-colourable and it will be necessary to delete sufficient number of vertices to make it 2-colourable or bipartite. A vertex that has been deleted will be connected to both the ports of the memory. As noted previously, the access conflicts of such a point with any other point can always be resolved.

This permits a straight forward formulation of the problem. In order to make the graph 2-colourable at minimum cost, the least number of vertices should be deleted from the conflict graph, which is the Minimum Node Deletion problem (MND). Example 6.2 briefly illustrates the application of this formulation.

Example 6.2 The conflict graph obtained for the transfers in example 6.1 is shown in figure 6.3. It contains *odd cycles* and therefore it is not 2-colourable. However, the deletion of vertex a makes the graph bipartite. The deletion of vertex c would provide a similar solution. However, deletion of either of the vertices b or d alone does not suffice to make the graph bipartite. The connections of figure 6.1 result from the deletion of vertex a. □

We now examine the relationship between the switch cost and the nodes deleted. We make an assumption that will be usually satisfied in practical situations. We assume that a point in the circuit reading from the memory will also be a destination for other sources in the circuit. We will also assume that, in general, each port will be connected to more than one point in the circuit (reading or writing from the memory). In this

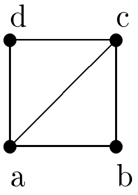


Figure 6.3: Conflict graph for circuit points

case connection of a point to both the ports leads to the use of an additional switch or multiplexer channel. Thus deletion of n nodes corresponds to the use of n additional switches. In section 6.11 we consider a more general treatment of the problem where this assumption is not made. There we shall, however, have to sacrifice the graph theoretic formulation and its advantages.

6.4 GA for the Minimum Node Deletion Problem

In view of the complexity of the PA problems we have chosen to apply the genetic algorithm (GA) technique to solve them. The GA technique has been applied for solving several practical hard problems [68]. In the next section we briefly introduce the genetic paradigm. A more detailed description of GA may be found in the appendix. The GA technique will also be used for scheduling as well as allocation and binding later.

6.4.1 The Genetic Paradigm

A solution methodology called the Genetic Algorithm, proposed by John Holland [69], has been found suitable for hard optimization problems. In this work, this methodology has been considered for determining the smallest set of vertices to be deleted to make a graph bipartite. The characteristic feature of the GA is that it works on a population of candidate solutions. GA simulates the process of adaptation in which the solutions in a population undergo a number of changes in the genetic character, over a number of generations. John Holland proposed that such a simulation of the adaptation process can be an efficient approach for solving hard optimization problems.

The general structure of the GA is as follows [70].

step 1 : Create the initial population of solutions.

step 2 : Evaluate the population of solutions for the fitness function value.

step 3 : As per a reproductive plan generate offspring solutions from pairs of parent solutions by the *crossover*. The parent solutions are selected as per a selection policy. The crossover operator helps the exploration of the search space. In order to ensure that the search space explored is not closed under crossover, another genetic operator called the *mutation* is used on the working population to perturb a given solution. It is used at a low level. These genetic operators and the parent selection policy constitute a reproductive plan.

step 4 : Evaluate the offspring solutions.

step 5 : The replacement policy is used to determine which solutions in the population will be replaced by the newly generated solutions in the current generation. The *elitist* policy is one where a fraction of the best solutions in the current population is never replaced.

step 6 : Repeat the steps 3 to 5 for a number of generations.

step 7 : Report the best solution in the population as the final solution.

Genetic algorithms rely on a suitable encoding of the solution. The encoded solutions are usually represented in bit strings on which the conventional crossover operations are performed. Finding a suitable encoding scheme which will work well with these crossover operations has been a difficult task. The GA is an enumerative method in which some new solutions are enumerated in successive generations. In terms of GA, the solutions in the population collectively represent a number of small building blocks, referred to as *Schemata*. The application of the genetic operators according to the reproductive plan, should lead to the juxtapositioning of compatible and better fit small building blocks in successive generations. The processing of many schemata concurrently by the GA is referred to as *implicit parallelism*. It has been shown in [44] that the lower bound on the number of schemata processed in parallel is $O(N^3)$, N being the population size, for the conventional bit string encoding and single point crossover [68]. More details on the basics of the GA technique may be found in [69, 70]. A more elaborate explanation of GA is presented in Appendix A.

6.4.2 Algorithm for MND

Solution representation Convention bit string representations sometimes mask the structure inherent in the solution. Davis [70] pointed out that employing non-bit-string solutions for specific optimization problems is advantageous. For MND we have found it convenient to represent the solution directly as three sets. The first two sets contain the vertices corresponding to each of the two colours. The third set contains vertices that could not be two coloured and which are chosen for deletion.

Example 6.3 For the graph of figure 6.3 one solution could be $< \{b, d\}, \{c\}, \{a\} >$. This corresponds to the connections of figure 6.1. The last set of the tuple corresponds to the set of deleted vertices which are connected to both the ports. Another solution could be $< \{a\}, \{c\}, \{b, d\} >$. This corresponds to the connections of figure 6.2. □

Fitness function The fitness function is defined as

$$g = |set \;\; of \;\; deleted \;\; vertices| \tag{6.1}$$

Minimization of g is the objective.

Initial population generation Each member of the initial population is a randomly generated valid solution. While generation a solution, each vertex of the graph is tested for possible membership in one of the two partially constructed colour classes. In case of a failure in inclusion to one colour class, the membership for the other colour class is checked. In case of a repeated failure the vertex is marked for deletion. The sequence in which the vertices are visited while constructing a solution is also random.

Reproductive plan The parent selection policy, crossover and mutation operations constitute a reproductive plan. For a particular generation we have selected two parent solutions for crossover to generate each child solution, randomly without replacement from the current population. Only one offspring has been generated as a result of a single reproduction. The number of reproductions performed in one generation is determined by the crossover rate.

The crossover is performed as follows. The offspring solution first inherits a subset of a colour class from one of the parent solutions. During crossover, larger colour classes are chosen for inheritance with a higher probability, while smaller ones are selected with a lower probability. The inherited class is now augmented with uncoloured vertices according to the algorithm in figure 6.4. The augmentation is based on a graph colouring algorithm presented in [71]. The algorithm is successively applied to the two inherited colour classes. The augmentation algorithm is as follows. Let V be the set of vertices of the graph. Let the initial inherited colour class on which the algorithm is applied be X. $\Gamma(X)$ is defined as the subset of $V - X$, such that for each element of the subset there is an element in X to which it is connected by an edge. The effect of the steps (1) and (5) of the augmentation algorithm is to remove from Y all those vertices which have an edge with at least one vertex of X. The newly defined Y has the property that any of its vertices can be augmented to X. The process of augmentation continues till the set Y becomes empty. The vertex in Y that is to be selected is determined by the simple heuristic of step (3) of the algorithm.

After the first colour class of the offspring is formed, the second colour class is also formed similarly. The set from which the second colour class is inherited is as follows. Let P_1 and P_2 be the two parents. Let S_{11} and S_{12} be the two sets in P_1. Similarly, let S_{21} and S_{22} be the two sets in P_2. Suppose that the first colour class had been formed from S_{11} of P_1. Let

$$c_0 = \frac{|S_{11} \cap S_{21}|}{|S_{11} \cup S_{21}|} \qquad and \qquad c_1 = \frac{|S_{11} \cap S_{22}|}{|S_{11} \cup S_{22}|}.$$

The values c_0 and c_1, $0 \leq c_0, c_1 \leq 1$ represent the affinity of S_{11} with S_{21} and S_{22} of P_2, respectively. Let $S = S_{2(i+1)}$, such that $c_i \leq c_{1-i}, i \in \{0, 1\}$. S is the colour class of P_2 which is less affine to S_{11}. Normally the second colour class of the offspring is inherited from S, otherwise inheritance is from S_{12}. The vertices that could not be included in the two colour classes of the offspring are placed in the third set for deletion. This method of construction ensures that each solution constructed is a valid solution.

During crossover, inheritance of only a part of the colour class may be considered equivalent to the process of mutation. The *mutation rate* instead of being kept fixed, is

```
1.  Y = V − Γ(X)
2.  while (Y ≠ ∅)
3.  {   among all y ∈ Y let x have
        the minimum degree in Y
4.      X = X ⋃ {x}
5.      Y = Y − ({x} ⋃ Γ(x))
6.  }
```

Figure 6.4: The augmentation algorithm

varied with the standard deviation of the fitness value of the candidate solutions. Should the fitness function values tend to become uniform the mutation rate goes up.

Replacement policy In our implementation all the offspring generated in the current generation replace the maximum cost solutions in the current population. This corresponds to the survival of every new offspring generated for at least one generation. Any existing better solution found survives since the worst solutions are always replaced. This corresponds to an elitist policy.

6.4.3 Deceptibility of the Crossover

It has been shown in [36] that if the crossover operation is free of type II deceptibility, then the GA may be expected to lead to the optimal solution. The crossover would be free of type II deceptibility, if on crossing two solutions with high fitness value, the resulting new solution also has a high fitness value [36]. The crossover used here has not been proved to be strictly free of type II deceptibility, but it is likely to be so. We show this by probabilistic arguments. First a probabilistic analysis of the augmentation algorithm is presented.

Let the two colour classes be B_1 and B_2 and let the third set be D. The analysis is applicable for random graphs satisfying the following:

1. $|B_1| = |B_2| = m$.
2. $|D| = k$, thus $|V| = 2m + k$.
3. By definition B_1 and B_2 are independent, (that is, there are no edges between any pair of elements of a particular set). An edge may be present between a member of B_1 and a member of B_2 with probability p. An edge may be present between a member of B_1 and a member of D with probability p. An edge may be present between any two members of D with probability p.
4. With probability $q = 1 - p$ the edge in question is absent.

Let $X \subset B_1$ and $|X| = r$.

Y, as computed in step (1) of the augmentation algorithm is $(B_1 - X) \bigcup (B_2 - \Gamma(X)) \bigcup (D - \Gamma(X))$.

It may be shown that $|B_2 - \Gamma(X)| \approx mq^r$.

Similarly it may also be shown that, $|D - \Gamma(X)| \approx kq^r$.

Let $d_Z(v)$ be the expected degree of $v \in Z \cap Y,\ Z, Y \subseteq V$, in Y.

Let $d_{B_1} = d_{B_1}(v) \approx mpq^r + kpq^r$. This is the expected degree of a vertex of $B_1 \cap Y$ in Y.

Let $d_{B_2} = d_{B_2}(v) \approx (m - r)p + kpq^r$. This is the expected degree of a vertex of $B_2 \cap Y$ in Y.

Let $d_D = d_D(v) \approx (m-r)p + mpq^r + (kq^r - 1)p$. This is the expected degree of a vertex of $D \cap Y$ in Y.

$$d_2 = d_D - d_{B_1} = (m - r)p - p = (m - (r + 1))p \tag{6.2}$$

$$d_1 = d_{B_2} - d_{B_1} = p(m(1 - q^r) - r) \tag{6.3}$$

It is desirable that $d_1 > 0$ and $d_2 > 0$, for this will ensure, with high probability (**whp** [71]), that if X turns out to be a subset of B_1 or B_2 then it will be augmented by members of B_1 or B_2, respectively. To satisfy $d_2 > 0$ it is necessary that $r < (m - 1)$. Thus when the subset is being augmented with the last element, the algorithm is not expected to guarantee the selection of the correct element. However, in the stochastic environment of the GA this does not pose a serious problem.

To satisfy $d_3 > 0$ it is necessary that

$$m > \frac{r}{1 - q^r} \tag{6.4}$$

For $r = 1$, it is necessary that $m > \frac{1}{p}$. For somewhat large values of m and not too sparse graphs this will be satisfied. Also, for $r = m - 1$,

$$m - \frac{m-1}{1 - q^{m-1}} = \frac{1 - mq^{m-1}}{1 - q^{m-1}}$$

Again for somewhat large values of m, this expression is positive. Now consider the function $\frac{x}{1-q^x}$, $x > 0$.

$$\frac{d}{dx}\left(\frac{x}{1 - q^x}\right) = \frac{1 - q^x(1 - x\ln q)}{(1 - q^x)^2} = \frac{1 - e(\ln(ez^x)/(ez^x))}{(1 - q^x)^2}, where \quad z = 1/q.$$

Depending on the value of q the derivative may be negative for small values of x, for larger values of x it is positive and approaches 1. Thus, if (6.4) is satisfied for $r = 1$ and $r = m - 1$, then it will be satisfied for all intermediate values of r. This, in general, will not be true for all members of the population. However, in the stochastic environment of GA it will be satisfied by at least a few members of the population.

It is reasonable to assume that solutions whose colour classes are *close* to the colour classes of an optimal solution will have relatively high fitness values. It has also been ensured that the augmentation algorithm will, *whp*, augment an inherited colour class which is a subset of an optimum colour class with the appropriate vertices. Thus, solutions with high fitness values, when combined through crossover should also result in solutions with high fitness values.

6.5 Estimation of Minimum Number of Nodes Deleted

By definition the expected number of nodes to be deleted to render a graph $G_{n,p}$ bipartite is

$$E(d) = \sum_{i=0}^{n-2} iP(i,n,p),$$

where $P(i,n,p)$ is that probability that the deletion of i nodes will make a graph $G(n,p)$ bipartite. The analytic expression for $P(i,n,p)$ is too complex to be of any practical value. In view of this we take an alternative approach to estimate $E(d)$.

Let $G_{n,p}$ be a random graph, with n nodes and each of the $\frac{n(n-1)}{2}$ edges being present with probability p. Consider the possible bi-partitioning of m nodes of this graph. Let $q = (1-p)$ be the probability that the edge is absent. Let $l = \lfloor \frac{m}{2} \rfloor$. Let k nodes be in one partition, $(m-k)$ remaining nodes be in the other partition. If these two partitions are bipartite then there will be no edge between a pair of nodes from either of these partitions. Consider now that there is a set D of d nodes, distinct from the chosen m nodes. This set represents the d nodes which must be deleted to make $G_{n,p}$ bipartite, where $n = m + d$. The set D should be minimal in the sense that each node of D must be connected by edges to at least one node of each partition. (Unless D is minimal, some of the nodes in D may be easily augmented to one the two sets, and a smaller number of nodes would have to be deleted. This would make the estimate quite useless.)

Using the probability of an edge being absent between a pair of nodes from both these partitions, we compute two quantities S_1 and S_2 such that:

$$S_1 = \sum_{k=1}^{\lfloor \frac{m}{2} \rfloor - 1} \binom{m}{k} q^{\frac{k(k-1)}{2} + \frac{(m-k)(m-k-1)}{2}} (1-q^k)(1-q^{(m-k)})^d$$

S_1 represents the expected number of bipartite partitions of m nodes when k are in one partition and $m-k$ are in the other, $k = 1..\lfloor \frac{m}{2} \rfloor - 1$, and

$$S_2 = \binom{m}{l} q^{\frac{l(l-1)}{2} + \frac{(m-l)(m-l-1)}{2}} (1-q^l)(1-q^{(m-l)})^d$$

S_2 represents the expected number of bipartite partitions of m nodes when l are in one partition and $m-l$ are in the other. S_2 is computed separately to avoid double counting when m is even. The expected number of bipartite partitions of $G_{m,p}$ is

$$N_{m,p} = \left\{ \begin{array}{ll} S_1 + S_2 & \text{if m is odd} \\ S_1 + \frac{S_2}{2} & \text{if m is even} \end{array} \right\}.$$

For $G_{n,p}$ the expected number of bipartite partitions if d nodes are deleted at random is

$$M_{n,p,d} = \left(\begin{array}{c} n \\ d \end{array} \right) N_{n-d,p}$$

.

Consider the minimum value of d for which $M_{n,p,d} = b \geq 1$. Suppose that $b \approx 1$. If d nodes are deleted from a graph $G_{n,p}$, on an average the number of bipartite partitions will be b. Some of these graphs will actually have more than b bipartite partitions. Let there be a particular graph G_1, of the type $G_{n,p}$, with $L \gg b$ bipartite partitions, when d nodes are deleted from it. The presence of such a graph would tend to drive the expected number of bipartite partitions to some value $b' > b$. To balance this effect there must be approximately $L-1$ graphs of the type $G_{n,p}$, for which the number of bipartite partitions is zero, which may be associated with G_1 so that the average number of bipartite partitions is restored to b. These graphs, which have no bipartite partition, will require more than b nodes to be deleted. On an average, therefore, at least b nodes will have to be deleted from each graph of the type $G_{n,p}$. Thus d serves as a lower bound on the average number of nodes to be deleted to render a graph $G_{n,p}$ bipartite.

A pragmatic conjecture It is desirable to have a value of d which will serve as a more pragmatic estimate of the expected number of nodes to be deleted to make a graph $G_{n,p}$ bipartite. Such a pragmatic estimate has been designed empirically as follows: Let $B_1 = M_{n,p,d}$, for the minimum value of d such that $B_1 \geq 1$. Let $B_2 = M_{n,p,d}$, for the value of d which maximizes B_2. Let $B_P = \sqrt{B_1 B_2}$. Find the maximum value of $d = d_P$ for which $M_{n,p,d} \leq B_P$. This has been used as an estimate of $E(d)$. It has been experimentally observed that this estimate closely matches the cost of the actual solutions returned by the GA.

6.6 Experimentation for Dual Port Memory PA

The GA for MND is referred to as GA2 here. The experimentation consists of two parts. The first part of the experimentation deals with the testing of GA2, while in the second part the quality of the estimate has been tested.

GA2 has been implemented in C in the UNIX environment on a SUN 3/280. It has been tested on graphs of both small and relatively large number of vertices. While testing GA2 on the smaller graphs, it has been possible to compare the results against the exact solutions. This testing has been done on random graphs of the type $G_{n,p}$, described in section 6.5. Twelve sets of random graphs of ten, twelve, fourteen and sixteen vertices with edge probabilities of 0.3, 0.5 and 0.7 were generated. The testing for each set was carried out on thirty graphs of that type. For these small graphs (*upto 16 vertices*), in each case the GA was able to obtain the optimal solution.

For the relatively larger graphs it was not feasible to find the exact solution for comparing the result obtained by GA2. Therefore, a different method of testing has been employed here. GA2 was now tested against random graphs which have been generated such that the upper bound on the number of nodes to be deleted is known. The method of constructing random graphs with a known upper bound on the number of nodes to be deleted has already been explained in section 6.5. The test results for graphs of 20, 30, 40 and 60 vertices and various edge probabilities have been presented in table 6.1. In this table the first column shows the number of vertices, $|V|$, in the graph and the second column the edge probability p'. For each $< |V|, p' >$ combination of a row of the table thirty random graphs of that type were generated so that no more than the number of vertices specified in the last column, the upper bound, needs to be deleted to render the graph bipartite. The third column is the average number of vertices deleted by GA2. It will be observed that deletion of GA2 is very close to the upper bound, occasionally doing slightly better. A similar method of testing, for another graph problem (the graph 3-colourability problem), has been used in [72].

In the second part of the experimentation the quality of the estimate has been tested against results obtained by running GA2. Random graphs of the type $G_{n,p}$ have been used here. The test results for graphs of 10, 20, 30, 40 and 60 vertices and edge probabilities 0.082, 0.107, 0.154, 0.250 0.400 and 0.500 have been presented in table 6.2. In this table the first column shows the number of vertices, $|V|$, in the graph and the second column the edge probability p. For each $< |V|, p >$ combination table thirty random of the type $G_{n,p}$ were generated. The third column is the lower bound estimate while the fourth column is the pragmatic estimate. The last column is the actual deletion by GA2. We observe that the pragmatic estimate comes very close to the deletion by GA2.

6.7 Formulation for Triple Port Memory PA

A brief introduction to the triple port memory PA has already been given in section 4.3.3. There we have taken a simplistic view of this problem as that was sufficient to derive the hardness results for this problem. We now examine the problem in more detail so as to find a formulation that will lead to a solution of this problem.

A triple port memory has three sets of address and data access ports. We assume, again, that some registers have been packed into a triple port memory which have data transfers with some of the points in the circuit. As usual we would like to perform the assignment so that the total number of links of the circuit points to the ports of the memory is minimized. Here too, we develop a graph theoretic formulation. But unlike the dual port memory PA case we are not able to map it onto a conventional graph problem. In case of a three port memory either one or two or three points of the circuit can access the memory in one time step. The access of the memory by a single point

no. of. nodes	edge prob.	deletion [a] by GA2	cost u.b.
20	0.082	0.000	0
20	0.107	0.000	0
20	0.154	0.903	1
20	0.250	3.483	4
20	0.400	6.903	7
20	0.500	8.903	9
20	0.650	10.70	11
30	0.082	0.000	0
30	0.107	1.000	1
30	0.154	3.677	4
30	0.250	8.677	9
30	0.400	13.61	14
30	0.500	16.80	17
30	0.650	19.70	20
40	0.082	1.709	2
40	0.107	3.806	4
40	0.154	8.516	9
40	0.250	15.35	16
40	0.400	22.29	23
40	0.500	25.48	26
40	0.650	28.77	29
60	0.082	7.451	8
60	0.107	12.38	13
60	0.154	20.38	21

[a] The deletion in each line has been reported as the average obtained by running GA2 on 30 individual random graphs with known upper bounds.

Table 6.1: Performance of GA2 on random graphs where an upper bound on the number of nodes to be deleted is known

no. of. nodes	edge prob.	l.b. est.	prag. est.	deletion by GA2 [a]
10	0.082	0	0	0.067
10	0.107	0	0	0.290
10	0.154	0	0	0.548
10	0.250	0	0	0.870
10	0.400	2	2	2.290
10	0.500	2	3	2.935
10	0.650	4	4	4.419
20	0.082	0	0	0.516
20	0.107	0	1	1.677
20	0.154	1	2	2.709
20	0.250	4	5	5.516
20	0.400	7	8	8.870
20	0.500	9	10	10.161
20	0.650	11	12	12.226
30	0.082	0	1	2.612
30	0.107	1	4	4.161
30	0.154	4	7	6.935
30	0.250	9	11	11.129
30	0.400	14	16	16.032
30	0.500	17	18	18.064
30	0.650	20	21	20.935
40	0.082	2	4	5.806
40	0.107	4	7	8.451
40	0.154	9	12	12.064
40	0.250	16	18	18.322
40	0.400	23	24	24.000
40	0.500	26	27	26.741
40	0.650	29	30	29.712
60	0.082	8	13	14.903
60	0.107	13	18	18.838
60	0.154	21	25	25.129
60	0.250	31	34	33.419
60	0.400	40	42	41.193
60	0.500	44	45	44.870
60	0.650	48	49	48.667

[a] The deletion in each line has been reported as the average obtained by running GA2 on 30 individual random graphs of the type $G_{n,p}$.

Table 6.2: Comparison of the estimator against the number of nodes deleted by GA2.

of the circuit in one time step does not require any special consideration. Access of the memory by two or three points, in a time step, gives rise to two different kinds of access conflicts. When three points access the memory in the same time step, it is necessary that our connection scheme of these three points to the ports should permit us to satisfy these access through distinct ports. Similarly, when two points access the memory in one time step, it should be possible to connect these points to two distinct memory ports. It is, therefore, necessary to represent these conflicts so that these constraints are properly reflected.

We try to construct a hyper graph to reflect the constraints imposed by these accesses. Let there be n points accessing the triple port memory. The conflict graph is a hyper graph H consisting of n vertices. Let V be the set of vertices of H, $|V| = n$. The edges of H may include two vertices for access by two points in a particular time step. Such a hyper edge will be called a 2-edge. The access by three points, in the same time step, is represented by a hyper edge including the corresponding three vertices. Such a hyper edge is called a 3-edge. Thus if points p_1 and p_2 access the memory in a time step then H will have an 2-edge $< p_1, p_2 >$. Similarly, for access by points p_1, p_2 and p_3 the 3-edge $< p_1, p_2, p_3 >$ will be present in H. Each port is represented by a colour. For the three ports we arbitrarily designate three colours R, G and B. The vertices in H must, therefore, be assigned colours. To model the connection of a point to multiple ports we permit a vertex to by assigned any non-empty combination of the three designated colours. For a vertex $v_i \in V$, let s_i be the set of colours assigned to it.

Definition 6.1 *A set of colours $s \subseteq \{R, G, B\}$ will be called a 2-colour if $|s| = 2$.*

The colours assigned to the vertices should satisfy the following properties so that the accesses to the memory will be satisfied.

1. If vertices v_1 and v_2, coloured with s_1 and s_2, such that $|s_1| = |s_2| = 1$ and $s_1 = s_2$, then v_1 and v_2, together, should not be part of any edge in H. This is to ensure that two points connected to the same and only port of the memory are free of all access conflicts.

2. If v_1 and v_2, coloured s_1 and s_2 are connected by a 2-edge, then there should be colours c_1 and c_2 such that $c_1, c_2 \in \{R, G, B\}, c_1 \neq c_2$, and $c_1 \in s_1$ and $c_2 \in s_2$. This ensures that two vertices that access the memory simultaneously are connected to separate ports so that their access can be satisfied.

3. Similarly, if v_1, v_2 and v_3, coloured s_1, s_2 and s_3 are connected by a 3-edge, then there should be colours c_1, c_2 and c_3 such that $c_1, c_2, c_3 \in \{R, G, B\}, c_1 \neq c_2 \neq c_3$, and $c_1 \in s_1, c_2 \in s_2$ and $c_3 \in s_3$.

It is evident from the above definition of colouring that a 3-edge implies each of its embedded 2-edges. The colouring of the graph remains unaffected with the omission of the implied 2-edges from H. Thus a conflict hyper graph (CHG) can be reduced by removing the implied 2-edges. ¿ From now on we shall assume that the CHG is in a reduced form. Example 6.4 shows the construction of a CHG from a sequence of concurrent transfers.

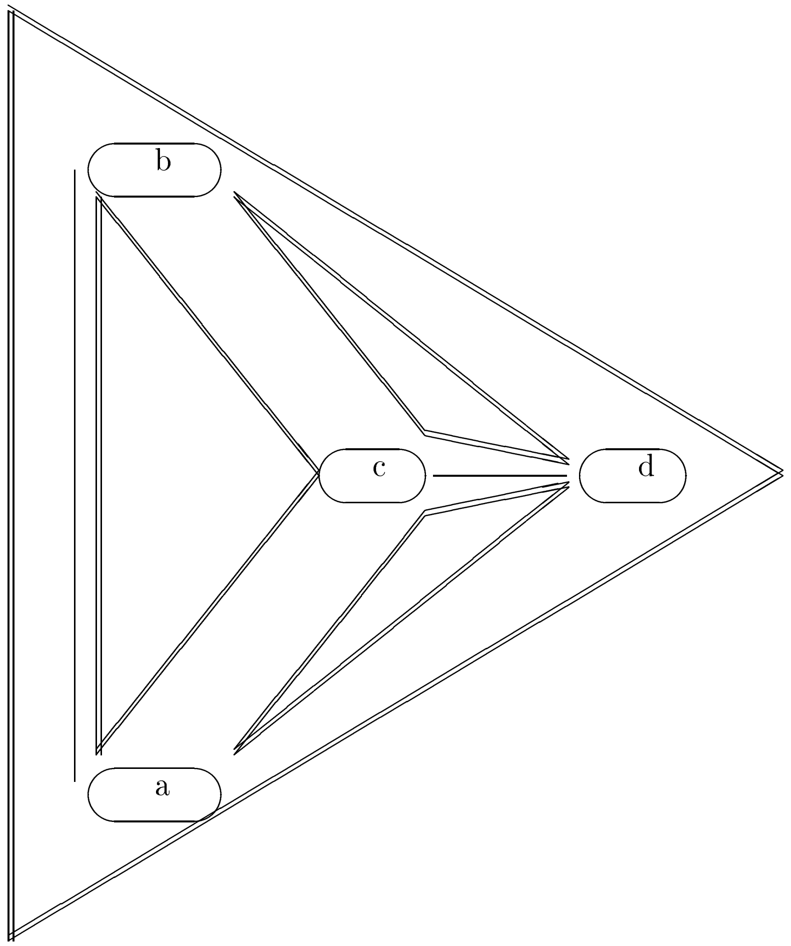


Figure 6.5: An un-reduced CHG

Example 6.4 Assume that r_1, r_2, r_3 and r_4 be registers packed into a triple port memory. Let a, b, c and d be some points in the circuit. Let us consider the following pairs of concurrent transfers of six time steps.

$$a \leftarrow r_1,\ b \leftarrow r_2,\ c \leftarrow r_3$$
$$b \leftarrow r_3,\ c \leftarrow r_4,\ d \leftarrow r_1$$
$$a \leftarrow r_2,\ d \leftarrow r_3,\ c \leftarrow r_5$$
$$d \leftarrow r_2,\ a \leftarrow r_4,\ b \leftarrow r_1$$
$$a \leftarrow r_3,\ b \leftarrow r_1,$$
$$c \leftarrow r_2,\ d \leftarrow r_4$$

The 3-edges resulting from these transfers have been shown in figure 6.5 in double lines. It may be noted that, in this case, all the 2-edges are implied by one or more 3-edges. □

It may be noted that if a point is connected to all the three ports then its access conflict with any other point accessing the memory can always be resolved. Therefore, the vertex associated with this node in the conflict hyper graph can be considered to be deleted. This is similar to the deletion of a node from the conflict graph for the dual port memory PA when the point associated with a vertex is connected to both the ports. However, if a point is connected to only two ports of a triple port memory then some access conflicts may still remain to be resolved and so the vertex cannot be considered to be deleted. This somewhat complicates the colour assignment to the vertices of the CHG. The colouring process is now explained.

Since the colouring of the CHG is relatively more involved, first a check is made to see which vertices may be assigned single colours using graph colouring algorithms for simple graphs. For this purpose a projection of the CHG H to a simple graph G is defined as follows. G has the same (isomorphic) set of vertices as H. Two vertices in G are connected by an edge if they are connected by a 2-edge or a 3-edge in H. The projection of the CHG of figure 6.5 is shown in figure 6.6. We attempt to colour the simple graph G with the three colours R, G and B. If the colouring is successful then the port assignment may be readily performed with each point being connected to just one port, of the corresponding colour. If the colouring is unsuccessful, then some vertices still remain uncoloured. The corresponding circuit points will have to be connected to two or three ports, i.e. they will have to take two or three colours. The single colours being R, G and B, the two colours will be $\{R, G\}$, $\{G, B\}$ and $\{R, B\}$, to be referred to as *RG*, *GB* and *RB*, respectively. RGB will be used to denote the 3-colour $\{R, G, B\}$. The graph shown in figure 6.6 is not colourable using three colours. Example 6.5 depicts a situation where the projection of the CHG is colourable using three colours.

Example 6.5 Assume that r_1, r_2, r_3, r_4 and r_5 have been packed into a triple port memory. Let a, b, c and d be some points in the circuit. Let us consider the following sequence of concurrent transfers in two time steps.

$$b \leftarrow r_3, c \leftarrow r_4, d \leftarrow r_1$$
$$a \leftarrow r_2, d \leftarrow r_3, c \leftarrow r_5$$

The hyper graph and its projection to a simple graph (which is colourable using three colours) are shown in figure 6.7. □

Definition 6.2 *Given a 2-colour $s, s \subset \{R, G, B\}$, the colour c, such that $c = \{R, G, B\} - s$, will be called the resolving colour of s.*

Thus the resolving colour of RG is B.

Definition 6.3 *Suppose that a vertex has been assigned a 2-colour s_1. Vertices v_1 and v_2 will be said to constitute a c-conflict if s_2, coloured v_2 satisfies $s_2 \subseteq s_1$ and v_1, v_2 and some other vertex v form a 3-edge.*

Definition 6.4 *Let V be the set of vertices of a CHG and $< v_1, v_2 >, v_1, v_2 \in V$. The set of vertices $O = \{v | v \in V, < v_1, v_2, v >$ is 3-edge and v_1, v_2 make a c-conflict $\}$, will be called the orbit of $< v_1, v_2 >$.*

Example 6.6 depicts a situation where two vertices of a CHG make a c-conflict.

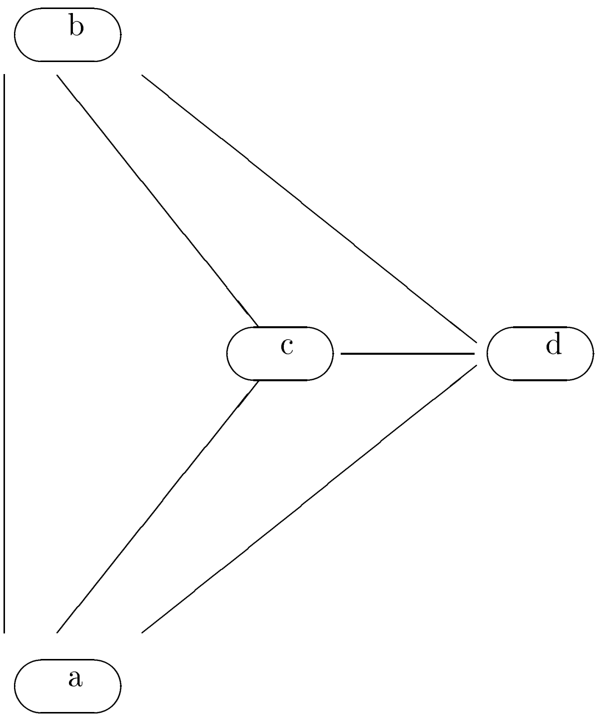


Figure 6.6: Projection of a CHG to a simple graph.

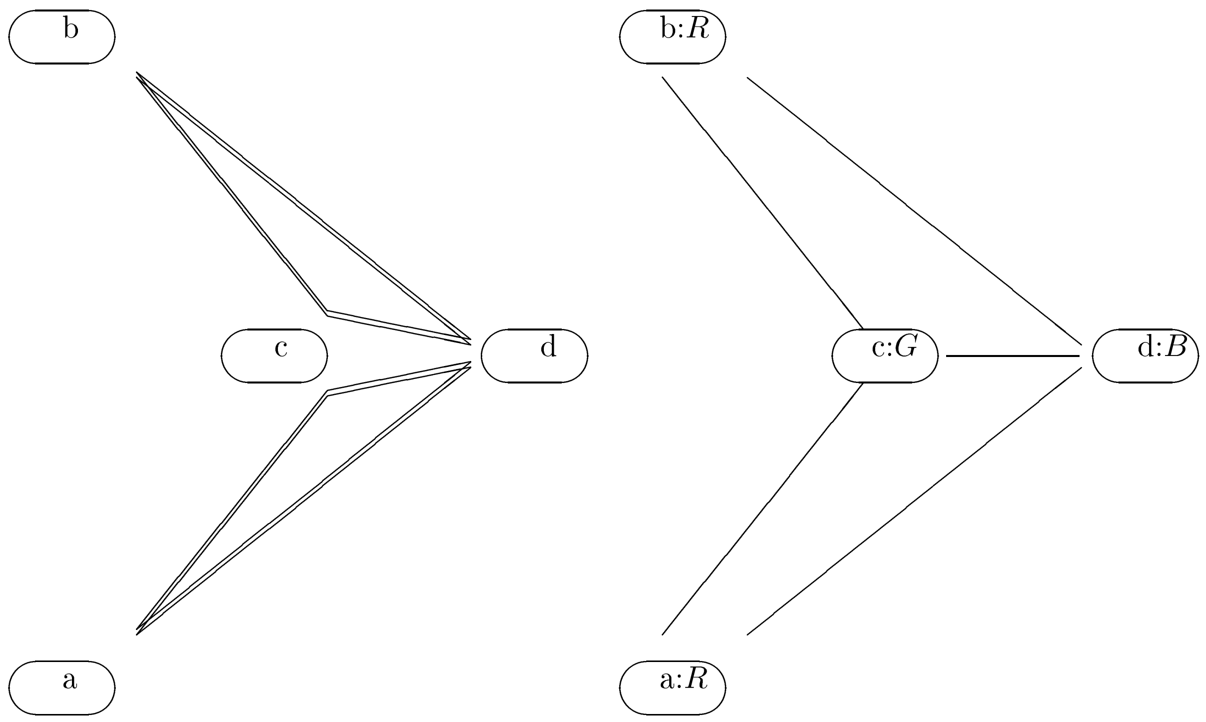


Figure 6.7: A CHG and its projection which is 3-colourable.

