UNIVERSITY OF SOUTHAMPTON

MULTIPLE OBJECTIVE OPTIMISATION OF DATA AND CONTROL PATHS IN A BEHAVIOURAL SILICON COMPILER

by

Keith Richard Baker

A dissertation submission for the title of
Doctor of Philosophy.

Department of Electronics and Computer Science,
University of Southampton.

September, 1992

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

MULTIPLE OBJECTIVE OPTIMISATION OF DATA AND CONTROL PATHS IN A BEHAVIOURAL SILICON COMPILER

by Keith Richard Baker

The objective of this research was to implement an "intelligent" silicon compiler that provides the ability to automatically explore the design space and optimise a design, given as a behavioural description, with respect to multiple objectives. The objective has been met by the implementation of the MOODS Silicon Compiler. The user submits goals or objectives to the system which automatically finds near optimal solutions. As objectives may be conflicting, trade-offs between synthesis tasks are essential and consequently their simultaneous execution must occur. Tasks are decomposed into behaviour preserving transformations which due to their completeness can be applied in any sequence to a multi-level representation of the design. An accurate evaluation of the design is ensured by feeding up technology dependent information to a cost function. The cost function guides the simulated annealing algorithm in applying transformations to iteratively optimise the design.

The simulated annealing algorithm provides an abstractness from the transformations and designer's objectives. This abstractness avoids the construction of tailored heuristics which pre-program trade-offs into a system. Pre-programmed trade-offs are used in most systems by assuming a particular shape to the trade-off curve and are inappropriate as trade-offs are technology dependent. The lack of pre-programmed trade-offs in the MOODS system allows it to adapt to changes in technology or library cells. The choice of cells and their subsequent sharing are based on the user's criteria expressed in the cost function, rather than being pre-programmed into the system.

The results show that implementations created by MOODS are better than or equal to those achieved by other systems. Comparisons with other systems highlighted the importance of specifying all of a designs data as the lack of data misrepresents the design leading to misleading comparisons.

The MOODS synthesis system includes an efficient method for automated design space exploration where a varied set of near optimal implementations can be produced from a single behavioural specification. Design space exploration is an important aspect of designing by high-level synthesis and in the development of synthesis systems. It allows the designer to obtain a perspicuous characterization of a design's design space allowing him to investigate alternative designs.

CONTENTS

ILLUSTRATIONS

TABLES

ACKNOWLEDGEMENTS

The author wishes to acknowledge the following people: Professor K G Nichols for supervision and encouragement during the initial stages of this research and in obtaining the author's research assistantship, without which the research would not have been possible. Mr A J Currie for supervising the latter stages of the research and checking the technical and grammatical content of this thesis and Dr M Zwolinski for dealing with numerous queries and problems with the computer systems. Thanks should also go to other members of the University and family for their help and encouragement during the research, especially to Lynne Buddle for proof reading this thesis[1].

[1] This thesis has been written in accordance with the *British Standard Recommendations for the Presentation of thesis and dissertations*, BS4821:1990, *Abbreviations of titles of Periodicals*, BS4148 and *Specification for numbering of divisions and subdivisions in written documents (point-numbering)*, BS5848; as well as the notes set out in the booklet *Completion of research degree candidature*, University of Southampton, July 1991.

1 INTRODUCTION

The design of electronic systems is a highly complex process, where each integrated circuit (IC) may take several man years to complete. The design process starts with the design specification and terminates in chip fabrication using a set of masking plates containing images of the IC structures. The advances in VLSI technology mean that computer aided design tools play an essential part in synthesizing circuits in a reasonable time. Early tools consisted of automated layout tools, such as, mask design rule checkers and tools for placement and routing of small frequently used logic blocks (standard cells). The manual layout of logic blocks was later complemented by optimisers which rearranged mask structures, subject to design rules, in order to compact the layout. The early 80s saw the creation of a new generation of tool, the *silicon compiler*, which, from a structural description of a design synthesized an implementation using standard cells.

During the late 80s and early 90s further increases in the complexity of integrated circuits resulted in an abstractness, from layout, of the synthesis tools and design descriptions. The abstract design description represents the design's required behaviour and the silicon compiler was given the ability to select components and design style in order to optimise the design. Synthesis tools attempt to produce a hardware implementation that is optimal with respect to some aspect of the design. Automated optimisation of circuits is necessary to produce area efficient and/or fast designs. Usually the area of the design is optimised whilst maintaining a constraint on the speed, or vice versa.

The silicon compiler should be part of a complete compilation environment that may include other compilers and tools such as multi-level simulators, timing verifiers, testability rule checkers and enforcers, and automatic test pattern generators [1]. Due to the evolution of synthesis tools the definition of a silicon compiler is vague, however, it has best been defined as follows: "a silicon compiler is an *optimising* transformation program that produces *manufacturable* integrated circuit designs from *intelligible* descriptions"[2]. The *manufacturable* and *intelligible/optimising* aspects of the silicon compiler often occur separately in layout tools and high-level synthesis systems respectively. The mythical term "silicon compiler" is more usually applied to any program that compiles a description whose output will eventually be a manufacturable IC design and is typically classified by its input detail and architectural model. An "intelligent" silicon compiler is one that can make trade-offs and provide the designer with an insightful characterization of design alternatives [3].

The incentives for silicon compilers include short design time and therefore reduced cost, the possibility for the designer to explore different strategies and technologies and correctness by construction. This makes the realisation of low-volume application specific designs (ASICs) cost effective. In addition designers often require to explore a range of implementations for each design specification (*design space exploration*) and may not require the optimum solution but one that satisfies several simultaneous constraints.

The silicon compiler is intended to provide access to silicon for systems designers. The use of a behavioural description, that is, one which describes what a design does rather than how it is implemented, places the capability of designing VLSI circuits in the hands of those not skilled in VLSI design. It allows the designer to concentrate on the functionality of the design. Silicon compilation should be a "non-interactive process"[4] and the majority of users "need not worry about the hardware implementation of the description"[5]. However, the skilled user may desire to fine tune the resulting implementation and so some optional interaction is deemed necessary.

Design is a multi-stage process of refinement through various levels of representation (from behavioural to layout) with occasional backtracking to an earlier stage, the cause of which is largely due to human errors. By using a design process which has been proved to be correct the resulting implementation can be guaranteed to be correct; this is correctness by construction. Correctness by construction given by the use of silicon compilers ensures the final implementation to be functionally equivalent to the input specification, that is, both interact with the environment in the same way. Therefore if the design specification is verified to be correct by functional simulation, then the implementation will also be correct. Correctness by construction can be guaranteed by proving the correctness of each of the synthesis steps applied during compilation.

In addition to the above advantages silicon compilers allow early error detection by semantic checks at high levels thus resulting in less errors. The possibility of protecting technology knowledge will also be an important advantage in future commercial systems.

Synthesis is the refinement of a design from an abstract level to a less abstract, lower level, during which some optimisation usually takes place. The levels of representation range from the highest, most abstract level, the functional (behavioural) specification, to the lowest, most specific level, the layout. Between the functional and layout levels are, in decreasing abstractness, the architectural, register-transfer, logic, and circuit levels. In general high-level refers to the functional through to register-transfer levels and low-level refers to the logic through to layout levels. The earlier definition of a silicon compiler encompasses a wide variety of tools, such as, layout, logic synthesis and high-level synthesis tools.

High-level silicon compilation comprises the following issues:

a. definition of an input and output specification,
b. definition of an architectural model,
c. definition of an internal representation,
d. high-level synthesis,
e. hardware synthesis and layout, and
f. design space exploration.

The remainder of this chapter introduces and discusses the issues of high-level silicon compilation and high-level synthesis in particular behavioural synthesis; the main topic of this research. The last section describes the project objectives. Further information on the general issues of silicon compilation can be found in references [1,6,7,8,9,10], however, the publication date should be borne in mind as some theories and views may be outmoded.

The rest of this thesis is organised as follows: Chapter 2 is a literature survey of previous high-level synthesis systems and shows the current research status in the area of silicon compilation. It describes and compares the systems and illustrates their drawbacks and how they have been overcome in this synthesis system; the MOODS Silicon Compiler. Chapter 3 describes the input, output and architectural models chosen and the optimisation techniques used in this system. Chapters 4 and 5 detail the transformations and optimisation algorithms and Chapter 6 describes the results and compares MOODS with existing systems. Chapter 7 sums up what has been achieved and gives suggestions for further work.

1.1 INPUT AND OUTPUT SPECIFICATION

Silicon compilers are classified by their input specification which can be either structural, architectural or behavioural (algorithmic). A structural description specifies the circuit structure, its components and the connectivity between them. The structure may be described at various levels of abstraction, from switch or gate level to high-level using parameterization. Similar levels of abstraction are found in the architectural and behavioural languages, both of which are functional languages. A functional language specifies the circuit's input/output mapping. The structure is not explicitly specified however algorithmic languages such as MacPitts [11] and Silc [4], have predictable structural semantics, that is, the functional constructs imply certain structural elements in a predictable way. Functional languages without predictable structural semantics are behavioural languages. A language is not only defined by its semantics but also by its interpretation by a synthesis system, for example, a behavioural language could be interpreted as an architectural language using direct compilation where each construct produces a pre-defined circuit structure.

A behavioural description specifies the relationship between system inputs and outputs by describing data structures and functions to manipulate them. Their physical structure is not described as the emphasis is on what a design does and not how to do it. A behavioural description documents the design in a readable, technology independent way. It frees the designer from selecting a good implementation as it does not include design decisions such as timing and parallelism. Explicit parallelism is a useful aid in writing readable design specifications and therefore may occur in a behavioural description; however, explicit parallelism is not adhered to during behavioural synthesis. In addition variables and data structures are not bound to registers or memory and operations are not bound to functional units or control states. The lack of premature bindings allows for more optimisation opportunities as it does not limit the design space as in architectural or structural languages.

The input language may include parallelism, data typing, macro expansions and subroutine calls and must be general enough to describe a large class of problems. Some compilers limit the language to reduce the design space, this simplifies the compiler and limits it to particular design styles; for example, the First compiler [12] for digital signal processing (DSP) applications.

The input language is usually compiled to an intermediate form and optimisations such as dead code elimination and constant folding are done at this stage. It is usual to have a functional simulator that uses the intermediate form as its input, thus ensuring the correctness of the design. The intermediate code is at the register transfer level and consists of operations, register transfers and next states. A typical example of an intemerdiate code is the Value Trace (VT) described in Section 2.3 which is used in the CMU-DA systems [13,14,15].

The output of a silicon compiler should, by the definition of an ideal compiler [16], be mask layouts suitable for use in the fabrication of integrated circuits (ICs). However, the process is often divided into synthesis and layout steps. The compiler performs the synthesis and generates a netlist of library cells which is used to generate the final layout. The reason for this is to limit the computational explosion that would result from directly synthesizing mask details from high-level descriptions. Library cells are also used to increase correctness and reduce design time by increasing the granularity of the output. In early structural level compilers a fixed floor plan was used, for example in MacPitts [11] and First [12], however in general high-level silicon compilers this has proved inefficient. It is important, however, to pass information between synthesis and layout tools, for example, constraining net lengths and/or feedback of layout effects on performance, in order to provide predictable performance characteristics [17].

The input description and output implementation should be functionally equivalent which can be determined by simulation. Functional equivalence is defined as follows: for the same initial conditions and external inputs, equivalent programs must produce exactly the same external events in the same order [18].

1.2 ARCHITECTURAL MODEL

The architectural model falls into set categories the most common of which are, data path only and data path plus control unit. Data path only architecture is limited in that it can only provide designs implementing calculations and not decisions. An example of this is DSP applications in Spaid [19] and First [12]. The data path plus control unit architecture has operations in the data path which determine the next state of the controller. The controller can be state or transition based, for example Scholyzer [20] or Camad [21] respectively. The data path typically consists of functional units at the register-transfer level which exist in soft libraries consisting of parameterized cells that can be tailored to the desired functions.

To avoid race conditions master slave registers are often used and clocked logic changes at the end of a cycle, therefore, a signal can only travel through at most one register per clock cycle [22].

An effective way to cut down the search space is to make architectural decisions and constraints. Synchronous systems use a centrally clocked controller which activates processes at fixed time steps (for example, a micro-program as in Camad [23] or FSMs as in Maha [24]). However, it may suffer from clock skew and the controller complexity increases with parallelism. Asynchronous systems use many independently clocked or self-timed modules. Self-timed modules use a delay unit for each part of combinational logic. The output of the delay unit creates the acknowledge signal from a delayed request signal. The delay unit has a fixed delay and for the purpose of reliability, tends to be over estimated using worst case values. Reliability and speed, therefore, depends on the accuracy of technology dependent delay models.

1.3 INTERNAL REPRESENTATION

Graphs are used as internal representations as they conveniently describe the design at various levels of abstraction, from behavioural to structural. They are used as intermediate representations during the synthesis processes.

A control and data flow graph can be implemented as a single graph where nodes represent data or control operations and edges represent results. Alternatively they can be implemented as two separate graphs, where a control graph conveys information about the sequence of operations and a data flow graph specifies data dependencies. The nodes of a data path graph represent variables, constants and functional units and the arcs represent information flow. Data flow graphs may have conditions on arcs to *gate* data path signals; where the conditions are generated either by other data flow nodes or by the controller.

Two methods commonly used to represent a control graph are modified Petri-nets and precedence graphs. Petri-nets consist of places and transitions connected by directed edges. The transitions represent actions and operations that are executed whenever flags on places are marked as true. When an action is performed the marking from an input place is immediately passed to the output place, thereby comprising a token passing mechanism. Petri-nets do not include timing, only a partial ordering. Timing can be introduced by holding a token a fixed time, long enough for operations to be completed; for example, the extended timed Petri-net (ETPN) model in Camad [23]. The holding time may be equal to the clock period in synchronous circuits or the delay of combinational logic in self-timed systems. A precedence graph is a directed graph indicating the order of its nodes, the operations, by the edges of the graph.

1.4 BEHAVIOURAL SYNTHESIS

Behavioural synthesis, the highest level synthesis, is the conversion of a behavioural description to a structural implementation. The behavioural description contains implied operation ordering and some specific parallelism. The ordering of operations and any constraints, for example I/O timing, are all that must be maintained throughout behavioural synthesis. Parallelism, unit assignments and other bindings are not made in the description but design decisions concerning them are performed during the synthesis process. Behavioural synthesis consists of the following sub-problems:

a. allocation of operators to functional units,
b. allocation of variables to storage elements,
c. scheduling of operations,
d. allocation of interconnects,
e. translation, and
f. binding.

These synthesis tasks are discussed individually below.

The allocation of data path operators to functional units and variables to storage elements is a many to many mapping and involves sharing units and combining different units into ALUs. In some systems [25] allocation consists of two steps; firstly the selection of a set of functional units to execute the operations and secondly the assignment of operations to specific units.

The timing of units to be combined must be observed as the corresponding operations cannot be executed concurrently on a single merged unit. Many methods have been used to perform allocation; from user specified as in MacPitts [11] to clique partitioning to create ALUs as in Facet [26]. Clique partitioning is the partitioning of a graph, G, into the minimum set of disjoint cliques; where a clique, C, is a fully connected sub-graph of G (that is every node connects to every other node) and C is not contained within a larger fully connected sub-graph of G. Several methods involve heuristics to determine the order to allocate data path units. *Critical path first* is a common method used in Maha [24], Slicer [27] and Hal [25], which allocates units with the lowest freedom first. Freedom is the delay allowable in an operation without lengthening the critical path and is equivalent to path slack in project management. The units with high freedoms will have more opportunities to share resources as their instructions can be moved within the time frame given by the freedom. By allocating the lowest freedom first units have a greater possibility of sharing resources.

Scheduling involves allocating operations to time slices (time-slice allocation) or control states (state allocation). Operations within a time step are executed concurrently, with registers being loaded and read at the step boundaries. Dependencies between concurrent operations results in the operations being chained, that is, one operation will follow on from another. The scheduling of operations can be performed using various techniques, the simplest, with the exception of relegating the task to the designer, is "as soon as possible" (ASAP) compaction as used in Scholyzer [20]. ASAP scheduling places operations into the earliest time slot subject to data dependencies. A refinement of this is ASAP scheduling with conditional postponement as used in Mimola [28], where operations are postponed whenever operator concurrency exceeds the available resources.

List scheduling involves sorting operations in topological order, as defined by their dependencies. The operations are then iteratively scheduling into control steps using a priority function. The priority function determines the order in which operations are placed in a control step. Urgency scheduling, as used in Elf [29], is a form of list scheduling whose priority function uses urgency measures based on freedoms and the possibilities of sharing operators. Other techniques include iterative scheduling and control graph partitioning using algorithmic methods such as clique partitioning as in Facet [26].

Translation changes part of a design to another more useful or efficient one at the same level of abstraction. For example, if a multiplier has been specified but it does not exist in the cell library then it may be translated into an implementable form, such as cascaded adders. A translation may result in an improved design by allowing for better scheduling or the possibility to share data path units.

The allocation, scheduling and translation sub-problems are all interdependent. For example, two operations which use a similar operator could share it given that they do not occur concurrently, whereas if they do occur concurrently then the operator must be duplicated. The sub-problems may be done in any order or simultaneously, the method chosen will depend on the optimisation strategy adopted by the system.

Binding fixes the result of a synthesis process. In allocation a data path unit is bound to a physical unit, while in scheduling the operations are bound to specific times. Binding can occur during or after a process. For example, if allocation is performed one unit at a time and once only then the binding may occur with the allocation, however if a method such as linear integer programming is used the binding cannot occur until after. Where binding is done in the language (as in the case of many structural specifications) the appropriate synthesis process is not performed; this makes for a simple compiler but restricts the compiler's optimisation opportunities. Language bindings give the user the ability to perform area-speed trade-offs. An example of this is the explicit specification of parallelism in MacPitts [11].

To generate architecture we require to bind elements to structural components and operations to control states. The bindings performed are:

a. Operations to control states,

b. Operators to functional units,

c. Variables to storage elements, and

d. Nets to interconnects.

1.5 OPTIMISATION

Optimisation is important as an implementation generated using direct compilation from the design specification is likely to be far from optimal in all aspects of the design. There are two reasons for this: Firstly, the language used to describe the design may be limited therefore additional input is required. Secondly, the designer is unlikely to write optimal input descriptions which will synthesize to an optimal implementation.

There are good reasons for not writing optimal design descriptions, such as, increased readability, changes in the designer's constraints and requirements and last minute corrections to the design. Design alterations are inevitable after functional simulation and are part of the design process. The alterations are rarely elegantly included in the original description but added by rather Heath Robinson methods. The changes are non-optimal and usually include the creation of a large number of temporary registers, most of which are superfluous.

The optimality of a design depends only on the designer's interpretation of what the optimum implementation should be.

There are two types of optimisation, global optimisation and local optimisation. A local optimisation produces the best result at a local region of the design, for example, combining two control states or changing the implementation of a data path unit. A global optimisation is one that is the best result for the whole of the design, for example, an allocation of storage such that the total number of units is minimised. Local optimisations can be used to provide simple incremental improvements to the design, however, this inevitably leads to local minima and a different sequence of optimisations could produce a better result. Global optimisations investigate trade-offs between differing results using methods such as linear programming and clique partitioning.

Local minimum traps can be overcome by guiding local optimisations using a cost function that takes into account the global aspects of the design. This will lead to a global optimisation <u>only</u> if backtracking or design degradation is provided and local optimisations do not bind design decisions, that is to say, they may be overruled by subsequent design decisions.

Backtracking is done when it is found that an optimisation performed earlier in the synthesis process has resulted in the design not achieving the user's requirements. The synthesis is reversed (backtracked) to the point where the offending optimisation was performed, it is then reapplied in a different way that will hopefully produce a better final result.

Design degradation is achieved by transforming the design so as to produce a worse design with respect to the user's requirements. This allows an optimisation to be applied that was previously restricted by the design's position in the design space. Backtracking differs from design degradation in that the design is degraded to a previous position in the design space whereas the latter degrades the design to a possibly new position in the design space.

Optimisations commonly performed include the elimination of registers for temporary variables which are stable over their use, loop unrolling, operator sharing, extracting parallelism and serialisation.

1.5.1 OPTIMISATION STRATEGIES

As the synthesis sub-problems are interdependent, the order in which they are performed greatly affects the resulting implementation. For example, a binding of two operations to the same control step prevents any sharing of their operators; conversely, binding two operators to one functional unit prevents the operations being performed concurrently. A fixed line of reasoning can often lead to inferior results as trade-offs are not explored [30].

There are two approaches to optimisation either iterative, by synthesizing a correct solution and iteratively transforming it to optimise the objectives, or constructive, by performing optimisations as the design is constructed.

In an iterative optimisation scheme, a naive implementation with no sharing of units or control states is synthesized from the input description. The implementation is then improved using translation steps to transform the design. The methods used include knowledge based expert systems, heuristics and brute-force. An advantage with using an iterative approach is that the optimisation process can be terminated before the optimal design, and still result in a correct solution.

In the constructive optimisation approach the implementation is constructed in such a way as to meet the designer's objectives; if they are not reached then the design is re-constructed with different control parameters [6]. Strategic serialisation is a common constructive approach used in many systems (see Chapter 2). It consists of creating an ASAP schedule and re-scheduling subject to resource constraints. Algorithmic techniques such as mathematical programming and clique partitioning are usually applied to single synthesis tasks with little consideration for other tasks. Constraints can be used to give subsequent tasks an opportunity to achieve their targets. Although they can guarantee optimal solutions to the synthesis sub-problems to which they are applied, they are also NP-complete. This is made worse by any backtracking or re-synthesis if the user's objectives are not met.

1.5.2 Cost Function

A cost function represents the state of the design within the design space. A cost function is required in an intelligent silicon compiler which must use it to make decisions on how the design should be optimised. Many systems which optimise with respect to only one criterion follow a pre-defined optimisation route where the optimisations are programmed into the synthesis algorithms. For example, Scholyzer [20] maximises speed and where there is a choice between functional units the smallest is chosen. These systems do not use a cost function to direct an optimisation algorithm as the synthesis tasks are aimed specifically at one optimisation objective; for example, Scholyzer's synthesis tasks are aimed at extracting parallelism. However these systems may use a cost function to inform the designer of the design's "goodness" or a local one to aid bindings.

A cost function is used in conjunction with the optimisation criteria to guide the synthesis process. It can help select the sub-process to perform next or help in the evaluation of trade-offs between processes. Trade-offs are essential for producing optimal designs when optimising more than one aspect of the design. If allocation and scheduling are performed separately trade-offs can not be made, therefore the interaction between or the simultaneous execution of the sub-processes is vital.

To produce real circuits it is important to be able to specify constraints and target objectives on various aspects of the design. These criteria should be in absolute units and not in terms of data path unit counts or the number of control states. Characteristics of the design must be determined, for example critical paths, which are used to aid the evaluation of the cost function and so direct optimisations. For example, the delay can be reduced by extracting parallelism on the critical path and area reduced by sharing units off of the critical path. McFarland [31] demonstrates how an inadequate cost model can distort the design space and make implementations seem optimal. He shows that a complete cost model taking into account aspects such as wiring is important in making the correct design decisions.

For systems that allow the user to specify various absolute target objectives the cost function must be accurate so that the targets may be met with confidence in the final implementation. To achieve this, technology dependent information must be fed up to the synthesis system for use in the calculation of costs such as data path area, power and delay. Layout factors such as interconnections have been shown [31] to have a significant effect on hardware costs and not taking them into consideration will result in a poor design. These can be estimated from the operations [32], however, accurate costs can only be obtained after layout is complete [6]. Constraints are often added to the cost function by penalising configurations which violate them.

1.6 HARDWARE SYNTHESIS AND LAYOUT

Hardware synthesis is creating a detailed implementation from the internal representation and producing the required output, for example a netlist of parameterized cells. This involves synthesizing the data path (data path synthesis) and the controller if the architectural model includes one. Control synthesis selects the clocking scheme and generates the controller according to the architectural model. Control generation has to be done after data path synthesis as control signals are unknown [6].

Layout of the design is done by conventional placement and routing methods. In some compilers layout is done according to a fixed floor plan as in First [12] or columns of fixed width cells between which are wiring channels as in the CAL-MP system [33] used by Scholyzer [20]. General layout methods can be guided by a floor plan devised by the synthesis system. This method is used in the Camad [21] system and although the computational cost increases it can be justified by the availability of accurate wiring costs for a design. Layout costs, such as wiring, can be fed up to a synthesis system, or alternatively constraints on wiring lengths can be passed to the layout tools in order to achieve predictable performance characteristics [17].

1.7 DESIGN SPACE EXPLORATION

The design space is an n-dimensional space, where n is the number of different aspects of the design monitored by the designer. For example, a 2-dimensional design space might consist of area and time as in Figure 1.1. Each design can be represented in the design space by a point, with better designs being closer to the origin. The design space is divided into two regions which correspond to design points which are either achievable or un-achievable. The curve separating these regions is the *optimal design curve* which asymptotically approaches minimal values for aspects of the design as others approach infinity, that is, a minimum value for a single objective. For a design space with two or more dimensions the design which is considered optimal will depend on the designer's objectives. For example, if the designer requires the design, represented by the design space in Figure 1.1, to be the fastest given that the area should be no more than $1000\mu m^2$ then the optimal design is given by the intersection of the optimal design curve and the line area=$1000\mu m^2$.

A design is considered optimal, and therefore lies on the optimal design curve, if no other design exists with a better value for each dimension in the design space. For a two-dimensional design space, a design is optimal if no other occurs in the region to the south-west of it. The optimal design "curve" is actually discrete as the implementations consist of discrete components, therefore, the set of optimal designs (those lying on the curve) is finite. The design space can be characterized by specifying n+1 points lying on the curve, where n is the number of dimensions in the design space. The points consist of the n asymptotes, representing the best implementation for a single objective and the

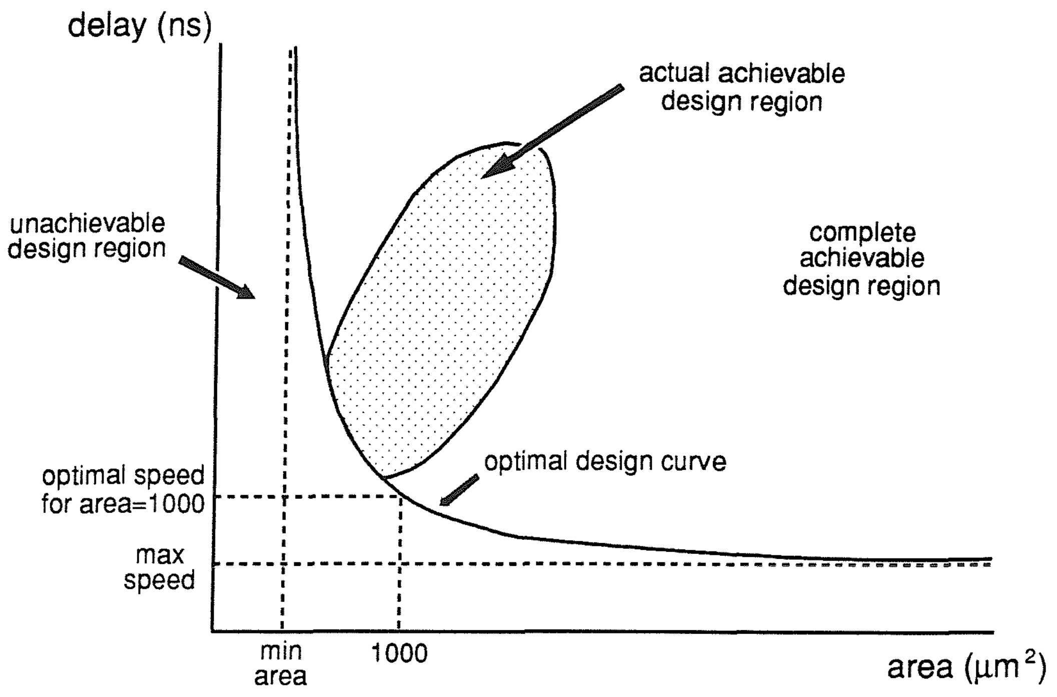


Figure 1.1 The area-time (AT) design space.

point closest to the origin, representing the best overall implementation. Therefore a two-dimensional area-time (AT) design space can be characterized by three points, the two asymptotes indicating the minimum area and minimum delay designs and a point closest to the origin representing a compromise design. The lower bounds of the design space, the asymptotes, can be predicted [34]; therefore without performing any synthesis an indication is given of whether the designer's goals can be successfully met.

In reality only a proportion of the points in the achievable design region may be obtained as indicated by the *actual achievable design region* in Figure 1.1. The limitation in design space may be due to a number of factors, which include dependencies between design aspects, early design decisions being bound in the synthesis system and limitations in the optimisation process caused by sequentially performing the synthesis tasks or assumptions made in the internal representation. Selective early binding of some design decisions can effectively reduce the search space for an optimal design however premature binding can restrict the process inappropriately.

The possibility for the designer to investigate different implementations of one specification is called design space exploration. Some systems claim to perform design space exploration by changing the design constraints and re-synthesizing, however this is an ability possessed by all synthesis systems using constraints. By allowing some randomness in the synthesis tasks different implementations may result without adjusting constraints. For manual optimisation systems, design space exploration occurs as a consequence of the user changing the design, however he has no knowledge of the optimality of the design. For example, Emerald [35] explores the design space by manual manipulation of its intermediate code; the Value Trace. The exploration of the design space should be an automatic process such that a set of designs are found which characterize the design space for a given design. An intelligent compiler aims to provide the user with an insightful characterization of design alternatives as well as the optimal design (trade-off) curve [3]. In addition to illustrating the range of designs that can be achieved from one specification the design space is an important tool in the design of synthesis systems as it can graphically show the effect of changes to synthesis algorithms.

1.8 PROJECT OBJECTIVES

The objective of the project is to implement an "intelligent" silicon compiler that provides the ability to optimise a design, given as a behavioural description, with respect to multiple objectives. The user submits goals or objectives to the system which automatically finds an optimal solution in the context of the user specified constraints [36]. As the objectives may be conflicting, trade-offs between synthesis tasks are essential and consequently simultaneous execution of the tasks must occur. The absolute state of the design within the design space must be accurately represented using a global cost function. Therefore technology specific data is required so that optimisations performed at an abstract level can be guided by a cost function that takes into account low-level details. The compiler is more accurately described as a high-level behavioural synthesis system as layout is not performed.

The compiler must also include an efficient method for automated design space exploration allowing the user to investigate alternative designs on the optimal design curve. The use of design space exploration allows the designer to obtain a perspicuous characterization of the design space for each design and thus determine whether a design can satisfy a variety of simultaneous constraints.

2 Literature Survey of High-Level Synthesis Systems

High-level synthesis research can be divided into numerous categories. The following literature survey details many high-level synthesis tools and systems, collectively known as silicon compilers. The categories used here are based on architectural or application restrictions and optimisation abilities. Each reference is accompanied by its year of publication in order to chronicle them. The description of a compiler is preceded by its name or author in emphasised text to allow ease of future reference.

2.1 EARLY SYSTEMS

A literature survey on high-level synthesis systems would not be complete without a description of three well known systems, Bristle Blocks [37] (1979), MacPitts [11,38] (1982,3) and First [12] (1983). **Bristle Blocks** is essentially a cell layout tool. A hierarchical structural description, where the lowest level are library cells, is directly compiled to a fixed layout. The cells are procedural and can be dimensionally re-configured to fit a regular fixed layout. The controller is a micro-controller or Turing machine where the micro-code, word width and encoding are supplied by the user. The user also defines buses and core elements which are bound in the description.

The **MacPitts** system also has a micro-program architecture. Other restrictions include a fixed width data path and limited parallelism. A micro-programmed machine is generated to implement the required parallelism bound in the description and consists of a counter and control logic implemented as a Weinberger Array. The FSM is analyzed and similar units controlled from different states are merged (including mutually exclusive ones). Registers are bound in the description and can not be shared.

MacPitts and **First** use a bit slice architecture. First uses a data path only pipelined architecture where operator synchronisation and delay insertion are supplied by the user, thus increasing description complexity. A fixed floor plan consisting of two rows of operators with a central wiring channel is used.

The Bristle Blocks, MacPitts and First systems use structural descriptions with considerable language bindings, therefore allowing no or little (in the case of MacPitts) optimisation. The systems are akin to module assemblers and use parameterized cells or module generators; a property common to all high-level synthesis systems. The use of an intermediate representation compiled from the source language and the separation of layout from synthesis are common to most subsequent high-level systems. These three systems also have a fixed architecture rather than a general one.

2.2 ARCHITECTURE SPECIFIC SYSTEMS

Architecture or application specific systems have the advantage of a restricted design space which can simplify the system and optimisation process, however, a knowledgeable designer is required to select the correct system. The synthesis community is divided, with some believing that a general behavioural synthesis system which produces good optimised designs is not possible [1]. They choose to create a collection of specialised systems each aimed at a particular application or architectural target. One such set of systems is the **Cathedral** silicon compilers [39] (1988). Cathedral I is for bit serial filters, II is for multi-processor based architectures with regular interconnect and synchronous data passing protocol and III is for bit slice architectures. Cathedral is restricted to a fixed clock and constant time functional units therefore simplifying design trade-offs. Hardware resources are set prior to scheduling thus minimum delay for a given set of resources is the optimisation objective.

The **Appolon** system [40] (1985) uses pre-defined architectural templates. The synthesis steps performed include: architectural design consisting of simplifying complex operations and scheduling them, one to one operator allocation, PLA/ROM based controller synthesis and manual design of miscellaneous parts. The data path is a bit slice structure and the parallelism bound in the description determines the number of sub-operative parts, where an ALU is generated for each one. Both iterative and constructive methods of data path synthesis were studied.

A pipeline architecture is the subject of other systems such as **Sehwa** [32] (1988) which uses a similar method as the Maha system (described in Section 2.5) from the same author. Both systems find designs between the highest performance and lowest cost. The system described by **Hartley and Jasica** [5] (1988) generates a complete pipeline style where all operations are in parallel, therefore scheduling and resource sharing are not required. The input is behavioural which is translated to an intermediate level that identifies structural elements. The order of operations is determined from data dependencies which can not be cyclic. After delay insertion the structure is laid out using a standard floor plan. A linear program is used to insert delay cells to ensure proper synchronisation of operator inputs. The minimum number of delay cells to be added and so least area increase is determined by moving operations in their time frame thus allowing input and output delays to be combined.

Another pipeline system described by **Choi** [41] (1992) uses a similar process to force directed scheduling (see Hal in Section 2.5). This system determines a set of feasible functional units. Critical path units are scheduled and delays are inserted to avoid resource conflicts in parallel critical paths. The remaining operator types are iteratively scheduled; where the order is determined using a measure based on the *force* and density of operations in a control step.

Pipelining is not limited to architecture specific systems, for example, Devadas and Newton [42] (1989) is a general system which has been extended to allow pipelining. Similarly MOODS could produce pipelined architectures by extending the intermediate code and using an appropriate cell library.

The above systems are application or architecture specific and are therefore not comparable to MOODS which has a general distributed architecture aimed at a wide range of applications. The above systems do however have similar limitations to the general ones described below.

The synthesis process is divided into a set of synthesis tasks, as described in Section 1.4. Due to the interaction of synthesis tasks the way in which they are performed has a significant bearing on the type of optimisation possible and the resulting designs. The remaining compiler descriptions are categorised on their optimisation abilities; starting with those that minimise one aspect of the design and ending with those that optimise a number of criteria subject to user constraints. The former can only be classed as synthesis tools as they do not perform all of the synthesis tasks, whereas the latter can be classed as systems of which the MOODS synthesis system is a member.

2.3 SYSTEMS WITH A FIXED MINIMISING CRITERION

The CMU-DA suite of tools, of which the **EMUCS** [43] (1983) tool is one, use the Value Trace (VT) intermediate code. The Value Trace consists of a set of directed acyclic graphs (DAGs). The nodes represent operations and arcs are carriers representing data flow. Each DAG may form a branch of a *select* operation representing IF or CASE constructs, that is, a DAG is a block of instructions to be executed having one entry point and one exit point. Each block is represented as a two dimensional list having all operations in a horizontal list executed at the same time step, where the horizontal lists are chained vertically in the order of execution.

The EMUCS tool allocates hardware and attempts to find the minimum cost design given a Value Trace which has previously been scheduled. The cost can be any quantitive feature (but is typically area) which is approximated by a weighted sum based on the technology. The operators given in the scheduled VT input are iteratively bound to abstract cells in order to gradually construct a data path. At each iteration cost tables are constructed and an analysis made to decide what to bind. The cost tables reflect the feasibility of binding each element by specifying the cost of changing an existing functional unit to accommodate an additional operation. The operation to be bound and to which unit, is selected using a min-max approach that attempts to minimise the final cost by minimising the potential loss at each iteration. Thus the least costly operation over a number of steps is bound based on minimising the incremental cost associated with the cell being considered. Only functional units with equal bit widths can be merged. This cost based greedy algorithm uses no global views of the design and allows little backtracking; consequentially the result is only a local optimum.

Another allocator from CMU (probably the predecessor to EMUCS) is described by **Parker and Hafer** [44,45] (1978,1982). This tool first allocates storage and I/O followed by the allocation of functional units one by one. If an appropriate unit exists in the data path it may be shared otherwise a new one is entered. Multiplexers are then created at unit inputs where required. Allocation is in terms of generic components and is specific to the style of data path used, either bussed or distributed.

The EMUCS tool minimises one aspect of a design, its cost, by performing one synthesis task; allocation. The following systems also minimise one aspect of a design, either area or delay but in addition use limited heuristics to improve a second aspect of the design.

Facet [26] (1983) is another allocator tool, again part of the CMU-DA suite of tools. Facet applies clique partitioning to the synthesis tasks to minimise either storage, interconnects or operators by forming special ALU groups. The clique partitioning algorithm uses the common neighbourhood property to produce near minimal cliques. A graph node is a neighbour to another if an arc exists between them. If a third node is connected to two neighbouring nodes then it is a common neighbour of the pair. A VT input can be compacted to form an ASAP schedule by moving operations to the point where their inputs are defined. Graphs are formulated for each synthesis problem from the VT and clique partitioning applied. In register minimisation for example, a compatibility graph is constructed using lifetime analysis, where nodes represent registers and arcs join mergable pairs. Register minimisation gives priority to combining registers with pure data transfers between them as this increases speed and also reduces area. VT compaction is repeated after register clustering. The synthesis tasks are sequentially applied to the design in a fixed order with no communication between them therefore the design space is not explored.

The **Emerald** system [35,46] (1984,6) allows the user to perform initial VT code compaction by either serialising operation pairs or moving an operation to another VT block. Other VT changes involve transformations such as converting instances of a counter to an adder or local incrementer. In this way alternative data paths can be achieved. The Facet tool is used by Emerald to synthesize the data path, where design costs are represented in terms of component counts, bits or gate counts.

The **Silc** [4] (1985) system is similar to MacPitts. Silc performs placement and routing and has storage and FSM states (scheduling and two level parallelism) bound in the description. The Silc chip is a collection of FSMs communicating by an asynchronous protocol; the first level of parallelism. The second level of parallelism occurs through each state controlling a set of operations. Heuristics are used to improve the FSM logic in the sum of products form. As with MacPitts, Silc shares functional units that occur on different FSM states and takes mutual exclusion into account. However, In addition Silc makes unit choices, for example between a carry look ahead adder and a ripple carry adder, based on speed.

The **Scholyzer** system [20] uses the Scholar language [47,48] which is compiled to an intermediate code (ICODE). The ICODE is used to create a control graph where arcs represent control flow and each operation is assigned to a node which represents a control state. Sequential and parallel sections of the graph are ASAP compacted subject to contention tests and declared variables. Further compaction is achieved by applying local transformations to fork and join type nodes, thus producing a maximally parallel ASAP schedule. Limited functional unit sharing is performed after scheduling where similar non-concurrent operators with common input or output variables are shared. Registers are bound in the description therefore no register optimisations are performed, however, some operations implementable using registers are detected.

The **Yorktown Silicon Compiler** [49] (1987) from IBM is an advancement from its **Elm** [50] (1983) logic transformation system which provides an expansion of generic operators to primitive boolean and register blocks. Elm's boolean algebra is manipulated using technology dependent transforms which may be applied using various strategies and the best result selected in light of user constraints. In the YSC the design uses modules each with its own data path and controller distributed within the data path. The data flow steps are scheduled to facilitate a sequential controller and a fast design is the objective with later optimisations to reduce area. For each functional unit a parameterized boolean expression exists. The entire boolean structure is flattened and partitioned if it is too large for the logic synthesizer. The decomposition of a design to the logic (gate) level results in a large data structure which may be computationally expensive to optimise. The references on page 36 describe recent research at IBM which has concentrated on high-level synthesis and design space exploration.

An improvement on the above method of minimising one aspect followed by another is to take the second aspect into consideration whilst performing the initial minimisation. This can be done by considering operator similarities or critical path and freedom measures. The system described by **Girezyc and Knight** [29] (1984) considers the critical path. It allocates a previously functionally optimised control and data flow graph using a greedy algorithm and makes hardware assignments by adding cells composed of a register, multiplexer and operator. During allocation the critical path is considered so that the previous timing constraint is maintained. Both critical path and operator similarity are utilised in the system described by **Hong, et al.** [51] (1987). Scheduling uses a data flow graph with arcs specifying dependencies. The critical path is determined and scheduled ASAP. Non-critical operations are then scheduled and an attempt made to reduce concurrency between similar operations by delaying their schedule. Pure data transfers are scheduled ASAP. Initially a path search algorithm using lifetime analysis is used to group variables into registers. After scheduling, operations are clustered into functional units taking into account variable clusters found previously.

2.4 SYSTEMS WITH A FIXED CONSTRAINED CRITERION

The logical progression from minimising particular aspects of a design is to allow constraints to be placed on them. The following systems allow a constraint on one design aspect, area or delay, whilst minimising the other. Most allow an area constraint in the form of limiting the available hardware resources and strategically serialising a maximally parallel implementation until enough resource sharing can be performed in order to meet the resource constraint. The main differences between the methods lie in the way in which a synthesis task is taken into account by another in order to allow the constraint to be met.

One of the first systems to use strategic serialisation was **Mimola** [28] (1979). The Mimola language can be structural or behavioural both of which are compiled to a design database containing all design information. The initial design is a maximally parallel design where the parallelism is bound in the description. Hardware is allocated at each control step (micro-instruction) using a set of hardware resources. The allocator adds resources, modules and connections, one by one from the hardware set. It is assumed that the hardware can be shared between micro-instructions. If no resources are available in the set for an allocation the operation is delayed (serialised) and registers added for intermediate results, thus allowing a resource to be shared. The designer restricts the available resources to force sufficient serialisation to meet his goals. Measures are given on design aspects such as area, power, speed and micro-program requirements. To aid the designer select which resources to remove from the set, utilisation figures are determined by statistical analysis, where low utilised resources are candidates for removal.

The **Autonomy and Attraction** tools [52,53] (1981,2) use strategic serialisation in the synthesis of a micro-program controller. Given a data path they trade bit compaction for word compaction to meet a cost (area) constraint. The autonomy tool, which is applied first, uses a partitioning algorithm that isolates operations to be controlled independently of micro-instruction encoding. It makes a dedicated field in the control word for these operations therefore removing them from all sets of micro-operations. Operations are selected to increase the encodeability of the others up to the point of increasing a pre-defined word width. The attraction tool determines, subject to the constrained instruction width, which micro-operations to execute in parallel and which to encode in separate micro-instruction formats by forming clusters to minimise encoding. Operation pairs are iteratively merged into clusters using attraction weights. At each iteration the weights are calculated using the freedom measure to determine the probability of pairs of operations occurring in the same time slot. The highest pair are merged into a cluster, thus resulting in the least degradation in opportunity for parallelism.

