UNIVERSITY OF SOUTHAMPTON

A Behavioural VHDL Synthesis System using Data Path Optimisation

by

Alan Christopher Williams

A thesis submitted for the degree of
Doctor of Philosophy.

Department of Electronics and Computer Science,
University of Southampton

October, 1997

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

A Behavioural VHDL Synthesis System using Data Path Optimisation

by Alan Christopher Williams

MOODS (Multiple Objective Optimisation in Data and control path synthesis) is a synthesis system which provides the ability to automatically optimise a design from a behavioural to a structural VHDL description. This thesis details two sets of enhancements made to the original system to improve the overall quality of the final hardware implementations obtained, and expand the range of the accepted VHDL subset.

Whereas the original MOODS considered each functional unit in the target module library to be a purely combinational logic block, the 'expanded modules' developed for this project provide a means of implementing sequential multi-cycle modules. These modules are defined as technology-independent templates, which are inline expanded into the internal design structure during synthesis. This enables inter-module optimisation to occur at the sub-module level, thus affording greater opportunities for unit sharing and module binding. The templates also facilitate the development of specialised interface modules. These enable the use of fixed timing I/O protocols for external interfacing, while maintaining maximum scheduling flexibility within the body of the behaviour.

The second set of enhancements includes an improved implementation of behavioural VHDL as input to the system. This expands the previously limited subset to include such elements as signals, wait statements, concurrent processes, and functions and procedures. These are implemented according to the IEEE standard thereby preserving the computational effects of the VHDL simulation model.

The final section of work involves the development and construction of an FPGA-based real-time audio-band spectrum analyser, synthesised within the MOODS environment. This design process provides valuable insights into the strengths and weaknesses of both MOODS and behavioural synthesis in general, serving as a firm foundation to guide future development of the system.

Contents

List of Figures

Acknowledgements

I would like to acknowledge the help and support of the following people:

My supervisor, Andy Brown, for his constant encouragement throughout the last four years, and especially for putting up with me over the long haul of the past couple of months. His prompt, and very positive, proof-reading of this thesis has made the final stages of the write-up almost enjoyable, and his endless phone calls and questions about computers have been a constant source of entertainment.

My wife, Liz, for putting up with numerous late nights and lost weekends, judiciously keeping out of my way, and generally making it all bearable.

Keith Baker, for creating MOODS in the first place, explaining its intricacies and peculiarities, and for helping me while away the wee small hours with the odd blast of a rocket launcher.

Andrew Currie, for supervising the early stages of this work, and Mark Zwolinski for being a generally helpful guy, and a fertile source of ideas and information.

Finally, I would like to say a special thank you to Zaher Baidas, and Marc Castells, with whom I spent many a frustrating hour staring at a computer screen, attempting to coax some life out of an FPGA.

Chapter 1

Introduction

In recent years, advances in VLSI technology have made the design of integrated circuits an ever more complex task: in 1982 a state of the art 80286 microprocessor contained a mere 100,000 transistors; today, the latest Pentium II chips pack in 5.5 million transistors, with projections indicating that by the turn of the century devices containing around 100 million will be possible. This trend has been accompanied by a dramatic decrease in both product life cycle and time to market throughout the electronics industry. The combination of these factors has lead to the development of a wide range of CAD systems designed to simplify the entire IC development cycle, ranging from basic layout tools to complete system simulators and synthesis environments. These have become an essential part of the design cycle as technology has progressed.

Since the early 1980s, much research has focused on the development of silicon compilers [1, 2], which are able to synthesise a circuit implementation from a design description. Initial efforts were directed towards synthesis from a structural description however as device complexity increased, higher level design tools and languages were developed. This abstraction of the design mirrors the way in which software development has progressed from small programs written in low-level assembly language, to the large applications and object oriented programming tools of today.

In addition to a shortened design time, synthesis systems offer a number of further advantages over conventional methods. By investigating various optimisation strategies, the designer can explore a large range of different physical implementations, trading off such factors as area, speed and power consumption. The abstractness of the description also facilitates the investigation of different technologies and target platforms, such as the Application Specific IC (ASIC) versus the Field Programmable Gate Array (FPGA). Furthermore, automation of the design process considerably reduces the chance of errors

being introduced, and also facilitates the straightforward inclusion of test structures in the final implementation.

Behavioural synthesis transforms a circuit description from the algorithmic and data structure level, the user describing what the circuit will do rather than how it should be implemented. This leaves the refinement and optimisation of the design to the synthesiser, thus allowing a system designer to concentrate on the overall functionality and reducing the need for a detailed knowledge of VLSI design.

MOODS (Multiple Objective Optimisation in Data and control path synthesis) [3, 4, 5] is a behavioural synthesis system which transforms a VHDL (Very High Speed IC Hardware Description Language) [6, 7] description into a structural netlist suitable for processing by a low-level logic optimisation and placement/routing tool. It is an "intelligent" silicon compiler that allows automatic trade-offs to be made between different aspects of a design, using technology-dependent characterisation data (such as area and delay) fed up from a module library. Thus, MOODS is able to make high-level design decisions based upon an accurate low-level model of the target technology.

1.1 Objectives and Achievements

This thesis details further research into various aspects of the original MOODS synthesis system [3]. These cover three principal objectives:

1. Provide the existing system with the ability to exploit sequential low-level operators, normally mapped onto combinational library cells, in order to expand the range of synthesised implementations attainable. These *expanded modules* are primarily aimed at reducing circuit area at the cost of a degradation in speed.

2. Enlarge the set of VHDL constructs supported by the system to enable the synthesis of complex designs exploiting the full power of concurrent, communicating processes.

3. Obtain experience of the design process using behavioural synthesis, and MOODS in particular, through the implementation of a complete synthesised system from initial conception through to final hardware construction.

These objectives have been achieved through two sets of enhancements made to MOODS:

1. An innovative approach is taken to the use of sequential operators through the development of a *module expansion* capability whereby the modules themselves are described as clocked sequential sub-circuits, which are expanded in the body of the main design thus allowing optimisation both external to and within their structure. The resulting synthesised systems are generally smaller and also, in some situations, faster.

2. Substantial improvements to the VHDL front-end compiler enable the system to synthesise designs featuring a much more varied and flexible subset of the VHDL language.

Both of the above sets of improvements are exploited in the design of a real-time audio band spectrum analyser, providing valuable insights into design techniques suitable for behavioural synthesis, and a physical demonstrator of the capabilities of the MOODS system.

The thesis is split into nine chapters. Chapter 2 provides a general introduction to behavioural synthesis, looking at the typical concepts and algorithms used in the major research systems currently being developed. This is followed in Chapter 3 by a detailed discussion of some related research, influential in the development of both the expanded modules, and the VHDL enhancements.

Chapter 4 describes the basic MOODS synthesis system, discussing its overall operation, together with a more detailed examination of the core synthesis sub-tasks. This provides a basis for a detailed examination, in the later chapters, of the low-level effects the various enhancements have on the optimisation process.

The design and implementation of expanded modules is described in Chapter 5, along with several additional improvements required to make the low-level structure of MOODS more flexible. Their operation from a user perspective is also considered, along with a detailed analysis of the effect of many different expansion types on a number of substantial benchmark designs. Chapter 6 follows on from the earlier discussions in Chapter 3 concerning the implementation of VHDL for behavioural synthesis. In this case, the conversion of VHDL to intermediate code (ICODE), used for direct input into MOODS, is examined in relation to the highly restricted subset originally supported. The

effects of faithfully implementing certain elements of the VHDL simulation model are also considered, in terms of their area and delay overhead for typical synthesised designs.

The improvements discussed in the previous chapters are put to use in the design, development and hardware construction of a real-time spectrum analyser discussed, in detail, in Chapter 7. This provides an in-depth look at the typical problems encountered during the high-level synthesis of a complex system, and develops a number of design techniques of general use. As a direct consequence of this design process, the need for a number of enhancements to the system becomes apparent. These are briefly discussed in Chapter 8, forming a prelude to the future development of a second-generation synthesis system. Finally, Chapter 9 concludes by summarising the key points covered in the whole thesis.

A number of appendices are also included providing additional information on various aspects of the work carried out. In particular, Appendix A contains two papers detailing work, not reported elsewhere in the thesis, on the possibilities for synthesising circuits with an *on-line testability* feature. This exploits the time periods during which low-level modules are idle to provide a real-time indication of the health of a system under typical operating conditions.

Chapter 2

An Overview of Behavioural Synthesis

Behavioural, or high-level synthesis [8, 9, 10, 11] is the process of transforming an abstract specification of the *behaviour* of a system, into an equivalent *structural* description that satisfies a set of user constraints and goals on factors such as final delay and total area. This structure may then be processed by logic-level optimisation and layout tools to obtain a final *physical* implementation of the system, based on a particular target technology. Behavioural, structural and physical are generally identified as the three *domains* of hardware descriptions [12]. For the purposes of high-level synthesis: *behavioural* specifies the abstract operation of the design in terms of the algorithmic relationship between its inputs and outputs; *structural* describes the topology of the design as an interconnection of low-level components in the form of a netlist; and *physical* defines the construction of final hardware in a form suitable for the target technology, eg. geometric patterns for custom silicon, or cell placement and routing in an FPGA.

2.1 Behavioural Synthesis

The process of transforming behaviour, described in some sequential language such as VHDL or C, is illustrated in Figure 2.1, which shows the flow of data in a generic behavioural synthesis system. The behavioural description is initially compiled into an internal data structure holding a complete multi-level representation of the design through all stages of the synthesis process. A series of synthesis sub-tasks are then applied, which together transform the behavioural input into a suitable structure, according to various objectives for circuit performance, such as minimum delay or area. Once a final implementation has been achieved, a structural description is output in the form of a netlist based on components in a low-level cell/module library. This may then be simulated, in conjunction with the initial behaviour, to verify the correctness of the design, and obtain

detailed timing information. The final task is to process the netlist using low-level logic synthesis and layout tools, to realise a physical hardware implementation.

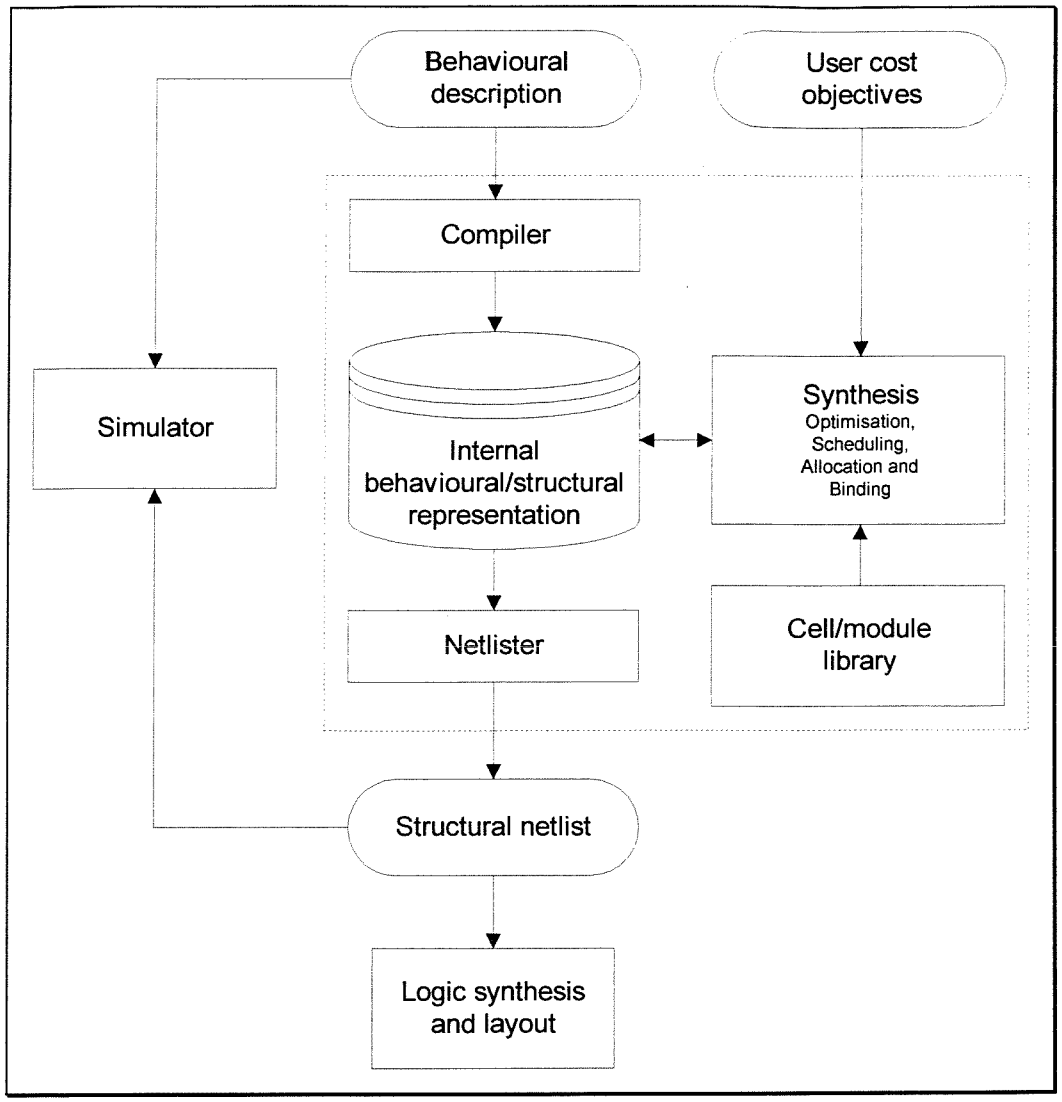


Figure 2.1 Generic behavioural synthesis system dataflow

2.2 Compilation and Internal Representation

The first synthesis task to be performed is the compilation of the behavioural description into an internal representation to which synthesis operations may be applied. This is essentially a direct translation of the input into a graph-based data structure describing the flow of control and data through the system, which will eventually be implemented as a controller state-machine and data path in the final structural description. Some language-level optimisations may be performed at this stage such as dead code elimination or expression simplification [9, 13, 14], however most systems perform these optimisations after compilation during the synthesis process itself.

In order to fully describe the design throughout the synthesis process, the internal data structure must be capable of representing both behaviour and structure in a consistent manner suitable for the targeted application. For example, a simple dataflow graph (DFG) [15] representation is most suited to behavioural descriptions containing straight-line code [16] (ie. no loops or conditional branches), where a series of operations is continuously applied to an infinite data stream. Such a model would be suitable for certain digital signal processing applications like digital filtering.

On the other hand, many applications contain a significant amount of control logic (loops, if-then-else constructs), and thus require a more comprehensive representation describing both data, and control flow. Generally, systems either take an integrated approach, modelling both data and control in a single hierarchical structure, such as in the *Control-Dataflow Graph* (CDFG) [10, 17, 18], or separate them into individual, but interrelated, control and data flow graphs as in the *Extended Timed Petri-net* (ETPN) [19, 20] and the *Value-Trace* (VT) [9, 21, 22]. Extra structural and scheduling data may also be maintained, such as in the *Value Trace with Control Steps* (VT/CST) [23], which links dataflow in a VT description, to a scheduled control steps (CST) structure, via a set of *tags*.

By way of an example, Figure 2.2 shows a fragment of VHDL code for calculating either root of a quadratic equation. The associated CDFG and ETPN representations immediately after compilation are shown in Figure 2.3 and Figure 2.4 respectively.

```
  ....
t1 := 4 * a * c;
t2 := b * b;
s := sqrt(t2 - t1);
IF sel = 1 THEN
  r := -b + s;
ELSE
  r := -b - s;
END IF;
  ....
```

Figure 2.2 VHDL quadratic equation solver

The CDFG is a hierarchical structure, which at the top level, describes the control flow through the system as a directed graph, where each node comprises a separate dataflow graph (DFG) representing a block of assignments or conditional statements. Thus, Figure 2.3 splits up into three DFG blocks: one for the first three sequential statements, plus two more for each of the conditional branches. The main goal when constructing the CDFG is to represent as much behaviour as possible in data flow, thus allowing the scheduler maximum freedom during control state assignment (see section 2.4).

In contrast, Figure 2.4 shows the corresponding ETPN structures generated by CAMAD [24]. This models the data path as a directed graph, where vertices represent individual functional (operators) and storage (variables) units. Data flows via conditional arcs, with connections only being made if the arc is activated by a control signal (S_n signals generated from the control part). The control flow on the other hand, is described by the passage of *tokens* through a Petri-net. Here, each vertex represents a control state, which is activated when it receives a token, thereby activating the associated data path arcs via its S_n signal. State transitions occur when a token is passed on, via conditional arcs, conditions being generated by the data path. Both parts are derived directly from the behavioural code, with a data path vertex for each operation and variable, and a control state for each VHDL statement. Hence in Figure 2.4, each line has its own control state where the conditional block (*if sel = 1*) is modelled by states S_5 and S_6, selection being based upon the condition C_1, generated from the data path comparator block.