Example 6.6 Consider the graph of figure 6.5. Suppose that a has been assigned the 2-colour RG and b the colour R. Then a and b make a c-conflict. The orbit of the 2-edge $< a, b >$ is $\{c, d\}$, corresponding to the access of the memory by a, b and c in one time step and a, b and d in another, in the transfers of example 6.4. Assuming that R, G and B represent the first, second and the third ports of the memory, respectively, the point a will be connected to the first and second ports, while b will be connected to the first port only. In order to satisfy the accesses of a, b and c simultaneously, the colour for c should include B. Similarly the colour for d too should include B. It should be noted that B is the resolving colour for RG. □

¿ From the above examples we observe that if a vertex v is assigned a 2-colour then, the vertices of the orbit of each c-conflict of which v is a member must possess the resolving colour. Unless this condition is met some of the accesses to the memory will not be satisfied.

We can conceive of the colouring of the CHG as follows. First we form the projection of the CHG and try to colour it using three colours. If it is successful then we are done. Otherwise, some of the vertices will have to be assigned 2-colours. In this process it might be desirable to re-colour some of the vertices that were coloured earlier. The vertices that could not be assigned even a 2-colour are now assigned 3-colours.

Example 6.7 In the graph of figure 6.5 the edges are $< a, b, c >, < a, b, d >, < b, c, d >$ and $< a, c, d >$. Vertices a, b and c have been assigned colours R, G and B. If d is coloured RG, then a ad b form the orbit of the 2-edge $< c, d >$. Clearly, the accesses represented by the 3-edge $< a, c, d >$ cannot be satisfied if d is assigned RG. Vertex d may be assigned RGB. This colouring has been shown in figure 6.8. Alternatively a may be coloured RB and d as RG, other vertices being coloured the same. □

Based on this formulation we now present a GA for port assignment of triple port memories. In this formulation a colour represents connection to a port. As in section 6.3 we make the assumption that a point reading from the memory also recieves data from other places in the circuit. We also assume that each port will be connected to more than one point (reading or writing to the memory. With this assumption the incremental cost of connecting a point to a single port is 0, for two ports it is 1 and for three ports 2.

6.8 GA for the Triple Port Memory PA

Solution representation For this problem too a direct solution representation has been used. The representation consists of a total of seven sets. Of these three sets are for vertices that have been assigned a single colour or for points that are connected to a single port of the memory. There are three sets for vertices that have been assigned 2-colours, and finally there is a set for vertices that had to be 3-coloured or are deleted from the graph.

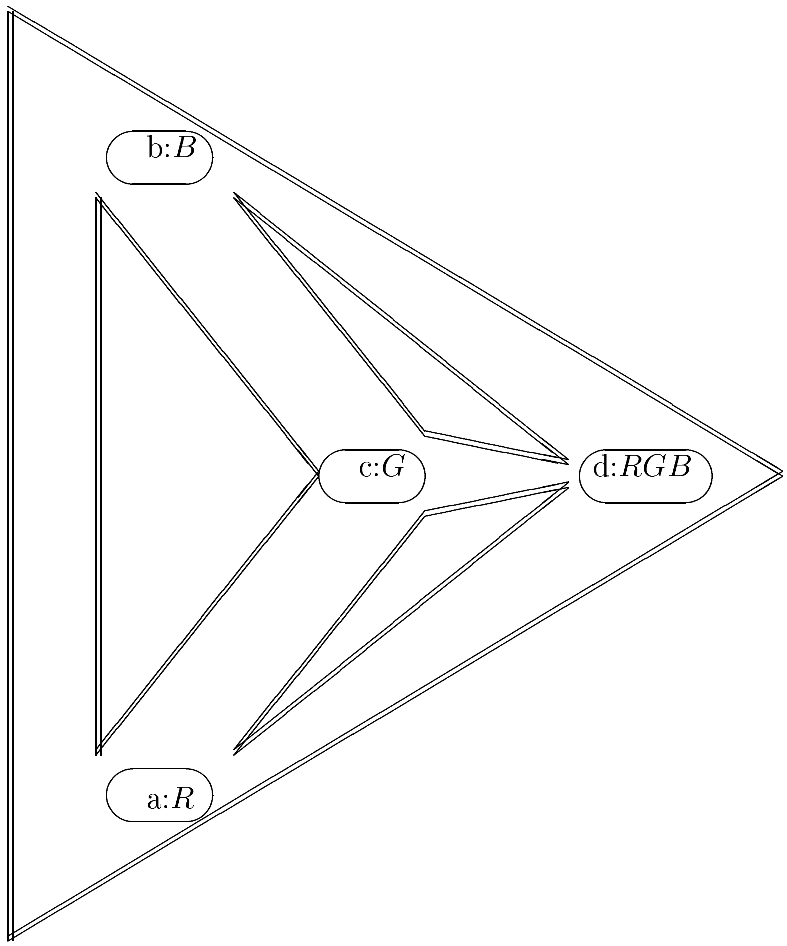


Figure 6.8: A coloured CHG

Fitness function Let V be the set of vertices of the CHG s_i be the set of colours assigned to vertex v_i. The fitness function is defined as

$$g = \sum_{i|v_i \in V} |s_i| - |V| \tag{6.5}$$

This cost function models the additional cost incurred due to the connection of a point to more than a single port. Minimization of g is the objective.

Initial population generation Each member of the initial population is a randomly generated valid solution. As before, the vertices are visited in a random sequence. Each vertex is first tested for possible membership in the three sets of single colours, one by one. In case of a failure testing is next done for membership in the three sets of 2-colours. A vertex which could not be included in any of these six sets of colours is then placed in the 3-colour set (RGB).

Reproductive plan The parent selection strategy is similar to the GA for MND. The crossover, however, has been tailored to suit this problem. We have not used a separate mutation operation, as some mutation in in the solution is inherent in the crossover that has been designed.

The first thing to note is that the CHG is a labeled graph and each colouring of the CHG may be associated with two other isomorphic colourings which result from a renaming of the single colours. The 2-colours, however, cannot be renamed independently of the single colours, in general. The required renaming of the 2-colours is implied by the renaming of the single colours. It is desirable to take this non-trivial isomorphism into account, as far as possible, while performing crossover. This is necessary to avoid a sub-standard crossover between two parent solutions which are isomorphically proximal. Thus the first step in performing crossover is to derive a one-one mapping between the single colour classes of the two solutions. This is done in the following algorithm (figure 6.9). The crossover proceeds with the formation of the singly coloured sets of vertices, followed by the sets of vertices coloured with two colours. The remaining vertices are placed in the 3-colour set. The formation of the singly coloured set of vertices is similar to the way the colour classes are formed for the GA for MND. Thus a subset of a set of vertices is first inherited. This set is then augmented with with vertices, if possible. While augmenting a set of singly coloured vertices it is only necessary to check that the new vertex does not have an edge (2-3dge) with any of the existing members of that set. The choice of sets for inheritance is made as follows. Vertices are first inherited from the largest set of singly coloured vertices of the better fit parent. At the same time the set in the second solution on which this set had been mapped is marked off. The larger of the remaining two sets in the second solution is used to form the next set in the child solution. At the same time the corresponding set in the first solution is marked off. The remaining set of singly coloured vertices in the first solution is used to construct the third set of singly coloured vertices in the child solution.

The sets of 2-coloured vertices are now formed in the child solution. The membership of vertices in a 2-colour set is governed by the vertices which constitute the c-conflicts

```
procedure map_set
sing_a, sing_b:     array 1..3 of set of integer;
/* the three sets of vertices assigned single colours, in each parent */
done_b:     array 1..3 of boolean; /* initialized false, and */
/* indicating that mapping of a single colour of the second parent is done */
map_a:     array 1..3 of integer;
/* the mapping from the first parent to the second */

for i=1 to 3 do
{    maxsim = -1;
     for j=1 to 3 do
     {    if (done_b[j]) continue;
          sim = (sing_a[i] ∩ sing_b[j]) / (sing_a[i] ∪ sing_b[j])
          if (sim > maxsim)
          {    maxsim = sim;
               match = j;
          }
     }
     done_b[match] = true;
     map_a[i] = j;
}
```

Figure 6.9: The mapping procedure.

and their associated orbits. The vertices of the orbits will certainly belong to some other set. In the child solution these vertices, in general, will not belong to similar sets of the parent solution. Therefore, an arbitrary subset of a secondary colour class may not be feasible. Thus each vertex v that is introduced to a set of doubly coloured vertices has to be checked for membership in that set. As noted in section 6.7 the membership criterion will be satisfied if each vertex of the orbit of a c-conflict of which v is a member has the resolving colour. The method of choosing the sets from which the inheritance is to be done is similar to that used for forming the sets of singly coloured vertices. Finally, the vertices which could not be included in any of these six sets are placed in the set of vertices which will be given the 3-colour (RGB). In most cases the colouring of offspring will have some differences with respect to both the parents. In view of this we do not keep a separate provision for mutation, as further forced changes might weaken the crossover significantly.

Replacement policy This is identical to the scheme followed for GA2.

6.9 Estimation of Cost of Triple Port Memory PA

An estimate has also been developed for the triple port memory PA in the lines of the estimator for the dual port memory PA or MND. However, this estimator would not be as useful as the estimator for MND. The reason for this is two fold. One is that the time required to perform the estimation is fairly large and secondly the quality of the estimate is not as good as that with the MND estimator. Nevertheless, the time requirement for graphs for around seventeen vertices, and the quality of the estimate are reasonable, on a SUN 3/280.

Let G be the CHG and let V be its set of vertices. Let $N = |V|$. Suppose that as a result of the colouring n_1 vertices are coloured R, n_2 as G and n_3 as B. Thus $n = n_1 + n_2 + n_3$ is the number of vertices assigned a single colour. Similarly let m_1 vertices be coloured RG, m_2 as GB and m_3 as RB. Thus $m = m_1 + m_2 + m_3$ is the number of vertices assigned a 2-colour. Therefore, $d = N - n - m$ of the vertices are assigned the 3-colour. Let p_3 be the probability that any three vertices of V are connected in a 3-edge, and p_2 the probability of a 2-edge, in the CHG. The probabilities p_3 and p_2 are not independent. However, it is convenient to proceed with their explicit values. Let $q_3 = 1 - p_3$ and $q_2 = 1 - p_2$. Let $a = \frac{N!}{(N-n-m)!n!m!}$. This is the number of ways in which n of N vertices may be chosen for assigning a single colour, m of $N - n$ vertices may be selected for assigning 2-colour. The n vertices may be partitioned into three sets of sizes n_1, n_2 and n_3 is $b = \dfrac{n!}{(n-n_1-n_2)!n_1!n_2!r_f}$ ways, where

$$r_f = \begin{array}{l} 6, \text{if } n_1 = n_2 = n_3 \\ 2, \text{if } n_i = n_j \neq n_k, \qquad i \neq j \neq k, i, j, k \in \{1, 2, 3\} \quad \text{and} \ . \\ 1, \text{if } n_1 \neq n_2 \neq n_3 \end{array}$$

The number of ways in which m vertices may be partitioned into three sets of size m_1, m_2

and m_3 is $c = \dfrac{m!}{(m-m_1-m_2)!m_1!m_2!}$. The probability that the sets of sizes n_1, n_2 and n_3 can be singly coloured is

$$p_n = q_2^{\dfrac{(n_1(n_1-1)+n_2(n_2-1)+n_3(n_3-1))}{2}}.$$

Similarly, the probability that the vertex sets of sizes m_1, m_2 and m_3 may be 2-colourable is $p_m = {q_3}^s$, where $s = \dfrac{m_1(m_1-1)(n_1+n_2)}{2} + \dfrac{m_2(m_2-1)(n_2+n_3)}{2} + \dfrac{m_3(m_3-1)(n_3+n_1)}{2} + m_1n_1n_2 + m_2n_2n_3 + m_3n_3n_1$.
The above products essentially enumerate the possible orbital vertices for possible c-conflicts for the members of the sets of 2-coloured vertices. The probability that the d vertices will have to be deleted is found as $p_d = t^d$, where $t = (1 - {q_3}^{n_1n_2+m_1(n_1+n_2)}) \cdot (1 - {q_3}^{n_2n_3+m_2(n_2+n_3)}) \cdot (1 - {q_3}^{n_3n_1+m_3(n_3+n_1)})$.
The expected number of partitions of the CHG into $n_1, n_2, n_3, m_1, m_2, m_3$ and d vertices denoted by

$$N_{|V|,\, p_2,\, p_3}(n_1, n_2, n_3, m_1, m_2, m_3, d) \text{ is equal to } a \cdot b \cdot c \cdot p_n \cdot p_m \cdot p_d.$$

We are, however, interested in the final cost of port assignment. We, therefore, compute the value

$$M_{|V|,\, p_2,\, p_d}(C) \quad = \sum_{\substack{m_1+m_2+m_3+2d=C \\ n_1+n_2+n_3=|V|-m-d \\ m_1,m_2,m_3,d>0 \\ n_1,n_2,n_3>0}} N_{|V|,\, p_2,\, p_3}(n_1, n_2, n_3, m_1, m_2, m_3, d).$$

We then proceed in the lines of the pragmatic estimation for MND. First determine C_1 that just gives a non-zero value for $B_1 = M_{|V|,p_2,p_d}(C_1)$, and determine C_2 that maximizes $B_2 = M_{|V|,p_2,p_d}(C_2)$. Let $B_P = \sqrt{B_1B_2}$. Determine C_E such that $B_E = M_{|V|,\, p_2,\, p_d}(C_E)$ just exceeds or equals B_P. We use C_E as the pragmatic estimate of the cost of this port assignment. C_1 would be a lower bound cost estimate.

6.10 Experimentation for Triple Port Memory PA

The GA for the triple port assignment has also been tested on random graphs, with known upper bound on the cost of the assignment. Some test results are presented in table 6.3. The generation of the random graphs is based on several parameters. The parameters are the sizes of each of the vertex sets for each set of colours, viz. R, G, B, RG, RB, GB and RGB; and the probability of a 3-edge. Separate 2-edges have not been generated, only 3-edges have been generated. This is because a 2-edge is very likely to be subsumed by a 3-edge. With each configuration of the vertex sets several edge probabilities (p_3) have been used. This is to experimentally locate those sets of problems which will pose the maximum difficulty to GA3. It may be observed from the table table that in the tightest situation the error is about 10%. As the edge density goes up the problem gets simpler and the GA also does much better than the upper bound. The results have been tabulated in table 6.3 for several combinations of these parameters. As mentioned in section 6.7 the incremental cost of a 2-colouring is 1 and that for 3-colouring is 2. Thus a problem instance of triple port memory PA having n_2 vertices bearing 2-colours and n_3 vertices bearing 3-colours will have a cost of $n_2 + 2n_3$. This is the upper bound of the cost U of the solution. The average cost of the solved instances generated using a particular set of parameters is denoted by $\bar{C}$. Each line of table 6.3 indicates the parameters used for generating the problem instances, the upper bound on the cost for each of these instances, the average cost of the solutions obtained by GA3. The last column of the table indicates the difference $U - \bar{C}$. Clearly it is desirable for this difference to be positive. The table lists the most difficult situations used to test GA3. In most case the difference is positive or very slightly negative, except for one set of parameters where it is -0.833, indicating the use of about one switch more. The table indicates that GA3 has always performed better on graphs with a higher edge probability, where the average cost $\bar{C}$ always less than the upper bound U.

The estimate for the triple port assignment has been tested on randomly generated graphs. For this estimate it is necessary to have both the 2-edge probability (p_2) and the 3-edge probability (p_3). Not having the means to generate random CHGs where these two probabilities may be controlled independently, we had to resort to another method. We generated several random graphs for a fixed p_3. For each of these graphs we counted the number of 2-edges (including the implied ones) and the 3-edges and then estimated the actual values of $p_2^{'}$ and $p_3^{'}$ for these graphs. We grouped together the graphs with identical $p_2^{'}$ and $p_3^{'}$ values and computed the average cost of triple port assignment on these graphs. However, the number of similar graphs, with respect to $p_2^{'}$ and $p_3^{'}$ have been small, and so the results are statistically less reliable. We have used these results only as an indicator for the estimator for the triple port memory PA. The results have been tabulated in table 6.4. The first three columns indicate the number of vertices in the graph, the edge probabilites p_2' and p_3', respectively. The average cost and the standard deviation, of the solutions obtained by solving instances having the parameters indicated in the first three columns is given in the next two columns, $\bar{C}$ and σ_C, respectively. The values of the paragmatic and l. b. estimates are listed in the last two columns.

6.11 General Port Assignment

6.11.1 Formulation

So far in this chapter we have discussed solution and estimation methods for dual and triple memories based on graph or hyper graph representations. These representations have the advantages of being compact, useful for developing heuristics which may be incorporated in the solution techniques and easier to analyze. However, a few practical aspects of the problem are not easily incorporated in the graph based formulation. We

$\|R\|$	$\|G\|$	$\|B\|$	$\|RG\|$	$\|RB\|$	$\|GB\|$	$\|RGB\|$	p_3	U	$\bar{C}$	$U-\bar{C}$
10	10	10	2	2	2	1	0.019	8	8.766	-0.766
10	10	10	2	2	2	1	0.022	8	9.133	-1.133
10	10	10	2	2	2	1	0.025	8	8.400	-0.400
10	10	10	2	2	2	1	0.029	8	8.966	-0.966
10	10	10	2	2	2	1	0.032	8	8.733	-0.733
10	10	10	2	2	2	1	0.035	8	8.366	-0.366
10	10	10	2	2	2	1	0.039	8	8.133	-0.133
10	10	10	2	2	2	1	0.043	8	7.966	0.033
10	10	10	2	2	2	1	0.047	8	8.233	-0.233
9	9	9	2	2	2	2	0.019	10	8.966	1.033
9	9	9	2	2	2	2	0.021	10	9.400	0.600
9	9	9	2	2	2	2	0.023	10	10.033	-0.033
9	9	9	2	2	2	2	0.026	10	10.133	-0.133
9	9	9	2	2	2	2	0.036	10	10.366	-0.366
9	9	9	2	2	2	2	0.049	10	9.933	0.066
9	9	9	2	2	2	2	0.064	10	10.033	-0.033
9	9	9	2	2	2	2	0.083	10	10.000	0.000
9	9	9	2	2	2	2	0.110	10	10.000	0.000
6	5	7	1	2	2	1	0.01	7	0.266	6.733
6	5	7	1	2	2	1	0.02	7	1.700	5.300
6	5	7	1	2	2	1	0.03	7	3.966	3.033
6	5	7	1	2	2	1	0.04	7	5.400	1.600
6	5	7	1	2	2	1	0.05	7	6.166	0.833
6	5	7	1	2	2	1	0.06	7	6.700	0.300
6	5	7	1	2	2	1	0.07	7	6.733	0.266
6	5	7	1	2	2	1	0.08	7	6.966	0.033
6	5	7	1	2	2	1	0.11	7	7.000	0.000
6	6	6	2	2	2	1	0.010	8	0.366	7.633
6	6	6	2	2	2	1	0.020	8	2.933	5.066
6	6	6	2	2	2	1	0.035	8	5.833	2.166
6	6	6	2	2	2	1	0.045	8	7.400	0.600
6	6	6	2	2	2	1	0.061	8	8.066	-0.066
6	6	6	2	2	2	1	0.080	8	8.300	-0.300
6	6	6	2	2	2	1	0.106	8	8.033	-0.033
6	6	6	2	2	2	1	0.140	8	8.000	0.000
6	6	6	2	2	2	1	0.193	8	8.000	0.000

Table 6.3: Performance of GA3 on random graphs where an u.b. on the number of nodes to be deleted is known

$\|V\|$	p'_2	p'_3	$\bar{C}$	σ_C	prag. est.	l.b. est.
15	0.41905	0.04396	3.7	0.94	5	3
15	0.47619	0.05275	4.8	0.40	5	3
15	0.48571	0.05714	5.6	0.49	5	3
15	0.54286	0.06154	5.2	0.83	6	4
15	0.54286	0.07033	6.2	0.43	6	4
15	0.60000	0.08132	6.5	0.50	7	5
15	0.62852	0.09890	8.0	0.71	7	5
15	0.65714	0.09890	8.4	0.49	8	6
15	0.68571	0.10549	9.0	0.00	8	6
15	0.68571	0.11648	8.7	0.45	8	6
15	0.72381	0.11868	10.0	0.00	9	7
15	0.73333	0.15824	10.5	0.50	9	7
15	0.75238	0.16264	11.0	0.00	9	8
15	0.78095	0.17143	11.2	0.69	9	8
15	0.81905	0.17802	12.0	0.00	10	9

Table 6.4: Comparison of the estimator against the number of nodes deleted by GA3.

did not find any elegant method to represent multi-cycle accesses. The representation of varying switching requirement for different points that access the memory is not so difficult. For the graph based formulations we had assumed that points that read from the memory also recieve data from other parts of the circuit. While this assumption will be satisfied most of the time, occasional exceptions may be expected. For completeness we therefore present a method to deal with the practical situations which include:

a) No restriction on the interconnection of points accessing the memory.

b) Accesses may be multi-cycle, and

c) The ports may be non-uniform – *read/ write/ read-write.*

We first examine the interconnection in the general scheme. Let us consider points in the circuit that read from the memory. Such a point might receive exclusively from the memory or from other parts of the circuit as well. In the former situation if the point is connected to a single port no switch is required and two switches are required for connection to both the ports. In the latter case if the point is connected to k $(1 \leq k \leq 2)$ ports then k switches will be required. Thus if there are two points p_1 and p_2 of the first and second type, respectively, and if there is a choice regarding which one is to be connected to both the ports, then p_2 should be chosen. This results in a saving of one switch. For points writing to the memory, connection of all but one of these points to a single port of the memory may result in saving one switch. This saving will be obtained only if the port to which this singleton point is connected is a write-only port or no other point reads from this port. The graph formulations presented above do not make this distinction. However, these can be accommodated in the representation by associating 0/1 weights with the vertices.

The conditions which should be satisfied by the assignments in the framework considered in this section are as follows:

- All accesses in the same time step should go through different ports.
- A multi-cycle access should be confined to a single port in all the time steps.

When there are a total of n ports, of which n_1 have read capability and n_2 have write capability, the necessary condition is that in any time step, there may be at most n accesses to the memory, of which no more than n_1 are read and n_2 are write. This condition, however, does not guarantee the existance of an assignment, as shown in example 6.8.

Example 6.8 Consider the following sequence of accesses given for a memory with a read port and a read-write port, assuming that variables a and b are present in the memory. The number of cycles for which the access should be sustained is indicated in parantheses adjoining the transfer.

1. *write a* (1)   *read b* (2).
2. *read a* (2).
3. *write b* (1).

The *write a* access in the first step and *read a* in the second step must be assigned to the read-write port, as a result the *write b* in the third step cannot be assigned to a port with write capability. It may be noted that the two conditions cited above are satisfied by this example. □

Based on these conditions we now propose a GA for a k port memory. The input to the GA is the explicit sequence of accesses to the memory along with the partial data path structure.

6.11.2 A Simple GA for General PA

Solution Representation A representation similar to the bit string is used. Instead of bits, interger fields are used. The total number of fields equals the number of accesses. Each field denotes the port to which this access has been mapped.

Fitness function This function returns the total cost of the switches required at the the ports of the memory and at the points that read from the memory.

$$g = \sum_{i, 0 \leq i \leq k} \# \text{ switches at port } i + \sum_{p \mid \text{p reads from memory}} \# \text{ switches at } p \tag{6.6}$$

The objective is to minimize the switch cost.

Initial population generation This is generated by randomly assigning port numbers to the transfers while ensuring that the two aforementioned conditioned are satisfied.

Reproductive plan The parents are selected with a bias towards the more fit members (having lower switch cost). A fixed number of offsprings are generated by crossover. These solutions replace the least fit members of the current population. This policy ensures that all the solutions generated get a chance to compete atleast once in the solution generation process. It, however, does not guarantee that the solution will participate in a crossover. The offspring is built through a crossover of two parent solutions, thus selected. The port assignments of the offspring are inherited from the parents, time step by time step, starting from the first time step. If a multi-cycle access is present in the current time step, then it is necessarily assigned to the port where it was assigned in the previous time step. An assignment diagram for the bindings in the current time step is constructed for both the parents to facilitate the crossover. First of the assignments to some of the ports are inherited from the first parent. Then inheritance is attempted from the second parent. Now some conflicting assignments will be encountered, and they have to be resolved. At the end of the inheritance a few accesses may still remaim unmapped to a port in the offspring. These a assigned at random to the free ports. The crossover for a time step is illustrated through example 6.9.

Replacement policy This is identical to the scheme followed for GA2.

Example 6.9 The example illustrates the crossover for triple port memory PA for accesses $< a_{i,t},\ a_{i+i,t},\ a_{i+2,t} >$ in a hypothetical time step t. The port assignments for the two parents in the current time step are $< 0, 2, 1 >$ and $< 1, 0, 2 >$. The corresponding assignment diagrams are shown in figure 6.10. Let us choose to inherit the remaining transfers from the second parent. Suppose that the binding for port 0 is inherited from the first parent, i.e. transfer i is mapped to port 0. While attempting to inherit the assignment to port 1, we notice that the transfer has already been assigned. We then use the method shown in procedure *map_trans_port* (figure 6.11) to find a transfer to map to this port, if possible. The assignments at the end of the current time step are $< i, i + 1, i + 2 >$. □

6.11.3 Experimental Results

For GA2 and GA3 presented earlier we were able to design a reasonably sound experimental testbed to measure the performance of these algorithms. For random graphs we had reasonably accurate measures of the upper bounds. This was facilitated by our ability to characterize these graph on the basis of the number of vertices and the edge probabilites. While the present GA is able to handle complex situations like multicycling and non-uniform ports elegant experimentation using the properties of random graphs is now no longer possible. Further more there is a dearth of benchmarks for such problems. In this situation we performed experimentation in the following manner.

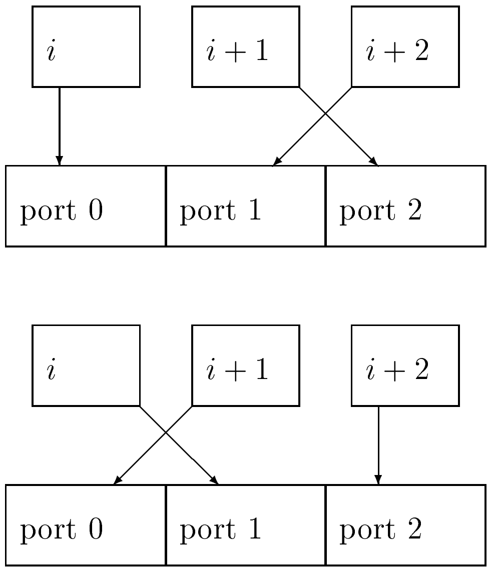


Figure 6.10: Assignment diagrams for the two parents.

```
procedure map_trans_port (p)
p:    integer; /* indicating the port for which a transfer is sought */
pa1:    array 1..m of set of integer; /* transfer to port mapping of pnt. 1 */
ipa2:    array 1..k of set of integer; /* port to transfer mapping of pnt. 2 */
/* a NULL entry in ipa2[p] indicates that no transfer is mapped to port p */
t = ipa2[p]; /* transfer mapped to port p in pnt. 2 */
if (t is multi-cycle or t = NULL)
{     return(NULL);
}     else
if (t is unmapped)
{     return(t);
}     else
do {
      p = pa1[t];
      t = ipa2[p];
}     while (t is mapped and t is not multi-cycle)
if (t is multi-cycle or t cannot be satisfied by p)
{     return(NULL);
}     else
return(t);
```

Figure 6.11: Procedure to find a tranfer to map.

We first tested the GA on individual examples involving multi-cycling and non-uniform ports. We then tested this GA for the inputs of GA2 and GA3. In the latter case we expected the general GA to perform a little worse than GA2 and GA3. We we interested to find the exact amount of degradation. The summary of results is as follows:

- For graphs with a small number of vertices (~ 5) optimal results are obtained.
- For larger graphs (upto 40) vertices the results were 5-10% worse than the corresponding results for GA2 and GA3.
- Medium size hand coded examples involving multi-cycling and non-uniform ports were successfully run.

This shows the advantage of the graph theoretic formulation and also intelligent crossovers. However, when multi-cycle transfers are present the graph theoretic formulation cannot be used to obtain a feasible assignment. For single cycle accesses, the deficiency in the graph theoretic formulation is made up for by the better quality of the solutions obtained.

6.12 Conclusion

An efficient genetic algorithm has been developed to solve the dual port memory PA or the minimum node deletion problem. A method has also been developed, using the model of random graphs, to estimate the number of nodes that must be deleted for the minimum node deletion problem. In a similar manner a GA has also been developed to solve the triple port memory PA. We have also developed an estimator for estimating the cost of PA for a triple port memory. However, this estimator is not as promising as the one developed for the dual port memory PA. An important reason is that the PA for triple port memories is computationally for difficult than the former problem.

The estimator for the dual port memory can serve as a valuable tool to aid design space exploration when it is necessary to evaluate numerous packings of registers into a dual port memory. The use of the estimator would be especially enhanced when GA2, the GA that has been developed for MND, is used to solve the dual port memory PA. This is because the estimator closely estimates the number of nodes actually deleted by GA2.

Chapter 7

Design Space Exploration and Scheduling

7.1 Introduction

In the previous two chapters we have proposed solutions to some some individual problems related to interconnect optimization like RIO, MIO and dual and triple port PA. In this and the next chapter we shall propose a two phase solution to the entire DPS problem. The input to the entire DPS problem is a set of optimized data flow graphs and some design parameters. We have chosen the design parameters to reflect some important architectural aspects, such as the number of buses, f.u. sites and system interface ports. At the end of DPS we are required to find one or more "optimized" implementations for a design problem input to the synthesis system. The objectives of optimization in this situation is multi-fold in the sense that we seek to optimize not only the area cost estimate of the data path but also its performance, measured as a function of the length of the schedule of each basic block (b.b.). This makes the synthesis problem a multi-criteria optimization problem. The criteria now being considered are area (estimated through the cost of individual components) and performance of the final design. Others like power dissipation and testability may also be considered.

A feature of most multi-criteria optimization problems is that the criteria are often non-commensurate and sometimes conflicting. It is therefore difficult to combine the criteria into a single cost function. We take the approach of representing the cost of a design as a tuple of costs of the individual objectives. This is similar to the approach taken in Stewart et al. [37] One cost tuple is said to be better than another distinct tuple if the cost of each criterion of the first tuple is no worse than the corresponding costs of the other tuple. A design whose cost tuple is better than that of another design is said to dominate that design. The global problem of optimization is to find the set of designs which are not dominated by any other designs. The set of feasible designs satisfying the design parameters constitute the design space. Each design point in the design space corresponds to an estimate of hardware requirement and performance computed as a function of the schedule time. Thus an algorithm for DPS needs to consider techniques not only for scheduling, allocation and binding but also for a systematic exploration of the design space to locate these non-dominated designs. The starting point of design space exploration often revolves round the basic scheduling problem.

In our two phase scheme for DPS we first consider the problem of *scheduling and design space exploration* and in the second phase we take up the problem of *allocation and binding*, where we construct the actual data paths from a set of scheduled flow graphs. We make use of a bus based interconnection structure for the data paths. The latter problem includes operation to functional unit binding, transfer to bus binding and mapping of variables into storage. More accurate cost estimates of the data path are available after allocation and binding. We do design space exploration (DSE) using a combination of controlled search, heuristic and genetic scheduling techniques. The enormous size of the state space of the state space of the allocation and binding problem makes enumerative search prohibitive in terms of time. Instead we use a genetic algorithm (GA) approach for this problem since GA inherently looks at alternative solutions simultaneously. In this chapter we discuss our approach to DSE and scheduling and present the allocation and binding scheme in the next chapter.

Conventional scheduling algorithms require a time constraint or a specification of the available f.u.'s. In a practical DPS situation neither the appropriate time constraint nor the appropriate f.u. requirement will be known in advance. Through DSE we systematically explore several combinations of time constraints and hardware resource configurations that are feasible. We use the concept of multi-criteria optimization and arrive at several configurations with different performance and f.u. requirement estimates. We employ a multi-objective search approach to perform design space exploration and scheduling. In our scheme we have a state space generation mechanism coupled with an estimator for obtaining various <hardware cost, performance> estimates. A controlled depth first branch and bound is used to determine the hardware cost estimate and produce a partial schedule for a given time constraint. This actually corresponds to a localized exact or near exact exploration of a region of the entire design space. In order to contain the combinatorial explosion, the computational effort to be spent on DSE can be controlled by certain parameters. While designing these tools we also permit the designer to impose design parameters and then examine the design space for possible designs which satisfy these parameters. We have chosen these parameters to reflect some important architectural aspects, such as the number of buses, the number of f.u. sites, the number of system ports, etc. over which the designer may wish to have some control.

We have already noted that design space exploration, to start with is centered around the basic scheduling problem. At the heart of the DSE mechanism is the controlled search based resource estimation and the partial scheduling (REPS) algorithm. The basic DSE technique makes use of the REPS algorithm to estimate the hardware requirement, as tightly as possible, so that the design parameters are also satisfied. REPS also returns a partial or complete schedule depending on the situation. This way the design points are computed. Scheduling, however, is an NP-hard problem and for large problem instances it may be necessary to settle for a restricted search. In this case the design points obtained are approximate (lower bounds) and the schedules may be partial in the sense that the degree of freedom of some operation may still be more than one. To meet this situation a local DSE mechanism has been developed to explore the neighborhood of such a design point to obtain one or more non-dominated design points for which feasible schedules will exist. The local DSE mechanism also produces a feasible schedule for each design point that it returns. Such schedules are obtained using existing scheduling techniques which perform scheduling using the precedence constraints and sometimes the available hardware resources. We have also developed a genetic list scheduling technique to make use of the partial schedules generated by REPS.

In the following we present more details of our solution to the problem of design space exploration (DSE) to generate a set of schedules which will represent non-dominated designs. The inputs for design space exploration are explained in the next section. The estimates used by REPS for hardware cost and schedule time are discussed in section 7.3. REPS itself is presented in section 7.4 Then the overall DSE mechanism (which uses REPS) is explained in section 7.5. After this the GA based list scheduling algorithm is presented in section 7.6. The experimental results are presented in section 7.7. We conclude in section 7.8.