Elf [29,54] (1985) uses list scheduling where the order of scheduling is based on urgencies. A weight is determined by taking the minimum number of cycles to execute an operation plus its maximum successor weight, that is, its ASAP time. The urgency is the ratio of weight to spare cycles to a time constraint. Therefore when the scheduling of an operator is delayed, to allow sharing, its urgency increases which raises its priority for scheduling.

The **Design Automation Assistant (DAA)** [55,56] (1985) creates an initial data path by allocating units by direct compilation using VT block minimum delay information. Unchanging items such as variables and ports are allocated first followed by functional units. A maximally parallel design is created using an ASAP schedule and the design is strategically serialised using an expert system guided by estimators. The expert knowledge is contained in 300 rules determined by interviewing real designers. The rules are mostly local, for example merging registers and are technology sensitive. The knowledge is incomplete and although more rules result in improved designs the system becomes slower. Cost estimators determine the cost of upgrading functional units and interconnects to contain an additional module. Module generators build technology dependent modules specified in DAA output.

The **Caddy** system [57,58] (1989,90) (Carlsruhe Digital Design System) also uses rules which form local optimisations, these use commutativity type axioms. Global optimisations used involve folding operators, folding variables based on lifetime analysis and loop unrolling. The optimisations are applied to data and control flow graphs which are directly compiled from DSL a Pascal like language [9] where unnecessary registers and data transfers have been removed. Some explicit control and parallelism is bound in the language, however additional parallelism is extracted. The optimisations are applied in order to minimise area subject to timing constraints. Scheduling is performed by list scheduling using resource limits and freedom, taking into account mutually exclusive instructions. Register minimisation is done separately using a graph colouring approach. A graph colouring algorithm finds the minimum number of colours such that each node when coloured is not adjacent to a similar colour. After optimisation a one to one mapping using parameterized structure generators is performed. The output is a hierarchical netlist which provides an interface to other Caddy tools.

The **S(p)licer** tools [27] (1986) also use a resource constraint, which is used to guide scheduling. Slicer creates a preliminary ASAP schedule and determines critical paths and operator freedom using unit times based on the fastest units. An optimised schedule is created a state at a time by binding operations to functional units from each ASAP state starting at the first. The operations in an ASAP state are ordered on increasing freedom (critical path first) and each assigned to the new state until the resource limit is reached for that state; a new state is then created. The Splicer tool uses a greedy algorithm to assign structural components, sharing where possible. A depth first branch and bound method is used to find a fair solution quickly and subsequent best solutions are retained.

Raj [59] (1986) describes a system which uses operator similarities. Operations are assigned to micro-instructions (conceptually similar to control steps) to form an ASAP schedule. Operations are delayed where similar functional units could be shared, thus scheduling the operations in different time steps. A hardware allocator then uses a greedy method which allocates operations to hardware one at a time, sharing where possible or creating additional hardware if sharing is not possible. The allocation continues until it is complete or a hardware constraint is reached. If the constraint is reached, the user must move an operation to a succeeding state and re-allocate.

2.5 SYSTEMS WITH A CONSTRAINED CRITERION

Similar strategic serialisation methods to those described above are used in the following two systems which allow goals to be set on either cost (resource constraint) or speed.

The **Maha** system [24] (1986) creates a data flow graph and assigns delay and area figures to elements based on the average of those components capable of implementing the operations. Using the delay figures the operation freedoms and critical paths are determined and a lower bound on resources and delay can be found. The data flow graph is divided into *n* equal time steps, initially equal to the maximum functional unit delay, and the timing constraint checked for violation. Operations are scheduled in order of increasing freedom thereby allowing greater flexibility for unit sharing as operations with a greater freedom can be scheduled later. Critical path operations are allocated on a first come first served basis, sharing where possible. Possible states for non-critical operations are examined in sequence and the earliest chosen with priority given to sharing units. If the resource cost is exceeded the data flow graph is divided into more steps, *n* increased and scheduling repeated. To minimise the cost *n* is increased up to the point prior to timing constraint violation therefore allowing maximum unit sharing. To minimise the delay resources are added up to the point of cost violation therefore allowing maximum parallelism.

Hal [25,60,61] (1986,7) also allows goals to be set on cost or speed and performs allocation and scheduling separately but not independently. Hal performs a default allocation and ASAP scheduling similar to Slicer and Maha from which freedom figures are determined. The timing constraint (if specified) is used to decide on the control step partition to which operations are scheduled. The order in which to schedule operations is not determined by freedom as in Maha but by *force* which attempts to balance the distribution of operations that make use of similar resources between control states. The force is determined thus: for each operation type in each control step the sum of the probability that the operation is scheduled in that step is taken. For a given operator the place to schedule it is determined using forces corresponding to the change in probability between steps. The smallest force of all operations is scheduled in its time slot and the process repeated excluding the already scheduled operations. After scheduling, locally optimal hardware allocation is performed to find a set of cells to implement the data path subject to the goals. Cells may implement more than one function. Lastly, register optimisation and binding of cells taking interconnects into account is performed.

Recent research at IBM by **Camposano** [26] (1990) uses strategic serialisation to determine the area-time (AT) trade-off curve. Camposano describes a method of creating a set of designs, in order to explore the design space, by initially generating a maximally parallel "as fast as possible" (AFAP) design and then introducing extra control states in order to allow more hardware sharing and thus navigate the design along the AT curve in the design space. An extra state is created by splitting an existing one such that the area of the operations in each state are approximately equal. It is important to note that the area considered is that of the data path only; interconnection, storage and control areas are not considered and the delay is taken to be the number of control states. The state splitting process does not take into account the critical path therefore the speed may be degraded by splitting critical path nodes before non-critical paths have been examined; thus resulting in a non-optimal AT curve. **Camposano and Bergamaschi** [27] (1991) use a data path graph with control constructs which is compacted to reduce path lengths. An AFAP schedule is created by moving operations into parallel branches. Operations may be duplicated thus allowing one operation to be scheduled in more than one control state. Although an innovative idea it would be of little use in the MOODS system because if the operation can be scheduled in more than one control state then the states would be merged thus reducing the controller.

2.6 DESIGN ENVIRONMENTS

Some of the tools and systems described above have been incorporated into larger systems which allow the designer to control the design process. By changing the order of the synthesis tasks or by applying transformations to the initial description the designer can produce a variety of designs, thus exploring the design space. The approach to high-level synthesis by CMU is to develop a collection of synthesis tools such as those CMU-DA tools described above and those described by **Balakrishnan, et**

al. [64] (1988) and **Tseng and Siewiorek** [65] (1981) which provide the grouping of variables into multi-port memories and the synthesis of buses respectively. These tools can then be manually controlled from one system **[13,14]** which uses a global database (GDB) that is derived from a register-transfer level language (ISPS) used to describe the designs behaviour. A common graph structure, the Value Trace (VT), which is able to describe the design at various linked levels of representation, is determined from the GDB.

The **Sugar** system [66] (1985) from CMU uses heuristics to automate the application of VT transformations previously applied interactively [13]. The transformations that can be applied to the VT are, for example, dead code elimination, partitioning, folding, code motion, inline expansion, flattening of nested conditionals and pipeline formation. Code motion shapes the VT using user's goals so that hardware binding will give a design which meets the goals. The transformations help create a more synthesizable VT with less bias introduced in the designer's description. Sugar groups increment and decrement operations with other arithmetic operations. It uses a "least commitment" style of binding design decisions to reduce negative effects of interactions between synthesis tasks. Early tasks may be used to gather information and then redone later in the design process.

The **System Architects Workbench** [15] (1987) is the latest generation of the CMU-DA system and is a group of tools that operate interactively on the VT compiled from ISPS. The VT can be manipulated using the same code transformations used in Sugar, after which scheduling and allocation is performed by CSteps and EMUCS respectively. The Systems Architects Workbench includes **Aparty** [67] (1991) an architectural partitioning tool which determines subsets of behaviour to implement on separate chips. Aparty uses multi-stage clustering techniques and communicates information to other synthesis tools.

Another design aid environment is **Spaid** [19] (1989) which provides tools to explore architectural alternatives for DSP applications. The designer can specify hardware resources and timing constraints such as throughput or latency. Transformations can be applied to the initial description to utilise operator properties, for example the association property, or to satisfy timing constraints, for example re-timing. Transformations and resource selection can be performed in any order to allow exploration of design alternatives. Spaid then performs an initial schedule and allocation which is repeated depending on unit utilisation analysis. Lastly registers and multiplexers are minimised.

The **Adam** synthesis system [30] (1991) is a framework manager for design tasks and is unique as it does not create a design but constructs a design plan which is later executed, if the constraints are met, using tools such as Sehwa and Maha. Initially the system estimates the lower bounds of the area and delay based on maximum usage of operators in the data path. This is similar to the prediction of design space bounds made by Parker, et al. [34]. The estimate is used to decide on a module set and design style thus reducing the design space. Heuristic rules are used to guide the tool choice and select tool parameters. A plan graph is constructed where the root is the initial design and pre-conditions aid the construction which follows a depth first search method. The properties of each terminal state are measured and depending on the result tool parameters are adjusted or the planning continued until the goals are met. A range of designs can be produced with the designer's interaction. A multi-dimensional cost function is possible, however, as estimates are used in the design plan accuracy would be jeopardised.

2.7 INTELLIGENT SYSTEMS

The systems reviewed so far optimise a design either by minimising one or two aspects of the design or maintaining a constraint on one aspect whilst minimising the other. If a goal is not reached and the synthesis task fails, human interaction is necessary to adjust the available resources, in the case of strategic serialisation methods, or alter the initial description in methods which assume language bindings. These systems can therefore be considered open loop systems. The remaining systems described below close the design loop and allow trade-offs between different design aspects; therefore they are often classed as *intelligent* compilers. Although trade-offs have been seen in for example the strategic serialisation systems, the trade-offs being performed were only *one way*; that is, sacrificing speed in order to improve cost or area. In those systems no provision is made for automated backtracking, design degradation or re-design if a synthesis task fails.

The **Ulysses** system [68] (1985) provides the simplest way to close the design loop. Ulysses is an interactive tool integration environment for the CMU-DA tools which contains its own design data representation and translators between existing synthesis tools. It contains knowledge of which tool to activate when a conflict occurs thereby providing automated re-design. A similar approach is used by **Chippe** [69] (1990) in which trade-offs are iteratively corrected using analysis of the constructed designs; therefore it is never too late to change a design decision. The design is analyzed with respect to the goals on area, power or delay. An appropriate action is taken by modifying the design through changing the parameters and constraints (primarily resource constraints) passed to the design refinement tools, which re-iterate one or more of the synthesis tasks. The refinement tools used are Slicer and Splicer. The evaluator gives area, power and delay measures as well as unit usages, unit dead time, unit overlap and critical paths. Tool parameter changes are determined using rules which are scored by the evaluator and the highest selected. Although the strategy is still one of generating a maximally parallel design and then strategically serialising it Chippe can, by using its rules and evaluator, provide more intelligent tool parameter choices and optimise to a wider variety of design aspects.

Ulysses and Chippe perform re-design by global iterations, that is, complete synthesis tasks are iterated. This is necessary because they utilise existing synthesis tools which are incapable of *two way* trade-offs. Systems which use local iterations must allow two way trade-offs and therefore perform the allocation and scheduling tasks simultaneously. Four systems use this approach, one of which is MOODS; the other three are described below.

The **Camad** system [21,22,23] (1986,7) produces an initial implementation using a timed Petri-net model which is compiled from a Pascal-like description. The architecture is an asynchronous one where tokens are held or delayed long enough for the operations to be performed. Token firing conditions are used for conditional branches and synchronisation transitions are used where a node depends on more than one predecessor. This model is the asynchronous equivalent to the synchronous controller model used in Scholyzer and MOODS. The design is measured using a set of matrices which represent the design aspects and a vector of scales representing the priority of the matrices. Goals can be placed on both area and delay. The initial implementation assumes maximum resources and parallelism. The optimisation minimises the difference between the goals and the design measure by iterative improvement. The critical paths and signals are used to choose which sub-part of the design to transform. If the performance goal is not reached then the critical path is shortened either by parallelising operations or by shortening token hold times, that is, operation execution times. Likewise if the area goal is not reached units are shared preferably off of the critical path and operations serialised to facilitate the sharing. The algorithm is a complex tailored heuristic one which requires a complex flagging system to prevent transformations undoing previous improvements and thus alternating. By using the critical path the algorithm is biased towards performance. Design partitioning can be performed and is taken into account during optimisation.

The system described by **Devadas and Newton** [42] (1989) also uses local iterative improvements which are applied to the design using a simulated annealing algorithm. The design is compiled from a 'C' description and entered into a two dimensional grid where control states are represented by the rows and each row contains a number of parallel items to be executed in a given state. Each item may include one or more chained operations. The synthesis tasks are thus formulated as a 2D placement problem. The iterations consist of generating additional rows subject to dependencies and moving operations within the grid. The design cost is determined using estimated measures; for example, a measure for the number of buses required would be the maximum number of distinct sources and the number of sinks in all time slots. The cost measures are summed and scaled by weights specifying their relative importance, as also done in Camad. Constraints are included in the cost function by penalisation. The annealing end condition is when the cost function has not changed for three consecutive temperature steps. The number of moves per temperature step is reported to affect the solution profoundly, however no mention is made of how the user determines it!

The simulated annealing system from **Safir and Zavidovique** [70] uses a similar cost function to Devadas and Newton, however, time is only represented by the number of states. The cost function also incorporates a "silicon surface time utilization" (STU) measure which is the ratio of the sum of silicon area used during each machine cycle to the product of total area and number of machine steps. The STU measure is similar to the unit utilization measure which forms part of the design goodness measure in the MOODS system. The iterations consist of shifting nodes of a control and data flow graph between time steps. A parallel task allocates generic components. As with Devadas and Newton no details are given on how the annealing schedule is determined.

2.8 SUMMARY OF HIGH-LEVEL SYNTHESIS SYSTEMS

All of the above systems suffer from one or more of the weaknesses listed below, all of which the MOODS system attempts to overcome. Apart from these weaknesses one major problem has been highlighted by **McFarland** [31] (1987) which is concerned with the shape of the area-time curve. The AT curve is assumed to be a relatively smooth trade-off curve running from the high cost, high speed designs to the slower less expensive designs. This assumed shape is a result of the assumption of area being synonymous with data path operators and speed with control path length. This basic cost model is used by many synthesis systems from which the traditional data path versus control path trade-offs are made. McFarland has shown that by using a more comprehensive cost model which takes into account interconnect area, control hardware and multiplexers the AT curve takes on a completely different shape making the assumed trade-offs less valid. Therefore systems which include trade-off assumptions pre-programmed in the optimisation process are likely to produce non-optimal designs.

An example system (one of many) demonstrating each of the following weaknesses is given, followed by a brief description of how the MOODS system overcomes it.

1. Architecture or application specific (Cathedral) - MOODS is aimed at general applications and uses a distributed data path plus controller architecture.
2. Use of language bindings and structural descriptions (Bristle Blocks) - The input to MOODS is a behavioural one where no bindings are assumed. Parallelism and variables may be defined in the description but are not necessarily adhered to. Their only effect is to determine the structure of the initial un-optimised implementation.
3. Computationally expensive and a run time explosion with design size (DAA) - The MOODS run time is controllable and can yield a correct design if terminated early. A trade-off can be made between the run time and design quality.
4. Limited design model, that is, synthesis tasks are applied in a fixed sequence which limits the design strategy and results in pre-defined trade-offs (Maha) - MOODS

iteratively applies transformations using a stochastic process to simultaneously perform the allocation and scheduling tasks. Therefore no fixed design model is used.

5. Local rather than global minimum is found (EMUCS) - MOODS avoids local minima by using reversible transformations allowing the design to be temporarily degraded and thus climb out of local minima. The global minimum (or near minimum) is found with the aid of a global cost function. Due to their limited design model many systems do not have (or require) a cost function as trade-offs are pre-programmed in the optimisation process.
6. Goals limited to one or two aspects or minimised aspects (HAL) - Again this is due to the limited design model and the lack of design evaluation through a cost function. By using a global cost function and stochastic process, which contains no pre-programmed trade-offs, MOODS can synthesis to any design aspect. The complex interaction between design aspects and their resulting trade-offs are another reason why more than two goals are rarely optimised using tailored heuristics. Encapsulating the trade-offs in an algorithm results in a complex set of heuristics as for example in the Camad system.
7. Design decisions made too early which require re-design to correct (S(p)licer) - MOODS uses reversible transformations therefore it is never too late to correct an inappropriate design decision.
8. Inaccurate evaluation or estimation of the design, that is, design goals are based on unit or bit counts rather than real quantities and do not consider control or interconnect factors (Spaid) - Although MOODS does not currently take interconnects into account there would be no algorithm changes to do so as the interconnects would be part of the cost function. Real design quantities are used to constrain the design, for example, area in microns rather than bits and delay in seconds rather than control states. The design quantities in MOODS are produced by feeding up technology dependent information to the design evaluation procedures. The use of real quantities gives the designer realistic information; it is unlikely that he is concerned with the type of resources utilised but would like to know if the design will fit the chip die.
9. Restricted control model, that is, the optimisation process is bound to one controller type which again pre-defines the trade-offs (Silc) - The costs associated with the MOODS controller implementation are included in the cost function therefore changes to the controller style are reflected in it and thus taken into account

during optimisation. No restrictions on parallelism are made, such as limited depth, therefore other controller styles can be used.

10. Limited design space exploration - Due to fixed optimisation strategies and trade-off assumptions most systems can only provide design space exploration through manual intervention; for example, by changing the design description or available resources. The MOODS system can automatically explore the design space and provide a varied set of implementation from one description; a feature possessed by no other system.

The closest systems to MOODS are those described by Devadas and Newton, and Safir and Zavidovique, however, there cost functions are fixed, whereas, the MOODS multiple objective cost function is specified by the designer. Devadas and Newton estimate the cost of a design and Safir and Zavidovique approximate speed to the number of time steps; both of which introduce opportunities for errors. MOODS however, uses technology dependent information fed up from a cell library. The use of technology dependent information means that variations in trade-offs caused by technology variations are also taken into account. MOODS also provides a wider range of transformations similar to those in Camad, however they are not applied using pre-programmed trade-offs as in Camad.

3 DEVELOPMENT OF THE MOODS SILICON COMPILER

As mentioned in Section 1.8 the purpose of this work is to develop a silicon compiler that optimises a design with respect to multiple objectives set by the designer and automatically explore the design space. The system is implemented as the MOODS Silicon Compiler, acronym for Multiple Objective Optimisation in Data path (and control path!) Synthesis. The compiler centres around a global optimisation mechanism that is guided by a global cost function. It is the flexibility of the optimisation mechanism and the accuracy of the cost function which determines the optimality of a design subject to the designer's objectives.

3.1 MOODS INPUT AND OUTPUT SPECIFICATION

The design specification is given by a behavioural description which is compiled to an intermediate code (ICODE). A behavioural description documents the design in a readable, technology independent way. It is an abstract representation as it avoids premature bindings, therefore allowing more optimisation opportunities. All languages constrain the design description to a particular style as they have a finite syntax. A silicon compiler may detect fixed structures caused by the language syntax and translate them into an improved form. For a large range of language constructs this would be a difficult process, therefore the number of constructs is minimised in the intermediate code. Operations are decomposed into two-input instructions to reduce the number of different constructs and provide technology independence. The effect of this is to lengthen the description when compiled to the ICODE.

For each module in the description the ICODE consists of a set of processes each with a unique process number. Each process represents an instruction and has associated with it an activation list, that is, a list of processes to be activated when the current process ends. The sequencing of operations is similar to a Petri-net and uses a token passing mechanism to activate processes. A process may start only when all preceding processes have terminated, indicated by a token. A collect instruction is used where the preceding processes are executed concurrently. Its effect is to wait for a specified number of tokens before activating subsequent processes [47].

The ICODE represents the behaviour of the design at the register-transfer level and is the design description input for MOODS. The ICODE is generated from a behavioural source language using a language compiler. The language compiler provides syntax checks and optimisations such as dead code elimination and constant folding. Functional simulation tools are essential to the designer to verify the operation of his design and are applied either to the source description or to the resulting ICODE.

The source language may be either SCHOLAR [48] or a high-level subset of ELLA [71,72]. The SCHOLAR input is compiled to ICODE using the SCHOLAR language compiler and may be simulated using a functional simulator which is part of the language tools. An ELLA to ICODE interface [73] allows behavioural descriptions in high-level ELLA to be compiled to ICODE. Unfortunately ELLA is of a structural nature, therefore behavioural constructs such as loops are difficult to implement. Despite this, a behavioural interpretation is possible which may be correctly simulated using the ELLA simulator. Apart from giving the user a choice of source languages ELLA provides access to the tools available in the ELLA environment. An example of the SCHOLAR language and the resulting ICODE is given in Figure 3.1, where an instruction with no activation list is assumed to activate the next instruction in the code sequence. The full ICODE instruction set is given in Appendix A.

```
           SCHOLAR                                   ICODE
$(                                  3    Move #2 2  Act 4, 5
b2:=2                               4    Or 2 3 2   Act 9
      $[                            5    And 4 5 6
      b2:=b2 OR c3                  6    Ifnot 6    Act 8
      IF (d4 AND e5) THEN           7    Move #0 5  Act 9
            e5:=0                   8    Move #0 4
      ELSE                          9    Collect 2  Act 4, 5
            d4:=0
      $] REPEAT
$)
```

Figure 3.1 Example behavioural description and resulting ICODE.

The final optimised implementation generated by MOODS is represented in terms of parameterized cells, therefore it is natural for the output to be a netlist of parameterized cells. The required cells can then be taken from a cell library to produce a layout using conventional placement and routing methods.

The netlist is produced in the form of ELLA text using the simple MAKE, JOIN and LET statements. Each cell in the cell library is described as an ELLA macro which is instantiated in the netlist output. The clock runs to all register and control parts and is implicit in the cell descriptions by the use of the DELAY operator. By using ELLA as the output the designer has access to the ELLA tools allowing the final implementation to be simulated, where the ELLA time units represent clock cycles. If the source language used was ELLA then the same simulation vectors with little modification can be used to simulate both the initial description and final implementation; thus their functional equivalence can be verified. An example simulation of both initial description and final optimised implementation is given in Appendix B. After simulation at the cell level the design can be implemented using existing systems available from the ELLA environment. Variable names in the initial description are maintained throughout synthesis and are used in the netlist output thereby allowing the designer to probe known variables during simulation.

The ELLA netlist together with the parameterized cells represent the design at the gate/logic level. This may be flattened and further optimised by low-level logic synthesis systems which can optimise expanded cells and across cell boundaries. The true cost associated with the optimised cells can be represented in the cell library database which can detail pin overheads and different combinations of ALU functions within an ALU. For example, the cell library data takes into account the fact that the area increase in adding a function to an ALU will not be the area of the isolated function but will be less due to the shared resources within the ALU. A pin is a connection to a functional unit, whereas, a port is an module I/O connection; these definitions apply throughout the thesis.

Although the data and control parts are thought of as separate during the synthesis processes, they are combined in the final cell netlist. This gives layout tools more flexibility in achieving a compact layout. Separate netlists for the data and control parts could be produced, therefore permitting, through some post-processing, different controller implementations; such as a microcontroller.

3.2 MOODS ARCHITECTURAL MODEL

The chosen target architecture is a general one, therefore a data path plus control unit is used as the architectural model. The general architecture adds little restriction to the design space and allows the design of circuits from various application areas.

MOODS implementation independence is maintained by adopting a distributed data and control path. This distributed structure is represented by a netlist of abstract parameterized functional units and storage elements which implement instructions in the input description. The control path is expressed in terms of control states and control signals. Initially the distributed structure is sufficiently implementation independent to allow the design to be bound in a variety of ways; for example, implementing functional units as single cells or grouping into ALUs, merging registers into memories and implementing the control unit as a finite state machine or microcontroller.

A few implementation assumptions relating to the clocking scheme and timing of operations must be considered before detailing the synthesis steps. The clocking scheme adopted will match the cell library, which will typically be a single phase or two phase non-overlapping clock. VLSI cell libraries using these are smaller, simpler and more widely available than their multi-phase counterparts. The clock is assumed to run to all registers and control parts. Registers are loaded by controlling the load enable inputs and storage is assumed to occur at the end of a clock cycle, at which point the controller changes state. Each control state has a *status* output signal which when high indicates that the state is active.

The timing diagram of Figure 3.2 shows the timing for an addition operation. The vertical bars on the clock signal signify when the controller changes state and when registers are loaded. The time taken for an operation is dependent on the operation itself, its implementation method and the technology used. The clock period is used during the optimisation of the design and is assumed abitrarily long if not specified by the user. The clock period used to drive the final implementation will be equal to or greater than the maximum control node delay. If a clock period has been specified by the user then an operation will be multi-cycled if its execution time exceeds the clock period. If an operation is multi-cycled then additional control states must be added when the time

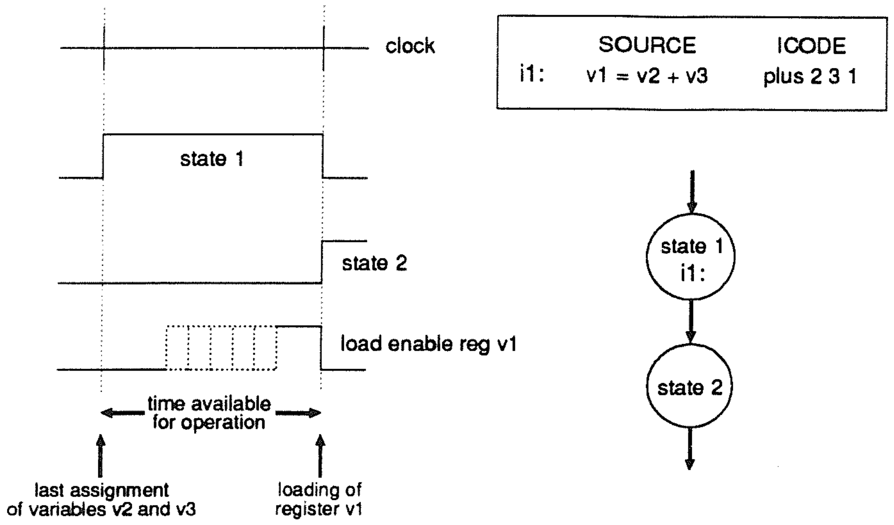


Figure 3.2 Timing diagram for the architectural model.

between the last definition of the operation's input variables and the assignment of the output variable is less than the execution time of the operation. Alternatively a faster implementation of the operation can be used which may reduce the number of additional control states but will inevitably increase the data path area. This illustrates the trade-offs possible between control and data path area and circuit speed.

3.3 DESIGN REPRESENTATION

A design representation is required as a vehicle for applying the synthesis steps. It must be flexible enough to represent the design throughout the synthesis process from the initial behavioural level to the final structural level. The design is represented as two graphs, a data path graph and a control graph. Two separate graphs are used to allow graph manipulations on one without affecting the other. This makes it easier to trace control or data paths and combine or split graph nodes. In addition to these graphs are lists specifying the variables used in the ICODE, ICODE modules and control signals.

All graphs and lists have links tagging points in one graph or list to points in another. The links aid searches through the data structures and more importantly provide a correlation between the behavioural description contained in the instructions of the control graph and the implementation contained in the structural representation of the data path graph [14]. This allows a multi-level representation to be maintained throughout the synthesis process and so reflect optimisations at both levels. An example of the links between data structures is shown in Figure 3.7.

The control graph, variable list and module list are constructed using the ICODE description. For each module in the ICODE a separate control graph is formed and an entry made in the module list. The module list contains pointers to the start and end nodes of the control graph and a pointer to the parent module in which it was declared. Also included is the module header that lists the input and output ports to the module, which for the main program represent external I/O connections and possibly IC pins.

The variable list contains an entry for every variable in the ICODE. Each entry has a pointer to the data path unit storing the variable (see Figure 3.7). The list includes both declared and compiler created variables which are treated equally throughout synthesis. Only I/O variables are reserved, that is they can not be optimised away, thus preserving their original roles. The only difference between the variable types is that the declared variable has a user defined name and bit width which defines operator bit widths whereas the compiler variable has no name and its bit width is derived from the width of the operations using it. Compiler variables are usually created where nested expressions existed in the source description. They are therefore only used once, that is, their lifetime consists of a single write and single read operation thus making them good candidates for elimination during synthesis and is the reason why some compilers (such as Scholyzer [20]) treat them differently.

3.3.1 THE CONTROL GRAPH

The control graph depicts control state sequencing information and instruction precedences. It is cyclic, where cycles represent control structures with loops. Each node in the graph represents a distinct control state, within which is a set of instructions to be executed.

The control graph is defined as follows:

$$\text{CONTROL GRAPH} = (\mathbb{S}, \mathbb{A})$$

where $\mathbb{S} = (s_1,s_2,...,s_n)$ is the set of *nodes* and $\mathbb{A} = (a_1,a_2,...,a_n)$ is the set of *arcs*, which is a subset of the cartesian product $\mathbb{S} \times \mathbb{S}$.

A node s = (type, I_s, A_{in}, A_{out}, attributes) consists of:

type	An attribute indicating the node type, either *general*, *fork*, *conditional*, *dot*, *call* or *collect*, as explained below.
I_s	A set of instructions, $I_s \subset \mathbb{I}$, where $\mathbb{I}$ is the complete set of instructions in the ICODE.
A_{in}	A set of input arcs, $A_{in} \subset \mathbb{A}$
A_{out}	A set of output arcs, $A_{out} \subset \mathbb{A}$

attributes, these consist of:

collect_N	The number of tokens for a collect node.
delay	An estimate of the time taken for instructions I_s to execute.
node_enable	The status signal indicating an active control state.
loop_its	The number of times a node is enabled during a single pass of the control graph - used in critical path calculations.
slack	The critical path slack (often called mobility or freedom).

An arc a = (s_s, s_t, c_g, FBA) consists of:

s_s	A single start node, $s_s \in \mathbb{S}$
s_t	A single terminal node, $s_t \in \mathbb{S}$
c_g	A condition signal which gates the arc, that is, if c_g evaluates to true and the start node is active then at the end of the current clock cycle the arc can activate the terminal node.
FBA	An attribute indicating that the arc is a feedback arc thus creating a control loop, the removal of which renders the graph acyclic.

The node type describes the class of control unit to be used when implementing the controller. It relates to the arc configuration connecting to the node and any special control considerations imposed on the controller by particular ICODE instructions. The nodes are described in order of precedence, with general being the lowest and collect the highest. Figure 3.3 illustrates the nodes.

A *general* node contains ICODE instructions other than collect, module call and conditional instructions (IFs) and has at most one input arc and one output arc. A *fork* node is the same as a general node except that it has two or more output arcs and therefore marks the start of a parallel section, where all its successor nodes are executed concurrently.

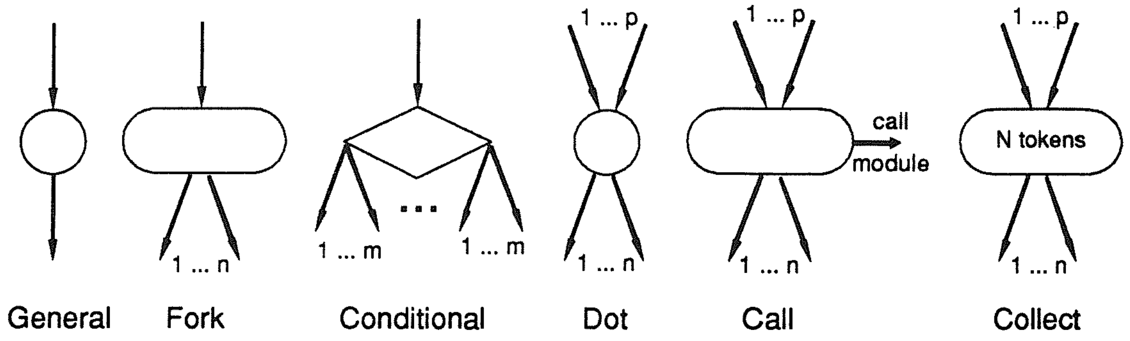


Figure 3.3 Illustration of control graph nodes.

A *conditional* node, like a fork node has one input arc and two or more output arcs. The output arcs of both general and fork nodes have conditions set to true, however, in conditional and higher precedence nodes the output arc conditions depend on other signals. The same condition may be applied to many output arcs thereby rendering the node a fork type under that condition. The arc conditions originate from a conditional ICODE instruction in the node, such as IF, COUNT (FOR loop test), and SWITCHON (CASE) instructions. A *dot* node is similar to the conditional node except that it has two or more input arcs and may be activated by any one of them. The dot node is equivalent to an *or-join* node in other systems and is the counterpart to the conditional node.

A *call* node is different to all other nodes as it contains only a module call instruction. It may have any number of input and output arcs, although the output arc conditions will be true as the node will not contain a conditional instruction. When activated, the call node also activates the start node of the module to be called. When this has finished execution, indicated by the activation of its terminal node, the call node activates its successors.

A *collect* node is the same as a dot node except that it contains a collect instruction indicating that it cannot activate its successor nodes until a fixed number of tokens, given by the collect_N attribute, have been collected. The tokens are synonymous with active input arcs. The collect node is equivalent to an *and-join* node in other systems and is the counterpart to the fork node, however, they do not necessarily bear a one-to-one relationship; for example, nested parallel sections may terminate on a common collect node. In general terms a "fork type" node is one with two or more output arcs and a "join type" node is one with two or more input arcs.

The control graph for each module is a connected, cyclic, bipartite graph. The nodes are joined by arcs indicating control flow. Control loops are represented by feedback arcs indicated by the FBA arc attribute. The removal of all feedback arcs renders the graph acyclic. A control graph has only one start node, from which any other node in the same module is reachable via the feedforward arcs. If a node has no input or output arcs, excluding feedback arcs, then it must be a module start or end node respectively. A module may have more than one end node but only one start node. The graph can be partitioned into sequential and parallel sections. A sequential section is a part of the control graph where nodes are unconditionally activated, one at a time, in a sequential manner. Each node in a sequential section has only one input arc and one output arc except for the first and last nodes which may have any number of input and output arcs respectively. A parallel section is a part of the graph starting with a fork type node and ending with a dot type node. A parallel section encloses two or more sequential sections which may be executed concurrently.

Each control node contains the set of instructions to be executed in its control state. The instructions may be dependent on each other, that is, the output of one is the input to another. The dependencies between instructions are represented by acyclic instruction graphs within the control node and are used to determine the path and thus the time required for a correct result to propagate through the data path. The instruction graphs must be acyclic as cycles represent feedback which may cause instabilities. Each instruction graph is given a unique *group number* indicating that instructions with the same group number are in the same control state and dependent on one another, possibly indirectly via other instructions or control signals. Within each control state the groups are executed concurrently, at the end of which the results of the instructions executed are loaded into registers. Intermediate results are not stored unless they are required by instructions in subsequent control states.

An instruction $i \in \mathbb{I}$ contains, as well as the data specific to each ICODE instruction, a number of attributes consisting of:

c_f	Condition for firing, if c_f evaluates to true then instruction i is executed.
impl_link	Implementation link indicating which data path unit is implementing the instruction.
$\mathbb{M}_i$	Mutually exclusive instructions - the set of instructions which are never executed concurrently with instruction i.
Group No.	The instruction graph to which i belongs.
Delay	The time for i to execute.
End time	The time from the start of the control state for i to end execution.
$\mathbb{Dp}_i$	Predecessor instructions - the set of instructions within the control state that i directly depends on.
$\mathbb{Ds}_i$	Successor instructions - the set of instructions that depend directly on i within the control state.

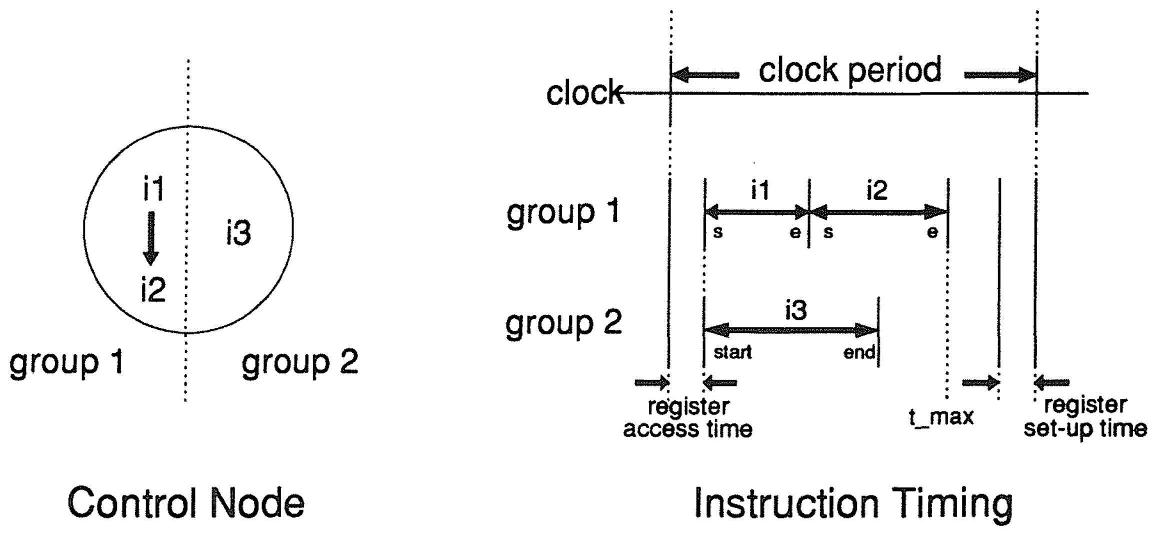


Figure 3.4 Example of instruction group timing within a control node.

The instruction graphs within control states are described using the predecessor and successor instruction sets. Figure 3.4 shows the timing information for three instructions in a control state; i_1 and i_2 are dependent and form the instruction graph, group 1, while i_3 forms instruction graph, group 2. The groups within a control node execute concurrently and for the purpose of timing calculations, instructions within a group execute according to the instruction graph. For example, i_1 and i_2 are considered to execute sequentially owing to their dependency. Register accesses occur at the control state boundary as shown in Figure 3.4. If a clock period has been defined then instructions with an execution time greater than the clock period will be multi-cycled in the initial and subsequent control graphs. A special instruction is used to indicate the continuation of an instruction in the successor nodes.

The initial control graph is constructed from the ICODE description by placing each instruction in a separate control state with its condition for firing set to true. The nodes are then linked with arcs according to the activation lists, with all arc conditions being set to true except those with a conditional instruction in the preceding node. Dot nodes are added where instructions have identical activation lists containing more than one process. An example control graph constructed from the ICODE example of Figure 3.1, is shown in Figure 3.5. Note the addition of a dot node caused by instructions i3 and i9 having equal activation lists containing more than one process. Beside each node is a list of the instructions ($i_n(c_f)$) within the node, where the condition for firing is omitted for c_f=true. For arc conditions other than true the signal (s_n) is placed beside the appropriate arc. The node enable signal generated by a control node is indicated inside the node.

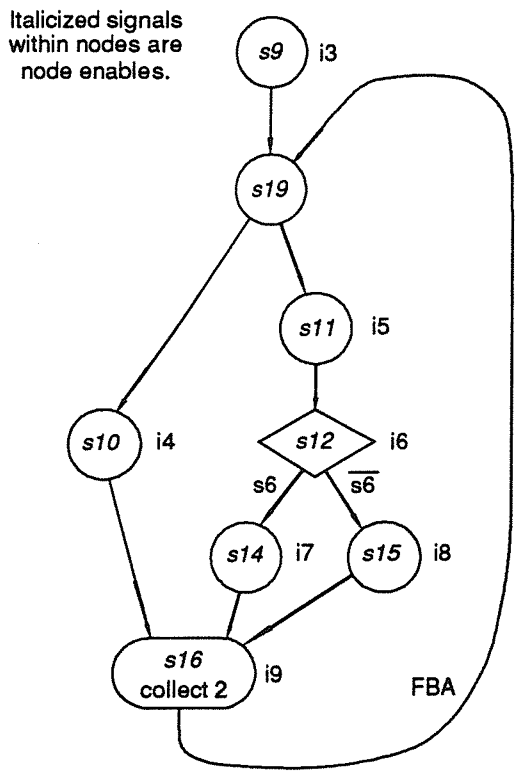


Figure 3.5 Example of an initial control graph.

To complete the initial control graph a test is made to ensure that instructions executed in concurrent sections of the graph are not contentious. Additional information is extracted from the control graph after its creation and before performing any optimisations. This information does not change during the application of transformations and therefore a significant reduction in computational effort can be made by generating and retaining the information instead of generating it each time it is required. There are two sets of information that can be extracted, firstly, the minimum feedback arc set (MFBAS) which when removed from the control graph renders it acyclic and secondly, mutual exclusion between instructions.

The MFBAS is generated in two stages, firstly, taking the permanent of the adjacency matrix by recursive expansion and secondly, the construction of a boolean function which when manipulated yields the required arc set [74,75,76,77]. For large general graphs the computation is "hard". However, in a control graph there are few feedback arcs, which in addition to matrix reduction methods result in a matrix where in many cases the MFBAS can be directly obtained without any further algebraic manipulation. The reduction methods involve removing and noting self arcs which by their definition are feedback arcs and removing single input control nodes, where the input arc can never be a feedback arc as the node would be inaccessible in the acyclic graph.

Mutual exclusion occurs between a pair of instructions that can never be executed concurrently, therefore the instructions may share hardware even when executed in the same control state as they are not executed together. For example, in Figure 3.5 instructions i7 and i8 are mutually exclusive owing to the preceding conditional node. Mutual exclusion is determined for all instructions in conditional branches of the control graph by recursively analyzing conditional nodes. For a given node, each instruction in one branch with branch condition s_b is mutually exclusive to all instructions in all other branches with branch condition $s \neq s_b$.

After optimisation the control graph is likely to take on a different appearance. Many instructions may occur in a control state and arc conditions and conditions for firing will have changed.

3.3.2 THE DATA PATH GRAPH

The data path graph is constructed from the variable and module lists and the ICODE instructions in the control graph. It describes the data dependencies and paths between the functional units which implement the instructions. Each instruction and variable is initially assigned its own data path unit, however as the structure is optimised this may no longer be true as unit sharing may occur.

The implementation of a design is divided into two parts, the control part, implementing the controller and the data part, implementing functional units and storage. The control part is implicit in the control graph described in the previous section and does not, therefore, require representing in the data path graph. It is assumed to exist as an implicit unit with clock and data path control signals as inputs and control signals, such as state enable, register load and register clear, as outputs.

The nodes of the data path graph represent functional, storage, boolean and interconnect units and the arcs represent constants and input, output and control connections. Each graph node is parameterized according to the bit width and contains a pointer to the library cell which implements the unit. Functional type nodes represent data path units that implement arithmetic or logical functions. Storage type units implement registers, ROM, RAM, or I/O ports, as well as some ICODE instructions specific to the control inputs of storage units; these are the MOVE, TRUE, HIGHZ and COUNT instructions. The recognition of storage functions is important for cost effective designs [44]. Boolean type units are used as an alternative representation to a network of simple logic gates. The network is converted from the data path representation to a condition signal representation which can be implemented with the controller condition signals. The interconnect units depict multiplexers (MUX) as defined by the architectural model; however they could depict buses in a bus based architectural model. Initially multiplexers are only required on register inputs, where variables are assigned more than one value. No other multiplexers are required as no unit sharing occurs in the initial data path graph. The need for an interconnect unit is apparent by the connection of more than one signal to an input of a unit. It is therefore implicit in the graph structure and can be added after optimisation to avoid excessive graph manipulations during optimisation.