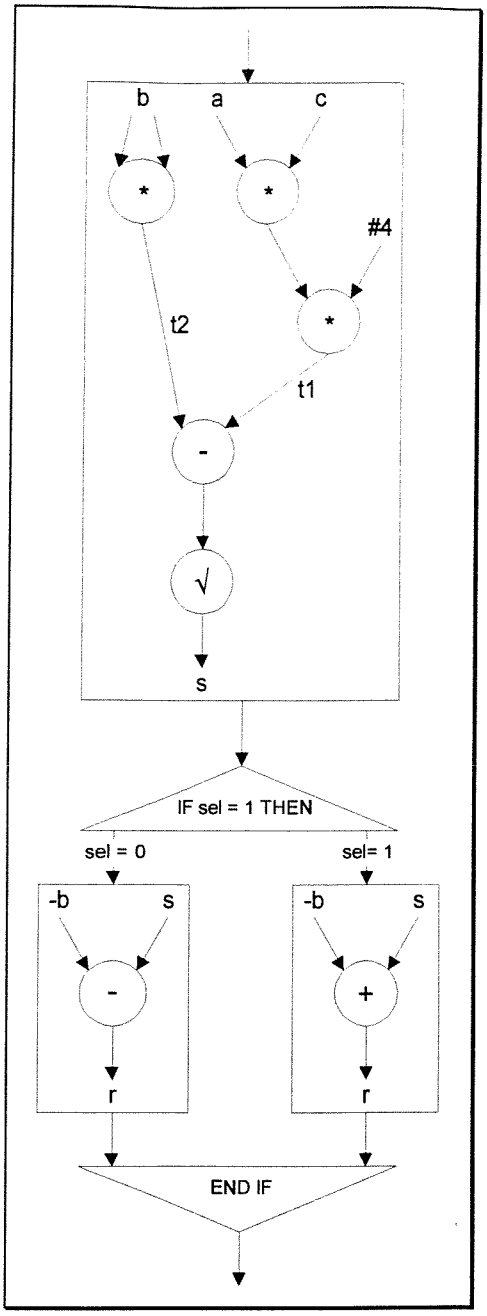


Figure 2.3 CDFG for the quadratic equation solver

The representation chosen for a particular system is largely governed by the approach taken towards the various synthesis sub-tasks. The CDFG is typical of systems featuring *constructive* scheduling algorithms (see section 2.4.1), such as Chippe [17], where the synthesised structure is built up from scratch over several stages. The principal aim of the CDFG therefore, is to reduce the design as much as possible, into a purely data flow description, allowing synthesis maximum freedom to optimise the scheduling of operations to control states. It also attempts to improve register use, not assuming any partitioning of the storage requirements and thus enabling the allocation algorithm (mapping of source variables to data path storage units) as much freedom as possible. Some of the CDFGs drawbacks are

its lack of any explicit dependency information between sequential control blocks (thus independent blocks cannot be parallelised), and problems associated with memory dependencies [18]. These deficiencies may be countered by adding extra information into the data structures.

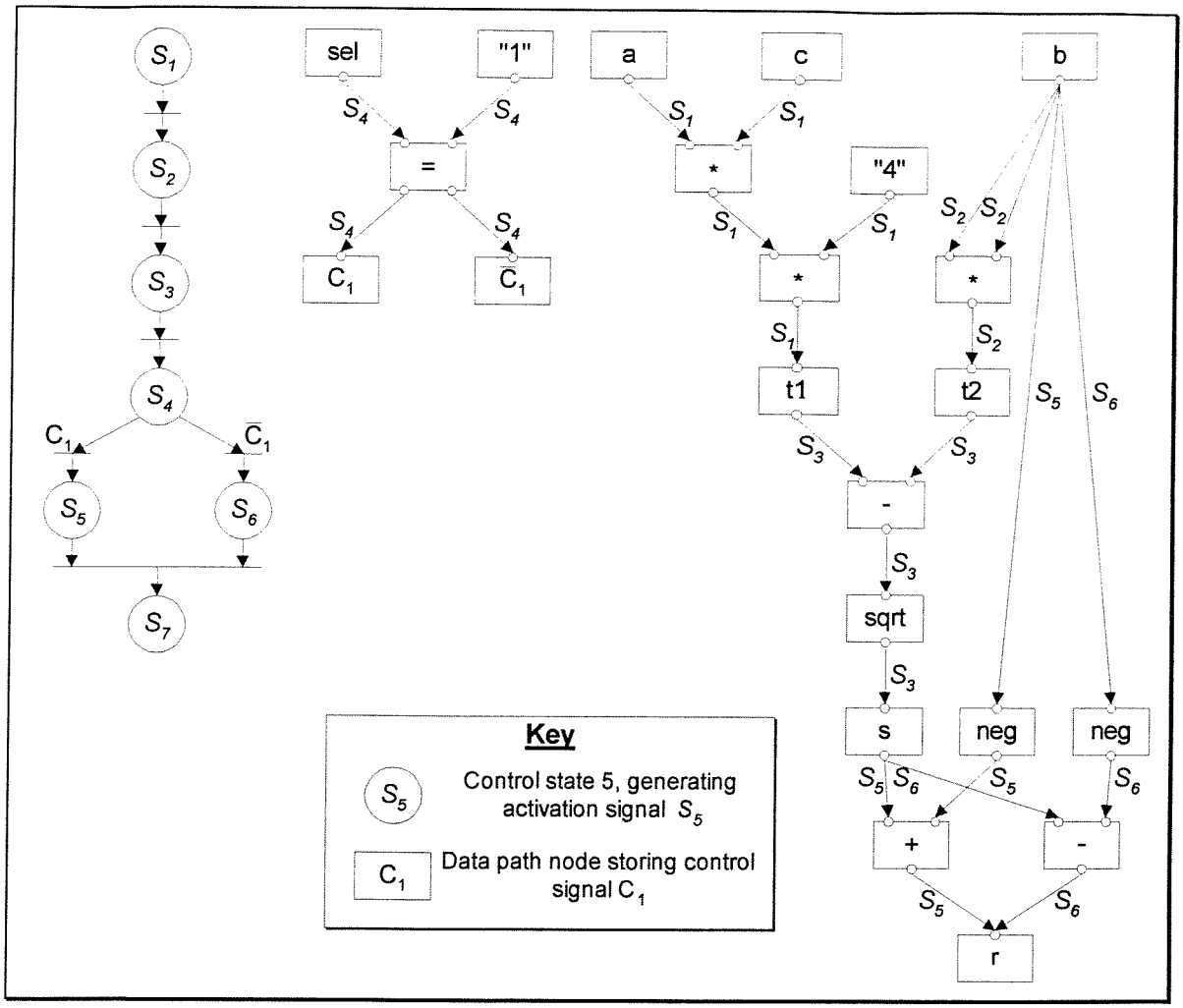


Figure 2.4 ETPN for the quadratic equation solver

The ETPN model, on the other hand, is intended for a system that applies *iterative improvements* to an initial naive implementation, thus the internal representation almost exactly mirrors the elements and structure of the final description (registers, ALUs, etc.). In this case, the main drawback is the early choice of an implicit scheduling of operations, which increases the degree of dependence on the quality of the original behavioural description.

2.3 Synthesis and the Design Space

Synthesis is the process of transforming a behavioural description, in the form of its initial internal representation, into a structural implementation, optimised according to objectives set by the user, such as minimum execution time for a given maximum area or power consumption. The manipulation of these objectives, both automatically and under user control, forms the basis for exploration of the *design space* [3, 9, 25], which is defined as the *n*-dimensional space describing all possible implementations of a particular input description, in terms of *n* design aspects. For example, Figure 2.5 shows a two-dimensional design space in terms of area and delay.

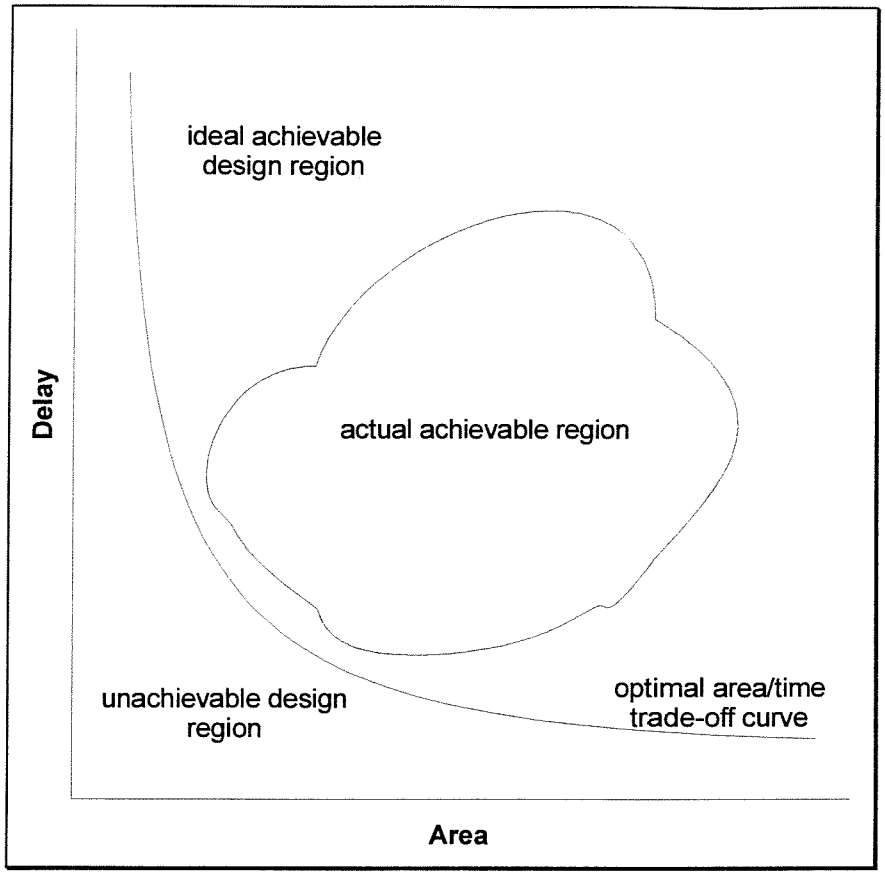


Figure 2.5 An area-time design space

Design space exploration is one of the key features of behavioural synthesis systems, allowing the user to investigate a whole host of alternative structural implementations from a single behavioural description. The design space can be partitioned into various regions, the scope of which depends on many factors including the original design description, the target technology, and the abilities of the synthesis system itself [26]. For any particular design and technology, the space can be divided into two regions containing the set of *achievable* and *unachievable* implementations, as shown in Figure 2.5. Here,

these two regions are separated by the optimal *area-time trade-off curve*, along which the most efficient hardware implementations lie (on discrete points). For a given system, only a finite portion of the achievable region will be covered. The characteristics of this *actual achievable region* depend on the details of its implementation, such as the algorithms, and the design space modelling methods [26] employed.

The synthesis process is usually divided into 4 sub-tasks: *behavioural optimisation*, *scheduling*, *allocation*, and *module binding and controller synthesis*. The exact definition of what constitutes each of these is subject to some variation depending on the synthesis system concerned, however all follow the same basic model:

- *Behavioural optimisation* - in much the same way as quality software is written for human readability and maintainability, so a hardware engineer using behavioural synthesis will tend to write a description with more regard for clarity than optimal implementation. Features such as procedures, constants, and temporary variables, enable the description to be easily understood and modified, however, they can also add a considerable overhead to the synthesised implementation. The types of optimisations applied at this stage are similar to their software counterparts encountered in optimising compilers, such as *common sub-expression elimination*, *loop unrolling*, *inline expansion*, and *constant propagation*. These are either applied directly to the behavioural description before or during the compilation process [13, 14, 27, 28] or more usually, to the initial internal representation created directly after compilation [21, 29]. For example, a microprocessor may employ a separate procedure to implement each of its instructions. If these are synthesised as discrete blocks, there will be no sharing of resources between any procedures or the main bulk of the design. The input description, therefore, defines a partial (and probably non-optimal) structure limiting the scope of the design space explored. Inline expansion solves this problem by flattening the original hierarchy, thus providing synthesis with maximum freedom to optimise the design.

- *Scheduling* is the task of assigning operations to particular control steps, whilst attempting to optimise the control graph in such a way as to achieve the objectives of the user. Typically this means optimising the schedule with respect to timing and/or resource constraints, which in its simplest form comes down to trading parallelism in the control graph for extra hardware in the data path to achieve the desired delay. For

example, the three multiply operations in Figure 2.3 could either be implemented in a single state requiring three separate multipliers, or in separate states, allowing all three operations to share the same multiply unit (depending on the allocation chosen). This trade-off between area and delay is complicated by the fact that although the former solution requires less states than the latter, the cycle time is considerably larger being defined by the longest combinational block, ie. the cascaded multiplies (from *4*a*c*), assuming operations cannot overlap control states (multicycling). Of course in reality, any worthwhile behavioural optimiser would have transformed the multiply by 4 into a bit slice (or shift left by 2).

- The goal of *allocation* is to optimise the amount of hardware required to implement the data path according the chosen operation schedule. Data path operators, storage blocks and interconnects are mapped onto functional units (adders, multipliers, ALUs etc.), memories and communication elements (buses and multiplexors). The main objective is to partition the data path into an optimal selection of blocks, sharing both functional and storage units as required. In the second example schedule described above, all three multiplications may be implemented by a single multiplier, together with associated busing of the inputs to select the desired operation. In addition, all the storage elements may be implemented using three registers (assuming *a* and *c* are not required later) partitioned, for example, into groups: *a*, *t1*, *s* and *r*; *c* and *t2*; and *b*.

- *Module binding and controller synthesis* - once the data path elements have been allocated they must then be assigned to technology-dependent hardware blocks implemented from units in the target cell/module library. This is often considered as part of the allocation phase, which may be able to use library-dependent characterisation data thereby enabling synthesis to take more accurate consideration of the size and delay of units chosen. A similar operation must also be performed on the control graph to implement circuitry for generating the appropriate data path signals. This is usually implemented either as a hardwired state machine or a micro-coded controller.

2.4 Scheduling and Allocation

Scheduling and *allocation* are central to the synthesis and optimisation of structure from behaviour. These two tasks are heavily interrelated as demonstrated by the example of the

three multiply operations in the quadratic equation solver: if all three are allocated to one multiply unit, each must be scheduled in a separate control state; on the other hand, if the operations are scheduled in one control state, they must be allocated to separate units. Clearly any binding decision (those which cannot later be undone) made early on in the synthesis process can have a profound effect on the range of achievable implementations, thus restricting the design space. The problem of ordering scheduling and allocation so as to minimise these restrictions is tackled in a number of ways by the various different synthesis systems.

The simplest solution is to impose some general resource limit prior to scheduling, followed sequentially by allocation and binding, which usually takes the form of a limit on the number of functional units available for a particular type (eg. only one multiplier), and is, in effect, performing a very rough global allocation step. This *open-loop* approach is taken in some of the older systems such as MIMOLA [30] and Flamel [31]. A refinement to this method is to close the design loop by re-synthesising with modified resource constraints, once an initial implementation has been obtained. For example, Chippe [17] dynamically modifies the parameters and constraints used to control scheduling and allocation via a set of rules. These respond to an evaluator, which assesses the current implementation with respect to user objectives on area, delay and power. Individual tasks, or the entire synthesis loop, may be re-iterated ad-infinitum to further improve the design.

An alternative method adopted by a few systems is to perform allocation before scheduling, producing implementations optimised for area within specified timing constraints. The CADDY [32] system, for example, performs complete data path synthesis from compilation through to module binding. Area is minimised using both global and local rule-based optimisations within local timing constraints specified in the behavioural description. The data path is then scheduled, optimising the number of states within the constraints imposed by the allocation and timing specifications.

The final approach, and the one used by MOODS, is to attempt to perform both scheduling and allocation simultaneously using a stepwise refinement method. Examples include HAL [33], YSC [1, 34] and CAMAD [19, 35]. HAL implements a *force-directed* scheduling/allocation algorithm that attempts to minimise operator concurrency, and therefore the number of functional units, by distributing similar (ie. shareable) operations as evenly as possible within the schedule, based on user constraints. The IBM Yorktown

Silicon Compiler (YSC) on the other hand, starts by synthesising a maximally parallel implementation, on which state-splitting is performed to reduce the cycle time, and allow more sharing of hardware resources to decrease area. This process iterates until timing and resource constraints are achieved.

In a similar vein, CAMAD forms an ETPN representation (such as Figure 2.4) with an initial schedule and allocation taken directly from the behavioural data flow. The system then applies a range of semantic-preserving local transformations to the design, which together perform re-scheduling, re-allocation and re-binding. This is an iterative process controlled by a global heuristic algorithm, and guided by an analysis of the current state of the design with respect to user constraints on speed and size.