7.2 Inputs to DSE

7.2.1 Operation Precedences

This is the most important input to the DSE algorithm. For practical design examples there will be a number of basic blocks, and for each b.b. there will be precedence constraints on the operations in that b.b. The precedence constraints between operations are restricted to be partial orders. This is not a severe restriction because a sequence of operations in a basic block gives rise to such constraints [7]. Each type of operation is also assigned an execution time, to indicate the number of time steps over which the operation will execute. The execution time of an operation is determined by the speed of the hardware implementation of that type of operation.

7.2.2 Design Parameters

In order to explore the designs which are possible for a given behavioural specification in reasonable time and in a structured manner, it is desirable to guide the design process with some user specified parameters. These parameters should be simple and easily visualizable by the design engineer. Our design space exploration (DSE) scheme uses the following two parameters.

NFUS This indicates the number of sites where hardware operators will be clustered. However, f.u.'s need not be formed during scheduling. No two hardware operators at the same site may receive inputs or deliver outputs in the same time step. The hardware operators in the f.u.'s are in this sense mutually exclusive. Clearly, *NFUS* simple operations may be performed in the same time step. The clustering has been done to facilitate the optimizations at the time of physical design. *NFUS* has an effect on the controller cost, a small value of *NFUS* compared to the total

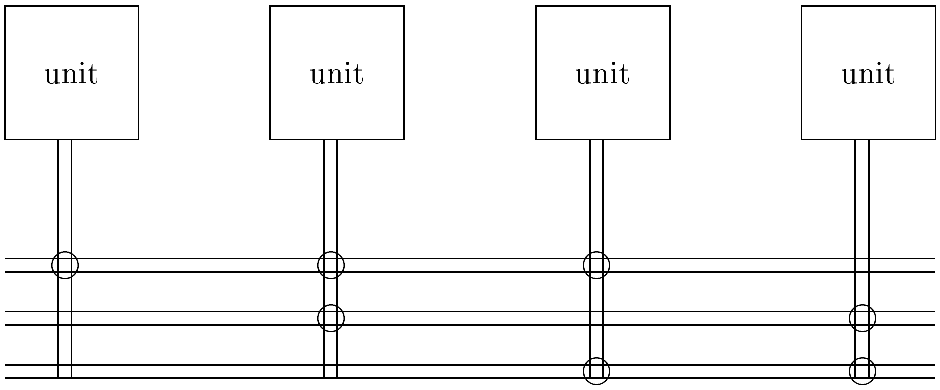


Figure 7.1: The interconnection framework.

number of operation units will permit significant optimizations in the design of the controller, while a comparable value will permit little redundancy and therefore less optimization. *NFUS* constrains the scheduling algorithm which has to ensure that no more than NFUS simultaneous operations take place in a time step.

NBUS This is the maximum number of logically distinct buses in the system. Presently, each communication path between units is abstracted as a bus. In our model of implementation, as shown in figure 7.1, two or more units are connected to each main bus. The connection may be switched or direct.

NVREF This is the maximum number of distinct variable references permitted in any time step. Reading and writing to a variable are considered distinct accesses. Variables that are simultaneously accessed cannot be packed into the same single port memory. This parameter is used to have a check on the number of simultaneous variable accesses.

Though the above parameters are independent they are well co-related. It may be expected that $NBUS \approx 3NFUS$ and $NVREF \approx 3NFUS$. The exact equality may not hold for several reasons. In some cases the value generated by an operation could have to be stored in more than one storage location. Such a situation is depicted in figure 7.2 where the + operation is annotated with two labels, indicating that the output of this operation would have to be transferred to the locations where these variables have been mapped. In general, such variables will not be mapped to the same location, and the transfers will be distinct. Specifications where additional transfers are frequent, a slightly higher value of NBUS and NVREF than $3NFUS$ could be desirable.

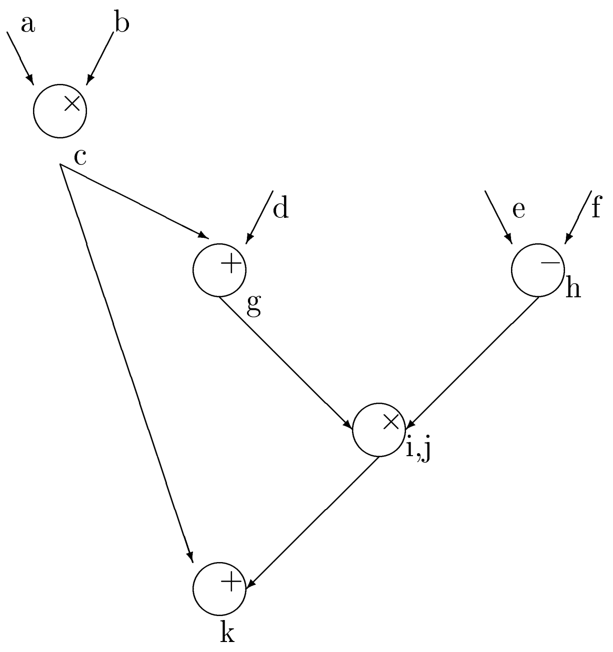


Figure 7.2: A sample directed acyclic graph.

7.3 Measures for DSE

We shall often have to consider a region of the design space and efficiently determine the design point or points in this region which will be feasible and worth retaining. In general we shall have to resort to scheduling to answer this question. However, scheduling could be computationally intensive. We, shall therefore rely on heuristic measures not only to aid scheduling but also to arrive at our decision as early as possible. Therefore, estimates which try to compute area by actual floor planning schemes are ruled out at this stage. We shall first indicate the type of measures that we would like to compute and then suggest some basic methods to compute them.

7.3.1 Estimates of Hardware Requirement

We would like to estimate the hardware requirement to check for the feasibility of a design region and to locate feasible and possibly non-dominated design points. We would, therefore, like to estimate the following: i) hardware operators, ii) storage elements, iii) buses and iv) switching elements. The maximum number of operations to be executed in any time step determines the minimum number of f.u.'s that will be required. This should not exceed the number specified as the design parameter. The bus requirement also needs to be determined to ensure that the other design parameter is not violated. For the purpose of DSE it is desirable to estimate these as accurately as possible. However, at this early stage of design it is difficult to have reliable estimates of all the three types of RTL components mentioned above. Among these, it is easiest to estimate the requirement of hardware operators. It is also possible to estimate the storage requirement before scheduling has been done [43], but this estimate is, relatively, less reliable. It is most difficult to estimate the switch requirement before scheduling has been performed. After scheduling, the hardware operator requirement and the storage requirement can be estimated more accurately. For straight-line code the minimum storage cost can be easily obtained using the *left edge algorithm* [21]. At this stage a better estimate of the switch requirement can also be obtained for a point to point interconnection scheme. For a bus based interconnection scheme a reasonable estimate of switch requirement can be obtained after transfers have been mapped to buses [25].

Thus, while working with incomplete schedules it is may be computationally inefficient to include the switch cost. When scheduling is complete the design can be evaluated more accurately by using better estimates of storage, switch and hardware operator cost. In the next subsection we indicate the computation of lower bound estimates for specific hardware operators, total number of operations in any time step and the bus requirement.

7.3.2 Estimators for DSE

Estimation of Resources for Specific Operations

Given a DAG, we would like to estimate the number of each hardware operator for realizing each kind of operation to schedule the DAG in (say) n time steps. This estimate can be obtained as a lower bound. The method of determining the lower bound is similar to the techniques proposed in [46, 23]. We first introduce the notion of a *window* which we shall use to compute the estimates. A contiguous sequences of time steps is referred to as a window. Given a DAG to be scheduled in n time steps, there can be n windows of size one, $n-2$ windows of size two, ..., and one window of size n. Thus there can be a total of $\frac{n(n+1)}{2}$ windows with n time steps. For determining the estimate it is necessary to determine the earliest and latest times at which each operation in the DAG (of a b.b.) may be scheduled. These are most conveniently determined from the ASAP and ALAP schedules, where the operations are scheduled as early or as late as possible, respectively.

The construction of the lower bound is now explained. First consider any window in the given DAG of $j, j \leq n$, steps and starting at time step $i, 1 \leq i \leq n$. Consider any operation o in the DAG, let the earliest time step where it can be scheduled be $t_{a,o}$ and the latest time step where it can be scheduled be $t_{l,o}$. If $t_{a,o} \geq i$ and $t_{l,o} < i+j$, then in each and every possible schedule of the DAG o must lie in the aforesaid window. Let the operation o be of type x. Let there be a total of m operations of type x restricted, in the same manner, to lie in this window, then at least $l^r_{x,i,j} = \lceil \frac{m}{j} \rceil$ hardware operators to realize the operations of type x in the DAG. Let $l^r_x = \max_{i,j} l_{x,i,j}, 1 \leq i \leq n, 1 \leq n-i-j+1 \leq n$. Then l^r_x too is a lower bound on the number of hardware operators for operations of type x. Similarly let L^r_x be the maximum of l^r_x over all the DAG's. This too is a lower bound. This is the principle that has been used to derive the l.b.s on the number of operation units of type x for the design. If C_x is the cost per unit for an operator of type x then the estimate of the resource cost is defined as $\sum_x L_x^r C_x$.

Estimation on the Total Number of Operations per Time Step

This metric is required to ensure that the parameter *NFUS* is not violated. This metric is found in a manner that is very similar to the method explained above, for the previous metric. Only, in this case, no distinction is made between the different types of operations, and all the operations occurring in a window are counted. Therefore, this estimate is also obtained as a lower bound.

Estimation for Buses

The bus requirement is estimated by examining the transfers that take place in various windows. Each operand of an operation contributes to a transfer. Transfers also arise due to variable assignments. As usual we consider the transfers that will be restricted within the window under consideration and then compute the lower bound on the number of concurrent transfers. Common variables which form inputs to operations need to be handled carefully. For the purpose of computing a lower bound transfers arising from the same variable to operations which are neither ancestors or descendents of one another may be counted only once, otherwise they may be considered distinct.

Estimation for Variable Accesses

The number of distinct variable accesses is determined by examining the variable accesses that take place in various windows. Each input and output operand of an operation contributes to a variable access. As usual we consider the accesses that will be restricted within the window under consideration to compute the lower bound. Input operands named by the same variable need careful handling. Like the l.b. determination for buses variable accesses by operations which are neither ancestors or descendents are counted only once, otherwise they may be considered distinct.

While working with a behavioural specification (BS) and the associated parameters, we would like to have an idea of the cost of the various designs that are possible without having to go through those designs in full detail. At this early design stage the hardware cost can be estimated with only a limited accuracy. The schedule time is dependent on the duration of the clock cycle and the number of time steps in the schedule. We will, in general, only be concerned with the number of time steps. However, when the intermediate representation of the BS consists of multiple b.b.'s the effective schedule time of the design needs to be suitably defined. We now examine the estimation of schedule time.

The above estimation methods are applicable to individual DAG's. For multiple DAG's these estimators need to be applied to each of those DAG's. The global l.b. is obtained by merging the individual l.b.'s.

7.3.3 Schedule Time Estimation

When we are dealing with a design of multiple b.b.'s it becomes desirable to combine the schedule times of the several b.b.'s into a single time cost. This may be done in many ways. One way is to consider the weighted sum of the schedule times of the b.b.'s. The weights reflect the relative execution times of the b.b.'s, a higher weight indicating more execution time. These weights may be computed as follows:

1. Experimentally determine the (relative) probabilities of the branches of each conditional branch in the flow graph. Determination of such probabilities is sometimes done using speculative computation based on multiple branch prediction [73]. This is taken to be the weight of the arc coming out of a condition.

2. The weight of an arc leading to a loop body of a fixed number of iterations n, is taken to be n.

3. For a loop body whose iterations is not fixed, it is necessary to determine the expected number of iterations. This is taken to be the weight of the arc leading into this loop body.

4. The weight of all other arcs is taken to be 1.

5. For *while* loops the b.b. where the condition testing is done (the loop head) is not considered to be within the body of the loop, whereas for *repeat-until* loops the condition testing is done within the body of the loop.

6. Remove the *back edges* from the flow graph. The modified flow graph is now acyclic.

7. The weight of each b.b. is now taken to be the product of the weights of all the arcs leading to that b.b. from the starting b.b. While computing the weight of the loop head of a *while* loop the weight of the edges incident on that b.b. is taken to be augmented by n, where n is the expected number of iterations of the while loop.

If the b.b. schedule times are combined this way then the average performance of the system is optimized. Example 7.1 shows the weight computation for a hypothetical flow graph. However, the time needed by the system to react to a new input is not easily predictable.

Example 7.1 Consider the flow graph of figure 7.3. The edges have been annotated with the appropriate weights as explained above. The basic blocks are mentioned in bold. The parenthesized numbers indicate the schedule time of the b.b.'s. The weight of each basic block may now be computed as follows.

b.b. 1: 1

b.b. 2: $2 * (1 + n) = 2 + 2n$

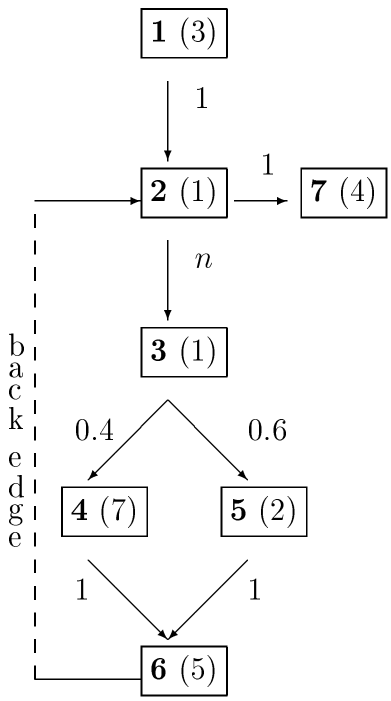


Figure 7.3: A flow graph of b.b.'s illustrating branching and looping.

b.b. 3: $1 * n * 1 = n$

b.b. 4: $7 * 0.4 * n * 1 = 2.8n$

b.b. 5: $2 * 0.6 * n * 1 = 1.2n$

b.b. 6: $5 * (1 * 0.4 + 1 * 0.6) * n * 1 = 5n$

b.b. 7: $4 * 1 * 1 = 4$

The weighted time for the process indicated in the figure is the sum of the above weights, which $7 + 12n$. □

When the target system has to be synchronously interfaced with other systems it becomes desirable to have definite input and output timing. To meet this requirement, first of all, each loop in the BS should perform a fixed number of iterations or the expected value of the number of iteration must be known. Secondly, the b.b. times have to be combined using the following rules:

1. The time of a basic block is the number of time steps in the b.b.

2. The time of a sequence of b.b.'s is the sum of the times of those b.b.'s.

3. The time of a conditional is the sum of the maximum time of its branches, and the time required to evaluate the condition.

4. The time of a loop is the product of the time of the loop body and the number of iterations of the loop.

Example 7.2 shows the time computation for a hypothetical flow graph.

Example 7.2 Once again refer to the figure 7.3. The maximum execution time of the process is

$$3 + 2 * (n + 1) + 3 * n + \max(7, 2) * n + 5 * n + 4 = 9 + 17n.$$

□

7.4 Search Algorithm for Resource Estimation and Partial Scheduling (REPS)

The resource estimation and partial scheduling algorithm uses the estimators described in the previous section to determine the resource cost for a given schedule time. It also returns a complete schedule if required. This requirement is controlled by a threshold. If the threshold W is set to one then it returns a complete schedule. If $W > 1$ then it returns a partial schedule in the sense that the degrees of freedoms (DOF) of all operations are suitably reduced but some may still have non-zero DOF. It is, however, ensured that the DOF of all operations will be less than W. We have found experimentally that these estimators work better for smaller DAG's. Thus the REPS algorithm partitions the DAG, if necessary, into smaller DAG's, applies the estimator to these partitions and combines the estimates for the different partitions to arrive at the final estimate. The schedules of partitions are combined to return the partial schedule obtained. The REPS algorithm does a systematic search of the problem space using DAG partitioning as the state space decomposition procedure. The details are now explained.

7.4.1 DAG Partitioning

A threshold W on the maximum size DAG for which the estimate will be accepted without further partitioning is specified by the designer. The partitioning scheme involves splitting the n time steps, in which to schedule (a partition of) the DAG, if $n > W$, into $\lceil \frac{n}{w} \rceil$ bands, each of at most W time steps. Each operation of the DAG is restricted to lie in only one of these bands. For operations whose, ASAP and ALAP times, t_a and t_o, lie within a band, nothing needs to be done. For other operations it is necessary to take a decision regarding the band where it should be restricted to be scheduled. A poor decision regarding the band where the operation should be placed could give rise to a high and sub-optimal resource cost estimate. A search must, therefore, be conducted on the DAG to take the right set of decisions. We have employed a depth first branch and bound scheme. The process of decomposition continues recursively till the size of none of the partitions of the current DAG are more than W. It may be noted that if $W = 1$ then the search would finally produce a complete schedule of the graph, to minimize the resource cost. Such an algorithm would, in general, fail to find a schedule in reasonable time for relatively large problems. By having $W > 1$ we are able to reduce the amount of search that will be incurred. The resource requirement of a particular type of resource in the design is the maximum requirement of that resource over all the partitions of a particular DAG. We now explain the search mechanism.

7.4.2 The Search Scheme

The memory requirement for storing the partial solutions is high. Thus we have chosen the depth first search branch and bound (DFBB), whose memory requirement is minimal. In the search scheme the partitioned DAG's are treated like separate DAG's. If the number of time steps within which the DAG needs to be scheduled does not exceed W then no more repartitioning is done and the current estimates are accepted. Otherwise, it is split into two smaller DAG's. The splitting is done near about the middle so that the two sub-problems generated are of similar size. If there are one or more operations crossing the boundary then all the possibilities of distributing these operations need to be tested out. This is where the search comes in. We perform the search by explicit backtracking. In order to keep track of the moves a stack (stack1) is used. For an operation that crosses the partition boundary there are three moves to be made, which are i) it has to be scheduled in the top half, ii) it has to be scheduled in the lower half and iii) its original freedom has to be restored. The first two moves are forward moves, while the third move is there to perform backtracking. The first move is performed right away while the other two moves into the stack (stack1), in that order. After making a move the ASAP and ALAP schedules are recomputed. When the move made is of the forward type, the resource estimate is computed. This is a lower bound estimate and may increase as the depth of the search increases. If this estimate exceeds the estimate of the best design found so far, then the current move is rejected and backtracking is initiated. Move rejection followed by backtracking also takes place if the resource estimate after the move is found to be infeasible with respect to the design parameters. Initially there is no solution and so at the beginning a dummy solution of very high cost is assumed. This solution is replaced by the first (partial) feasible solution that is found. Backtracking has been illustrated through example 7.3.

Example 7.3 Consider REPS for a hypothetical b.b. containing only plus and minus operations, to be scheduled in eight time steps. Assume that the value of W is five and the current l.b. on the requirement of adders and subtracters for the best solution identified so far is $< 2, 3 >$. Partitioning is required and the partition boundary may be taken to be time step 5. Suppose that the time frame of an operation O of the b.b. is found to be [3..7]. While performing REPS this operation is first restricted to be

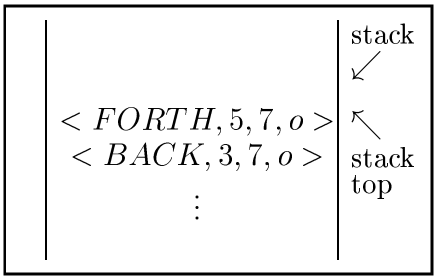


Figure 7.4: Stack used to store moves.

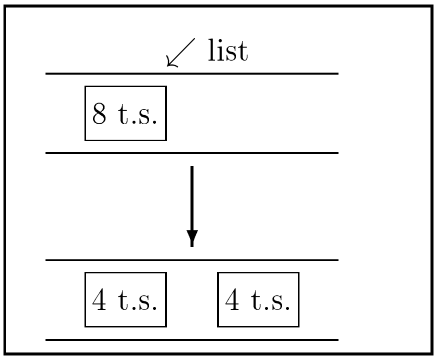


Figure 7.5: List used to store partitions of a b.b.

scheduled in the interval [3..4]. For backtracking the entries $< BACK, 3, 7, o >$ and $< FORTH, 5, 7, o >$ are pushed into the stack, as shown in figure 7.4.

Restricting the operation O in [3..4] may not lead to a partition of the b.b. However, if the l.b. computed now turns out to be $< 3, 3 >$, it becomes necessary to backtrack. The stack is popped and the operation O is now restricted between $< 5, 7 >$, as obtained from the stack, and the search goes on. □

When partitioning is done, it becomes necessary to handle multiple basic blocks. A list is used to handle these b.b.'s. To start with the initial basic block is entered in the list. The list is then passed to REPS to estimate the resource requirement for a single b.b. The b.b. at the head of the list is examined. In case the b.b. is a small one, then it is temporarily removed from the head and placed in another stack (stack2). Otherwise, it is partitioned into two smaller blocks and entered at the end of the list. By placing the partitioned b.b.'s at the end of the list, it is ensured that the sizes of the b.b.'s in the list are near about the same. Therefore, the resource estimate obtained is, in some sense, a useful one. Example 7.4 illustrates the partitioning process.

Example 7.4 Consider the b.b. of example 7.3. The b.b. is initially passed to the procedure *estimate* through a list as shown in the figure 7.5. It is extracted from the head of that list for further processing in *estimate*. In example 7.3 the operation O when restricted to be scheduled in the interval [5..7] leads to a partitioning of the b.b. along time step 5. The two new b.b.'s obtained as a result are appended to the list, as shown.

Both these b.b.'s are to be scheduled in four time steps, which is smaller that W $(= 5)$. Both these b.b.'s qualify as simple and the l.b. for the new partial solution can now be computed.

After that the two b.b.'s are extracted from the end of the list, merged into a single b.b. The operation O is now relaxed to its original time frame using the popped move $< BACK, 3, 7, o >$ from the stack. In this particular no more moves need to be made to compute the resource requirement for the b.b., which is now put back into the head of the list. The procedure *estimate* now returns the computed l.b. for the b.b. It also leaves the list the way it was when it was invoked. □

When the list becomes empty, it is assured that the sizes of all the (partitioned) b.b.'s is less than or equal to W. When this condition is satisfied no more partitioning needs to be done, and the set of stacked (stack2) b.b.'s constitute the partial schedule. If $W = 1$, then this is also the complete schedule and corresponds to a feasible design point. Otherwise, the design point found is an approximate one. If this point corresponds to a design with a better (lower) resource estimate than that of the best stored design then it replaces that design, otherwise it is rejected and backtracking is initiated. The algorithm terminates with a failure if there exists a partition where the design parameters of *NFUS* and *NBUS* cannot be possibly satisfied.

It may be noted that, if instead of appending the b.b.'s to the list, they were pushed back into the head then balancing of block sizes would not have been achieved. The blocks at the head of the list would be partitioned till they became simple and made way for blocks behind. Thus a lot of time would be spent in refining the blocks at the anterior of the list which could go waste if a block at the end of the list turned out to have an acceptably high value of l.b. on the resource requirement.

The requirement for each resource is generated in the form of the a tuple $< m, w, j >$, where m is the number of units of that entity occurring in a window of size w in the b.b. j. Such tuples are generated for the maximum number of operations per time step, the bus requirement, the storage access point requirement and the requirement for hardware operator for each type of operation. $\lceil \frac{m}{w} \rceil$ is the l.b. on that resource. Tuples, instead of the resource requirement, are generated because this information is needed by the exploration heuristic used in the DSE tool (section 7.5.1).

The search mechanism explained above has one anomaly. The problem is that when REPS is being done with a relaxed time constraint then the search space turns out to be far larger than when REPS is being done with a tighter time constraint. This situation is addressed by running an approximate scheduling algorithm on the current b.b. before going ahead to partition it into smaller b.b.'s. Basically a time and resource constrained scheduling type approximate scheduling algorithm is required. If a time constrained scheduling type algorithm is used then the final resource requirement after scheduling should not exceed the l.b. estimate of the resource requirement, while satisfying the design parameters. If an resource (hardware operators) constrained scheduling type algorithm is used then the time steps required should not exceed the available number of time steps. If the approximate scheduling algorithm terminates successfully then the current b.b. may be assumed to satisfy the l.b. on the resource estimate and the remaining b.b.'s may be examined.

The pseudo code for the search scheme is given in figure 7.6. The first line of the procedure *REPS* checks whether the block currently being handled is simple. A block is considered to be simple if either i) the block is to be scheduled in no more than W time steps or ii) the block can be scheduled using the approximate algorithm without violating either the current time constraint or the current level of resource estimate. We illustrate the working REPS through the following example.

```
procedure REPS(bblock,list)
if (bblock is simple)
{   if (queue is empty)
    {   if (present cost is better than best cost)
            replace best solution with present solution
        if (current l.b. is not exceeded)
            return TRUE
        else return FALSE
    }   else
    {   extract bblock from head of list
        return REPS (bblock /* from list head */, list)
        push back bblock /* from list head */ back into list head
    }
}   else
{   determine partition time step of bblock
    check for operation in bblock whose time frame crosses the partition time step
    do
    {   while (such an operation exists)
        {   determine the asap time and alap time of the operation
            push <BACK, asap time, alap time, operation> into stack
            push <FORTH, partition time, alap time, operation> into stack
            restrict operation to lie between asap time and (partition time - 1)
            do
            {   recompute time frames of operations
                compute the l.b.'s of the resource requirements
                if (the l.b.'s are infeasible or l.b.'s indicate an inferior design
                    do
                    {   pop <flag, asap time, alap time, operation> from stack
                        restrict operation to between asap time and alap time
                    }   while (flag = BACK and stack is not empty)
            }   while (backtracking was done and stack is not empty)
            if (stack is empty) return
            check for operation in bblock whose time frame crosses the partition time step
        }
        subdivide /* current */ bblock into two sub-blocks
        append these sub-blocks at the end of the list
        extract bblock from head of list
```

```
        lb_match = REPS (bblock /* from list head */, list)
        push back bblock /* from list head */ back into list head
        remove the two bblocks from the tail of the list
        cleave the two bblocks back to a single bblock
        do
        {   pop <flag, asap time, alap time, operation> from stack
            if (flag = BACK or lb_match)
                restrict operation to between asap time and alap time
        }   while (flag = BACK and stack is not empty)
        if (lb_match)
        {   while (stack is not empty)
                pop <flag, asap time, alap time, operation> from stack
            return TRUE
        }       recompute time frames of operations
        if (stack is not empty and no operation crosses the partition time boundary)
        {   compute the l.b.'s on the resources
                if (l.b.'s are infeasible or l.b.'s indicate inferior design
                {   pop <flag, asap time, alap time, operation> from stack
                    restrict operation to between asap time and alap time
                }   while (flag = BACK and stack is not empty)
        }
    }   while (stack is not empty)
    return FALSE }
```

Figure 7.6: Pseudo code for REPS.

Example 7.5 We consider the DAG shown in figure 7.7 for scheduling in ten time steps. We first note that DAG's of this type pose a difficulty for the hardware l.b. estimator described earlier. If we compute the l.b. for scheduling in nine time steps then the estimator will report a requirement of two adders, whereas three adders will be actually needed.

For illustrating the working of REPS we consider a schedule in ten time steps for the aforementioned DAG. In figure 7.8 we illustrate the main actions taken by REPS. An inspection of figure 7.7 reveals that two adders will be required. In the start state (state 0) of figure 7.5 the correct l.b. is obtained, but the approximate scheduling algorithm fails to schedule using two adders in ten time steps. Hence partitioning of the DAG is required. We choose the fourth time step for partitioning. The time frame of the operation marked 1 in figure 7.7 spans across this time step. We restrict it to be scheduled on or before time step four (state 1). This decision does not complete the partitioning of the DAG. Partitioning is completed after the operation marked 2 in figure 7.7 is scheduled after time step four (state 2) in figure 7.5. The l.b. continues to be three and this time the approximate scheduling algorithm succeeds in finding a schedule without violating the l.b. This case is recorded as the current best solution. Backtracking is initiated. Since the l.b. of the parent state (state 1) matches with the

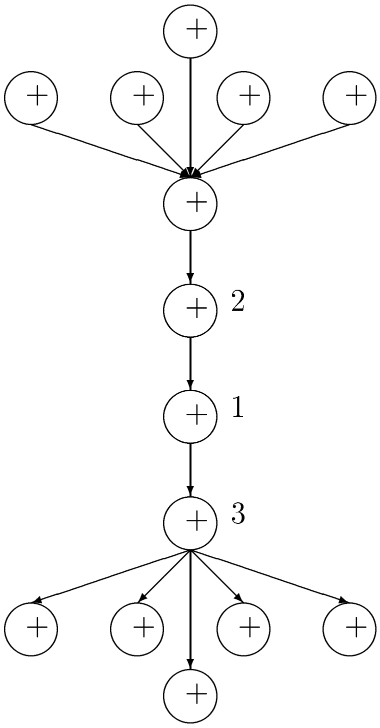


Figure 7.7: DAG for example 7.5.

current cost, the other child of state 1 is not generated. Backtracking is continued to state 0. Now the other option of scheduling the operation marked 1, above or at time step 4 is exercised, leading to state 3. The l.b. continues to be two and the partitioning is completed by restricting the node marked 3 in figure 7.7 on or before time step 4 (state 4). The approximate scheduling algorithm succeeds and a new better solution is recorded. Backtracking is initiated and continues to the start state and the algorithm terminates. In each of the dashed boxes the status of the queue is also shown. After partitioning, in state 2 and state 4, the two smaller DAG's are entered into the queue. The schedule of the approximate algorithm which obtained the best solution (for this problem) is accepted as the schedule. It must be noted however, that in some cases partial schedules may be returned when $W > 1$ and l.b.'s and u.b.'s from solutions obtained by approximate algorithms do not match. □

7.4.3 Special Handling of Operations

Intuitively, the objective of partitioning is problem reduction through decomposition. Special care has to be taken for operations whose execution does not get completed within a single time step. Such cases arise from two types of implementation of operations. One is when the implementation is simple multi-cycle and the other is when the implementation is pipelined. We consider only simple arithmetic pipelined implementations because these are the ones that are most commonly used in practice in DPS. When multi-cycle or pipelined operations are present they may sometimes cross partition boundaries. In such situations the computation of the lower bound is a little more

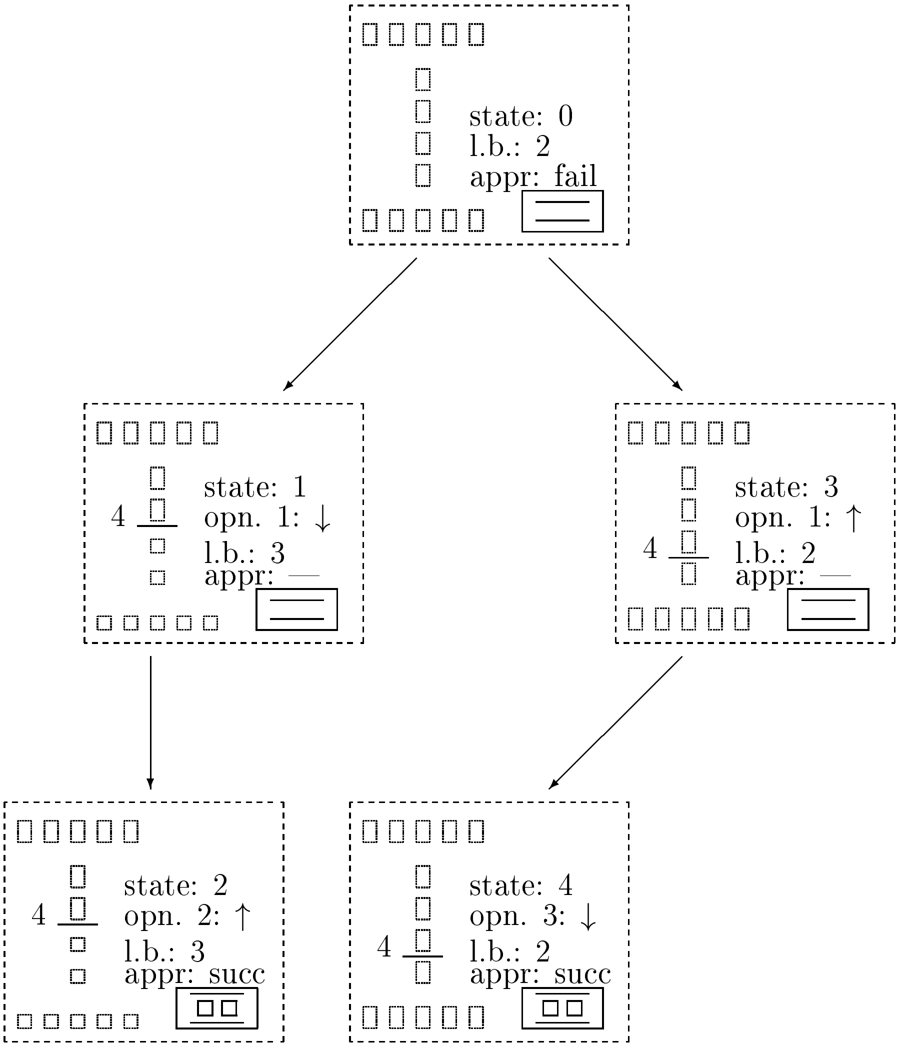


Figure 7.8: REPS Search tree for example 7.5.

involved. Operations that are implemented by multi-cycle hardware operators will be referred to a multi-cycle operations. Similarly operations having pipelined implementation will be referred to as pipelined operations.

REPS handles multi-cycle operations in the following manner. Suppose that the time frame of a multi-cycle operation of k time steps crosses the partition boundary set at time t. Up to k possibilities need to be examined. These are, initiating the operation at times earlier than time step $t-k$, initiating the operation at times $t-k, \ldots, t$, and at times later that t. Initiation of the operation at specific times is the additional overhead for handling multi-cycle operations. When more than a single operation crosses the partition boundary, partitioning is initiated with the operation requiring the least number of cycles for its execution.

Handling of pipelined operations is as follows. Consider a p-stage pipelined implementation of an operation of type x. The result of such an operation will be obtained $p-1$ time steps after initiation. Therefore, while scheduling the number of time steps to execute operations of type x should be taken a p. The minimum number of primitive pipelined operators of type x is taken as the lower bound of the number of initiations of operations of type x in any window in a partition. Like multi-cycle operations there is an additional overhead involved for dealing with pipelined operations when their time frame crosses a partition boundary. The scheduling decisions that need to be taken in this case are identical to those for the multi-cycle operations case.

7.4.4 Handling Multiple Basic Blocks

The procedure *REPS* can accept multiple b.b.'s in the starting list instead of a single b.b. and produce the correct estimate of the resources. If the time constraint on all the b.b.'s of the design are not close to each other then the advantage of balanced partitioning will be lost. Therefore, multiple lists are used, one for each b.b. These lists are processed as explained in procedure *REPS*, except that the list which is to be processed currently is chosen to be the one for which the average size of the b.b.'s is the largest. This way balancing is achieved even when dealing with designs involving multiple b.b.'s.

7.5 Scheme for DSE

We now describe the overall scheme for design space exploration. At the heart of the DSE technique is the resource estimation and partial scheduling algorithm (REPS) which is repeatedly invoked with varying time constraints. The time constraints with which REPS is invoked is determined by the exploration heuristic in section 7.5.1. With each invocation REPS either indicates that the time constraint is not feasible or it returns a design point and a schedule. The design points thus obtain are used to obtain the design space. When a new design point is obtained one of the three conditions will be true.