The data path graph is defined as:

$$\text{Data Path Graph} = (\mathbb{U}, \mathbb{N})$$

where $\mathbb{U} = (u_1,u_2,...,u_n)$ is the set of data path *units* and $\mathbb{N} = (n_1,n_2,...,n_n)$ is the set of *nets*.

A unit u = (node_type, cell_type, N_{in}, N_{out}, $C_{control}$, impl_links, n_bits, lo_bit, area, power) consists of:

node_type	An attribute indicating the node type, which can be: *functional*, *storage*, *boolean* or *interconnect.*
cell_type	A pointer to the cell database indicating the parameterized cell used to implement unit u.
N_{in}	A set of input nets, $N_{in} \subset \mathbb{N}$
N_{out}	A set of output nets, $N_{out} \subset \mathbb{N}$
$C_{control}$	A set of control signal inputs as described below.
impl_links	A set of links indicating which instructions are implemented by the data path unit.
n_bits	A parameter indicating the number of bits.
lo_bit	A parameter indicating the index of the lower bit, the index of the upper bit is given by: lo_bit + n_bits - 1.
area	An attribute indicating the estimated area occupied by the unit when implemented using the cell given by cell_type.
power	An attribute indicating the estimated power consumed by the unit when implemented using the cell given by cell_type.

A net n = ($type_{in}$, $value_{in}$, $range_{in}$, pin_{in}, $type_{out}$, $value_{out}$, $range_{out}$, pin_{out}, i_{act}, i_{wact}, var) consists of the following attributes:

$type_{in}$	The type of input that is connected to the net, it can be: a *unit*, a *control signal* or a *constant.*
$value_{in}$	The input value; its interpretation depends on the type attribute. For a unit it is a pointer to a unit, for a control signal it is a signal number and for a constant it is a value.

$range_{in}$	The output bit range for a unit input.
pin_{in}	The output pin type for a unit input.
$type_{out}$	The type of input that the net connects to, it can be: a unit or control signal.
$value_{out}$	The item that the net drives, it is dependent on the type attribute. For a unit it is a pointer and for a control signal it is a signal number.
$range_{out}$	The input bit range of the connecting unit.
pin_{out}	The input pin type of the connecting unit.
i_{act}	The instruction number that has caused the net to be created. This may be used later for the generation of MUX control inputs.
i_{wact}	The instruction number that reads from the net. For nets to registers this will equal i_{act}. This is used to keep track of ALU inputs and may be used later for the generation of interconnection control signals.
var	The variable number and relevant active period (see Section 4.1.9) transmitted over the net.

A control signal $c \in C_{control}$ is used to direct a condition signal to a control input of a data path unit. The control signal $c = (c_s, pin, range, i_{act})$ consists of the following attributes:

c_s	Condition signal which when true activates the relevant control input on the unit.
pin	The input pin type that the control signal activates.
range	The bit range to be activated on the input pin.
i_{act}	The instruction number causing the control input.

An example data path is shown in Figure 3.6 and represents the initial data path for the example ICODE given in Figure 3.1. The control of data flow through the graph can be determined by identifying node enable and control signals in the control graph of Figure 3.5 with those used and generated in the data path.

The initial mapping of ICODE to data path elements is similar to that used in Scholyzer [20], except that all signals are stored. Input variables to all modules and

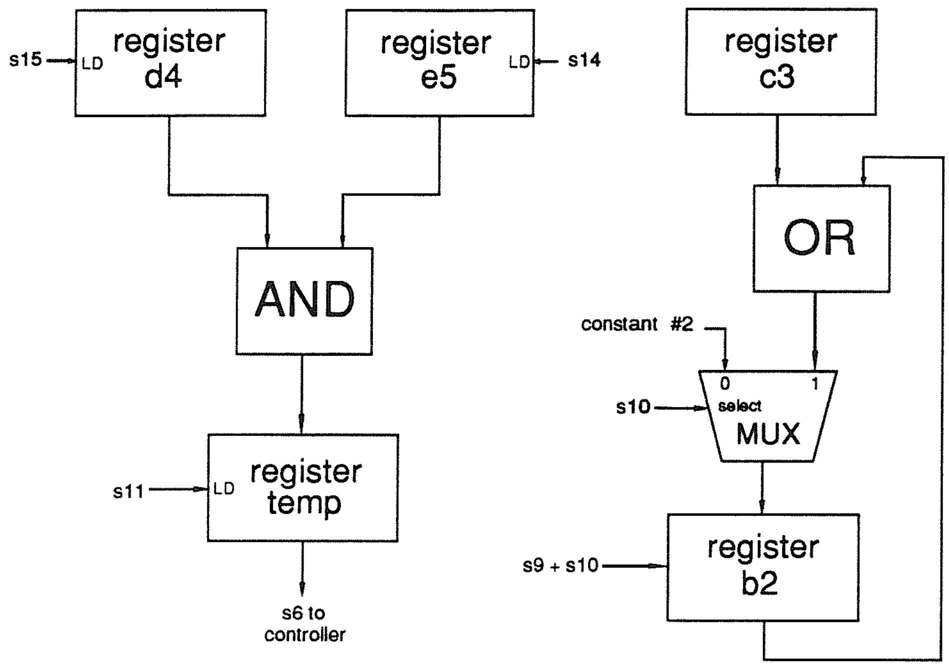


Figure 3.6 Example of an initial data path graph.

output variables from internal modules are mapped to ports, where, ports connecting to internal modules are later mapped to nets. Output variables from the main program are mapped to registers. Storage nodes such as registers and counters are tailored according to the instructions which write to them. A *common destination* (CD) list is created for each variable and consists of the set of instructions that write to the variable. The CD list is analyzed to determine the type of storage unit required and any control attributes. Instructions which represent reset or set operations (for example a:=0 or a:=2^n-1, where n is the bit width of the variable *a*) are mapped to the appropriate set and reset control pins. A variable whose CD list contains only set, reset and increment (a:=a+1) or decrement (a:=a-1) operations is mapped to a counter and the count control pin set accordingly.

ICODE operations are mapped to units which are implemented by a set of *basic* library cells that are assumed to exist in the cell library. As the implementation is optimised the units may be implemented by *non-basic* cells which the user may enter into the cell library.

Between instructions in the control graph and nodes in the data path graph are *implementation links*, these indicate which data path units are implementing which

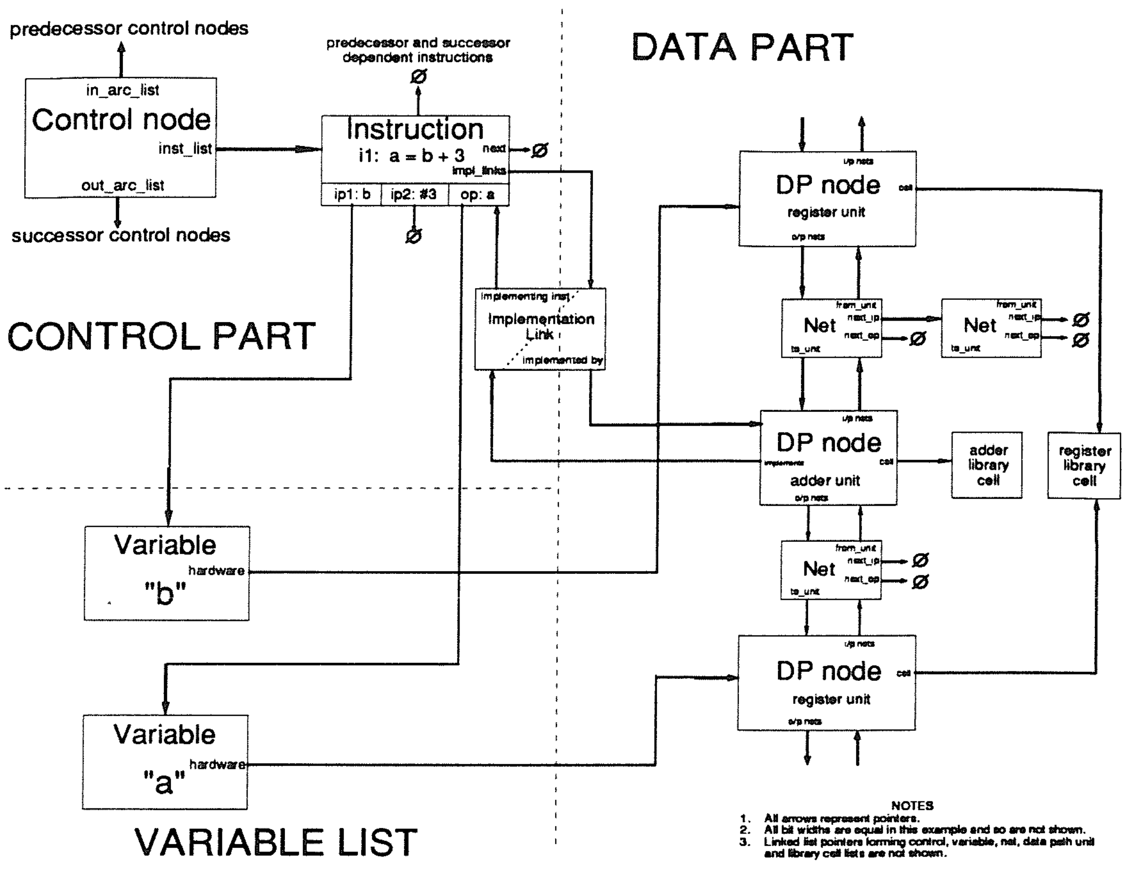


Figure 3.7 Design data structure showing links between control and data path graphs and variable list.

instruction(s). They provide a fast method of determining whether units are shared between instructions, which would otherwise require computationally expensive searches. Figure 3.7 gives an example of the implementation link and other data structure links between the data structures associated with a control node containing a single instruction. The single instruction forms its own instruction graph and therefore has no dependencies in the control node, shown by the null (∅) pointer.

3.4 MOODS OPTIMISATION STRATEGY

When selecting the optimisation strategy a number of requirements have to be taken into consideration. The first and most important is the ability to perform the synthesis tasks simultaneously, thereby facilitating trade-offs between tasks. The trade-offs are essential for producing optimal designs when optimising with respect to more than one aspect of the design. The second consideration is the ability to explore the design space quickly. This can be done by re-synthesizing the design. The re-synthesis computation time can be reduced by synthesizing from the current control and data path graphs rather than the initial control and data path graphs. This is based on the assumption that most practical optimal solutions in the design space will be closer to the current design space position than the initial position, therefore requiring less computation to reach them from the current position. A third consideration is to allow the designer to manually adjust the implementation to include his quirks or refine the implementation.

Algorithmic approaches using linear programming have been reported as having a computational explosion for even the smallest of practical designs. For multiple objectives integer goal programming [78] must be used. Linear programming is a special case of goal programming, therefore the use of goal programming would be impractical from the computational point of view. Other algorithmic approaches, such as clique partitioning, were considered and although they are applied to individual synthesis tasks, they can take constraints into account which allow for better results from subsequent synthesis tasks. However, the problem arises whereby the constraints may not be the best ones to achieve a particular goal in subsequent processes or there may be too many or too few of them. This is due to the lack of feedback from later synthesis processes to earlier ones and because of this a global optimum cannot be achieved. In addition, the constraints are not in terms of real circuit parameters such as area and delay and would therefore be meaningless to the user.

Design space exploration in the algorithmic methods must be achieved by re-synthesizing the design from the initial point in the design space. Re-synthesizing is normally done from the initial design point, however, in very few circumstances a partially synthesized design between synthesis tasks can be used which avoids performing preceding tasks.

An iterative optimisation strategy is used in MOODS as it overcomes the problems described above. Iterative optimisation is achieved by breaking the synthesis tasks into a number of local transformations, some associated with allocation and others with scheduling and translation. This allows the simultaneous consideration of the synthesis tasks which is recognised as being extremely difficult [79] and can result in complex algorithms or simplified design models. However any implementation can be obtained by manipulating a directly compiled design [9] and the opportunistic design modifications in the iterative method show the greatest power [69].

The transformations are considered as non-binding, allowing previous design decisions to be overruled, therefore providing the opportunity for design degradation and so the basis for global optimisation. Transforms may be applied at either the behavioural or structural level as the use of a multi-level representation reflects changes to the design at both levels. The transforms are *complete*, that is, a transform applied to a correct design will always result in a correct design; this eliminates interaction between transformations and allows them to be applied to the design in any sequence. To achieve global optimisation a global cost function is required to steer an optimisation algorithm in applying the transformations. The separation of the transforms, cost function and optimisation algorithm means that different optimisation techniques and strategies can be developed and incorporated in the MOODS system. The transformations, cost function and optimisation algorithms are the subject of subsequent chapters.

As the strategy is iterative design space exploration can be efficiently achieved by stopping the optimisation algorithm, changing the objectives and continuing from the current design point. This constitutes *dynamic design space exploration.* The use of non-binding transformations ensures that design degradation can be done, which may be necessary in reaching the new objectives. The transformations can also be applied manually to the design. This gives the designer the ability to make accurate local modifications without the need for blindly adjusting objectives or applying inaccurate constraints and re-synthesizing from the initial design.

Binding is assumed to take place after the design has been optimised and the final implementation proved to be to the designer's requirements and can be considered as *delayed binding* [66]. The design is then written to the design files and hardware synthesis and layout performed.

4 Transformations

This chapter describes the transformations used in the MOODS synthesis system. All references to program code are shown in the `Courier` typeface, where the data structures are listed in Appendix C and program examples are written in pseudo-C.

For multiple objective optimisation and fast design space exploration, trade-offs between synthesis tasks are essential. To facilitate trade-offs and avoid local minimum traps caused by their sequential execution, the synthesis tasks, scheduling, allocation and translation, must be performed simultaneously. This is achieved by dividing the synthesis tasks into a number of transformations which may be iteratively applied to the design in any sequence. The transforms may occur at either the behavioural or structural level as changes at one level are reflected in the other by the use of a multi-level representation.

To allow the application of transformations in any order each must be *complete*, that is, it transforms the graph structures from one valid implementable design to another without the need to apply additional transformations to tidy the effects of the previous ones. Therefore a transformation to remove parallel control path arcs does not exist as a separate transformation but as a sub-transform used in transformations that generate parallel arcs as a consequence of removing control nodes.

The correctness of the transformations is essential to ensure the design remains functionally equivalent to the original specification throughout the synthesis process. The semantic preserving properties of transformations should be easy to prove if they are simple [22]. Mathematical models of behaviour have been devised elsewhere [18,80] which can be used to prove the correctness and functional equivalence of a design after the application of transformations.

The transformations are considered non-binding, allowing the effects of previous transformations to be overridden, therefore delaying the binding of design decisions [66]. Each transformation will affect many aspects of the design and are general in that they are not directed towards improving a particular aspect of the design. They may improve one aspect of the design while degrading other aspects. However, they can be associated with a particular synthesis task.

The amount of improvement, if any, is dependent on the section of the design to which the transform is applied, as well as the user's objectives represented by the cost function and technology dependent data. It is the purpose of the optimisation algorithm to apply the transformations in such a way that the user's criteria are met. To give the optimisation algorithm many stratagems, the generality of the set of transformations is important. For this reason some transformations produce a design degradation by undoing the effects of previous transformations. This provides the basis for global optimisation in the same way as simulated annealing does through degradations by "hill climbing" moves [81]. None of the transformations make assumptions concerning possible trade-offs. For example, the statements "trade-offs on storage are not significant"[35] and "short bit width operations are ignored because it is not profitable to share hardware among them"[66] are only half truths as the costs and therefore the trade-offs depend on technology dependent information. Assumptions such as these cause premature bindings and unnecessarily restrict the design space.

Most of the transformations are local, that is, they are applied to a small part of the design. Each transformation consists of four distinct steps:

1. **Data selection.** Data selection involves selecting a transformation and the design data on which to apply it. The transformation type and associated data are entered into the appropriate fields of a transformation data structure which is used as a vehicle to pass information between the program procedures. In the parameter list of procedures given here the transformation data structure is called *td.* In the manual optimisation mode the user sets the fields by interacting with the program in the `select_trans(td)` procedure, whereas in the automatic optimisation mode the information is created by the `auto_select_trans(td)` procedure according to the requirements of the optimisation algorithm.
2. **Testing.** The information entered in the data selection step is minimal, with additional data required to perform the allotted transformation being generated during the testing stage. All of the information is tested by the procedure `test_trans(td)` and appropriate error codes given when the transformation cannot be performed.
3. **Estimation.** The estimation step, performed by the procedure `estimate_trans(td)`, takes the transform data and estimates the effect the selected transform will have on the cost function. This estimation is used by the

optimisation algorithm to determine whether the transform would be instrumental in reaching the user's objectives.

4. **Execution.** The selected transformation is performed by procedure `perform_trans(td)` only if the optimisation procedures, after evaluating the estimation, consider it appropriate.

Each of the five procedures associated with the above steps consist of a case statement which selects the appropriate data selection, testing, estimation or execution procedures according to the transformation type initially chosen in the data selection step. For example, if the sequential merge transform is selected then `test_trans(td)` will call the `test_seq_merge(td)` procedure. If any error occurs due either to incorrect data selection or the selected data failing the testing step then a suitable error code is returned. The error code is used to issue an error message to the user in the manual mode or to direct the automated optimisation algorithm. The first three steps, data selection, testing and estimation, provide more information to the user and/or optimisation algorithm and do not make any permanent changes to the design. If an error code is generated during these steps then the transformation is prevented from being performed by the final execution stage.

The remainder of this chapter describes the tests performed on the design data structures and each of the transformations. Section 4.1 describes a collection of the important tests which are used during the testing stage of the transformations. The scheduling and allocation transformations are described in Sections 4.2 and 4.3 respectively, with the translation transformations being in the appropriate sections. Some of the transformations are similar to those used in other systems [20,23].

4.1 TRANSFORMATION TESTS

To determine the pending success of a transformation the selected data is checked using an appropriate collection of tests. Many of the tests and calculations performed during the synthesis process involve traversing graph structures in order to determine various characteristics of the design and validate transformation data; for example, critical path analysis in the control path, delay calculations in the instruction graphs or reachability and variable lifetime tests. The general recursive graph traversal algorithm used to perform these tasks is shown in Figure 4.1. The algorithm shown traverses a graph forwards from a given start node in a depth first fashion. An equivalent algorithm that traverses a graph backwards from a given end node is obtained using predecessor instead of successor nodes. This procedure is similar to those used in other graph theory applications [82].

```
traverse_graph(node)
{
if (!node || node has been evaluated) return;
if (insufficient data to evaluate node) return;
evaluate node;
for (each valid successor node)
      traverse_graph(successor);
}
```

Figure 4.1 The general graph traversal algorithm.

The tests and analysis described here represent the most significant analysis topics employed in all high-level synthesis systems. Some of the tests described are utilised in more general cases. In addition to these, numerous simple tests are also used during synthesis, however these are mentioned where necessary and require no explanation as they only involve testing single fields in the data structures. For example, to determine whether a control path arc is a feedback arc only requires its FBA field to be checked.

4.1.1 REACHABILITY TEST

A control node n_j is reachable from node n_i if there exists a path from n_i to n_j in the acyclic control graph, that is, the path does not include any feedback arcs. The reachability test is performed by the `is_reachable(`n_i`,`n_j`)` procedure which marks all nodes after n_i in the acyclic graph. If n_j has been marked then it is reachable from n_i and *true* is returned. If n_j has not been marked then it is not reachable and *false* is returned indicating n_j precedes n_i or both nodes are in parallel branches of the graph. The marking is done by the recursive procedure `trace_temp_forward(start)` shown in Figure 4.2 which is derived from the general graph traversal procedure, where `start` is the control node to mark forward from. The field *temp* is used as a "visited" flag which is initially set to *false*; the evaluation of each node consists of setting *temp* to *true*. A valid successor node is one that is not reached via a feedback arc.

```
trace_temp_forward(start)
struct control_node *node;
{
if (!node) return;
if (start->temp) return;        /* already visited */
start->temp=true;
for (each output arc in the list start->out_arc_list)
        if (!out_arc->is_FBA)
                trace_temp_forward(out_arc->succ_node);
}
```

Figure 4.2 Recursive node marking procedure.

4.1.2 Mutual Exclusion Test

Mutual exclusion occurs between a pair of instructions which can never be executed concurrently owing to their occurrence in different branches of a conditional construct. Mutual exclusion is determined as described in Section 3.3.1; where for each instruction a set of mutually exclusive instructions is obtained. The set of mutually exclusive instructions is stored in a linked list of pointers in the instruction data structure.

The mutual exclusion test is performed by the `test_mutual_exclusion(`i_j`,`i_k`)` procedure which tests the mutual exclusion list associated with instruction i_j for a pointer to the instruction i_k, that is $i_k \in \mathbb{M}_j$, where $\mathbb{M}_j$ is the set of instructions mutually exclusive to instruction i_j. The order of the instructions given in the procedure parameters is not important as if $i_k \in \mathbb{M}_j$ then the converse $i_j \in \mathbb{M}_k$ is also true.

A pair of data path units may also be mutually exclusive if every instruction using one unit is mutually exclusive to every instruction using the other unit, that is, the units are never active at the same time. Procedure `test_reg_mutual_excl(`r_1`,`r_2`)` tests two registers, r_1 and r_2, for being mutually exclusive.

4.1.3 Hardware Sharing Test

To permit the concurrent execution of a pair of instructions they must not share a data path unit in their implementation, unless they are mutually exclusive. The hardware sharing test is performed by the procedure `test_sharing(list1,list2)`, where list1 and list2 are two sets of instructions that are to be executed in the same control node. The procedure tests each instruction in list1 against each instruction in list2. If the instruction pair is not mutually exclusive and a data path unit is shared between them, indicated by the implementation list, then a *true* is returned. Otherwise the next pair are tested until the end of the lists whereupon a *false* is returned.

4.1.4 Dependency Test

The dependency test determines whether there is a dependency arc between a pair of instructions, i_j and i_k. A dependency arc is created between two instructions that are in the same control state, where the result of one affects the operation of the other, that is, it may be an input or a term in the firing condition. The dependency list is similar to the mutual exclusion list in that each instruction has a linked list of pointers to dependent instructions. However as the dependency arcs describe the instruction graph, which changes during synthesis, so the arcs themselves must also change.

The dependency test is performed by the `is_dependent(`i_j, i_k`)` procedure which tests the dependency list of i_j for the successor dependent instruction i_k, that is $i_k \in \mathbb{Ds}_j$, where $\mathbb{Ds}_j$ is the set of successor dependents of i_j. The order of the instructions given in the procedure parameters is important as if $i_k \in \mathbb{Ds}_j$ then the converse $i_j \in \mathbb{Ds}_k$ is not true and may never be true due to the restriction that the instruction graph is acyclic. However using the predecessor dependent list $\mathbb{Dp}$, $i_k \in \mathbb{Ds}_j$ is the same as $i_j \in \mathbb{Dp}_k$.

4.1.5 Contention Tests

Two instructions i_j and i_k, i_j precedes i_k, may be executed concurrently if their concurrent result is equal to the original result. For example, given the following pair of sequential instructions:

i_1: `a=b+c;`

i_2: `d=a+1;`

and that a=0, b=2 and c=3 the values of a and d for their sequential execution will be 5 and 6 respectively. However, for the concurrent execution of the instructions the values of a and d will be 5 and 1 respectively. The variation in the value of d indicates that instruction i_2 is dependent on the output of instruction i_1 and that contention exists when they are executed concurrently. Contention occurs when there is a dependency or a variable access violation between the instructions. An access violation occurs when the instructions write to the same variable, which for concurrent execution will give conflicting results. Note that if the instructions are mutually exclusive then contention between them cannot exist as they are never executed concurrently.

Contention is determined by testing the instruction's variables in the following way [52]:

Let $SV(i_p) = \{v \mid v \in$ set of source variables of i_p, $i_p \in \mathbb{I}\}$ and

$DV(i_p) = \{v \mid v \in$ set of destination variables of i_p, $i_p \in \mathbb{I}\}$

be the set of inputs and outputs, respectively, for an ICODE instruction. The source variables include the input variables for the firing condition of instruction i_p as well as its input variables.

```
contention_ij_ik(i_j,i_k)
struct instruction *i_j,*i_k;
{
if (test_mutual_exclusion(i_j,i_k)) return 0;
if (DV(i_j) ∩ DV(i_k) ≠ ∅) return 2;   /* access violation */
if (DV(i_j) ∩ SV(i_k) ≠ ∅) return 1;   /* dependency       */
return 0;                              /* no contention    */
}
```

Figure 4.3 Contention test procedure.

The procedure `contention_ij_ik(i_j,i_k)` returns a 0, 1, or 2 for no contention, dependency or access violation respectively, thus giving an indication of the type of contention between the instructions tested. A further test is necessary if the inputs to both instructions are from single port memory variables. The test, $SV(i_j) \cap SV(i_k) = \emptyset$ ensures that both instructions do not read from the same memory.

The procedure shown in Figure 4.3 is used in further contention test procedures which test an instruction against a node and a node against a node. The test performed is indicated by the latter part of the procedure name. For example, the procedure `contention_nj_nk(`n_j`,`n_k`)` tests the contention of all instructions in node n_j against all instructions in node n_k and returns the highest contention test result.

The contention test is used to determine the insertion of dependency arcs in the instruction graphs and thus indicate the implied ordering of concurrent instructions. The dependency arcs are important to maintain the correct instruction ordering when serialising concurrent instructions; for example if instruction i_k depends on i_j then i_j must precede i_k when serialised. However, the lack of a dependency arc between concurrent instructions does **not** imply the instructions can be serialised in any order. Consider the concurrent instructions:

i_1: `a=b+c;` i_2: `d=a+1;`

and that in the previous control state a=0, b=2 and c=3, thus after execution a=5 and d=1. The two possible serialisations, i_1 precedes i_2 and i_2 precedes i_1 would produce the results a=5, d=6 and a=5, d=1 respectively; the latter being the correct result as it is the same as that produced by the concurrent instructions. The correct result is obtained if and only if the serialisation does not result in any dependency between the instructions.

```
test_contention(list_j,list_k)
struct instruction *list_j,*list_k;
{
for (each instruction i_j in list_j)
     for (each instruction i_k in list_k)
          if (!is_dependent(i_j,i_k) && contention_ij_ik(i_j,i_k))
               return true;
return false;
}
```

Figure 4.4 Procedure to test for the creation of additional contention on instruction serialisation.

The procedure `test_contention(`$list_j$`,`$list_k$`)` shown in Figure 4.4, takes two concurrent sets of instructions with any dependency arcs between them intact and determines whether the correct behaviour is maintained by the serialisation $list_j$ precedes $list_k$. Each instruction in $list_j$ is tested for contention with each instruction in $list_k$. If the instructions are contentious and a dependency arc does not exist between them then the serialisation will result in additional dependencies and thus a change in behaviour. Each instruction pair is tested until either extra contention is found, whereupon *true* is returned or all instructions have been tested whereupon *false* is returned. Note that the dependency test is performed first as it is less computationally intensive.

4.1.6 Jump Test

The jump test is used to determine whether a pair of consecutive instructions (i_j, i_k) can be executed in reverse order without affecting the results. This is only possible if the instructions are independent of each other or they are mutually exclusive, in which case they are independent in time rather than in terms of variable accesses. An independence in time has a higher priority than a variable independence, as a time independence implies no conflicting variable access occurs. A warning is issued if the instructions are found to write to the same variable. This indicates that no instruction previously tested, when jumping many instructions, was found to access the variable, therefore the first instruction may be superfluous.

```
jump_i_i(ij,ik)
struct instruction *ij,*ik;
{
if (test_mutual_exclusion(ij,ik)) return true;
if (DV(ij) ∩ SV(ik) ≠∅) return false;
if (SV(ij) ∩ DV(ik) ≠∅) return false;
if (DV(ij) ∩ DV(ik) ≠∅) {
      printf("superfluous instruction ij");
      return false;
      }
return true;
}
```

Figure 4.5 Instruction jump test procedure.

Let SV and DV be as defined on page 71. The `jump_i_i(ij,ik)` test shown in Figure 4.5 returns *true* if the order in which the instructions are executed can be swapped. This procedure is used in further jump tests to determine whether an instruction may jump a node or a node jump a node. For example, the procedure `jump_n_n(`n_j`,`n_k`)` determines whether all instructions in node n_j can jump all instructions in node n_k; if so, the control nodes may be swapped.

4.1.7 MOVE TESTS

The move test determines whether an instruction i can be moved from node n_j to node n_k, where n_j precedes n_k. The test is performed by the procedure `test_move_one_inst(i,`n_j`,`n_k`)`. For the test to succeed three conditions must be met:

1. There must be a non-divergent path between the pair of nodes. This is determined by traversing the graph from n_j to n_k, each node visited (excluding n_k) must have a single feed-forward output arc.
2. Instruction i must be able to jump each node visited in condition 1 (excluding n_j and n_k). This is determined using the `jump_i_n(i,n)` procedure.
3. There must be no access violation between instruction i and the final node n_k. This is tested for using the `contention_ij_nk(`i_j`,`n_k`)` procedure. A dependency may exist between them in which case the instruction is placed at the start of the dependent instruction graph.

The instruction is not tested with n_j and therefore it is assumed that all instructions dependent on i in the same node will also be moved. The procedure `test_move_insts(i,`n_j`,`n_k`)` tests each instruction in the list given by i for moving from node n_j to node n_k.

4.1.8 DELAY TESTS

Delay tests are required to ensure that an instruction or set of instructions when inserted in a control node will not make the control node exceed the clock period. There are two procedures for delay testing a single instruction and an instruction graph, `combine_i_n_delay_test(i,n,time)` and `insts_delay_test(i,n,time)` respectively. The instruction(s) are added to the given node using the `add_inst_group(i,n)` procedure which inserts the dependency arcs that make up the instruction graphs.

The `add_inst_group(i,n)` procedure operates as follows: when inserting the instructions the contention test is made between each instruction to be added to the node and all instructions in the node. The result of the test determines how the instruction is to be inserted. If no contention exists then the instruction is entered in the node as a separate instruction graph with its own group number. If a dependency exists then the instruction is added to the instruction graph to which it depends by inserting a dependency arc. The group number for the instruction will be the same as the instruction graph group number to which it depends. If an access violation occurs then the test fails as instructions cannot be inserted into the node, in this case an "instruction insertion" error code is returned from the procedure.

Once all instructions have been successfully added to the node its new delay is determined using the `calc_node_delay(n)` procedure. If the new node delay exceeds the parameter *time* then a "delay test failure" error code is returned, otherwise a "delay test pass" code is returned. In all cases the instruction(s) are removed from the node and the graph structure reverts to as it was before calling the delay test procedure.

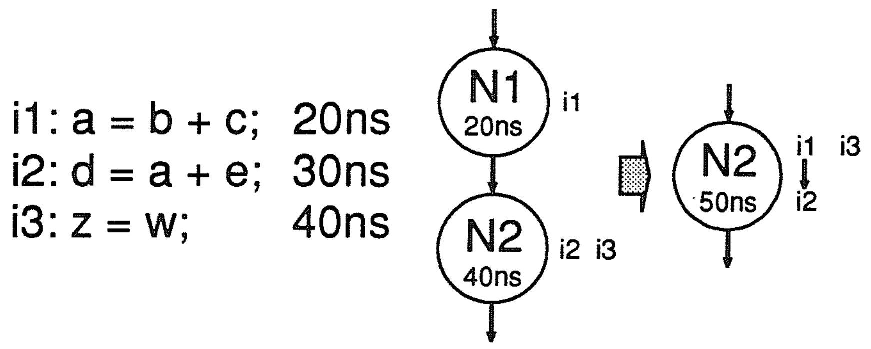


Figure 4.6 Example of delay test.

Figure 4.6 illustrates the delay test involved in merging a pair of control nodes. The set of instructions in node N2 are tested for inclusion in node N1. Instruction i2 is dependent on i1 through variable *a*, resulting in an instruction graph which establishes the new node time of 50ns. The delay test will fail for *time*s of less than 50ns. Note that if the destination variable of i3 had been *a* instead of *z* then an access violation would occur and the test would fail regardless of *time*.

4.1.9 LIFETIME ANALYSIS

The problem of lifetime analysis is well understood in the area of software compilation [83] and some similarity to lifetime analysis in silicon compilation exists. A variable is *live* when it contains data which is required by subsequent instructions and if destroyed would affect the behaviour of the design. A live period starts from the time when the data is created, written to the variable, t_w, to the time when it is last used, read from the variable, t_r. The lifetime of a variable, L_{var}, may consist of many such live periods (t_{w1}-t_{r1}, t_{w2}-t_{r2}, ... , t_{wn}-t_{rn}) between which occur the *dead* periods. A register is *active* over the times that the variables stored by it are live, that is the register active time is the union of the lifetimes of its variables.

Figure 4.7 shows a sequence of control nodes containing instructions that write to or read from variables *a*, *b* or *c*. The instructions themselves are not relevant in illustrating lifetime analysis, only the variable and read or write operation performed by the instruction, shown by the r or w subscript, are important. The time slot, *t*, to which a control state belongs is determined by numbering the control nodes using a variation of the traverse graph procedure shown in Section 4.1 (page 68). The start node is given the time slot *t*=1 and the evaluation `max(predecessor t) + 1` is used to determine the subsequent node time slots. Using the time slot information the variable lifetimes can be determined (see Figure 4.7). The register active times are determine using the `find_active_times(reg)` procedure which returns a linked list of active periods for the given data path register. The input, output and some selected control nets are used to determine the register active times. Each net contains fields specifying the variable transmitted and the instructions writing to and reading from the net, that is, the source and sink instructions. The time

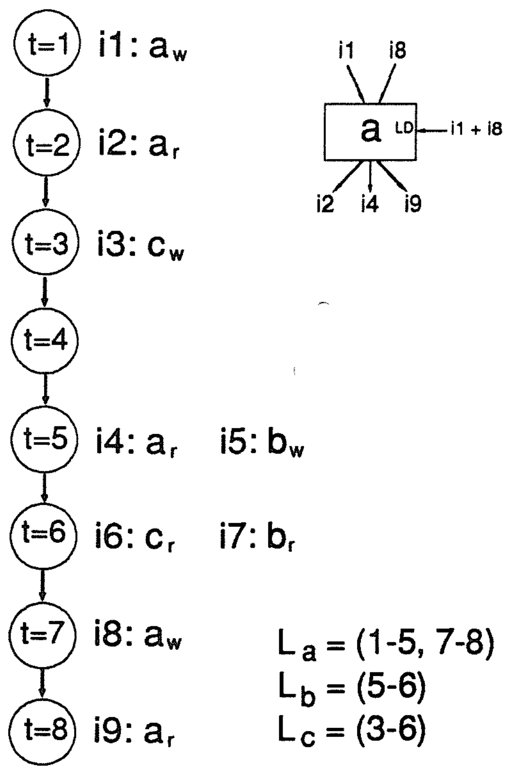


Figure 4.7 Example of lifetime analysis.

slot for each instruction is found by locating the control node containing it. The instruction and variable information are entered into the list of register access times in time order. Register *a* and the instructions accessing it are shown in the top right corner of Figure 4.7.

Additional access times must be entered to accommodate control loops and register controls such as count or shift. Control loops must be taken into account to ensure that data used from one iteration to the next is preserved. If a loop starts at a time slot within a register active period then the first access to the register within the loop must be a read; therefore the variable being read must be maintained to the end of the control loop so that it is correct on the next iteration. This is achieved by entering an additional read at the end of the loop. The feedback arcs, which mark the start and end of control loops, are used to determine where additional read accesses are required.

Register control inputs are associated with write operations, as in the case of set and clear controls. However, register controls such as count or shift represent the modification of data held by the register. As count and shift controls imply the use of existing data a read access must be entered into the access list with the write access.

The register access list is pruned to remove unnecessary accesses; there are three cases where accesses can be removed:

1. a read can be removed if another read follows it and it is not the first,
2. a write can be removed if another precedes it and it is not the last, or
3. both a read and write can be removed if they occur at the same time slot, thereby concatenating two abutting live periods.

Case 2 implies that the earlier write is redundant, however this may not be the case as the second could be a conditional operation.

The final pruned list of register access times consists of alternate write/read accesses where live periods occur from write to read times. The access list details the live periods for all variables stored by the register. Access times for specific variables could be obtained by filtering the variables when scanning the register nets by using their variable transmitted fields.

In register sharing it is necessary to test whether two registers have overlapping live periods; this is done by the procedure `test_non_overlapping_times(a1,a2)` as shown in Figure 4.8. As a register write occurs at the end of a control state and reads within control states, the end of one live period, the read, may occur in the same time slot as the start of another, the write, without overlapping. This is the case with the non-overlapping lifetimes L_a and L_b of Figure 4.7, where the end of the first live period of L_a and the start of L_b occur at t=5. The lifetime L_c overlaps with both L_a and L_b.

```
test_non_overlapping_times(a1,a2)
struct access_list *a1,*a2;
{
get first write-read pair from a1 (wr1,rd1);
get first write-read pair from a2 (wr2,rd2);
while (write-read pairs) {
    if ((wr1<=wr2 && rd1>=wr2) || (wr1>=wr2 && rd2>=wr1))
        return false;                   /* overlap    */
    if (wr1<wr2) get next write-read pair from a1 (wr1,rd1);
    else get next write-read pair from a2 (wr2,rd2);
    }
return true;                            /* no overlap */
}
```
Figure 4.8 Procedure to test for non-overlapping lifetimes.

4.2 SCHEDULING TRANSFORMATIONS

The scheduling transformations are primarily concerned with changes in the control path graph. Their only effect on the data path is to add or change control signals and registers and to change the state machine implementing the control path. These effects are taken into account when calculating the cost function.

4.2.1 Sequential Merge Transformation

The sequential merge transformation attempts to merge two control graph nodes, n_j and n_k. The two nodes are tested to ensure that they are in the same sequential section of the control graph and that n_j precedes n_k. The instruction lists associated with each node are tested for hardware sharing using the `test_sharing(i_j, i_k)` procedure and the instructions in n_j tested for moving to n_k using the `test_move_insts(i_j, n_j, n_k)` procedure. If the move is possible then the `insts_delay_test(i_j, n_k, ck)` procedure is performed to ensure the clock period is not exceeded when the nodes are merged. If any of the tests fail then an appropriate error code is returned.

The transformation is performed by calling the procedure `merge_seq_nodes(td)` which adds the instructions in n_j to n_k using the procedure `add_inst_group(i_j, n_k)`. It also calculates the new node delay using `calc_node_delay(n_k)` and attempts to remove node n_j from the control graph using `remove_control_node(n_j)`. Figure 4.9 shows the successful application of the sequential merge transformation on nodes N1 and N3. Note that instruction i1 has jumped node N2 and formed an instruction graph with i3.

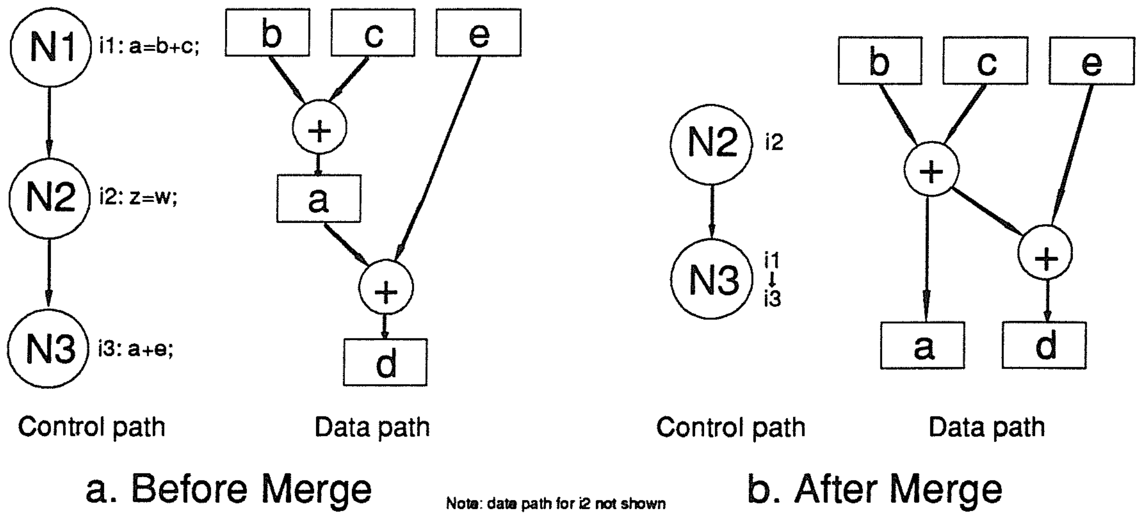


Figure 4.9 Example of the sequential merge transformation.

In addition to the operation of the procedure `add_inst_group(i,n)` as described above, the procedure has a parameter (not shown) which indicates when the instruction group to be added will be a permanent change to the design, as in the execution of transformations, rather than a temporary change as in their testing. When the parameter indicates a permanent change the register on which a dependency between two instructions exists will be bypassed (see Figure 4.9b). The register must be bypassed to ensure the correct behaviour is maintained, that is the dependency arc between i_1 and i_3 indicates that the new value of variable *a* is input to i_3 rather than the previous value. As registers are loaded at the end of each clock cycle the new value of *a* would not have replaced the previous value until the end of the control state, thus leaving insufficient time to complete the dependent instruction.

To bypass a register the variable access for the reading instruction is made from the output of the writing instruction and not the register output. This transformation is only performed in some systems [20] when the variable is a temporary one generated by the compiler and has single read and write instructions. This is because a distinction is made between user defined and compiler generated variables. Therefore in these systems the pair of instructions would have originally been one in the form: d = b + c + e; whereas in MOODS the instructions may have been written as separate instructions with variable *a* being user defined. The MOODS compiler therefore performs a behavioural optimisation that merges equations in the original description. The register is retained in the data path and loaded with the value taken by *a*. However, if no further accesses are made from it, shown by a null output netlist, then it is not included in the cost calculations and is removed from the data path after optimisation.

The `remove_control_node(n)` procedure attempts to remove an empty control node, that is one containing no instructions, from the control graph while maintaining the correct graph structure. The node must have a single input or single output arc. Once the node has been deleted the input and output arcs are connected. If the node had a single input arc then it is deleted and the output arc list connected to its start node. Similarly, if the node had a single output arc then this is deleted and the input arc list connected to its end node. The newly re-connected list of arcs is analyzed and if any arc connects a fork node to a collect node and has a *true* gate condition then it is deleted and the collect N tokens reduced by one. The collect function is not required when the number of tokens reaches one. Parallel arcs, that is arcs having the same start and end nodes, may have been created by the removal of the node. These are merged using the procedure `merge_para_out_arcs(n)` which combines the arc gate conditions and deletes all but one of the parallel arcs. If only one arc remains between the start and end nodes then their role as fork and join nodes is no longer necessary therefore the procedure attempts to remove them by calling `remove_control_node(n)`. The recursive nature of the `remove_control_node(n)` procedure ensures that all unnecessary nodes are removed.

4.2.2 Parallel Merge Transformation

The parallel merge transformation attempts to merge a subset of the successor nodes of a given fork type node that are connected through arcs having the same activation condition. Each successor node with the chosen input arc condition is tested for only having one input arc (that from the fork node) and is marked for merging.