2.4.1 Scheduling Algorithms

Scheduling algorithms can be partitioned into two basic types: *constructive*, which attempt to create an optimum schedule from scratch by sequentially adding operations until all have been scheduled; and *transformational*, which attempt to improve an initial schedule, usually maximally serial or parallel, by means of local transformations such as chaining and state merging.

The simplest constructive schedulers use the *as-soon-as-possible* (ASAP) and *as-late-as-possible* (ALAP) algorithms. These process operations in a fixed, top-down or bottom-up order, scheduling each one in either the earliest (ASAP), or the latest (ALAP) possible time step. The order in which operations are processed is determined by the topology of the data flow graph, ie. based solely on the data dependencies. Figure 2.6a shows an example data flow graph and its associated ASAP schedule. Resource constraints are taken into account through the use of *conditional postponement*, delaying an operation if the current time step exceeds pre-set resource limits. For example, whereas in Figure 2.6a operator *c* is scheduled in cycle 1, in Figure 2.6b, due to a resource limit of only 1 multiplier unit, it has been postponed to cycle 2. This highlights the main deficiency of the algorithm: the fixed ordering does not assign any priority to the operators (other than data dependency), so critical operations (those affecting the total delay) may be postponed by non-critical ones. In the example, the total delay increases since operator *d*, on the critical path, is postponed due to its dependency on the result of *c*. Some of the earlier systems used this algorithm, such as Emerald/Facet [36] and Flamel [31].

List scheduling takes a more sophisticated approach by ordering operations according to a local *priority function*. Each control step is scheduled sequentially and, when a resource conflict occurs, operations are postponed according to their priority. For the sample example as above, Figure 2.6c shows the result of list scheduling where the priority for each operator is calculated as the length of the data path to the end of the block (marked in braces in Figure 2.6a), a definition used in the BUD [37] system. An alternative priority function is the *mobility* used by SLICER [17, 38], which is the difference between the ALAP and ASAP schedule for a given unit, thus operations on the critical path will have a mobility of zero and will be assigned the highest priority. In addition, when all critical operations for a particular control step have been scheduled, the system attempts to include the critical operations for the next control step. If these can also be included then one less control state will be required.

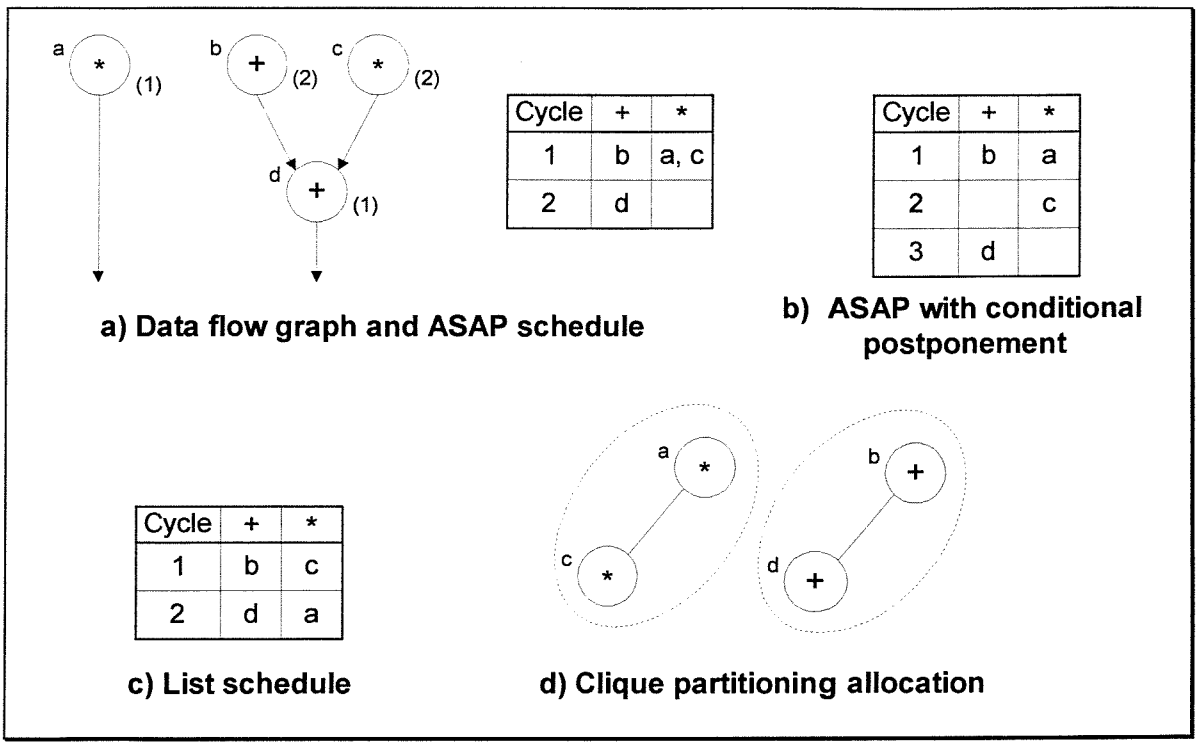


Figure 2.6 Examples of scheduling and allocation algorithms

All the above algorithms use greedy strategies which make binding decisions on a first come first served basis with no backtracking or lookahead. Judgements are based on local considerations, which may be optimal for one locale, but do not necessarily produce a global optimum. Other systems attempt to take a more global view using modified versions of the above algorithms, together with a *closed-loop* approach to modify the optimisation if it goes awry, and try again. For example, MAHA [39] uses the notion of *freedom* for the priority in an enhanced list scheduling algorithm. Here, the critical path is

first determined and scheduled using the minimum number of time steps (ASAP), and is then followed by the non-critical operations, which are list scheduled in order of their *freedom.* This is defined as the time difference between the inputs of an operation becoming valid, and its outputs being required, ie. the difference between ALAP and ASAP schedules. If at any point a cost constraint cannot be met, the critical path is re-scheduled with additional time steps and the process re-iterated. A similar approach is taken by the *force-directed* scheduler used in HAL (described earlier). Although these systems allow a degree of backtracking to undo poor design decisions through a complete re-synthesis, they are still relatively inflexible due to the global nature of their iteration schemes and do not allow true two-way trade-offs to be made.

Transformation-based approaches iteratively apply a set of local transformations to an initial schedule, guiding the design toward the objectives specified by the user. This technique has been adopted by a number of systems including the Yorktown Silicon Compiler [1, 34] (described earlier), CAMAD [19, 35], MOODS and the system described by Devadas and Newton [40]. Although these utilise similar sets of transformations, such as state merging and state splitting, they differ in the methods used to select sections of the design to be operated upon. The Yorktown compiler uses a local approach, splitting the largest control states (ie. those controlling the greatest data path area) in an initially maximally parallel schedule, in order to increase operator sharing within timing constraints. This suffers from the same problem as the constructive algorithms in that it only considers the effects on a per-control state basis, and hence may not find a global minimum. For example, if the largest state is on the critical path, it will be split, thus decreasing area and increasing the total delay. However a better solution may well be to split two smaller states which are off the critical path, thus decreasing the area but leaving the delay unchanged.

An alternative approach taken by the latter three systems is to formulate the synthesis and optimisation problem in terms of the minimisation of a *cost function.* This is a measure of the global optimality of the current design configuration with respect to the target objectives. The method adopted to solve this problem varies: CAMAD uses a complex heuristic algorithm which targets the critical path when applying transformations in order to improve the cost function. The algorithm iterates until a minimum is found, however this is not necessarily a global minimum as CAMAD does not allow design degradation. MOODS and Devadas and Newton approach this problem using simulated annealing,

which allows a certain amount of design degradation to occur in order to avoid local minima. Heuristic and simulated annealing optimisation within MOODS is discussed in more detail in Chapter 4.

The various techniques described above form the basis for the scheduling operations in most systems, often with some added features such as the ability to perform operator *chaining* (cascading two dependent combinational data path units in a single control state), *multicycling* (allowing an operator to execute over several control states) and *pipelining*.

2.4.2 Allocation Algorithms

As described earlier, allocation is the task of grouping data path elements (operators, storage and interconnect) together, binding each group to a functional or storage unit, which may then be implemented by a physical hardware module (during module binding). Most systems perform this step after the data path has been scheduled, thus the allocation process performs a many to many mapping in such a way as to minimise cost (typically area) whilst not violating the schedule (timing constraints). Allocation algorithms, akin to their scheduling counterparts, broadly split into two groups: *iterative/constructive* and *global*.

The *iterative/constructive* algorithms iteratively build up an allocation, operation by operation, within any resource and scheduling constraints. The algorithms used to select both the operation to allocate, and the functional unit to which it will be bound, are at the heart of the process. The simplest approach processes operations in a fixed order, usually based on the data/control flow topology, using *local* rules to greedily assign them to an appropriate functional unit. Such an algorithm is employed in the data-memory allocator [41, 42] used in the early CMU-DA system. More sophisticated algorithms make use of *global* rules which may consider a number of alternative allocations before choosing the most suitable, based on the minimisation of some cost metric such as the incremental area.

EMUCS [43, 44], a successor to the data-memory allocator in the CMU-DA project, is an example of an iterative algorithm using global selection rules, which attempts to minimise overall area within resource constraints when performing allocation and binding from a scheduled data flow. It implements a greedy iterative algorithm, which uses a *cost table* to summarise the incremental cost of binding each unallocated data path unit to every

available (and suitable) hardware unit. Each iteration builds a new cost table, and calculates the difference between the two minimum allocations for each operator. This figure is, in effect, the cost of not performing the minimum allocation, assuming that on the next iteration it will no longer possible. The allocation performed on each iteration is the one with the largest difference, ie. the one which, if not performed immediately, would result in the greatest area increase. Since this algorithm does not allow backtracking, it does not necessarily provide a minimum allocation as any poor decisions cannot be reversed.

Some systems use iterative algorithms based on the application of transformations to an initial naive allocation, including such operations as *operator sharing* or *register folding* (sharing). Transformations are targeted using a number of different algorithms: Flamel [31] attempts to share all pairs of operations within scheduling constraints, selecting between multiple choices based on the greatest local area saving. CAMAD [19, 35] on the other hand, uses a complex heuristic that includes scheduling, allocation and binding in the same algorithm. MOODS and the Devadas and Newton system [40] take a more general approach, bunching all scheduling and allocation transformations together, and applying a simulated annealing algorithm to the whole design in an attempt to perform multi-way trade-offs. These systems also provide backtracking transformations such as operator and register splitting which allow them to undo poor decisions.

In contrast to the above, *global* algorithms are not based on iteration, instead analysing the data path as a whole, and attempting to simultaneously allocate all elements. A number of approaches are taken to determine an optimal partitioning. The Facet [45] allocator is an alternative to EMUCS in the CMU-DA system and uses a graph-based clique-partitioning algorithm, which attempts to partition the data path into blocks that may be assigned to particular functional and storage units. It builds up a graph, where vertices represent the data path elements, which, for any given mutually-exclusive pair, are connected by an arc to indicate that they may be shared. Thus a maximal partitioning of fully interconnected vertices represents the minimum hardware cost. This NP-complete problem may then be solved using a variety of heuristic algorithms. Figure 2.6d (shown earlier) is an example graph for the list schedule of Figure 2.6c, showing the partitioning of operations into two sets, each allocated to a single data path unit.

Another global approach is to perform an exhaustive search of all possible allocations pruning the design space with a branch and bound algorithm, a method used in SPLICER [38] and FIDIAS [46]. SPLICER performs an exhaustive search of all possible allocations for a limited number of control steps, the best of which is then chosen as the starting point for another iteration of the algorithm on the following control steps. The depth of the search may be adjusted to provide a more optimal result, however, the exponential increase in processing time means that only small designs are feasible. FIDIAS uses a similar process but employs extra bounding heuristics to prune the design space considerably by aborting any search that results in more than a small increase in the cost of the previous step allocated. Since the system allows a small amount of degradation, it is able to climb out of local minima, at least for the small set of designs described in the paper.

2.5 Modules and Module Libraries

Module binding is the process of mapping abstract data path units to specific hardware implementations chosen from a *module library*. Each module describes a particular functional unit at the appropriate level for the target post-processing tools, be it a customised geometric layout, a collection of standard cells, or an RT level structural description. Depending on the requirements of the synthesis system, the library may contain exactly one implementation per unit, in which case the module binding exercise is a basic one-to-one mapping, or may include multiple forms, such as ripple-carry and carry-lookahead adders, which are selected according to particular performance requirements. In this case, the library must be capable of providing accurate characterisation data for each module, generally in the form of area and delay estimates, which are used to guide the scheduling and allocation processes.

The most basic form of module library is simply a database that maps functional units onto a pool of library cells defined by the target system, possibly including some that are hand-crafted for a particular design. Although easy to implement, this approach has the drawback of forcing the library to be entirely technology-specific. The set of available modules is therefore limited to those provided by the particular target system, or requires an extensive layout-level customisation effort. A more advanced approach comprises a set of parameterisable *module generators* [47] each of which encompasses a range of module

implementations based on a set of input parameters such as bit-width, required pin-out etc. Module generators fall into two basic groups: cell compilers and architectural-level module optimisers.

Parameterised *cell compilers* [48] are an extension of the basic standard cell design process, where a library of pre-designed technology specific cells are provided by an ASIC (Application Specific Integrated Circuit) manufacturer. For example, the generators described by Gajski and Lin [49] are parameterised cell compilers that allow users to customise many aspects of the cell design, such as the number of inputs into a basic gate and its drive capabilities. They still suffer, however, from a degree of technology-dependence being based on the abutment of pre-defined cells.

By taking advantage of layout compactors [47, 50] to automatically force a design to conform to design rules, a considerable amount of technology independence may be obtained. The Cathedral II [51, 52, 53] silicon compiler includes an *intelligent module generator environment*, which uses an interpreted *lisp*-based design language to describe how leaf cells are placed and connected. The language uses *loop* and *if.. then.. else* constructs to automate the structure, together with variables and input parameters to specify, for example, ROM size and adder bit-width. A router and compactor connect and optimise the design, and a symbolic layout editor is used to create leaf cells, which are designed in a technology-independent manner and are then compacted based on a set of technology design rules.

Architectural-level generators make use of the hierarchical nature of arithmetic and logical operations, both in module generation and characterisation. By abstracting module structure in terms of basic gates they are able to achieve a high degree of technology independence. The disadvantage, however, is that they must rely on low-level logic optimisation, and placement/routing systems in order to finally achieve a physical implementation. Some hierarchical generators are discussed in more detail in Chapter 3.

2.6 Summary

This chapter has outlined the overall synthesis process and the main sub-tasks involved. It has highlighted some different methods employed by a number of systems, providing an insight into some of the key factors involved in the design of a behavioural synthesis

system. One of the most important, is how to deal with the interdependent nature of these tasks and how effectively trade-offs are made between the many design aspects. This affects the ability of a system to effectively explore the design space without falling into local minima traps. Chapter 4 describes the MOODS synthesis system in more depth, and elucidates on the manner in which these issues are addressed.

Chapter 3

Background Material

This chapter presents background material describing influential research in the development of *expanded modules* (described in Chapter 5) and the VHDL compiler enhancements (Chapter 6). It is split into two main sections: section 3.1 describes a number of systems that exploit the inherent hierarchy in arithmetic operations in a variety of ways within a synthesis environment; while section 3.2 examines some recent research into the application of VHDL specifically to behavioural synthesis.

3.1 Exploiting Module Hierarchy

The implementation and integration of module libraries is an oft-neglected aspect of the synthesis process. Most systems assume the existence of an arbitrary pool of modules provided by the lower level target tools (logic optimisation or placement/routing), along with performance characterisation data, but pay little attention to the internal construction of either the module library or the modules themselves. As will be shown in Chapter 5, the expansion of the internal structure of a module within the synthesis loop can significantly improve the quality of the final implementation, particularly when applied to the larger units. This requires a detailed knowledge of the internal composition of more complex modules in terms of their less sophisticated brethren (eg. multipliers made from adders and shifters), thereby exploiting the inherent hierarchy within arithmetic and logical units. This hierarchical nature has been put to a number of uses in various VLSI design tools ranging from sophisticated module databases, such as Fred [54], to complete parameterisable module optimisers, such as those discussed by Theeuwen et. al. [55].

3.1.1 Hierarchical Module Estimators/Generators

In order to integrate module binding into the synthesis process, a design system must be able to evaluate alternative module realisations for a particular data path operation. Thus a library should provide detailed physical and operational information for each configuration (bit-width, number of inputs, number of outputs etc.) of every available module, together with some mechanism for selecting modules based on the data path operations implemented (eg. return all 8-bit adders).