The point is dominated by existing design points. In this case this design point has to be discarded.

The point dominates a set of the existing design points. All the dominated points have to be discarded and the new point has to be incorporated in the design space.

It neither dominates, nor is it dominated by other design points. The point is simply incorporated in the design space.

In this manner the fastest design requiring maximum hardware, the slowest design requiring minimum hardware and intermediate non-dominated designs are obtained. If the parameter W is set to one then we have a complete schedule and the corresponding hardware requirement. If $W > 1$ then REPS will generally return a partial schedule and an approximate hardware requirement. In the latter case it is desirable to obtain the complete feasible schedules, which will be needed for performing subsequent allocation and binding. These schedules will have to be obtained using approximate scheduling algorithms. The detailed scheme of obtaining complete schedules from approximate design points is explained in sections 7.5.2 and 7.5.3.

7.5.1 Exploration Heuristic

The resource cost estimation scheme described above requires the number of time steps for each b.b. to be specified. To start with, the number of time steps for each DAG is set to its critical length, and then REPS is invoked. The resulting resource requirements are computed from the tuples, as explained above, and examined. It was mentioned in section 7.4.2 that the requirement for each hardware resource or the requirement of f.u.'s, buses, etc. are generated in the form of the a tuple $< m, w, j >$, where m is the number of units of that entity occurring in a window of size w in the b.b. j. In case the any of the design parameters is violated a corrective action is taken as follows. Suppose that a design parameter X having the value v_X is violated, i.e. $\lceil \frac{m_X}{w_X} \rceil > v_X$. Consider the effect of adding $i, i > 0$, time steps to the DAG of the b.b. j_X. Now the earliest time of each operation o, $t_{a,o}$ remains unaltered, but $t_{l,o}$ goes up by i. Therefore, each operation previously restricted to lie in a window of size w will now lie in a window of size $w + i$. A minimal number of time steps $t_X > 0$ is added to w_X so that i.e. $\lceil \frac{m_X}{w_X + t_X} \rceil \leq v_X$. REPS is invoked after making the correction.

The DSE retains the set of mutually non-dominating design points that have been found. When a set of design point, characterized by the time constraints and the resource cost estimate, is found to be feasible it is compared with the stored design points. If is dominated by any point then it is not included in the set. If it dominates any point of the set then it replaces that point. Exploration continues with a new set of constraints, generated as follows. For each operation O whose requirement exceeds unity, we identify the DAG's where it is required maximally. In each of these DAG's we determine the time t by which the time constraint of that DAG should be relaxed so that the new requirement of the operator will be one less, i.e. $\lceil \frac{m_O}{w_O + t} \rceil = \lceil \frac{m_O}{w_O} \rceil - 1$. Let t_O be the maximum of all the times computed above. Let D_O be the DAG where this relaxation may be effected. Let $O^\star$ be the operation for which t_O has the minimum (non-zero) value of all the t_O's. Let $D_{O^\star}$ be the corresponding DAG.

We now relax the time constraint on the b.b. for $D_{O^\star}$ by $t_{O^\star}$ so that $\lceil \frac{m_{O^\star}}{w_{O^\star} + t_{O^\star}} \rceil = \lceil \frac{m_{O^\star}}{w_{O^\star}} \rceil - 1$. This is the heuristic use to conduct the exploration of the design space. Exploration is terminated when the resource requirements of all the operations becomes unity.

7.5.2 Scheduling Schemes for Use with DSE

We have noted that the REPS generates ⟨*hardware cost, performance*⟩ estimates and a schedule for a given design input. When the grain of partitioning is a single time step, the schedule obtained is necessarily complete. The schedule is obtained using a combination of successive partitioning and application of approximate scheduling. For a larger grain of partitioning the schedules obtained may be partial. In the partial schedule an operation, instead of being confined to a single time step is now restricted to be in a partition, which could extend over a few (about five) time steps. For subsequent allocation and binding complete schedules are needed. Most of the existing scheduling algorithms, like FDLS [41], can be adapted to work with the partially scheduled DAG's generated by REPS. However, the performance of such modified heuristic algorithms may not match the performance of the original algorithm. We have also developed a genetic list scheduling algorithm to work with the partial schedules output by REPS.

There is a second and more important aspect that needs to be addressed. It may be noted that the resource estimates are lower bounds and not exact estimates. It is, therefore, quite possible that for a time constraint and a set of hardware operators, as indicated by a design point, a feasible solution might not exist. Even if such a solution does exist, it might be missed out by the approximate scheduling algorithm. However, feasible solutions will be present in the neighborhood of a design point. We would like have schedules with pragmatic hardware requirement and performance. We, therefore, resort to a systematic generation of schedules in the neighborhood of a design point reported by REPS and retained by the DSE mechanism as a non-dominated design point. We rely on existing scheduling algorithms and use them in an appropriate framework. Such a local exploration scheme should be capable of examining the neighborhood of a design point for feasible non-dominated solutions using approximate scheduling algorithms. The choice of polynomial time techniques here is emphasized, for otherwise an exact method could be used to obtain the schedule in the first place. In the next sub-section we examine the local exploration scheme.

Thus after the first phase of DSE we have a set of design points. With each design point we also have the set of partitioned DAG's which had lead to its f.u. estimate component. At this juncture we complete the schedules of these partitioned DAG's using standard algorithms like FDLS [41] or the scheduling method proposed in [23]. The solutions obtained from this completion gives us upper bound (u.b.) estimates. If these match with the lower bound estimates obtained through DSE, we can terminate with accurate design points and schedules. On the other hand, if the u.b.'s and the l.b.'s differ, we explore around the estimated design point for feasible schedules leading to non-dominated <performance, f.u. requirement> design points. That is, we make limited search (in polynomial time) around the estimated design points obtained earlier.

```
procedure relax(dpoint)
{   set of non-dominated designs = ∅
    try to find schedule with constraints specified in dpoint
    incorporate schedule and actual design point corresponding to schedule into
      the set of non-dominated designs
    while (dpoint could no be satisfied)
    {   dpoint1 = dpoint
        relax time constraint of dpoint1
        while (dpoint1 is not dominated by a design in the set of non-dominated designs)
        {   try to find schedule with constraints specified in dpoint1
            incorporate schedule and actual design point corresponding to schedule into
              the set of non-dominated designs)
            if (dpoint1 is satisfied) then break
            relax time constraint of dpoint1
        }
        relax resource constraint on dpoint
        if (dpoint is dominated by a design in the set of non-dominated designs) then
            break
        try to find schedule with constraints specified in dpoint
        incorporate schedule and actual design point corresponding to schedule into
          the set of non-dominated designs
    }
}
```

Figure 7.9: Heuristic Relaxation Scheme for Local Exploration.

Our study of some list scheduling algorithms shows that these algorithms usually terminated with optimal solutions for small DAG's. Therefore, in our state space generation we performed decomposition in a balanced manner to ensure that the sub-problems generated after DSE are small and more suitable for existing scheduling algorithms.

7.5.3 Local Exploration

Given a design point for which only a partial schedule is available, we first try to schedule using the available time and resource constraints to check for the existence of a solution. If such a schedule is found then we are done. In case scheduling fails for the time and resource constraint as indicated by the design point then the time constraint as well as the resource constraint can be relaxed. The relaxation of these constraints also constitutes a search space. We have adopted a heuristic relaxation scheme, as shown in figure 7.9. The algorithm works in polynomial time. This relaxation scheme effects both resource and time constraint relaxation. Otherwise, it initiates time relaxation on a copy of the the design point, keeping the original one for resource relaxation. The time is relaxed in steps and for each new constraint a schedule is found. The actual schedule found may not satisfy the constraint; anyway the schedule along with the actual design point corresponding to the schedule is incorporated in the set of non-dominated designs. It is necessary to *incorporate* a schedule even if it does not satisfy the design constraint, to accommodate the inadequacy of the approximate scheduling algorithm, and ensure a proper termination of the approximate scheduling scheme. The time constraint is relaxed till the new design point is dominated by one of the designs in the set of non-dominated designs. This marks the end of a run of time constraint relaxations.

Now the resource constraint on the original design is relaxed and the entire process is repeated till the new resource constraint turns out to be dominated by one of the designs in the set of non-dominated designs.

The approximate scheduling algorithm that has been used here is the force directed list scheduling algorithm [41]. However, any other approximate scheduling algorithm can also be used. The quality of the design space actually will be governed by the quality of the approximate scheduling algorithm.

Example 7.6 We summarize the working of the overall DSE technique through this example. We refer to figure 7.10. In this figure the filled circles ($\bullet$) indicate the actual non-dominated design points of a hypothetical design that we would like to uncover through DSE. For small problems where REPS can be run with $W = 1$, these points are directly obtained. We consider a scenario where REPS is invoked with $W > 1$. The design points returned by REPS are indicated by '+' and $\times$. These are approximate design points. We do not consider the points indicated by $\times$ because these are dominated by the points marked $+$. The feasible schedules in the neighbourhood of these points (indicated by the large circles) are now found by local exploration. These points are marked by empty circles ($\bigcirc$). It may be noted that some of these points coincide with the design points found by REPS (marked $+$). These are optimal schedules because the l.b. and the u.b. costs are identical. In other cases these are distinct. $\square$

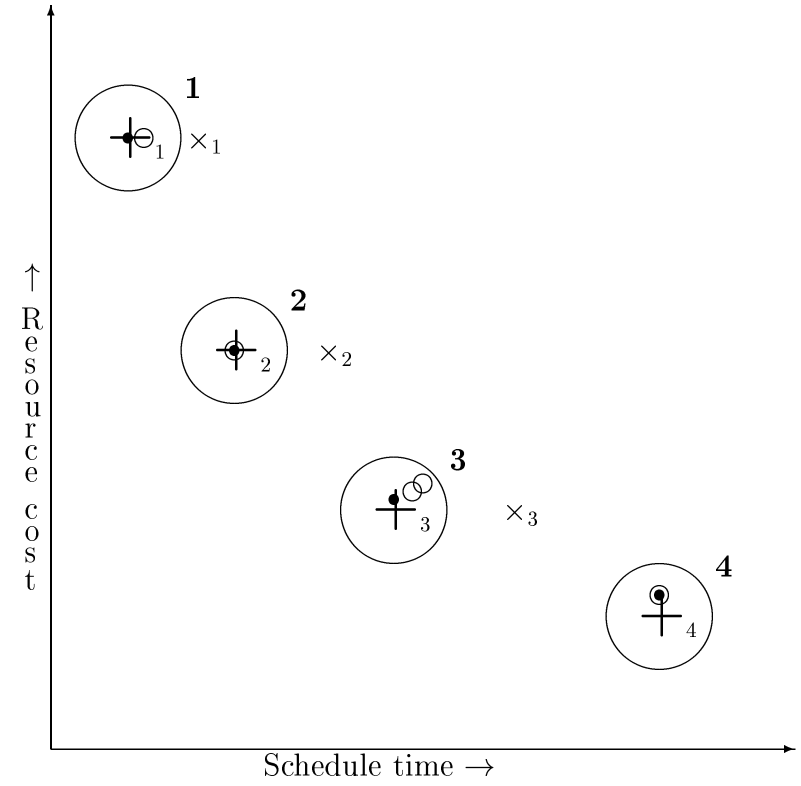


Figure 7.10: Illustration of basic DSE scheme.

7.6 Genetic List Scheduling Algorithm

We complete our DSE paradigm with the presentation of a genetic list scheduling (GLS) technique. We make use of existing scheduling algorithms for the local exploration method described in section 7.5.3. We have also developed a GA based list scheduling scheme which may be directly interfaced with REPS. This will provide a study of GA for scheduling. GLS directly handles a few other practical design requirements as well. These are : i) scheduling of variable assignments and ii) handling of assignment of the output value of an operation to several variables. We have already visited the problem of scheduling variable to variable transfers in chapter 3 and seen that the problem is NP-hard. The operations and other transfers which are dependent on an assignment are indicated in the precedence relations. Special care, however, needs to be exercised for deferring assignment of the output of an operation to variables. This is because the value generated by the operation is directly available when the operation delivers the output. A deferred assignment, therefore, has to rely on another variable which has already been assigned the value. It is necessary to choose a variable which has not been redefined already. The scheduling of multiple assignments is also a similar problem. Unlike most of the other GA's that we have developed we do not use an intelligent crossover here. A brief description of the GA steps are mentioned below. Exhaustive testing of this algorithm has been done and the results are presented later in this chapter.

Solution representation The representation indicates the time step in which each operation has been scheduled. This is the most important information. We also indicate contiguous sequence of time steps in each schedule which correspond to the partitions in the partial schedule generated by REPS. The method of determining these partition boundaries is indicated in the next paragraph. In addition, the representation also indicates the time step in which each transfer to a variable has been scheduled. When the value of an operation is assigned only to a single variable it is not scheduled explicitly but assumed to take place when the operation produces output. Each DAG is also tagged with the number of time steps actually required to schedule it. This is required because schedules of several lengths will be present in the pool of solutions.

Determining partition boundaries It may be noted that the normal size of a partition will be at least two time steps. If every partition is of a single time step then we already have a schedule. We start scheduling each DAG from the last time step and move to the beginning. To start with the last time step corresponds to the last partition of the DAG. We therefore mark the current partition as the last partition. Suppose that the current partition is i and the current time step is j. We would like to identify the beginning of partition i and the end of partition k. Normally $k = i - 1$. However, the partition i may sometimes overrun some of the previous partitions, so we need to accommodate this possibility as well. Initially when i is the last partition k is set to $i - 1$. While considering time step j we would like to know whether this belongs to partition i or marks the end of partition k. We do this as follows.

Let l_i be the last time step in the partition i in the current schedule. Let S_i be the set of operations scheduled from the end of partition i to time step $j-1$ (both inclusive). Let R_i be the set of operations in partition i of the input. Let S_k be the set of operations scheduled in time step $j-1$ and j. Let R_k be the set of operations in partition k of the input. We compute $f_i^j = \frac{S_i \cap R_i}{S_i \cup R_i}$ and $f_k^j = \frac{S_k \cap R_k}{S_k \cup R_k}$. Let f_i^{j-1} and f_k^{j-1} be the previous values for time step $j-1$. If j is the last time step of the DAG then $f_\star^j$ is taken as 0. Let $\delta_i = f_i^j - f_i^{j-1}$ and and $\delta_k = f_k^j - f_k^{j-1}$. If $\delta_i < 0$ and $\delta_k \geq 0$ it suggests that $j-1$ is the last time step of partition k otherwise we are likely to be within partition i. However, the functions are not smooth in practice and the decision is a little more involved. Let t be the first time step of partition i. If partition i appears to have terminated when $j \geq t$ we check for the recurrence of this condition in the next time step. If there is a recurrence then time step $j-1$ is taken as the last time step of partition k. On the other hand if $j \leq t$ and $\delta_k \geq 0$ or $\delta_i \leq 0$ we assume that j is the last time step of partition k. If j moves into the region of partition $k-1$ then k is set to $k-1$.

Fitness function In section 7.3.3 we have we have examined some methods of combining the schedule times of individual basic blocks into a single measure of the overall performance of the design. We use this performance to indicate the fitness of a schedule. This is essentially a weighted sum of the schedule lengths of each DAG (for a multiple basic block design).

Initial population generation The GA works with a population of a fixed number of solutions. To start with this population of solutions has to be created. Each solution is generated by list scheduling all the DAG's available as input. While generating a schedule we also classify contiguous time steps into partitions to correspond to the partitions generated by REPS. The boundaries of a partition are fixed on the basis of the operations schedule in the current partition and whether the current time step is "close" to a partition boundary in the input partial schedule to GLS. For convenience an ALAP list scheduling is used, unlike the conventional ASAP list scheduling. Here the *ready list* contains operations all whose successors have been scheduled. Scheduling starts from the last time step and continues towards the beginning till all the operations in the DAG, or partition, are scheduled. The selection of an operation from the ready list of operations is essentially random. However, preference is given to operations which are present in the current partition of the input set of partitions. Ready deferred assignments are also selected at random.

Reproductive Plan A fixed number of offspring solutions, determined by a parameter to the GA, are produced in each generation. Two solutions chosen from the population at random with a bias towards better fit parents serve as parents for a new offspring solution. The generation of a child solution is based on ALAP list scheduling. A *ready list* of operations and transfers is maintained. Operations from the ready list may be scheduled in the current time step. It may be noted that scheduling should satisfy the design parameters. Thus the total number of transfer, variable accesses, etc. to be scheduled in the current time step must be appropriately chosen. If excess transfers are scheduled then they will block the buses and it will not be possible to schedule operations even if hardware operators are available.

We are given NFUS f.u.'s and the number of hardware operators for each kind of operation present in the DAG's. We are also permitted to use NBUS buses and NVREF distinct variables accesses. Up to NFUS operations that may be scheduled in any time step. We would like to schedule a part of these by inheritance and the rest independently. We first fix the number n $(0 \leq n \leq NFUS)$ of operations which we would like to inherit, as follows. We first determine a number p $(0 \leq p \leq 1)$, so that $p(1-p)^{NFUS} \approx 1$. We then determine numbers p_j, $1 \leq j < NFUS$; we take $p_{NFUS} = 1 - \sum_{j=1}^{NFUS-1} p_j$. We use the numbers p_j $(1 \leq j \leq NFUS)$ as the probabilities of inheriting j operations. We then generate a random number and determine the number of operations n that we will try to inherit. If the ready list does not contain so many operations then n is set to the number of operations in the ready list. This number is determined afresh for each time step.

We now describe the inheritance of operations in the current time step. Let the current time step be j and the current partition of the schedule being formed be k. Let the last time step of this partition be l. We have already scheduled $l-j$ time steps of the current partition. We shall consider the $(l-j+1)$-th time step of the k-th from the bottom of the k-th partition of each parent solution schedule, (if such a time step exists in that partition). Let these time steps be t_1 and t_2 for the two parents. For each parent p $(1 \leq p \leq 2)$, let S_k^p be the operations scheduled in the k-th partition. R_k is the set of operations in the k-th partition of the input. We compute the function $g_i = \frac{S_k \cap R_k}{S_k \cup R_k}$ for parent i. While inheriting form the k-th partition we give preference to the solution with a higher value of g_i.

While trying to inherit the n operations, as mentioned above, we do the following n times. First we determine the parent p_i from which this inheritance will occur randomly using g_1 and g_2. We then consider operations at random from the ready list which have been scheduled on or later than time step t_i of parent p_i. If such an operation is found then it is scheduled in the current time step. After operations are inherited we similarly inherit variable transfers which are ready. This completes the inheritance of operations and transfers. If there are still operations and transfers pending in the current time step and there are also hardware resources available, we randomly schedule as many operations and transfers as possible.

The above scheduling process in continued till all the operations and transfers have been scheduled.

Replacement Policy We have used the replacement policy that we have used for most other GA's here. The policy is to replace the least fit members of the current population with the newly generated solutions.

7.7 Experimentation

The techniques proposed in this chapter have been implemented and tested. The implementation has been done in C in a UNIX environment. The experimentation consists of two parts. In the first part we run our tool on randomly generated partial orders. In the second part we have performed DSE on some common examples like Facet [17], differential equation solver [43] and elliptic wave filter [42]. We now describe our experimentation.

7.7.1 Experimentation on Randomly Generated Partial Orders

Our basic scheme here is to generate a number of problems for a particular problem specification and study the nature of the solutions obtained using our DSE tool. In particular we would like to examine the following.

- Usefulness of the design space exploration concept at the level of scheduling.
- Importance of taking design parameters into account.
- Performance of our genetic list scheduling technique (GLS).

We have used relatively large examples for doing the experiments. Owing to the hardness of the scheduling problem it was not feasible to obtain their exact solutions. We have therefore adopted a two fold strategy to check the quality of results obtained. Firstly, we generate the input partial orders (p.o.) so that we get to know one feasible design point. We would expect the design points obtained by the DSE technique to dominate this design point. This is similar to the upper bound based testing scheme we proposed and used to test the PA techniques. Secondly, we have run the force directed list scheduling (FDLS) [41] and the lower bound based scheduling algorithm (LBBS) [23] along with GLS to obtain a comparison of results.

We now explain the exact experimentation that has been done. We have performed three sets of experiments. Each set is characterized as follows:

The design parameters. For one set of experiments we have used two f.u.'s and for the other two we use three f.u.'s, i.e. NFUS is either two or three. NBUS and NVREF are both chosen as $3 \star NFUS$.

The number of operations in the p.o.'s. We have considered p.o.'s with twenty, twenty five and thirty operations.

The hardware operators used to generate the p.o.'s. The primitive hardware operators used are $\oplus$ (ex-or), $-$ (minus) and $+$ (plus) with costs 10, 9 and 8, respectively. The operators used for the three sets of experiments are $\langle \oplus, -, 2+ \rangle$, $\langle \oplus, -, 2+ \rangle$ and $\langle 2\oplus, 2-, 2+ \rangle$.

We generate the p.o.'s so that we shall have an upper bound on the time needed to schedule the p.o. using the f.u.'s specified, and the design parameters are also satisfied.

Total number of experiments	30
Distribution of design points	2 (7 cases), 1 (23 cases), (total 37 cases)
Number of cases where actual schedule for GLS exceeds estimated time	2
Number of cases where FDLS requires more than permitted number of f.u.'s	34
Number of cases where GLS outperforms FDLS	3
Number of cases where FDLS outperforms GLS (possibly using more f.u.'s)	2
Number of cases where LBBS requires more than permitted number of f.u.'s	34
Number of cases where GLS outperforms LBBS	20
Number of cases where LBBS outperforms GLS (possibly using more f.u.'s)	1

Table 7.1: Summary for p.o.'s with 20 operations.

This will give us the required feasible design point for each input. We do not attempt to find the exact schedule using DSE but rather a set of approximate design points with their associated hardware resource estimate and partial schedule. In all experiments the granularity of the search (that is the value of W) for REPS is five time steps. For each design point the partial schedules along with the hardware resource requirements are passed on to GLS for final scheduling. GLS finds a schedule satisfying the design parameters using the given hardware resources. For each design point we also run FDLS with the hardware operators and LBBS with the schedule time on the original p.o. We compare the results obtained with those of GLS. In most cases these algorithms require more f.u. sites than NFUS. We have, therefore, also run GLS with one and two additional f.u.'s without altering the other design parameters, to obtain a better comparison of results. The detailed results for the three sets of experiments have been tabulated in appendix C. We now present a summary of the results obtained in tables 7.1, 7.2 and 7.3.

We observe in tables 7.1, 7.2 and 7.3 that the number of design points for a given p.o. and a set of design parameters is reasonably large. In our experimentation with DAG's of thirty operations there is only on DAG (or p.o.) having only one design point in its design space. This is also true in our experimentation with eight DAG's of twenty five operations. Moreover, tables C.2, C.1 and C.3 of the raw data for DSE indicate that the distribution of the points can be arbitrary. This leads us to conclude that DSE is a useful design step in data path synthesis. From the tables it is also evident that GLS is very competitive with the two scheduling algorithms with which we have done

Total number of experiments	8
Distribution of design points	2 (7 cases), 1 (1 case), (total 15 cases)
Number of cases where actual schedule for GLS exceeds estimated time	2
Number of cases where FDLS requires more than permitted number of f.u.'s	8
Number of cases where GLS outperforms FDLS	2
Number of cases where FDLS outperforms GLS (possibly using more f.u.'s)	NIL
Number of cases where LBBS requires more than permitted number of f.u.'s	11
Number of cases where GLS outperforms LBBS	6
Number of cases where LBBS outperforms GLS (possibly using more f.u.'s)	NIL

Table 7.2: Summary for p.o.'s with 25 operations.

Total number of experiments	8
Distribution of design points	4 (2 cases), 3 (2 case) 2 (9 cases), 1 (1 case) (total 33 cases)
Number of cases where actual schedule for GLS exceeds estimated time	5
Number of cases where FDLS requires more than permitted number of f.u.'s	16
Number of cases where GLS outperforms FDLS	7
Number of cases where FDLS outperforms GLS (possibly using more f.u.'s)	NIL
Number of cases where LBBS requires more than permitted number of f.u.'s	24
Number of cases where GLS outperforms LBBS	23
Number of cases where LBBS outperforms GLS (possibly using more f.u.'s)	NIL

Table 7.3: Summary for p.o.'s with 30 operations.

the experimentation.

7.7.2 DSE on Common Examples

Tables 7.4, 7.5 and 7.6 indicate the design points obtained after design space exploration of Facet, Diffeq. and Elliptic Wave Filter, respectively. All these designs are for single cycle implementations of the operations. The first two columns indicate design parameters. The design points obtained after design space exploration are indicated under "DSE", while the actual results obtained after allocation and binding are indicated under "Actual". While computing the costs of the f.u.'s, the cost of each hardware operator is taken as follows: $\mathrm{cost}(/) = 160, \mathrm{cost}(\star) = 160, \mathrm{cost}(+) = 20, \mathrm{cost}(-) = 20, \mathrm{cost}(<) = 10, \mathrm{cost}(|) = 10$ and $\mathrm{cost}(\&) = 10$. The allocation and binding has been done by the GABIND scheme presented in the next chapter. Each block of rows in a table indicates the design points obtained for a particular set of parameters. For Facet the design points obtained by DSE match the actual designs obtained after allocation and binding. This is also true for the elliptic wave filter example in table 7.6. For Diffeq. the actual implementation of the design points indicated in rows 2, 3 and 4 of table 7.5, require an additional adder in each case. For two f.u.'s and seven time steps for Diffeq. the operations scheduled in three time steps were as follows: $<\ \star\ +\ >$; $<\ \star\ -\ >$; and $<\ +\ -\ >$. Therefore, although at most one $+$ and one $-$ are scheduled in any time step, it is not possible to have an f.u. configuration using two f.u.'s where at least a $+$ or $-$ is not repeated. For the case with three f.u.'s and seven time steps, however, such a problem did not exist. Yet an additional $+$ was used to keep the switch cost low. The implementation of Diffeq. using two f.u.'s in seven time steps is especially nice, requiring only three switches, (filled circles in figure 7.12). The design point indicated in row 2 is for designing with only two f.u.'s, whereas there are four types of operations distributed over seven time steps. For the design point indicated in row 3 of table 7.5 the number of time steps is four, exactly equal to the length of the critical path in the data flow graph. Data paths of various schedules obtained after DSE and scheduling for the differential equation example are given in figures 7.11 to 7.14. The filled circles indicate switched connections and unfilled circles indicate unswitched connections. The variables in each memory are indicated. (However, this does not indicate memory size as explained later in chapter 8). The direction of arrows indicate read or write ports. No direction is specified for read/write ports. Details of data path structure are given in chapter 8.

Schedule for Diffeq. using two f.u.'s and six time steps

Data paths for this schedule is given in figure 7.11.

```
  x = dx +  x       v1 = dx *  x
 v0 =  u *  3       v6 =  u * dx
 v2 = v0 * v1       v3 =  y *  3
        x <  a       y =  y + v6
```

num. of F.U.s	num. of bus	DSE			Actual		
		h/w opr. req.	F.U. cost	num. time steps	h/w opr. req.	F.U. cost	num. time steps
2	6	1⋆, 1/, 1+, 1-, 1&, 1—	380	5	1⋆, 1/, 1+, 1-, 1&, 1—	380	5
3	9	1⋆, 1/, 2+, 1-, 1&, 1—	400	4	1⋆, 1/, 2+, 1-, 1&, 1—	400	4
		1⋆, 1/, 1+, 1-, 1&, 1—	380	5	1⋆, 1/, 1+, 1-, 1&, 1—	380	5

Table 7.4: DSE results for Facet.

num. of F.U.s	num. of bus	DSE			Actual		
		h/w opr. req.	F.U. cost	num. time steps	h/w opr. req.	F.U. cost	num. time steps
2	6	2⋆, 1+, 1-, 1<	370	6	2⋆, 1+, 1-, 1<	370	6
		1⋆, 1+, 1-, 1<	210	7	1⋆, 2+, 1-, 1<	230	7
3	9	2⋆, 1+, 1-, 1<	370	4	2⋆, 2+, 1-, 1<	390	4
		1⋆, 1+, 1-, 1<	210	7	1⋆, 2+, 1-, 1<	230	7

Table 7.5: DSE results for Diffeq.

num. of F.U.s	num. of bus	DSE			Actual		
		h/w opr. req.	F.U. cost	num. time steps	h/w opr. req.	F.U. cost	num. time steps
3	9	2⋆, 3+	380	18	2⋆, 3+	380	18
		2⋆, 2+	360	19	2⋆, 2+	360	19
		1⋆, 2+	200	21	1⋆, 2+	200	21
		1⋆, 1+	180	27	1⋆, 1+	180	27

Table 7.6: DSE results for Elliptic Wave Filter.

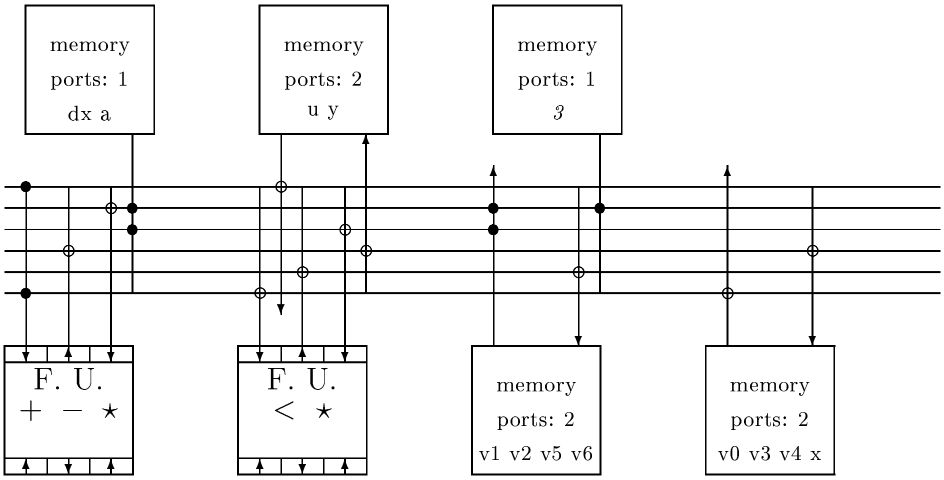


Figure 7.11: Diffeq. data paths for two f.u.'s and six time steps.

```
v4 =  u - v2      v5 = v3 * dx
 u = v4 - v5
```

Schedule for Diffeq. using two f.u.'s and seven time steps

Data paths for this schedule is given in figure 7.12.

```
v0 =  u *  3
v1 =  x * dx       x =  x + dx
v3 =  y *  3            x <  a
v2 = v0 * v1
v6 =  u * dx
v4 =  u - v2      v5 = v3 * dx
 u = v4 - v5       y =  y + v6
```

Schedule for Diffeq. using three f.u.'s and four time steps

Data paths for this schedule is given in figure 7.13.

```
v0 =  u *  3      v1 =  x * dx       x =  x + dx
v2 = v0 * v1      v3 =  y *  3            x <  a
v4 =  u - v2      v5 = v3 * dx      v6 =  u * dx
 u = v4 - v5       y =  y + v6
```

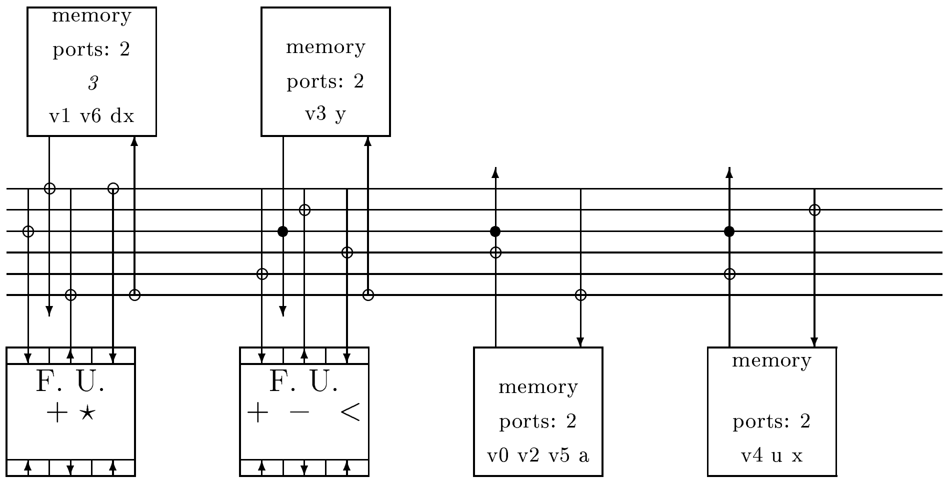


Figure 7.12: Diffeq. data paths for two f.u.'s and seven time steps.

Schedule for Diffeq. using three f.u.'s and seven time steps

Data paths for this schedule is given in figure 7.14.

```
v0 =  u *  3
v1 =  x * dx         x =  x + dx
v3 =  y *  3
v2 = v0 * v1
v6 =  u * dx
v4 =  u - v2        v5 = v3 * dx
      x <  a         u = v4 - v5        y =  y + v6
```

7.8 Conclusion

A given behavioural specification can have a large number of RTL implementations. We can partially characterize RTL implementations by means of design parameters like the number of f.u.'s and the number of buses, when we consider a bus based data path. Even for a given set of design parameters a number of designs are possible which differ in their hardware requirement and performance. These designs constitute a design space which needs to be systematically explored to find the non-dominated designs. The early part of this design space exploration (DSE) problem revolves around the basic scheduling problem which is NP-hard.