If the tests are successful and more than one node is marked for merging then the instructions in each marked node other than the first are moved into the first node, see Figure 4.10a and Figure 4.10b. The instruction graphs in the nodes will remain executed in parallel therefore no hardware sharing or contention tests require to be performed. The output arcs of the empty nodes are concatenated with those of the first node and the empty nodes removed using the `remove_control_node(n)` procedure as in Figure 4.10b. As a result of the concatenation of output arcs any parallel arcs which exist will be merged by the `merge_para_out_arcs(n)` procedure. The delay of the resulting node will be the maximum delay of the individual nodes.

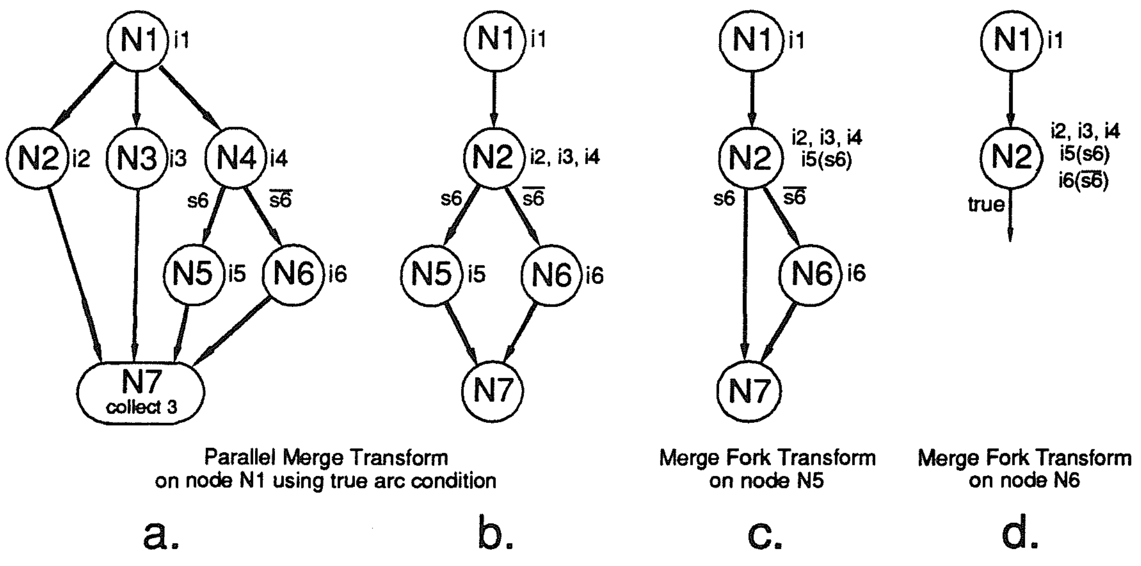


Figure 4.10 Example of the parallel and fork merge transformations.

4.2.3 Merge Fork and Successor Node Transformation

This transformation merges a node, n_n, given as a successor to a fork type node with the fork node itself. The instructions, i_n, in node n_n are tested for contention with those in the preceding fork node, n_f, using the contention_nj_nk(i_f, i_n) procedure. If an access violation occurs then the transformation fails. A dependency type contention will cause dependency arcs to be added by the add_inst_group(i_n, n_f) procedure when the transform is performed. The tests test_sharing(i_n, i_f) and inst_delay_test(i_n, n_f, clock) are also performed to ensure that no hardware sharing violation occurs and that the resulting node delay is shorter than the clock period.

If the tests are successful then the transformation may be performed by calling the add_inst_group(i_n, n_f), calc_node_delay(n_f) and remove_control_node(n_n) procedures. For example, Figure 4.10c, shows N5 merged with N2; note that instruction i5 now has a condition for firing s6 and that the arc with the gating condition of s6 has not been deleted. However, on merging N6 with N2, as in Figure 4.10d, N6 is removed and the resulting parallel arcs between N2 and N7 are merged (by procedure merge_para_out_arcs(n)) causing the combined gate condition to be a tautology, indicated by the presence of a single arc from N2. In addition, the empty node N7, previously a join type node, is also removed.

4.2.4 Group on Register Transformation

This transformation is primarily concerned with reducing the number of registers in the data path. As described in Section 4.2.1 (page 79), a register can be bypassed if the writing and reading instructions are in the same control state. If the register is only used by these instructions (as in register *a* of Figure 4.9) then it can be removed from the data path. This transformation takes a register with one writing instruction and one reading instruction and attempts to combine the instruction groups to which they belong into the same control node. The groups must be moved rather than the instruction as by moving an instruction from a group to another node requires the insertion of a register, therefore no overall gain would be made. The instruction group associated with the writing instruction is tested for (a) sharing hardware with the instructions in the node containing the reading instruction, (b) moving to the reading node and (c) the combined instructions exceeding the clock period.

If the tests are successful then when the transform is executed the writing instruction group is added to the node containing the reading instruction. If this results in the writing node having an empty instruction list then an attempt is made to remove it from the control graph. Figure 4.9 shows the result of the group on register transformation when applied to register *a*. Note that the transformation is equivalent to a behavioural transformation which expands instructions. For example in Figure 4.9 the separate instructions i1 and i3 have been combined into one (i3) shown by the dependency arc in the resulting instruction graph, therefore i3 becomes `d=b+c+e`.

This transformation is similar to grouping in Scholyzer [20], however in MOODS it is applied during optimisation and not as a separate stage before. It can be applied to any register having one read and one write instruction regardless of whether it is user defined or compiler generated. The advantage of applying this transform during optimisation can be seen in reference to Figure 4.9. In Scholyzer the grouping is done before optimisation therefore eliminating the possibility of sharing.the adders used by instructions i1 and i3, whereas MOODS may perform either grouping or sharing during optimisation, the final decision being dependent on the user's objectives.

4.2.5 Ungrouping Transformations

The ungrouping transformations allow a degradation in the control graph by splitting nodes into sequential sections. This may then allow hardware sharing between the ungrouped instructions and so facilitate trade-offs between objectives in the cost function. There are two ungrouping procedures `ungroup_group(n,g)` and `ungroup_time(n,t)` which ungroup the instructions in a given node *n* either by extracting a given group *g* or dividing the node into a set of nodes each having a delay, for groups of more than one instruction, no longer than time *t*. In the latter ungroup transformation more than one control node may be created.

The procedure `extract_inst_group(n,g)` is used to extract the instruction group *g* from the given node. Instructions extracted from the control node are tested against the instructions remaining in the node using the `test_contention(i`$_{ext}$`, i`$_n$`)` procedure to ensure no extra dependency is created between them when the extracted instructions are executed before the control node. If no dependencies are created, a new control node is inserted before node *n* and the extracted group is entered as its instruction list.

In procedure `ungroup_time(n,t)` a subset of the instructions in the node are extracted by the procedure `extract_insts_greater(n,t)` such that their *end_times* are less than time *t* or they have no predecessor dependent instructions. The subset of instructions are tested for creating additional dependencies. If none are created a new control node is inserted before node n and the subset of instructions entered into it. The ungrouping of node n is repeated until the set remaining after extracting the instructions becomes the empty set. The testing stage of the transformation extracts subsets of instructions and tests for the creation of extra contention; each subset is added to a dummy node which at the end of testing is transferred back to the original control node. When performing the transformation each subset is entered into its own node. The dependency arcs between the subset and other subsets are removed and the corresponding registers included in the data path. To include a register access back into the data path requires a further test to ensure that the register is not in use during the execution of the control node. The additional use of the register may occur if the register had been shared.

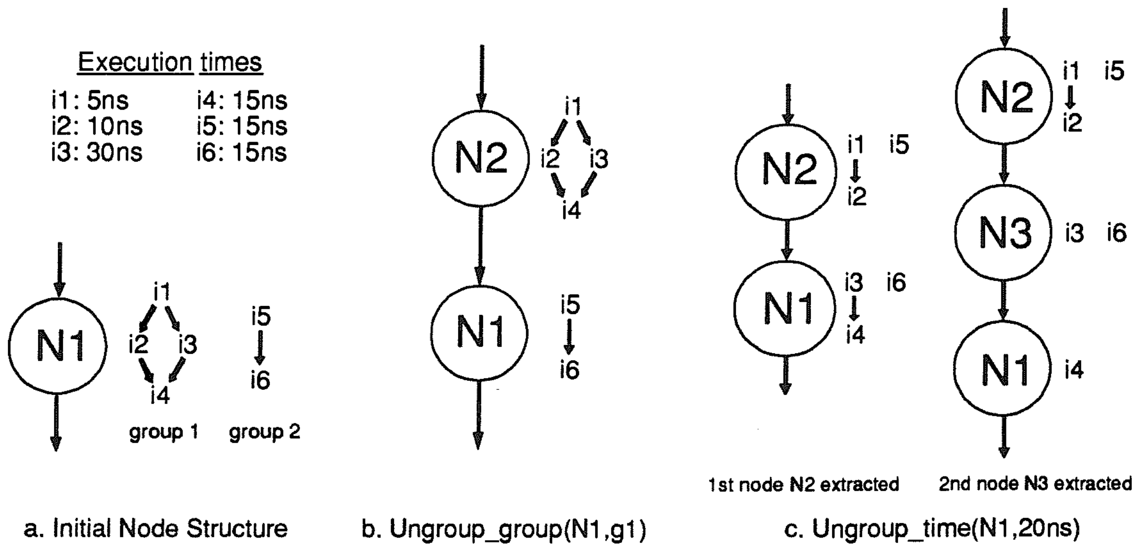


Figure 4.11 Example of the ungrouping transformations.

An example of both ungrouping procedures is shown in Figure 4.11. In the ungrouping a group transform, group 1 has been extracted into the preceding control node N2. In the ungroup on time transform, the time chosen was 20ns resulting in two extra control nodes being generated. Note that instruction i3, having an execution time of 30ns, was selected for node N3 because it was the start of the instruction graph after the first node was extracted and not selected on its *end_time*.

4.2.6 Multicycle Transformation

Multicycled instructions are continued into successor control nodes by the addition of dummy instructions. The dummy instructions are given instruction numbers equal to the negative of the original instruction and point to the data held by the original.

The multicycle transformation, `multicycle_control_node(n,ck)`, is not strictly a transformation to be used on its own. It is used to maintain the clock period given by the user which is considered an absolute objective that must be met whatever the cost. The transformation is applied to the set of instructions within a control node. Each instruction to be multicycled is assumed to be in a separate group which can be ensured by ungrouping the node into time slices using the procedure `ungroup_time(n,ck)`. For each instruction whose *end_time* exceeds the given clock period the instruction is flagged and the number of additional cycles required to execute the instruction is calculated (given by *trunc(end_time/clock)*). The maximum number of cycles is noted. For the maximum number of cycles a new control node is created and each flagged instruction i_f analyzed.

If the delay of i_f is less than t, where t equals twice the clock period minus the register access time for i_f, then a final dummy instruction $-i_f$ is created and entered into the new node. The instruction i_f is un-flagged and the delay of $-i_f$ set to the delay of i_f minus the clock period *ck*. If the result of this is less than or equal to zero then it was the register access time for the instruction that caused the violation of the clock period. In this case the delay of $-i_f$ should be set to zero. This will cause the output registers for the instruction to be loaded on the clock cycle following the original instruction. If, however, the result is positive then the original instruction delay is set to the clock period minus the register access time. That is, the original instruction utilises all of the clock cycle and the dummy instruction completes the instruction's execution.

If the delay of i_f is greater than *t* then the delay of $-i_f$ is set to the clock period *ck* and the original delay decremented by *ck*. Figure 4.12 illustrates the multicycle transformation for two instructions whose register access times are assumed to be zero.

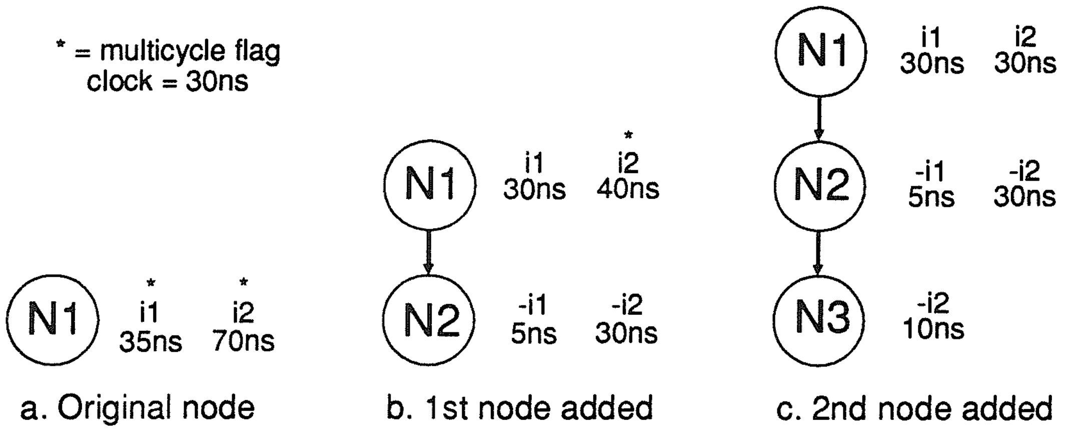


Figure 4.12 Example of the multicycle transformation.

The clock period chosen is 30ns therefore an additional two control nodes are required. Figure 4.12b and Figure 4.12c show the sub-graph created after the addition of each node.

4.2.7 Clock Period Transformation

The clock period transformation is used to set or change the clock period. The only test performed is to ensure that the period chosen is greater than the register access time plus its set up time. The transformation is performed by firstly un-multicycling each instruction using the procedure `un_multicycle_inst(i)`, which deletes the dummy instructions -i and resets the delay of the original instruction i. Next, if the clock period is being decreased (which it is when being set as the clock period is assumed to be arbitrarily long when not set) then each control node must be ungrouped to the clock period by calling the procedure `ungroup_time(n,clock)`. Lastly each control node that exceeds the clock period must be multicycled using the transformation, `multicycle_control_node(n,clock)`, described in Section 4.2.6.

4.3 ALLOCATION TRANSFORMATIONS

The allocation transformations are primarily concerned with changes in the data path. As the data path is a structural representation of the design the transformations involve manipulating the units and cells that implement them. As mentioned in Section 3.3.2 on page 60 the instructions are mapped to a set of data path units which are implemented using a *basic* set of parameterized cells. The mapping is done by direct compilation which produces the initial data path graph. In addition to the *basic* cell set, the user may add additional cells to the cell database using the cell database editor. Using the editor the user enters technology dependent details such as area, power, inherent delay, input pin capacitances and delay factors. In addition to the technology dependent data the user also enters behavioural data such as the functions that the cell implements. From this the program can determine which cells can be used to implement a given set of functions.

4.3.1 COMBINE UNITS TRANSFORMATION

Both hardware sharing and ALU creation are performed by the combine units transform. The transformation takes as its input a pair of data path units and attempts to combine them. Its success is highly dependent on the comprehensiveness of the cell database. The procedure `test_combine_units(td)` tests the data path units. The units must be functional types and the instructions that they implement must be non-concurrent. The non-concurrency of the instructions means that no two instructions can be executed together, which is determined by testing every pair of instructions. If they are mutually exclusive then they are never executed together and no further tests are made. If not then they are tested for occupying control nodes which are reachable but different using the reachability test. If this test fails for any instruction pair, the transformation fails and the procedure returns an appropriate error code.

When the tests have been successful the test procedure searches the cell database for the subset of cells which implement the set of functions implemented by the units to be combined. This is defined as:

$$C(u_1,u_2) = \{c \mid c \in \text{ set of cells which satisfy } (f(u_1) \cup f(u_2)) \subseteq f(c_n),\ c_n \in \mathbb{C}\}$$

where $f(u_{1,2})$ are the sets of functions that each unit implements, $f(c_n)$ is the set of functions cell c_n implements and $\mathbb{C}$ is the set of cells in the database. If $C(u_1,u_2) = \varnothing$

then no cell exists in the cell library to implement the combination of functions $f(u_1) \cup f(u_2)$ and a "units can not be combined" error code is returned from the procedure.

If cells exist then the cell $c \in C(u_1,u_2)$ implementing the least number of functions is selected as the combined cell. The reason for this selection is so that when combining units of equivalent function, as in hardware sharing, the same or equivalent cell will be selected that implements the one function. For example, two adders will be combined into an adder and not an ALU implementing the addition function. The cell selection process can combine any mixture of ALUs and basic cells to form new ALUs as long as a suitable cell is contained in the cell database.

To perform the transformation the procedure `combine_units(td)` is used. This concatenates the lists of input nets, output nets, control signals and implementation links into one unit and deletes the other. The execution time of the instructions implemented by the new unit must also be updated and any multicycling must be performed. If an ALU has been created then its select inputs must be added. These are control signals which select the appropriate ALU function and are also used to maintain the correlation between the unit function and the instructions. Figure 4.13 shows the creation of an ALU implementing plus and minus functions. Note that the implementation links of instructions i1 and i2 now point to the same data path unit thereby preventing them from being merged into the same control node.

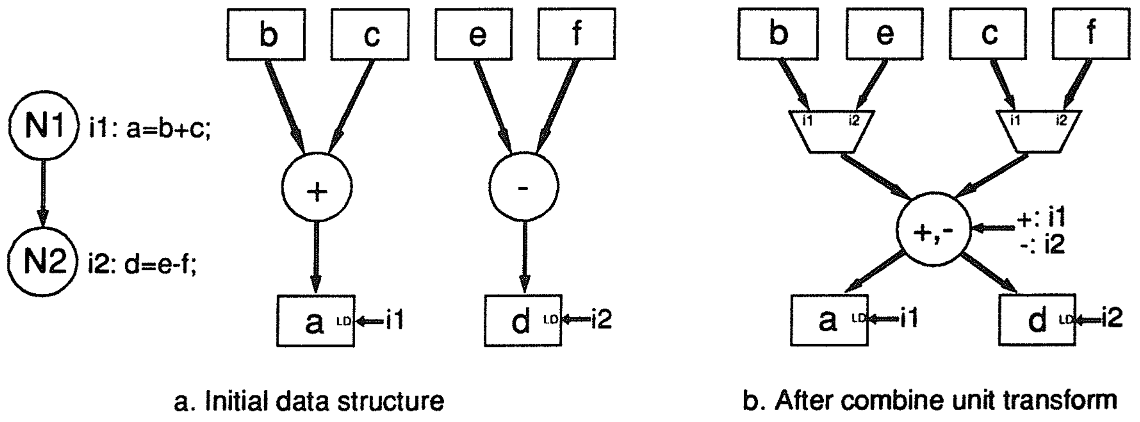


Figure 4.13 Example of the combine units transformation.

4.3.2 Uncombine Unit Transformations

There are two transformations which uncombine data path units that were previously combined using the combine units transform described in Section 4.3.1. No tests are required, except for checking the entered data, as the changes made to the design deal with the design at only the structural level. The first transformation is performed by the procedure `uncombine_single_unit(td)`, which divides the original unit into two separate units. A new unit is created by removing an instruction implemented by the original unit and implementing it using a basic cell. The inputs, outputs, control signals and implementation links associated with the instruction are moved to the new unit. The select control signal is no longer required as the new unit only implements one instruction and can therefore be deleted.

The second transform is performed by the procedure `uncombine_unit(td)`, which for each instruction implemented by the original unit, indicated by the implementation links, creates a basic data path unit. That is *n* units, implemented by basic cells, are created where *n* is the number of instructions implemented by the original unit. The inputs, outputs and control signals associated with each instruction are moved to their corresponding units and all select control signals are deleted. The separation of nets and control signals is straightforward as each data structure has fields indicating the variable transmitted on it and the instruction which creates and uses it.

For both uncombine transformations each instruction's execution time is reset and multicycling is performed where necessary. In the example shown in Figure 4.13 the uncombine transform is the opposite of the combine transform and in this case the transformation from Figure 4.13b to Figure 4.13a could have been performed by either of the uncombine transformations.

4.3.3 Alternative Implementation Transformation

An alternative cell may be found by selecting one from the set of cells that implements the functions performed by the unit. A different cell may be selected which better utilises the clock period. For example, a cell may be chosen to reduce the slack or spare time within a control node and so provide a further trade-off mechanism with other design aspects such as area. Alternatively a cell may be chosen that reduces the delay of the maximum delay control node which sets the clock period thereby allowing a reduction in the clock period and circuit delay. Like the uncombine unit transform the changes to the design occur at the structural level, however a test must be made to ensure an alternative cell exists.

The set of alternative cells is determined using the method described in Section 4.3.1, which finds the subset of cells capable of implementing a combined unit. For example, the alternative implementation to a carry propagate adder would be a carry look-ahead adder as both implement the same PLUS operation. The alternative cells found will depend on the comprehensiveness of the cell database. If an alternative implementation is found then the implementation of the data path unit can be changed by altering its cell pointer and updating its technology dependent data. The execution time of instructions implemented by the unit are updated as before.

4.3.4 Register Sharing Transformation

The register sharing transformation, performed by the `share_registers(td)` procedure, combines two storage units, that is registers and counters. Memory (ROM and RAM), module ports or I/O registers may not be shared. In order to share two storage units they must either be mutually exclusive or have non-overlapping lifetimes; the tests being performed by the procedures `test_reg_mutual_excl(`r_1, r_2`)` and `test_non_overlapping_times(`a_1, a_2`)` respectively. If the units can be shared then the implementation links and input, output and control nets for both units are concatenated. The nets are assigned to one of the units which becomes the shared unit and the other one is deleted. The bit width of the resulting unit is made wide enough to accommodate all variables stored in it. Ideally all variables should be normalised to a common low bit in order to minimise the bit width of shared storage units.

As a result of register sharing pure data transfers may be removed from the behavioural specification. This is illustrated in Figure 4.14, where registers storing variables *a* and *c* have been combined. The resulting register has a *self load* caused by the pure data transfer, i2. Self loads are retained in the data structure during synthesis to maintain the correlation between nets and instructions, however they are marked for deletion after

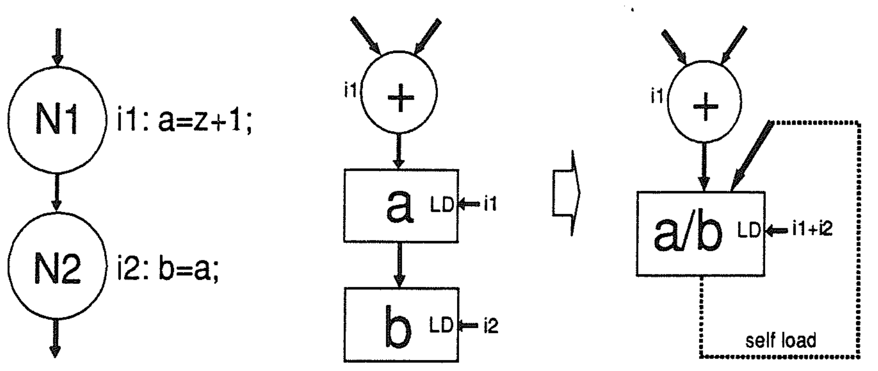


Figure 4.14 Removal of pure data transfers by register sharing.

optimisation. It is interesting to note that the same final data path would be obtained if instruction i2 had been moved to node N1 causing register *a* to be bypassed.

4.3.5 Register Unsharing Transformations

As with the uncombine unit transforms there are two transformations which unshare storage units that were previously combined using the register sharing transformation described in Section 4.3.4. Again no tests are required, except for checking the entered data, as the changes made to the design deal with the design at only the structural level. The checks consist of ensuring the hardware exists and that it stores more than one variable. The first transformation is performed by the `unshare_single_reg(td)` procedure which separates a single variable from a storage unit. The implementation links and input, output and control nets relating to the given variable are extracted from the storage unit pointed to by the hardware field of the variable structure. A new storage unit is created and the extracted nets and variable hardware pointer assigned to it. Both the original and new units require the bit widths, cell type and area to be set.

The second register unsharing transformation, performed by the `unshare_reg(td)` procedure, unshares all variables stored in a given storage unit. The variable list is scanned and the `unshare_single_reg(td)` procedure called for each variable with a hardware pointer pointing to the given unit.

5 OPTIMISATION ALGORITHM

In the context of intelligent silicon compilation the iterative optimisation of a design can be divided into two parts, the evaluation of the design and the optimisation algorithm. The algorithm applies the transformations, described in Chapter 4, to the design based on its evaluation. This chapter is divided into two parts corresponding to the evaluation of the design and the optimisation algorithms.

5.1 DESIGN EVALUATION

5.1.1 THE COST FUNCTION

The design is evaluated with the aid of the cost function, which represents the state of the design within the design space. It is used in conjunction with the user's multiple objectives to guide the optimisation algorithm. As described in Section 1.5.2, the cost function must be accurate and in terms of absolute design aspects, rather than control state or device counts, although these are useful in providing a comparison between synthesis systems. The design aspects to be monitored by the cost function must be global, that is, they evaluate the design as a whole and not a sub-section of it.

The accuracy of the cost function is ensured by feeding up technology dependent information from the cell database. The area and power used by each control and data path unit and the execution time of each instruction are calculated using this information, which in turn is used in the cost function. An entry exists in the cell database for each parameterized cell that the MOODS system may use in an implementation. The database information consists of the cell bit width, area, power, propagation delays and pin overheads including delay factors and input capacitances. Register set-up times and some special information particular to certain cells is also included.

For multiple objectives the cost function must be flexible in giving the user freedom to specify objectives on any number of design aspects that the system monitors. It should also incorporate information on the optimisation priority given to the selected design aspects, which can be used by the optimisation algorithm to indicate the effectiveness of a transformation and determine how trade-offs should be performed. For these reasons a cost function similar to that used in goal programming applications [78] was used. Goal programming attempts to satisfy objectives as closely as possible rather than absolutely as in traditional linear programming methods. The cost function (achievement function in goal programming terms) consists of a priority vector, where each priority element specifies design aspects on which objectives have been set. In goal programming only objectives expressed in common units can be assigned the same priority as no comparison can be made between different quantities.

Each objective function G_i is expressed as a function of the data structure describing the design's data and control path, *dcp*, which is analogous to decision variables in mathematical programming; thus:

$$G_i = f_i(dcp) \tag{5.1}$$

In goal programming terms an objective function is expressed as follows:

$$f_i(dcp) + (n_i - p_i) = b_i \qquad where \quad n_i, p_i, b_i \geq 0 \tag{5.2}$$

b_i is the goal or target value which $f_i(dcp)$ must either satisfy, exceed or be less than. n_i and p_i are the negative and positive deviations of $f_i(dcp)$ from its target, b_i, respectively. In general it is desired to select *dcp* such that $f_i(dcp)$ is either, (a) greater or equal to b_i, (b) less than or equal to b_i or (c) equal to b_i which is achieved by minimising a linear function of the deviation variables, g(n,p); that is, either (a) minimise n_i, (b) minimise p_i or (c) minimise $n_i + p_i$, respectively.

To formulate the achievement function, the function g(n,p) for each objective is associated with a priority level, therefore:

$$\boldsymbol{a} = \{P1[G_1(\boldsymbol{n},\boldsymbol{p})],\ P2[G_2(\boldsymbol{n},\boldsymbol{p})],\ \dots,\ Pn[G_n(\boldsymbol{n},\boldsymbol{p})]\} \tag{5.3}$$

where $G_k(\mathbf{n},\mathbf{p})$ is the kth linear function of deviation variables. The size of **a** is equal to the number of priorities which is less than or equal to the number of objectives. As **a** is an ordered vector the P's can be dropped, therefore:

$$\boldsymbol{a} = \{G_1(\boldsymbol{n},\boldsymbol{p}),\ G_2(\boldsymbol{n},\boldsymbol{p}),\ \dots,\ G_n(\boldsymbol{n},\boldsymbol{p})\} \tag{5.4}$$

which is minimised by the optimisation algorithm. $G_1(\mathbf{n},\mathbf{p})$ is the highest priority function, which in goal programming represents absolute objectives, that is, ones which must be achieved. An achievement function $\mathbf{a}_1$ is considered better than $\mathbf{a}_2$ if the first non-zero component of $\mathbf{a}_1$ - $\mathbf{a}_2$ is negative given that all components of $\mathbf{a}_1$ and $\mathbf{a}_2$ are non-negative.

The MOODS cost function is a variation of the goal programming achievement function and allows any objective to be associated with a priority. As objectives measured in different units are not directly comparable the function of deviation variables, $G_k(\mathbf{n},\mathbf{p})$, is replaced by a vector of g(n,p) functions for the objectives associated with priority k, therefore, $G_k(\mathbf{n},\mathbf{p})$ becomes:

$$g_k = \{g_{kl}(n_1,p_1),\ g_{k2}(n_2,p_2),\ ...,\ g_{km}(n_m,p_m)\} \qquad \textbf{(5.5)}$$

where each $g_{ki}(n_i,p_i)$ represents an objective given by equation (5.2) above.

In synthesis not only is it required to know whether one cost function is better than another but also by how much. Typically the two cost functions are $\mathbf{cf}_{est}$ and $\mathbf{cf}_{pres}$ representing the cost functions for the next and present designs respectively. The difference between the two cost functions, ΔE, represents the change in energy between the functions and thus the design. ΔE is determined by constructing a third vector, E, the energy change vector, whose elements represent the change in energy at each priority, thus:

$$E = \{E_1(g_1),\ E_1(g_1),\ ...,\ E_n(g_n)\} \qquad \textbf{(5.6)}$$

where $E_k(\mathbf{g}_k)$ is the combined objective change for priority k. For each g_{ki} in $\mathbf{cf}_{est}$ and $\mathbf{cf}_{pres}$ the change between them, Δc_{ki}, is evaluated and for each priority level, k, their average is taken, thus:

$$E_k = \frac{\sum_{i=1}^{m} \Delta c_{ki}}{m} \qquad \textbf{(5.7)}$$

where m is the number of objectives at priority k. The function used to determine Δc_{ki} may take into account other objective factors such as initial or target values, however their difference was initially taken, thus:

$$\Delta c_{ki} = g_{ki}(n_i,p_i)_{est} - g_{ki}(n_i,p_i)_{pres} \qquad \textbf{(5.8)}$$

The change in energy, ΔE, between the cost functions is given by the first non-zero component, E_k, of the energy change vector, E. The cost function $\mathbf{cf}_{est}$ is considered better than $\mathbf{cf}_{pres}$ if ΔE is negative.

In synthesis it is desired to select *dcp* such that $f_i(dcp)$ is less than or equal to b_i which is achieved by minimising p_i, therefore equation (5.8) becomes.

$$\Delta c_{ki} = p_{ki_{est}} - p_{ki_{pres}} \tag{5.9}$$

The use of the positive deviation in equation (5.9) to determine Δc_{ki} allows changes in objectives that have already reached their target, that is negative deviations, to be ignored and thus permits lower priority objectives to influence ΔE. The methods used to determine Δc_{ki} and whether one cost function is better than another will greatly influence the choices made by the optimisation algorithm and therefore the final design. Experiments to find a better change in energy function are described in the results Section 6.1.1.

The data structure representing the cost function contains both $\mathbf{cf}_{pres}$ and $\mathbf{cf}_{est}$, as well as the initial and target values for each objective (see `struct cost_fn` in Appendix C). The evaluation of the next cost function with respect to the current is performed by the `evaluate(cost_fn)` procedure which returns the value ΔE. The evaluation is done whenever estimating the cost of a transformation, thereby providing information on the effect of the transformation on the user's objectives. A negative value of ΔE indicates that the transform improves the design.

From the graphical method of solving linear and goal programming problems an objective may be termed *reachable* if the area bounded by it, the axis and any higher priority objectives intersects with the achievable region of the design space, otherwise the objective is termed *minimising*. The n-dimensional design space can be characterized, as described in Section 1.7. Each of the n asymptotes is found by optimising the design with a cost function having one of the objectives at priority one with a minimising target. such as zero. All other objectives are at a lower priority, again with minimising targets. The final characterization point lies as close to the origin as possible indicating a good all round implementation. It is found using a cost function with all objectives at the same priority with minimising targets, thus giving them an equal opportunity for improvement. The characterization of the design space by the system indicates the range of designs achievable by the system rather than by manual methods or other systems.

The aspects of the design currently monitored by the system and therefore available as cost function objectives are area, power and delay. In addition to these a net count objective can be used which counts the number of one source, one sink interconnects used in the implementation. Any aspect of the design could be monitored and included in the cost function by including the relevant cost calculation procedures. The area, power and delay calculations are described below.

5.1.2 Area and Power Calculations

The area of the design is calculated as follows:

$$area = \Sigma a_{dp} + \Sigma a_{cp} + \Sigma a_i \qquad \textbf{(5.10)}$$

where a_{dp}, a_{cp} and a_i are the areas for a data path unit, a control unit implementing a control node and an interconnection respectively. At present the interconnect costs are not taken into account in the cost calculations as it is difficult to obtain accurate estimations prior to complete layout, however it is recognised that interconnect costs have a significant effect on hardware costs (see Section 7.2).

The area of a data path unit is a function of its type, bit width and the cell that implements it. The area of each unit is calculated as it is created, initially during the construction of the initial data path and subsequently as a result of applying transformations. There are two components that make up the area of a data path unit, the main area and the area overhead of additional connections, "pins". The main area is a multiple (n) of the cell area implementing the unit's combination of functions[1]. The multiple n is derived such that n times the cell bit width is equal to or greater than the bit width of the unit that it implements. Pin overheads may be specified in the cell database to indicate the area overhead incurred if the pin is used[1]. The unit's netlist is

[1] A cell may have different area and delays for different combinations of the functions it is capable of implementing, this is to take into account optimisations performed by low-level optimisation tools which would remove unused functions from a parameterized cell. Similarly pin overheads specify the additional cost of retaining a pin which would otherwise be optimised away by low-level tools.

scanned for pins that incur an overhead which is calculated as a multiple *n* of the pin area.

The power used by a design and by a data path unit is calculated in an identical way to the area described above except that the data structure fields relating to power are used.

5.1.3 Delay Calculations

The delay of an instruction is a function of the inherent propagation delay of the operator and the propagation delay factor of the output, that is:

$$t_i = ipd_{op} + pdf_{op} * \Sigma ic_d \tag{5.11}$$

where ipd_{op} is the inherent propagation delay of the operator, pdf_{op} is the propagation delay factor of the output and ic_d is the input capacitance of the data path unit that the operator drives. These parameters are provided by the cell database. To determine the time required to execute an instruction graph, instruction delays are accumulated as the graph is traversed using a variation of the graph traversal algorithm shown in Figure 4.1 on page 68. The evaluation of an instruction consists of determining its completion time which is given by:

$$t_{end_i} = \max(predecessor\ t_{end}) + t_i \tag{5.12}$$

where the predecessor instructions are given by the predecessor dependency arcs. The completion time for the instruction graph, T_G, is given by the maximum t_{end} time within the graph. The maximum graph time, T_{Gmax}, is used to determine the time required to execute a control state. The register accesses, which occur at the control state boundaries as dictated by the architectural model, are taken into account when determining the delay of a control node, as follows:

$$T_n = ipd_{R_{max}} + pdf_{R_{max}} * ic_{OP_{max}} + T_{G_{max}} + st_{R_{op}} \tag{5.13}$$

where ipd_{Rmax} and pdf_{Rmax} are the inherent propagation delay and propagation delay factor for the input register, R_{max}, to the first instruction in group G, the maximum delay instruction graph. ic_{OPmax} is the total input capacitance driven by register R_{max} and st_{Rop} is the set up time for the output register written to by the last instruction in group G.

Figure 5.1 illustrates the control node delay calculation for a node containing two dependent instructions (one instruction graph).

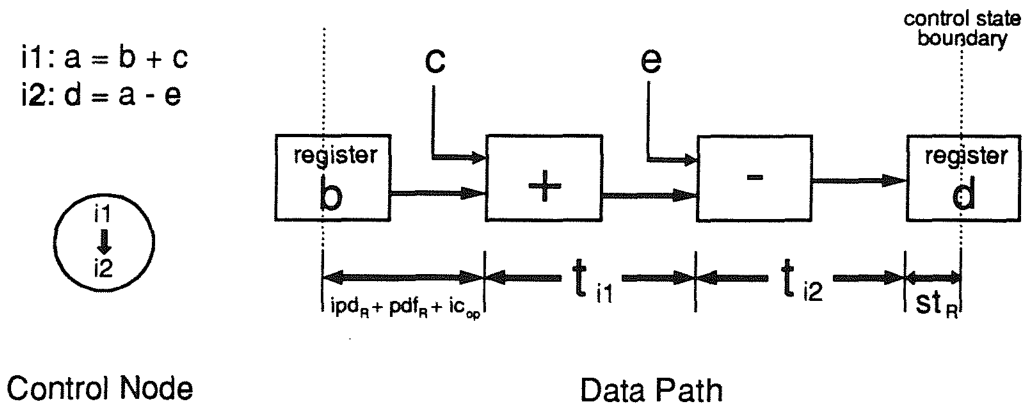


Figure 5.1 Control node delay calculations.

The execution time for a design is defined as the product of the clock period and the number of cycles required to traverse the critical path, as described below. The clock period is taken to be either the maximum control node delay or a user defined clock period which may be specified as an absolute objective.

5.1.4 Critical Path Analysis

Critical path analysis can only be performed on an acyclic graph such as an activity graph used in project management. In an activity graph the nodes represent events and the timing constraints are represented by the arcs. It is necessarily acyclic, for if a cycle did exist, some of the activities could never commence. The calculation of the critical path may be determined using graph algebra [82] and linear programming or the more traditional forward and backward pass methods [84]. The latter method was chosen owing to its simplicity when recursively programmed using the graph traversal algorithms. The forward pass establishes earliest end times and the backward pass the latest end times. From these the slack can be determined and so the critical paths, given by nodes with zero slack.

The main difference between the control graph and an activity graph is that the control graph is cyclic. Therefore it must be made acyclic to allow critical path analysis to be performed. The acyclic graph is created by the removal of the minimum feedback arc set as described in Section 3.3.1 on page 56. Although this results in a usable graph it would be somewhat naive to assume that its critical path represents that of the design. This is because the nodes enclosed by the loops formed by the feedback arcs may be executed many times. The number of iterations of each loop can be taken into account by the use of the *loop_its* field in the control path data structure. Initially all *loop_its* are set to one, indicating that each node is executed once in the acyclic graph. A user specified loop iteration file is read immediately after creating the initial control graph. The file indicates whether a loop is to be taken into account and how many iterations the loop performs. For all the nodes contained in the loop the *loop_its* field is multiplied by the number of iterations performed by that loop. When determining the critical path the number of clock cycles required to execute a node is given by the *loop_its* field.

Nodes on the critical path are those with a slack of zero. There will be at least *c* critical paths, where *c* is the number of end nodes in the control graph. The slack for node *n* is given by:

$$\text{slack}_n = \text{latest end time } (\text{let}_n) - \text{earliest end time } (\text{eet}_n)$$
$$\text{eet}_n = \max(\text{eet}_{\text{pred}}) + \text{loop\_its}_n$$
$$\text{let}_n = \min(\text{let}_{\text{succ}} - \text{loop\_its}_{\text{succ}})$$

Critical path analysis consists of four stages:

1. recursively determine earliest end times,
2. for all end nodes set latest time equal to earliest time,
3. recursively determine latest end times, and
4. set slack to latest end time minus earliest end time (let - eet).

Figure 5.2 shows the critical path analysis for a simple control graph containing a loop of three iterations. The execution time for this example is five clock cycles and the critical path consists of nodes N1, N3 and N5.

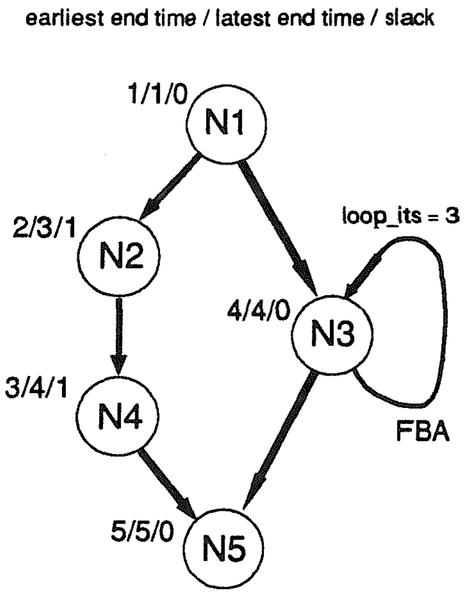


Figure 5.2 Critical path analysis.

It is worth noting that the length of the critical path can be found by performing stage 1 of the critical path analysis and is given by the maximum end time of the end nodes in the control graph. All four stages are performed as the slack information may be helpful during optimisation in selecting the transformation data.

5.2 OPTIMISATION ALGORITHMS

In the MOODS synthesis system the design may be optimised either manually or automatically using an iterative optimisation algorithm. The manual method entails the user to manually apply transformations to improve the design. The user has access to the cost function and evaluation routines to guide him, however it is essential that the user can visualise the design as it is being optimised. This requires the user to draw the initial implementation and update it as transformations are successfully applied. This results in a laborious process when optimising a complete design, however the manual option is essential for making adjustments to an already optimised design either to improve it or to include some of the designer's quirks.

Iterative optimisation consists of selecting transformations and applying them to the design in such a way that the user's criteria are met. The method of selecting and applying transformations constitutes the optimisation algorithm. There are two approaches to iterative optimisation, namely tailored and adaptive heuristics [81]. In the tailored heuristic approach the cost function is analyzed and a transformation chosen and applied depending on the current position of the design within the design space relative to the required position set by the user's objectives. For example, if the user has set an area objective which has not yet been met then a transformation which performs an area reduction is selected. This approach is used in Camad [23] and Chippe [69]. The adaptive heuristic method arbitrarily selects a transformation and its effect on the design is estimated. The transformation is applied depending upon the analysis of the estimation with respect to the user's objectives. This approach is used by Devadas and Newton [42].

There are advantages to both approaches. For example, the tailored heuristic approach guarantees an improvement with each iteration, when no improvement occurs the optimisation ends. However, this leads to local minima. The adaptive heuristic approach may also suffer from local minima traps if a straightforward iterative improvement method, such as pairwise exchange, is adopted where only improvements are accepted. By occasionally applying transforms that have an adverse effect on the design, transitions out of local minima are possible, this is the basis of more advanced adaptive heuristic methods such as simulated annealing [85,86,87]. Transitions out of local minima can not be done in the tailored heuristic approach as transformations are chosen, using a "tuned" heuristic method, to improve the design.

The tailored heuristic approach has the disadvantage that the method used to select a transformation must be changed when incorporating extra design aspects and must survey all changes that each transformation makes to a design. As can be seen in the Camad and Chippe systems this leads to a complex heuristic selection process even for only two or three objectives. In these traditional "tuned" heuristic approaches an essential subtle structure that underlies the problem must be found. Using this knowledge a heuristic solution tuned to the nuances of the structure can be crafted. For multiple objectives the problem becomes "dirty", with numerous, contradictory constraints and complex cost functions [85].

An adaptive heuristic method was chosen for the optimisation algorithm as, by its abstractness from both the design and the transformations, it avoids the construction of complex "tuned" heuristics. This abstraction allows any number of different objectives to be incorporated into the cost function with additional objectives requiring the minimum of program changes. When adding new objectives only the estimation and cost calculation routines require updating, the optimisation algorithm will remain unchanged. Adaptive heuristics although "no substitute for a well designed tailored heuristic"[81], offer the only solution to many "hard" problems such as high-level synthesis; where synthesis tasks are performed simultaneously in order to provide automated design space exploration and multiple objective optimisation.