Fred [54] is an object-oriented module database, originally designed as a support tool for VLSI architecture design and floorplanning. It enables a designer to rapidly investigate the various modules available, modelling physical, electrical, timing, clocking and functional properties via a set of general procedures (*methods*). The Fred system has also been used as the technological base for BUD [26, 37, 56] (part of the Algorithms to Silicon project), and an enhancement to the original Design Automation Assistant [57, 58]. BUD employs layout-level module data from Fred to guide the initial allocation of a VT description before scheduling. In this way its authors hoped to alleviate some of the problems associated with traditional top-down synthesis, where many of the most critical design decisions are made without any consideration of the low-level implementation.

Fred builds a module database from a set of classes[1] each of which is ultimately derived from the base class *module-mixin*. This defines a standard set of properties and access functions for characterising a module, ie. a standard interface. Top-level classes representing the real physical modules are derived via generic functional abstractions, for example, *ripple-carry adder* is derived from *generic adder*, which is itself derived from the base class *module-mixin*. As the hierarchy is ascended, each level provides more detailed module properties, with *primitive* components (leaf cells, abstract generic functions) modelled via a simple lookup table, and *non-primitive* components, which make up the physical module set, modelled as a hierarchical network of sub-components. Fred provides a number of routines to accurately compute the characteristics of non-primitive modules through the properties of its constituent sub-modules, taking into account technology dependent placement and interconnect data to estimate the final layout. A

[1] Fred is written in *Flavors*, an object-oriented extension to Lisp. For this discussion standard C++ terminology will be used, which differs slightly from Flavors, but is based on the same general principles.

module designer may augment a class definition with customised estimation routines to override this default modelling if desired, to obtain swifter or more accurate results.

Fred performs searches of the database through two alternative views, which order the modules around either *generic* or *function relations*. Generic relations group a module according to the generic functional type of which it is an implementation (identical to the class inheritance tree). Thus a search for a generic adder may return a module set composed of ripple-carry, carry-select, and carry lookahead adder modules. *Function relations* on the other hand, describe a module in terms of the data path operators it is able implement. For example, searching for an increment operator may return an adder, an increment module and an ALU.

Fred illustrates two important principles: firstly, the use of a hierarchy to describe module structure enables it to make accurate calculations based on the constituent sub-components of a module, and also allows a degree of technology-independence since updating the primitive components will automatically update the non-primitive ones. Secondly, the use of an object-oriented paradigm to describe the relationships between different modules and their generic forms aids searching, and allows the library to be organised in a logical hierarchical fashion. Fred is, however, only a library database, albeit a rather sophisticated one, and simply holds data on a set of external module generators, each of which implement a *fixed* topology. Fred does not perform any optimisation of the module structure based on user input requirements, it simply describes what is already available, thus only a limited exploration of the design space for a particular function is possible.

Another system, similar in structure to Fred, is the module library used in the FACE environment [59]. This is also based on an object-oriented database where each module is described via *structural* and *physical views*, intended for simulation and physical layout respectively. Characterisation modelling is not performed however, and module properties are only obtained through physical realisation using placement and routing tools. Thus the library is simply a storage medium for generic relations, and leaves the analysis and selection of different module implementations up to other parts of the design system.

BADGE [60, 61] integrates module generation with a hierarchical library, which can synthesise a low-level implementation, optimised according to user input requirements for area and delay. The BADGE library comprises a graph, constructed from a set of *generic*

arithmetic building blocks, building block *generators* and a set of technology-specific *target* units. The generic blocks refer to abstract functional unit types, such as a multiplier or adder, which define a standard interface, but contain no structure. These are mapped onto real components implemented either by an element of the target library, if a suitable one exists, or by a building block generator, which generates a module description from a set of interconnected lower level generic parts, eg. a ripple-carry adder described as a network of full and half adders.

Module synthesis is based on the optimisation of a *configuration tree*, describing all possible generator invocations required to completely define a module in the target technology. Figure 3.1 shows a partial configuration tree for a ripple-carry adder, comprising three types of node: an *OR node*, which represents a generic component implemented by exactly one of its successor nodes; an *AND node*, which is a real generated module, connected to all of its constituent sub-components; and leaf nodes, formed from the set of pre-defined target library cells. In order to obtain a complete module implementation in terms of target cells, a path through the configuration tree must be developed. This is called the *configuration plan* and describes the series of generator invocations required to transform each generic part into a suitable real part. At each step, a generic component is *refined* into a real module, hence the term *stepwise refinement* to describe the process.

A configuration plan is iteratively built up from the configuration tree via a directed search using a branch-and-bound algorithm. Heuristic selection procedures are used to guide the path taken through *OR* nodes (ie. which real implementation to use for a generic block) based on estimates of building block performance (area and delay). These figures are obtained using a set of *estimation formulae* to model the characteristics of each real block in terms of its constituent generic sub-blocks. When an *OR* node is encountered during tree traversal, a *formula evaluator* determines performance estimates for each successor node which, in conjunction with requirements specified by the user, are used to prune the search tree. A number of algorithms are implemented to perform the selection of branches to be eliminated. These include a simple *hill-climbing* search that only chooses the best estimated alternative, a *beam* search which selects a limited group of alternatives, and an exhaustive search where every possible combination of module implementations is considered before choosing the closest to requirements. Once a configuration plan has

been developed, the appropriate generators are invoked in the correct order to synthesise the completed module as a circuit description of interconnected target cells.

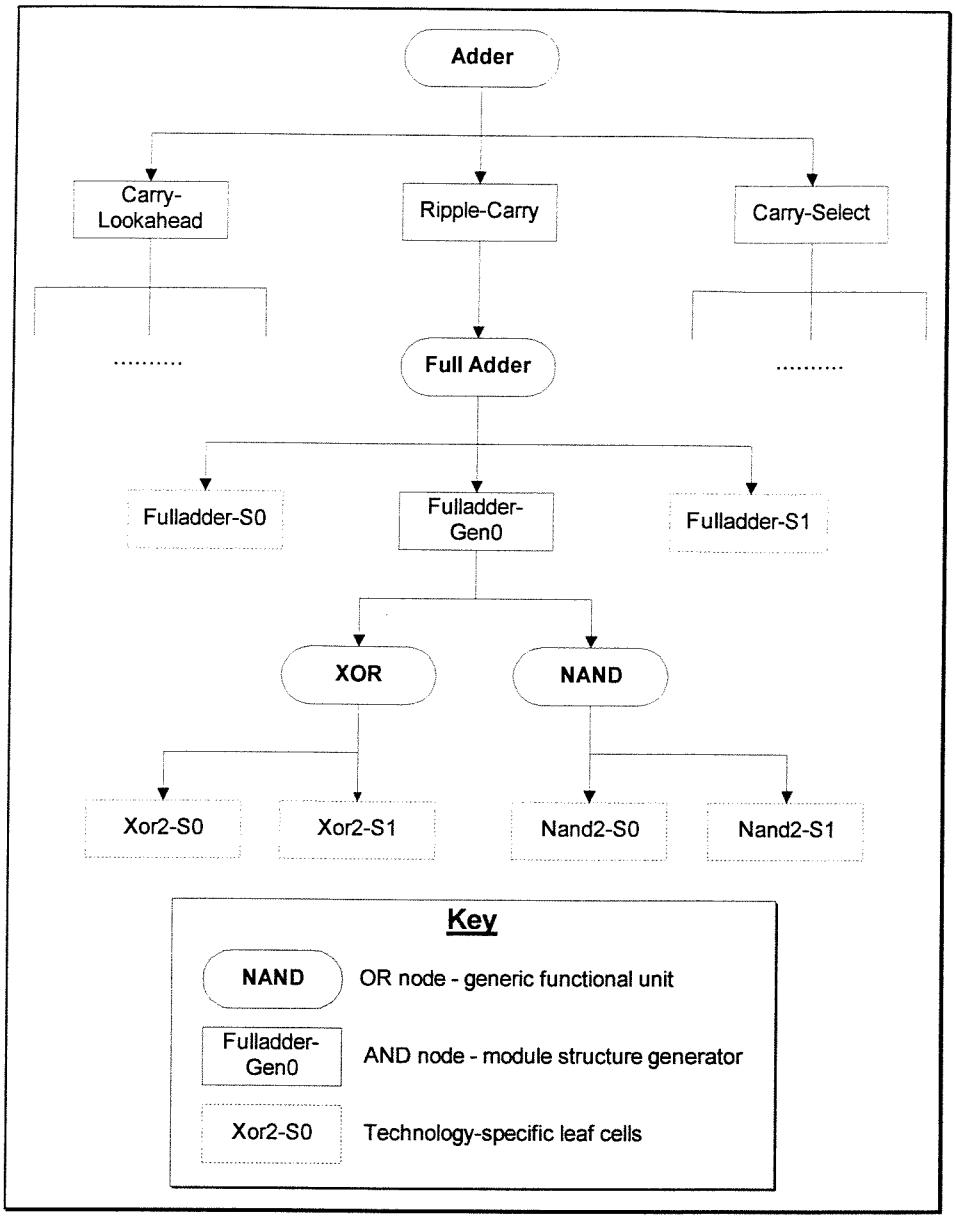


Figure 3.1 Partial configuration tree for an adder functional unit

Since the only technology-dependent parts of the library hierarchy are the leaf nodes, BADGE is able to target different cell libraries through a simple exchange of primitive components. However, these blocks represent only a basic set of gates common to most cell libraries, and do not therefore make optimum use of any more complex cells which may be available. This problem is dealt with in a final step, which performs logic-level optimisation and technology mapping using a graph-matching algorithm to substitute sets of basic gates with more complex implementations, thus producing a final circuit structure optimised both to the target technology library and the user input requirements.

All the systems described thus far utilise the natural hierarchy of arithmetic operations to organise a module database into a physical and logical structure. These databases contain a number of alternative topologies capable of implementing each functional unit type thereby allowing a limited design space exploration. In addition, the BADGE system further optimises the basic structures through the selection of different low-level cells provided by the target library.

The generators described by Theeuwen et. al. [55] take a different approach to module optimisation using the *block principle* to produce a wide range of different module structures from a single parameterisable generator. Unlike BADGE, which concentrates on optimising a fixed topology (eg. speeding up a ripple-carry adder), this method operates at a higher level by adapting the structure of the module itself, through a serialisation process trading off module area against the number of cycles required to perform the operation. Indeed the generators described only produce a netlist based on functional units and registers, with technology mapping postponed until a final logic synthesis stage.

The block principle splits the module into a number of smaller sub-operations processing only a portion of the input data, which is divided into a number of sub-blocks. The entire function is iteratively built up over several clock cycles, with the number of iterations dependent on the input partitioning. Thus the architecture is able to produce a range of implementations ranging from purely combinational (no sub-blocks, one iteration) to fully serial (minimum sub-block width, lots of iterations).

The authors describe two generators to demonstrate the block principle in action. The first is a multiplier based around a configurable structure, using a single combinational multiply, an adder, and an output shift register, together with the necessary control logic. A simplified block diagram is shown in Figure 3.2. This structure is a direct implementation of the standard long multiplication method, where the inputs are multiplied bit-by-bit with the result shifted and summed to form a partial product. In this case, the inputs are split into blocks that are multiplied, one pair per iteration, by a combinational multiplier. The result is added to the shifted partial product stored in the output register, and the process repeated for another pair of input blocks. By changing the block sizes, the number of cycles required to perform the entire operation is altered, together with the size of the required multiplier and adder blocks. Thus when the block size equals the input register width, the adder and shifter disappear completely yielding a

purely combinational multiply module. At the other extreme when the block size is one bit, the resulting circuit is a simple sequential shift and add, the combinational multiply becoming an AND gate. The other generator described is a barrel shifter that splits the input into a number of blocks, each of which is shifted by a combinational shifter in a single cycle and the result built up in an output register. Hence the structure configuration ranges from a purely combinational shifter to a sequential bit-by-bit shift register circuit.

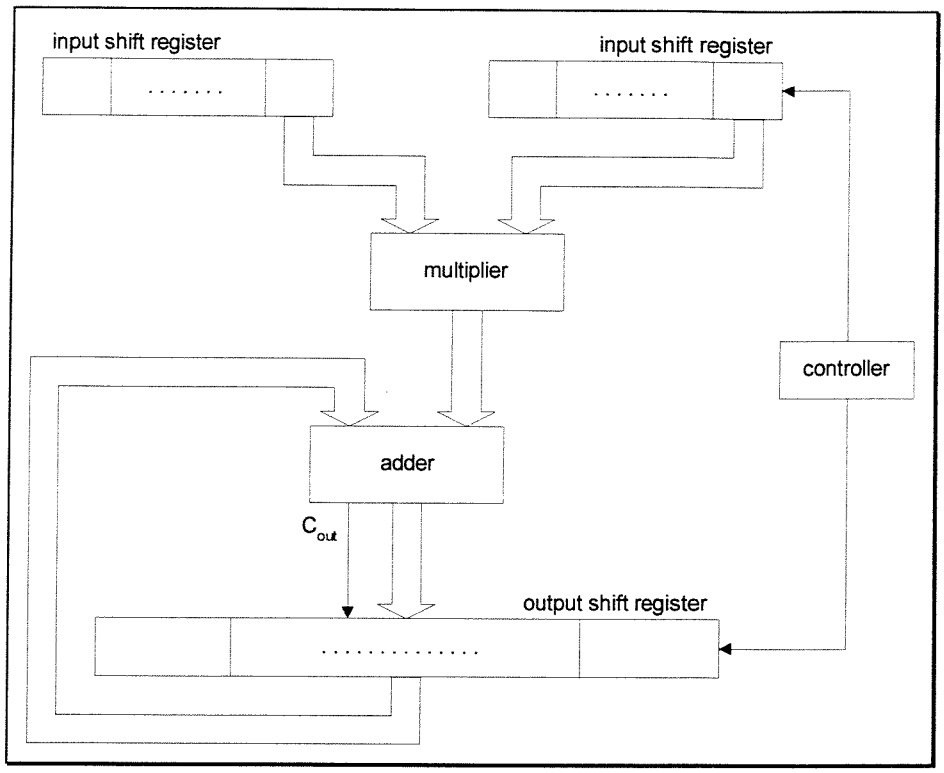


Figure 3.2 Block multiplier architecture

The higher level approach taken by Theeuwen et. al. is based on the rationale that optimisation of the overall module structure has a much greater effect on the circuit characteristics than just optimising the low-level cells. This, coupled with the broad range of possible implementations, enables a much more comprehensive coverage of the design space for a particular functional unit than the systems described earlier. Since the generators produce a structural netlist, the use of a system such as BADGE to implement the individual sub-modules is not precluded, thus obtaining the best of both worlds. As always, there are a number of drawbacks associated with this higher level approach: no characterisation data is provided by the generators, other than the number of cycles required, thus the only method for accurately evaluating a module is to actually implement it down to the logic level. This may be tackled relatively easily by taking the same

approach as Fred or BADGE to estimate performance based on the characteristics of the individual sub-modules, thus enabling trade-offs to be considered within the generators.

3.1.2 Exploiting Hierarchy within the Synthesis Loop

All the systems described in the previous section are used to generate module descriptions isolated from the main synthesis loop. Although characterisation data is used to direct the scheduling and allocation tasks, such as in the BUD/Fred combination, the actual realisation of the module structures occurs after synthesis, during the final module binding stage. Thus modules are considered as isolated black boxes, with any interaction between them occurring at this external level. The two papers discussed below utilise hierarchy within the synthesis process to various different ends. In particular, they attempt to improve the final implementations obtained through a limited flattening of the module or operator hierarchy, thereby providing more scope for optimisation during synthesis.

Gasteier et. al. [62] make use of the block principle developed in [55] to implement complex operators in a VHDL description at the register transfer level (RTL). They describe a manual process to inline expand certain operators with a structural equivalent produced by the module generator. This is done to expand the limited set of operators supported by the target technology library, and to provide a level of design space exploration not normally available in RTL synthesis systems.

The process described involves identifying critical operators in the source VHDL and substituting these with a *component* instantiation that performs the replaced operation. The authors describe an example application to demonstrate the use and effectiveness of the method. This contains a pair of multiply operations, expanded into two *BlockMult* components that are implemented by block multipliers generated as described earlier. The module generation stage is a separate user-controlled program producing both a behavioural and structural VHDL description for a particular module. These are used to implement the *BlockMult* component during simulation and synthesis respectively. The behavioural description models the operation of the module at the cycle level, using performance estimates to ensure that simulation results show the correct cycle-by-cycle behaviour of the system. During synthesis this is replaced by the structural version, which is an RTL description optimised for the appropriate target tools. Investigation of different

module structures is achieved through user interaction with the generator program, and manual iteration of the entire synthesis loop.

To facilitate user control over the module generation phase, the generator described in [55] is integrated into a mini design environment. This takes user input in the form of a number of cycles, area and cycle delay specifications, and produces a set of behavioural/structural descriptions of modules that fulfil the criteria. Selection of suitable implementations is achieved by generating every possible architecture and selecting the most appropriate according to estimated performance. Estimators are provided to calculate module characteristics based on their structure, and the performance of the constituent sub-modules. These figures are generated according to three different target technology models: cell-based layout is modelled via gates and interconnects, FPGAs via registers and quad input logic functions, and PALs via registers and two-level logic functions.