We have proposed a scheme for doing this exploration using a combination of controlled search, approximate and genetic scheduling techniques. The search is based on

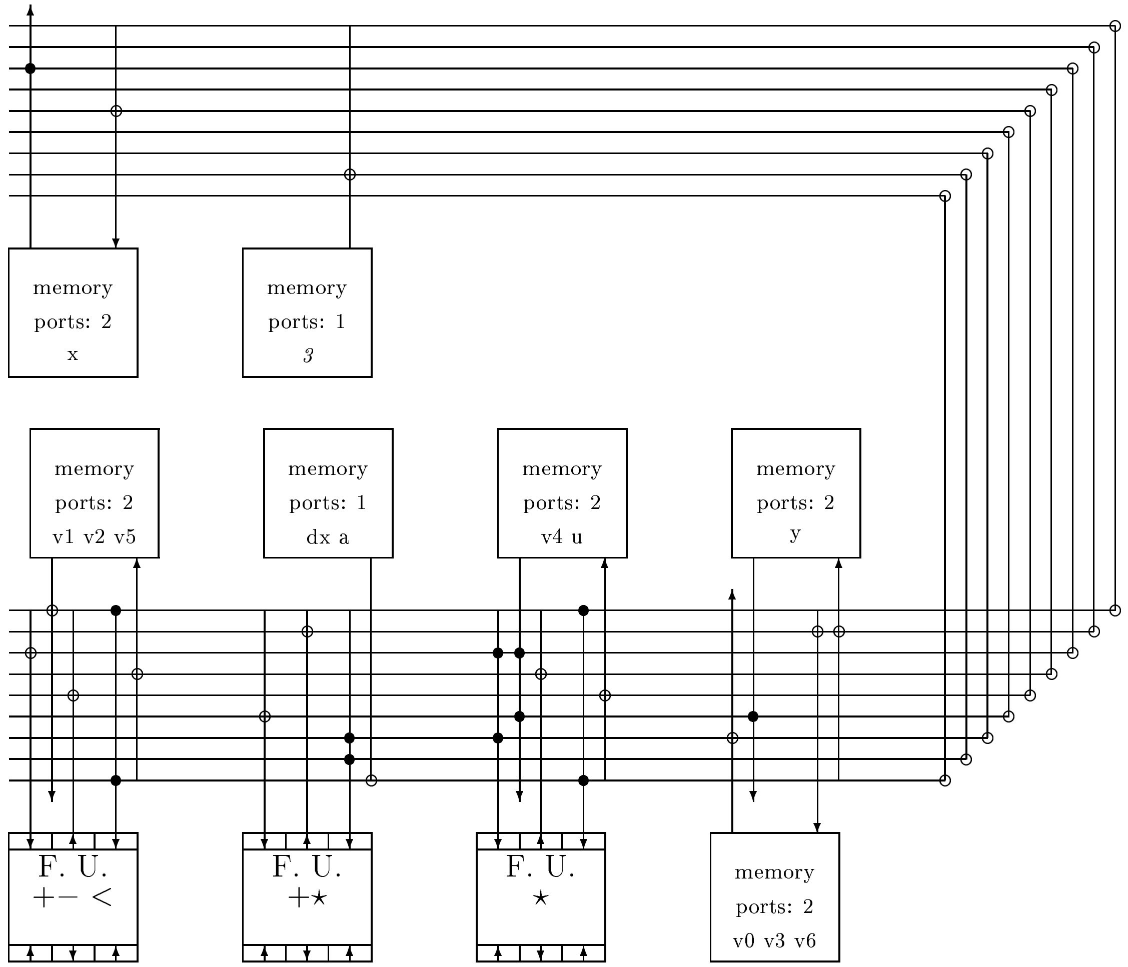


Figure 7.13: Diffeq. data paths for three f.u.'s and four time steps.

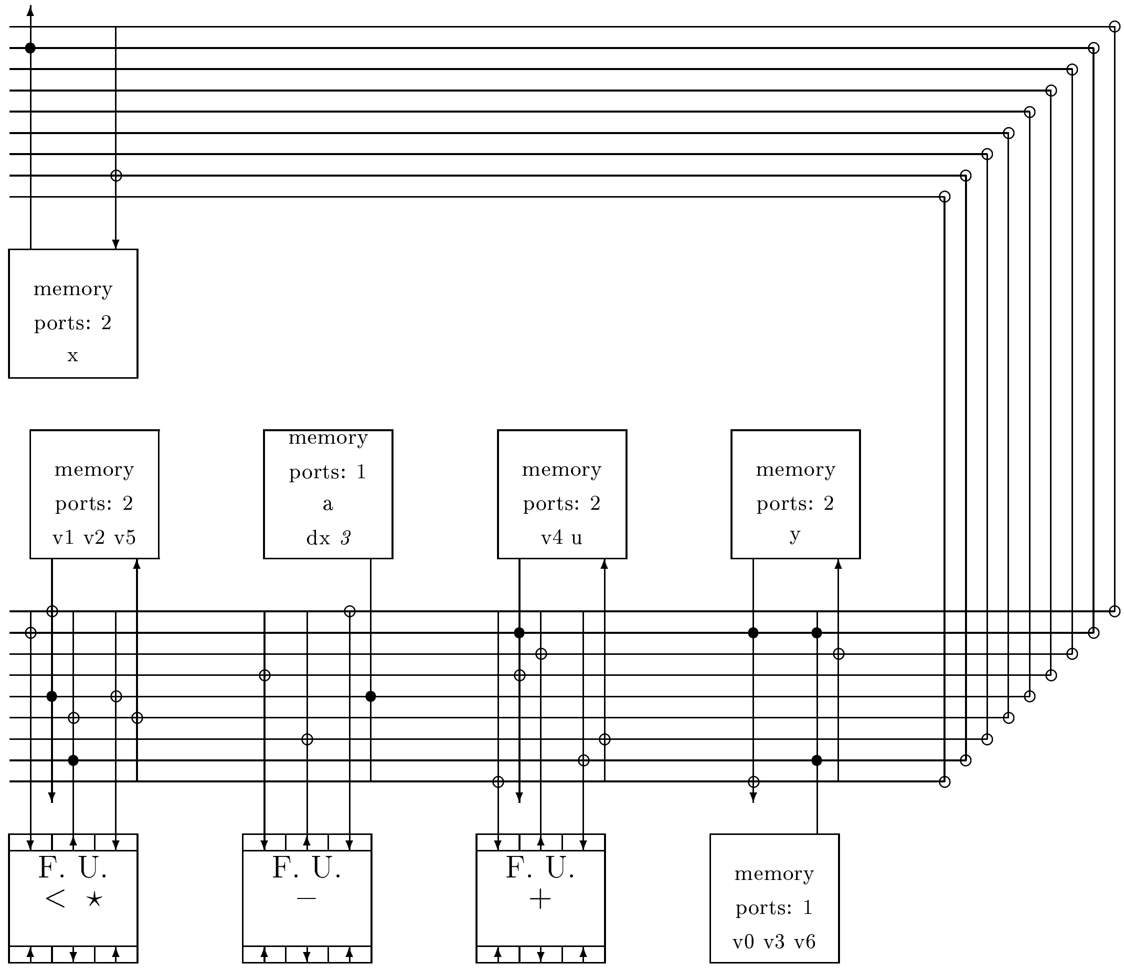


Figure 7.14: Diffeq. data paths for three f.u.'s and seven time steps.

depth first branch and bound (DFBB). DFBB has the advantage of requiring minimum space in the host machine where it has to run. It is necessary to conserve space because the storage for a single (partial) solution, itself, is considerable, We have used a balanced problem decomposition scheme for the DFBB. This has the advantage of partitioning the original problem into smaller subproblems of nearly equal sizes. The importance of doing DSE has been demonstrated through experimentation on randomly generated DAG's of various sizes. Through this experimentation we have noted that the design space for a problem typically has a number of non-dominated design points whose hardware requirements and performances are quite arbitrary. We have also highlighted the importance of working with design parameters to obtain schedules for which data paths having a fixed number of functional unit sites and buses may be constructed. This is advantageous for the subsequent construction of data paths from these schedules. We have also applied the genetic paradigm to the scheduling problem to develop the genetic list scheduling algorithm (GLS). GLS has been demonstrated to compare very favourably with some existing scheduling techniques. It also handles some other practical aspects of scheduling for DPS, like variable assignments, to satisfy the design parameters. We have applied our DSE techniques to some common examples like Facet, differential equation solver and elliptic wave filter and constructed data paths from the schedules obtained. We have noted close conformity with the estimates obtained with DSE and the actual hardware used in the data paths. In the next chapter we discuss the details of allocation and binding, which constitutes the second phase of our solution to the entire DPS problem.

Chapter 8

Allocation and Binding

8.1 Introduction

In the previous chapter we have proposed a method for design space exploration and scheduling as the first part of our solution to the entire data path synthesis problem. In this chapter we propose techniques for the formation of data paths from scheduled data flow graphs. The main inputs required for allocation and binding are the scheduled data flow graphs, the compatibility information of the variables, the design parameters and the cost of the hardware elements. The tasks performed by the allocation and binding program, which we call *GABIND*, are binding of operations to functional units, transfers to buses, formation of functional units, formation of storage, mapping of variables to storage units and interconnect formation. The output is an optimized data path which can correctly implement the computation indicated in the scheduled data flow graphs. The optimization of the data path is intended to jointly minimize the cost of the f.u.'s, the storage units and the switches for interconnection. This cost is a measure of the estimated area required by the data path. In view of the hardness results proved for this problem in chapter 4 we have used the genetic algorithm, instead of a deterministic technique, to solve this problem. GA is increasingly being used to solve EDA problems [74, 75, 76, 77]. However, to the best of our knowledge use of GA for DPS is relatively new.

The construction of the data path is guided by two design parameters, viz. the number of functional units and the total number of buses to be used therein. Other information like the cost of primitive hardware operators, etc. are also required. The output is the optimized RTL data path. Support for multi-cycle operations, the use of memories and pipelined implementation of (complex) operations in the data path is provided. Multi-cycling makes it possible to use a small clock cycle for fast operations while requiring slower operations to be executed over two or more time steps. The interconnection style used here is bus based. The choice of a bus based scheme is motivated by the fact that in a data path comprising of components requiring high interconnectivity the bus based scheme is expected to require fewer active interconnect elements (the switches), than a point to point scheme. Storage is implemented using *multi-port memories and register files* in addition to individual registers. A memory is a regular structure. By placing several variables in a single unit the number of independent sources and sinks of data is reduced.

The *number of f.u.'s* and *buses* are user specified design parameters. The former indicates the total number of sites where operations may be performed, while the latter indicates the total number of paths for carrying data transfers, in the context of allocation and binding. These parameters permit the user to have a reasonable control over the type of design that will be generated by the system. It may be noted, that each f.u. site leads to additional wiring overhead. The number of f.u. sites is, therefore, an important design parameter. In the absence of a good estimate of the routing area, the number of buses used in the design assumes an important role as an empirical estimate. The maximum number of cells that may be present in a register file is also taken as a design parameter. This is typically a small number so that the access time of a memory will not exceed that of a single register by an order of magnitude. A low access time of the memory units is required if the data path is to have a good performance.

Presence of a *common source* of multiple data transfers can be identified. The multiple transfers originating from this source may be routed through common buses to make use of existing links and switches. This has been clarified through example 8.1.

Example 8.1 In figure 8.1, a value from *source* is to be sent to $d1$, $d2$ and $d3$, in the same time step. These are actually three distinct transfer $source \rightarrow d1$, $source \rightarrow d2$ and $source \rightarrow d3$ having *source* as their common source. Assume that, as a result of some other binding decisions the links indicated by the single circles are already present. If the transfers to $d1$ and $d2$ are mapped onto the topmost bus then the existing links will be reused. On the other hand, if $source \rightarrow d1$ is mapped onto the next bus then the link indicated by the double circle will also have to be established. This will lead to the use of additional switches to multiplex the inputs to $d1$. On the other hand it would be desirable to route the transfer to $d3$ through the last bus, on account of the existing connections. It is thus evident that a judicious mapping of transfers with common source to the same bus may be used to optimize the interconnection in the data path. □

Functional and arithmetic pipelining are two schemes for enhancing the throughput of a system. Arithmetic pipelining makes use of pipelined hardware operators such as pipelined multipliers and dividers. Functional pipelining, on the other hand, enhances the throughput by scheduling the operations so as to enable the overlapped execution of several data sets. Thus functional pipelining is transparent to the allocation and binding tool. Special support is necessary for arithmetic pipelining, which has been provided in GABIND.

While the tool incorporates a number of features it does not fully support two features available in some of the contemporary tools. These features are operation chaining and operation commutation.

Operation chaining has the advantage that, at times, it can do away with the necessity of storing intermediate results between operations. This feature can be used to enhance the throughput of the system. However, to be of much practical use, the feature has to be used extensively between the operations in the dependence graph. This entails a rich connectivity between the f.u. sites, implying high interconnect cost. To circumvent an

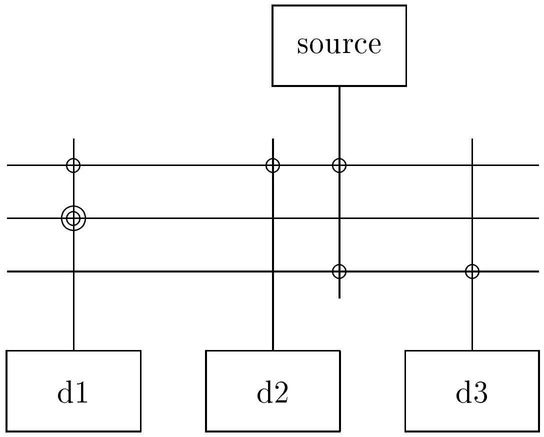


Figure 8.1: Two transfers with a common source.

excessive overhead, a more complex formulation would be required. This was the main reason for not incorporating operation chaining at this stage. As such operation chaining can efficiently implemented in a bus based interconnection structure. A simpler scheme of operation chaining is transparent to the GABIND program and may, therefore, be readily accommodated. In this scheme templates of chained operations are distinctly identified and marked as special operations. Direct implementation for these operation templates are input along with implementations of other operations. The direct implementation of the templates are formed by appropriately interconnecting primitive implementations of the chained operations. A more sophisticated implementation could be obtained by performing a logic optimization of the boolean function represented by the cascade of operations in a template.

The facility to automatically determine the operation commutativity was also not implemented for similar reasons. It may be noted, however, that design refinement through operation commutation is a much easier problem, although it is still a NP-Hard problem. This refinement scheme may be categorized as *post optimization* and has been considered in section 8.5.7 of this chapter.

The techniques which have been proposed in this chapter have been tested on prevailing examples such as Facet, Differential equation solver and Elliptic wave filter. For the last two examples arithmetic pipelining has also been used. The results obtained have been encouraging. We continue with a description of the data path structure in the next section and inputs to the tool in section 8.3. We then present the problem formulation and the genetic algorithm to solve the problem. We next give a detailed description of the algorithmic crossover specifically designed for the problem. The heuristic algorithm used in the crossover is explained next. We finally present the experimental results.

8.2 Data Path Structure

The data path is a bus based interconnection structure. The data path is constructed using f.u.'s, storage elements, system ports, buses and switches to regulate data transfers. A typical such structure is shown in figure 8.2. In the figure a switch is represented by a filled circle. Unswitched or direct physical connections are indicated by empty circles. All data transfers are restricted to take place over the buses. There is no direct link between the components, instead all components are connected to one or more buses. The storage elements may be registers or single port or multi-port memories. In the figure only single and dual port memories are used. Each of the k ports of a k-port memory is individually connected to the buses. A pure read or write port is indicated by an arrow emanating from or incident on the port, respectively. A set of variables are mapped onto each memory. In the figure these are indicated by their indexes, on the memory. This *does not* indicate the number of cells in the memory. The computation of the actual number of cells in a memory is indicated in section 8.2.1. The capability of an f.u. is determined by the set of operations that it implements. In the figure the operations that a f.u. can execute are indicated on that f.u. The number of f.u.'s and buses are both specified by the user as design parameters. As a result of the optimization, the allocation and binding program may not use all the allocated f.u.'s and buses. For example, in the figure bus 1 is left unused. Similarly a f.u. may also be unused.

When any part of the circuit is driven by multiple sources, it is necessary to provide switches between the sources and the destination. For example, bus 0 of figure 8.2 is driven by two memories. Both these memories are connected to the bus via switches. When a receiving point has exactly one driving source, it is assumed that there is a unswitched connection between these two points. This is the case for the connection of the output of the leftmost f.u. with bus 3 in figure 8.2. Otherwise, the connection is switched (indicated by filled circles).

8.2.1 Cost of Data Path

The cost of the data path is the sum of the cost of the individual components. The components are broadly categorized in four groups, viz. functional units, storage units, interconnection elements and interface components, (such as the system ports). As noted earlier, we do not consider the cost of physically routing the wires.

The cost of the components are computed as follows:

Functional units The cost of a functional unit is governed by capability (i.e. the operations it implements) and the cost of the primitive operators, which is available to this tool as an input, through a file of operation costs.

The capability of a f.u. is determined by the operations that are mapped on it, during the binding process. The mapping is considered over all the control steps. This has been explained through example 8.2. If the cost for a given configuration is available in a data base of f.u. configuration costs then the cost of that configuration may be directly retrieved from there. Otherwise, the cost of

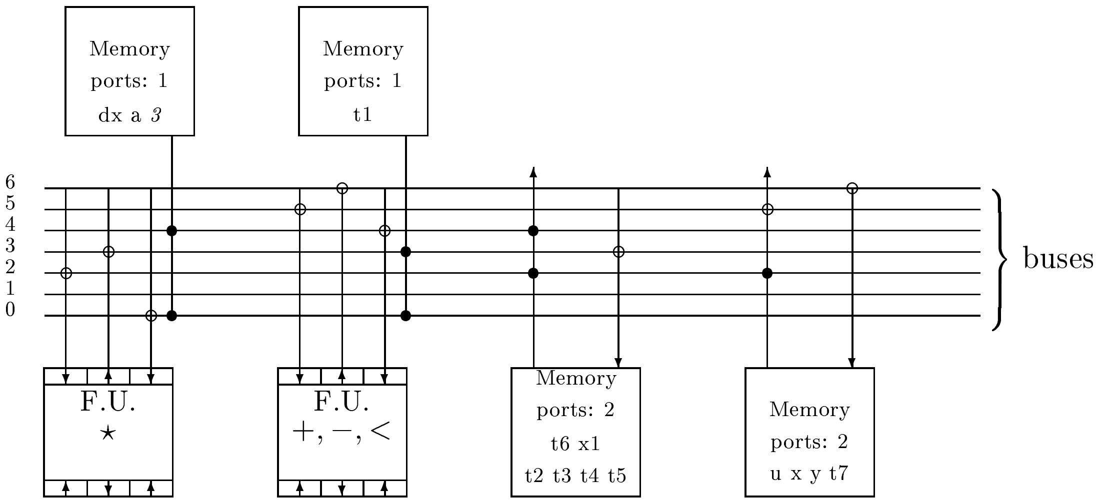


Figure 8.2: A typical data path.

the f.u. is the sum of the costs of individual operators of the operations that it should realize.

Storage unit Data storage is implemented using memories, in general. The cost of some memory configuration will be available in a data base of memory costs. For other configurations the cost of a memory is computed with the knowledge of the number of ports that it has and the number of memory cells that it houses. In order to achieve low access time for a memory, the maximum number of cells that a memory have is restricted to some predefined "small" number. As the number of cells increases, the internal capacitive load on the driver circuits increases and this is reflected in a larger access time. The block structure of a p-port memory with n cells is shown in figure 8.3. The cost $c_m(n, p)$ of a p-port memory with n cells is computed as

$$c_m(n, p) \;=\; n(\alpha p \;+\; \beta) \;+\; \gamma p,$$

where α, β and γ are constants. β is the cost of each cell. γ is the cost of the driver and other logic per port of the memory. α is the cost of the access logic per port per cell, indicated by the $(n \cdot p)$ dashed boxes in figure 8.3.

Switch A switch is required whenever there are more than one points in the circuit driving a particular point. The cost of a switch is taken to be a predefined constant. In the circuit there will be a switch for each line of the bus and the cost of a switch for a b bit bus will be $b \cdot s$, where s is the cost of a single switch. However, if we work with data paths of fixed width then the cost of each "macro" switch is also taken to be a constant number. An individual switch, in the current technology, is likely to be implemented as a CMOS pass transistor switch, illustrated in figure 8.4.

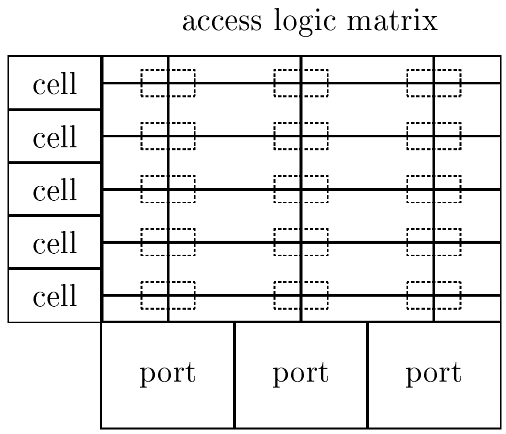


Figure 8.3: The block structure of a multi-port memory.

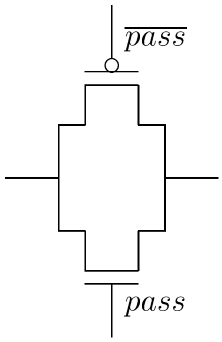


Figure 8.4: A CMOS transmission gate.

Interface element A interface element will be a read, write or read/write port. The cost of such a port is available to the tool as predefined constants.

Example 8.2 Suppose that a data flow graph, scheduled in 5 (say) time steps, has among other operations several additions and subtractions. Assume that the data flow graph has additions scheduled in time steps 0, 2 and 3 which are bound to f.u. 0. Also suppose that a subtraction scheduled in time step 1 is also bound to this f.u., and no other operation is bound to this f.u. in any other time step.

We may conclude that capability of f.u. 0 is addition and subtraction, which may be represented as $< +, - >$. If the cost of this combination of operations is available then it can be used directly, otherwise the sum of the costs of an adder and a subtracter will be taken as the cost of this f.u. $\square$

8.2.2 Considerations for Multi Port Memories

GABIND supports the use of multi-port memories in the construction of the data path. When multi-port memories are used the port assignment problem also needs to be solved. It has been shown in chapter 4 that PA for dual and triple port memories is NP-hard and relative approximation for triple port PA is NP-hard. Thus it may not always be desirable to solve every instance of the PA problem that arises with the use of multi-port memories. A number of options are possible and an appropriate one may be chosen by the design engineer. Some cases of PA that can arise and the ways to handle these are described below.

1. PA is trivial when an individual register is used.

2. PA is trivial for single port memories.

3. PA is also trivial for dual port memories with one read port and one write port.

4. PA is non-trivial for most other cases.

GABIND, therefore, provides a option to use only memory types where the PA is trivial. When memories with non-trivial PA is used one of the following options may be used.

1. Determine the exact PA cost using the PA programs described in chapter 6.

2. Use the probabilistic PA cost estimator, developed in chapter 6, to estimate the cost of the PA. For dual port memories this is a very viable option because cost given by our probabilistic estimator match very well with the actual costs obtained through actual PA.

3. Use a lower bound estimate of the cost of port assignment. Consider a k-port memory, $2 \leq k \leq 3$, and assume that each point is connected to just one port and n_1 write to the memory. The lower bound l_1 of the switch required for the n_1 points may be computed as $\max(0, n-k)+\max(\min(n-k, 1), 0)$. Also assume that n_2 points read from the memory. For each of these n_2 points let w_j, $1 \leq j \leq n_2$ denote whether the point is also connected any other part of the circuit. If one of these points is connected to some other places then $w_j = 1$, otherwise $w_j = 0$. The lower bound l_2 of the switch required for the n_2 points due to the port assignment is $\sum_{j=1}^{j\,=\,n_2} w_j$. The lower bound of the cost of PA for the memory in question is, therefore, $l_1 + l_2$. This is only the partial multiplexer switching cost due to PA for the memory in question. The other multiplexer switching cost must be computed separately to arrive at the total multiplexer switching cost of the design.

If one of the last two options are used then an actual PA must be obtained by running the PA programs on the final design.

8.3 Inputs for GABIND

GABIND requires several inputs. These are the scheduled data flow graph (SDFG), the set of primitive operators, the design parameters and the basic block decomposition of the operations and transfers. Each of these are explained briefly. Some of the pre-processing done by GABIND prior to application of the GA is also mentioned.

8.3.1 The Scheduled Data Flow Graph

This is the most important input to GABIND. This input may easily be generated from any scheduling algorithm and consists of two parts. In the first part the operations that are performed are listed according to the time step where they are scheduled. In the second part all the required data transfers are specified, time step by time step. The advantage of expressing the data transfers separately is that transfers, such as variable to variable transfers, which are not directly associated with any operation can be easily specified.

While doing binding the operations are bound to f.u.'s, the transfers to the buses and the variables to the memories. There is a provision to specify that operations in two or more contiguous time steps should be bound to the same f.u. Similarly, it is possible to specify that transfers in two or more contiguous time steps are to be bound to the same bus. These features are especially convenient for expressing multi-cycle operations. An operation needs to be executed over multiple clock cycles if the operator to implement this operation is not fast enough to do the work in a single clock cycle. For this the f.u. which houses this operator has to be dedicated to the completion of the operation for the required number of clock cycles. During this time the inputs to the f.u. must be steadily maintained. These requirements can be satisfied by making use of the above provisions.

The format for expressing the SDFG is now explained. Each part, viz., the operations and the transfers, are listed time step by time step. An operation is formatted as *<op_type>* *<mc_flag>*. Its type (like addition, subtraction, etc.) is identified by (*op_type*), expressed as a non-negative integer. Presence of multi-cycling for an operation or a transfer is indicated by means of an associated flag (*<mc_flag>*). A transfer is identified by its source and its destination. It may be single cycle or multi-cycle and this is indicated by a flag – the *mc_flag*. The format of a transfer is *<source>* *<destination>* *mc_flag*. A source or destination may be either an input or output of an operation, in general, termed as a *port of an operation* or a *behavioural variable* or a *interface port of the system*. A source or destination is formatted as *<point_type>* *<index_1>* *<index_2>*. The field *point_type* indicates which of the aforementioned categories the source or destination is. When the *point_type* indicates an operation *index_2* represents the port of the operation which participates in the transfer. It is otherwise unused. In this case the actual operation in the corresponding time step, in the first part of the specification where the operations are listed, is identified through *index_1*. Such an exact identification is necessary because more than one operation of a particular type may be scheduled in a time step. When *point_type* indicates a variable or a system interface port *index_1* is used to identify the corresponding entity. A variable is identified by the index of the variable and a port by the index of the port. Example 8.3 features a simple SDFG and the accompanying input specification. This example also illustrates the specification of a multi-cycle operation.

It may be noted that the format used to specify the SDFG is neither intended nor convenient for hand coding. It is designed to be an intermediate notation and meant to be automatically generated by the scheduler after scheduling the DFG's. It is quite easy to see that automatic generation of this format from any scheduling algorithm is straightforward.

Example 8.3 Consider the scheduled DFG of figure 8.5. The SDFG could be represented textually as follows:

1. $x = a * b$ (2-cycle) $y = c - d$
2. `x = a * b (from previous step)`
3. $z = x + y$

Assume that operation types (<*op_type*>) are as follows: $\star \Leftrightarrow 2, - \Leftrightarrow 1$ and $+ \Leftrightarrow 0$. The variable indexes are as follows: $a \Leftrightarrow 0, b \Leftrightarrow 1, c \Leftrightarrow 2, d \Leftrightarrow 3, x \Leftrightarrow 4, y \Leftrightarrow 5$ and $z \Leftrightarrow 6$; and *point_type* $\Leftrightarrow 1$ indicates operation and *point_type* $\Leftrightarrow 0$ indicates variable, these could be formatted as below. It will be noted that the last member of each field is the multi-cycle flag. The operations are specified in the first part as follows:

1. 2 0   1 0
2. 2 1   -1 0
3. 0 0   -1 0

The first line indicates a multiplication ($\star$) and a subtraction ($-$) in the first time step. The second line indicates a $\star$ which is continued from the first time step by multi-cycling in the second time step. The third line indicates a single $+$. The transfers are specified in details in the second part as follows:

1. 0 0 0 1 0 0 0   0 1 0 1 0 1 0   -1 0 0 0 0 0 0
   0 2 0 1 1 0 0   0 3 0 1 1 1 0   1 1 2 0 5 0 0
2. 0 0 0 1 0 0 1   0 1 0 1 0 1 1   1 0 2 0 4 0 0
   -1 0 0 0 0 0 0   -1 0 0 0 0 0 0   -1 0 0 0 0 0 0
3. 0 4 0 1 0 0 1   0 5 0 1 0 1 1   1 0 2 0 6 0 0
   -1 0 0 0 0 0 0   -1 0 0 0 0 0 0   -1 0 0 0 0 0 0

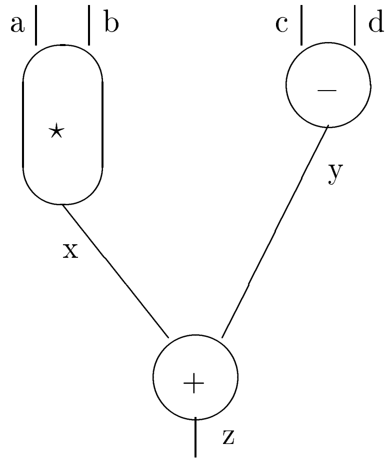


Figure 8.5: A scheduled DFG.

Each line has provision to accommodate six transfers. In the first time step only five transfers are present. The first two transfers are $< 0\,0\,0 \quad 1\,0\,0 \quad 0 >$ and $< 0\,1\,0 \quad 1\,0\,1 \quad 0 >$ The first of these indicates a transfer having variable a as its source and its destination is *operation port 0* (left input) of the operation in column 0 (first column) in the first line of the operation list. This is the $\star$ operation (in the first time step). The second transfer indicates a transfer having variable b as its source and its destination is *operation port 1* (right input) of the operation in column 0 (first column) in the first line of the operation list. The output of the $\star$ operation takes place only in the second time step because $\star$ is a 2-cycle operation. The left and right inputs of $\star$ have to be sustained through the second time step. This is indicated by formatting the first two transfers in the second time step in the same way as those in the first time step, the only difference being that the multi-cycle flag of these transfers is now set to 1. □

8.3.2 The Basic Block Decomposition

GABIND is capable of working with multiple basic blocks (b.b.). The presence of multiple basic blocks is transparent during binding. But the information regarding the basic block boundaries is necessary for determining the variable life times, which is explained in the next subsection. The information regarding the decomposition is included in the file where the SDFG is stored. It is basically a list indicating linear sequences of operations and transfers of each basic block. We now go on to describe some preprocessing that needs to be done to derive the information necessary for variable merging and feasible packing of variables into memories. It may be noted that the variable life times and access times are are not primary inputs, but are computed by GABIND using the b.b. decomposition and the schedule of operations prior to use of the actual genetic algorithm.

Preprocessing for memory formation The behavioural variables are classified into two groups which are *local* and *global*. The local variables are restricted to be live within a single basic block. Global variables may cross basic block boundaries. In section 3.5 the construction of DAG's from a sequence of statements for a straight line code has been explained. Declared variables are often reused while writing a piece of behavioural code. As a result the total number of values that are defined in the behaviour often exceeds the total number of declared variables that are used in the statements. For synthesis, however, each and every value should be associated with a variable name. This requirement is satisfied by the translator of the behavioural specification which automatically generates new variables for this purpose. These variables are called *temporary* variables [7]. It is easy to understand that *such temporary variables* are a special class of local variables.

While implementing storage two types of operations need to be performed on the variables. One is packing variables into memories, and the second is merging variables into a single storage cell. The first operation requires knowledge of the access times of variables, while the second operation requires knowledge of the their life times.

Determination of Access times The access times of variables can be easily determined by inspecting the SDFG, all the variables used or defined are said to be accessed in that time step. *While packing a set of variables in a k port memory, it is necessary to ensure that no subset of these variables give rise to more than k accesses to the memory in any time step.* This restriction is distinct from the restriction that no more than k variables should be simultaneously accessed in any time step. This is because a variable may give rise to two accesses in some time steps, if it is both read and written in that time step. In particular, suppose that there are k_r read ports, k_w write ports and k_{rw} read/write ports. Let there be r read accesses and w write accesses in any particular time step. The relations $r \leq k_r + k_{rw}$, $w \leq k_w + k_{rw}$ and $r + w \leq k_r + k_w + k_{rw}$ should be satisfied. This makes it necessary to use an efficient data structure in the implementation to reveal the access pattern of the variables. Such a data structure has been used in the implementation of GABIND.

Computation of Life times The life times of variables is used to construct the life time conflict graph. The vertices of this graph correspond to the variables. Two vertices are connected by an edge if and only if the two variables are live at the same time. The life time of a variable in each basic block is the sequence of time steps after its definition to its last use, before its next definition, if any. An easy way to determine the life time following the above definition is to start from the last time step of the basic block and move time step by time step towards the beginning of the block. Whether the variable is taken to be live or dead at the end of the basic block is determined by means of a data flow analysis. In case it is found that the variable will have to be used in some successor of the current basic block then its status is live, otherwise it is dead. In each time step the status of a variable is re-evaluated and then it is marked according to its status. If the variable is defined in a time step its status is made dead, if it is used then its status is made live; in this order.

The data flow analysis is as follows: For each basic block, b_i let u_i, d_i, I_i and O_i represent the sets called *uses, definitions, in* and *out*, respectively. Their meanings are as follows:

Uses All the variables used but not defined within this basic block.

Definitions All the variables defined within this basic block.

In All the variables live on entering the basic block.

Out All the variables live at the exit of the basic block.

These sets are related as follows:

$$I_i = (u_i \cup O_i) - d_i$$

and

$$O_i = \bigcup_{b_j \ succ \ b_i} I_j.$$

The sets u_i and d_i are fixed for each basic block. I_i and O_i can be solved iteratively [7], taking their initial values as $\emptyset$.

8.3.3 The Design Parameters

The design parameters required are the number of f.u. sites (NFUS) and the number of buses (NBUS). The implication of the second parameter is evident from the name. That of the first parameter is now explained.

The f.u. configuration need not be specified in advance, it is an outcome of the binding process. What is required is the number of f.u. sites. In each time step, the operations are bound to the f.u. sites. Thus the number of f.u. sites specified should not be less than the number of operations scheduled in any time step. As explained in section 8.2.1 the capability of a f.u. is determined by the distinct types of operations mapped on it over all the time steps.

8.3.4 The Primitive Operators

A primitive operator is the physical implementation of an operation in the behavioural specification (like adder, etc.). Each such primitive operator (like adder or multiplier) may be fully combinational or pipelined. A pipelined operator also carries the information about the number of single time step pipe stages that it has. This information is necessary to avoid output conflict of operations that may be mapped to the same f.u., in different time steps. In general a number of implementations of an operation will be available in the module library. These implementations will vary in their area, delay, power and other characteristics. We need to choose a particular implementation for each operation for use in the data path construction. Each primitive operator will, therefore, be associated with a unique cost. A elegant scheme for grading components in a module library on the basis of physical design objectives has been suggested in [40]. Alternatively, several designs may be obtained for different choices of physical modules. The cost and other information of the selected components are read in at startup form a user supplied data base. An operation and the corresponding operator are identified by means of a unique number, the *op_type*.

8.4 GA Based Solution to Allocation and Binding

8.4.1 Prologue

We have use the genetic paradigm to solve the allocation and binding problem. An inherent advantage of GA is its implicit parallelism. Another positive feature is that at termination several competitive solutions are available in the population of solutions. It is possible to incorporate multiple heuristics in the solution technique. In the design of the GA for this task a number of factors had to be taken into account. In view of the complex nature of the problem a structured representation indicating the various bindings has been used. The *deceptability* inherent in the problem entails careful parent selection and an intelligent crossover. In our early experimentation we found that a simple splicing type crossover is not suitable for this problem and so we have used an algorithmic crossover, based on the *force* heuristic. A population control mechanism to sustain maximum diversity in the population, while at the same time retaining solution with good overall and also partial fitness has been adopted. It was not sufficient to simply give preference to better cost solutions in the genetic pool for selection as a parent for crossover. We used the cost of the memory elements of the data paths as a secondary cost in deciding whether to retain the solution or to replace it with a newly generated solution. This was guided by our observation of the importance of an appropriate memory configuration in the solution.

The SDFG, in general, will have multi-cycle operations and operations mapped to pipelined units. The operations and transfers in parts of the SDFG having multi-cycle or pipelined operations need to be handled time step by time step, as illustrated in example 8.4. Where only single cycle operations are present the time steps can be considered in any order to do the binding. Similarly transfer mappings also need to be done step by step. Memory formation, on the other hand, is based on the simultaneous consideration of the lifetime and access information gathered from all the time steps. These factors need to be taken into account while designing the crossover.