5.2.1 The General Adaptive Heuristic

For any optimisation problem we require to find a solution that minimises a cost function *cf* subject to particular constraints. A maximisation problem is solved by minimising the negative of the cost function. Solutions that satisfy the constraints are *feasible* solutions and the feasible solution with the minimum cost function is the *optimal* solution, that is *cf* is optimal. In the context of iterative synthesis a feasible solution is one that when implemented will correctly perform the original design specification. In the MOODS system the transformations are complete therefore every solution is a feasible solution, however this is not true in the Devadas and Newton system [42] whose transforms may result in non-feasible solutions which must be determined using constraints that check the design's correctness. For example two instructions writing to the same register may be made to execute concurrently, this error violates a constraint which is enforced by making the cost function arbitrarily high. In the MOODS system the transformation would be filtered out at the testing stage.

The general form of an adaptive heuristic to find a feasible solution with a near optimal cost function is shown in Figure 5.3. The significant components used in this algorithm are described in the following paragraph.

```
General_Adaptive_Heuristic()
/* general form of an adaptive heuristic for
   combinatorial optimisation */
{
S = $S_0$;                      /* initial solution */
Initialise heuristic parameters;
do {
      do {
            NewS = perturb(S);
            if accept(NewS, S) S = NewS;
            } while !(time to adapt parameters);
      Adapt parameters;
      } while !(terminating criterion);
}
```

Figure 5.3 The general form of an adaptive heuristic for combinatorial optimisation.

The initial feasible solution, S_0, must first be generated, from which other feasible solutions are iteratively obtained, the current one being S. In synthesis any feasible solution can be obtained by manipulation of a directly compiled design [9]. The manipulation of the current solution is performed by the `perturb(S)` function which generates a new solution. The `accept(NewS,S)` function determines whether or not to accept the new solution and make it the current working solution. `accept(NewS,S)` is a function of the cost functions for both S and NewS and of the heuristic parameters. The *adapt parameters* section changes the heuristic parameters which may include the perturbation function, acceptance function, S or the criterion *time to adapt parameters*. The parameters are changed so that the solution converges on a near optimal solution. This modification of the parameters gives rise to the term *adaptive*, where changes may be made by the algorithm using some learning mechanism or by the user using his own learning mechanism.

Many algorithms may be obtained from the general adaptive heuristic, two of which are described here. The simulated annealing approach was selected as the primary algorithm as it has been used with varying success in numerous other applications [85,86]. Some problems have the "right character"[85] for simulated annealing and as with all adaptive heuristics the "proof of the pudding is in the eating"[81]; suggesting that the algorithm must be implemented in order to determine its suitability to solving a problem. Simulated annealing is an intriguing instance of artificial intelligence as the computer can arrive almost uninstructed at a good solution. In addition, the connection between natural phenomena and problem solving in simulated annealing can provide useful insights into optimisation which can be used to develop alternative algorithms such as the sequence heuristic described in Section 5.2.3.

5.2.2 Simulated Annealing

Simulated annealing [81,85,86,87] is best explained with the aid of a configuration space. The optimisation problem is to find some configuration of *n* parameters that minimises the cost function. The configuration space indicates the cost evaluated by the cost function for particular configurations. The configuration space for *n* parameters defines an *n*-dimensional surface. Figure 5.4 shows a configuration space for $n=1$. To change the solution a small random perturbation is made to the current configuration. If only good perturbations are accepted, as with iterative improvement, the final solution is likely to be a local minimum, as would occur in Figure 5.4. In simulated annealing bad or "uphill" configurations are probabilistically accepted based on a temperature parameter. For high temperatures the probability of acceptance is large whereas at low temperatures it is small. For a given temperature all configurations with a cost less than and a *band* of configurations with a cost greater than the current configuration will be

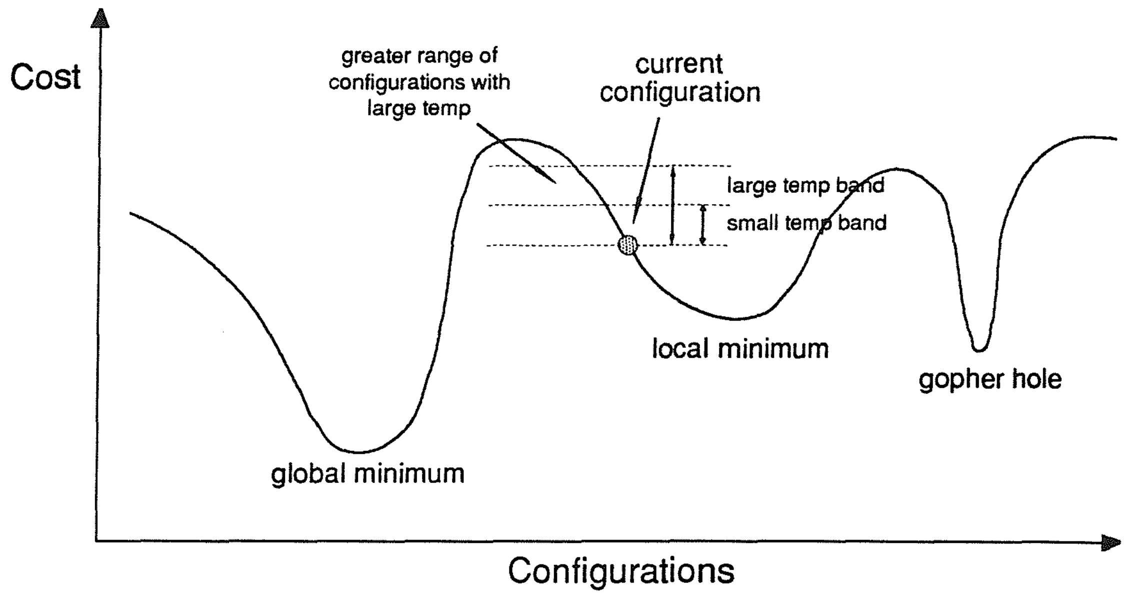


Figure 5.4 A one-dimensional configuration space.

accepted. The width of the *band* is dependent on the temperature, a high temperature gives a wide band and a low temperature a narrow band. As can be seen on the configuration space of Figure 5.4, for a small uphill perturbation a high temperature encloses a wider set of configurations defined by the band than does a low temperature. This means that for high temperatures less steps are required to exit minima therefore the configuration has a higher probability of moving between adjacent local minima. As the temperature is decreased the solution will settle in the deepest local minimum, the global minimum. The reason for this is that the probability of moving from a local minimum is controlled by the width of the band, therefore as the global minimum is deepest there is less chance of the solution moving from it. The success of simulated annealing depends on the landscape of the configuration surface. A mostly flat landscape, as in placement problems, will anneal well, however one with "gopher holes" (deep, steep-sided local minima) may be impossible to anneal as moves will result in falling into these holes which will be increasingly difficult to get out of as the temperature decreases, thus trapping the solution in a local minimum.

The difficulty of moving from local minima is also dependent upon the transformations applied to the design. This was found to be the case when a high priority was given to the area objective in early tests of the system, which did not include the `uncombine_single_unit()` transform. Units were merged into ALUs which invariably improved the area, however when an attempt was made to uncombine an entire ALU the degradation in area exceeded the band defined by the temperature. This prevented the acceptance of the transform and so a move out of the local minimum.

```
auto_optim(td)
/* Simulated annealing algorithm */
{
for (temp=Tstart; temp>=Tend; temp*=Tstep)
    for (iterations=0; iterations<Istep; iterations++) {
        auto_select_trans(td);
        if (test_trans(td)) {
            estimate_trans(td);
            E = evaluate(cost_fn);
            if (E <= 0 || rand() < exp(-E/temp))
                perform_trans(td);
            }
        }
}
```

Figure 5.5 The simulated annealing algorithm.

The simulated annealing algorithm, `auto_optim(td)`, is shown in Figure 5.5, where the initial solution and parameters are set outside the procedure. The perturbation function is implemented by the `auto_select_trans(td)` procedure which selects a transformation and its associated data to apply to the design using random selection with heuristic steering methods. The selection process consists of firstly determining the transformation type, either scheduling or allocation; an equal probability is given to each. Secondly, a node is selected on which to perform the transformation, yet to be determined. A data path node is selected for allocation type transformations and a control node for scheduling type transformations. Given the node and transform type, the transform itself and any additional data are selected by random selection and steering heuristics. For example, suppose that a scheduling type transform is to be selected and a general node (one input and one output arc) has been chosen. The heuristics used dictate that either an *ungrouping* or *sequential merge* transform must be selected, as the *parallel merge* and *merge fork and successor* transforms require the selected node to be a fork type. The *group on register* transform is considered by the selection routine to be an allocation transformation applied to a register data path unit. Out of the possible transformations one is randomly selected; suppose it is the *sequential merge* transform. A further steering heuristic may be used which dictates that the second node to be selected must be in the same sequential section as the first node and reachable from the first node. The steering heuristics used do not bias the selection of the transformation data as any other data used would result in failure during the testing step; therefore its only effect is to reduce the quantity of unsuitable selected data reaching the `test_trans(td)` procedure.

The effect of the transform is estimated by the `estimate_trans(td)` procedure and the change in cost function calculated by the `evaluate(cost_fn)` procedure described in Section 5.1.1. The value returned from this procedure is analogous to the change in energy, ΔE, in the Metropolis procedure [88]. If $\Delta E \leq 0$, the transform is accepted and performed. The case $\Delta E > 0$ is treated probabilistically: the probability that the transform is accepted is $P(\Delta E) = e^{-\Delta E/T}$. A random number uniformly distributed in the interval (0,1) is used to determine the probabilistic outcome. A random number is generated by procedure `rand()` and compared with $P(\Delta E)$. If it is less than $P(\Delta E)$, the transform is accepted and performed; if not a new transform is selected.

The simulated annealing process consists of raising the temperature of the design such that it "melts", that is, transforms that degrade the design are accepted almost as often as ones that improve it. The temperature is then lowered in slow stages until the design "freezes" and no further changes occur. At each temperature the simulation must continue long enough for a steady state to be reached. The sequence of temperatures and the number of transformations per temperature is the *annealing schedule.* Simulated annealing establishes gross features of the design at higher temperatures and fine details at lower temperatures.

The annealing schedule in the `auto_optim(td)` procedure is set by the variables *Tstart*, *Istep*, *Tstep* and *Tend* which set the starting temperature, the number of iterations per temperature step, the reduction between temperatures and the termination temperature. These parameters are currently set by the user where the start and end temperatures represent the boiling and freezing temperatures of the design. The value of *Tstep* should be fine enough to avoid "quenching" the design. Experiments to establish a procedure to determine the best annealing schedule for a design are shown in Section 6.1.2.

5.2.3 SEQUENCE HEURISTICS

Despite the success of simulated annealing it has a few major disadvantages; firstly, as demonstrated in Chapter 6, the algorithm parameters which describe the annealing schedule are difficult to obtain. It may be possible to automate the process such that the algorithm finds the schedule, however, this should only be done if an optimisation algorithm which produces better designs or executes in a shorter design time can not be found. Secondly, simulated annealing is slow compared to other optimisation methods. The reason for this is that bad perturbations are accepted on a random basis, therefore a design is degraded before it reaches a local minimum in order to find the global minimum. However, there are often many near optimal local minima which represent suitable near optimal design configurations. The design bounces between these near optimal minima eventually settling in one of them, which is in theory the deepest and therefore the global minimum.

The simulated annealing approach of occasionally accepting bad perturbations has a great deal of worth. There are, however, other ways to accomplish this, such as the sequence heuristic, which can avoid the disadvantages of simulated annealing. The use of sequence heuristics has been shown to be an improvement over simulated annealing [81,89], however the proof of the pudding is, once again, in the eating! The sequence heuristic algorithm is shown in Figure 5.6. The sequence heuristic accepts a bad perturbation only if a good perturbation has not been found over a sequence of attempts. The current length of the bad perturbation sequence is given by the variable `length`. The `accept(NewS,S)` function of the general adaptive heuristic shown in Figure 5.3 operates as follows: a new solution NewS with cf(NewS) >= cf(S), that is a positive ΔE, is accepted if and only if the last L perturbations on S failed to generate a solution with cf(NewS) < cf(S), that is a negative ΔE. If L perturbations have failed then the current bad perturbation is accepted and the length parameter updated. The *adapt parameters* phase also keeps track of the best solution found so far before applying the bad perturbation. The sequence length L can be updated by increasing it to $L\times\beta$ or $L+\beta$ as in this case. The process terminates when the sequence length reaches the maximum value, Lend.

```
seq_heuristic(td)
{
L = Lstart;                    /* initial sequence length */
length = 0;
do {
    do {
        auto_select_trans(td);
        if (test_trans(td)) {
            estimate_trans(td);
            E = evaluate(cost_fn);
            if (E <= 0) {
                perform_trans(td);
                length = 0;
                }
            else length++;
            }
        } while (length<=L);
    save design if best so far;
    length = 0;
    perform_trans(td);
    L+=β;
    } while (L<=Lend);
}
```

Figure 5.6 The sequence heuristic procedure.

Unlike simulated annealing the sequence heuristic approach does not rely on the artificial notion of temperature and has fewer parameters to adjust, therefore making it more elegant. The acceptance of degradations only when found to be in a local minimum, as opposed to the random acceptance of degradations as in simulated annealing, is likely to result in a faster algorithm as the degradations are applied in a controlled fashion. Before applying a degradation the current design is saved; after a set computation time the process ends and the best design saved is selected. This approach is in fact an elegant variation on the Monte Carlo method where a set of designs are generated using iterative improvement and the best one from the set is selected. The difference lies in the choice of the starting point for each Monte Carlo run: in the Monte Carlo approach the initial implementation is used, whereas in the sequence heuristic method a slightly degraded version of the last design is used. This ensures that the probability of finding a better design is greater. In addition the length of the sequence is increased each time a degradation takes place, thus gradually increasing the probability of finding a better design before applying a degradation.

5.2.4 Automated Design Space Exploration

As mentioned in Section 1.7 the exploration of the design space by an intelligent silicon compiler provides the user with an insightful characterization of design alternatives as well as trade-off curves [3]. The synthesis system designer can also use the design space to improve and develop new optimisation algorithms.

Design space exploration entails finding a number of near optimal designs, each a point in the design space. In the MOODS system, a two or three dimensional design space is characterized by creating a two or three objective cost function respectively. A number of designs are generated for cost functions consisting of each objective at a high priority while other objectives are at a low priority. The target value of the high priority objective is varied from 25% to 100% of the initial value in order to generate a range of designs. The initial design used to generate each design point may be either the initial design generated from the intermediate code or a previous design; where the latter constitutes *dynamic* design space exploration. A previous design point can be used as the transformations are reversible thereby allowing the degradation necessary to trade design aspects and move to a different design point. For each design point the automated design space exploration procedure creates an appropriate cost function, initialises the adaptive heuristic parameters and calls the optimisation algorithm. The resulting designs not only show the variation possible but also give a useful insight into the effectiveness of the optimisation algorithm.

The set of design points is analyzed and a subset extracted such that each point in the set is better in at least one criterion than all the others in the set, that is, the subset consists of only points on the optimal design "curve".

6 RESULTS

This chapter is divided into a number of sections each concerned with a particular aspect of behavioural synthesis using the MOODS synthesis system. The sections are as follows:

6.1 Determination of simulated annealing parameters in order to obtain correct annealing schedules and reliable results. This includes an investigation of: changes to the cost function in order to provide a stable start temperature for a given design, temperature reduction methods, a method to establish annealing schedules and the effect of the random number sequence on implementations.

6.2 Initial and optimised results for a selection of benchmark designs [90] using a comprehensive cell library.

6.3 Comparison of the MOODS system to other synthesis systems, where cell libraries and cost functions comparable to those used in the other systems are used.

6.4 The use of design space exploration to investigate the optimisation process and characterization of designs.

The MOODS synthesis system consists of approximately 22000 lines of 'C' and runs on a MicroVax 3100 workstation. On execution MOODS either creates an initial design implementation or restores a previously saved design. MOODS then displays the MOODS prompt, from which the user may examine the design, specify a cost function, initialise the algorithm parameters and start either optimisation or automatic design space exploration. After manipulating the design the user exits MOODS which causes bindings to take place and netlist files to be generated [91].

6.1 DETERMINATION OF THE SIMULATED ANNEALING PARAMETERS

The simulated annealing parameters comprise those that describe the annealing schedule, the temperature reduction function used to determine the next temperature step and the function used to calculate the energy change vector. As described in Section 5.2.2 the annealing schedule parameters are:

T_{start}	the initial temperature,
T_{end}	the final temperature,
I_{step}	the number of iterations to apply at each temperature step, and
T_{step}	the reduction made to the temperature in order to reach T_{end} from T_{start}.

6.1.1 Providing a Cost Function Independence for the Annealing Schedule

One of the problems associated with the simulated annealing algorithm is the determination of the annealing schedule. It would be desirable if the schedule was consistent for all designs and all cost functions. However due to the dependence of ΔE on the design through technology dependent data, an independence from this is not possible. A partial independence from the cost function would be desirable, therefore giving one annealing schedule for each design. The most fundamental annealing parameter is the start temperature T_{start}. The following section explains how the cost function was changed so as to provide an independence of T_{start} from the cost function.

The cost function consists of a prioritised list of criteria each with its associated goals (targets). The use of a prioritised cost vector ensures that criteria are optimised in the order specified. The effect a transformation has on the design is given by the first non-zero element of the energy change vector (see Section 5.1.1). The energy change vector is determined by averaging the change in energy of all criteria at each priority, thus resulting in a single value for each priority. How the change in energy is calculated will affect the annealing schedule between different designs and cost functions. The change in energy for a given criterion, c, is given by:

$$\Delta c = p_{est} - p_{pres} \qquad \textbf{(6.1)}$$

where p_{est} and p_{pres} are the positive deviations from the target for the estimated and present positions of the design in the design space respectively.

In order to show and thus attempt to reduce the variations in the annealing schedule with differing cost functions, four typical cost functions were selected for application to an average design. The Kalman filter benchmark [90] was chosen as it represented an average design used later in the results. The four typical cost functions selected were:

1. delay at priority 1 and area at priority 2 (d@p1, a@p2),
2. area at priority 1 and delay at priority 2 (a@p1, d@p2),
3. area and delay criteria at priority 1 (a@p1, d@p1), and
4. delay at priority 1 with a target of 4000ns and area at priority 2 (d@p1 to 4000, a@p2).

Targets other than those specified above are minimising to zero.

The start temperature of the annealing schedule should be relative to the freezing point of the design, therefore it is the freezing point which must be determined. This is given by the point where the degradations applied to the design start to reduce significantly. For each temperature step during the annealing schedule the cost of the degradations applied to the design was plotted on a temperature-cost graph. The cost of a transformation is given by ΔE (as defined in Section 5.1.1) where the cost of degradations is given by positive ΔE and cost of improvements by negative ΔE. For each temperature step the positive and negative ΔE are tallied and used in the temperature-cost graphs. Note that ΔE was scaled resulting in larger cost figures in order that temperature figures were not excessive. The cost of the degradations is used as it represents the actual change to the design. The number of degradation transforms applied to the design does not correspond to the actual change occurring to the design. This is due to the variation in ΔE depending on which transformation has been selected and on what part of the design it is to be applied. This is demonstrated in the graph of Figure 6.1b where both the number of degradation transformations (marked by triangles and scaled by $\times 40$) and the cost associated with them (marked by squares) have been plotted. This shows that transformations which have a large effect on the design are applied early in the optimisation process and those having a small effect, thus refining the design, are applied later; a characteristic of simulated annealing.

The temperature-cost graphs for the cost functions given above are shown in Figure 6.1; note that as the optimisation process proceeds the temperature is being reduced, that is, in the temperature-cost graphs the design advances from right to left. A linear temperature reduction function is used, that is, the temperature is reduced by a fixed amount each step ($T=T-T_{step}$), thereby giving an even distribution of points along the x-axis. The vertical lines represent the start of freezing which is taken to be 85% of the maximum of the smoothed curves (the solid lines). For the cost functions with delay included as a major objective, ie, at priority 1, the freezing point is in the range 125-

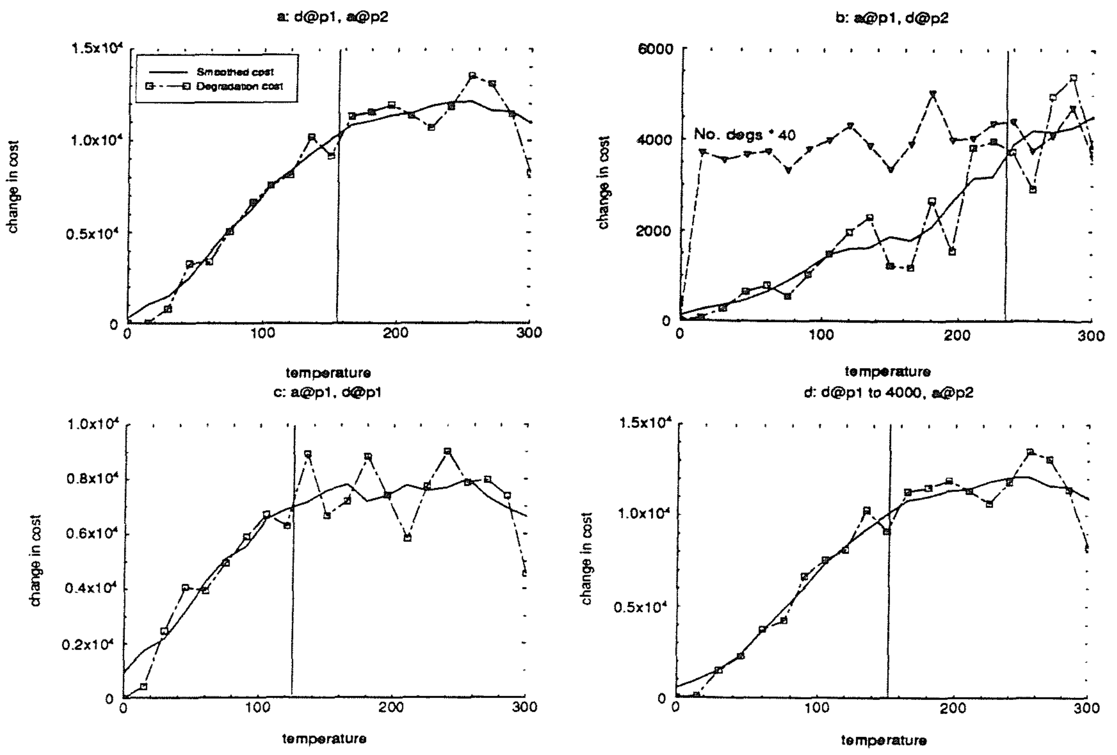


Figure 6.1 Variation of freezing point with cost function calculated using equation (6.1).

155, whereas with area as the only major objective the freezing point increases to 235. Delay is the dominant objective as its change in energy, Δc, is small compared to that of the area objective.

To make ΔE less vulnerable to variations in Δc for various criteria and thus less vulnerable to changes in the cost function, the change in energy for each criterion (Δc) was normalised with respect to its initial value, c_{init}. Therefore equation (6.1) becomes:

$$\Delta c = \frac{p_{est} - p_{pres}}{c_{init}} \qquad \textbf{(6.2)}$$

The above evaluations were repeated using equation (6.2) to determine ΔE. The resulting graphs are shown in Figure 6.2. The freezing point for the area objective at priority 1 has moved into the range of the other three freezing points. All four cost functions have freezing points in the range 125-160 which is considered acceptable.

For each of the benchmark designs used in the results the freezing point was determined using the above method by performing an initial optimisation using an estimated preliminary schedule. The freezing points were used to calculate the start temperatures, T_{start}, shown in the table of annealing schedules, Table 6.1 on page 124.

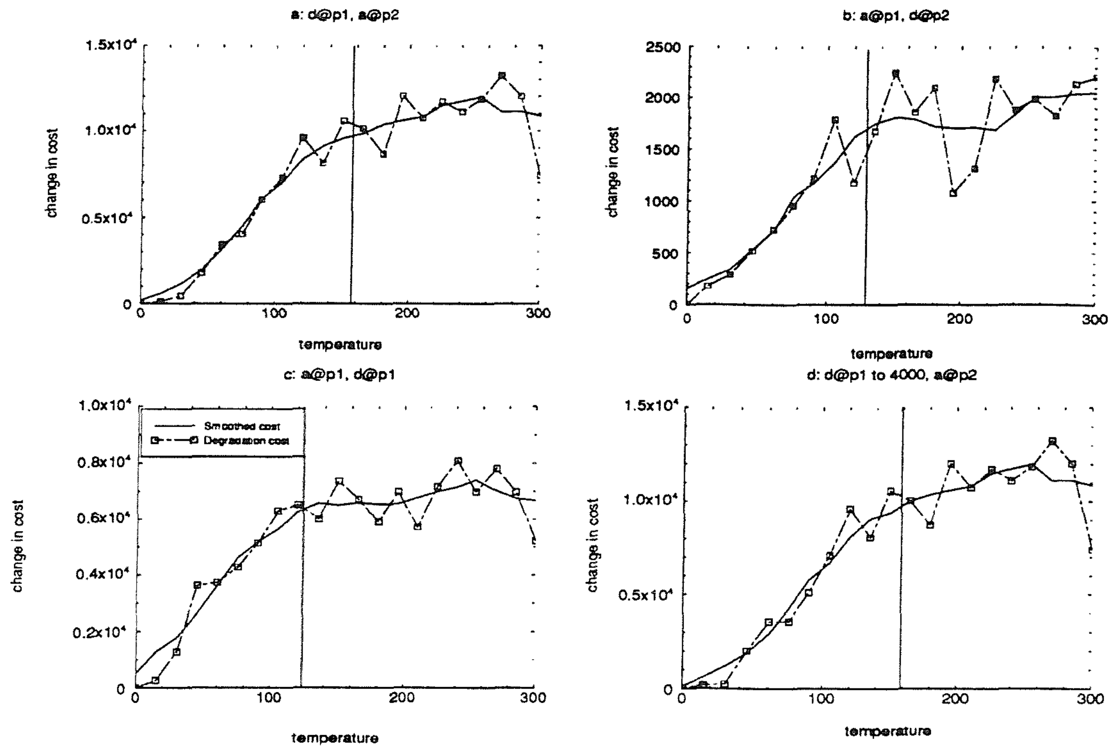


Figure 6.2 Variation of freezing point with cost function calculated using equation (6.2).

6.1.2 Finding the Annealing Schedule

The annealing parameter T_{start} is determined using the temperature-cost graph as described in Section 6.1.1. Throughout the experiments and results the end temperature T_{end} will be set to zero where only improving transforms would be applied to a design. The end temperature could be determined by monitoring the changes in the cost function. For example, in the Devadas and Newton system [42] the annealing process is terminated when the cost function has not changed for three consecutive temperature points. It is possible that this approach may lead to premature termination if the design becomes trapped in a local minimum which is difficult to get out of. It is unknown how long it would take to select the appropriate transformation in order to exit such a gopher hole, however it will vary with the complexity of the design as the probability of selecting a particular part of the design will decrease with increasing design complexity. Therefore a greater number of iterations will be required to exit a gopher hole for larger designs. The zero temperature terminating condition is therefore a safer option that introduces no additional parameters which would affect the determination of the annealing schedule and so the evaluation of the optimisation algorithm.

The other annealing parameters (T_{step} and I_{step}) are difficult to determine, however an estimate of the total number of iterations, I_{total}, given by equation (6.3), can be made by examining the cost of improvements measure.

$$I_{total} = (1 + \frac{T_{start} - T_{end}}{T_{step}}) * I_{step} \tag{6.3}$$

The set of temperature-cost graphs in Figure 6.3 show how the cost of improvements (the lower curve) and cost of degradations (the upper curve) vary with parameters T_{step}, I_{step} and I_{total} for the FRISC1 benchmark design. The values chosen for T_{start} and T_{end} were 200 and 0 respectively. T_{start} was chosen to include the freezing point which was previously found to be 100. The values for T_{step} were 2, 10 and 20 for the left, middle and right columns of graphs respectively and I_{step} was chosen such that the total number of iterations, I_{total}, applied to the design was approximately 10000, 15000 and 20000 for the top, middle and bottom rows of graphs respectively. The resulting costs shown were summed over a range of temperature steps equal to the largest step used thereby giving comparable graphs. The cost curves were then smoothed as shown by the solid curves.

In optimising a design to a particular cost function the design cost will be reduced by a particular value equal to the distance between the initial and optimised design points in the design space. The costs at temperature point T=0 are a good aid in determining whether sufficient iterations have been performed in order to optimise the design. If transformations were applied to the design at this point (only improvements are applied at T=0) then the design may not be optimal. On the other hand if sufficient iterations have been performed then few improvements will be possible at T=0. This is demonstrated in Figure 6.3 where in the graphs of the top row insufficient iterations have been performed thus the design was still being improved at T=0. As the total number of iterations is increased the bulk of the improvements to the design (shown by a bulge between the smoothed curves) occur at a higher temperature, with improvements at lower temperatures being applied only to counteract the effect of degradations. The value of I_{total} where the number of improvements applied at T=0 becomes a minimum is the minimum total number of iterations required to optimise the design. This value appeared to be independent of the size of T_{step}, however the quality of the design was not. Using this method the minimum value of I_{total} was determined for each benchmark and entered into the table of annealing schedules, Table 6.1 on page 124. Given I_{total} and a value for T_{step}, I_{step} was calculated using equation (6.3).

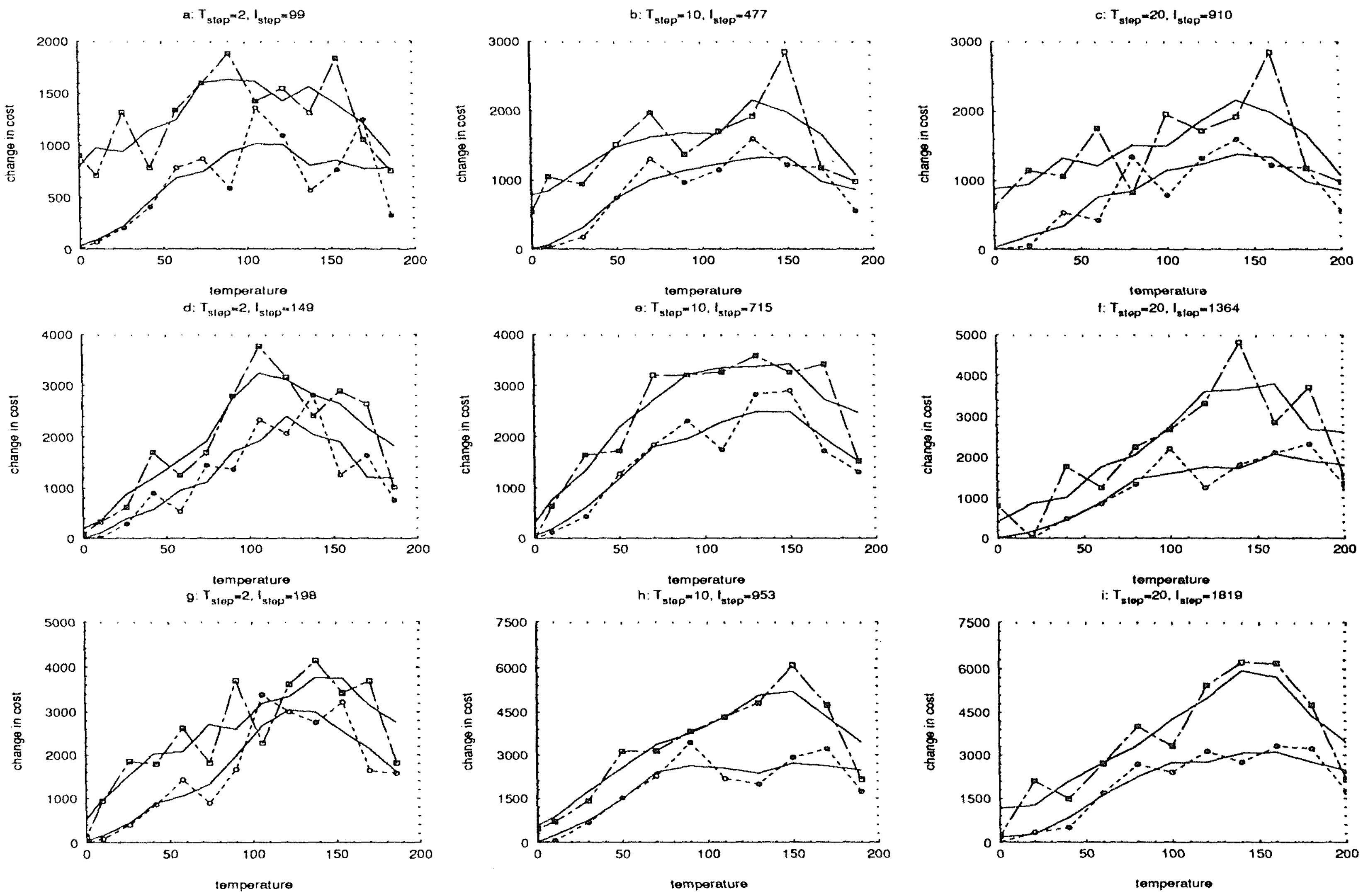


Figure 6.3 Variation in cost curves with different step and iteration values.

Total number of iterations are 10000, 15000 and 20000 for the top, middle and bottom rows respectively.

6.1.3 Investigation of the Temperature Reduction Function

According to simulated annealing theory better designs will be produced by using fine temperature steps as quenching would be avoided. The set of designs produced from the schedules used in Figure 6.3 produced the opposite effect, that is a coarse value of T_{step} resulted in better designs. To verify this and to show that a better design is achieved by using more iterations, the annealing schedules of graphs b, d, f and h of Figure 6.3 were each used to extensively explore an area-time design space using the automatic design space characterization ability of MOODS. The set of designs which characterized each design space were limited to 25 design points and the resulting design spaces are shown in Figure 6.4.

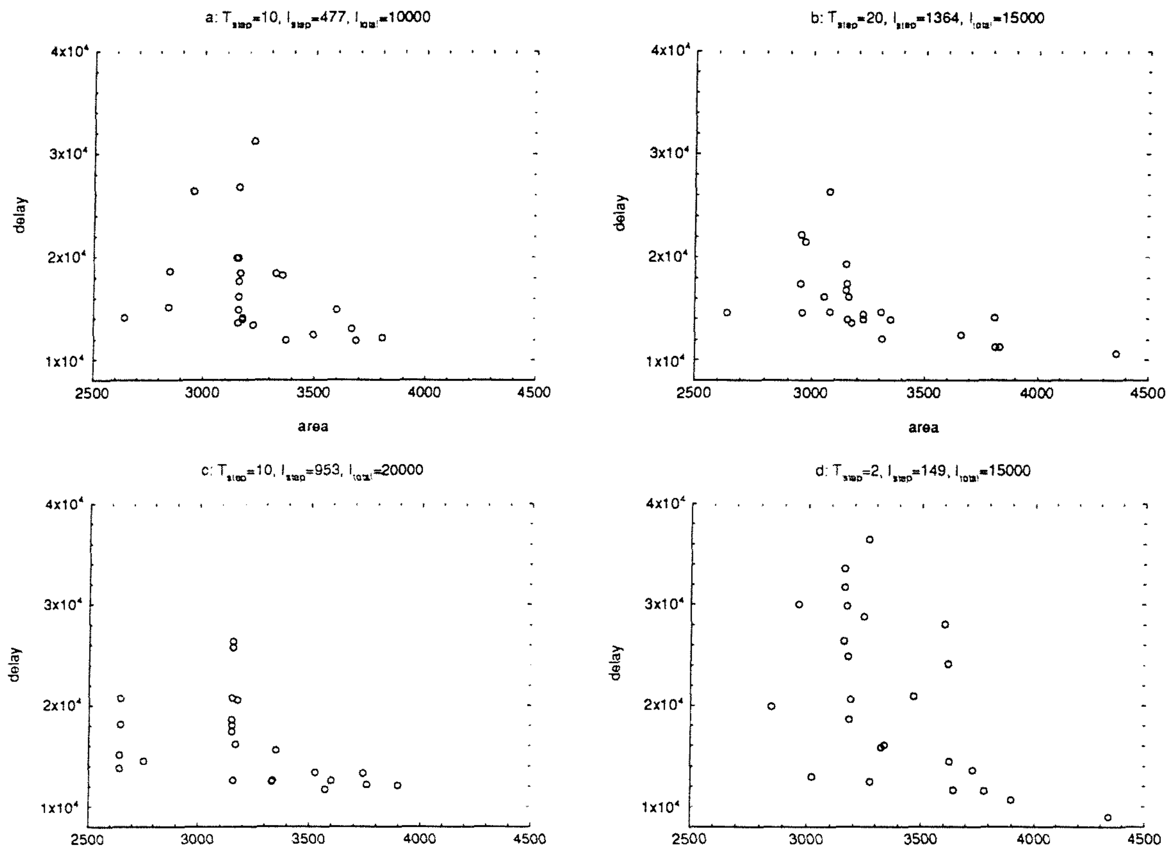


Figure 6.4 AT design spaces for FRISC1 using various annealing schedules with linear temperature reduction.

The design space of Figure 6.4a compared to that of Figure 6.4c show that better designs result when more iterations are applied. Also, the design space of Figure 6.4d compared to Figure 6.4b clearly shows that worse designs have been found when a finer temperature step has been used. The reason for this is that as optimisation progresses it becomes considerably easier to degrade the design than improve it. This suggests that the improvement of using the coarser temperature step is due to the greater number of iterations being applied at T=0. Thus for large T_{step}, I_{step} is also large giving more opportunity at T=0 to compensate for previous degradations. Conversely, when T_{step} is small, I_{step} is also small therefore allowing less opportunity for compensation and resulting in a worse design space. The finer T_{step} in Figure 6.4d allows for a few degradations at the extra $T \neq 0$ steps (steps t=2, 4, 6 and 8) and the reduced value of I_{step} means that insufficient iterations are applied at T=0 to compensate for them. The implication of this is that more iterations require to be performed at the low temperature points and so the linear temperature reduction method used to obtain the temperature-cost graphs is inadequate for optimisation.

To increase the number of iterations applied at low temperatures either the number of iterations per temperature step can be increased with decreasing temperature or more low temperature steps created by reducing the temperature in a non-linear fashion. The latter method is in line with traditional simulated annealing approaches and involves proportionally reducing the temperature using the reduction function:

$$T_{new} = T_{present} \times p \tag{6.4}$$

where p is the proportional counterpart of T_{step}. The proportional temperature reduction function of equation (6.4) was used to generate the design spaces of Figure 6.5. The graphs are at the same scale as those in Figure 6.4 and use equivalent schedules. The value of p was chosen so that the number of temperature steps between the start and end temperatures were the same. T_{start} and T_{end} were 200 and 0 respectively, thus making design spaces a through to d of Figure 6.4 comparable with design spaces a through to d of Figure 6.5.

By comparing the AT design spaces of Figure 6.4 with those of Figure 6.5 it can be seen that the function used to determine the next temperature step plays a significant role in the quality of the resulting design spaces. In all four design spaces the proportional temperature reduction has resulted in a better clustering of design points therefore it was made the default reduction method in the MOODS system. The linear reduction method can be optionally selected when producing temperature-cost graphs. An improvement can still be seen for an increase in the value of I_{total} between

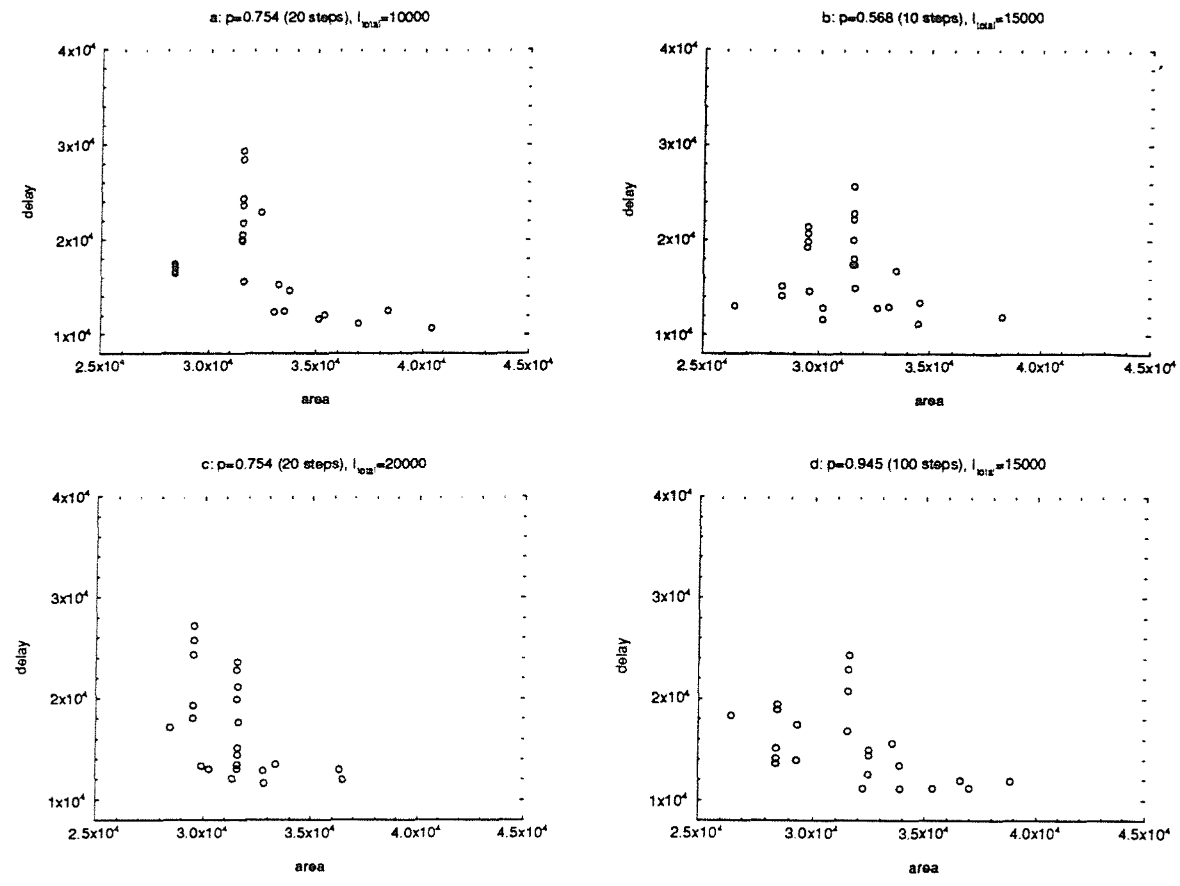


Figure 6.5 AT design spaces for FRISC1 design using the same schedules as in Figure 6.4 but with a proportional temperature reduction.

Figure 6.5a and Figure 6.5c; however no reduction in quality can be seen in Figure 6.5d when compared to Figure 6.5b as previously observed in Figure 6.4. As there is little difference between Figure 6.5b and Figure 6.5d it would seem that the value of I_{step} is less significant. This is illustrated in Figure 6.6 where quenching schedules have been used. As expected the linear temperature reduction produced a good design space, Figure 6.6a, due to the increased iterations at T=0 and the proportional temperature reduction produced a slightly worse design space, Figure 6.6b, due to quenching. As the design spaces are still quite good it not only implies I_{step} to be less significant but also that the temperature reduction function could be improved by applying more iterations at temperatures in the region of T=0.

The results produced by the experiments relating to linear temperature reduction show that iterations applied at T=0 are an important fine refinement process in the optimisation of a design. For this reason iterations are made to occur at T=0 even though start and step values may not have resulted in T=0 being achieved.