The results obtained demonstrate the wide range of implementations achieved for the example application, showing a large spread in final area and delay figures at the layout level. This is much wider for larger bit-widths, unsurprising considering that the size of a combinational multiplier is typically $O(N^2)$. Despite good results, the method as described is clearly unsuitable for all but the smallest of designs. The process not only requires manual operation of the module generators, but also significant modification of the original VHDL source. This needs to be automated, allowing the user more freedom to investigate different possibilities without having to go through the rigmarole of continually editing source code and re-synthesising the entire design. Also, from a behavioural synthesis perspective, the use of RTL means that no scheduling or allocation optimisations are performed, at least not automatically, and so there is no optimisation between modules, ie. they are still treated as black boxes.

The final work detailed in this section exploits the hierarchy both within a CDFG and, to a limited extent, within data path operators. The flattening of these structures is combined with a novel approach to scheduling, which emphasises local timing relationships in an attempt to resolve some of the problems associated with the commercial application of behavioural synthesis.

Ly et. al. [63] describe the use of *behavioural templates* in the timing constrained scheduling of a CDFG (see Chapter 2). A behavioural template defines the fixed timing

relationships between a number of data path operations, locked into a relative schedule. Templates may be overlapped in time, and combined during scheduling to form new templates, while maintaining the fixed timing between the constituent operations. Figure 3.3 shows two example templates which are merged as result of scheduling the second to start one cycle after the first.

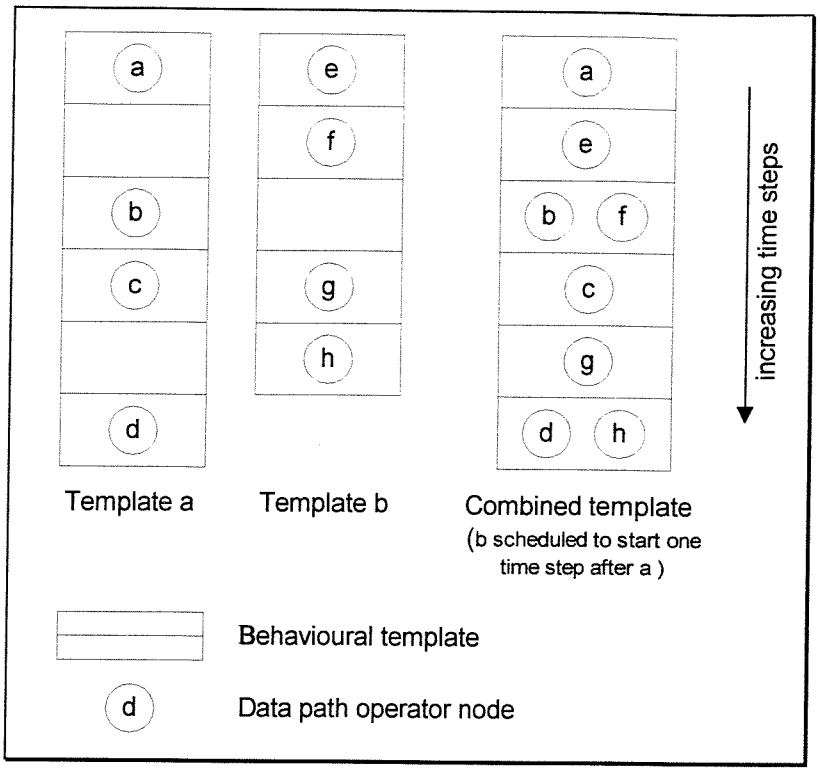


Figure 3.3 Behavioural template example

Scheduling of the CDFG is formulated in terms of behavioural templates using standard scheduling algorithms. Initially, one template is created for each data path node. From these, a set of time frames is determined specifying the range of time steps to which each template may be scheduled. These are calculated from the difference between the initial ASAP and ALAP schedules (time frames are equivalent to the freedom measure used in MAHA [39]). The scheduling operation implements an iterative/constructive algorithm (list or greedy scheduling) to successively assign each as-yet unscheduled template to a time step within its time frame, merging templates as it progresses. Following each iteration, the time frames are updated to reflect the new state of the design; these are then used, together with resource constraints, to determine priority functions that direct the scheduling process, repeating until completion.

The key feature of the template-based approach is its ability to maintain a relative schedule from start to finish. This enables the system to provide a number of useful facilities :

- *Timing constraints* - the fixed nature of the timing relationships within a behavioural template allows user-specified local timing constraints between any pair of operations to be implemented simply by locking them into a template at the required relative positions, prior to the start of the main scheduling operation. Other operations may then be scheduled within the template without violating the initial specifications.

- *Multicycle and pipelined operations* - most synthesis systems implement multicycle and pipelined operations as a single operation scheduled over several sequential time-steps. Behavioural templates enable a more flexible approach to be taken whereby the multicycle operation is split into several single cycle sub-operations (or in the case of pipelines, pipe-stages) that are locked in a template. This is used in [63] to demonstrate a novel method for scheduling memory I/O operations, automatically taking into account constraints on accesses to I/O ports. By implementing each stage of the memory I/O protocol as a separate sub-operation within a template, the exact interface timing may be defined. In addition, resource constraints on these sub-operations can prevent, for example, concurrent accesses to the same address bus. The case of pipelined units is also considered: most synthesis systems dedicated to pipelining (such as SEHWA[64, 65]), allow the overlapping of pipelined operations and the sharing of individual pipe-stage units as a specialised form of scheduling. Using behavioural templates, the sharing of pipe-stages (modelled as template sub-operations) becomes an inherent feature of the scheduling process, thus requiring no pre-defined assumptions to be coded into the algorithms.

- *Pre-chaining* also fixes operations within a template as a prelude to scheduling, however in this case, the operators concerned are all forced into the same time step. This allows small units, such as logic gates or bit manipulations, to be forced into a single time step thereby eliminating unwanted registers and cycle delays.

- *Hierarchical scheduling* performs inline expansion of each CDFG sub-graph as a behavioural template before scheduling, effectively flattening the graph hierarchy. This

results in improved flexibility when scheduling, and allows interactions between sub-graphs not normally considered in the standard algorithms.

Results reported demonstrate the use of complex memory operations in a number of example designs. These emphasise the importance of the features described above to commercial applications. In particular, the ability to fix local timing constraints, especially within multicycle operations, enables synthesised systems to interface effectively with the outside world through well defined, inviolable I/O protocols.

3.1.3 Summary

The systems described in this section all make use of hierarchy in an attempt to enhance the size and quality of the design space available for exploration. Elements of the different approaches are developed in the MOODS implementation of expanded multicycle modules, detailed in Chapter 5. The *behavioural templates* in particular bear a striking resemblance to MOODS *expanded templates*, albeit without the timing constraints, even though the two were developed independently over similar time periods.

3.2 VHDL for Behavioural Synthesis

VHDL (Very High Speed IC Hardware Description Language) [6, 7] is the IEEE standard hardware description language. Originally targeted largely toward simulation, it has in recent years enjoyed increasing popularity as an input language for both RT level [66, 67] and behavioural synthesis [68, 69, 70, 71, 72, 73]. This is mainly due to the proliferation of associated support tools such as simulators [74, 75] and analysis packages [76], coupled with the fact that it is an accepted standard, allowing easy integration of different CAD systems and the development of a transferable skills base.

VHDL models a digital system using *entities* and *architectures* to define its interface and operation respectively. The body of the design, implemented in its architecture, comprises a number of different levels of abstraction incorporating both structure, using concurrent statements (basically a netlist of interconnected components), and behaviour, via *processes* forming algorithmic descriptions built up from sequential statements. The design of VHDL largely as a simulation language dictates many of its features, in

particular its foundation on an event-based model. This leads to a number of problems when implementing VHDL in a synthesis environment, especially at the behavioural level.

Early attempts to synthesise VHDL tended to use a highly restricted subset or re-define some of the more hazardous areas. One such example is the IBM VHDL Design System [66], which only accepts concurrent statements, thereby eliminating any behavioural process blocks. This simply performs VHDL library management and direct technology mapping, thus the language is basically used as a sophisticated netlist description. A more advanced system, VSS [68, 77], implements a larger subset, including sequential processes. The design is first compiled into a data flow graph, which is then processed according to whether the user has specified logic, RT or behavioural level synthesis. This affects the type of implementation and optimisation performed, ranging from basic technology mapping at the logic level, to more sophisticated state assignment at the behavioural level. Within processes however, the VHDL subset is still limited and there appears to be no implementation of *signals* or any form of inter-process communication.

An attempt to quantify the differences between simulating VHDL and the needs of behavioural synthesis is made by Camposano et. al. [78]. This paper presents the results of a study into the feasibility of high-level synthesis from a behavioural VHDL description, focusing on a number of sequential constructs and in particular on the implications of behavioural synthesis on process communication and synchronisation, *wait* statements, *signals* and *procedures*. The authors develop a set of twelve rules for the restriction and interpretation of the VHDL semantics as applied to high-level synthesis, which are summarised in Table 3.1 below.

These rules are, to some extent, applicable to all behavioural VHDL synthesis systems as they mostly refer to inherent simulation properties with no hardware analogue. There are however a number of doubtful interpretations. For example, rule two specifies that sensitivity lists, such as *wait on input1*, are not synthesisable. This is based on the declaration that a sensitivity list detects asynchronous edges, which are un-implementable in synchronous synthesised systems. However, it ignores the behaviour of the sensitivity list which is to wait until a signal changes state before proceeding, thus the sensitivity list should be implemented by continually monitoring the signal until it changes. In effect, the sequential nature of the implementation synchronises the signal edge to the system clock.

1	*Processes* do not execute in zero time (as defined in the standard), but take a number of clock cycles depending on the schedule synthesised. Thus implicit synchronisation of parallel processes cannot be guaranteed and should therefore be performed explicitly via handshaking - while this is generally true, a number of systems do force synchronisation between processes [73, 79] through their control structure.
2	*Sensitivity lists* specify asynchronous edges and cannot therefore be used in behavioural synthesis - this is debatable, see above.
3	Time expressions should be converted into cycle counts and used as local timing constraints - although this is a sensible use of VHDL timings, it entails a significant hardware overhead [79].
4	In general, signal values will be latched - this is defined in the standard, which specifies that signals hold their values until they are modified.
5	Synthesis often implements loops as specified, requiring at least one clock cycle per iteration - this is different from RTL synthesis, where loops are expanded into chained combinational blocks [67, 80], thus restricting them to a small number of fixed iterations. No such limitation need be applied to behavioural synthesis.
6	No recursion is allowed in procedures - while this is generally true, there are approaches which could allow it, to a limited depth [81]
7	*Procedures* may either specify an initial design partitioning or should be inline expanded to flatten the hierarchy - there are a number of ways in which procedures may be implemented, each with their own merits [81]
8	*Assert* statements are for verification during simulation and have no equivalent hardware model.
9	Like asserts, file objects have no hardware analogue.
10	Dynamic variable indexing implies addressable memory.
11	*Access* types require dynamic memory management, which is not generally supported in synthesis systems - this depends, however, on the target architecture, which could provide memory management, eg. a general purpose CPU.
12	Clocking is implicit within processes. If concurrent blocks are also included, the clock signal should be identified using an attribute - this is true of most high-level systems, however some require explicit *wait on clock* statements to define timing [69, 82]; it could be argued that this is not really behavioural synthesis.

Table 3.1 Rules for interpreting VHDL in behavioural synthesis

Although [78] describes the implications of applying behavioural synthesis to the VHDL simulation model from a user perspective, it does not consider practical implementation details applicable to a synthesised architecture model. Most language elements may be readily realised by way of a direct mapping to a design flow representation: *variables*, control constructs (loops, if-then-else), and *procedures* are common to most synthesis languages and have been extensively researched. However, as demonstrated by rule one,

timing within VHDL processes requires a significant departure from the IEEE standard when targeted at behavioural synthesis.

The VHDL simulation model specifies that within a process, time only passes when a *wait* statement is encountered. Any sequential blocks between waits are considered to execute with zero delay, however when synthesised, these blocks (containing any number of complex statements, including data-dependent loops, multiple procedure calls etc.) may take many cycles to execute. This has profound implications as the semantics of those VHDL elements affected by timing (signals and waits) should be forced to conform to the simulation model in order to preserve the behaviour of the design. For example, an assignment specifies the scheduling of an event on the appropriate signal, which is only executed when the following wait statement is encountered. Any further reads of the signal before this wait refer, therefore, to the original value and not the assigned one. During simulation, this behaviour is intuitive since the entire sequence between waits is executed within zero time, however it does not hold for the synthesised system where the new value must be temporarily stored until a wait statement is encountered. This is complicated by the fact that the time delay may be data dependent and therefore not determined until run time.

The behaviour of signals and waits also affects process synchronisation and I/O timing. Since, in the simulation model, time only passes within wait statements, all waits and signal assignments occur simultaneously across all parallel executing processes. Hence VHDL implicitly synchronises parallel processes at wait statements. The behavioural model, however, cannot guarantee that waits in multiple processes will coincide in time, some extra effort must therefore be applied either in the form of explicit handshaking [73] or through a more sophisticated control model [72, 79].

These problems are examined in a number of recent publications that fall into two general groups: the synthesis of signals and processes, and the synthesis of fixed I/O and timing constraints.

3.2.1 Signals, Waits, and Process Synchronisation

Ramachandran et. al [83] investigate the implementation of signals external to processes (ie. without timing), concentrating on their behaviour when assigned to concurrently, in

more than one process. Such a mechanism is useful for handshaking in explicit process synchronisation, particularly prevalent in control-dominated applications. The authors do not consider the implications of the signal timing model on assignments within the processes, indeed they assume that signals operate similarly to variables with assignment occurring immediately. This restriction however has little effect on the external signal behaviour considered in the paper. In order to explain the operation of the VHDL signal types, a number of conceptual hardware model are described for each of the three *signal kinds* : *simple* (the default), *bus* and *register*. These models are then used as the basis for synthesised hardware implementations.

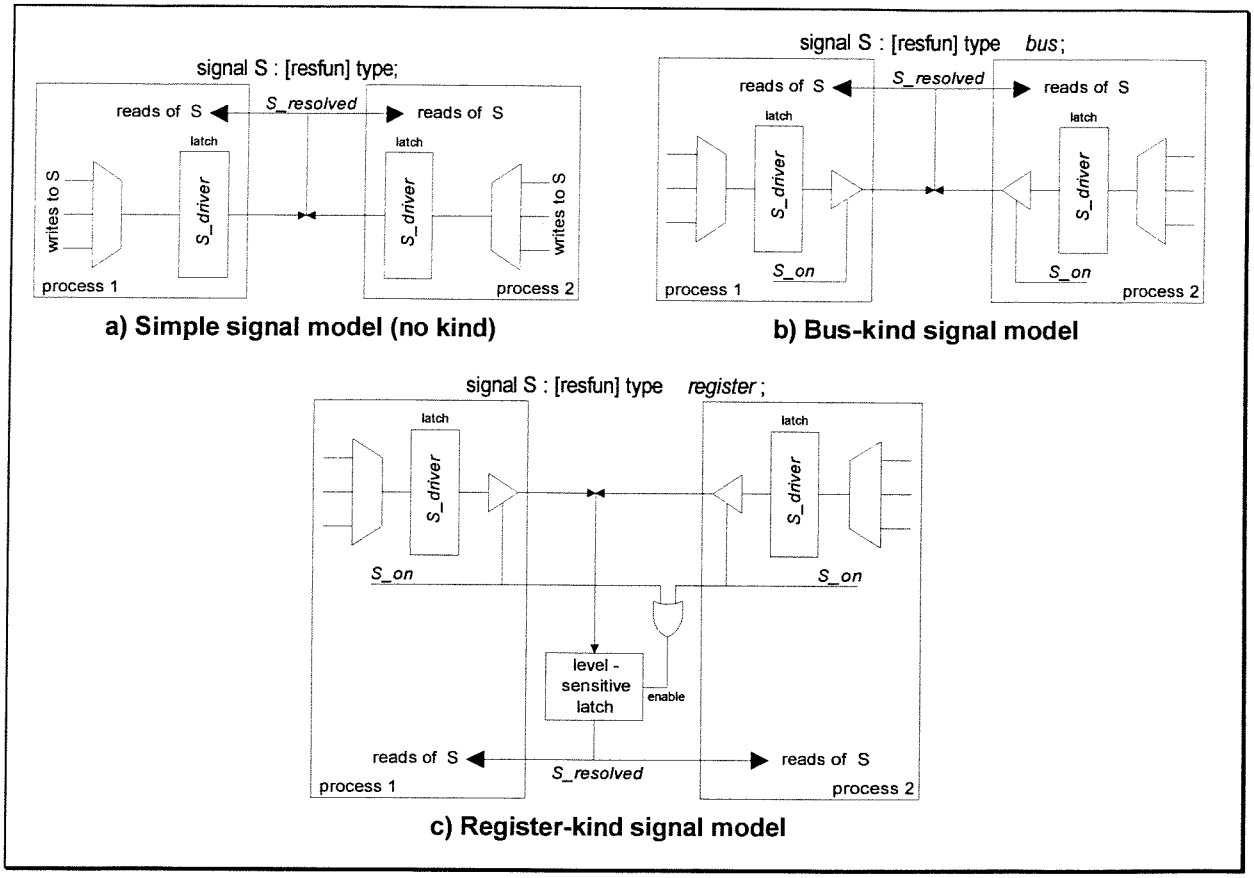


Figure 3.4 Conceptual hardware models for signals external to processes

The conceptual models developed are shown in Figure 3.4 (reproduced from [83]), illustrating how *drivers* in different processes are resolved by a *resolution function* to determine a single global value for the appropriate signal. In each case, the signals within the processes are individually latched to maintain their local values over several clock cycles (see earlier). The figure clearly shows the basic differences between the three kinds: *simple* signals are always connected together via the resolution function, which arbitrates between the different driving values. Signals may not be disconnected. *Bus* signals are

similar to their simple counterparts, but are enhanced by allowing each signal to be disconnected using a *null* assignment. This models a basic tri-state bus, allowing all drivers to be turned off, in which case the resolution function determines the bus value. Finally, *register* signals are the same as *bus* signals, except that when all the drivers are turned off, the last valid signal value is maintained. In the model this is implemented with a level-sensitive latch, which latches when any one signal drivers is enabled. The resolution function is assumed to model the characteristics of the particular target technology, thus in the hardware implementation it disappears, becoming simply an interconnection of wires. This assumption, while modelling a restricted resolution function, need not be generalised for all synthesis systems: resolution functions could easily be implemented as an arbitration block, synthesised like any other VHDL procedure.