Example 8.4 Consider the transfers given below:

$x = a * b$ (2-cycle)	$y = c - d$
$\mathtt{x} = \mathtt{a} * \mathtt{b}$	$u = c * d$
$z = x + y$	$\mathtt{u} = \mathtt{c} * \mathtt{d}$

The multiplications require two time steps. For example the necessity of implementing $x = a * b$ in two time steps is indicated by repeating the operation in the next time step as `x` $=$ `a` $*$ `b`. Assume that the above transfers are to be mapped using two f.u.'s, identified as 0 and 1. If the operation mapping is done haphazardly then the subtraction in the first time step and the addition in the last time step might both be mapped on f.u. 0, in which case the multiplications cannot be mapped. □

The basic steps in the formation of a new solution through crossover of two parent solutions, in our method, are briefly listed below. First construct the memories of the new solution through a sub-crossover of the memories of the parent solutions. It is then necessary to proceed time step by time step to obtain the complete solution. In each time step the following need to be done. i) Complete the essential transfer and operation bindings to satisfy the data transfer and execution requirements of multi-cycle and pipelined operations. ii) Perform operation to f.u. bindings for the pending operations for this time step. iii) Perform transfer to bus bindings for the pending transfers for this time step. Each f.u. has to be implement the operations that are bound to it. Thus after all the operations have been bound, each f.u. has to be examined to determine the types of operations that it must implement. While binding operations to f.u.'s care has to be taken to minimize the cost of the f.u.'s as far as possible.

The last two steps are jointly called the *completion* step. The completion step in GABIND is done by a combination of inheritance through crossover and a heuristic algorithm. For this a force directed heuristic algorithm, called the *completion algorithm* has been incorporated in the crossover to improve the chances of a better offspring when two solutions are crossed. This algorithm has been designed to make use of the existing partial structure to satisfy the pending bindings of operations and transfers. Thus while binding transfers to buses preference is given to a binding that not only can be satisfied using existing links and switches connected to the bus but also leaves open a similar opportunity for the maximum number of pending transfers. A similar criterion is used for computing the forces for the operation to f.u. bindings. In a time step the binding of pending operations precedes the binding of pending transfers. A limited lookahead for transfer forces has been incorporated in the computation of the operator forces.

It may be noted that the above binding decisions are not independent and one set of binding decisions essentially cannot be taken without knowledge of the other. We rely on iteration of the design steps to achieve a refinement of the design. This is also a natural feature of GA.

8.4.2 Steps for the GA

The steps followed in the GA for solving the problem are as follows:

1. *Design representation*: Since the splicing type crossover has not been used we do not use a bit vector representation of the solution. Instead, a record structure of three fields, one for each class of binding decisions, is used. Each field is a structured array. For each operation and each transfer, the corresponding field indicates the f.u. index or the bus index to which it is bound, respectively. The binding of a variable indicates the memory index and the number of ports that it

has. Example 8.6 indicates a possible binding and the corresponding data path for the SDFG of example 8.3.

2. *Initial population generation*: A population of initial data paths created by randomly generating feasible bindings. Essentially, for each initial solution feasible operation to f.u. binding, transfer to bus binding and grouping of variables to form memories are performed at random. The cost of each solution is computed and stored, The population control data structures are created. After the completion the actual optimization process through the generation of new solutions through crossover can be started.

3. *Replacement policy*: The basic replacement policy is designed to ensure that all solutions generated stay in the population for at least one iteration. This is done by introducing all the solutions generated through crossover during one iteration of the GA into the population, replacing an equal number of existing solutions. The offsprings are not immediately placed in the main population, but are stored in an adjoint pool, to be introduced in the main population once all the offsprings for the current iteration are generated. The basic replacement policy is to displace the lowest cost solutions.

   While designing the system we have observed that it is not a good policy to implement the replacement purely as mentioned above. This because the current low cost solution has a tendency to displace other solutions from the population with its copies thus destroying the diversity of the solutions. We have been able to arrest this phenomenon using a two pronged strategy.

   First we have made provision for the retention of a minimum number of solutions having the k^{th} best cost, $k > 1$. This policy is implemented for up to a fixed value of k. The implication of this policy is illustrated through example 8.5.

   The second provision is specifically aimed at maintaining a good diversity of memory configurations in the population of solutions. The policy is to ensure that the minimum number of distinct memory configurations in the population does not fall below a certain minimum number. This condition may not be satisfied at the beginning, but once sufficient number of memory configurations are produced, it is ensured that this condition is satisfied. When the number of distinct memory configurations is large only the solutions with the best configurations are kept track of for this purpose. The minimum number of solutions for each configuration retained in this manner is also a programmable value. The implication of this policy is that a solution with a high cost may be retained because it has a low cost memory configuration. This policy will be revisited when we consider the parent selection strategy for crossover.

4. *Parent selection*: In each iteration of the GA a specific number of offsprings are generated by crossover. Each offspring is generated from two parent solutions chosen from the population. The basic policy is to select two solutions from the population at random. Normally a solution is selected only once. This corresponds to sampling without replacement.

The actual policy has a number of modifications to this basic scheme, made with the intention of coercing some crossovers between solutions which are either genetically close or which are more fit. The provision to choose parents that are genetically close had to be made to keep a check on the amount of type II deceptability [36] during crossover, to some extent.

To implement crossover between better fit parents a list of solutions whose cost is less than some threshold, determined on the basis of the distribution of the solution costs in the population, is maintained. Solutions can be picked up from this list at random for crossover.

Genetic closeness is difficult to determine. Some amount of closeness is determined by grouping solutions having the same memory configuration. Such solutions differ only with respect to the operation and transfer bindings. Memory configurations to be considered are chosen using two criteria. The first is the cost of the configuration. A certain number of groups of solutions of low cost memory cost is maintained. Secondly, solution groups of identical memory configuration are also maintained according to minimum cost of solutions. Such groups contain those solutions whose memory configurations are the same as that of some low cost solution.

5. *Crossover*: This is the most important step in the GA. The traditional single point or multiple point crossovers have been found to be unsuitable for this problem. This is because the individual binding decisions are most not independent. In general, it is not possible to alter one decision without altering a few others. For this an algorithmic aspect has been incorporated into the crossover along with inheritance.

   While designing the crossover the inheritance mechanism had to be designed so that a tentative partial data path (TPDP) could also be obtained. This structure is to be used to evaluate the quality of a new binding. The heart of the crossover is a *completion algorithm* which takes the TPDP and other inputs to generate a complete data path. The details of the crossover and completion algorithm are explained later.

6. *Stopping criterion*: The GA is necessarily run for a certain minimum number of iterations. Every time there is an improvement it is run for at least another fixed number of iterations, in the hope of another improvement within that time. In the absence of any improvement and expiry of the alloted number of iterations the algorithm is terminated and the data path, based on the binding decisions is output.

Example 8.5 Assume that the population size is 30 and the number of second best cost solutions to be retained is 8. Suppose that, during the present generation, 26 of the 30 solutions are of cost 100 and 4 are of cost 120. Also assume that 10 new solutions (in addition to the existing 30 solutions) have been generated of which 5 are of cost 100 and 5 are of cost 120.

If the replacement policy had been to replace 10 highest cost solutions (of the present population) with the newly generated solutions then all the 4 solutions of cost 120 and any 6 solutions of cost 100 would have been replaced with the new solutions.

With the cost based retention policy only 1 solution of cost 120 and 9 solutions of cost 100 will be replaced with the new solutions. This helps to increase the diversity of the pool of solutions. □

Example 8.6 The SDFG of example 8.3 requires 3 time steps. The multiplication only requires the inputs in the first as well as the second time steps, the output is produced only in the second time step. The operation bindings and then the transfer bindings are given below:

1.	0	1
2.	0	-
3.	1	-

1.	0	1	3	4	5
2.	0	1	5	-	-
3.	3	4	5	-	-

Five buses (distinct transfer paths) are required for this example. The variable bindings for a, b, c, d, x, y and z are as follows: $< 0, 2 >, < 1, 2 >< 2, 2 >< 3, 2 >< 2, 2 >< 3, 2 >$ and $< 3, 2 >$, assuming that the variables $a \ldots d$ are not needed at the exit of the given basic block of code. The '2' in the second field of a variable binding indicates that the memory has two ports. For this design individual registers suffice, the input and the output of the register are treated as two ports of the register. An inspection of the transfers reveal that bus 5 is driven by both the f.u.'s and so a 2-to-1 multiplexer is required. This is the only switching needed for interconnection. □

8.5 Details of Crossover

The basic steps to implement the crossover are as follows:

1. Determining the specific bindings of each parent solution which should be considered for inheritance in the crossover (Prominent Bindings).
2. Obtaining a correspondence between the data path structures of the two parent solutions.
3. Constructing the inheritance plan for operations and transfers.
4. Constructing the memories through inheritance.

5. Generating the offspring using the inheritance plan and the tentative partial structure.

Each of these steps are now explained.

8.5.1 Prominent Bindings

The bindings of the offspring are partially formed by inheriting these from the parent solutions. The cost of a solution is very sensitive to the bindings because an unfavourable binding could give rise to additional switching elements. For this reason the bindings in the parent solutions are first graded. A 0/1 gradation is performed for the operation and transfer bindings. In the implementation the better bindings are marked *core* while the inferior are marked *non-core*. Variable to memory bindings are graded in a continuous scale. The schemes of making the gradation are now explained.

Transfer binding gradation

Each transfer bound to a bus. The aim of this gradation is to measure the appropriateness of the mapping. The intuitive notion behind this scheme of gradation is that a bus serves to connect a subset of the set of points in the circuit. The source and destination of a transfer mapped to a bus are included within this subset. The subset should preferably be a small one, but is unknown. Inappropriate transfers mapped to a bus will introduce extraneous points into this set. The first step in the gradation process is, therefore, to estimate which of the points connected to the bus are extraneous.

The points that are connected to a bus are grouped into two sets, which are not disjoint in general. These are the points that drive the bus and the points that are driven by the bus, called *source* and *destination* sets. The *access frequency* of a point in either set is defined as the number of transfers that use this point. The point is considered to be extraneous if

$$(access\ \ frequency\ \ of\ \ point) < \begin{array}{l}(1-\alpha_t)(average\ \ access\ \ frequency)\\ +\ \alpha_t(minimum\ \ access\ \ frequency)\end{array}, \tag{8.1}$$

where α_t is a constant. If $\alpha_t \approx 1$ then the minimum access frequency receives more weight and most of the points qualify. If $\alpha \approx 0$ then the condition becomes tighter and few points qualify.

The appropriateness of a transfer is measured with respect to either the sources or destination or both, in which case the source point or the destination point or both should be should be non-extraneous.

Operation binding gradation

The binding of an operation to a f.u. has two implications. On the one hand the binding of a variable to a f.u. affects the capability of the f.u. and hence its cost. On the other hand it affects the connectivity of the f.u. to the buses and, therefore, the interconnect cost.

These two aspects may be combined together to determine the appropriateness of the binding. However, we have treated them separately. One routine has been written to consider appropriateness with respect to the f.u. capability, and another for the interconnection.

While considering f.u. capability the frequency of use of each operation type of the f.u. is computed and using a formula similar to 8.1, the appropriateness of mapping an operation on a f.u. is. computed.

The gradation with respect to connectivity for f.u.s is done using a method similar to that for buses. Instead of considering a single source, now multiple sources have to be considered. The gradation at a particular time, in general, is done with respect to a subset of the ports of the f.u., available as parameters to this routine.

Both the methods are used. The choice of a particular method is random, though not uniform, from generation to generation.

Variable binding gradation

For a particular memory the points that access it are determined. The importance of each such point is defined as the number of variables of the memory that are accessed by that point. The importance of a variable is defined as the $\sum(\textit{importance of points that access the variable})$. The *spread* of a memory is defined as the total number of points accessing the memory. The relative importance of a variable is defined as

$$\frac{(\textit{minimum spread among all memories})\sum(\textit{importance of variable})}{\alpha_v(\textit{spread})(\textit{maximum importance of variable in memory})}, \qquad (8.2)$$

where $\alpha_v \geq 1$ is a constant.

The value of the relative importance of a variable never exceeds one. This measure has been designed to reflect the homogeneity of access of the memory variables. This measure is not used every time during memory formation but is used from time to time, at random, from generation to generation.

8.5.2 Correspondence Between Data Path Elements of Parent Solutions

The representation of a solution is not unique. If the solution uses n_f f.u.'s, n_b buses and n_m memory units then the same solution can be expressed in $n_f!n_b!n_m!$ ways. While performing the crossover it is necessary to resolve this multiplicity. It is necessary to find a correspondence between the data path elements of the same class. The correspondence between the f.u.'s and the buses is found by a fast bipartite matching. The number of buses and f.u.'s are same in both the solutions thus a surjective mapping will exist between these elements. In general the number of memory units used in two distinct

1. procedure **match_vertices**
/* find a match between n_1 vertices of one set with n_2 vertices of another set */
2. for $(i = 0; i < n_2; i{+}{+})$
3.   for $(j = 0, max\_affinity = -\infty; j < n_2; j{+}{+})$
4.   { determine affinity between vertex i and vertex j
5.   if $affinity > max\_affinity$ or $(affinity == max\_affinity$ and $drand48() > p_t)$
6.   { $affinity = max\_affinity$
7. $match_i = j$
8.   }
9.   }

Figure 8.6: Algorithm used for vertex matching.

solutions will be different. The mapping between the memory units will, therefore, be only an injective mapping.

Since the mapping has to be done several times the optimal bipartite matching algorithm, whose complexity is $O(n^3)$ in the number of vertices, is not used. A simpler algorithm, shown in figure 8.6, is used instead. It will be noted, in line 5, that that ties between equally good choices are broken at random.

The affinity calculation between the elements are done as follows:

Affinity between $f.u._i$ of first solution and $f.u._j$ of second solution is computed as

$$\frac{|O_i \cap O_j|}{|O_i \cup O_j|},$$

where O_i is the set of operations of the solution mapped onto alu i. The mapping from f.u.'s of the first parent to those of the second is stored in an array *f.u._map*.

Affinity between bus_i of first solution and bus_j of second solution is computed as

$$\frac{|T_i \cap T_j|}{|T_i \cup T_j|},$$

where T_i is the set of transfers of the solution mapped onto bus i. The mapping from buses of the first parent to those of the second is stored in an array *bus_map*.

Affinity between $memory_i$ of first solution and $memory_j$ of second solution is computed as

$$\frac{|V_i \cap V_j|}{|V_i \cup V_j|},$$

where V_i is the set of variables of the solution mapped onto memory i. If both the solutions have the same number of memories then the mapping is from the first parent to the second, otherwise from the parent with smaller number of memories to the other. Mapping is stored in an array *mem_map*.

8.5.3 Inheritance Plan

A correspondence between the data path elements of the two parent solutions is first computed. The plan consists of two parts, operation inheritance plan and transfer inheritance plan. Both the plans are constructed time step by time step. In a particular time step either the operator inheritance is computed first, followed by transfer inheritance, or vice-versa. The two methods are explained below. While constructing the inheritance plan of a solution one of the two schemes is followed; the choice is made probabilistically.

Before applying these procedures in a time step the multi-cycle operations and transfers are processed. Bindings of multi-cycle operation or transfer initiated in an earlier step which spills into the current step are retained without change.

Handling operations before transfers

Under this scheme a decision is made regarding which f.u.'s will be used to guide operation inheritance from the first parent and the second parent solutions. This is indicated in a vector *asvec*; an entry of 0 indicates first parent and 1 indicates second parent. In each time step the f.u.'s are processed one by one. While processing f.u.$_i$, if $asvec[i]$ indicates the first parent, then the operation in the first solution that is mapped onto f.u.$_i$ is considered. If this operation happens to be a *core* operation of that f.u. and it has not already been mapped then it is tentatively mapped to f.u.$_i$ of the child solution.

If $asvec[i]$ indicates the second parent then a similar sequence is carried out with the second solution. The only difference is that instead of considering f.u.$_i$, f.u.$_{f.u.\_map[i]}$ is considered.

Transfer inheritance plan is carried out by inspecting the buses one by one. The first parent is processed and then the second parent is taken up. The transfer mapped on the bus is inspected. Only *core* transfers which have not already been inherited are considered for inheritance. If the transfer is associated with an operation and that operation has been inherited from the parent being considered now, then the transfer is inherited on that bus.

Handling transfers before operations

This case is handled in a similar way. An array *bsvec* indicates the parent from which the transfer of a bus should be considered for inheritance. The transfers, under this scheme, are inherited just the way operations are inherited under the previous scheme.

For operation binding the buses are scanned and the transfer, if any, that is planned to be inherited is examined, to determine the parent from which it is inherited. If it is associated with an operation then the f.u. binding of that operation is tentatively inherited from the parent from which the transfer has been inherited, provided that the operation has not already been mapped and the target f.u. has not already been used up in the current time step.

8.5.4 Memory Formation

The memories are formed one by one. Some of the variables to be placed in the memory are first inherited from the corresponding memories of the first parent and then from the second parent. Since multi-port memories are permitted, it is necessary to specify how many ports will be there in the memory being constructed. Initially, the actual number of ports is taken to be one. The tentative maximum number of ports to be used is inherited from one of the parent memories.

The decision to attempt inheritance of a variable from a memory of the parent solution is taken in one of the following two ways, both of which are probabilistic in nature. The inheritance may be governed by the *register inheritance probability* parameter, which is a fixed value. Alternatively, it is governed by the *importance* of the variable in the memory, as defined in section 8.5.1.

The actual introduction of a variable into a memory is attempted by the *mem_intro* procedure. This procedure first checks whether the candidate variable can be placed in the memory without increasing the number of ports currently required by the memory. It increases the number of ports after repeated attempts to introduce variables into the memory have already failed. The maximum number of ports that a memory can have is restricted to three. When the option to use memories with trivial port assignment only is active, then the maximum number of ports automatically is restricted to two and the variables that are to be placed in the memory are selected so that there will never be two simultaneous read and write accesses to the memory.

After inheritance is completed, in general, there will still be variables to be mapped to memories. These remaining variables are packed into the memories already constructed during inheritance. Those variables which could not be packed into these memories are packed into new memories. The choice of memories to be packed is governed by a simple heuristic. The heuristic is to choose the variable for which the number of unmapped variables that can still be packed into this memory without increasing the number of ports is maximum.

The tentative partial structure

The tentative inheritance plan implies a tentative partial structure. The plan is said to be tentative because the bindings therein are not guaranteed to be incorporated into the parent solution. While constructing the parent solution these bindings are given priority. The inheritance plan implies a partial data path. The structure of the data path is tentative because the plan itself is tentative. The implication of the data path is now explained.

It will be noted that the number of f.u.'s and buses in the data path is fixed and the memories have already been formed. Only two aspects remain, and these are f.u. formation and formation of switched and unswitched links between the ports of the f.u.'s and the memories, to the buses of the data path. As noted earlier, links between the memory and the buses can be formed at this stage only if the port assignment of the memory is trivial. For the non-trivial case the links have to be obtained through explicit port assignment.

The tentative capability of a f.u. is determined by the types of the operations tentatively mapped onto that f.u. If a transfer is planned to be mapped onto a bus then links might be tentatively formed. If the transfer involves access to a memory port or to a system port then a tentative link will be formed between the bus and that port of the memory or to that system port. If the transfer involves an operation which has been tentatively mapped onto a f.u. then a tentative link will be formed between between the bus and the port of the f.u. with which the transfer is associated.

8.5.5 Final Generation of the Offspring

At this stage the memory formation is completed. The actual operation and transfer bindings are now made. This proceeds time step by time step and consists of three phases.

Implied bindings

The first phase involves making the bindings that are implied by bindings of multi-cycle operations in earlier time step. A multi-cycle operation stays bound to the same f.u. in all its cycles; the transfers which supply its source operands remain bound to the same buses. The result of the multi-cycle operation is available in the last cycle of its operation, and its transfer to the destination is not an implied transfer.

Bindings by inheritance

The bindings of operations and transfers in the inheritance plan are now considered. First the operations are processed and then the transfers are handled. For each operation binding in the plan if the corresponding f.u. is not busy then the actual binding is recorded, and the f.u. capability is updated, if necessary. In case the binding cannot be made and the f.u. is currently not equipped to implement that type of operation, then the possibility of having that operation type on that f.u. is reduced. It may be noted that in the tentative data path the capability of each f.u. was built up on the basis of the operation bindings in the inheritance plan.

Similarly, transfer bindings are inherited but with some additional processing. While making a transfer binding if the existing links between and f.u.'s, system ports and the memories with the buses suffice to support the transfer then the binding is directly made. If new links need to be introduced at both the source and the destination of the transfer then the inheritance is not made. If only one new link is needed then the inheritance is done probabilistically. Whenever a new link is introduced the data path is updated.

Completion of the pending bindings

In general some operations and transfers will still remain unmapped. These bindings are made using a force directed completion algorithm. First the operations and then the transfers are bound. The decisions are made in a best first approach, selecting the binding that leads to the least force. The completion algorithm is explained in the next section.

8.5.6 The Completion Algorithm

This is based on a force directed approach which has been pioneered in HAL [15]. A force directed approach has also been used in SAM [27]. We use a completely new set of forces which are now explained. This algorithm uses the partial structure to guide the actual bindings in the new solution. In this process the actual data path structure is found by augmenting and appropriately modifying the initial partial structure. The following definitions will be useful in explaining the algorithm.

XCOST Cost of a multiplexer switch.

LKY A constant used with likely cases.

ULK A constant used with unlikely cases.

DEF A constant used with a case that has occurred.

Total unmapped operations (U_o) Number of operations in the current time step which are yet unmapped.

Total unmapped operations of type y (U_o^y) Number of unmapped operations of type y.

Number of available f.u.'s (A_f) No operation has yet been mapped onto these f.u.'s.

Cost of operation type y ($C[y]$) The cost of a f.u. will increase by this amount to be able to implement an operation of type y.

Distribution graph of operations ($D_o[u, y]$) Distribution graph of operations of type y on f.u. u.

$$D_o[u,y] = \begin{cases} \text{if operation type } y \textit{ is present in f.u. } u \\ \text{then} \quad -\dfrac{U_o^y}{A_f}.C[y] \\ \text{else} \quad \begin{cases} \text{if likely to be present} \\ \text{then} \quad \dfrac{U_o^y}{A_f}.C[y].\text{LKY} \\ \text{else} \quad \dfrac{U_o^y}{A_f}.C[y].\text{ULK} \end{cases} \end{cases} .$$

Total unmapped transfers (U_t) Number of transfers in the current time step which are yet unmapped.

Total unmapped transfers associated with port p of f.u.'s (U_t^p) Number of transfers in the current time step associated with port p of f.u.'s which are yet unmapped.

Total unmapped transfers associated with memory m (U_t^m) Number of transfers in the current time step associated with memory m which are yet unmapped.

Number of available buses (A_b) No transfer has yet been mapped onto these buses.

Load-1 (L_1) Switch load when only operation is not f.u. mapped. $L_1 = \dfrac{\text{XCOST}}{A_f}$.

Load-2 (L_2^p) Switch load when neither operation is not f.u. mapped nor transfer is bus mapped. $L_2 = \dfrac{\text{XCOST}.U_t^p}{A_f.A_b}$.

Load-3 (L_3) Switch load when only operation is f.u. mapped but transfer is not bus mapped. $L_3 = \dfrac{\text{XCOST}}{A_b}$.

Forces due to operation mapping

While considering an operation (in column r of the specification) of type y to map on f.u. u two types of forces are computed. These are the self force and other forces. These force are exerted on f.u. u and other f.u.'s $l, l \neq u$.

The forces on f.u. u are as follows:

$$\text{self force :} \qquad F_{os}^{uy} = \frac{A_f - 1}{A_f}.D_o[u, y].$$

The force due to other operations of type y_o is as follows:

$$\text{other force :} \qquad F_{ox}^{uy_o} = \begin{cases} \text{if } y \neq y_o & -\dfrac{U_o^{y_o}}{A_f}.D_o[u, y_o] \\ \text{else} & -\dfrac{U_o^y - 1}{A_f}.D_o[u, y] \end{cases}.$$

The forces on f.u. $l, l \neq u$ are as follows:

$$\text{self force :} \qquad F_{os}^{ly} = \left(\frac{-1}{A_f} + \frac{U_o^y}{A_f(A_f - 1)}\right).D_o[l, y],$$

$$\text{others :} \qquad F_{ox}^{ly_o} = \frac{U_o^{y_o}}{A_f(A_f - 1)}.D_o[l, y_o].$$

The total force due to operations when an operation of type y is considered for mapping onto f.u. u is:

$$F_o^{uy} = \sum_z \sum_w (F_{os}^{zw} + F_{ox}^{zw}),$$

computed over all the available f.u.'s.

Lookahead forces between f.u.'s and buses due to transfer distribution perturbation

The lookahead is to estimate the effect on interconnect when the operation in column r of the specification is mapped onto f.u. u. The transfers associated with the ports of the operation are considered for mapping on the available buses. As usual the self force as well as the other forces are computed. Thus the lookahead is on f.u. port p and bus s, while trying to map operation of column r in f.u. u, and considering the effect on f.u. l and bus b. This involves the computation of force on the link between bus b and the port p of f.u. l. The computation of these lookahead forces is now explained. First a number of values are computed using which the forces are found. The values are determined using the algorithm mentioned below.

```
procedure fu_lookahead()
{       if (the link is surely present)
        then ff = DEF
        else  if (likely)
              then ff = LKY
                   ff = ULK

        if (bus b is not available)
        then dg = ff ⋆ L1
        else  if f.u. l is available
              then dg = ff ⋆ L2
              else  dg = ff ⋆ L3

        if (the transfer of operation on column r on port p is bus mapped)
        /* In this case bus s will be only this bus */
        {     if (bus l is unavailable or b ≠ s)
              {     del_prob_s = 0
              }     else
              {     del_prob_s = (l == u ? ds1_prob1 : ds0_prob1)
                    if (l == u)
                    then sf = (ff == DEF ? -c1 : (ff == LKY ? -c2 : c3))
                    else  sf = (ff == DEF || ff == LKY ? c4 : c5)
              }

              if (bus b is unavailable)
              {     if (b == s)
                    then del_prob_ow = 0
                    else  del_prob_ow = (l == u ? dow1_prob : dow0_prob)
              }     else /* bus b is unavailable */
              {     if (f.u. l is available)
                    then del_prob_ow = (l == u || b == s ? dow1_prob1_1 : dow0_prob1_1)
                    else  del_prob_ow = 0
```

```
                }
        }   else /* the transfer is unmapped */
        {   if (f.u. l is unavailable or bus b is unavailable)
            {   del_prob_s = 0
            }   else
            {   del_prob_s = (l == u && b == s ? ds1_prob2 : ds0_prob2)
                if (l == u && b == s)
                {   sf = (ff == 1 ? -c6 : (ff == LKY ? -c7 : c8))
                }   else
                if (l == u || b == s)
                then sf = ((ff == DEF || ff == LKY) ? c9 : −c10
                else  sf = 1
            }
            if (bus b is unavailable)
            {   if (l == u || b == s)
                { del_prob_ow = dow1_prob
                }   else
                if (f.u. l is available)
                then del_prob_ow = dow0_prob
                else  del_prob_ow = 0
            }   else /* bus b is available */
            if (f.u. l is available)
            {   del_prob_ow = (l == u || b == s) ? dow1_prob2_1 : dow0_prob2_1)
            }   else
                del_prob_ow = (l == u || b == s) ? dow1_prob2_2 : dow0_prob2_2)
        }

        if (l == u && b == s)
        {   of = (ff == DEF ? -c11 : (ff == LKY ? -c12 : c13))
        }   else {   if (ff == DEF)
            then of = 1
            else  if (ff == LKY)
                then of = c14
                else  of = c15
        }   else
            of = c16
}
```

This force component is computed as

$$F_{fp}^{lb} = dg \star (del\_prob\_s \star sf + del\_prob\_ow \star of).$$

The total lookahead force from f.u. ports to bus links is computed as

$$F_f^{us} = \sum_z \sum_w \sum_u F_{fu}^{zw}.$$

Lookahead forces between memories and buses due to transfer distribution perturbation

Following the earlier convention, the transfer of port p of the operation in column r is considered for mapping on bus s. It is assumed that the transfer involves memory m. The forces developed at the links between bus b and memory z are computed. The value ff is defined as before.

$$\text{self force :} \qquad F_{vs}^{bz} = \begin{cases} \text{if} \quad b == s & \begin{cases} \text{if} \quad m == z: & \mathit{ff}.\dfrac{A_b - 1}{A_b}.\dfrac{U_t^m}{A_b} \\ \text{else :} & 0 \end{cases} \\ \text{else} & \begin{cases} \text{if} \quad m == z: & \mathit{ff}.\dfrac{-1}{A_b}.\dfrac{U_t^m}{A_b} \\ \text{else :} & 0 \end{cases} \end{cases} .$$

$$\text{other force :} \qquad F_{vs}^{bz} = \begin{cases} \text{if} \quad b == s & \begin{cases} \text{if} \quad m == z: & \mathit{ff}.\max(0, U_t^m - 1).\dfrac{-1}{A_b}.\dfrac{U_t^m}{A_b} \\ \text{else :} & \mathit{ff}.U_t^z.\dfrac{-1}{A_b}.\dfrac{U_t^z}{A_b} \end{cases} \\ \text{else} & \begin{cases} \text{if} \quad m == z: & \mathit{ff}.\max(0, U_t^m - 1).\dfrac{-1}{A_b(A_b - 1)}.\dfrac{U_t^m}{A_b} \\ \text{else :} & \mathit{ff}.U_t^z.\dfrac{-1}{A_b(A_b - 1)}.\dfrac{U_t^z}{A_b} \end{cases} \end{cases} .$$

Forces for transfer to bus mapping

Force on bus b due to transfer f when transfer r is mapped onto bus s. The value ff is as defined earlier.

$$
\begin{array}{l}
\text{if } (s == b) \\
\{ \quad \text{if } (r == f) \text{ /* self force */} \\
\qquad \{ \quad \mathit{delprob} = \dfrac{A_b - 1}{A_b} \\
\qquad\qquad \mathit{cf} = (\mathit{ff} == \mathit{DEF}? - c_{17} : c_{18}) \\
\qquad \} \quad \text{else /* other */} \\
\qquad \{ \quad \mathit{delprob} = \dfrac{-1}{A_b} \\
\qquad\qquad \mathit{cf} = \dfrac{(\mathit{ff} == \mathit{DEF}? - c_{19} : (\mathit{ff} == \mathit{LKY}? - c_{20} : c_{21}))}{U_t - 1} \\
\qquad \} \\
\} \quad \text{else} \\
\text{if } (r == f) \text{ /* self */} \\
\{ \quad \mathit{delprob} = \dfrac{-1}{A_b} \\
\qquad \mathit{cf} = (\mathit{ff} == \mathit{DEF} || \mathit{ff} == \mathit{LKY}? - c_{22} : c_{23}) \\
\} \quad \text{else} \\
\{ \quad \mathit{delprob} = \dfrac{1}{A_b(A_b - 1)}
\end{array}
$$

$$cf = (ff = DEF? - c_{24} : (ff == LKY? - c_{25} : c_{26}))$$
}

This component of the force is computed as

$$F_{bf}^{rs} = \frac{XCOST}{A_b} \star delprob \star cf.$$

The overall force of mapping the transfer f to bus b is

$$F_b^f = \sum_r \sum_s F_{bf}^{rs}.$$

8.5.7 Operation Commutation

We now explain the technique of generating the optimal commutation of commutative operations mapped onto a particular f.u. In general, a f.u. will house commutative as well as non-commutative operations. We consider only dyadic commutative operations. A graph theoretic formulation has been developed.

Consider a f.u. which houses a set of commutative as well as non-commutative (unary and binary) operations. Each operation will take one or two inputs which will originate from points in the system. These points are already fixed as a result of the binding step. We construct a conflict graph whose vertices correspond to the points that supply operands to the operations mapped onto the f.u. There will be an edge between two vertices if and only if the points supply operands to the same operation. The aim is to colour this graph using two colours. The two colours correspond to the f.u. ports. In general this graph will not be two colourable and it will be necessary to delete some of the vertices. The deleted vertices will have to be connected to both the ports. This is similar to the formulation for the dual port memory PA problem.

However, there is one important difference between this problem and the dual port memory PA problem. Since some of the points source operands to non-commutative operations mapped onto the f.u., if any, these points will necessarily have to be connected to a particular port of the f.u. In the graph, therefore, these vertices will be pre-coloured. Two colouring and vertex deletion will now have to be done on the graph subject to the pre-colouring of these vertices. We do not present the detailed GA for this problem but only mention that the GA for dual port memory PA, with minor modifications can be used to solve this problem. We illustrate the above formulation through example 8.7.

Example 8.7 Consider the following operations mapped onto a particular f.u. in various time steps.

1. $t_1 = x_1 + x_2$
2. $t_2 = x_1 - x_3$
3. $t_3 = x_2 \star x_4$

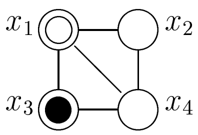


Figure 8.7: Conflict graph for operation commutation with pre-coloured vertices.

4. $t_4 = x_1 + x_4$

5. $t_5 = x_3 + x_4$

The conflict graph for the commutation selection problem in shown in figure 8.7. The filled inner circle indicates that the vertex x_3 has been pre-coloured to 1. The hollow inner circle indicates that the vertex x_1 has been pre-coloured to 0. It is evident that if x_1 is deleted then x_2 and x_4 can be assigned colour 1 and 0, respectively. □

8.6 Experimental Results

We now present some experimental results for the GA based allocation and binding tool *GABIND* described in this chapter. We report the results of our experimentation on Facet example, differential equation solver and elliptic wave filter. The results have been tabulated along with results of some other well known systems available with us. The schedules used for all these examples are available in appendix B. While running these examples we had to supply costs of the hardware elements. We have not used actual costs from design libraries. The costs that we have used have been chosen to reflect the relative sizes of the data path elements. For example the cost of a multiplexing switch is less than that of an adder, which in turn is considerably less than that of a multiplier. The same cost combination has been used for all examples; only the multiplexing switch in some cases is taken as five units as against the normal value of eight units. These cases are explicitly indicated in the tabulated results.

In the tables the result of our system have been labeled *GABIND*. The first column of each table indicates the name of the system. The second column gives the number of multiplexing switches required and the third column the number of links. In general our system generates data paths using memory elements. The fourth column is titled *# cell*. For other systems (which do not use memories) this represents the number of registers used in their designs, for our system it represents the number of distinct storage slots (memory cells and registers taken together), "normalized" for comparison with other systems. This "normalization" is done because constants and some some other storage elements are usually not included while counting the register usage by most systems. We also give a summary of the memory configuration in the column titled *memory config.* A memory configuration of the form $< x, y, z >$, indicates a total of z cells in y memories

System	# mux.	# link	# cell	memory config.	CPU time
Facet	11	—	8	—	—
Splicer	8	—	7	—	3s
HAL	6	13	5	—	—
Vital-NS	6	12	5	—	1.5s
GABIND(1) [1]	8	10	5	$< 2,3,3 >$ $< 1,1,2 >$	25s
GABIND(2) [1]	7	11	6	$< 2,3,4 >$ $< 1,2,2 >$	27s
GABIND	5	11	6	$< 2,3,4 >$ $< 1,2,2 >$	28s

Table 8.1: Results for Facet example for 4 time steps and using 3 f.u.'s.

each having x ports. For example $< 2, 3, 7 >$, represents three memories of two ports housing a total of seven cells. We now present the results.