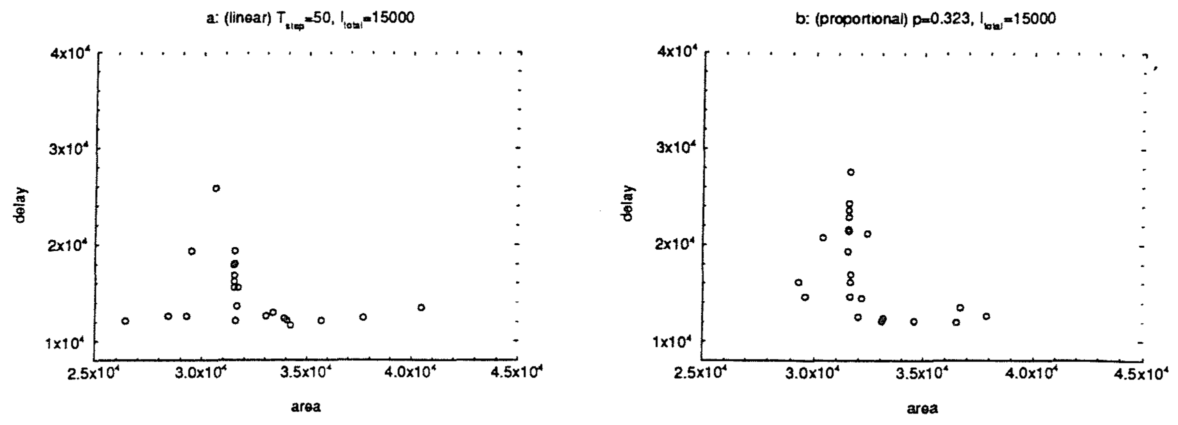


Figure 6.6 AT design spaces for FRISC1 using a quenching schedule for linear and proportional temperature reductions.

6.1.4 Investigation of the Effect of the Random Number Sequence on Implementations

The simulated annealing algorithm is a stochastic process where the selection of transformations and their subsequent application are dependent on random numbers. Consequently the question arises: "Does the random number sequence influence the resulting implementations?". In order to determine this a typical design was optimised using different seeds in the random number generator to produce varied random number sequences. Figure 6.7 shows the design points for 20 area optimised and 20 delay

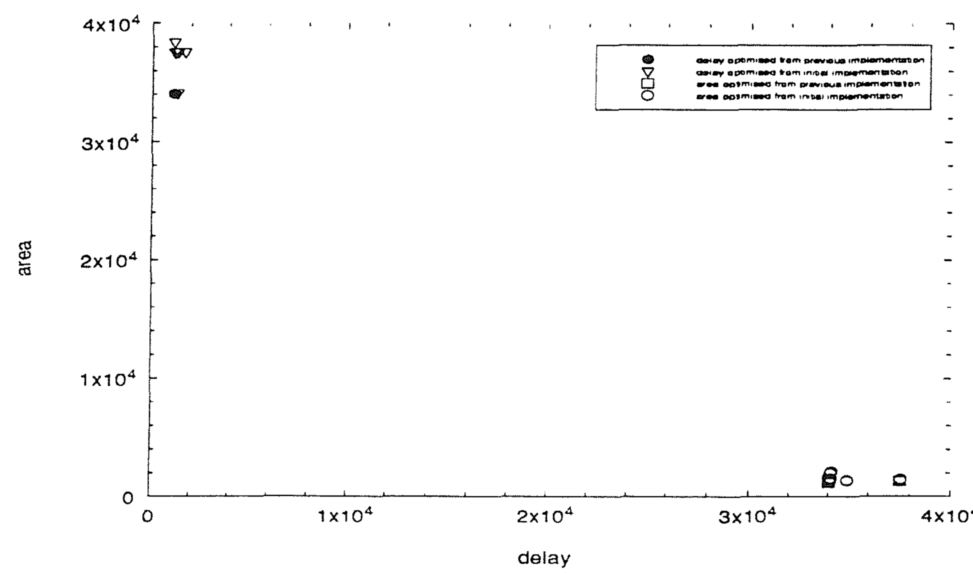


Figure 6.7 Variation in implementations due to arbitrary random number seeds for both area and delay optimisation objectives.

optimised implementations. Half of the designs (10 each of the area and delay optimised implementations) were generated using the initial implementation and the other half using previous implementations.

The close grouping of design points for both the area and delay optimised implementations show that the random number seed and its resulting sequence has a minimal influence on the ability of the optimisation algorithm to consistently reach a point in the design space. Implementations generated using previous rather than initial implementations have resulted in slightly improved design points due to their already optimised starting points. There is a high, but not 100% probability that an achievable design point can be consistently reached. Therefore in the reporting of results it is fair to select the best result, that is one that best achieves the original objectives, from an exploration of the design space. The results of implementations of benchmark designs given in Sections 6.2 and 6.3 have been selected from explorations of the design space. The best implementations have been reported due to the high probability of obtaining those results although this can not be guaranteed due to the stochastic nature of the simulated annealing algorithm.

6.2 BENCHMARK RESULTS

The results given in this and subsequent chapters use the annealing schedules shown in Table 6.1 as determined using the methods described in Section 6.1.2. T_{end} is always set to zero and the number of temperature steps was chosen to be 50 to avoid quenching. Table 6.1 gives the correct values of T_{step}, the linear reduction quantity and *p*, the proportional reduction factor, for the 50 temperature steps which includes the additional step, T=0. The schedules derived represent the minimum schedule required to achieve reasonable results, the number of iterations may be increased to improve the design points, however the law of diminishing returns applies. There seems to be little correlation between design size and any part of the annealing schedule. The schedule is probably more dependent on the complexity of the design and the number and type of operations used rather than the design size. Some correlation would be expected due to the relationship between design size and complexity, however this may be only apparent for very large designs.

	No. of ICODE lines	Start temp T_{start}	Total No. iterations I_{total}	Temp step for 50 steps with linear or proportional temp reduction.		No. of iterations per temp step. $I_{step} = I_{total} / 50$
				T_{step} for LTR	*p* for PTR	
CBM2	47	66	13000	1.35	0.910	260
CHIP	474	44	50000	0.90	0.919	1000
FRISC1	117	100	12000	2.04	0.903	240
FRISC2	181	121	33600	2.47	0.899	672
KALMANIO	83	176	36300	3.59	0.893	726
KALMANIO2	83	137	42900	2.80	0.897	858
ELLIP	55	110	33000	2.24	0.903	660
MC6845	91	154	28500	3.14	0.896	570
TAXI	37	143	7000	2.92	0.897	140
PARKER	30	210	18200	4.29	0.889	364
TSENG	47	220	15400	4.49	0.888	308
WINOGRAD	92 (46)†	82	20800	1.67	0.906	416

† Half of the instructions are register transfers caused by the ELLA to ICODE translator.

Table 6.1 Annealing schedules for the benchmarks.

The design data for the initial un-optimised implementation of each benchmark is shown in Table 6.2. All of the subsequent implementations reported in this and the following sections are derived from the same initial implementations. Using the annealing schedules in Table 6.1 the benchmark designs were optimised and the results derived as described in Section 6.1.4. The design data for the smallest (area optimised) and fastest (delay optimised) implementations of each benchmark are reported in Table 6.3 and Table 6.4. In generating the optimised results of Table 6.3 and Table 6.4, a comprehensive cell library was used. This consisted of the set of basic cells and included ALUs implementing plus and minus operations (+, -), shift operations (SR, SL) and comparison operations (=, <, >, !=). Alternative cells were also included in the library which implemented most functional type units including the (+, -) ALU. The alternative cells implement functions either faster and larger or slower and smaller than their basic counterparts. In the results tables the cells chosen to implement functional units are depicted by *s* or *f* indicating whether small or fast cells have been selected.

	Area (μm²)	Delay (ns)	Maximum control node delay (ns)	Number of control nodes	Critical path length	MOG usage measures (%)			Number of nets	Number of registers counters, rams (words), (total bit width)	Number of MUX units (bits): inputs	Functional Units	
						clock	regs	units				Total units (bits)	Number of units and type
CBM2	30697	1168.7	40.3	45	29	38	20	2	137	18r, (169)	8 (79) 26	47 (250)	1 neg, 32 =, 3 RS, 8 LS, 3 +
CHIP	1640204	8651.8	47.8	457	181	5	21	1	748	235r, 14c, 3a (3k), (2569)	21 (302) 114	194 (2622)	4 not, 13 <=, 42 =, 10 !=, 30 +, 30 -, 22 or, 1 RS, 6 LS, 19 and
FRISC1	43867	2866.5	44.1	114	65	4	30	11	121	15r, (165)	12 (177) 44	31 (244)	3 +, 1 not, 20 =, 2 !=, 3 RS, 1 -, 1 and
FRISC2	82257	3132.0	58.0	155	54	6	35	0	138	15r, (165)	8 (128) 45	41 (434)	1 not, 18 =, 2 !=, 3 RS, 12 +, 4 -, 1 and
KALMANIO	297088	5397.6	69.2	73	78	20	20	1	101	19r, 10c, 6a (673), (399)	10 (105) 28	21 (200)	1 <=, 10 =, 5 *, 5 +
KALMANIO2	297128	5397.6	69.2	81	78	20	9	0	101	19r, 10c, 6a (673), (399)	10 (105) 28	21 (200)	1 <=, 10 =, 5 *, 5 +
ELLIP	149285	3051.0	67.8	53	45	8	22	1	125	44r, (689)	7 (122) 14	34 (544)	8 *, 26 +
MC6845	32254	2448.5	41.5	90	59	3	29	0	121	42r, 4c, (213)	5 (60) 11	36 (187)	29 =, 6 +, 1 and
TAXI	8506	1136.2	43.7	38	26	18	19	0	42	13r, 1c, (31)	1 (8) 4	13 (37)	7 =, 3 and, 3 +
PARKER	35453	738.0	41.0	29	18	21	8	2	70	14r, (70)	4 (32) 12	22 (176)	9 -, 7 +, 6 !=
TSENG	23386	6975.0	225.0	46	31	1	46	0	55	16r, (121)	12 (96) 24	8 (64)	1 div, 1 -, 1 *, 3 +, 1 or, 1 and
WINOGRAD	167588	10983.6	67.8	172	162	29	7	0	224	132r, (1520)	0 (0) 0	46 (460)	12 *, 20 +, 14 -

Table 6.2 Initial implementation data for the benchmarks.

	Cost function priority		Area (μm^2)	Delay (ns)	Maximum control node delay	Number of control nodes	Critical path length	Maximum chain length	MOG usage measures (%)			No. of nets	Number of registers counters, rams (words) (total bit width)	Number of MUX units (bits): inputs	Functional Units units same as initial design for both implementations not shown	
	delay	area							clock	regs	units				Total units (bits)	No. of units and type ALUs in brackets. f=fast, s=small
CBM2	1	2	20149	518.7	39.9	13	13	3	48	56	6	119	6r, (61)	6 (61) 27	44 (203)	1 +f, 6 SLf, 3 SRf, 1 (shift)
	2	1	17142	604.5	40.3	15	15	3	44	80	6	108	5r, (52)	4 (43) 22	43 (197)	1 +f, 1 SLf, 7 SLs, 1 SRf
CHIP†	1	2	1546876	10775.8	125.3	180	86	6	6	25	2	681	160r, 4c, 3a (3k), (1642)	40 (646) 160	180 (2509)	9 (+,-)s, 7 (+,-)f, 17 -s, 2 >, 10<, 40 =, 8 !=, 10 <=, 1 SLf, 5 SLs, 1 SRf, 19 &, 12 comp, 13 +s
	2	1	1529843	12682.5	169.1	172	75	8	6	27	2	675	152r, 4c, 3a (3k), (1381)	40 (646) 162	180 (2465)	7 (+,-)s, 10 (+,-)f, 20 -s, 3 >, 12<, 38 =, 8 !=, 8 <=, 1 SIf, 5 SLs, 1 SRf, 18 &, 13 comp, 10 +,
FRISC1	1	2	33334	1209.0	52.7	42	26	4	12	36	44	111	13r, (133)	11 (161) 41	29 (212)	1 (+,-)s, 1 (+,-)f, 2 SRf, 1SRs
	2	1	31498	1475.6	46.5	42	28	4	12	37	44	111	13r, (133)	13 (193) 45	27 (180)	1 (+,-)s, 1 +s, 1 SRf
FRISC2	1	2	59818	735.0	52.5	14	14	3	53	40	2	123	13r, (133)	11 (176) 51	32 (290)	3 +f, 4 -f, 3 SRf
	2	1	26663	1440.6	68.6	21	21	2	49	52	2	102	13r, (133)	12 (192) 46	24 (162)	1 (+,-)s, 1SRf
KALMANIO	1	2	286671	2937.6	81.6	37	36	3	27	45	2	97	14r, 2c, 6a (673), (250)	11 (184) 34	21 (200)	1 (+,-)f, 4 +s, 2 *f, 3 *s, 1 comp
	2	1	269778	3747.4	93.0	42	41	3	23	37	2	86	16r, 2c, 6a (673), (248)	12 (200) 38	14 (88)	2 +s, 1 *s, 1<=
KALMANIO2	1	2	284529	2101.8	67.8	31	31	2	40	48	0	99	14r, 6a (673), (220)	13 (216) 40	20 (184)	5 +s, 4 *f
	2	1	268062	3264.0	81.6	40	40	2	34	19	0	87	12r, 6a (673), (214)	13 (232) 42	13 (85)	1 (+,-)s, 1 *s

† Design information from limited design space exploration due to system error.

Table 6.3 Synthesis results of benchmark designs using the comprehensive cell library.

	Cost function priority		Area (μm^2)	Delay (ns)	Maximum control node delay	Number of control nodes	Critical path length	Maximum chain length	MOG usage measures (%)			No. of nets	Number of registers counters, rams (words) (total bit width)	Number of MUX units (bits): inputs	Functional Units units same as initial design for both implementations not shown	
	delay	area							clock	regs	units				Total units (bits)	No. of units and type ALUs in brackets. f=fast, s=small
ELLIP	1	2	135638	531.2	142.1	4	4	7	92	40	18	118	20r (305)	22 (352) 53	23 (368)	3 (+,-)f, 11 +f, 3 +s, 4 *f, 2 *s
	2	1	38282	4302.0	119.5	36	36	2	43	45	15	87	20r (305)	16 (256) 62	3 (48)	1 (+,-)s, 2 *s
MC6845[†]	1	2	28176	193.2	55.5	4	4	6	76	15	2	115	35r, 1c, (151)	7 (70) 15	35 (195)	4 (+,-)f, 2 +s, 3 comp, 25 =
	2	1	23230	316.8	105.6	3	3	13	49	9	3	109	33r, 1c, (139)	6 (67) 12	35 (190)	1 (+,-)s, 4 +s, 29 =
TAXI	1	2	6022	219.6	37.0	6	6	5	69	33	1	37	8r (25)	5 (18) 11	10 (20)	1 +f, 6 =
	2	1	4262	360.0	60.0	6	6	5	54	33	1	35	8r (25)	3 (10) 7	11 (21)	1 +s, 7 =
PARKER	1	2	22689	317.7	105.9	3	3	5	92	4	20	58	8r (22)	7 (56) 20	15 (120)	3 +s, 7 -f, 5 !=
	2	1	8627	541.1	77.3	7	7	3	71	27	16	53	9r (30)	6 (48) 28	8 (64)	1+s, 1 -f, 6 !=
TSENG	1	2	22166	1406.4	234.4	6	6	6	31	33	4	55	16r (121)	12 (96) 24	8 (64)	3 +s, 1 -s
	2	1	17980	1707.2	213.4	8	8	5	29	48	5	56	16r (121)	14 (112) 31	5 (40)	1 (+,-)s
WINOGRAD	1	2	133300	1286.6	91.9	14	14	5	54	13	7	171	66r (752)	11 (132) 22	43 (430)	8 (+,-)f, 2 (+,-)s, 3 +f, 10 +s, 8 -f, 7 *f, 4 *s
	2	1	57812	14698.5	119.5	123	123	3	24	41	5	184	43r (512)	23 (322) 133	7 (70)	1 (+,-)s, 6 *s

† Design information obtained by single optimisation runs.

Table 6.4 Continuation of Table 6.3. Synthesis results of benchmark designs using the comprehensive cell library.

In all implementations the area and delay of the controller, as described by control cells in the cell library, was made small compared with that of functional units therefore results will show the typically accepted controller versus data path trade-offs. It should be noted that MOODS will adapt to different trade-off curves which are dependent on and thus defined by the cell library data. As described in Chapter 2 this is a capability possessed by very few synthesis systems.

The area and delay optimised implementations reported in Table 6.3 and Table 6.4 show a trade-off between area and delay. The least trade-off occurred in the CHIP and KALMANIO designs both of which contain RAMs. The RAMs used are limited to single port memories and can not be bypassed as with registers; these restrictions limit merging and sharing optimisations and thus limit the actual achievable design region. In general more variation occurs where ROM/RAM is not used and functional units implement either similar operations or operations common to individual cells both of which can be easily merged.

The delay of an implementation is the product of the critical path length and maximum control node delay. In many cases the critical path length is short and maximum node delay (the clock period) is long for delay optimised designs. A long clock period does not necessarily produce a slow design, as demonstrated by the CHIP and MC6845 designs where the faster implementations have a longer critical path length than their area optimised designs. Therefore it is insufficient when reporting data for delay optimised designs to give either critical path length or node delay; both figures are required in order to compare the real speed of implementations.

In comparing register numbers the total bit width (the sum of storage bit widths) is proportional to the area occupied and so is a more representative figure for area than the number of units. The number of registers in both the area and delay optimised implementations have been reduced by similar amounts, however, the mechanism used to optimise registers is different. Registers are required to store data between control states, therefore for a given design a greater number of states requires more registers to store data used by instructions spread among them. Fewer control states require fewer registers which are optimised away using the bypassing mechanism described in Section 4.2.1. When there are many control states and registers, the registers are optimised using the sharing mechanism performed by the register sharing transformation. It may be thought that due to the similar reduction in registers caused by bypassing and sharing, register optimisation can be performed as a separate synthesis task; however this is not the case as the optimisation opportunities for registers are highly dependent on the scheduling of instructions.

The number of multiplexers used in an implementation has increased in the optimised implementations due to the increase in unit sharing shown by the total number of functional units. In comparing the area of units used in a design the total number of bits is a good representation of the area used. For multiplexers the number of inputs is also an important figure. The units which make up each design have been selected in the implementations such that a greater proportion of fast cells are used in the delay optimised design than in the area optimised design.

An attempt has been made to determine how good a design is without reference to the user's objectives. The measure of goodness (MOG) measures give a guide as to how well the clock period, registers and units are utilised and are similar to other utilization measures [28,70]. For the clock period this consists of analyzing the slack time in each control state. The slack time is given by the difference between the end time of the maximum instruction graph and the clock period. The register and unit usages are determined by calculating the ratio of the number of control steps during which they are in use, to the critical path length. The unit usages are scaled by the probability of execution given by the conditional branch probabilities to allow for the possibility of mutually exclusive instructions sharing functional units. As a consequence of this the unit usage figures are small. In all designs the clock period is better utilised in the delay optimised implementations as would be required in achieving a fast design; similarly the unit usage should reflect the optimisation of the area, however this is not apparent due to the small unit usage figures. In all except the Kalman filter design the register usage is better for the area optimised implementations. This is not due to register sharing as indicated by similar register figures for both implementations but due to the increased critical path length which would increase the register active times compared to their inactive times.

To determine the effect of inline expansion two design descriptions which use modules, FRISC1 and KALMANIO, were transformed such that the modules were expanded inline resulting in the FRISC2 and KALMANIO2 descriptions. The modules in the

FRISC1 description are called more than once which causes an increase in the number of instructions in the expanded description; as a consequence an increase occurs in the area and delay of the initial implementation (see Table 6.2). Alternatively, the KALMANIO description only calls its modules once therefore in the expanded description no increase in the number of instructions occurs and there is little increase in the area and delay of the initial implementation. Despite the initial increases in area and delay the optimised implementations for the expanded designs are better than those for the un-expanded designs. The results shown in Table 6.3 demonstrate that inline expansion produces better implementations and where modules are called more than once, as in FRISC, a wider range of implementations occur. Better implementations will always result from inline expansion as optimisation restrictions, caused by module boundaries, are removed. In the area optimised FRISC2 design the user defined ALU module in FRISC1 which was expanded in the FRISC2 has been recreated by the system as it reduces the area most, this is shown by the single (+,-) ALU in the functional units column. The effect of inline expansion is illustrated further using design space exploration in Section 6.4.3.

6.3 COMPARISON OF SYNTHESIS SYSTEMS

In order to compare the designs produced by different synthesis systems sufficient data must be available which correctly describes the implementations. As mentioned earlier it is not enough just to count units and the critical path length when the objective is reducing area and/or delay. The results of popular benchmarks produced by other systems have been extracted from relevant papers with the exception of Scholyzer which was available for actual use.

When comparing with another system the MOODS cost function was set to the equivalent optimisation criteria of the other system. The cell library used by MOODS was constructed such that it reflected the cells and units available to the system being compared. Where the other system used ALUs, as in the TSENG example, a similar ALU was added to the MOODS cell library and given competitive parameters so that there was a good chance of it being used; of course this can not be guaranteed due to the stochastic nature of the MOODS system.

6.3.1 Comparing MOODS with SCHOLYZER

A detailed comparison of MOODS can be made with the Scholyzer system as it uses the same intermediate code (ICODE), generates a similar initial implementation, creates the output as a netlist of parameterized cells and all of the implementation data is available. The MOODS cost function was set such that delay was the high priority objective with area as the low priority objective, which is similar to the built in optimisation criteria of Scholyzer. The MOODS cell library contained only basic cells as assumed available by Scholyzer. The cell library data was used to calculate, using the same methods, the actual area and delay of the implementations produced by both systems; therefore the results shown in Table 6.5 are directly comparable. Most of the implementations produced by MOODS are of a similar speed or faster than the Scholyzer implementations. This is primarily due to the lack of binding by MOODS of user defined variables in the input description.

Both systems were run on a MicroVax 3100 workstation. Scholyzer's execution times varied from 9s for the TAXI design to 150s for the FRISC1 design; the execution time being highly correlated to the design size. MOODS execution times varied from 30s for the TAXI design to 100mins for the MC6845 design and are highly dependent on the annealing schedule. Although in general the MOODS execution times are an order of magnitude greater than those of Scholyzer the variety and quality of implementations achieved is considered more important. The MOODS execution times are also degraded by the fact that the current version of MOODS was compiled using no compiler optimisations.

Further improvements of the MOODS system over the Scholyzer system can be seen by examining the implementations produced from a simple small example (TEST). The cell library used was the comprehensive one as described in Section 6.2. The behavioural description shown in Figure 6.8 was compiled using the SCHOLAR language compiler. The resulting ICODE was then used to generate circuits from both the MOODS synthesis system and the Scholyzer system. A short behavioural description was used so that the resulting implementations could be shown graphically. Despite its shortness the implementations produced illustrate the major differences between the MOODS synthesis system and the Scholyzer system.

	Synthesis system	Area (μm^2)	Delay (ns)	Maximum control node delay (ns)	Number of control nodes	Critical path length /short	Maximum chain length	Number of nets	Number of registers, counters, rams (words) (total bit width)	Number of MUX units (bits): inputs	Functional Units (basic cells)	
											Total units (bits)	Number of units and type
CBM2	SCHOLYZER	28782	384.0	38.4	10	10	1	82	18r, (169)	8 (79) 26	15 (134)	3 +, 8 LS, 3 RS, 1 neg
	MOODS	16094	705.6	50.4	14	14	3	101	5r, (52)	5 (52) 18	40 (167)	2 +, 3 LS, 3 RS, 1 neg, 32 =
FRISC1	SCHOLYZER	36949	1050.4	40.4	43	26 / 9	2	146	11r, 1c, (147)	12 (177) 40	18 (153)	1 +, 1 -, 10 &, 2 !=, 3 RS, 1 not
	MOODS	38187	1035.0	41.4	39	25	4	114	13r, (133)	13 (193) 44	29 (212)	2 +, 1 -, 1 &, 2 !=, 2 RS, 1 not, 20 =
KALMANIO	SCHOLYZER	286577	2460.5	66.5	37	39	3	148	9r, 9c, 6a (673), (187)	9 (89) 26	17 (143)	4 +, 5 *, 7 &, 1 <=
	MOODS	272840	2423.5	65.5	38	37	2	88	17r, 2c, 6a (673), (276)	13 (216) 38	14 (88)	2 +, 1 *, 10 =, 1 <=
ELLIP	SCHOLYZER	150039	1130.5	66.5	17	17/3	1	175	45r, (705)	7 (112) 14	34 (544)	26 +, 8 *
	MOODS	87057	618.0	224.8	3	3	7	121	16r, (241)	30 (480) 72	17 (272)	13 +, 4 *
MC6845	SCHOLYZER	22696	243.6	40.6	6	6/2	2	86	20r, 6c, (167)	4 (56) 8	13 (88)	2 +, 10 =, 1 &
	MOODS	24112	226.0	50.9	4	4	8	108	31r, 4c, (143)	7 (82) 16	32 (155)	2 +, 29 =, 1 &
TAXI	SCHOLYZER	4039	284.2	40.6	7	7	2	31	3r, 1c, (21)	1 (8) 3	2 (12)	1 +, 1 =
	MOODS	4106	285.6	56.9	6	6	5	35	7r, (21)	3 (10) 7	11 (24)	1 +, 7 =, 3 &
PARKER	SCHOLYZER	30472	307.2	38.4	10	8/3	1	77	8r, (64)	11 (88) 23	17 (136)	5 +, 6 -, 6 !=
	MOODS	22689	317.7	105.9	3	3	5	58	8r, (22)	7 (56) 20	15 (120)	3 +, 7 -, 5 !=
TSENG	SCHOLYZER	23326	2237.0	223.7	10	10/3	1	87	16r, (121)	12 (96) 24	8 (64)	3 +, 1 -, 1 *, 1 div, 1 or, 1 &
	MOODS	23186	1446.0	241.0	6	6	5	55	16r, (121)	12 (96) 24	8 (64)	3 +, 1 -, 1 *, 1 div, 1 or, 1 &

Table 6.5 Comparison of benchmarks synthesized by Scholyzer and MOODS.
A basic cell library and delay optimising cost function has been used in MOODS for comparison with Scholyzer.

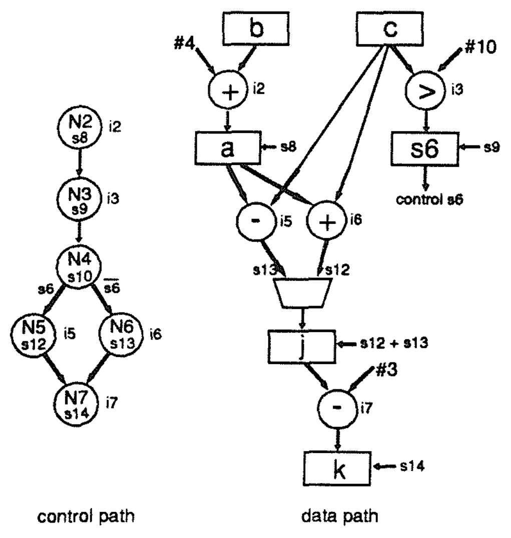
```
program test(b,c/k)          1   PROGRAM "test" 2 3 / 1 ACT 2
$(    define $(                         VAR "a" 4 <10 : 0>
            a<10:0>                     VAR "b" 2 <10 : 0>
            b<10:0>                     VAR "c" 3 <10 : 0>
            c<10:0>                     VAR "j" 5 <10 : 0>
.           j<10:0>                     VAR "k" 1 <10 : 0>
            k<10:0>          2   PLUS 2 #4 4
            $)               3   GR 3 #10 6
      $(                     4   IFNOT 6            ACT 6
      a := b + 4             5   PLUS 4 3 5         ACT 7
      if (c>10) then         6   MINUS 4 3 5
            j := a + c       7   MINUS 5 #3 1
      else                   8   ENDMODULE 1
            j := a - c
      j := j - 3
      $)
$)

         SCHOLAR                       ICODE
```

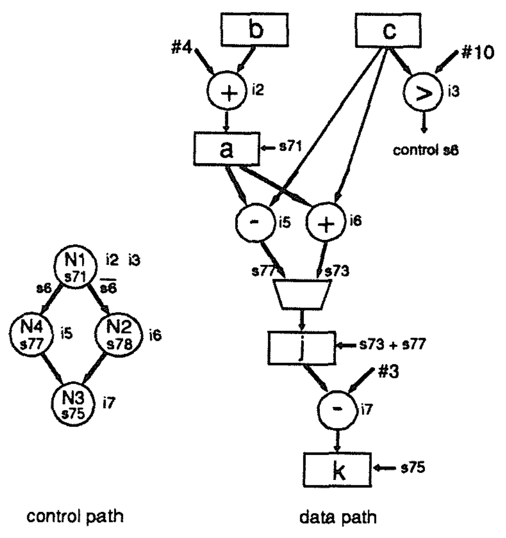
Figure 6.8 Example behavioural description.

Figure 6.9 illustrates the implementations achieved. Control signals are used to select the control paths and as multiplexer selectors and are shown next to the appropriate arcs and nets in the control and data path graphs respectively. Control signals are also used to load registers and select ALU functions and are shown on the data path graph by a horizontal arrow next to the appropriate unit. Node enable signals are shown in the control node that generates them and are used in conjunction with other control signals to select multiplexer inputs and ALU functions. Figure 6.9a shows the initial implementation created by both the MOODS and Scholyzer systems, as previously described in Chapter 3, where each instruction occurs in a unique control state. Figure 6.9b shows the implementation created by Scholyzer and Figure 6.9c and Figure 6.9d show two implementations created by MOODS.

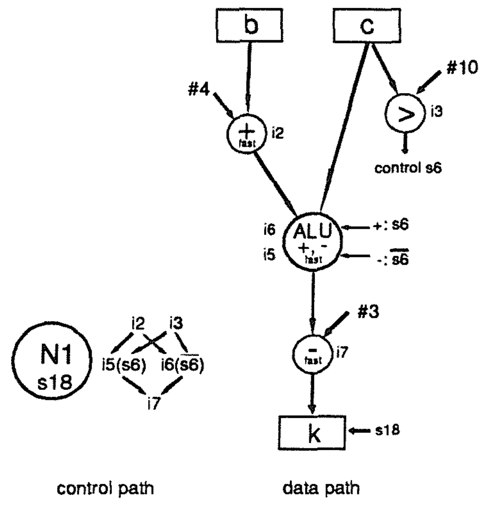
The design created by Scholyzer (Figure 6.9b) seems hardly improved when compared to the initial circuit. Scholyzer optimised the design in the following manner: firstly the control graph was compacted using an ASAP scheme. This resulted in nodes N2, N3 and N4 in the initial control graph being merged. The parallel section of the graph (nodes N5 and N6) could not be merged by Scholyzer as both instructions i5 and i6



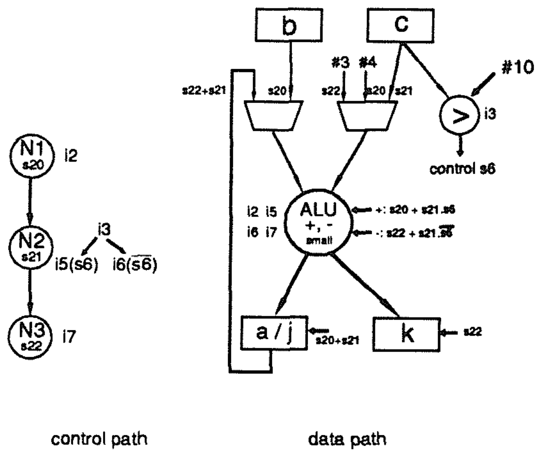
a. Initial circuit



b. Circuit created by SCHOLYZER



c. Fast circuit created by MOODS



d. Small circuit created by MOODS

Figure 6.9 Comparison of implementations for the TEST description.

write to the same register (although not concurrently). The mutually exclusiveness of the writes is not detected by Scholyzer and so the merge cannot take place. The parallel nodes cannot be combined with their predecessor node as a dependency exists between instruction i2 and instructions i5 and i6. In Scholyzer this is taken to be contentious as register *a* (the variable causing the dependency) is user defined. A distinction is made between user defined and compiler created variables. Only compiler generated registers (temp) can be removed, user defined registers cannot be optimised by bypassing as in the MOODS system. The second step in optimisation was to share operators. This resulted in no operators being shared as Scholyzer's sharing is extremely limited; only similar operators with common inputs or outputs can be shared and an area improvement must also result. This last condition is pre-programmed into the system as no cost function is used.

The designs generated by MOODS overcome all of the problems encountered with Scholyzer. In both designs produced by MOODS in Figure 6.9, the parallel instructions i5 and i6 have been merged into a common control state. This is allowed despite writing to the same register as MOODS detects that the instructions are mutually exclusive. In the fast implementation (Figure 6.9c) the dependency between instruction i2 and instructions i5 and i6 has been detected and on merging the instructions into the same control node, register *a* has been bypassed and dependency arcs added to the instruction graph. A similar situation occurs with instruction i7 and register *j*. The operators in the data path have also been implemented using fast cells, thus ensuring the fastest design has been created.

In the small implementation (Figure 6.9d) all control nodes have not been merged as sharing data path units was found by the system to improve the cost function greatest; instructions implemented by a common data path unit cannot occur concurrently. All of the *plus* and *minus* operators have been merged into one ALU. Out of the two ALUs in the database the smallest was selected. The MOODS system has detected that instructions i5 and i6 are mutually exclusive and may therefore share a data path operator even though they occur in the same control state. If the basic cell library had been used in generating the MOODS implementations then the ALU would have remained single *plus* and *minus* operators; the creation of the ALU demonstrates MOODS superior operator merging.

The above implementations were found by exploring the area-time design space for the design. The AT design space is shown in Figure 6.10, where the solid marks represent the implementations shown in Figure 6.9 and the two extreme designs on the optimal design curve are those shown in Figure 6.9c and Figure 6.9d. The position of Scholyzer's implementation in the design space was calculated using the information contained in the MOODS cell database, thus ensuring a fair comparison. A range of implementations were found between the fastest and smallest implementations as can be seen from the design points of Figure 6.10. The AT design space for TEST can be

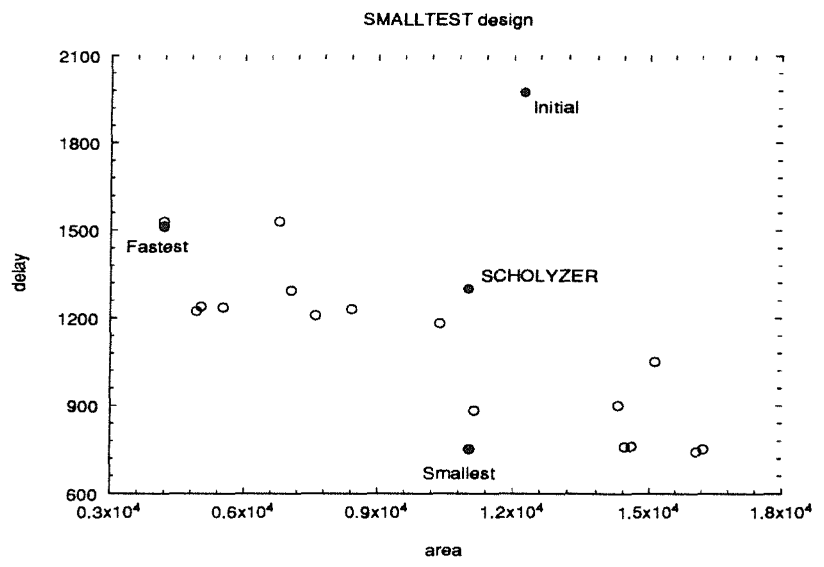


Figure 6.10 Explored AT design space for the TEST example.

characterized using the three points method described in Section 1.7 by specifying the points representing the fastest and smallest design points and the design point nearest the origin. Note that the optimal design curve is the set of best designs using these systems and not necessarily the best obtainable manually or otherwise.

6.3.2 Comparing MOODS with Other Systems

The following tables, Table 6.6, Table 6.7 and Table 6.8 show the results of the PARKER, ELLIP and TSENG benchmarks respectively. Each table consists of a compilation of the results given in various relevant papers as well as the results of comparable implementations generated by MOODS. The implementations generated by MOODS are in general competitive with those produced by the other systems. However, it should be noted that the MOODS system optimises a design with respect to real aspects of the design such as area (cost) and delay and as shown earlier the number of units and critical path lengths do not necessarily reflect these, therefore the relative real costs of implementations may be different from that implied in the tables below.

The results generated by MOODS for the PARKER design show that for the fast implementation the speed, if taken to be proportional to critical path length, is equal to the best produced by the other systems and for the small implementation the area, if taken to be the number of functional units, is also equal to the best produced by other systems. If both area and speed are taken into account, the secondary objective, that is,

Synthesis system and/or reference	Optimisation goal	No. of adders	No. of subtracts	No. of states	Control path length long / short	Maximum chain length
MAHA [24] Critical path first scheduling	speed	2	3	4	4	3
	cost	1	1	8	8	2
[63] Path based	speed	2	3	4	3/1	5
	cost	1	1	9	5/2	2
HAL [60] with mutual exclusion detection	speed	2	2	3	3	3
	average	2	1	4	4	3
	cost	1	1	8	8	2
SCHOLYZER	speed	5	6	10	8/3	1
MOODS with basic cell library	speed	4	6	3	3/3	5
	cost	1	1	9	9/9	4

Table 6.6 Comparison of systems for the PARKER benchmark.

area for the fast implementation and speed for the small implementation, is not as good as that produced by other systems. However, the real aspects of the implementations, taking into account the delay of individual units and area of registers and control, would result in different relative merit for the implementations.

The results generated by MOODS for the ELLIP design (see Table 6.7) show that faster and smaller implementations have been produced, however the argument given in the previous paragraph applies equally well to both the ELLIP and TSENG benchmarks. The number of nets shown in Table 6.7 can not be compared as the net count for MOODS relates to single ended nets, that is, nets having one source and one sink, whereas, nets in other systems are often counted as wired trees, that is, having one source and many sinks. In addition both control and data path nets are counted in MOODS but other systems count only data path nets.

The increase in multiplexer inputs is inevitable as no multiplexer optimisation is done by MOODS. The increase in registers is caused by the constraint that a design's output registers are not optimised in order to preserve their intended function.

Synthesis system and/or reference	Optimisation goal	No. of FSM / controller steps	No. of adders	No. of mults	No. of MUX inputs	No. of registers	No of nets†
[95] Source ref.	speed	17	4	4	--	--	--
HAL [60]	speed	17	3	3	31	12	56
	cost	21	2	1	--	--	--
[42]	speed	17	3	2	--	--	--
SPLICER [95]	cost	21	2	1	43	--	--
SPAID [19]	speed	17	3	2	26	17	22
	cost	21	2	1	19	19	14
MOODS Using basic cell library.	speed	14	3	3	79	22	113
	cost	37	1	1	68	25	97
Including ALU cell in cell library			ALU (+, *)				
APARTY [67]	speed	19	4		614 bits	192 bits	32 bits
MOODS	speed	15	3		224 bits	657 bits	1658 bits

† net count not comparable, see text.

Table 6.7 Comparison of systems for the ELLIP benchmark.

Table 6.8 compares the implementations of the TSENG benchmark produced by various systems. The implementations produced by MOODS were constrained to use particular ALUs to provide useful comparisons and the restriction on optimising I/O registers was removed so that register optimisations were comparable. Comparisons with other systems are difficult even though benchmarks are available and sometimes used. In comparing implementations it is important to know what other ALUs are available to the system from which to construct alternative implementations and what the costs associated with these units are compared to the ones used. In forcing MOODS to use particular ALUs for comparison of the TSENG benchmark the ALU costs were made low compared with other cells, however MOODS recognised this and often used more than one ALU as this was advantageous in reducing the overall cost of the design.

Synthesis system and/or reference	Optimisation goal	FUs and ALUs used in implementation	No. of registers	MUX data units (inputs)	No. of control states	No. of nets
FACET [26] Source ref.	speed	FACET ALUs (+, -, AND) (*, +, OR) (DIV)	8	6 (15)	4	31
HAL [25]			5	6 (13)	4	26
SPLICER [27]			7	4 (8)	4	14
MOODS			6	2 (16)	6	22
HAL [25]	speed	HAL ALUs (+, -, AND, OR, EQ) (*), (DIV)	6	-- (6)	4	28
	cost		5	-- (15)	>5	28
MOODS	speed		3	1 (2)	6	19
	cost		5	1 (2)	6	19
[51] Path search alg	speed	(+, -), (+, *, OR), (DIV, AND)	5	4 (9)	4	22
MOODS	speed		5	4 (8)	6	25
[42] SA alg	area	(+, -, *, DIV, AND, OR), (EQ)	8	2 (9) bus	8	--
MOODS			6	5 (10)	6	27
[42] SA alg	delay	(+, -, AND) (OR, DIV, *), (EQ)	8	4 (12) bus	5	--
MOODS			5	2 (4)	6	23

Table 6.8 Comparison of systems for the TSENG benchmark.

6.4 DESIGN SPACE EXPLORATION

The ability to explore the design space is an important aspect in both the development of a synthesis system and the optimisation and evaluation of a design. The design space has been used in Section 6.1.3 (pages 119 and 120) to determine an appropriate temperature reduction method and in Section 6.1.4 (page 122) to show the variation in design points due to different random number sequences. This section illustrates further how the design space may be used. In the context of optimisation a good design space is not only one that contains optimal designs but where the design points are clustered about the optimal design curve thus showing that near optimal designs can be consistently achieved.

6.4.1 Analysis of the Simulated Annealing Algorithm using Design Space Exploration

Figure 6.11 illustrates design spaces for the FRISC1 design. 25 design points were found and the same number of iterations were applied for each design space. Design spaces *a* and *b* used a schedule where $T_{start}=T_{end}=0$ (no schedule), that is, no degradations were applied to the design whereas design space *c* used the schedule given in Table 6.1. Design spaces *a* and *c* used a previously found design point in order to generate the

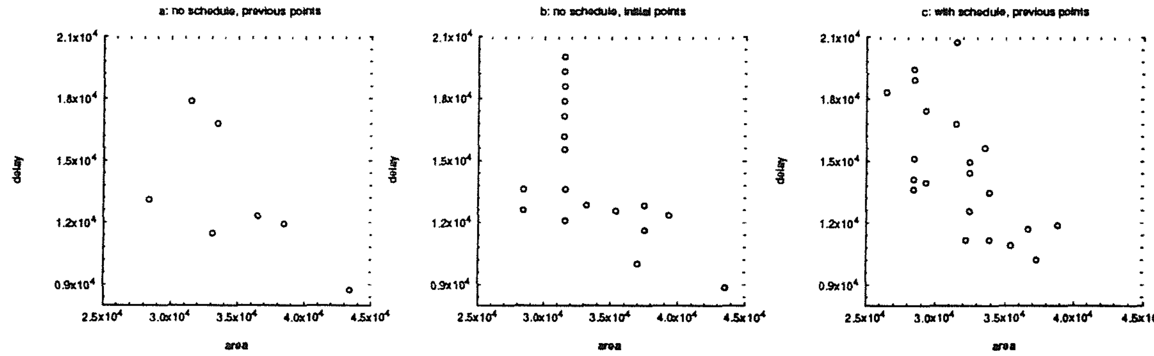


Figure 6.11 Comparison of improvement only and simulated annealing approaches and the use of initial or previous design points.

next whereas design space *b* used the initial implementation. Figure 6.7 showed that the starting point used, either initial or previous, has little effect of the position on the final design point. The slight improvement of design space *c* over *b* is therefore due to the annealing algorithm and as the difference is small, little backtracking (by design degradation) is required in reaching the design points. Design space *a* shows the importance and effectiveness of backtracking by design degradation inherent in the simulated annealing algorithm. With no schedule a previously optimised design point can not be degraded in order to find the next point therefore many points in design space *a* occur at the same position. The simulated annealing approach therefore finds slightly improved and a greater number of distinct design points than a no schedule (improvement only) approach.