The authors proceed to consider the physical implementation of their models. Signals may either be realised as a latch, for individual signals and small arrays or, in the case of large arrays, as external memory. The former are implemented directly from the conceptual block diagrams, assuming a technology-specific resolution function, while the latter are assigned to a memory block with one or more read/write ports. This requires a quantity of extra hardware to arbitrate between concurrent accesses to a limited number of memory port resources. The paper details an algorithm that generates a fixed-priority arbitration scheme, controlling multiple accesses through a simple request/acknowledge mechanism, which is used in place of the original signal read/write. This is similar to the method adopted in the FFT video processor design described in detail in Chapter 7.

The paper concludes with a brief description of a model for entity *ports*, which are identical to signals except that the port adds an additional driver to the resolution functions (for the *in* signal). *Inout* ports are split into read and write lines connecting to the appropriate parts of the signal model (the implementation of *inout* ports and external busing within MOODS is discussed in more detail in Chapters 6 and 7). The authors conclude that the restricted implementation of signal semantics in most VHDL synthesis systems is not a necessity, except in cases where large arrays are required, in which case a more complex resource model needs to be introduced.

While this paper deals with the implementation of signals external to processes, it assumes a simplified model for the internal process timing model. A number of the same authors

tackle this problem in a later paper [79]. This describes a system which pre-processes behavioural VHDL containing arbitrary signal and wait statements, transforming it into a functionally equivalent VHDL description utilising only standard sequential constructs such as variables and loops. These *wait/signal (W/S) transforms* are intended to bridge the gap between the currently supported VHDL subsets, which generally restrict the use of signals implementing them either as variables or not at all, and the full simulation semantics.

The system models signals through the use of additional variables (eg. to store the scheduled result of a signal assignment before the following wait), together with a set of flags to control the external signal model of Figure 3.4. A signal *S* is implemented with up to five variables local to each process, as described in Table 3.2 below (also see Figure 3.4).

S_driver	The current value of the signal driver, as written to by the process.
S_driver_next	The value of the signal scheduled to update *S_driver* at the next wait.
S_update	Boolean flag set to true when *S* is assigned so that *S_driver* will be updated from *S_driver_next* at the next encountered wait.
S_on	Boolean flag set to true if the signal is currently connected (ie. controls the signal tri-state buffer)
S_on_next	Scheduled value of *S_on*, to be updated at the next wait.

Table 3.2 W/S transform variables for modelling signal behaviour

The functionality of the wait statement is modelled using a data flow template based on these variables. Figure 3.5 shows an example wait statement, along with the transformed VHDL output, and the associated data flow graph. This illustrates the implementation of the three wait *clauses* and multiple driver signal model:

- *wait on <sensitivity list>* - wait until a signal in the sensitivity list changes state. This is implemented using a temporary register *S_old* to store the value of the signal on entry to the wait, and is continually compared to the real resolved signal value, *S_resolved*, until a change is detected, or a timeout occurs.

- *wait until <condition>* - wait until an event for which condition is true.The implementation evaluates the condition after the *wait on* loop has detected a change and, if it is true the wait terminates. If the condition is not true, the whole *wait on*

process begins again. Note that the signals appearing in a condition expression are implicitly included in the sensitivity list.

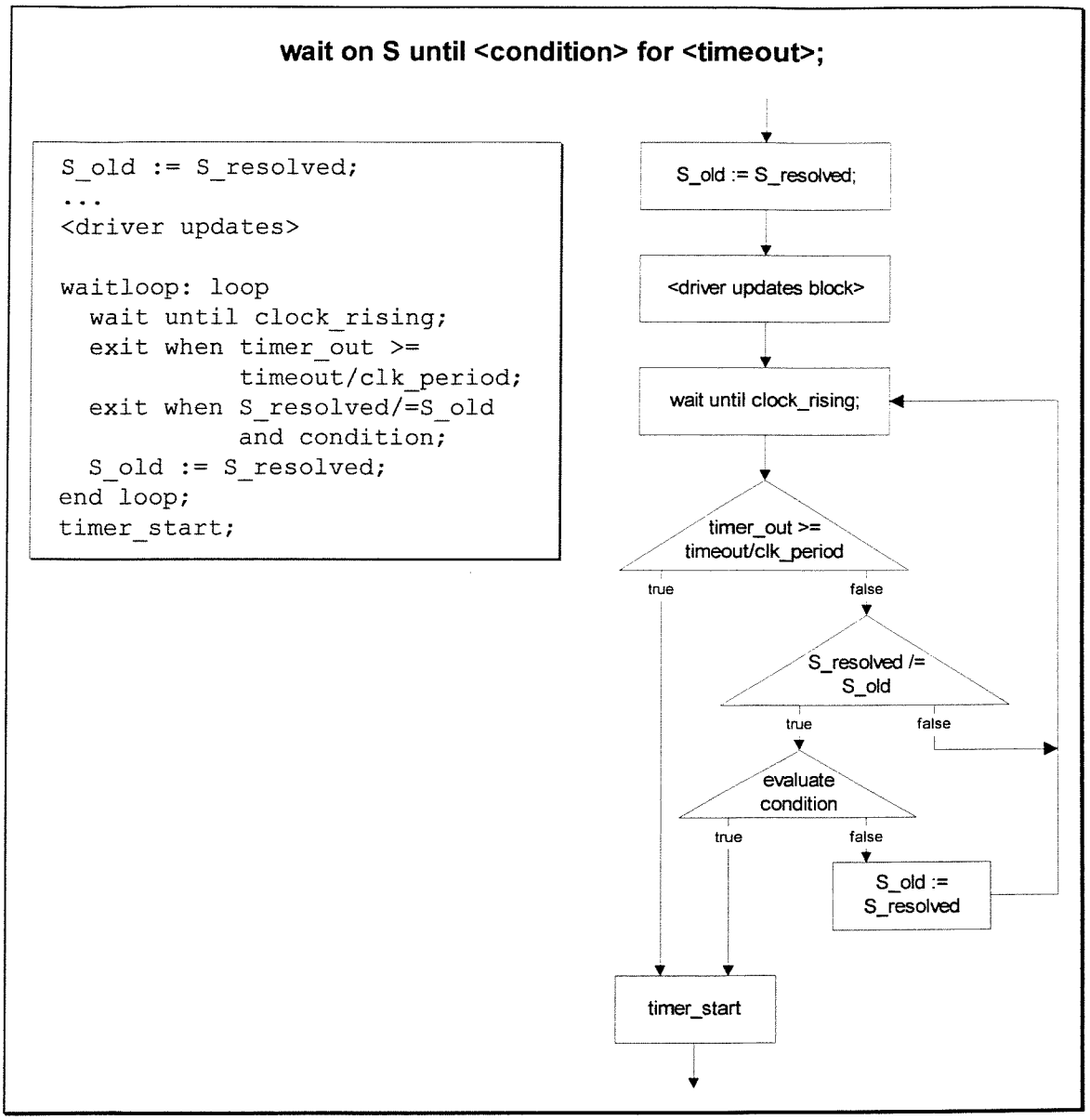


Figure 3.5 Example of wait/signal transformations

- *wait for <timeout>* - waits for the specified amount of time. If any other clauses are present, timeout specifies the maximum time spent waiting. This is implemented by means of a counter, which is reset just after the previous wait statement has completed, and is incremented on each clock cycle. If, within the wait loop, the counter reaches the timeout value (divided by the clock period), the wait will unconditionally terminate. Note that this means that the maximum wait time is the difference between the timeout and the time taken to execute the preceding sequential block (starting from the previous wait). Thus, any user-specified timing behaviour will be preserved, assuming the timeout provides sufficient time for the inter-wait optimised schedule.

The W/S transformations process the input file, replacing the original VHDL with a new description using the appropriate template variables and constructs. The authors complete the paper with a description of simplifications and synthesis optimisations to reduce the complexity of the transformed design. These include the elimination of *S_on* and *S_on_next* if the signal driver is never disconnected, and of *S_resolved* for *simple* signals (the most common form). The authors conclude that the wait/signal transformations enable the full expressive power of VHDL signals and wait statements to be used, within the constraints of the limited VHDL subsets recognised by most existing synthesis systems.

Although the paper describes an effective mechanism for implementing signals and waits, it glosses over a number of very important points. The most striking omission is the lack of any consideration of concurrent process synchronisation. No mechanism is provided to implicitly synchronise processes at waits, thus an explicit handshaking-based method is presumably intended. Furthermore, there is no detailed discussion of the additional hardware overhead required by the transformations. Figures are provided for the increase in size of the source VHDL for a number of examples, averaging at around 250%: a considerable increase. The authors claim that simplification and synthesis optimisation minimises this increase in the final synthesised implementation, however they do not provide any figures to back this up. As will be shown later in Chapters 6 and 7, the issue of the overhead imposed by the signal and wait semantics is of vital importance, especially when targeting limited capacity devices such as FPGAs.

A similar approach to the realisation of signals is taken in DSS [72] and CAMAD [20, 24, 73]. While the implementations may not be quite as comprehensive as [79], they do attack the problem of parallel process synchronisation at wait statements. DSS supports a deferred signal model, with two registers implementing each signal, updated by wait states in the controller. *Wait on* and *wait until* constructs are fully supported as in [79], *wait for* timeouts, however, specify timing constraints used to guide the scheduling operation. Whereas in [79] processes execute independently of each other, in DSS they are implemented as parallel executing FSMs, and are implicitly synchronised at waits by means of a *root FSM*, which also updates the appropriate signal values according to the VHDL simulation semantics. This method is similar to the *unrestricted* model described by Peres et. al. [73] detailed below.

[73] describes the approach taken to process synchronisation in CAMAD. This uses a substantial VHDL subset [24], compiled into an ETPN representation (see Chapter 2), allowing the use of procedures, multiple processes, signals and timing (interpreted as scheduling constraints). The main goal of the research is to preserve the computational effects of the VHDL simulation cycle with the minimum additional cost to the final implementation.

The authors present two methods used to model concurrent process synchronisation. The first, the *unrestricted* model, describes, in effect, a complete hardware realisation of the VHDL simulation cycle. This allows unrestricted use of signal and wait statements within processes, implicitly synchronising them at each wait. Processes are controlled from a separate FSM where synchronisation is achieved with the aid of a supervisor process to control deferred signal assignment and wait states. Whenever a process reaches a wait, it hangs until such a time as the supervisor process enables it to continue. The supervisor monitors the state of each concurrent process and only allows waits to evaluate once all processes are hung in a wait state. This is achieved using a single register per process to indicate its waiting state (ie. the register is set when the process reaches a wait). When all the registers are set, the supervisor updates all signal events, resets the registers and allows the hung processes to continue. In this way all processes are implicitly synchronised at wait statements, as per the VHDL standard.

Whereas the unrestricted model implements arbitrary signal/wait functionality and implicit process synchronisation, the *reduced synchronisation* model is only suitable for "well synchronised" designs, ie. those which formulate process synchronisation through the explicit use of handshaking signals, and hence do not require such a strict implementation of the simulation semantics. This model uses a message-passing mechanism with predefined *send* and *receive* procedures to implement synchronised communication between processes via signals. Signal assignment is achieved using the *send(signal, expression)* command in one process, with the value assigned being read in another using an associated *receive(signal)*. The *send* will hang the generating process until all processes requiring the signal value are executing a receive command. These in turn hang the receiving processes until a value is sent, hence the processes are automatically synchronised at the send/receive positions. Since this is the only mechanism allowed for inter-process signal communication, the deferred signal model is no longer necessary, thus eliminating the extra resources required in the unrestricted model. A VHDL pre-processor

is used to convert a description using the reduced synchronisation model, into a form suitable for simulation of the synthesised behaviour.

This paper presents a detailed investigation of some of the options available when implementing process synchronisation: an area ignored in most VHDL synthesis systems to date, which tend to radically restrict the VHDL subset accepted. The authors raise a number of pertinent points in this, and their previous paper [24], regarding the necessity of fully implementing the VHDL simulation cycle in synthesis, arriving at the conclusion that it may not be sensible, or even desirable, to model every last detail. This is particularly relevant to the work detailed in Chapters 6 and 7, which shows that there are a number of distinct disadvantages to using an unrestricted model.

3.2.2 Controlling Timing in the Optimised Schedule

The final aspect of VHDL behaviour covered in this overview is the local timing of signal events, particularly with respect to I/O timing. This is an important issue as it defines the external behaviour of a system, the specification of which is often (if not always) one of the main implementation requirements [84, 85, 86]. Without the ability to exactly specify interface protocol timing, connecting synthesised circuits to other sections of a large system is next to impossible.

The CALLAS system [69, 82] tackles this problem by requiring the designer to fully specify the desired schedule within a process, through the explicit inclusion of clock edges (using *wait until clock='1'*). These determine the exact position of control states in the final schedule, with no additional steps being included during synthesis, thus the system drastically restricts the scope for scheduling optimisations; indeed the insistence on a WYSIWYS ("What You Simulate Is What You Synthesise") paradigm, more or less relegates synthesis to the role of allocator and module binder. CALLAS also heavily restricts the accepted VHDL subset with the only signals being I/O ports, and no waits allowed other than clock edges, hence no support for concurrent processes.

CAMAD [20] takes a much more flexible approach allowing the designer to specify a number of possible timing constraints between both positions within a process, and events on different signals. The problem of integrating constraint specification into the VHDL description is tackled using a small set of library procedures. These act as place markers

within the process, defining the desired timing constraints relative to anchor points. The set of procedures allows either a minimum, a maximum, a minimum/maximum range, or an exact time difference to be specified. The resultant timing values are added to the ETPN representation in the form of *timing control arcs* on the control graph connecting the two relevant states with a maximum and minimum specified time [87].

The specification of timing constraints across process boundaries is also implemented providing that the reduced synchronisation model [73] is used. This utilises a VHDL *assert* statement to activate a *time constraint function* (using the same timing types as above) defining the desired timing between two events on any pair of signals. For example, the statement "*assert range_assert(a'TRANSACTION, b'TRANSACTION, 100 ns, 800 ns)*" specifies that the time between an event on signal *a* and an event on signal *b* must be between 100 and 800 nanoseconds. The use of the reduced synchronisation model is necessary to explicitly define the interaction of processes at a higher level than the standard wait/signal combination allows. This is important as there is no explicit correspondence between signal assignments and signal reads in standard VHDL, thus it is not generally possible to identify where to place the timing dependency arcs in the control graph.

Synthesis proceeds, maintaining the constraint ranges defined by the user. Afterwards, the actual synthesised timing figures are back-annotated into the behavioural description via the original timing constraint functions [88]. This allows the designer to simulate the behaviour with the exact timing information, and to iterate the design loop, refining the constraints as desired.