Facet Example

Table 8.1 presents the results for the Facet example, synthesized using three f.u.'s. We have reported three results obtained by our GA, namely, GABIND(1), GABIND(2) and GABIND. The first two are solutions in the population at termination while running the Facet example with a multiplexer cost of five units. The last result has been obtained by running the Facet example with a multiplexer cost of eight units. The data path for the Facet example indicates as GABIND is shown in figure 8.8. The f.u.'s for GABIND(1) are: $\langle + - \& \rangle$, $\langle + | \star \rangle$ and $\langle / \rangle$. The f.u.'s for GABIND(2) are: $\langle + | \star \rangle$, $\langle + - \rangle$ and $\langle \& / \rangle$. The f.u.'s for GABIND are: $\langle + \rangle$, $\langle + | \star \rangle$ and $\langle - \& / \rangle$. For GABIND(1) the number of memory cells required is minimum but the switching requirement turns out to be high, requiring eight switches. GABIND, on the other hand requires five switches and six memory cells. The number of links for all the three results vary between ten and eleven, which is a little less than what is reported for other systems. The variable mergings for this example have been done on the basis of the life time of the variables indicated in [17].

Differential Equation Solver

This is the next example that we have considered. In this example we also illustrate the use of design parameters. We report two sets of results for non-pipelined multipliers and one set of results for pipelined multipliers. They are tabulated in tables 8.2 and 8.3. The schedules used by GABIND for each of these cases is given in section B.2. For the first two cases the schedule is for four time steps. For the first set of results in table

[1]Multiplexing switch cost is 5 units.

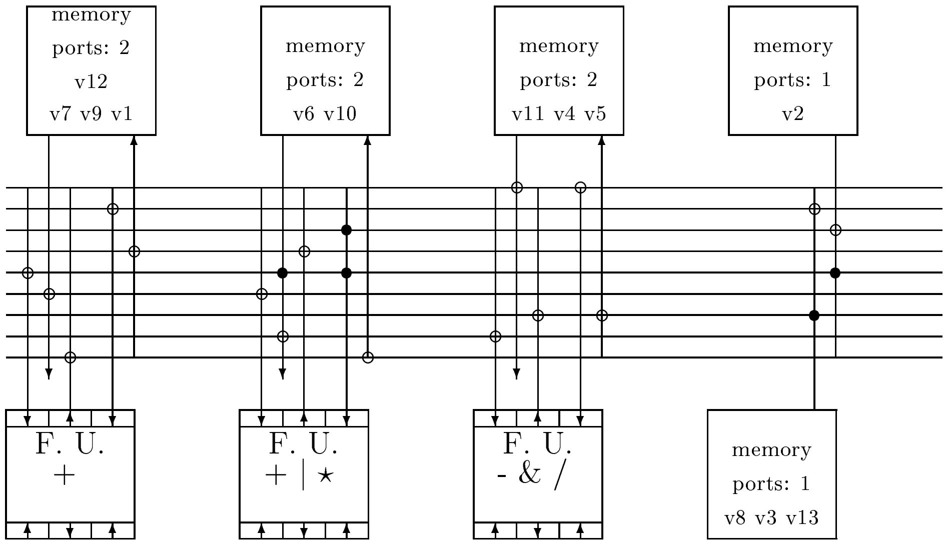


Figure 8.8: Allocation and binding for Facet.

8.2 the number of f.u.'s has been set to five for GABIND, because other systems have also employed five distinct f.u.'s (following our convention of counting). The f.u.'s used by GABIND are: $\langle\star\rangle$, $\langle\star\rangle$, $\langle+\rangle$, $\langle-\rangle$, $\langle<\rangle$. For the second set this parameter (NFUS) is set to three, because in the schedule at most three operations are executed in any time step. The f.u.'s used by GABIND are: $\langle+- <\rangle$, $\langle+\star\rangle$ and $\langle\star\rangle$. With five f.u.'s our design requires only eight f.u.'s eighteen links and five memory cells which compare very favourably with the other results for other designs tabulated in table 8.2. For synthesis with three f.u.'s corresponding results are not available from other systems. However, the results we get now are still competitive to the results of other systems for design with five f.u.'s in terms of multiplexing switches and storage cells. Results for the differential equation example with pipelined multipliers have been tabulated in table 8.3. For this case the f.u.'s used by GABIND are: $\langle\star\rangle$ and $\langle+- <\rangle$. Figure 8.2 mentioned earlier in this chapter is actually the data path for the pipelined multiplier version of Diffeq. Here too our design using only seven switches compare very well with results form other systems.

Elliptic Wave Filter

This is the last example we report for GABIND. We have performed the synthesis using pipelined multipliers for seventeen, eighteen and nineteen time steps. The number of f.u.'s required for the three cases are three, four and three, respectively. For seventeen time steps the f.u.'s used by GABIND are $\langle+\rangle$, $\langle+\rangle$, and $\langle\star\rangle$. For eighteen time steps the f.u.'s used by GABIND are $\langle+\rangle$, $\langle+\rangle$, $\langle+\rangle$, and $\langle\star\rangle$. For nineteen time steps the f.u.'s

[1]Multiplexing switch cost is 5 units.

Using single cycle multipliers and 5 f.u.'s					
System	# mux.	# link	# cell	memory config.	CPU time
Splicer	11	—	6	—	—
HAL	10	25	5	—	40s
Vital-NS	12	22	5	—	3s
GABIND	8	18	5	$< 2, 5, 7 >$ $< 1, 1, 1 >$	38s
Using single cycle multipliers and 3 f.u.'s					
GABIND [1]	12	16	6	$< 2, 4, 5 >$ $< 1, 2, 4 >$	32s

Table 8.2: Results for Diffeq. example for 4 time steps with an operation distribution: $\langle 2\star, +, -, < \rangle$.

System	# mux.	# link	# cell	memory config.	CPU time
HAL	13	19	5	—	120s
Vital-NS	13	17	5	—	2.5s
GABIND	7	13	5	$< 2, 2, 5 >$ $< 1, 2, 1 >$	24s

Table 8.3: Results for Diffeq. example for 8 time steps with two f.u.'s: $\langle 1$ (pipelined) $\star\rangle$, $\langle +- <\rangle$.

System	# mux.	# link	# cell	memory config.	f.u. config.	CPU time
HAL	31	—	12	—	3+, 2$\star$	120s
SAM	31	50	12	—	3+, 2$\star$	—
Vital-NS	32	50	11	—	3+, 2$\star$	110s
STAR	26	—	11	—	2+, 1$\star$	—
GABIND	29	29	13	$< 2, 5, 13 >$ $< 1, 1, 1 >$	2+, 1$\star$	210s

Table 8.4: Results for elliptic wave filter example for 17 time steps, with pipelined multipliers, each hardware operator in a different f.u.

System	# mux.	# link	# cell	memory config.	f.u. config.	CPU time
HAL	34	—	12	—	3+, 1$\star$	120s
SAM	30	40	12	—	3+, 1$\star$	—
Vital-NS	33	40	10	—	3+, 1$\star$	140s
STAR	—	—	—	—	—	—
GABIND	31	35	11	$< 2, 6, 11 >$ $< 1, 1, 1 >$	3+, 1$\star$	251s

Table 8.5: Results for elliptic wave filter example for 18 time steps, with pipelined multipliers, each hardware operator in a different f.u.

used by GABIND are $\langle + \rangle$, $\langle + \rangle$, and $\langle \star \rangle$. The results for these three cases have been tabulated in tables 8.4, 8.5 and 8.6, respectively. The data paths for the elliptic wave filter example for seventeen time steps is given in figure 8.9. In each case our results are better than some, but not all, of the results that have been reported. For nineteen time steps the result reported for SAM is exceptionally good. However, the memory formation for SAM is done manually, as reported in the literature [27]. Results of HAL are seen to be competitive for this example. We feel this is the result of a special storage formation technique incorporated in their system.

In general we observe that GABIND takes some more CPU time than other methods. This is essentially because the GA has to run for several generations and produce a number of new solutions in each generations. However, unlike other methods a number of distinct solutions having each having the best cost are usually obtained on termination of the GA. Other methods typically terminate with a single solution. It may also be noted that the relative increase in CPU time for GABIND with problem size is less than what it is for the other tabulated methods.

System	# mux.	# link	# cell	memory config.	f.u. config.	CPU time
HAL	26	—	12	—	2+, 1⋆	120s
SAM	21	40	12	—	2+, 1⋆	—
Vital-NS	29	40	11	—	2+, 1⋆	200s
STAR	28	—	11	—	2+, 1⋆	—
GABIND	27	33	14	$< 2, 4, 12 >$ $< 1, 2, 3 >$	2+, 1⋆	255s

Table 8.6: Results for elliptic wave filter example for 19 time steps, with pipelined multipliers, each hardware operator in a different f.u.

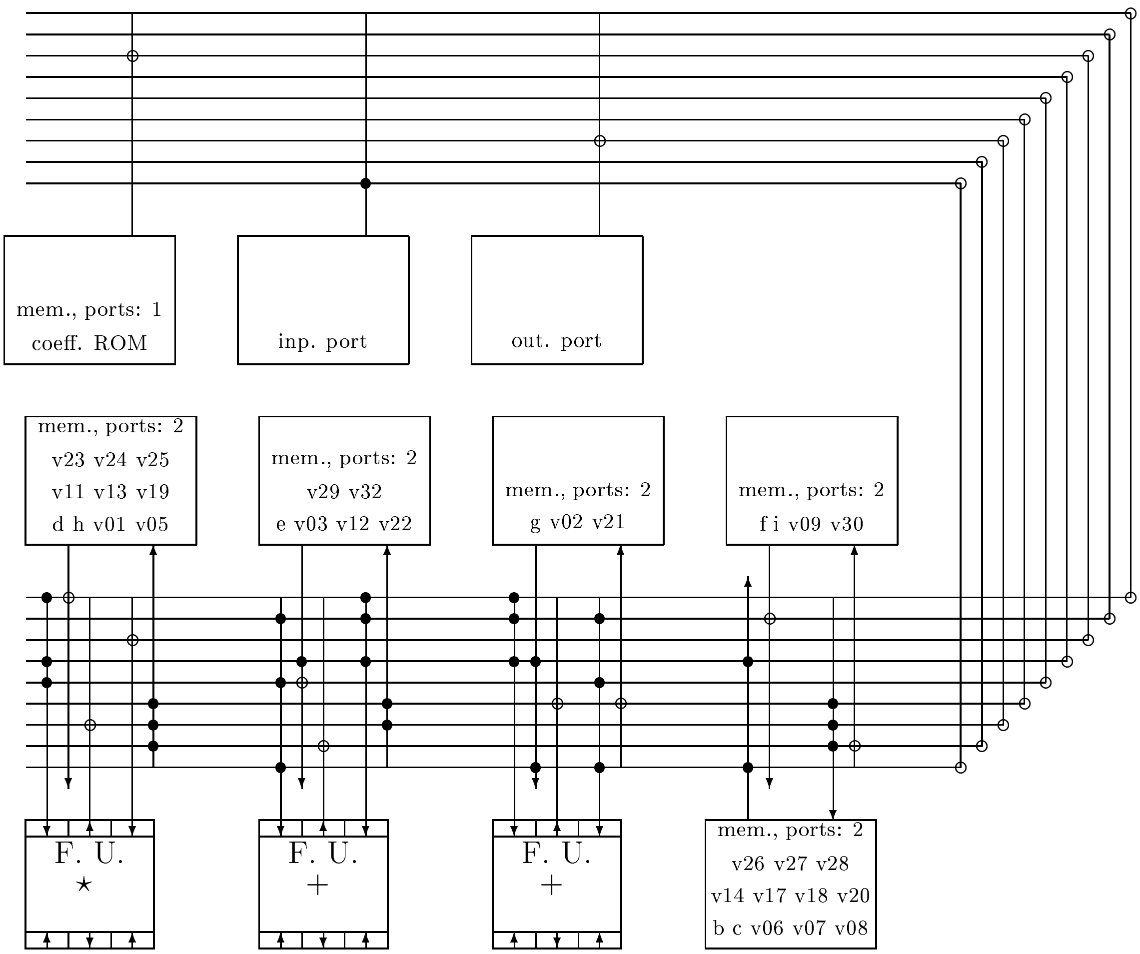


Figure 8.9: Allocation and binding for elliptic wave filter.

8.7 Conclusions

In this chapter we have considered the problem of constructing data paths from a given schedule of operations. For this problem we have proposed and implemented a GA based solution, which we refer to as GABIND. Our solution has several features. It is based on an integrated formulation to optimize the total cost of all the hardware elements that go into the construction of the data path, viz. functional units, multiplexing switches and storage elements. For implementing storage our tool supports the use of single and multi-port memories which are increasingly being used as on chip storage devices. The use of memories permits the encapsulation of several variables (which would normally not be mergable into a single register) within the boundary of a single component. This helps to reduce the total number of distinct components in the data path. As a result the data path is more compact and has less number of control points. This helps to reduce the size of the controller synthesis problem. We also support the use of a bus based interconnection structure which is beneficial when there are relatively large number of components in the data path between which data transfers take place. We have identified and used the presence of a *common source* of multiple data transfers for interconnect optimization.

Our allocation and binding tool GABIND is based on the genetic paradigm. For this GA we have developed a novel crossover based on a force directed completion algorithm. The GA has several advantages like implicit parallelism, ability to incorporate multiple heuristics and generating a population of solutions on termination. In this population there are often a number of distinct solutions having the best cost. Therefore, we not only benefit from the optimization but also get a set of data paths (having the same cost) for implementing the schedule. These data paths may be separately evaluated for subsequent design steps, like physical design and testing. The advantage of obtaining a multitude of solutions largely offsets the slightly longer CPU time requirement of the GA technique.

It is evident from the experimental results that GABIND comparable well with other reported systems. The present implementation of our system has a tendency to use a slightly larger number of cells for storage. For Facet and Diffeq. examples our system produces a data path with less number of multiplexer switches consistently. The number of links used is consistently low. This is mainly due to the use of memories which results in less number of distinct components. The results obtained on the standard examples have been promising.

Chapter 9

Conclusions

We have made a study of both theoretical and practical aspects of data path synthesis problem In this work we have examined the complexity of several synthesis sub-problems. We have then proposed solutions to the entire data path synthesis problem and to some interconnect optimization oriented sub-problems.

9.1 Contributions of Present Work

We have examined *scheduling with time constraints* and *resource constraints*. Both for scheduling and allocation we have considered a few new problems also. One such problem is the *scheduling of variable transfers to take place under the constraints imposed by the available hardware*. We call it the variable assignment problem. For allocation and binding we have considered the complexity of the *port assignment problem for dual and triple port memories*. We have examined the complexity of the general allocation and binding problem with special emphasis on *interconnect optimization*. In particular we have examined the complexity of *interconnect optimization for straight line code*. This problem is specially interesting because the register optimization problem has an efficient polynomial time solution. We have also examined the complexity of constructing *functional units of minimum cost*.

Summary of Scheduling Complexity Results

We have considered the problem of scheduling chains having two types of operations, on two functional units (one for each type of operation). We also consider the special case of scheduling only two chains.

- The problem of scheduling a set of chains having only two types of operations (unit execution times), on two f.u.'s (one for each type of operation), given a deadline D is NP-complete.

- The problem of finding a minimum length schedule of two chains of two types of operations using two f.u.'s, one for each type, is solvable in polynomial time.

We have obtained the following result for resource constrained scheduling.

- Scheduling m chains having only one type of operation with two f.u.'s, unit execution times, and one resource with limit 1 is NP-complete.

We also analyze the complexity of approximations of scheduling problems.

- Absolute approximation of scheduling DAG's is NP-hard for the problem of minimization of schedule length.
- Absolute approximation of scheduling DAG's with multiple operation types, given a deadline D, is NP-hard for the problem of minimization of the number of f.u.'s (where each f.u. implements only one type of operation).

The variable assignment problem is a new problem that we have considered, and for it we have the following two results.

- The problem of scheduling variable assignments in minimum number of time steps, subject to the availability of a fixed number of points from where these variables can be accessed is NP-hard.
- The absolute approximation of scheduling variable assignments in minimum number of time steps, subject to the availability of a fixed number of points from where these variables can be accessed is NP-hard.

Summary of Allocation and Binding Complexity Results

The first set of results that we have derived are on the complexity of the port assignment (PA) of dual and triple port memories, which are as follows.

- Port assignment for dual port memories is NP-complete.
- Port assignment for triple port memories is NP-complete.
- Relative approximation of port assignment for triple port memories is NP-complete.

The following two results concern the complexity of interconnect optimization.

- Register-interconnect optimization for straight-line code (SRIO) is NP-hard.
- Relative approximation of register-interconnect optimization for straight-line code is NP-hard.
- Relative approximation of general register-interconnect optimization is NP-hard.

For the problem of forming functional units at minimal cost we have obtained the following result.

- The problem of determining the assignment of operations to a fixed number (exceeding one) of functional units so as to minimize their cost is NP-hard.

Considering the complexity of approximations of the scheduling problem and the allocation and binding problem, we feel that the latter is a more difficult problem. Thus, while proposing solution to the entire data path synthesis problem we have used heuristic methods along with a controlled search scheme for design space exploration which is a scheduling type problem. For allocation and binding, on the other hand, we have made use of the genetic algorithm (GA).

Summary of Work Done for RIO and MIO

The RIO problem (as we have considered it) is possibly the simplest practical optimization problem in the category of interconnect optimization problems, which have shown to be hard in chapter 4. For this problem we have proposed a heuristic algorithm for which we have obtained satisfactory results. We have also considered the little more general problem of memory-interconnect optimization. For this problem we have developed an interesting formulation of the port assignment problem in terms of RIO. We have noted that results for MIO are not nearly as encouraging as that of RIO. This is expected, considering the increase in complexity of the problem and the limitation of heuristic methods. For the larger allocation and binding problem we have, therefore, resorted to a method (chapter 8) which is, in principle, more powerful than a purely heuristic technique.

Summary of Work Done for Port Assignment

An efficient genetic algorithm has been developed to solve the dual port memory PA based on a graph theoretic formulation. An estimator based on random graphs has been developed to to estimate the cost of port assignment without having to find the exact solution to the instance of the dual port memory PA. problem. A GA has also been developed to solve the triple port memory PA based on a hyper-graph formulation. The graph theoretic formulations are compact and sometimes helpful in developing heuristics for obtaining a solution. We have also developed an estimator for estimating the cost of PA for a triple port memory.

The estimator for the dual port memory serves as a valuable tool to evaluate numerous packings of variables into a dual port memory. The use of the estimator would be especially enhanced when GA2, is used to solve the dual port memory PA because the estimator closely estimates the number of nodes actually deleted by GA2. We have developed a GA for general purpose port assignment to handle situations like multi-cycling and non-uniform ports. However, GA2 and GA3 developed for the graph based compact formulations usually perform better than the GA for general purpose port assignment, for problem instances where these are applicable.

Summary of Work Done for Design Space Exploration and Scheduling

We have proposed a scheme for doing design space exploration using a combination of controlled search, approximate and genetic scheduling techniques. The search is based on depth first branch and bound (DFBB). DFBB has the advantage of requiring minimum space in the host machine where it has to run. It is necessary to conserve space because the storage for a single (partial) solution, itself, is considerable, We have used a balanced problem decomposition scheme for the DFBB. This has the advantage of partitioning the original problem into smaller subproblems of nearly equal sizes. We have demonstrated through experimentation that a problem instance of reasonable size typically has a number of distinct design points. Moreover, these design points are arbitrarily distributed in the design space. This makes the concept of design space exploration all the more relevant. We have also applied the genetic paradigm to the scheduling problem to develop the genetic list scheduling algorithm (GLS). GLS has been demonstrated to compare very favourably with some existing scheduling techniques. It also handles some other practical of scheduling for DPS, like variable assignments, to satisfy the design parameters. We have applied our DSE techniques to some common examples like Facet, differential equation solver and elliptic wave filter and constructed data paths from the schedules obtained. We have noted close conformity with the estimates obtained with DSE and the actual hardware used in the data paths. We have thus proposed and implemented a solution to the schedule time design space exploration problem for DPS.

Summary of Work Done for Allocation and Binding

For this problem we have proposed and implemented a GA based solution, which we refer to as GABIND. Our solution has several features. It is based on an integrated formulation to optimize the total cost of all the hardware elements that go into the construction of the data path, viz. functional units, multiplexing switches and storage elements. For implementing storage our tool supports the use of single and multi-port memories which are increasingly being used as on chip storage devices. The use of memories permits the encapsulation of several variables (which would normally not be mergable into a single register) within the boundary of a single component. This helps to reduce the total number of distinct components in the data path. As a result the data path is more compact and has less number of control points. This helps to reduce the size of the controller synthesis problem. We also support the use of a bus based interconnection structure. We have identified and used the presence of a *common source* of multiple data transfers for interconnect optimization.

We have developed a novel crossover based on a force directed completion algorithm for use in GABIND. The GA has several advantages like implicit parallelism, ability to incorporate multiple heuristics and generating a population of solutions on termination. These data paths may be separately evaluated for subsequent design steps, like physical design and testing.

9.2 Tools Developed

We present below a list of the software tools that have been developed as part of this work.

Reg_min (Register-interconnect optimizer.)

- **Input** Variable compatibility information, sources and destination of each variable.
- **Output** Grouping of variables into individual registers and optimized data path.
- **Description** This tool finds a mapping of variables to registers which minimizes the sum of the cost of the cost of the registers and the final cost of multiplexing switches in the target design.

Mem_alloc (Memory-interconnect optimizer.)

- **Input** Schedule of operations with details of the input and output operands of the operations.
- **Output** Grouping of variables into memories and registers, port assignment of each memory used, the variables to be mapped in each cell of each memory and optimized data path.
- **Description** This tool finds a grouping of variables into single or multi-port memories and registers. For each multi-port memory it computes the port assignment using an RIO type formulation. The grouping is done so that the final cost of the memories, registers and multiplexing switches is minimized.

GA2 (GA for graph theoretic formulation of dual port memory PA.)

- **Input** A graph representing the access conflicts between points that access a dual port memory.
- **Output** A list of vertices representing the points which should be connected to both the ports.
- **Description** This tool finds the minimum set of points accessing a dual port memory which must be connected to both its ports so that the access to the memory can be satisfied and at the same time the number of switches for interconnection is also reduced. It is implemented as a genetic algorithm.

GA3 (GA for hyper-graph based formulation of dual port memory PA.)

- **Input** A hyper-graph representing the access conflicts between points that access a triple port memory.
- **Output** For each point the ports to which it should be connected.
- **Description** This tool finds the set of ports to which the points accessing a triple port memory must be connected so that the access to the memory can be satisfied and at the same time the number of switches for interconnection is also reduced. It is implemented as a genetic algorithm.

GPA (GA for general dual and triple port memory PA.)

> **Input** The sequences of accesses to a dual or triple port memory.
>
> **Output** For each point the ports to which it should be connected.
>
> **Description** This tool performs port assignment for dual and triple port memories based on an exact formulation. Multi-cycle accesses are permitted. It is implemented as a genetic algorithm.

DSE (Tool for design space exploration.)

> **Input** A flow graph of basic blocks and design parameters.
>
> **Output** A set of competitive schedules and the corresponding requirement of hardware operators.
>
> **Description** This tool performs design space exploration for a given behavioural specification to produce a set of competitive schedules. For each schedule it also generates a resource estimate. A controlled search scheme is used to generate the schedule and obtain the resource estimate. For large problems the algorithm can be run to generate approximate resource estimates and partial schedules. When approximate estimates are generated, a heuristic technique is used to generate feasible non-dominated schedules and resource estimates.

GLS (Genetic list scheduling.)

> **Input** A partial order of operations or a partial schedule and a list of the availability of hardware operators.
>
> **Output** A feasible schedule.
>
> **Description** This tool generates a feasible schedule of minimum length using the given hardware operators. If a partial schedule is available, then it makes use of that information to guide its scheduling decisions. It is implemented as a genetic algorithm.

GABIND (Genetic allocation and binding.)

> **Input** A schedule of operations with details of the input and output operands of each operations and design parameters.
>
> **Output** An optimized RTL data path.
>
> **Description** This tool finds an optimized RTL data path structure for the given schedule and design parameters. It supports the use of multi-port memories as data path elements and uses a bus based interconnection style. It is implemented as a genetic algorithm.

Some algorithms proposed by other groups had been locally implemented for our use. These are: i) force directed list scheduling algorithm (FDLS) [41] and ii) lower bound based scheduling algorithm (LBBS) [23].

9.3 Future Work

Several complexity results for the scheduling problem and some complexity results for other problems already exist. We have proved that several scheduling and allocation problems and some of their approximations are NP-hard. The complexity of some DPS sub-problems still remain to be proved. Some of these are as follows:

- Approximation of the dual port memory port assignment problem.
- Scheduling DAG's with unit execution time operations on a fixed number n ($n > 2$) of f.u.'s to minimize schedule length.
- Complexity of scheduling a fixed number n ($n > 2$) of chains of two operation types on two processors one of each type.

The approximation of allocation and binding problems have been shown to be NP-hard. It is, therefore, unlikely that polynomial time approximations for these problems will be possible. We have successfully incorporated the use of a probabilistically good algorithm for the dual port memory PA based on the graph theoretic formulation.

- Probabilistically good approximate scheduling algorithms need to be developed for incorporation into crossover of new GA's for DPS scheduling.

We have developed a probabilistic estimator for dual and triple port memory with uniform read/write ports. During operation there are approximately two read accesses for every write access in the data path. Our estimator is directly applicable for estimating the cost of PA for triple port memories with a single write port and two read ports. A dual port memory with one read/write port and one write port is also an efficient memory structure suited for such a distribution of memory accesses. The GA's that we have developed could be easily adapted to handle PA for such memories. An estimator for this type of memory would be very desirable and we have the following problem.

- Development of an estimator to estimate the cost of port assignment for a dual port memory with a read/write port and a read port.

We have abstracted the influence of the design engineer on the structure of the data path using mainly two parameters. While this serves a useful characterization for many of the prevailing design examples, a more general scheme, such as the one suggested below, would be desirable.

- A language for the specification of behaviour and the abstract specification of data path structure for synthesis.

GABIND presently provides only partial support for implementing operation chaining. To reap the full benefit of operation chaining we shall have to consider sophisticated clock cycle synthesis along with the data path optimizations. This will lead to the following problem of multi-criteria optimization.

- Data path and clock cycle optimization for chained operation execution in a multi-criteria optimization framework.

We believe that data path synthesis and high level synthesis will play an important role in the design of ASIC's and other areas of VLSI design. Researchers have already proposed several schemes for integrating scheduling with allocation and binding [28, 27, 78]. A tremendous amount of work has been done in the area of physical design [79, 80, 81, 82]. An important future work would be as follows.

- Integrated data path synthesis and physical design for producing optimized data paths with a compact floor plan and efficient routing.

Appendix A

Genetic Algorithm

The concept of genetic algorithm and its importance have been discussed in chapter 1. A typical application of GA is in the problem of optimization. We now present a brief description of the algorithm in that context, along with its various operators and examples. For a detailed description, [68] can be consulted.

A.1 Genetic Operations

The genetic algorithm starts with a coding of the parameter set of the problem into a finite-length string over some finite alphabet. It initially generates a set of random solutions uniformly distributed over the space. At each generation, the solutions are rated on their performance, and the better fit ones (the ones with higher objective function) are retained. At each generation, the highly fit solutions are mated by different genetic operators to produce offsprings. These offsprings are retained to form the population for the next generation.

As an example, we consider the maximization of the function $f(x) = 20 - |x - 20|$ on the integer interval [0, 31], as shown in Figure A.1. The natural coding of the parameter is a binary string of 5 bits, over the alphabet $\{0, 1\}$. With a random selection of numbers from 0 to 31, let us select the initial population as

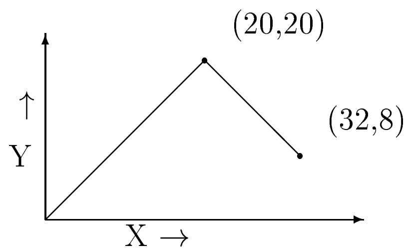


Figure A.1: The Function to Optimize in its Domain.

```
00101
11100
10111
10000
```

On this set of initial solutions, the genetic operators applied to produce the next generation are :-

- Reproduction
- Crossover
- Mutation

A.1.1 Reproduction

In reproduction, strings are copied according to the objective function f (the fitness value). The fitness value of the different solutions are shown in Table A.1. Higher the

String no.	Initial population	x value (unsigned integer)	Fitness value $f(x)$ $20 - \|x - 20\|$	$p(select)$ $f_i / \sum f_i$	Expected count $n.p(select)$	Actual count (from roulette wheel)
1	00101	5	5.0	0.10	0.40	0
2	11100	28	12.0	0.24	0.96	1
3	10111	23	17.0	0.34	1.36	2
4	10000	16	16.0	0.32	1.28	1
Sum			50.0	1.00	4.00	4
Average			12.5	0.25	1.00	1
Max			17.0	0.34	1.36	2

Table A.1: The Population String and their Fitness Values.

fitness value of a solution, higher the probability of its selection in the next generation. This biased probability of selection can be implemented in algorithm by a roulette wheel. By using this technique, we obtain the count of offsprings for the next generations as shown in the last column of Table A.1. The next generation of population is shown in the first column of Table A.2.

A.1.2 Crossover

In crossover, first the two parents and the crossover point are to be selected. In the example, we randomly select the solutions 1 and 4 from the population and select the

Mating pool after Reproduction	Mate	Crossover point	New offspring generated	x absolute value	Fitness $f(x)$
1011|1	4	4	10110	22	18.0
1|0111	3	1	11100	28	12.0
1|1100	2	1	10111	23	17.0
1000|1	1	4	10001	17	17.0
Sum					64.0
Average					16.0
Max					18.0

Table A.2: The Crossover Operation.

crossover point as 4. All the characters in the two string after this point are swapped, and hence the fifth bit of the solutions are exchanged as

$$\begin{array}{cccc|c} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{array} \quad \Longrightarrow \quad \begin{array}{cccc|c} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array}$$

Similarly, solutions 2 and 3 are crossed at point 1. The fitness values of the offsprings produced are shown in Table A.2. We note that the second operation produces no new offsprings, though the first produces a better fit solution.

A.1.3 Mutation

In mutation, a position in the string is selected at random and the value at that position is altered. It is done with a very low probability, p_m, of the order of 0.0001. In the example, to demonstrate, we select randomly the second solution and select position 3 for mutation. It produces

$$1 \quad 1 \quad \boxed{1} \quad 0 \quad 0 \qquad \Longrightarrow \qquad 1 \quad 1 \quad 0 \quad 0 \quad 0$$

Now the population is again ready for the reproduction to form new generation, and this continues for the given number of generations.

A.2 Schema

In the genetic algorithm it seems that only the pool of solutions in the population are processed. But actually, a host of other information in terms of pattern of the solution is processed implicitly. The pattern or the similarity of the solution is described in terms of *schema*. A *schema* is a *similarity template describing a subset of strings with similarities at certain string positions*.

For a string from the alphabet of k characters, the schema will be made up of $k+1$ characters — all the previous characters, and the *don't care* symbol *, which will match any value at that position. In the case of binary string, it will be made from the ternary alphabet 0, 1, *. For example the schema *10*1 describes the subset { 01001, 01010, 11001, 11011 }.

Definition A.1 *The* order *of a schema H, denoted as $o(H)$, is the number of fixed positions (for binary, the number of 0 and 1) present in the schema. For example, *10*1 has $o(H) = 3$, whereas the schema *1*** has $o(H) = 1$.*

Definition A.2 *The* defining length *of the schema H, denoted by $\delta(H)$, is the distance between the first and the last specific string positions. For example, *10*1 has $\delta(H) = 3$ while for *1*** $\delta(H) = 0$.*

If there are k characters in an alphabet, then the number of strings of length l will be k^l, while the number of schemata (*pl* of schema) will be $(k+1)^l$. Any such given string will represent 2^l schemata, since any position in the string can be replaced by the don't care symbol *. A population of size n of such strings will contain anything from 2^l to $n.2^l$ schemata, since there can be more than one example of a schema in the population.

As a numerical value, the example of the previous section has $2^5 = 32$ different possible strings compared to $3^5 = 243$ schemata. Any solution represents $2^5 = 32$ different schemata, and the whole population represents anything from 32 to $4 \times 32 = 128$ schemata. Hence a large number of information is represented by the population of only 4 solutions.

A.3 Fundamental Theorem of Genetic Algorithm

The growth of the expected number of schemata over the different generations is calculated here. At any time t, let the number of examples of a particular schema H, be $m(H,t)$. We try to find $m(H,t+1)$ after the application of the different operators.

Reproduction

Let $m_R(H,t+1)$ be the number due to application of reproduction alone. During reproduction, a string is copied to the next generation with the probability $p_i = f_i / \sum f_i$, and hence the expected number is $n.p_i$. The expected number of examples of schema H in the next generation is given as

$$\begin{aligned} m_R(H,t+1) &= \sum_{1}^{m(H,t)} n.\frac{f_i}{\sum f_i} \\ &= \frac{n}{\sum f_i} \sum_{1}^{m(H,t)} f_i \end{aligned}$$

$$\begin{aligned} &= \frac{n}{\sum f_i}.f(H).m(H,t) \\ &= m(H,t).\frac{f(H)}{\overline{f}}, \quad \text{since} \quad \frac{\sum f_i}{n} = \overline{f} \end{aligned}$$

The above iteration follows compound interest law, and it can be easily verified that the schema which are above average grows exponentially.

Crossover

The longer the defining length δ of a schema, the higher is the probability of its destruction. This is obvious from the schema *10*1 which is destroyed when a crossover point is selected between 2 and 4, whereas the schema *1*** survives any crossover. Hence in any crossover, a schema H of length l is destroyed with probability $p_d = \frac{\delta(H)}{l-1}$. When the probability of crossover is p_c, the survival probability p_{s_c} with crossover is given as

$$p_{s_c} = 1 - p_d \geq 1 - p_c.\frac{\delta(H)}{l-1}$$

Hence, $m_{RC}(H, t+1)$, the expected number of examples of the particular schema due to reproduction and crossover can be given as

$$\begin{aligned} m_{RC}(H,t+1) &= m(H,t).\frac{f(H)}{\overline{f}}.p_{s_c} \\ &\geq m(H,t).\frac{f(H)}{\overline{f}}.\left[1 - p_c.\frac{\delta(H)}{l-1}\right] \end{aligned}$$

Mutation

From the description of mutation, a given value of a string survives with a probability $p_{s_m} = (1 - p_m)$. Each schema survives, if all the fixed positions in it survives. Since each mutation is statistically independent, the schema survives from mutation with probability p_{s_m} as,

$$\begin{aligned} p_{s_m} &= (1-p_m)^{o(H)} \\ &= 1 - o(H).p_m \end{aligned}$$

since $p_m \ll 1$.