The six graphs of Figure 6.12 illustrate how design points migrate from the initial design point to the optimised design points. The design spaces are shown at equal intervals of iterations. At the start of the schedule the temperature is high therefore both

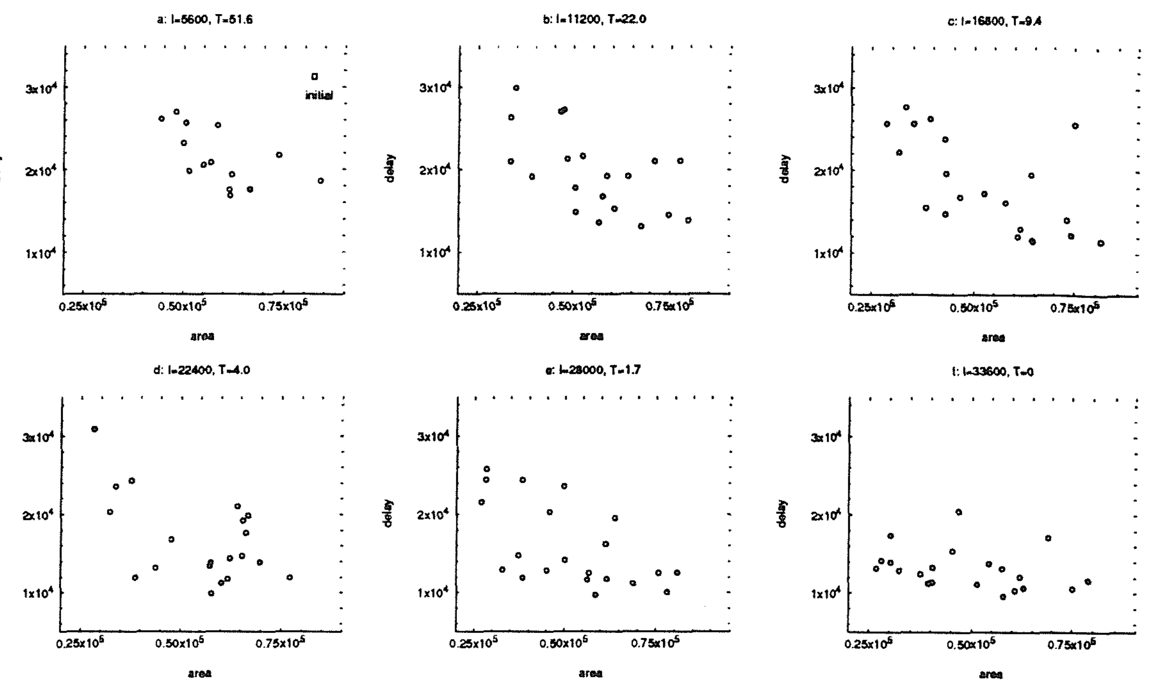


Figure 6.12 Illustration of how design points migrate from the initial point to the optimised points in the simulated annealing algorithm.

degradations and improvements are equally applied to the designs whose points consequentially lie between the worst (initial) design point and the optimal design curve (see Figure 6.12 graph a). A range of targets along the axis are used in automated design space exploration and as the number of iterations increases the design points become closer to their targets causing the points to spread out (graphs b to d). The optimal design points are obtained as the designs freeze (graph e). The points are improved further by iterations at T=0 causing some target values to be over reached; this is shown by the compression of points along the delay axis in graph f.

Both Figure 6.11 and Figure 6.12 show that most of the optimisation is done at the lower temperature steps, however, optimisation can not start at lower temperatures as the degradations which can be applied are temperature dependent. The results indicate that few design degradations are required in order to optimise a design. Some iterations are wasted in arbitrarily applying degradations, for example, there is little change between graphs b and c in Figure 6.12 despite applying 5600 iterations and a large temperature drop. This further illustrates the importance of the non-linear temperature reduction method.

6.4.2 Cost Function Priority Scaling

Analysis of early design spaces explored using a high and low priority, area/delay cost function showed that for an increase in the number of iterations per temperature step the high priority target was reached with greater accuracy. The fact that the higher priority objective had been met and was closer to the target should have resulted in more optimisation opportunities for the lower priority objective to be improved, however this was not always the case. The reason for this was thought to be in the determination of the change in energy, which is the first non-zero element in the energy change vector. The value returned is not dependent on its priority level therefore the design freezes at the same temperature for all priorities (due to the normalisation to stabilise T_{start}, see Section 6.1.1), which may result in a lower priority objective not being optimised as the design had frozen for all priorities. To allow lower priority objectives to be further optimised the change in energy was scaled by its priority thus lowering its freezing point.

Figure 6.13 shows the effect of scaling the change in energy ΔE by the cost function priority at which the change occurs. The annealing schedule shown in Table 6.1 was used for both design spaces. The two graphs have a spread of points covering similar areas of the AT design space. Two major differences are apparent; firstly, the best overall designs, that is the ones nearest the origin, do not occur in the "scaled" design space and secondly, a few faster designs have been found in the "scaled" design space.

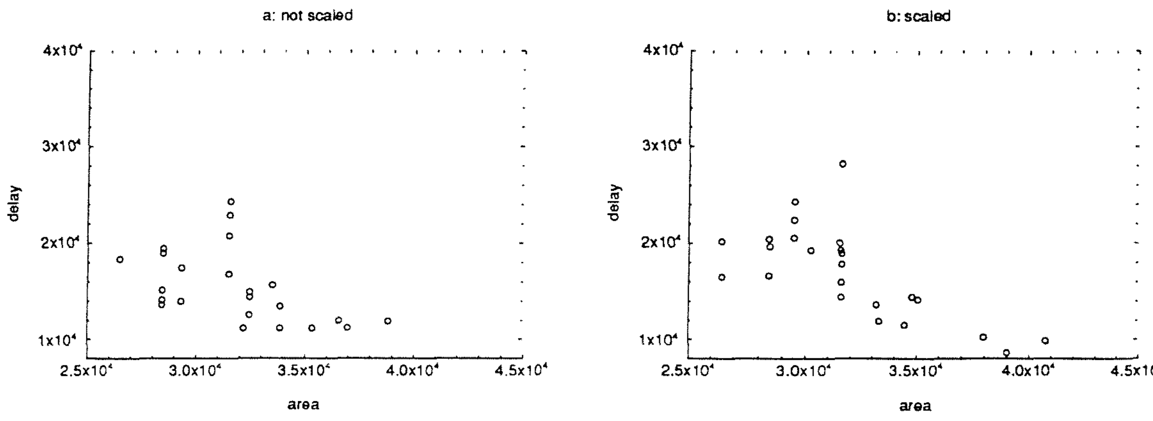


Figure 6.13 AT graphs showing the effect of scaling the change in energy ΔE by its corresponding priority in the cost function.

It would seem that the good overall designs have migrated to good designs in single design aspects. This may be due to small errors in estimating the effect of transformations. In comparing design spaces the clustering of designs about the optimal design curve is more important than the individual designs themselves. All the designs should be achievable using either the scaled or un-scaled methods as the transformations which produce designs have not changed only their application sequence. The priority scaled method for calculating the change in energy is not the default method used in the MOODS system; however the user may initiate its use if required using a command line parameter.

6.4.3 Design Analysis using Design Space Exploration

As shown by design spaces illustrated throughout this chapter the exploration of the design space gives the designer an insight into the trade-offs and consequentially the range of designs that can be achieved for a given design description. As described in Section 1.7 the design space may consist of any number of the design aspects monitored by the system. The two dimensional area-time (AT) design space is the archetypical design space where trade-offs are usually assumed to occur. Trade-offs between other design aspects are possible, however these are rarely seen in other systems. Synthesis systems are usually limited to an approximation of area and/or delay criteria and use pre-programmed trade-offs, resulting a high probability that real area-time trade-offs are not seen.

The MOODS synthesis system is capable of exploring an n-dimensional design space and can automatically explore two (as seen earlier) and three dimensions. Figure 6.14 shows a three dimensional design space consisting of area, delay and power. It can be seen that trade-offs occur between the area and delay criteria and the delay and power criteria; however almost no trade-off has occurred between the area and power objectives. This high correlation between the area and power criteria had been noted by Leive and Thomas [92]. The area and power criteria are related by cell construction and device characteristics and also by their method of calculation where both criteria are the sum of their respective area and power costs for each cell used. A further area-delay-power design space was created where additional cells that traded area and power were introduced into the cell library. The additional cells produced little improvement in area-

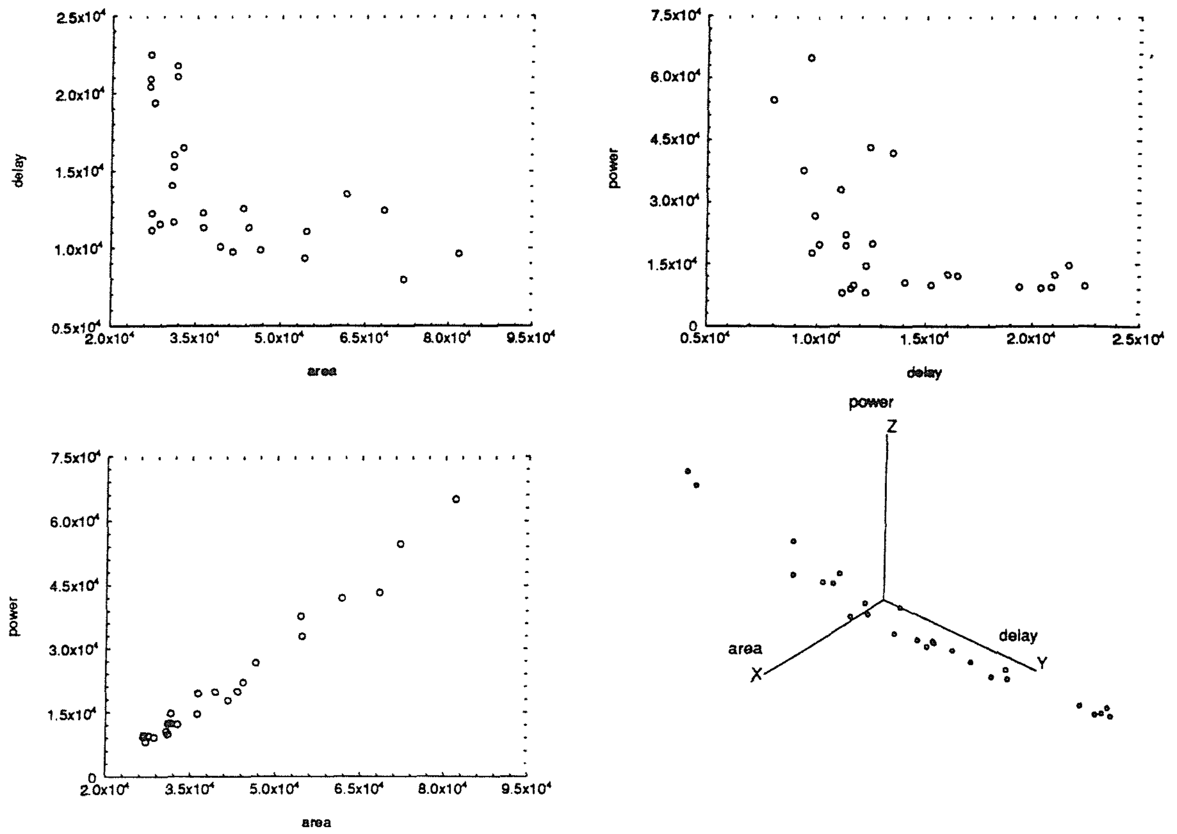


Figure 6.14 Automatic exploration of a three dimensional design space consisting of area, delay and power for the FRISC2 design.

power trade-offs thus indicating that the method of calculation has a greater influence on the correlation of criteria than the cells. Another three dimensional design space was generated using the area, delay and number of nets criteria. Again, a high correlation between the area and the number of nets was expected as the number of nets is closely related to the number of registers and shared units in the design. The resulting design space is shown in Figure 6.15.

As well as showing the range of designs and trade-offs between design aspects, the exploration of the design space can also be used to illustrate the effect of changes to the design description. As an example, module inline expansion was shown by the results of Table 6.3 to improve the design, however the improvement can be graphically illustrated using the design space. Figure 6.16 shows the design spaces for the descriptions without and with their modules inline expanded on the left and right of the figure respectively. The KALMANIO design has modules which are only called once, therefore by expanding the modules no additional units are created; however, by removing the module boundaries further optimisation opportunities are generated as optimisations

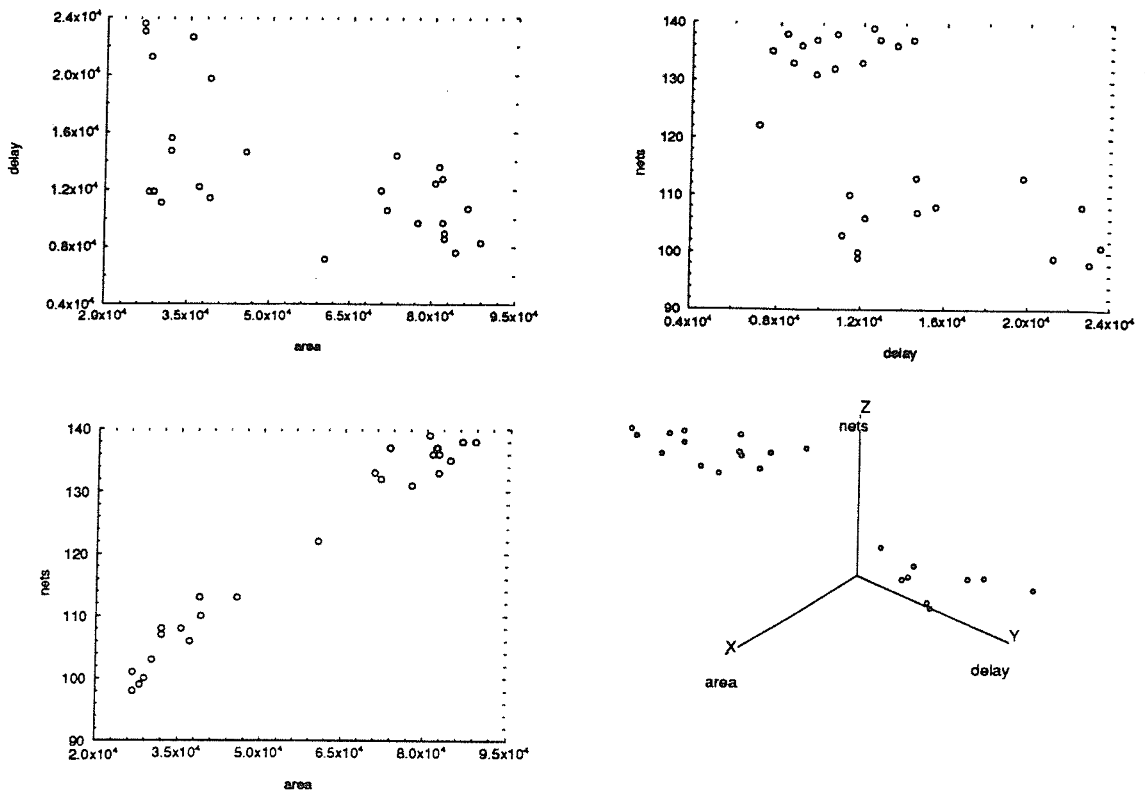


Figure 6.15 Automatic exploration of a three dimensional design space consisting of area, delay and number of nets for the FRISC2 design.

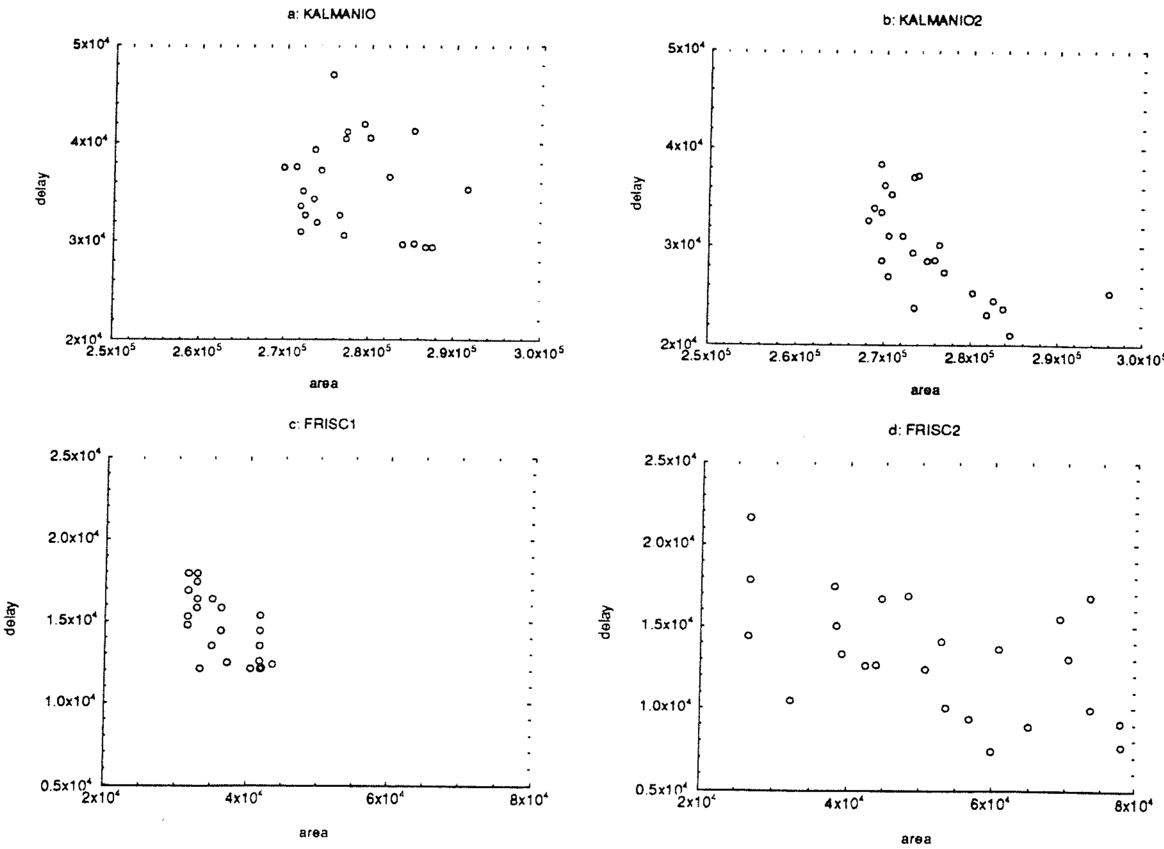


Figure 6.16 Design spaces illustrating the effects of module expansion.

between modules can not occur. The lack of additional units and the increased optimisation opportunities caused by module expansion are shown in the KALMANIO2 design space by the similar spread of design points which have been shifted towards the origin. In contrast to the KALMANIO design, the FRISC design calls its modules more than once therefore their expansion causes additional units to be created which can be used in further optimisations; this is shown by the increased range of implementations in the FRISC2 design space. The design points are closer to the origin, as in the KALMANIO2 design space, due to the removal of the module boundaries.

7 Conclusions and Further Work

7.1 CONCLUSIONS

The objective of the project has been met by the implementation of an "intelligent" silicon compiler, named MOODS, that provides both automated design space exploration and the ability to optimise a design, given as a behavioural description, with respect to multiple objectives. The modularity of the program and the completeness of the optimising transformations allows further optimisation algorithms to be easily incorporated into the system. This provides the conditions for concise controllable comparisons between different optimisation techniques.

The MOODS synthesis system, unlike many other systems, does not rely on pre-programmed optimisations. Many systems use tailored heuristics or algorithmic approaches based on a particular shape for the trade-off curve, as highlighted in Chapter 2. This has been shown by McFarland [31] to be inappropriate as trade-offs are technology dependent. The lack of pre-programmed optimisations in the MOODS system is possible through the abstractness of the simulated annealing algorithm, a specific case of the general adaptive heuristic. The MOODS system does not assume a shape to the trade-off curve but uses technology dependent information fed up from the cell library to evaluate its cost function which is used to determine the effectiveness of a transformation. Allocation is dependent on the cell library, where the choice of cells and their subsequent sharing are based on the user's criteria expressed in the cost function, rather than being pre-programmed into the system. In this way the system adapts to changes in technology and to the availability of cells in the cell library. The MOODS system can use any cell described in the cell library which implements one or more of the basic instructions, the most suitable ones being selected in order to met the user's objectives.

The results in Chapter 6 show that implementations found using the MOODS system are better than or equal to those achieved by other systems and that a varied set of implementations can be produced from a single behavioural specification. The varied set of good implementations is achieved through the generality of a comprehensive range of transformations, as well as by the "hill climbing" ability of the simulated annealing process. The results confirm that the opportunistic design modifications of the iterative method show the greatest power [69] and that larger designs allow more optimisation opportunities [44]. Experiments in Section 6.1 have shown that iterations applied at T=0 are an important fine refinement process and that the temperature reduction method is critical to the design quality.

The comparison of the results with those generated by other systems illustrates the importance of knowing all the design data in order to do a proper comparison; for example, a short clock period does not necessarily mean a fast implementation. The conditions for synthesis are also important in the comparison of systems; for example, some systems require the user to select the hardware from which to build an implementation, whereas other systems, including MOODS, select their own hardware. This should be taken into account when comparing systems as by specifying hardware the user is biasing, or in some cases binding, the hardware used in the final implementation.

Despite the execution time of MOODS being longer than that of other systems the improved variety and quality of the resulting implementations is considered more important. A design time of one minute or one hour is still faster than a hand crafted design.

The results of Section 6.4 show that design space exploration is an important aspect of designing by high-level synthesis and in the development of synthesis systems. The MOODS synthesis system includes an efficient method for automated design space exploration. It allows the designer to obtain a perspicuous characterization of the design space for a design and thus allows him to investigate alternative designs and determine whether a design can satisfy a variety of simultaneous constraints. The design spaces show that there are many near optimal implementations for a given description. A characterized design space can be used, as shown in Section 6.4, to investigate the effect of changes to the synthesis system. A design space shows graphically the impact of system changes, giving a better overall view of their effect than is obtained by individually synthesising designs.

The exploration of design spaces illustrates that there is a high correlation between some criteria, in particular area, power and the number of nets. This fact could be put to use in some systems to optimise criteria not explicitly optimised by the system; however, it would require an experienced designer to know how one criterion is related to another.

This would limit the system's potential users, defeating the object of high-level behavioural synthesis which is to provide a route to silicon for systems engineers. The ability of MOODS to simultaneously optimise a variety of criteria is therefore an important quality of a real synthesis system.

The MOODS system uses a general distributed architecture which is not targeted towards a specific application. Although good results have been achieved for a variety of applications it is considered that some applications requiring completely different design styles would benefit from application specific compilers.

The measure of goodness (MOG) which gives the usage of resources in a design has been shown to be an effective measure of the design's optimality without reference to particular objectives. In the compilation of the results it was noted that the designs flagged as being optimal by the automated design space exploration procedures had a better MOG than the non-optimal designs.

Register splitting, that is the separation of variable active times into distinct registers, would give an improvement in register sharing and reduce the bias introduced by the designer in the design description but only if variables in the description have more than one active period. However, this is rarely the case as the programming style used in writing behavioural descriptions means that users create additional variables when required rather than re-using existing ones. Of all the descriptions used in this project, which were written by a variety of designers, none had variables with sufficient active periods to allow better register sharing if the active periods had occurred in unique registers.

7.2 FUTURE WORK

Although MOODS is currently successful and can produce a range of implementations there is scope for further improvement. Possible improvements can be loosely divided into two areas, (a) improving the performance of the system, both in terms of producing better implementations and increasing its computational efficiency, and (b) promoting the use of the system by making it more user friendly and easier to integrate with other synthesis tools.

Changes to the system in order to produce better implementations can be subdivided into improvements to the transformations and improvements to the optimisation algorithm. Further transformations could be incorporated into the system, however, they would be increasingly specific to special cases, that is, they would be more like the rules in a rule based system. A rule applying transform could be devised which searches for a rule and a data structure pattern on which to apply it. Detailed studies would be required to determine effective rules that could be applied. Rules are usually concerned with translation, that is, replacing a sub-structure of the design with an equivalent one which improves the cost function or increases subsequent optimisation opportunities; for example, replacing a<(b+1) by a<=b or loop (un)winding. Additional transformations are required to perform multiplexer optimisation which although dependent on the other synthesis tasks can be effectively performed after the iterative optimisation of the design.

Changes to the optimisation algorithm comprise further investigation into the cost function and the method used to accept design degradations. As most transforms affect many design criteria it is unlikely that changes to low priority objectives would be returned from the `evaluate(cost_fn)` function. A two level cost function would therefore be adequate, thus providing a simpler cost function which could be more easily utilised in any additional optimisation algorithms. The results demonstrate that few degradations require to be performed in order to find a near optimal implementation, consequentially the simulated annealing algorithm may be wasting computational effort by randomly applying design degradations. The degradations require to be performed in a more controlled fashion in order to save design time. Sequence heuristics as described in Section 5.2.3 apply degradations only when no transformations that improve the design can be found. A tentative implementation of the sequence heuristic produced design times which were considerably longer than those of the simulated annealing algorithm. The reason for this is thought to be that the sequence heuristic wastes time in trying to find transformations which specifically improve or degrade the design, whereas, simulated annealing finds any transform and conditionally applies it. An alternative reason could be an implementation error, however a detailed investigation of the sequence heuristic was beyond the scope of the project. A great deal of research could be done in studying the sequence heuristic and in finding new adaptive heuristics that improve on simulated annealing, which despite its critics has proved to be a competitive approach to high-level synthesis. A good approach would be to combine tailored and adaptive heuristics; for example, during optimisation control graphs are often reduced to a sequential section by merging parallel nodes, this could be performed by a fast tailored heuristic prior to iterative optimisation.

Further improvements in the speed of the system could be obtained by determining all data dependencies before synthesis and either storing them using the present data structure but with an *active* flag or using look-up tables. This would help speed up contention tests which are computationally expensive procedures.

The addition of timing constraints is required in order to allow the user to specify a particular timing between events. This could be included by the addition of time dependency links between instructions in control states. These links could also be used to maintain multicycled instructions during optimisation. Although multicycled instructions can be generated they are not fully integrated with the transformations. Timing constraints which must be met may be difficult to integrate into the transformations and are likely to limit the optimisation opportunities. Timing links could also be used in the generation of pipelined architectures where the pipeline cells would be represented in the cell library.

Layout effects require taking into account in order to increase the confidence of an implementation reaching specific goals. Wiring effects can be estimated using a macro cell floorplanner as in Fasolt [93] or fed up from layout tools. Alternatively, synthesis tools can specify maximum wire lengths thus providing a predictable performance.

In order to improve the usability of the MOODS system and ensure its continued use, additional language interfaces and tools are required. At present the behavioural description may be written using either the ELLA or SCHOLAR languages. Although ELLA is suitable as the output structural netlist, as it can be simulated using the ELLA simulator, it is not an appropriate language for writing behavioural descriptions as it lacks simple algorithmic constructs such as loops. The SCHOLAR language, although a good functional language with is own simulator, is not widely used. Subsets of VHDL [94] would be a valuable improvement for both the behavioural input and structural output. Further structural output formats would also be a benefit as a range of logic and layout tools could be used.

Extra tools which would aid the designer in producing an implementation could include automated test structure and test pattern generation [95] and automated methods in finding the annealing schedule or other adaptive heuristic parameters. Additional flexibility would be achieved if the designer could define operators to be used in the description that could subsequently be optimised by the system. At present user defined modules can be described, however their instances are not optimised.

Currently the user interface for the MOODS system is a textual one. Once initiated the system creates an initial implementation and displays the MOODS prompt. The user issues commands at the MOODS prompt and data scrolls up the screen. When satisfied with the implementation the user exits the system whereupon output files are created. An improvement could be made in the presentation of the implementation to the designer. At present the data and control paths of the implementation are described in a number of output files and their nodes can be examine from the MOODS prompt. Alternative methods of design representation could either be textural, by back-annotation of the implementation to the original behavioural description, or by graphical output. Textural back-annotation would be useful as the designer could see how the original description had been altered by the system and could be useful in the development of new algorithms. Graphical output would also be valuable in order for the designer to visualise the implementation. By using platforms with graphical interfaces the user could watch the design being optimised and interact with it to produce the required implementation.

Appendix A
The Intermediate Code

The intermediate code (ICODE) is input to the MOODS system in the form of a binary file. The ICODE represents the design as non-recursive modules described at the register-transfer (RT) level. Each module has a variable declaration part and a process part. Each process part consists of a set of processes each with a unique process number. Each process represents an instruction and associated with it is an activation list, that is, a list of processes to be activated when the current process ends. A process may start only when all preceding processes have terminated, indicated by a token. A *collect* instruction is used where the preceding processes are executed concurrently. Its effect is to wait for a specified number of tokens before activating subsequent processes. Each instruction is specified by its name, the set of inputs and outputs and the activation list, where conditional instructions have two or more condition dependent activation lists. The ICODE is stored in a binary file, however a textural representation of the ICODE can also be created.

<u>The ICODE Instruction Set and Binary File</u>

<u>1. Logical and Arithmetic Operators.</u>

AND	*2 - inputs*		*n - outputs*	*process list*
33	I_1	I_2	n $O_1, O_2, \ldots O_n$	p $A_1, A_2, \ldots A_p$

OR	*2 - inputs*		*n - outputs*	*process list*
34	I_1	I_2	n $O_1, O_2, \ldots O_n$	p $A_1, A_2, \ldots A_p$

Similarly the following operators are defined in the same way:

XOR	35	*NOT* (1 INPUT)	30	*LSHIFT*	31
RSHIFT	32	*ROL*	11	*ROR*	12
PLUS	14	*MINUS*	15	*NEG* (1 INPUT)	17
EQ	20	*LS*	21	*LE*	22
NE	23	*GE*	24	*GR*	25
MULT	18	*DIV*	19		

2. Variable Access and Modification.

MOVE	*1 - input*	*n - outputs*	*process list*
50	I_1	n $O_1, O_2, \ldots O_n$	p $A_1, A_2, \ldots A_p$

TRUE	*n - outputs*	*process list*
4	n $O_1, O_2, \ldots O_n$	p $A_1, A_2, \ldots A_p$

HIGHZ	*n - outputs*	*process list*
16	n $O_1, O_2, \ldots O_n$	p $A_1, A_2, \ldots A_p$

MEMREAD	*RAM No.*	*index*	*n - outputs*	*process list*
9	I_1	I_2	n $O_1, O_2, \ldots O_n$	p $A_1, A_2, \ldots A_p$

MEMWRITE	*RAM No.*	*index*	*input variable*	*process list*
44	I_1	I_2	I_3	p $A_1, A_2, \ldots A_p$

3. Conditional Branching.

IF / IFNOT	*1 - input*	*process list (true)*	*process list (false)*
57 / 58	I_1	n $A_1, A_2, \ldots A_n$	m $A_1, A_2, \ldots A_m$

SWITCHON	*input*	*No. cases*	*Default label No.*	*(case const, label No.) 1..m*
70	I_1	M	$L_{default}$	$(C_1, L_1)\ (C_2, L_2)\ \ldots\ (C_m, L_m)$

COUNT	*2 - inputs*		*process list (eq)*	*process list (ne)*
48	I_{inc}	I_{term}	n $A_1, A_2, \ldots A_n$	m $A_1, A_2, \ldots A_m$

4. Directives.

VAR	*"name"*	*var No.*	*upper bit*	*lower bit*
109	n <n chars>	V_n	upb	lwb

ALIAS	*"name"*	*var No.*	*upper bit*	*lower bit*	*parent var*	*upb*	*lwb*
108	n <n chars>	V_n	upb	lwb	V_p	pupb	plwb

ROM	*"name"*	*var No.*	*upper bit*	*lower bit*	*low address*	*values 1..n*
41	n <n chars>	V_n	upb	lwb	lwaddr	n V_1, V_2, ... V_n

RAM	*"name"*	*var No.*	*upper bit*	*lower bit*	*low address*	*high address*
42	n <n chars>	V_n	upb	lwb	lwaddr	hiaddr

COUNTER	*"name"*	*var No.*	*upper bit*	*lower bit*
106	n <n chars>	V_n	upb	lwb

COUNTDN	*"name"*	*var No.*	*upper bit*	*lower bit*
107	n <n chars>	V_n	upb	lwb

LINE	*line No.*
28	Ln

LABEL	*label No.*	*process list*
54	L_1	n A_1, A_2, ... A_n

5. Special Instructions.

COLLECT	*No. of tokens*	*process list*
53	T	n A_1, A_2, ... A_n

PROGRAM	*"name"*	*n - inputs*	*m - outputs*	*process list*
76	n <n chars>	n A_1, A_2, ... A_n	m A_1, A_2, ... A_m	p A_1, A_2, ... A_p

MODULE	*"name"*	*n - inputs*	*m - outputs*	*process list*
74	n <n chars>	n A_1, A_2, ... A_n	m A_1, A_2, ... A_m	p A_1, A_2, ... A_p

MODULEAP	*Label*	*n - inputs*	*m - outputs*	*process list*
51	L	n A_1, A_2, ... A_n	m A_1, A_2, ... A_m	p A_1, A_2, ... A_p

ENDMODULE	*process No.*
77	A_1

REMARKS:

1. A process number is either a single number or a label. If it is a label this is indicated by a preceding zero byte.
2. An input is either a constant or a variable number. When it is a constant it is preceded by a zero.

APPENDIX B
ELLA SIMULATION EXAMPLE

Behavioural Synthesis using MOODS

The 5-point Winograd Fourier Transform (WIN) example has been used to generate area and delay efficient implementatons. The correctness of the resulting implementations are verified by simulation and comparison to the simulation of the source description; thus demonstrating that for this design correctness by construction has been achieved. The initial ELLA behavioural description of the Winograd Fourier Transform is shown below.

```
INT word_width = 6.
INT inter_word_width = 10.
INT real = 1, imag = 2.

# Winograd Multiplication constants  #
CONST coeffs_positive = (b"0100000000",
                         b"1011000000",
                         b"0010001111",
                         b"0110001010",
                         b"0010010110",
                         b"0001011101").
CONST coeffs_negative = (b"1100000000",
                         b"0101000000",
                         b"1101110001",
                         b"1001110110",
                         b"1101101010",
                         b"1110100011").

CONST init = b"0000000000".

FN WINOGRAD_5 = ([10]STRING[word_width]bit: ip_short) ->
                ([10]STRING[word_width]bit):
(SEQ
# Extend the internal word length to avoid overflows in the calculation    #
# Using sign extension                                                     #
VAR ip := [10]init;
[INT j=1..10]
   ip[j] := STRING [4]ip_short[j][1] CONC ip_short[j];

# Premultiplication additions #
LET
  s1 = (ip[(1*2)+real] PLUS_STR ip[(4*2)+real],
        ip[(1*2)+imag] PLUS_STR ip[(4*2)+imag]),
  s2 = (ip[(1*2)+real] MINUS_STR ip[(4*2)+real],
        ip[(1*2)+imag] MINUS_STR ip[(4*2)+imag]),
  s3 = (ip[(3*2)+real] PLUS_STR ip[(2*2)+real],
        ip[(3*2)+imag] PLUS_STR ip[(2*2)+imag]),
  s4 = (ip[(3*2)+real] MINUS_STR ip[(2*2)+real],
        ip[(3*2)+imag] MINUS_STR ip[(2*2)+imag]),
  s5 = (s1[real] PLUS_STR s3[real],
        s1[imag] PLUS_STR s3[imag]),
  s6 = (s1[real] MINUS_STR s3[real],
        s1[imag] MINUS_STR s3[imag]),
    s7 = (s2[real] PLUS_STR s4[real],
          s2[imag] PLUS_STR s4[imag]),
    s8 = (s5[real] PLUS_STR ip[(0*2)+real],
          s5[imag] PLUS_STR ip[(0*2)+imag]),

  # Perform multiplications #
    m0 = ((coeffs_positive[1] MULT_STR s8[real]),
          (coeffs_positive[1] MULT_STR s8[imag])),
    m1 = ((coeffs_positive[2] MULT_STR s5[real]),
          (coeffs_positive[2] MULT_STR s5[imag])),
    m2 = ((coeffs_positive[3] MULT_STR s6[real]),
          (coeffs_positive[3] MULT_STR s6[imag])),
    m3 = ((coeffs_negative[4] MULT_STR s2[imag]),
          (coeffs_positive[4] MULT_STR s2[real])),
    m4 = ((coeffs_negative[5] MULT_STR s7[imag]),
          (coeffs_positive[5] MULT_STR s7[real])),
    m5 = ((coeffs_negative[6] MULT_STR s4[imag]),
          (coeffs_positive[6] MULT_STR s4[real])),

    s9 = (m0[real][2..inter_word_width+1] PLUS_STR m1[real][2..inter_word_width+1],
         m0[imag][2..inter_word_width+1] PLUS_STR m1[imag][2..inter_word_width+1]),
    s10 = (s9[real] PLUS_STR m2[real][2..inter_word_width+1],
          s9[imag] PLUS_STR m2[imag][2..inter_word_width+1]),
    s11 = (s9[real] MINUS_STR m2[real][2..inter_word_width+1],
          s9[imag] MINUS_STR m2[imag][2..inter_word_width+1]),
    s12=(m3[real][2..inter_word_width+1] MINUS_STR m4[real][2..inter_word_width+1],
        m3[imag][2..inter_word_width+1] MINUS_STR m4[imag][2..inter_word_width+1]),
    s13=(m4[real][2..inter_word_width+1] PLUS_STR m5[real][2..inter_word_width+1],
        m4[imag][2..inter_word_width+1] PLUS_STR m5[imag][2..inter_word_width+1]),
    s14 = (s10[real] PLUS_STR s12[real],
          s10[imag] PLUS_STR s12[imag]),
    s15 = (s10[real] MINUS_STR s12[real],
          s10[imag] MINUS_STR s12[imag]),
    s16 = (s11[real] PLUS_STR s13[real],
          s11[imag] PLUS_STR s13[imag]),
    s17 = (s11[real] MINUS_STR s13[real],
          s11[imag] MINUS_STR s13[imag]);

# Assemble output at full internal precision #
LET long_output = (m0[real][2..inter_word_width+1],
            m0[imag][2..inter_word_width+1],
            s14[real],s14[imag],s16[real],s16[imag],
            s17[real],s17[imag],s15[real],s15[imag]);

# Select only word_length for each output #
LET output = [INT i=1..10]
long_output[i][inter_word_width-word_width+1..inter_word_width];

OUTPUT output
).
```

Simulation of the Behavioural Description

To simulate the initial behavioural description the Winograd description was compiled into an ELLA context with the arithmetic and shell functions. The shell functions convert the bit string inputs and outputs into integers. Using integers makes the simulation results easier to interpret.

Simulating the design gave the following results:

```
FN  WINOGRAD_TC
  *** time = 0 ***

Sim <- ma
WINOGRAD_TC = ? ? ? ? ? ? ? ? ? ?

Sim <- cp i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0
WINOGRAD_TC = i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0

Sim <- cp i/25 i/25 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0
WINOGRAD_TC = i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12

Sim <- cp i/1 i/2 i/3 i/4 i/5 i/6 i/7 i/8 i/9 i/10
WINOGRAD_TC = i/12 i/15 i/0 i/-6 i/-3 i/-5 i/-3 i/-1 i/-6 i/0

Sim <- cp i/10 i/10 i/10 i/9 i/8 i/4 i/3 i/2 i/2 i/1
WINOGRAD_TC = i/16 i/13 i/-4 i/10 i/-1 i/1 i/3 i/3 i/6 i/-2
```

Moods Synthesis

To synthesis the design the description was compiled in the ELLA environment and then converted to ICODE using the ELLA to ICODE interface. The ICODE is the input to MOODS from which a variety of implementations can be produced using different cost functions in the optimisation process.

The MOODS cell library was changed to include only the basic cells and one ALU performing the plus and minus operations. Two implementations were generated one optimsied for area, the other for delay. The annealing schedule used for both optimised designs was similar to the schedule derived in the results chapter and the main objective was given a priority of 1 and a minimising target (zero). The remaining objective was given a priority of 3 also with a minimising target.

The information on each design was extracted from the appropriate design analysis files (`.daf`) that were generated by MOODS during synthesis. To simulate the implementation the generated ELLA netlist was compiled into the ELLA environment with the basic and user parameterized cell macros and shell functions. When simulating a design each ELLA time step corresponds to a change in the state of the controller. It is important to know when the outputs are valid; this can be determined by studying the control graph file. In the implementations generated the outputs became valid at the end/start of the controller cycle. The start control node enable signal was monitored, therefore when this becomes true the output is valid.