Whereas CAMAD takes a broad view of the timing problem, allowing the user to specify almost any timing relationship possible, the method used in the CADDY system [89, 90] separates the interface from the algorithmic part of the design by abstracting signal operations, and hence all I/O protocol handling, into pre-defined procedures. Since signals are not permitted in the behaviour, the only form of wait statement accepted is the *wait for* clause. These are used to specify the partial scheduling of blocks of sequential code, effectively determining the number of control steps used. This is a little more flexible than the CALLAS approach, however, as it is not necessary to specify every single clock cycle.

The interface procedures are not synthesised using behavioural synthesis but are instead treated as component modules, which implement a pre-defined I/O protocol, eg. for bus timing or memory access. These components allow asynchronous access to external ports, while communicating synchronously with the synthesised behavioural section, thus the I/O timing is afforded finer granularity than the system clock allows. Each interface component is associated with a number of external ports forming the I/O interface, eg. a RAM would require address, data and read/write control. These ports constitute a single *logical port* from the point of view of the behavioural description, thus simplifying a complex I/O interface.

This method allows the synthesis process full freedom to optimise the scheduling within the gross constraints specified, while preserving the local timing behaviour of the interface protocols. However, since no signals are permitted within the process all I/O must use an interface procedure, thus any application-specific input or output requires additional procedures to be written, even in the simplest of cases. This makes communication with the outside world rather cumbersome and would be better implemented using a combination of VHDL signals and the logical port approach. The system also limits the complexity of the behavioural description, allowing only simple loops (no *next* or *exit* statements) and more importantly, limiting the designer to a single sequential process.

3.2.3 Summary

The research detailed in this section demonstrates how the desire for simulation/synthesis correspondence can lead to a substantial increase in the complexity of the synthesised hardware model. This is a direct result of the complex semantics of VHDL, which are primarily geared towards exact timing simulation, and the lack of any widely accepted behavioural synthesis subset. Many of the concepts discussed are embodied in the enhancements made to the MOODS VHDL compiler, described in Chapter 6. However, none of the above studies quantify the substantial overheads and drawbacks associated with a full realisation of the VHDL simulation model: as will be shown in Chapters 6 and 7, this can have a significant impact on the final size and performance of the design, in some cases posing the question, "can we really afford such a comprehensive VHDL implementation ?".

Chapter 4

The MOODS Synthesis System

This chapter details the principles and operation of the current incarnation of the MOODS synthesis system. The entire system comprises a number of separate programs performing various tasks in the synthesis of a VHDL behavioural description down to a hardware FPGA or ASIC based implementation. Generally speaking, "MOODS" encompasses all of these ancillary components, however throughout this text, unless otherwise stated the name MOODS refers solely to the central synthesis block. Figure 4.1 illustrates the gross system data flow showing the connectivity of the major sub-components. These split into five main sections, identified in the figure:

1. The VHDL behavioural description is initially pre-processed by the source level optimiser [13, 14, 91]. This performs VHDL-specific language-level optimisation geared towards minimising either area or delay.

2. The optimiser output, in the form of an optimised behavioural VHDL description, is then compiled into an intermediate form, the ICODE, suitable for direct input into the main MOODS synthesis package.

3. The central component in the entire system is the original MOODS data and control path synthesis engine [3], which itself decomposes into several sub-blocks as depicted in Figure 4.2. This performs scheduling, allocation and module binding according to user-defined optimisation objectives, and outputs a structural VHDL netlist suitable for the targeted logic synthesis and layout tools. The internal structure of MOODS revolves around a representation of the control and data flow through the synthesised system, initially created by a direct translation of the ICODE input to form a naive maximally serial implementation. Synthesis then proceeds using a set of local transforms to optimise sections of the control and data paths in a similar vein to CAMAD [19], with the selection and targeting of transforms performed by an optimisation algorithm guided

by a global *cost function*. The cost function evaluates the current design configuration with respect to the optimisation goals on area, delay and static power consumption using technology-specific information maintained in a separate module library database. This binds each data path unit to a physical module implementation and is interrogated by estimators to determine module characterisation data based on the targeted low-level tool. The use of low-level performance estimates fed up from the module library, similar to the approach taken in BUD (see section 3.1), enables MOODS to make technology-dependent trade-offs, while maintaining overall technology (and layout system) independence within the bulk of the system. Once optimisation is complete, the internal representation is converted into a technology-specific netlist using module interface information stored in the module library.

4. The final stage in the design flow is low-level logic optimisation and technology mapping which utilises third-party placement/routing tools, such as Cadence Synergy [92] or Cypress Warp [93], to transform the VHDL structural netlist into an ASIC or FPGA layout.

5. In addition to the structural description, MOODS allows the user to save a complete snapshot of the current internal design representation in a *design data structure* file, which may be restored at a later date as a starting point for further optimisation and manipulation. It also forms the input to a number of post-processing tools, which perform various analyses on the design, yielding statistics on module usage, activity times etc.

The following sections detail some of the major aspects of the system: the initial optimisation and compilation into ICODE, the internal design representation, estimators and transformations, and the global optimisation algorithms currently implemented.

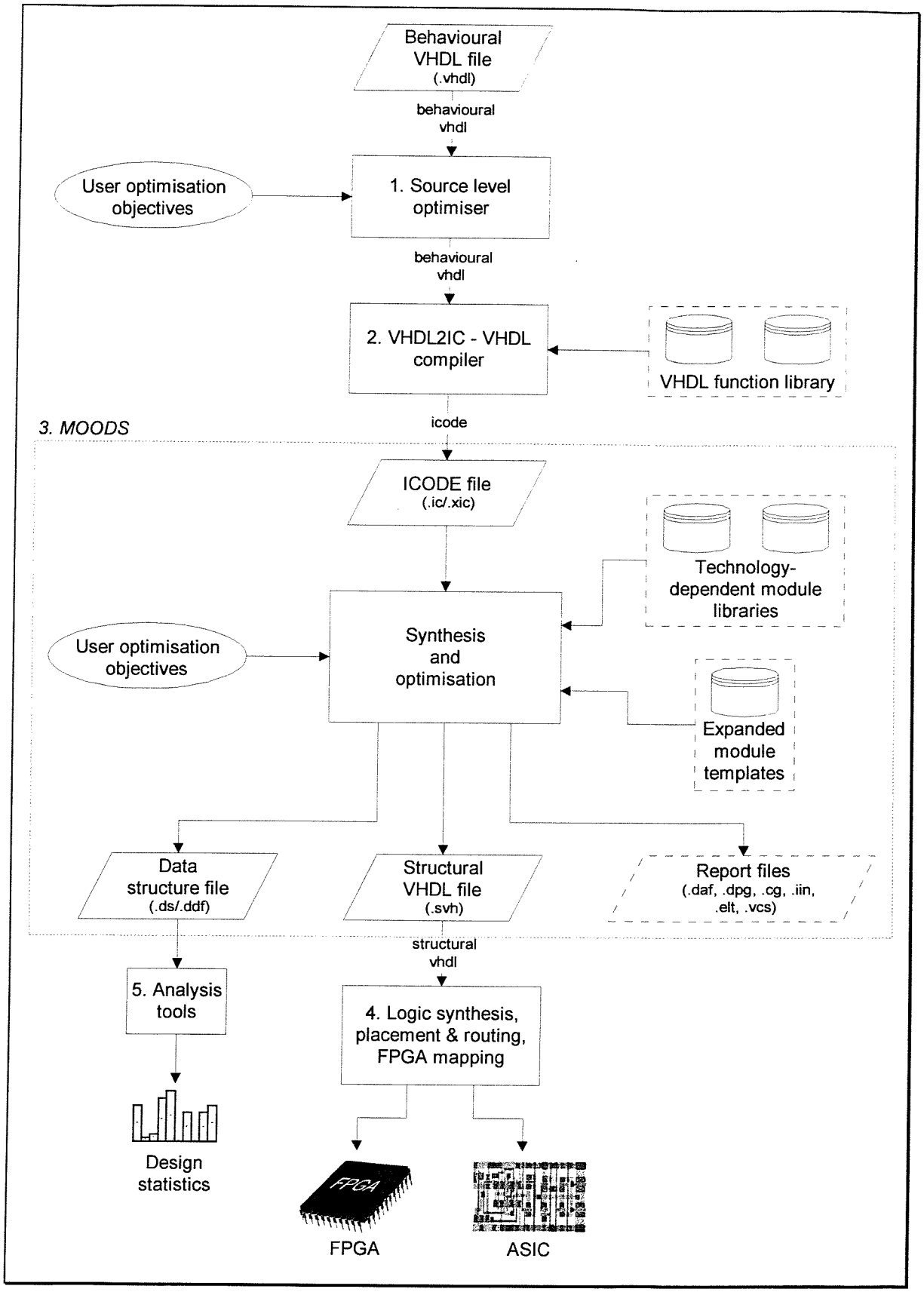


Figure 4.1 Complete system data flow

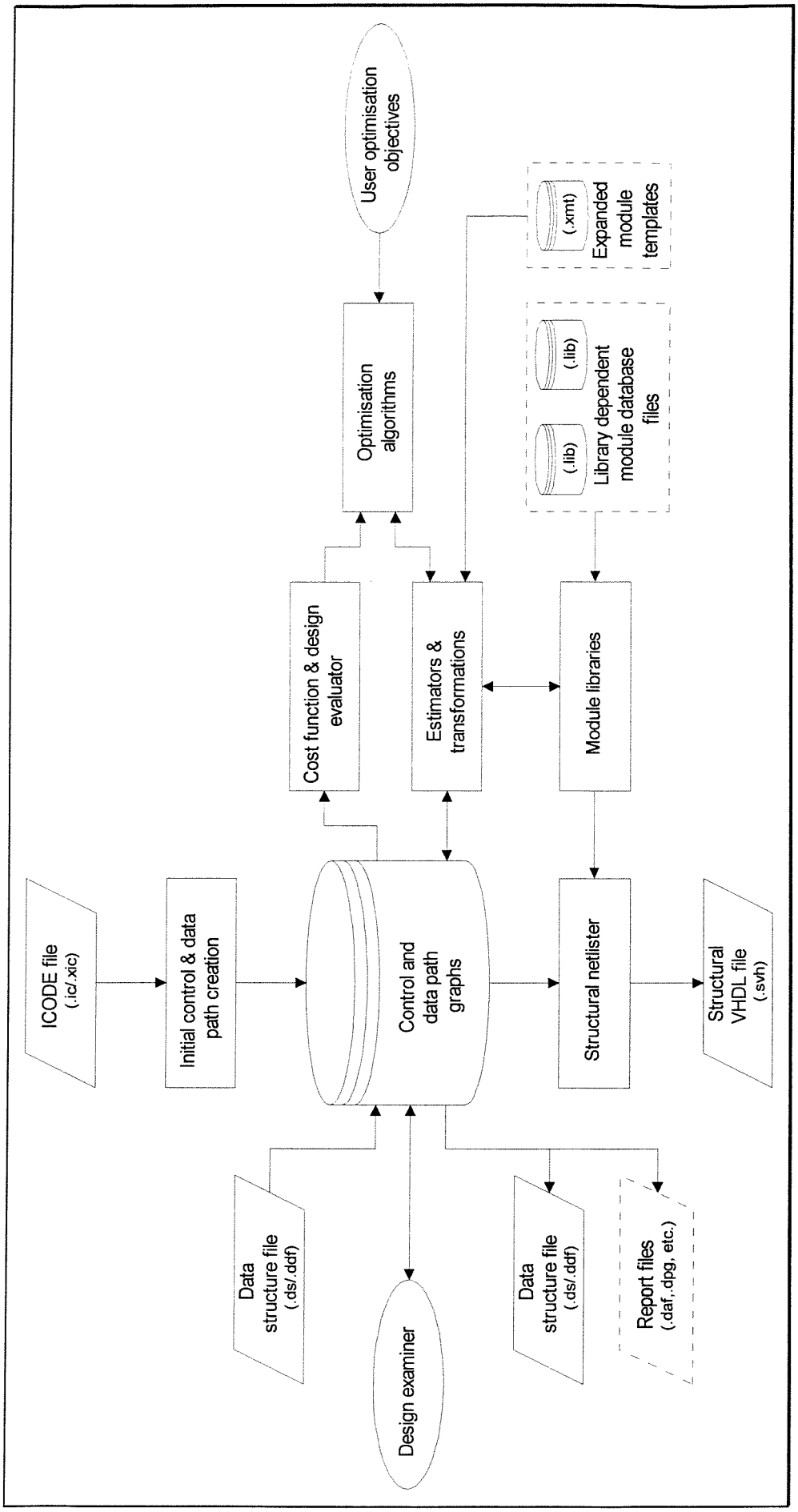


Figure 4.2 MOODS internal block structure

4.1 VHDL Optimisation and Intermediate Code

The main input to the entire synthesis system is a behavioural description written using an extensive VHDL subset (see Chapter 6). Prior to its optimisation within MOODS itself, this description passes through two separate pre-processing stages: the source-level optimiser, and the VHDL to ICODE compiler, VHDL2IC.

4.1.1 Source Level Optimisation

The source-level optimiser [13, 14] performs language-level optimisation on the behavioural source code. It implements a collection of twenty nine transforms which may be applied directly to a parse tree derived from the VHDL input. These facilitate a combination of area, delay or power optimisations, and are thus selected for application according to a user-specified optimisation priority (this need not necessarily be the same priority used for the later synthesis stage). Transformations are performed on the parse tree in order of the most significant first, thereby increasing the size of the data set for the smaller optimisations. The output of the source-level optimiser is also a behavioural VHDL description, modified according to the applied transforms. It may therefore be used as a standalone pre-processing stage for any suitable VHDL synthesis system, however in MOODS, it generates the input for the VHDL to ICODE compiler.

Many of the transforms are analogous to those normally encountered in traditional optimising software compilers [94], such as *loop unrolling*, *inline procedure expansion* and *common sub-expression elimination*. An additional number targeted specifically towards synthesised hardware description languages are also available. These exploit the flexibility provided by synthesis to customise the target architecture (impossible on a general purpose CPU), taking advantage of the way the language is interpreted, and the types of lower-level optimisation performed during scheduling and allocation. For example, *operation dependent algebraic simplification* attempts to minimise the set of different functional units required by transforming lesser-used operators into functions of those already present in the design. This increases the potential for module sharing during synthesis and is thus targeted towards area optimisation.

The optimisations performed have a potentially large effect on the final hardware implementation, by virtue of their high level of abstraction (ie. behavioural). Results reported for the source-level optimiser [14] suggest speed and area improvements in the region of 40%. Note that in the remainder of this thesis, the source-level optimiser is considered a stand-alone pre-processing tool with VHDL2IC forming the principal interface between a behavioural VHDL description, and MOODS (see below).

4.1.2 The Intermediate Code

As mentioned earlier, MOODS does not directly process the behavioural description at the source language level, but instead reads an intermediate code (ICODE), similar to a traditional assembly language, along with additional sequencing information. This forms a buffer between MOODS and the high-level description, allowing the use of several different languages (SCHOLAR [95], ELLA [96, 97], VHDL) each with their own ICODE compiler. The ICODE was originally developed as the output from the SCHOLAR language compiler [95, 98], describing functionality, sequencing and connectivity at the register-transfer level in a form suitable for direct mapping to a cell library, and employs simple two input operations to ensure technology independence (see Appendix B for details of the ICODE format).

The VHDL2IC compiler converts a VHDL subset into an ICODE representation, suitable for processing by MOODS. VHDL subprograms are translated into a set of ICODE *modules*, with the main architecture body implemented in a special *program* module, executed on a system reset. Each module comprises a set of numbered ICODE *processes* (not to be confused with VHDL processes) containing a single *instruction* together with an associated *activation list*. The sequencing of operations is based on a Petri-net style token passing mechanism, in which an instruction is only executed once it has been *activated* by another; the activation list specifying which instructions are to be executed once the current one has terminated.

All instructions operate on ICODE *variables*, which are either explicitly declared in the module header, or are temporary variables generated by the compiler. Explicit variables implement I/O ports, registers, aliases (bit-slices) and memory blocks, and represent the various signals and variables in the original VHDL input. They are specified in terms of

their bit-width and implementation (register, memory or alias). On the other hand, temporary variables are not declared, being created by the compiler on the fly during the decomposition of complex source expressions with their bit-width inferred from their connectivity. All ICODE variables represent simple bit vectors directly implemented by either registers or memory, therefore any abstract data types in the original description must be encoded during the compilation process.