Hence, the expected number of examples of the schema after all the operations is given as

$$\begin{aligned} m(H,t+1) &= m_{RCM}(H,t+1) \\ &= m_{RC}(H,t+1).p_{s_m} \\ &\geq m(H,t).\frac{f(H)}{\overline{f}}.\left[1 - p_c.\frac{\delta(H)}{l-1}\right].(1 - o(H).p_m) \\ &\geq m(H,t).\frac{f(H)}{\overline{f}}.\left[1 - p_c.\frac{\delta(H)}{l-1} - o(H).p_m\right] \end{aligned}$$

ignoring the lower order product terms.

This final result is the *Schema Theorem* or the fundamental theorem of genetic algorithm.

A.4 The Building Block Hypothesis

From section A.3, we conclude that short (less δ), low order ($o(H)$ is less) and highly fit ($f(H) > \overline{f}$) schemata are sampled, recombined and re-sampled to form strings of potentially higher fitness. These short schemata are given a special name *building blocks*. The complexity of the problem of building high-performance strings by trying every conceivable combination is reduced to constructing better and better strings from best partial solutions of past samplings. Whether they really form better strings or not has not been proved, but is shown empirically in numerous applications. A better insight is given in section A.6.

A.5 Amount of Implicit Parallelism

The genetic algorithm implicitly processes a large number of schema while processing only the solutions of the population. It is like searching through many gradients in the gradient search method in parallel. The amount of implicit parallelism can be measured by counting the effective number of schemata processed in a generation, since many of them die off in a generation.

Since the schema of larger length do not survive long, let us find the number of schemata of length l_s or less, which will survive that generation.. The average number of such schemata will have length $l_s/2$. The number of schemata of length l_s starting at a fixed position in the string is 2^{l_s-1}. Since this fixed position can be anywhere from 1 to $(l - l_s)$, we can have the given number of schemata $(l - l_s + 1)$ times. Thus for each solution, we have $2^{l_s-1}.(l - l_s + 1)$ schemata of length less of equal to l_s. For the population of size n, an overestimate of the number of the schemata, n_s is given as,

$$n_s \leq 2^{l_s-1}.(l - l_s + 1)$$

We take the population size as $n = 2^{l_s/2}$, so that all the schemata of size $l_s/2$ or more will have a good chance to be present. The number of schemata of the given length l_s is binomially distributed. Of them half will be of length less than $l_s/2$ and the other half greater than $l_s/2$. If we consider only the higher ones, the lower bound on n_s can be found as,

$$\begin{aligned} n_s &\geq \frac{n.(l - l_s + 1).2^{l_s-1}}{2} \\ &\geq \frac{n.(l - l_s + 1).(2^{l_s/2})^2}{4} \\ &\geq \frac{(l - l_s + 1)}{4}.n^3 \end{aligned}$$

substituting n for $2^{l_s/2}$.

Hence the number of schemata of a given length effectively processed is $O(n^3)$, where n is the solution size.

A.6 Deception and the Minimal Deceptive Problem

We consider the following 2-bit deceptive problem [36]. In these examples the function f is a mapping from bit patterns to fitness function values. In this example concept called *stability* is used. Given a set H of patterns is stability $s(H)$ is the probability that the two members when crossed will also be a member of H. H is qualified on the basis of H as follows.

$s(H) = 1$	stable
$s(H) \approx 1$	semi-stable
$s(H) \approx 0$	unstable

First we consider the non-deceptive case.

String	Fitness
00	1
11	5
10	4
01	3

H_4	$f^{-1}([5,+\infty))$	$\{11\}$;	stable
H_3	$f^{-1}([4,+\infty))$	$\{10, 11\}$;	stable
H_2	$f^{-1}([3,+\infty))$	$\{01, 10, 11\}$;	semi-stable
H_1	$f^{-1}([1,+\infty))$	$\{00, 01, 10, 11\}$;	stable

In the above table only H_2 is not fully stable. However, the instability is exhibited only on crossing the two least fit members of H_2, an unlikely event. We now consider the type I deceptive problem.

String	Fitness
00	3
11	5
10	1
01	4

H_4	$f^{-1}([5,+\infty))$	$\{11\}$;	stable
H_3	$f^{-1}([4,+\infty))$	$\{01, 11\}$;	stable
H_2	$f^{-1}([3,+\infty))$	$\{00, 01, 11\}$;	semi-stable
H_1	$f^{-1}([1,+\infty))$	$\{10, 00, 01, 11\}$;	stable

An inspection of the above table reveals that the semi-stable character of H_2 is exhibited on crossing the most fit member of H_2 with the least fit member, a likely event. We now consider the type II deceptability problem.

String	Fitness
00	4
11	5
10	1
01	3

H_4	$f^{-1}([5,+\infty))$	$\{11\}$;	stable
H_3	$f^{-1}([4,+\infty))$	$\{00, 11\}$;	stable
H_2	$f^{-1}([3,+\infty))$	$\{01, 00, 11\}$;	semi-stable
H_1	$f^{-1}([1,+\infty))$	$\{10, 01, 00, 11\}$;	stable

In this case crossing the most fit members of H_2 could produce a less fit offspring. This is a serious problem. An excessive amount of type II deceptability could undermine the genetic algorithm for the particular problem. Special precautions need to be taken to handle such situations.

Appendix B

Schedules of Examples

We present here the schedules used for the allocation and binding examples in chapter 8.

B.1 Schedule for Facet Example

```
 v6 = v2 + v3
 v7 = v1 * v6        v8 =  v6 -  v4
 v9 = v7 + v2       v10 =  v6 +  v8      v11 = v8 / v5
v12 = v9 | v2       v13 = v10 & v11
```

B.2 Schedule for Differential Equation Solver Example

Schedule for Diffeq. example with five f.u.'s.

```
 v0 = dx *  u       v1 =  3 *  x        x = dx +  x
 v2 = v0 * v1       v3 =  3 *  y             x <  a
 v4 =  u - v2       v5 = dx * v3       v6 =  u * dx
  u = v4 - v5        y =  y + v6
```

Schedule for Diffeq. example with three f.u.'s.

```
 v0 =  u *  3       v1 = dx *  x        x = dx +  x
 v2 = v0 * v1       v3 =  3 *  y             x <  a
 v4 =  u - v2       v5 = v3 * dx       v6 =  u * dx
  u = v4 - v5        y =  y + v6
```

Schedule for Diffeq. example with pipelined multipliers.

A multiplication like $v1 = u \star 3$, scheduled in some time step t, to be implemented on a p stage pipelined multiplier is indicated as `[v1] = u * 3`. This indicates that the inputs t and the constant 3 are input in time step t and the output obtained after $p-1$ time steps is assigned to the variable $v1$.

```
[t1] =  u *  3
[t2] =  x * dx      x =  x + dx      x1 =  x
[t3] =  u * dx
[t4] =  y *  3      y =  y + t3
[t5] = t2 * t1           x >  a
[t6] = t4 * dx
                   t7 =  u - t5
                    u = t7 - t6
```

B.3 Schedules for Elliptic Wave Filter Example

An input interface port is indicated by `inp` and an output interface port is indicated by `out`.

Schedule using pipelined multipliers in seventeen time steps.

```
                   v1 =   b + inp       i =   i +   h
                   v3 =  v1 +   c     v26 =   i +   e
                   v5 =  v3 +   d      v2 = v26 +   f
                   v6 =  v5 +  v2
 [v7] =  v6 * x
 [v8] =  v6 * x
                   v9 =  v3 +  v7
                  v13 =  v3 +  v9     v12 =  v8 +  v2
[v14] = v13 * x   v17 = v12 +  v2     v11 =  v9 +  v6
[v19] = v17 * x     d = v11 + v12
                  v18 =  v1 + v14
                  v22 =  v1 + v18     v20 =  v9 + v18
[v29] = v22 * x   v23 =   g + v20     v21 = v19 +   f
[v25] = v23 * x   v24 = v12 + v21     v27 = v21 +   f
[v32] = v27 * x   v30 = v29 + inp       e = v24 +   i
  [h] =   e * x     g = v25 +   g       b = v30 + v18
  out = v32         c = v23 + v28       f = v32 + v21
```

Schedule using pipelined multipliers in eighteen time steps.

```
  v1   =   b + inp       v2 =   e +   f
  v3   =  v1 +   c
  v5   =  v3 +   d
  v6   =  v5 +  v2
 [v7]  =  v6 *   x
 [v8]  =  v6 *   x
                         v9 =  v3 +  v7
 v13   =  v3 +  v9      v11 =  v9 +  v6       v12 =  v8 +  v2
[v14]  = v13 *   x        d = v11 + v12       v17 = v12 +  v2
[v19]  = v17 *   x
                        v18 = v14 +  v1
 v22   =  v1 + v18      v20 =  v9 + v18       v21 = v19 +   f
[v29]  = v22 *   x      v23 = v20 +   g       v24 = v12 + v21
[v25]  = v23 *   x      v26 = v24 +   h       v27 = v21 +   f
[v31]  = v26 *   x      v30 = v29 + inp
[v32]  = v27 *   x      v28 = v25 +   g         b = v30 + v18
    h =    h + v31        c = v23 +   g         g = v28
    e = v26 +    h        f = v32 + v21       out = v32
```

Schedule using pipelined multipliers in nineteen time steps.

```
                             v1 =   b + inp
                             v3 =  v1 +   c
                             v5 =  v3 +   d        v2 =   e +   f
                             v6 =  v5 +  v2
 [v7] =  v6 * x

 [v8] =  v6 * x              v9 =  v3 +  v7
                            v13 =  v3 +  v9
[v14] = v13 * x             v12 =  v8 +  v2
                            v17 = v12 +  v2       v11 =  v9 +  v6
[v19] = v17 * x             v18 = v14 +  v1         d = v11 + v12
                            v22 =  v1 + v18       v20 =  v9 + v18
[v29] = v22 * x             v23 = v20 +   g       v21 = v19 +   f
[v25] = v23 * x             v24 = v12 + v21
                            v30 = v29 + inp       v26 = v24 +   h
[v31] = v26 * x               g = v25 +   g       v27 = v21 +   f
[v32] = v27 * x               c = v23 + v28         g = v28
                              b = v30 + v18         h =   h + v31
  out = v32                   e = v26 +   h         f = v32 + v21
```

Appendix C

Results for DSE on Random Schedules

In this appendix we tabulate results obtained after applying DSE and genetic list scheduling on a number of randomly generated partial orders. We have run three sets of experiments whose results have been tabulated in tables C.1, C.2 and C.3. Each set of experiments is characterized as follows:

The design parameters: For one set of experiments we have used two f.u.'s and for the other two we use three f.u.'s, i.e. NFUS is either two or three. NBUS and NVREF are both chosen as $3 \star NFUS$.

The number of operations in the p.o.'s: We have considered p.o.'s with twenty, twenty five and thirty operations.

The hardware operators used to generate the p.o.'s: The primitive hardware operators used are $\oplus, -$ and $+$ with costs 10, 9 and 8, respectively. The operators used for the three sets of experiments are $\langle \oplus, -, 2+ \rangle$, $\langle \oplus, -, 2+ \rangle$ and $\langle 2\oplus, 2-, 2+ \rangle$, respectively. The results are tabulated in tables C.2, C.1 and C.3, respectively.

We generate the p.o.'s so that we shall have an upper bound on the time needed to schedule the p.o. using the f.u.'s specified and at the same time satisfying the design parameters. This upper bound T_U and the hardware resources correspond to a feasible design point for the hypothetical design represented by the p.o. Using DSE we obtain a set of approximate design points with their associated hardware resource estimate and partial schedule. *In all experiments the granularity value of W of the search using REPS is five time steps.* For each design point the partial schedules along with the hardware resource requirements are passed on to GLS for final scheduling. GLS finds a schedule satisfying the design parameters using the given hardware resources. For each design point we also run force directed list scheduling (FDLS) with the hardware operators and lower bound based scheduling (LBBS) with the schedule time on the original p.o. In most cases these algorithms require more f.u. sites than NFUS. We have, therefore, also run GLS with one and two additional f.u.'s without altering the other design parameters, to obtain a better comparison of results.

The column headings of the tables are as follows: The first column (T_U) is the upper bound on schedule time as mentioned already. The second and third columns (C_E and

T_E) are the cost and time estimates for each design point output by REPS. The fourth column (T_1) is the number of time steps required for GLS using the NFUS f.u.'s indicated in the fifth column (A_1). The columns C_L, T_L and A_L represent the processor cost, time steps required to schedule and the number of f.u. sites used by LBBS. Similarly, C_F, T_F and A_F are the corresponding metrics for FDLS. The columns T_2, A_2 and T_3, A_3 are the schedule times obtained by GLS using A_2 and A_3 f.u.'s, the hardware operators used are the same as that used to schedule with A_1 f.u.'s. Each table is organized as a number of horizontal blocks, one for every p.o. on which design space exploration has been done. For every non-dominated point of the design space for a given p.o. which is found, there is a row in corresponding block containing the results of running GLS, LBBS and FDLS on that p.o. with appropriate additional inputs specific to the design point. The additional design points for a p.o. are indicated by a '→' in the first column.

T_U	C_E	T_E	T_1	$\mathbf{A_1}$	T_2	A_2	T_3	A_3	C_L	T_L	A_L	C_F	T_F	A_F
11	27	14	14	3	14	4	14	5	27	14	3	27	14	3
→	35	9	9	3	9	4	9	5	35	9	4	35	9	4
12	27	10	10	3	10	4	10	5	27	10	3	27	10	3
→	45	9	9	3	9	4	9	5	45	9	4	45	9	4
11	27	11	11	3	11	4	11	5	36	11	4	27	12	3
→	46	10	10	3	10	4	10	5	46	10	3	46	10	3
12	27	10	10	3	10	4	10	5	27	10	3	27	10	3
→	35	9	10	3	10	4	10	5	44	9	5	35	10	4
11	27	15	15	3	15	4	15	5	27	15	3	27	15	3
→	35	9	9	3	9	4	9	5	35	9	4	35	9	4
12	27	10	10	3	10	4	10	5	27	10	3	27	11	3
→	35	9	9	3	9	4	9	5	44	9	5	35	9	4
11	27	13	13	3	13	4	13	5	27	13	3	27	13	3
→	35	9	9	3	8	4	8	5	35	9	4	35	8	4
12	27	13	13	3	13	4	13	5	36	13	4	27	13	3
→	35	12	12	3	12	4	12	5	36	12	4	35	12	3
→	36	11	12	3	11	4	11	5	46	11	5	36	11	4
→	44	10	11	3	10	4	10	5	46	10	5	44	10	4

Table C.1: Table for p.o.'s of 25 operations and an upper bound operator cost of 35.

T_U	C_E	T_E	T_1	$\mathbf{A_1}$	T_2	A_2	T_3	A_3	C_L	T_L	A_L	C_F	T_F	A_F
16	27	12	12	2	10	3	11	4	27	12	3	27	11	3
$\rightarrow$	35	11	12	2	10	3	10	4	45	11	5	35	10	4
15	27	10	10	2	9	3	9	4	27	10	3	27	9	3
11	27	14	14	2	14	3	14	4	27	14	2	27	14	2
$\rightarrow$	35	10	10	2	9	3	9	4	35	9	3	35	9	3
35	27	11	11	2	9	3	9	4	27	11	3	27	9	3
13	27	12	12	2	12	3	12	4	27	12	3	27	12	2
$\rightarrow$	35	10	11	2	9	3	9	4	35	10	4	35	9	3
13	27	11	11	2	9	3	9	4	27	10	3	27	9	3
13	27	11	11	2	10	3	10	4	27	11	3	27	10	3
15	27	10	10	2	9	3	9	4	27	10	3	27	9	3
16	27	11	11	2	9	3	9	4	27	11	3	27	9	3
13	27	13	13	2	12	3	12	4	27	13	3	27	13	3
$\rightarrow$	35	12	12	2	11	3	11	4	27	12	3	35	11	3
11	27	11	11	2	9	3	9	4	27	11	3	27	9	3
14	27	11	11	2	11	3	11	4	27	11	3	27	11	3
$\rightarrow$	37	10	10	2	8	3	7	4	37	8	4	37	7	3
12	27	10	10	2	9	3	9	4	27	10	3	27	9	3
11	27	10	10	2	9	3	9	4	27	10	3	27	9	3
12	27	11	11	2	11	3	11	4	27	11	3	27	11	3
11	27	11	11	2	11	3	11	4	27	11	3	27	11	3
12	27	11	11	2	9	3	9	4	27	11	3	27	9	3
13	27	11	11	2	10	3	10	4	27	10	3	27	10	3
11	27	10	10	2	10	3	10	4	27	10	3	27	10	3
11	27	10	10	2	8	3	8	4	27	8	3	27	8	3
13	27	10	11	2	9	3	9	4	27	10	3	27	9	3
14	27	12	12	2	10	3	10	4	27	12	3	27	10	3
12	27	11	11	2	9	3	9	4	27	11	3	27	9	3
12	27	11	11	2	11	3	11	4	27	11	3	27	11	3
11	27	12	12	2	12	3	12	4	27	12	2	27	12	2
$\rightarrow$	35	10	10	2	9	3	9	4	35	10	3	35	9	3
12	27	11	11	2	10	3	10	4	27	11	2	27	10	3
$\rightarrow$	35	10	10	2	10	3	10	4	27	10	3	35	10	2
13	27	11	11	2	11	3	11	4	27	11	3	27	11	2
13	27	10	10	2	9	3	9	4	27	9	3	27	9	3
13	27	11	11	2	10	3	10	4	27	11	3	27	10	3
14	27	11	11	2	10	3	10	4	27	11	3	27	11	3

Table C.2: Table for p.o.'s of 20 operations and an upper bound operator cost of 35.

T_U	C_E	T_E	T_1	$\mathbf{A_1}$	T_2	A_2	T_3	A_3	C_L	T_L	A_L	C_F	T_F	A_F
12	27	13	13	3	13	4	13	5	36	13	4	27	13	3
→	35	12	12	3	12	4	12	5	36	12	4	35	12	3
→	36	11	12	3	11	4	11	5	46	11	5	36	11	4
→	44	10	11	3	10	4	10	5	46	10	5	44	10	4
15	27	15	15	3	15	4	15	5	35	15	4	27	15	3
→	36	13	13	3	13	4	13	5	36	14	4	36	13	4
→	44	11	11	3	9	4	9	5	44	11	5	44	9	5
12	27	11	11	3	11	4	11	5	35	11	3	27	12	3
→	37	10	10	3	10	4	10	5	37	10	4	37	11	4
14	27	12	12	3	12	4	12	5	35	12	4	27	12	3
13	27	13	13	3	13	4	13	5	35	13	4	27	13	3
→	37	11	11	3	10	4	10	5	37	11	4	37	10	4
14	27	12	13	3	13	4	13	5	35	12	4	27	13	3
→	36	11	12	3	11	4	11	5	44	11	4	36	11	4
13	27	15	15	3	15	4	15	5	35	15	4	27	15	3
→	35	13	13	3	12	4	12	5	35	13	4	35	12	4
→	44	12	13	3	12	4	12	5	35	12	4	44	12	4
13	27	12	12	3	12	4	12	5	44	12	4	27	13	3
→	44	11	11	3	9	4	9	5	44	10	5	44	9	5
13	27	12	12	3	12	4	12	5	27	12	3	27	12	3
→	37	10	11	3	10	4	10	5	37	10	4	37	11	4
13	27	13	13	3	13	4	13	5	37	13	3	27	13	3
→	35	12	12	3	12	4	12	5	37	12	3	35	12	3
→	44	11	12	3	11	4	11	5	46	11	3	44	11	4
→	54	10	10	3	10	4	10	5	54	10	4	54	10	4
14	27	13	13	3	13	4	13	5	27	13	3	27	14	3
→	35	11	11	3	11	4	11	5	45	11	4	35	12	3
13	27	12	12	3	12	4	12	5	27	12	3	27	13	3
→	45	11	12	3	10	4	10	5	45	11	4	45	10	5
14	27	12	12	3	12	4	12	5	27	12	3	27	12	3
→	36	11	12	3	12	4	12	5	44	11	4	36	12	4
14	27	17	17	3	17	4	17	5	27	17	3	27	17	3
→	36	11	11	3	10	4	10	5	36	11	4	36	10	4

Table C.3: Table for p.o.'s of 30 operations and an upper bound operator cost of 54.

Bibliography

[1] C. Mead and L. Conway, eds., *System Timimg.* Addison-Wesley, 1980.

[2] M. Shadad, “An overview of VHDL language and technology,” *Procs. of the 23rd Design Automation Conference*, 1986.

[3] D. E. Thomas and P. Moorby, *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[4] P. Marwedel, “The MIMOLA design system :detailed description of the software system,” in *16-th Design Automation Conference proceedings*, pp. 59–63, 1979.

[5] D. Ku and G. D. Micheli, “Hardware C- a language for hardware design version 2.0,” Tech. Rep. Technical Report No. CSL-TR-90-419, Computer Systems Laboratory, Standford University, Standford, CA, Apr. 1990.

[6] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *COMPILERS Principles, Techniques and Tools.* Addison-Wesley Publishing Company, June 1987.

[8] L. Stok, *Architectural Synthesis and Optimization of Digital Systems.* PhD thesis, Eindhoven University of Technology, March 1991.

[9] T. Villa and A. Sangiovanni Vincentelli, “NOVA :state assignment of finite state machines for optimal two-level logic implementation,” *IEEE Trans. on C. A. D.*, vol. vol-9 No 9, pp. 905–924, Sep. 1990.

[10] G. DeMicheli, “Symbolic design of combinational and sequential logic circuits implemented by two level logic macros,” *IEEE Trans. on CAD*, vol. vol CAD-5, pp. 597–616, Oct. 1986.

[11] X. Du, G. Hachtel, B. Lin, and A. R. Newton, “MUSE : A multilevel symbolic encoding algorithm for state assignment,” *IEEE Trans. on CAD*, vol. vol 10 No 1, pp. 28–38, Jan. 1991.

[12] L. Stok, “Data path synthesis,” *INTEGATION, the VLSI journal*, vol. 18, pp. 1–71, 1994.

[13] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Procs. of the 25th ACM/IEEE Design Automation Conference*, 1988.

[14] J. Granacki, D. Knapp, and A. C. Parker, "The ADAM advanced design automation system: Overview, planner and natural language interface," in *Procs., 22nd Design Automation Conference*, pp. 727–730, June 1985.

[15] P. G. Paulin and J. P. Knight, "Force–directed scheduling in automatic data path synthesis," *Procs. of the 24th Design Automation Conference*, 1987.

[16] F. Brewer and D. D. Gajski, "Chippe: A system for constraint driven behavioural synthesis," *IEEE Trans. on C. A. D.*, pp. 681–695, July 1990.

[17] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital-systems," *IEEE Trans. on C. A. D.*, vol. 5, pp. 379–395, July 1986.

[18] G. D. Micheli and D. C. Ku, "Hercules: A system for high level synthesis," in *Procs. of the 25th ACM/IEEE DAC*, pp. 483–488, 1988.

[19] D. K. Banerjee, J. C. Majithia, T. C. Wilson, A. Basu, S. Sutarwala, and A. K. Majumdar, "High-level synthesis of data-paths from a behavioural description," *International Jou. of Computer Aided VLSI Design*, vol. 3, pp. 367–391, 1991.

[20] A. Kumar, *A Versatile Data Path Synthesis Approach Based on Heuristic Search.* PhD thesis, I.I.T. Delhi, Jan. 1993.

[21] F. J. Kurdahi and A. C. Parker, "Real: A program for register allocation," *Procs. of the 24th Design Automation Conference*, 1987.

[22] J. Lee, Y. Hsu, and Y. Lin, "A new integer linear programming formulation of the scheduling problem in data path synthesis," in *Procs. of the International Conference on Computer-Aided Design*, pp. 20–23, 1988.

[23] A. Kumar, A. Kumar, and M. Balakrishnan, "A novel integrated scheduling and allocation algorithm for data path synthesis," *Procs. of VLSI Design '91*, pp. 212–218, 1991.

[24] C.-T. Hwang and Y.-C. Hsu, "Zone scheduling," *IEEE Trans. on C. A. D.*, vol. 12, pp. 926–934, 1993.

[25] F.-S. Tsai and Y.-C. Hsu, "STAR - An automatic data path allocator," *IEEE Trans. on C. A. D.*, pp. 1053–1064, Sep. 1992.

[26] M. Rim, R. Jain, and R. De Leone, "Optimal allocation and binding in high-level synthesis," *29th ACM/IEEE Design Automation Conference*, pp. 120–123, 1992.

[27] R. J. Cloutier and D. E. Thomas, "The combination of scheduling, allocation and mapping in a single algorithm," in *Procs. of the 27th ACM/IEEE DAC*, pp. 71–76, June 1990.

[28] M. Balakrishnan and P. Marwedel, "Integrated scheduling and binding: A synthesis approach for design space exploration," in *Procs. of the 26th ACM/IEEE DAC*, pp. 68–74, 1989.

[29] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Trans. on C. A. D.*, vol. 8, July 1989.

[30] F. J. Kurdahi and A. C. Parker, "Plest: A program for area estimation of VLSI integrated circuits," *Procs. of the 23rd Design Automation Conference*, 1986.

[31] B. M. Pangrle, "On the complexity of connectivity binding," *IEEE Trans. on C. A. D.*, vol. 10, pp. 1460–1465, Nov. 1991.

[32] E. G. Coffman Jr eds, *Computer and Job Shop Scheduling Theory.* John Wiley & Sons., 1976.

[33] M. Srinivas and L. M. Patnaik, "A tutorial on genetic algorithms," *IEEE Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[34] R. L. Graham, "Bounds on some multiprocessing anomalies," *Bell System Technical Jou.*, vol. 45, pp. 1563–1581, 1966.

[35] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Procs. of the 8-th Design Automation Workshop*, pp. 155–169, 1971.

[36] M. D. Vose, "Generalizing the notion of schema in genetic algorithms (research note)," *Artificial Intelligence*, vol. 50, pp. 385–396, 1991.

[37] B. S. Stewart and C. C. White, "Multiobjective A*," *JACM*, vol. 88, no. 4, pp. 775–814, 1991.

[38] D. Sreenivasa Rao and F. J. Kurdahi, "Hierarchical design space exploration for a class of digital systems," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 1, pp. 282–295, Sep. 1993.

[39] P. K. Jha, C. Ramachandran, N. D. Dutt, and F. J. Kurdahi, "An empirical study on the effects of physical design in high-level synthesis," in *Procs. of VLSI Design '94*, pp. 11–16, 1994.

[40] M. B. Takla, D. W. Bouldin, and D. B. Koch, "Early exploration of the multi-dimensional vlsi design space," in *Procs. of VLSI Design '94*, pp. 413–416, 1994.

[41] P. G. Paulin and J. P. Knight, "Algorithms for high-level synthesis," *IEEE D. & T. of Computers*, pp. 18–31, Dec. 1989.

[42] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing.* Prentice Hall, 1984.

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

[44] L. B. Booker, D. E. Goldberg, and J. H. Holland, "Classifier systems and genetic algorithms," *Artificial Intelligence*, vol. 40, no. 2, pp. 235–282, 1989.

[45] R. Camposano, "Structural synthesis in the yorktown silicon compiler," in *Procs. of VLSI 87 Conference, Vancouver*, Aug. 1987.

[46] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. on Computer*, vol. C-21, pp. 137–146, Feb. 1972.

[47] J. D. Ullman, "Polynomial complete scheduling problems," *Operating Systems Review*, vol. 7, no. 4, 1973.

[48] M. R. Garey and D. S. Johnson, "Complexity results for multi-processor scheduling under resource constraints," *Procs. of 8-th Annual Princeton Conference on Information Sciences and Systems*, 1974.

[49] M. Garey and D. Johnson, *A guide to the theory of NP-completeness.* Freeman, San Fransisco, 1979.

[50] P. G. Paulin and J. P. Knight, "Force-directed scheduling for ASICs," *IEEE Trans. on C. A. D.*, June 1989.

[51] A. C. Parker, J. T. Pizarro, and M. Mlinar, "Maha: A program for data path synthesis," *Procs. of the 23rd Design Automation Conference*, 1986.

[52] E. W. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *Sequencing and Scheduling: Algorithms and Complexity, in Handbook in Operations Research and Management Sciences*, vol. 4: Logistics of Production and Inventory. North-Holland, 1993.

[53] B. Berger and L. Cowen, "Complexity results and algorithms for $\{<, \leq, =\}$– constrained scheduling," in *Procs. of 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, C. A.*, pp. 137–147, 1991.

[54] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press and McGraw Hill, 1990.

[55] E. Horowitz and S. Sahni, *Computer Algorithms.* Galgotia Press, New Delhi, INDIA, 1988.

[56] T. C. Wilson, D. K. Banerjee, A. Basu, J. C. Majithia, and A. K. Majumdar, "Port assignment in multiport memories for interconnection minimization in data path synthesis," in *Procs. of IFIP Working Conference on Logic and Architecture Synthesis, Paris*, May 1990.

[57] M. W. Krentel, "The complexity of optimization problems," *Jou. of Computer and System Sciences*, vol. 36, pp. 490–509, 1988.

[58] D. S. Johnson, "Worst case behaviour of graph colouring algoorithms," *Prooceedings of 5th South Eastern Conference on Combinatorics, Graph Theory & Computing*, 1974.

[59] C. Lund and Y. Mihalis, "On the hardness of approximate minimization problems," in *Procs. of the* 25^{th} *Annual ACM Symposium of the Theory of Computing*, 1993.

[60] S. M. Korman, *Graph coloring and related problems in operations research.* PhD thesis, Imperial College, London, 1975.

[61] F. Gavril, "Algorithms for minimum colouring, maximum clique, minimum covering by cliques and maximum independent set of a chordal graph," *SIAM Jou. of Computing*, vol. 1, pp. 180–187, 1972.

[62] P. G. Paulin, "Algorithms for high level synthesis with area and interconnect constraints," *Procs. of EuroAsic89*, pp. 144–158, Jan. 1989.

[63] C. A. Mandal, "ABS: An automated behavioural synthesis system," m. tech. project dissertation, Indian Institute of Technology, Kharagpur, Department of Computer Science and Engineering, 1989.

[64] T. C. Wilson, D. K. Banerjee, J. C. Majithia, and A. K. Majumdar, "Optimal allocation of multiport memories in datapath synthesis," in *Procs. of 32nd Midwest Symposium on Circuits and Systems Urbana Ill*, pp. 1070–1073, Aug. 1989.

[65] M. Balakrishnan, A. K. Majumdar, D. K. Banerjee, J. G. Linders, and J. C. Majithia, "Allocation of multiport memories in data path synthesis," *IEEE Trans. on C. A. D.*, vol. 7 No 4, pp. 536–540, Apr. 1988.

[66] S. Sutarwala, D. K. Banerjee, A. K. Majumdar, and J. G. Linders, "Gregmap: A design automation tool for interconnect minimization," in *Procs. of the Canadian Conference on VLSI, Halifax*, pp. 362–371, Oct. 1988.

[67] N. Deo, *Graph Theory, Applications to Engineering and Computer Science.* Prentice Hall of India, 1986.

[68] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. Inc., 1989.

[69] J. H. Holland, *Adaptation in natural and artificial systems.* Ann Arbor MI: The University of Michigan Press, 1975.

[70] L. Davis, *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold, 1991.

[71] A. M. Frieze, *Probabilistic Analysis of Graph Algorithms.* Springer-Verlag, 1989.

[72] S. Minton, M. D. Jhonson, A. B. Philips, and P. Laird, "Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems," *Artificial Intelligence*, pp. 161–205, 1992.

[73] U. Holtmann and E. Rolf, "Experiments with low-level speculative computation based on multiple branch prediction," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 1, pp. 262–267, Sep. 1993.

[74] J. Lienig and K. Thulasiraman, "A new genetic algorithm for channel routing," in *Procs. of VLSI Design '94*, pp. 133–136, 1994.

[75] R. Rajarajan and C. P. Ravikumar, "Genetic algorithm for scan path design," in *Procs. of VLSI Design '96*, 1996.

[76] N. Vijaykrishnan and N. Ranganathan, "A genetic approach for subcircuit extraction," in *Procs. of VLSI Design '96*, 1996.

[77] V. Saxena and C. P. Ravikumar, "Synthesis of testable pipelined datapaths using genetic search," in *Procs. of VLSI Design '96*, 1996.

[78] S. Devadas, H. K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG : state assignment of finite state machines targeting multilevel logic implementations," *IEEE Trans. on C. A. D.*, vol. vol 7 No 12, pp. 1290–1299, Dec. 1988.

[79] K. Doll, F. M. Johannes, and K. J. Antreich, "Iterative placement improvement by network flow methods," *IEEE Trans. on C. A. D.*, vol. 13, pp. 1189–1200, Oct. 1994.

[80] K. Lee and C. Sechen, "A new global router for row based layout," in *IEEE Intl. Conference on Computer Aided Design*, pp. 180–183, 1988.

[81] S. Goto and E. S. Kuh, "An approach to two dimentional placement problem in circuit layout," *IEEE Trans. on Circuits and Systems*, vol. CAS-25, pp. 208–214, Apr. 1978.

[82] S. B. Akers, "On the use of linear assignment algorithm in module placement," in *18-th ACM / IEEE DAC*, pp. 137–144, 1981.

Publications of Author

1. C. A. Mandal, P. P. Chakrabarti & S. Ghose, Complexity of Scheduling in High Level Synthesis, *to appear in VLSI DESIGN*.

2. C. A. Mandal, P. P. Chakrabarti & S. Ghose, Allocation and Binding for Data Path Synthesis Using a Genetic Approach, *to appear in Proceedings of VLSI Design '96*, Bangalore, INDIA, 1996.

3. C. A. Mandal, P. P. Chakrabarti & S. Ghose, A Framework for High Level Synthesis, *International Workshop on Artificial Intelligence*, I.I.M., Calcutta, March, 1994.

4. C. A. Mandal, P. P. Chakrabarti & S. Ghose, Complexity of Scheduling 2-Operation Chains and Some Other Related Scheduling Problems, *Proceedings of the Fourth National Seminar on Theoretical Computer Science*, IIT Kanpur, INDIA, pp. 171-180, 1994.

5. C. A. Mandal, P. P. Chakrabarti & S. Ghose, Interconnect Optimization Techniques in Data Path Synthesis, *Proceedings of VLSI Design '92, Bangalore*, pp. 85-90, 1991.

6. C. A. Mandal, P. P. Chakrabarti & S. Ghose, Register–Interconnect Optimization in Data Path Synthesis, *Microprocessing and Microprogramming*, pp. 279-288, vol. 33, 1991.

7. C. A. Mandal, P. P. Chakrabarti & S. Ghose, Allocation of Registers to Multi-port Memories Based on Register–Interconnect Optimization, *Advances in Modeling and Simulation*, vol. 25, number 4, 1991.

8. C. A. Mandal & P. Pal Chaudhuri, ABS: An Automated Behavioural Synthesis System, *Proceedings of VLSI Design '90, Bangalore*, pp. 18-23, 1991.

9. Other communicated papers.

Biodata of Author

Chittaranjan A. Mandal was born on 28th February, 1966 in West Bengal, India. He graduated with **B. Tech. (Hons.)** in *Computer Science & Engineering* in 1987 and **M. Tech.** in *Computer and Information Technology* in 1990, both from Indian Institute of Technology, Kharagpur. He is presently a Lecturer in the Department of Computer Science & Engineering, Jadavpur University, Calcutta. His research interests include VLSI design, high level synthesis, FPGA based synthesis, computer architecture and algorithms.