Design data for the un-optimised implementation:

```
CELLS USED IN UN-OPTIMISED DESIGN
        14   subtract                          172  general ctrl cell
        132  register                          20   adder
        10   IO port (temp)                    12   multiply

UN-OPTIMISED DESIGN DATA
storage:         132 units       1520 bits        active area =  82080.000 sq um
functional:      46 units        460 bits         active area =  86280.000 sq um
ports:           10 units        60 bits          active area =      0.000 sq um
interconnects:   0 units         0 bits           active area =      0.000 sq um
TOTAL:           188 units       2040 bits        active area = 168360.000 sq um
Control:         172 units              active area (approx) =     860.000 sq um
Critical Path Length: 162
Max Control Node Delay =    111.600 ns
MOG - Clk use: 20%, Reg use: 7%, Unit use: 0%, AVG: 9%
```

Area Optimised Design

Design data for the area optimised implementation:

```
CELLS USED IN FINAL DESIGN
        29   multiplexer                       1    subtract
        106  general ctrl cell                 94   register
        1    adder                             10   IO port (temp)
        4    ALU -,+                           9    multiply

DESIGN DATA AFTER OPTIMISATION
storage:         94 units        1107 bits        active area =  59778.000 sq um
functional:      15 units        150 bits         active area =  20140.000 sq um
ports:           10 units        60 bits          active area =      0.000 sq um
interconnects:   29 units        347 bits         active area =  24984.000 sq um
TOTAL:           148 units       1664 bits        active area = 104902.000 sq um
Control:         106 units              active area (approx) =     530.000 sq um
```

```
Critical Path Length: 105
Max Control Node Delay =    120.900 ns
MOG - Clk use: 29%, Reg use: 0%, Unit use: 2%, AVG: 10%

FINAL COST FUNCTION
      --- COST FUNCTION VECTOR ---
      CRITERION | area (sq um)   | T delay (ns)
      PRIORITY  |        1       |      3
      INITIAL   |165980.000      | 18079.199
      TARGET    |      0.000     |     0.000
      PREVIOUS  |102187.000      | 12573.600
      PRESENT   |102192.000      | 12694.500
      ESTIMATE  |102192.000      | 12694.500
```

Simulation results for area optimised moods design:

```
FN  WINOGRAD_TC
  *** time = 0 ***

Sim <- mc WINOGRAD_5.c71
c71 = b'0

Sim <- mc
WINOGRAD_TC = i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0

Sim <- cp  i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 b'1, ti +1
  *** time = 1 ***
c71 := b'1

Sim <- cp  i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 b'0, ti +102
  *** time = 2 ***
c71 := b'0
  *** time = 103 ***
c71 := b'1, WINOGRAD_TC = i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0

Sim <- cp  i/25 i/25 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 b'0, ti +102
  *** time = 104 ***
c71 := b'0
  *** time = 197 ***
WINOGRAD_TC := i/0 i/0 i/0 i/12 i/0 i/0 i/12 i/0 i/0 i/0
  *** time = 198 ***
WINOGRAD_TC := i/0 i/0 i/0 i/12 i/0 i/0 i/12 i/0 i/0 i/12
  *** time = 199 ***
WINOGRAD_TC := i/0 i/0 i/12 i/12 i/0 i/0 i/12 i/0 i/0 i/12
  *** time = 202 ***
WINOGRAD_TC := i/0 i/0 i/12 i/12 i/12 i/0 i/12 i/0 i/0 i/12
  *** time = 203 ***
WINOGRAD_TC := i/0 i/12 i/12 i/12 i/12 i/0 i/12 i/0 i/0 i/12
  *** time = 205 ***
c71 := b'1, WINOGRAD_TC := i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12
```

```
Sim <- cp i/1 i/2 i/3 i/4 i/5 i/6 i/7 i/8 i/9 i/10 b'0, ti +102
  *** time = 206 ***
c71 := b'0
  *** time = 299 ***
WINOGRAD_TC := i/12 i/12 i/12 i/-6 i/12 i/12 i/-3 i/12 i/12 i/12
  *** time = 300 ***
WINOGRAD_TC := i/12 i/12 i/12 i/-6 i/12 i/12 i/-3 i/12 i/12 i/0
  *** time = 301 ***
WINOGRAD_TC := i/12 i/12 i/0 i/-6 i/12 i/12 i/-3 i/12 i/12 i/0
  *** time = 304 ***
WINOGRAD_TC := i/12 i/12 i/0 i/-6 i/-3 i/12 i/-3 i/12 i/12 i/0
  *** time = 305 ***
WINOGRAD_TC := i/12 i/15 i/0 i/-6 i/-3 i/12 i/-3 i/12 i/12 i/0
  *** time = 307 ***
c71 := b'1, WINOGRAD_TC := i/12 i/15 i/0 i/-6 i/-3 i/-5 i/-3 i/-1 i/-6 i/0

Sim <- cp i/10 i/10 i/10 i/9 i/8 i/4 i/3 i/2 i/2 i/1 b'0, ti +102
  *** time = 308 ***
c71 := b'0
  *** time = 401 ***
WINOGRAD_TC := i/12 i/15 i/0 i/10 i/-3 i/-5 i/3 i/-1 i/-6 i/0
  *** time = 402 ***
WINOGRAD_TC := i/12 i/15 i/0 i/10 i/-3 i/-5 i/3 i/-1 i/-6 i/-2
  *** time = 403 ***
WINOGRAD_TC := i/12 i/15 i/-4 i/10 i/-3 i/-5 i/3 i/-1 i/-6 i/-2
  *** time = 406 ***
WINOGRAD_TC := i/12 i/15 i/-4 i/10 i/-1 i/-5 i/3 i/-1 i/-6 i/-2
  *** time = 407 ***
WINOGRAD_TC := i/12 i/13 i/-4 i/10 i/-1 i/-5 i/3 i/-1 i/-6 i/-2
  *** time = 409 ***
c71 := b'1, WINOGRAD_TC := i/16 i/13 i/-4 i/10 i/-1 i/1 i/3 i/3 i/6 i/-2
```

Delay Optimised Design

Design data for the delay optimised implementation:

```
CELLS USED IN FINAL DESIGN
        27   multiplexer              5    subtract
        97   general ctrl cell        98   register
        6    adder                    10   IO port (temp)
        9    ALU -,+                  11   multiply

DESIGN DATA AFTER OPTIMISATION
storage:         98 units       1184 bits        active area =   63936.000 sq um
functional:      31 units       310 bits         active area =   50270.000 sq um
ports:           10 units       60 bits          active area =       0.000 sq um
interconnects:   27 units       354 bits         active area =   25488.000 sq um
TOTAL:           166 units      1908 bits        active area =  139694.000 sq um
Control:         97 units               active area (approx) =     485.000 sq um
Critical Path Length: 95
Max Control Node Delay =    111.600 ns
```

```
MOG - Clk use: 34%, Reg use: 0%, Unit use: 1%, AVG: 11%

FINAL COST FUNCTION
      --- COST FUNCTION VECTOR ---
      CRITERION | T delay (ns)  | area (sq um)
      PRIORITY  |       1       |       3
      INITIAL   | 18079.199     |165980.000
      TARGET    |     0.000     |     0.000
      PREVIOUS  | 10602.000     |137659.000
      PRESENT   | 10602.000     |136939.000
      ESTIMATE  | 10602.000     |136939.000
```

Simulation results for delay optimised moods design:

```
FN  WINOGRAD_TC
  *** time = 0 ***

Sim <- mc WINOGRAD_5.c22
c22 = b'0

Sim <- mc
WINOGRAD_TC = i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0

Sim <- cp i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 b'1, ti +1
  *** time = 1 ***
c22 := b'1
```

`Sim <- cp i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 b'0, ti +90`

```
  *** time = 2 ***
c22 := b'0
  *** time = 91 ***
```

`c22 := b'1, WINOGRAD_TC = i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0`

```
  *** time = 92 ***
  *** time = 181 ***
c22 := b'1
```

`Sim <- cp i/25 i/25 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 b'0, ti +90`

```
  *** time = 182 ***
c22 := b'0
  *** time = 266 ***
WINOGRAD_TC := i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/12
  *** time = 267 ***
WINOGRAD_TC := i/0 i/0 i/0 i/0 i/0 i/0 i/0 i/12 i/0 i/12
  *** time = 268 ***
WINOGRAD_TC := i/0 i/0 i/0 i/12 i/0 i/0 i/0 i/12 i/0 i/12
  *** time = 269 ***
WINOGRAD_TC := i/0 i/0 i/0 i/12 i/0 i/0 i/12 i/12 i/0 i/12
  *** time = 270 ***
WINOGRAD_TC := i/0 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12
  *** time = 271 ***
```

`c22 := b'1, WINOGRAD_TC := i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12`

```
Sim <- cp i/1 i/2 i/3 i/4 i/5 i/6 i/7 i/8 i/9 i/10 b'0, ti +90
  *** time = 272 ***
c22 := b'0
  *** time = 356 ***
WINOGRAD_TC := i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/0
  *** time = 357 ***
WINOGRAD_TC := i/12 i/12 i/12 i/12 i/12 i/12 i/12 i/-1 i/12 i/0
  *** time = 358 ***
WINOGRAD_TC := i/12 i/12 i/12 i/-6 i/12 i/12 i/12 i/-1 i/12 i/0
  *** time = 359 ***
WINOGRAD_TC := i/12 i/12 i/12 i/-6 i/12 i/12 i/-3 i/-1 i/12 i/0
  *** time = 360 ***
WINOGRAD_TC := i/12 i/15 i/0 i/-6 i/-3 i/-5 i/-3 i/-1 i/-6 i/0
  *** time = 361 ***
c22 := b'1

Sim <- cp i/10 i/10 i/10 i/9 i/8 i/4 i/3 i/2 i/2 i/1 b'0, ti +90
  *** time = 362 ***
c22 := b'0
  *** time = 446 ***
WINOGRAD_TC := i/12 i/15 i/0 i/-6 i/-3 i/-5 i/-3 i/-1 i/-6 i/-2
  *** time = 447 ***
WINOGRAD_TC := i/12 i/15 i/0 i/-6 i/-3 i/-5 i/-3 i/3 i/-6 i/-2
  *** time = 448 ***
WINOGRAD_TC := i/12 i/15 i/0 i/10 i/-3 i/-5 i/-3 i/3 i/-6 i/-2
  *** time = 449 ***
WINOGRAD_TC := i/12 i/15 i/0 i/10 i/-3 i/-5 i/3 i/3 i/-6 i/-2
  *** time = 450 ***
WINOGRAD_TC := i/12 i/13 i/-4 i/10 i/-1 i/1 i/3 i/3 i/6 i/-2
  *** time = 451 ***
c22 := b'1, WINOGRAD_TC := i/16 i/13 i/-4 i/10 i/-1 i/1 i/3 i/3 i/6 i/-2
```

Remarks

The final cost functions of the two optimised designs show that a trade-off between area and delay has occurred. The major trade-off is between the length of the control graph and the sharing of data path units. Operations sharing a data path unit can not be executed concurrently and therefore must be in non-concurrent control states. Conversely, concurrent operations can not share the same data path unit. (Mutually exclusive operations are a special case exception which is recognised by MOODS.) The number of functional units (implemented by the add, subtract, ALU and multiply cells) indicate the amount of sharing. The initial number of functional units (in the un-optimised implementation) is 46. This is equal to the number of operations in the description as in the un-optimised design each operation is allocated one functional unit. In the area optimised design the number of functional units is 15 indicating that each unit shares on average 3 operations; its critical path length is 105. In the delay optimised design the number of functional units is 31, twice that in area optimised design (1-2 operations per functional unit). The reduction in sharing is accompanied by (traded off against) a decrease in critical path length to 95.

Appendix C
Data Structures

```
/*======================================================================*/
/* Structure definitions                                      21/7/89   */
/* Keith R Baker                               Southampton University   */
/*======================================================================*/

/* Structure containing all list heads for use in main program and for  */
/* passing information to and from procedures.                          */

struct all_heads {
   int high_sig,hi_group_no;
   struct module_node *mod_head;
   struct IGR_node *IGR_head;
   struct IO_arcs *arc_list;
   struct variable_node *var_head;
   struct DP_node *DP_head;
   struct condition_list *cond_head;
   struct label_node *label_head;
   struct net_data *net_list;

/* Structure containing all technology lists such as cell data.         */

   struct cell_data *cell_head;                    /* list of all cells */
   struct techno_units *units;
   struct cell_data *mux_cell,*reg_cell;     /* pointers to these basic cells */
   };

/* IGR linked list node, also indicates petri-net arcs.                 */

struct IGR_node {
   int node_no,delay;
   int temp;            /* this is used in the find critical path procedures, */
       /* the output of module start nodes, the resetting of the module ends, */
       /* the setting of loop its, the is_reachable and trace_temp_forward    */
       /* procedures and the multicycling of a node.                          */
   int slack;                            /* slack in critical path analysis */
   int node_en;                               /* node enable output signal */
   int loop_its;                                 /* no. of loop iterations */
              /* if loop its is zero then it is not considered in CP calcs */
   char node_type;
   int node_en_net_no;                /* net no given to carry node enable */
/* for a CALL cell the token net no is node_en_net_no + 1 and the activate +2 */
   int in_module;
   struct instruction *inst_list;         /* list of insts in the IGR node */
   struct IO_arcs *in_arc_list;
   struct IO_arcs *out_arc_list;
   struct IGR_node *next;
   int call_collect_n;                         /* call node n or collect n */
   struct cell_data *control_cell;
   };
```

```
/* Instruction list node used in IGR node.                                  */

struct instruction {
   int inst_no;                  /* neg No. indicates dummy instruction to No */
   int inst_type;
   int group_no;                         /* group inst is in within an IGR node */
   int delay;                                       /* delay of this instruction */
   int end_time;     /* this is set to the time inst will end within the node */
   int chain_no;                           /* AEAP position in dependency graph */
   int prob_exec;     /* this indicates the probability of execution as % age */
   struct impl_links *impl_list;
   struct mutual_links *mutual_list;
   struct condition_list *inst_cond;
   int const_ip1;
   union {
      int constant;    /* used for constants, memwrite I1(var no), module No. */
      struct variable_node *var;          /* used for var or switchon operand */
      } input1;
   int const_ip2;
   union {
      int constant;/* used for constants, end count, memwrite I2 add if const */
      struct variable_node *var;     /* used for vars, memwrite I2 add if var */
      struct inst_IO_list *in_list;     /* I/Ps for program, module, moduleap */
      } input2;
   union {
      int count_eq_sig;              /* out sig No for count = end count in2 */
      struct comp_list *sw_comp_list;                /* switch comparison list */
      struct inst_IO_list *out_list;   /* O/P destination list or memwrite IP */
      } output;
   struct inst_depend_list *pre_insts;             /* this inst depends on these */
   struct inst_depend_list *succ_insts;             /* these depend on this inst */
   struct instruction *next;
   };

/* the structure below is an arc that connects instructions within each IGR
   node. It is used to maintain the order of instructions based on their
   dependencies. The resulting graph will represent maximum parallelism.     */

struct inst_depend_list {
   struct instruction *pre_inst;                       /* inst at start of arc */
   struct instruction *succ_inst;                      /* inst at end of arc   */
   struct inst_depend_list *next_pre;  /* next inst that succ_inst depends on */
   struct inst_depend_list *next_succ;    /* next inst that pre_inst relies on */
   };

/* list of links that represent mutually exclusive pairs of instructions, note
   that there must also be a corresponding link from inst pointing to the inst
   that holds this link, ie, this is half of a link.                         */

struct mutual_links {
   struct instruction *inst;
   struct mutual_links *next;
   };
```

```
/* IGR combined input and output arcs list, condition list, inst_IO_list and  */
/* comp_list nodes.                                                           */

struct IO_arcs {
   int arc_no;                                   /* unique are no in list arc_no */
   int is_FBA;                          /* indicates if the arc is a feedback arc */
   struct IGR_node *pre_node;                              /* node arc comes from */
   struct IO_arcs *next_in;                            /* next arc in an input list */
   struct IGR_node *succ_node;                                /* node arc goes to */
   struct condition_list *cond;                                /* act cond of arc */
   struct IO_arcs *next_out;                            /* next arc in output list */
   struct IO_arcs *next;                                 /* next arc in arc_list */
   };

/* The signal no in a condition will always be positive, however a reference
   to it may be for an inverted signal, ie, -sig_no. For these sigs an inverter
   is assumed to be available. In many cases the inverter output is entered
   into the condition list as a seperate signal, eg, s333 = /s3. Therefore
   a reference to /s3 could be made as /s3 or s333.                          */

struct condition_list {
   struct equation_node *cond;
   int signal_no,flag;           /* flag is used in within adjust_cont_sigs() */
   int net_no;                    /* the net number given when numbering nets */
   struct condition_list *next;
   };

struct equation_node {                             /* tree format as in WAG */
   int value;         /* Value represents fn or var if leaf node, -ve for inv */
   int flag;                          /* tested flag used in comp_equations() */
   struct equation_node *parent,*next,*child;
   };

struct inst_IO_list {
   int const_IO;
   union {
      int constant;
      struct variable_node *var;
      } IO;
   struct inst_IO_list *next;
   };

struct comp_list {
   int const_value;                 /* const_value set to -1 for default sig */
   int signal_no;
   struct comp_list *next;
   };

/* structure used in variable list, structure depends on variable type.      */

struct variable_node {
   int var_no;
   int var_type;
   int lo_bit, hi_bit;
```

```
    char *name;                  /* all vars have names after tidy_lists() porc */
    int in_module_no;
    struct DP_node *hardware;
    union {
       struct {
          int lo_index,hi_index;
          int *data;
          } memory;
       struct {
          int hi_bit,lo_bit;
          struct variable_node *parent;
          } alias;
       } type;
    struct variable_node *next;
    };

/* Module list points to the start and end of programs and modules.         */

struct module_node {
   int module_no;                         /* process no of module instruction */
   char *name;
   struct instruction *header;                             /* IO instruction */
   struct IGR_node *start,*last;                   /* last is node before end */
   struct end_list *ends;
   struct module_node *parent,*next;
   int CP_length;          /* if CP_length = 0 then it has not been calculated */
   int optim_order;                /* Gives the order for optimising modules */
   struct call_list *called_bys;    /* list of modules which call this module */
   int CP_calc_type;             /* CP calc technique to use, between or total */
   };

struct call_list {                          /* list of dependents to module */
   struct module_node *called_by;
   struct call_list *next;
   };

struct end_list {                             /* list of end nodes to module */
   struct IGR_node *end;
   struct end_list *next;
   };

/* Label list indicates which label points to which set of processes.        */

struct label_node {
   int label_no;
   struct to *process;
   struct label_node *next;
   };

struct to {
   int is_label;
   int number;
   struct to *next;
   };
```

```
/* Data path list node depends on type and points to IO data path nodes.      */

struct DP_node {
   int node_no;
   char node_type;
   struct cell_data *DP_cell;    /* this has replaced cell_type */
   int n_bits,lo_bit,out_bits,temp;/* temp stores var expected in include reg */
   int area,twos_comp,power;
   struct impl_links *impl_list;
   int max_addresses;            /* max no of addresses = hi_index for memory */
   struct equation_node *boolean_eq;
   struct control_sig *control_sigs;
   struct net_data *input_list;
   struct net_data *output_list;
   struct DP_node *next;
   };

/* list of pointers to instructions implemented using a data path node        */

struct impl_links {                        /* hardware to instruction links */
   struct DP_node *impl_by;                                     /* hardware */
   struct instruction *impl_of;                              /* instruction */
   struct impl_links *next_impl_of,*next_impl_by;
   };

/* The data path net data and control inputs are associated with the          */
/* instructions activating them. Conditions are not set until after           */
/* optimisation as they are dependant on the node enable signals.             */

struct control_sig {
   int pin_type;                      /* pin that control signal connects to */
   int act_inst_no;                    /* if <1 and signal=null cond is true */
   int var_no;                         /* variable being affected by control */
   int active_no;                 /* variable active no associated with var_no */
   struct condition_list *signal;      /* else if signal=null cond on inst   */
   int range_hi_bit,range_lo_bit; /* not used for mem rd/wr, cnt/shft, select */
   int select_fn;        /* used for ALUs to select a particular function/inst */
   int delete;                                      /* see delete in net_data */
   struct control_sig *next;
   };

struct net_data {
   int net_no;         /* initially = signal no therefore not unique to net! */
   int flag;         /* used in adjust_cont_sigs() and add_inst_group() only */
   int delete;      /* used to indicate if a net may be deleted after optim,
                       eg, for inputs to bypassed registers. NB. the load
                       corresponding to this net will not be labeled, so we
                       must explicitly find and delete it.                   */
   int in_hi_bit,in_lo_bit;       /* indicates connecting bit range for input */
/* in_??_bit is the bit range of the output of the module that the input
                                               of the net that connects to. */
        /* or for consts the bit range if it was set or clear on a register */
   int in_type;                                            /* input net type */
   int out_pin;       /* output pin on DP unit that input of net connects to */
   union {                                  /* input to net, output from DP node */
      struct DP_node *start_node;
      int constant;
      int cont_sig;
      } in;
   int wr_act_inst_no;                                     /* inst writing to net */
   int wr_var_no;               /* variable number writing to net (const=-1) */
   int wr_active_no;                  /* active no associated with wr_var_no */
   int in_pin;          /* input pin on DP unit that output of net connects to */
   int out_hi_bit,out_lo_bit;    /* indicates connecting bit range for output */
   int out_type;                 /* that is output from net, input to DP node */
   union {
      struct DP_node *end_node;
      int cont_sig;
      } out;
   int act_inst_no;       /* inst reading from net, if zero then permanent I/P */
   int rd_var_no;                          /* variable number reading from net */
   int rd_active_no;                  /* active no associated with rd_var_no */
   struct condition_list *act_cond;    /* is not set until gen_control_sigs() */
  /* only one act_cond as a conds to read and write for a net is not required */
   struct net_data *next_out_net,*next_in_net,*next;
      /* next_in is the next net connecting to an input of the data path unit */
   };

/* Common destination list used in instruction analysis.                       */

struct common_dest_list {
   struct instruction *inst;
   struct common_dest_list *next;
   };

struct IO_var_list {
   int var_no;      /* -Ve var_no indicates memory and extra contention tests */
   struct IO_var_list *next;
   };

struct access_list {            /* list used to indicate register active times */
   int var_no;                  /* wr_var_no for reads and rd_var_no for writes */
   int inst_no;        /* act_inst_no for reads and wr_act_inst_no for writes */
   int clock_no;
   int is_read;
   int active_no;                                  /* sub lifetime active no */
   struct access_list *next;
   };

/* Data used in transformation routines is stored in the structure below.
   In manual operation it is set up by the select transformation procedure
   and in auto mode it is set up by the optimisation alg.
   The fields are used as follows:
      ->trans_type      Transformation selected to use
      ->test_OK         If true it passed tests
      ->clock           period of clock
      ->node1           node for multi-c,  Write node in group T
                        node to ungroup in ungroup_time_t and ungroup_group_t,
                        first node to merge in seq_merge_t.
                        successor node in LT_123_t.
                        fork node for parallel merge in LT_423_t.
    ->node2             Read node in group T
                        second node to merge in seq_merge_t.
                        preceeding node in LT_123_t.
    ->var1              Var asociated with group reg T
                        variable to unshare from reg in unshare_reg_single_t
    ->inst1             Writing inst in group reg T
                        Inst to unshare from unit in unshare_single_t
    ->inst2             Reading inst in group reg T
    ->insts             group of insts to move for grouping and seq_merge_t
    ->time              node time for ungroup_time_t
    ->n                 group no for ungroup_group_t
                        var_no to remove in unshare_reg_single_t
                        number of nodes created by ungroup_time_t, inc present
                        number of nodes with condition cond in parallel merge.
                        value of old clock in ck_change_t.
    ->n_regs            set by the by_pass and include register procedures
                        which increment it for each register change. Used by
                        merge and ungroup estimates, but generated by tests.
    ->new_delay         this is the delay of the resulting node when insts
                        are added to a node. It is used in the group register,
                        fork merge (LT1) and seq merge transform estimations.
                        It is also used in the ungroup_time_t transform, to
                        indicate the max delay for the new nodes to be created.
    ->DP_node1          First DP node for sharing or ALU combination
                        DP unit to apply alternative cell selection on
                        DP node to unshare in all unsharing transforms
    ->DP_node2          Second DP node for sharing or ALU combination
    -> cond             Condition of arc for merge parallel 423.
    ->functions         List of functions implemented by DP nodes for sharing
                        fn_delays struct is used in common with calc_combo_no()
    ->cell              Combined cell to implement fns in sharing transform
                        New cell for alternative cell selection
    ->delta_E           This is the change in energy of the system
    ->priority          This is the priority that delta_E refers to
    ->temp_T            Simulated annealing current temperature
    ->end_T             temperature to end simulation
    ->T_step            Quantity to reduce temp after each temp sim
                        if the upgrade is true then this must be less than
                        1 as it is multiplied with current temp.
    ->max_its,its       The maximum and present no of iterations at temp T
    ->da_file;          Pointer to the design analysis file.
                        Below is for auto select and optimisation only
    ->is_alloc          how many times are these selected, including failures
    ->is_schedule
    ->selected[16]      No of times transform has been selected/estimated
    ->tested_OK[16]     No of selected transforms pass the tests
    ->p_improve[16]     No of times transform has performed improvement
    ->p_degrade[16]     No of times transform has performed degradation
int estimate_analysis[max_no_trans][nets_crit+1][3][2]
                        The array 16 is for each transform type and the array 2
                        for the criterion, the array 3 stores the number of
```

```
                         occurances of the errors on p211, if the estimate was
                         exactly correct then no error is recorded.
                         The errors are recorded as under or over estimates
                         in the last array, ie, if estimate is less than pres
                         then it is an under estimate and if more it is over
                         The first element is over estimates the second under. */


struct transform_data {
   struct cost_fn_ele *cost_fn;
   int trans_type;
   int test_OK;
   int clock;
   struct IGR_node *node1;
   struct IGR_node *node2;
   struct variable_node *var1;
   struct instruction *inst1;
   struct instruction *inst2;
   struct instruction *insts;
   int time,n,new_delay,n_regs;
   struct DP_node *DP_node1,*DP_node2;
   struct condition_list *cond;
   struct fn_delay *functions;
   struct cell_data *cell;
   float delta_E;
   int priority, in_progress;
   float temp_T, end_T, T_step;
   int max_its, its;
   FILE *da_file;
   int selected[max_no_trans];
   int p_improve[max_no_trans];
   int p_degrade[max_no_trans];
   int tested_OK[max_no_trans];
   int is_alloc,is_schedule;
   int estimate_analysis[max_no_trans][nets_crit+1][3][2];
   };

/* Cost function vector is stored as a list of priorities, each consisting of */
/* a list of criteria associated with that priority.                          */

struct cost_fn_ele {
   int priority;
   struct criterion_ele *criteria;
   struct cost_fn_ele *next;
   };

struct criterion_ele {
   int criterion;
   int initial;
   int target;           /* target value (tv): reached according to opt_type  */
   int previous;         /* last value for criterion */
   int present;          /* present value for criterion */
   int estimate;
   struct instruction *start,*end;   /* insts specified for betweem CP calcs */
   struct criterion_ele *next;
   };

struct optim_cost {
   int no,power,area,delay,nets,mog,is_optim,cpu_time;
   int error;      /* indicates % error in reaching target, -ve = not reached */
   struct optim_cost *next;
   };

/* Cell info data structure. Cell data is stored as a linked list with pin    */
/* lists joined to it. The cell info file has cell_no, IPs_com, area,         */
/* inh_delay, and delay_factor for every cell. Following this is the number   */
/* of pin data sections, these consist of pin_type and fields specific to     */
/* that pin type. At present this is IP_cap for inputs only.                  */

struct cell_data {
   int cell_no, n_fns, IPs_com, n_bits, op_bit_type, quantity;
   char *name;
   int reg_set_up;
   struct combo_data *combined_fns;          /* always at least one of these */
   struct pin_data *pins;         /* also collect n penalty in control cells */
   struct cell_data *next;
   struct cell_alt *cell_alts;/* alts for cell. cells that cover all cell fns */
   struct cell_fn *cell_fns;   /* all insts implementable by cell, at least 1 */
   };

struct combo_data {
   int combine_no;
   struct fn_delay *fn_delays;                /* delays for each fn in combo */
   int  area, delay_factor, power;
   struct combo_data *next;
   };

struct fn_delay {     /* this is the inherent delay for a function in a combo */
   int fn_no;         /* one is entered for each inst for this combination    */
   int inh_delay;
   struct fn_delay *next;
   };

struct pin_data {
   int pin_type;
   int IP_cap;                                          /* input capacitance */
   int area_penalty;
   int power_penalty;
   struct pin_data *next;
   };

struct cell_fn {
   int fn_no;              /* first one is LSB when converting to combo no. */
   struct cell_fn *next;
   };
```

```
struct cell_alt {
   int alt_no;
   struct cell_alt *next;
   };

struct techno_units {                    /* see techno reader for info on these */
   char *time_str, *cap_str, *delay_factor_str, *power_str, *area_str;
   int time_off, cap_off, delay_factor_off, power_off, area_off;
   };

/*===============================END OF FILE====================================*/
```

References

1 Goldberg, A V - Hirschhorn, S S - Lieberherr, K J, "Approaches Toward Silicon Compilation.", IEEE Circuits and Devices, May 1985, pp. 29-39.

2 VLSI Design, Staff, "Silicon Compilers. Part 1: Drawing a Blank.", VLSI Design. September 1984.

3 Allen, Jonathan, "Performance-Directed Synthesis of VLSI Systems.", Proc. IEEE, Vol. 78, No. 2. Feburary 1990. pp 336-355.

4 Blackman, Timothy - Fox, Jeffrey - Rosebrugh, C, "The SILC Silicon Compiler: Language and Features.", Proc. 22nd DAC. 1985 IEEE. Paper 17.1, pp. 232-237.

5 Hartley, Richard I - Jasica, Jeffrey R, "Behavioural to Structural Translation in a Bit-Serial Silicon Compiler.", IEEE Trans. on CAD. Vol 7. No 8. August 1988. pp. 877-886.

6 Parker, Alice C, "Automated Synthesis of Digital Systems.", IEEE Design & Test, November 1984. pp. 75-81.

7 Werner, Jerry [editor], "Progress Toward the 'Ideal' Silicon Compiler. Part 1: The Front End.", VLSI Design. September 1983.

8 Werner, Jerry [editor], "Progress Toward the 'Ideal' Silicon Compiler. Part 2: The Layout Problem.", VLSI Design. October 1983.

9 Camposano, Raul, "Synthesis Techniques for Digital Systems Design.", Proc. 22nd DAC, 1985 IEEE. pp. 475-481.

10 Gajski, Daniel D - Dutt, Nikil D - Pangrle, Barry M, "Logic Design and Silicon Compilation for VLSI Design. Silicon Compilation (A Tutorial).", Proc. IEEE 1986 Custom IC Conf. NY. May 1986, pp. 102-110.

11 Southard, Jay R, "MacPitts: An Approach to Silicon Compilation.", IEEE Computer, December 1983, pp. 74-82.

12 Bergmann, Neil, "A Case Study of the F.I.R.S.T Silicon Compiler. ", Third Caltech Conf. VLSI, March 1983, pp. 413-430.

13 Walker, Robert A - Thomas, Donald E, "Behavioural Level Transformations in the CMU-DA System.", Proc. of the 20th DAC, ACM/IEEE, Miami, FL, June 1983.

14 Thomas, Donald E - Blackburn, Robert L - Rajan, J, "Linking the Behavioural and Structural Domains of Representation for Digital Systems Design.", IEEE Trans. CAD, Vol. 6, No. 1, 1987. pp. 103-110.

15 Walker, Robert A - Thomas, Donald E, "Design Representation and Transformation in the System Architect's Workbench.", Proc. Int. Conf. Computer Design (ICCD) 1987. pp. 166-9.

16 Werner, Jerry [editor], "The Silicon Compiler: Panacea, Wishful Thinking, or Old Hat?", VLSI Design, Vol. 3, No. 5, Sept/Oct 1982, pp. 46-52.

17 Hauge, Peter S - Nair, Ravi - Yoffa, Ellen J, "Circuit Placement For Predictable Performance.", Proc. Int. Conf. Computer Design (ICCD) 1987. pp. 88-91.

18 McFarland, Michael C, "On Proving the Correctness of Optimising Transformations in a Digital Design Automation System.", Proc. 18th DAC, IEEE Comp. Soc. DATC, June 1981, pp. 90-97.

19 Haroun, Baher S - Elmasry, Mohamed I, "Architectural Synthesis for DSP Silicon Compilers.", IEEE Trans. on CAD. Vol 8. No 4. April 1989, pp. 431-447.

20 Bergamaschi, Reinaldo A, "The Development of a High Level Synthesis System for Concurrent VLSI Systems.", PhD Thesis. Southampton University. December 1988.

21 Peng, Zebo, "Synthesis of VLSI Systems with the CAMAD Design Aid.", Proc. 23rd DAC, 1986 IEEE. pp. 278-284.

22 Peng, Zebo, "A Formal Approach to the Synthesis of VLSI Systems From Their Behavioural Descriptions.", Proc. 19th Hawaii Int. Conf on Sys. Sci, January 1986. pp. 160-7.

23 Peng, Zebo, "A Formal Methodology for Automated Synthesis of VLSI Systems.", PhD Thesis. Linkoping University, 1987.

24 Parker, Alice C - Mlinar, Mitch - Pizarro, Jorge, "MAHA: A Program for Datapath Synthesis.", Proc. 23rd DAC, Las Vagas, July 1986, pp. 461-466.

25 Paulin, P G - Knight, J P - Girczyc, E F, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis.", Proc. 23rd DAC, July 1986, pp. 263-270.

26 Tseng, Chia-Jeng - Siewiorek, Daniel P, "Facet: A Procedure for the Automated Synthesis of Digital Systems.", Proc. 20th DAC, 1983 IEEE. pp. 490-496.

27 Pangrl, Barry M - Gajski, Daniel D, "State Synthesis and Connectivity Binding for Microarchitecture Compilation.", IEEE 1986. pp. 210-213.

28 Zimmermann, G, "The MIMOLA Design System: A Computer Aided Digital Processor Design Method.", Proc. 16th Design Automation Conf., June 1979, pp. 53-58.

29 Girezyc, E F - Knight, J P, "An ADA to Standard Cell Hardware Compiler Based on Graph Grammers and Scheduling.", Proc. ICCD, 1984. October 1984.

30 Knapp, David W - Parker, Alice C, "The ADAM Design Planning Engine.", IEEE Trans. on CAD Vol. 10 No. 7, July 1991. pp. 829-846.

31 McFarland, Michael C, "Reevaluating the Design Space for Register-Transfer Hardware Synthesis.", Proc. Int. Conf. Computer Design (ICCD) 1987. pp. 262-265.

32 Park, Nohbyung - Parker, Alice C, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications.", IEEE Trans. on CAD. Vol 7. No 3. March 1988. pp. 356-370.

33 Silvar Lisco, "CAL-MP 10 - General Overview.", Document No: M014-3, March 1984.

34 Jain, Rajiv - Mlinar, Mitchell J - Parker, Alice, "Area-Time Model for Synthesis of Non-Pipelined Designs.", Proc. Int. Conf. Computer Design (ICCD) 1988. pp. 48-51.

35 Tseng, Chia Jeng - Siewiorek, Daniel P, "Emerald: A Bus Style Designer.", Proc, 21st Design Automation Conf., June 1984.

36 Johannson, D L - McElvain, K - Tsubota, S K, "Intelligent Compilation.", VLSI Syst. Design. Vol. 8, No. 4, pp. 40-46. April 1987.

37 Johannsen, D, "Bristle Blocks: A Silicon Compiler.", Proc. 16th DAC June 1979, pp. 310-313.

38 Siskind, J M - Southard, J R - Crouch, K W, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions.", Proc. Conf. Advanced Reseach in VLSI, January 1982, pp. 28-40.

39 Claesen, L - Catthoor, F - Goossens, G - et al, "Automatic Synthsis of Signal Processing Benchmark using the CATHEDRAL Silicon Compilers.", Draft version 22/1/88, Proc. IEEE 1988 CICC.

40 Jamier, R - Jerraya, A A, "APOLLON, A Data-Path Silicon Compiler.", IEEE Circuits and Devices Magazine, May 1985.

41 Choi, Y H, "Synthesis of pipelined data paths.", CAD Butterworth. Vol. 24, No. 1, January 1992. pp. 36-40.

42 Devadas, Srinivas - Newton, A. Richard, "Algorithms for Hardware Allocation in Data Path Synthesis.", IEEE Trans. on CAD. Vol. 8, No. 7. July 1989. pp. 768-81.

43 Hitchcock, Charles Y - Thomas, Donald E, "A Method of Automatic Data Path Synthesis.", Proc. 20th DAC. 1983 IEEE. pp. 484-489.

44 Hafer, Louis J - Parker, Alice C, "Automated Synthesis of Digital Hardware.", IEEE Trans. on Computers, Vol. 31, No. 2, February 1982. p93.

45 Parker, Alice C - Hafer, Lou, "The Application of a Hardware Description Language for Design Automation.", January 1978, pp. 349-355.

46 Tseng, Chia Jeng - Siewiorek, Daniel P, "Automated Synthesis of Data Paths in Digital Systems.", IEEE Trans. CAD, Vol. 5, pp. 379-395, July 1986.

47 Allerton, D J - Batt, D A - Currie, A J, "Second Progress Report: Silicon Compiler Project.", University of Southampton, dept. of Electronics. April 1983.

48 Jong, Ivan C. C., "SCHOLAR User Manual (v1.0).", Southampton University, Dept. Electronics and Comp. Sci. July 1988.

49 Camposano, Raul - van Eijndhoven, J T J, "Combined Synthesis of Control Logic And Data Path.", Proc. Int Conf Computer Design (ICCD) 1987. pp.327-329.

50 Bendas, J B, "Design Through Transformation.", Proc. 20th DAC, Miami, FL. June 1983.

51 Hong, Youn Sik - Park, Kyu Ho - Kim, Myunghwan, "Automatic Synthesis of Data Paths based on the Path-Search Algorithm.", Proc. Int Conf Computer Design (ICCD) 1987. pp. 270-273.

52 Nagle, Andrew W - Cloutier, Richard - Parker, Alice C, "Synthesis of Hardware for the Control of Digital Systems.", Trans. CAD of ICs and Systems, No. 4, October 82, pp. 201-12.

53 Nagle, Andrew W - Parker, Alice C, "Algorithms for Multiple-Criterion Design of Micro- Programmed Control Hardware.", Proc. 18th DAC, (Nashville, TN), June 1981. pp. 486-493.

54 Girczyc, E F - Buhr, R J A - Knight, J P, "Applicability of a Subset of Ada as an Algorithmic Hardware Design Language for Graph-Based Hardware Compilation.", IEEE Trans on CAD, Vol. CAD-4, No. 2, April 1985.

55 Kowalski, T J - Thomas, D E, "The VLSI Design Automation Assistant: What's in a Knowledge Base.", DAC, 1985, pp. 252-258.

56 Kowalski, T J - Geiger, D J - Wolf, W - Fichtner W, "The VLSI Design Automation Assistant: From Algorithms to Silicon.", IEEE Design & Test, August 1985, pp. 33-43.

57 Camposano, Raul - Rosenstiel, Wolfgang, "Synthesizing Circuits From Behavioural Descriptions.", IEEE Trans. on CAD. Vol 8. No 2. February 1989. pp. 171-180.

58 Hienrich, Kramer - Rosenstiel, Wolfgang, "System Synthesis using Behavioural Descriptions.", IEEE Proc. of EDAC, 25-28 February 1990, pp. 277-282.

59 Raj, Vijay K, "Another Automated Data Path Designer.", IEEE 1986.

60 Paulin, P G - Knight, J P, "Force-Directed Scheduling in Automated Data Path Synthesis.", Proc. 24th ACM/IEEE DAC 1987, pp. 195-202.

61 Paulin, P G - Knight, J P, "Extended Design-Space Exploration in Automatic Data Path Synthesis.", Proc. Canadian Conf. on VLSI, October. 1986, pp. 221-226.

62 Camposano, Raul - Bergamaschi, Reinaldo A, "Redesign Using State Splitting.", EDAC 1990, Glasgow, March. IBM, Research Report.

63 Camposano, Raul, "Path-Based Scheduling for Synthesis.", IEEE Trans on CAD, Vol 10, No 7, January 1991. pp. 85-94.

64 Balakrishnan, M - Majumdar, A - Banerji, D - et al, "Allocation of Multiport Memories in Data Path Synthesis.", IEEE Trans. on CAD. Vol 7. no 4. April 1988. pp. 536-540.

65 Tseng, Chia Jeng - Siewiorek, Daniel P, "The Modeling and Synthesis of Bus Systems.", Proc. 18th DAC, June 1981 IEEE. pp. 471-478.

66 Rajan, Jayanth V - Thomas, Donald E, "Synthesis By Delayed Binding Of Decisions.", 22nd Design Automation Conf. IEEE. 1985. pp. 367-373.

67 Lagnese, E D - Thomas, D E, "Archetectural Partitioning for System Level Synthesis of Intergrated Circuits", IEEE Trans on CAD, Vol. 10, No 7, July 1991. pp. 847-860.

68 Bushnell, M L - Director, S W, "ULYSSES: An expert-system based VLSI design environment.", in Proc. ISCAS 85, 1985.

69 Brewer, Forrest - Gajski, Daniel, "Chippe: A System for Constraint Driven Behavioral Synthesis.", IEEE Trans. on CAD. Vol 9. No 7. July 1990. pp. 681-695.

70 Safri, A - Zavidovique, B, "Towards a Global Solution to High Level Synthesis Problems.", IEEE Proc. of EDAC, 25-28 February. 1990, pp. 283-288.

71 Morison, J D - Peeling, N E - Thorp, T L, "ELLA: A Hardware Description Language.", IEEE conf. on Circuits and Computers, September 1982.

72 Morison, J D - Peeling, N E - Whiting, E V, "Sequential Programming Extensions to ELLA, with Automatic Transformation to Structure.", Proc. ICCD 1987, pp. 571-576, Rye Brook, NY, October 1987.

73 Baker, Keith R, "The ELLA to ICODE Interface.", Southampton University, Dept. Electronics & Comp. Sci. June 1990.

74 Younger, D H, "Minimum Feedback Arc Sets for a Directed Graph.", IEEE Trans. Circuit Theory. June 1963. pp. 238-245.

75 Lempel, A - Cederbaum, I, "Minimum Feedback Arc and Vertex Sets of a Directed Graph.", IEEE Trans. Circuit Theory, December 1966. pp. 399-403.

76 Yau, S S, "Generation of all Hamiltonian Circuits, Paths and Centers of a Graph, and Related Problems.", IEEE Trans. Circuit Theory, March 1967. pp 79-80.

77 Divieti, L - Grasselli, A, "On the Determination of Minimum Feedback Arc and Vertex Sets.", IEEE Trans. Circuit Theory, March 1968. pp. 87-89.

78 Ignizio, James P, "Goal Programming and Extensions.", Lexington Books. 1976.

79 McFarland, Michael C - Parker, Alice C - Camposano, Raul, "Tutorial on High-Level Synthesis.", Proc. IEEE 25th DA Conf., 1988, pp. 330-336.

80 Koelmans, A M - Burns, F P - Kinniment, D J, "Use of a Theorem Prover for Transformational Synthesis.", Computing and Control Division, 21st January 1991. No: 1991/014.

81 Nahar, Surendra - Sahni, Sartaj - Shragowitz, E, "Simulated Annealing and Combinatorial Optimization.", IEEE, 23rd DAC 1986. Paper 16.1, p. 293.

82 Carre, Bernard, "Graphs and Networks.", Oxford University Press 1979.

83 Ullman, J D - Aho, A V - Sethi, R, "Compilers: Principles, Techniques and Tools.", Addison-Wesley, Mass, 1986.

84 Moder, Joseph J - Phillips, Cecil R - Davis, E W, "Project Management with CPM, PERT and Precedence Diagramming. 3rd edition.", Van Nostrand Reinhold Company.

85 Rutenbar, Rob A, "Simulated Annealing Algorithms: An Overview.", IEEE Circuits and Devices Mag. January 1989. pp. 19-26.

86 Kirkpatrick, Scott - Gelatt Jr., C D - Vecchi, M P, "Optimization by Simulated Annealing.", Science. 13 May 1983, Volume 220, No. 4598. pp. 671-680.

87 Kirkpatrick, Scott, "Optimization by Simulated Annealing: Quantitive Studies.", Jrnl of Stat'cal Phys, Vol 34, Nos 5/6, 1984. pp. 975-986.

88 Metropolis, N - Rosenbluth, A - Teller A & E, "Equation of State Calculations by Fast Computing Machines.", Jr. Chem. Phys., Vol 21. p. 1087. 1953.

89 Nahar, S - Sahni, S - Shragowitz, E, "Experiments with Simulated Annealing.", 22nd Design Automation Conference, 1985, pp. 748-752.

90 Microelectronics Centre of Northern California, "High-Level Synthesis Workshop Benchmarks.", 1989, 1991.

91 Baker, Keith R, "The MOODS Synthesis System - User Manual V2.0.", Southampton University, Dept. Electronics & Comp. Sci. October 1992.

92 Leive, G - Thomas, D, "A Technology Relative Logic Synthesis and Module Selection System.", Proc. 18th DAC. IEEE Comp Soc DATC, June 1981, pp. 479-85.

93 Knapp, David W, "Datapath Optimization Using Feedback.", IEEE Proc. of EDAC, 12-15 March 1991, pp. 129-134.

94 Hands, J P, "What is VHDL?", CAD, Butterworth. Vol 22. No 4. May 1990. pp. 246-9.