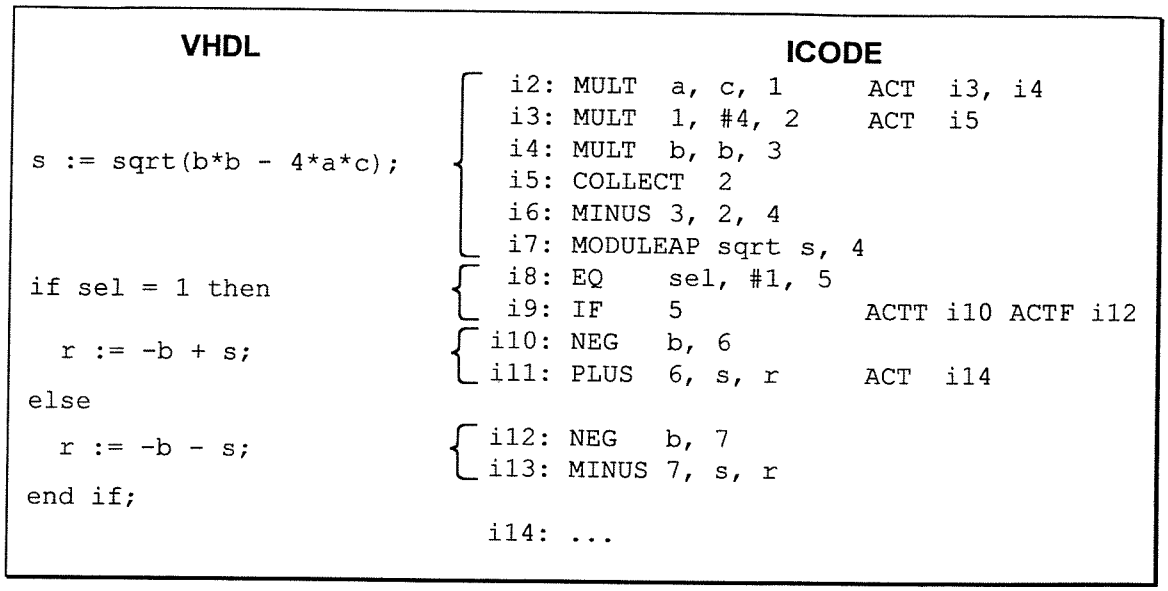


Figure 4.3 VHDL and ICODE for the quadratic equation solver

Figure 4.3 shows an ICODE implementation for a fragment of a quadratic equation solver example, illustrating some of the key features of ICODE, and its translation from VHDL:

- Each line is an ICODE *process* with an associated *process number* (often called the *instruction number*), which describes an ICODE *instruction* and its associated *activation list*. Where an activation list is omitted, the next instruction in the sequence is assumed to be activated automatically.

- Instructions are of the general form:

```
OPERATION <input1>, <input2>, <output>  ACT <activation list>
```

- Temporary variables, identified in the figure using numeric variable names, are only used once with their size inferred from their connectivity. For example, in instruction *i4*, temporary variable *3* will be twice the width of *b* to accommodate the full multiply result range.

- Concurrency is achieved simply by activating more than one instruction. Concurrent branches are joined using a COLLECT instruction which only terminates once a specific number of activations have been received. For example, instruction *i2* activates both instruction *i3* and *i4* which execute in parallel, the two threads of control merging on instruction *i5*.

- Conditional branches are achieved using IF statements with two activation lists, one for each of the true (ACTT) and false (ACTF) conditions. Thus instruction *i9* will pass control either on to instruction *i10*, if variable *5* is true (non-zero), or instruction *i12*, if *5* is false.

- Function calls are implemented with a MODULEAP instruction which activates the appropriate sub-module. The instruction terminates when the called module has completed. In the figure, instruction *i7* executes a separate sub-module to calculate the square root of *s* and returning the result in temporary variable *4*, at which point the main flow of execution continues.

Further details on VHDL2IC and the enhanced implementation of VHDL signals and processes may be found in Chapter 6, while Appendix B details the ICODE syntax.

4.2 MOODS Target Architecture

MOODS targets a distributed control plus data path architecture. This is a general purpose architecture which restricts the design space as little as possible, enabling MOODS to synthesise a whole range of applications with different data flow and control biases.

The ICODE instructions are implemented by arithmetic and storage units within the data path, and are executed (ie. scheduled) by states generated by the controller. The data path is implemented from a distributed network of functional modules, registers and multiplexors with the flow of data directed through the control of register loading and multiplexor selects via control signals. All modules are assumed to be purely combinational logic blocks, which are executed by loading the connected output registers at the end of the executing clock cycle. The controller itself is a synchronous non-deterministic FSM, where each state conceptually executes one or more ICODE instructions. Physically, the

controller is implemented using a one-hot state encoding with a direct mapping between control states, and controller registers. The internal representation however, makes no assumption as to the physical realisation, and it is entirely possible, therefore, to use alternative state encodings, or even a micro-coded architecture.

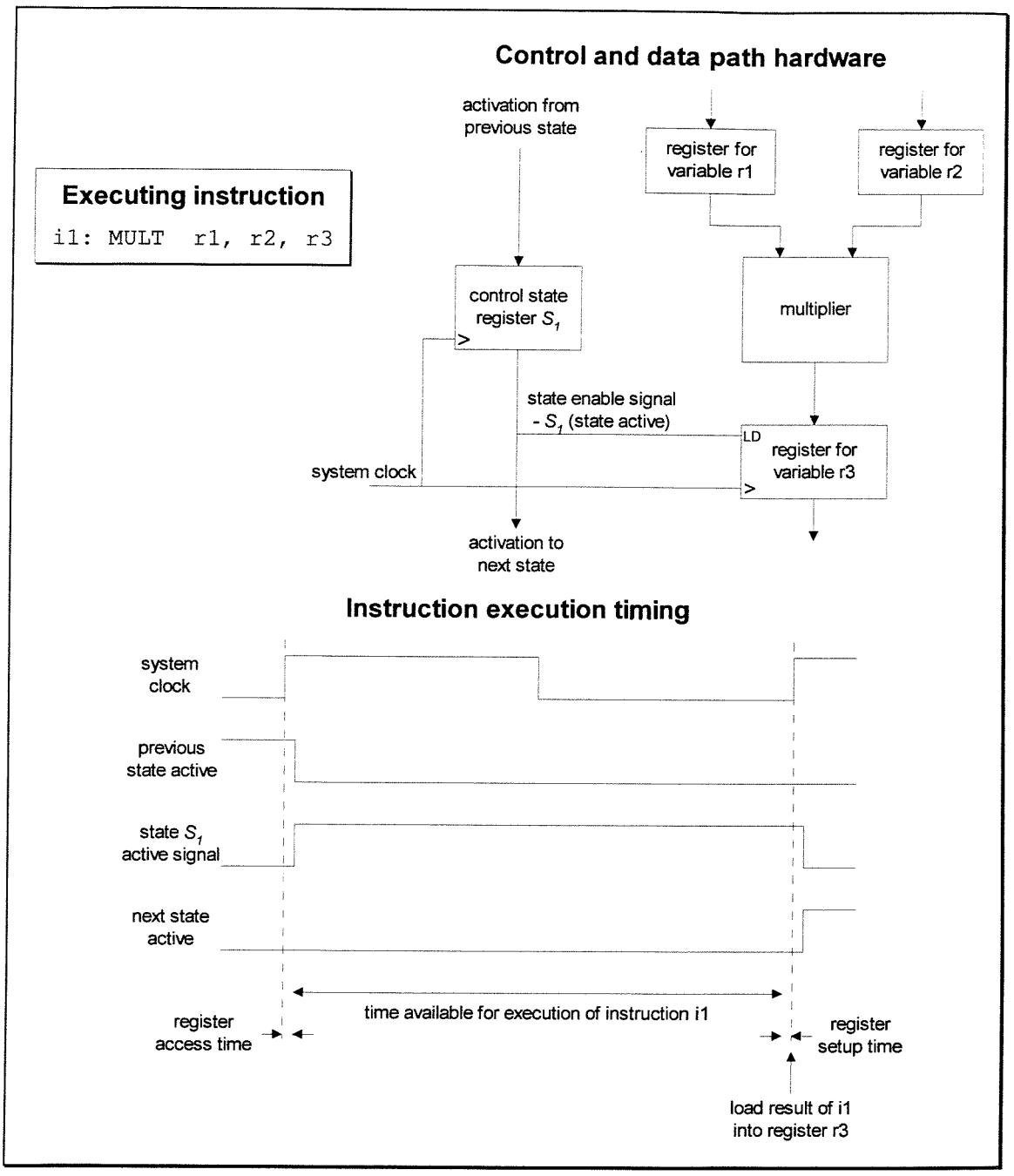


Figure 4.4 Target hardware architecture and instruction execution timing

A section of the controller and data path for a single example instruction is illustrated in Figure 4.4 showing the interconnection between the control and data part, together with the timing of the executed instruction and its associated output register loading. Some points of particular note are:

- Each control state is implemented by a special control register which generates signals used to drive the next state logic and control the data path.

- The system clock connects to every register and control cell in the circuit, with loading controlled via individual load enable inputs for each register bit.

- Execution of instructions is controlled via the load enable pins on the instruction output registers, activated by the executing control states. These cause the instruction result to be loaded into the registers at the end of the executing clock cycle (ie. the following rising edge).

- The single phase clocking scheme implemented by MOODS simplifies the synthesis process and is readily suited to common cell libraries and FPGA architectures.

4.3 Internal Design Representation

The internal representation is the central foundation around which the entire system is built (see Figure 4.2). It forms a coherent multi-level description linking both behaviour and structure [23], capable of describing a design at all stages throughout the synthesis process. MOODS models the control and data paths as two separate graphs, linked together via implementation links and control equations. This structure directly models the target architecture so that generation of the final netlist is simply a matter of outputting the graphs in a suitable format, in this case structural VHDL.

Figure 4.5 shows the initial control and data path graphs created by MOODS for the quadratic equation solver of Figure 4.3. These are generated directly from the ICODE with one control state per instruction, one register per variable and one functional unit per operation. It should be noted that at this stage, since each data path element is activated by only one control state, it is possible to superimpose the scheduled time steps over the data path graph, hence the combined view of Figure 4.5. During synthesis however, the sharing of functional units and registers means that there is no longer such a direct correspondence between the two graphs; a combined view of the entire design is therefore no longer feasible.

4.3.1 The Control Graph

The control graph defines the sequence of execution of the ICODE instructions where each graph node represents a single control state, during which a number of instructions are executed. Input and output arcs describe state transitions conditional on signals generated by the data path. For example, in Figure 4.5, the transition from state S_9 to state S_{10} or S_{12} is conditional on data path variable *s42*.

The instructions executed during a state are stored within the graph node as an instruction list. Acyclic sub-graphs within this list partition it into groups of dependent instructions (the result of operator chaining during scheduling). Each group is numbered, thus instructions with the same group number are sequentially executed in the same control state and are dependent upon each other, while instructions from different groups are independent, and hence execute concurrently. Within a group, the dependency graph specifies the order of execution. This is used to estimate the minimum delay required to execute the instructions in the control state. Figure 4.6 illustrates the execution of a state containing two concurrent instruction groups. This structure would result from a delay optimisation of states S_2, S_3 and S_4 in Figure 4.5. Note that the dependency lists do not represent any physical control constructs but simply describe the data dependencies formed within the data path due to chained operators.

Characterisation data (ie. inherent operator delay) for each instruction is obtained via tags linking instructions to the data path unit by which they are implemented. These in turn are linked to modules within the module library that feed up technology-specific performance data. Thus the estimated delay bears a close resemblance to the actual physical implementation, with the resulting state delay used to determine the maximum allowable clock rate for each state to execute.

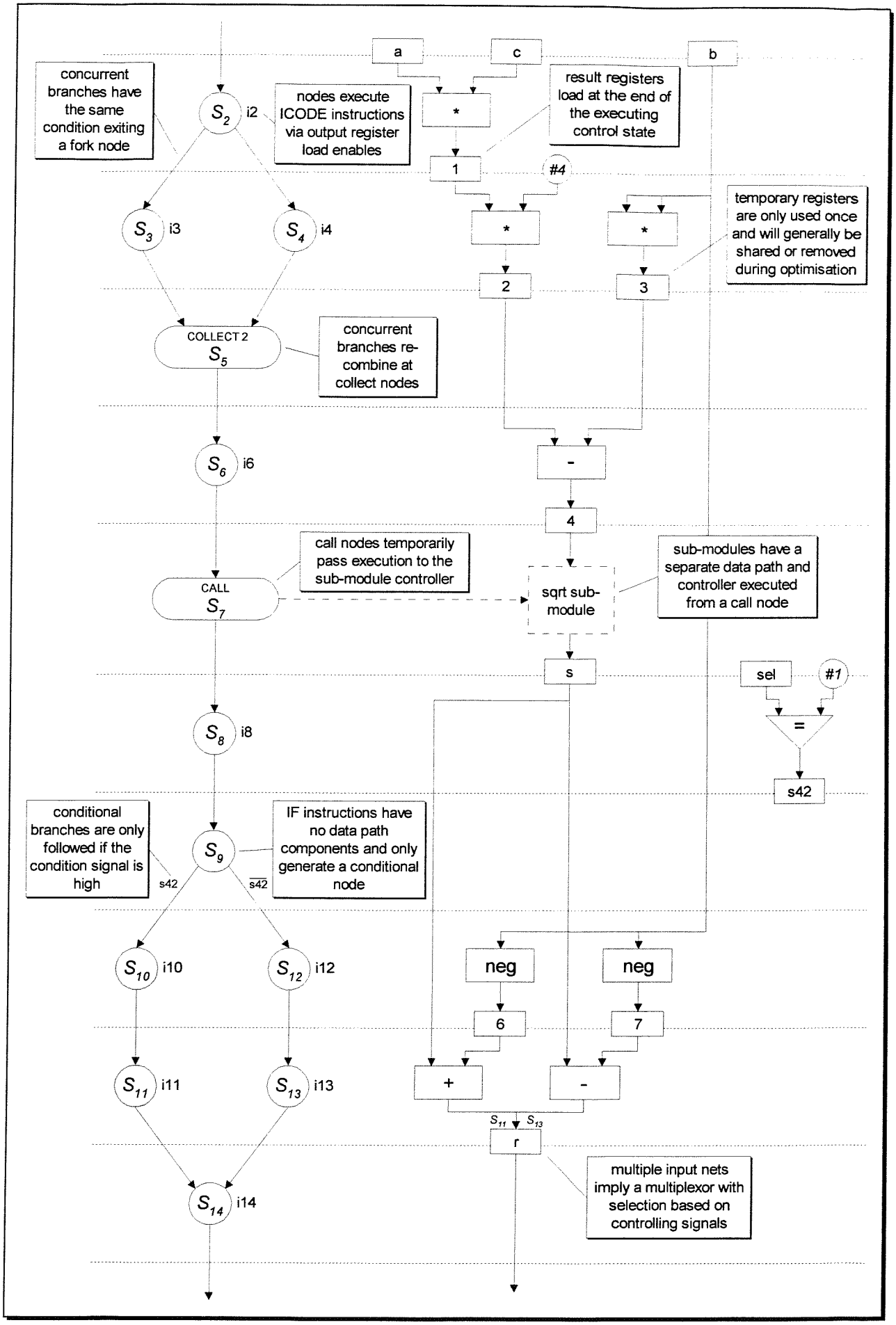


Figure 4.5 Initial control and data graphs for the quadratic equation solver

Control nodes fall into six basic classes according to their behaviour and connectivity:

- *general* nodes (eg. Figure 4.5, state S_6) have a single unconditional input and output arc and represent a simple sequential control state taking 1 cycle.

- *fork* nodes (eg. state S_2) feature a single input and multiple unconditional output arcs. This simultaneously activates all successor states forming the root of a concurrent branch.

- *conditional* nodes (eg. state S_9) have multiple conditional output arcs forming alternative paths through the control graph, and are the result of conditional IF and CASE instructions. Note that since all control arcs are explicit (ie. a state transition must occur on every clock cycle), the graph is incorrect if a situation can occur where no output arc is enabled.

- *dot* nodes (eg. state S_{14}) have multiple input arcs, any of which may activate the state, thus they represent the re-convergence of conditional branch sections into a single control node.

- *collect* nodes (eg. state S_5) represent ICODE COLLECT instructions and are the concurrent branch equivalent of dot nodes. A collect state will not terminate until a fixed number of input arcs (which may be less than the total number of inputs) are activated, thereby synchronising a number of concurrent branches. Note that unlike all the previous types, a collect node may hang while waiting for the correct number of input activations (ie. it has an implicit internal feedback arc).

- *call* nodes (eg. state S_7) implement the ICODE module call mechanism (MODULEAP instructions). Sub-modules are implemented as a separate control and data path graph. When a call node is encountered, control passes to the appropriate sub-graph to execute the module body. The call node itself hangs until the sub-module terminates, at which point control is passed back and the main graph continues.

Scheduling optimisations tend to merge control nodes together forming composites of the above types. This does not, however, apply to call nodes which are never shared.

In addition to the main control features described, a number of ancillary data structures are also associated with the control graph. These detail static information such as mutually exclusive instructions and the minimum feedback arc set. They are used throughout the synthesis process and are maintained centrally for computational efficiency. More details may be found in reference [3].

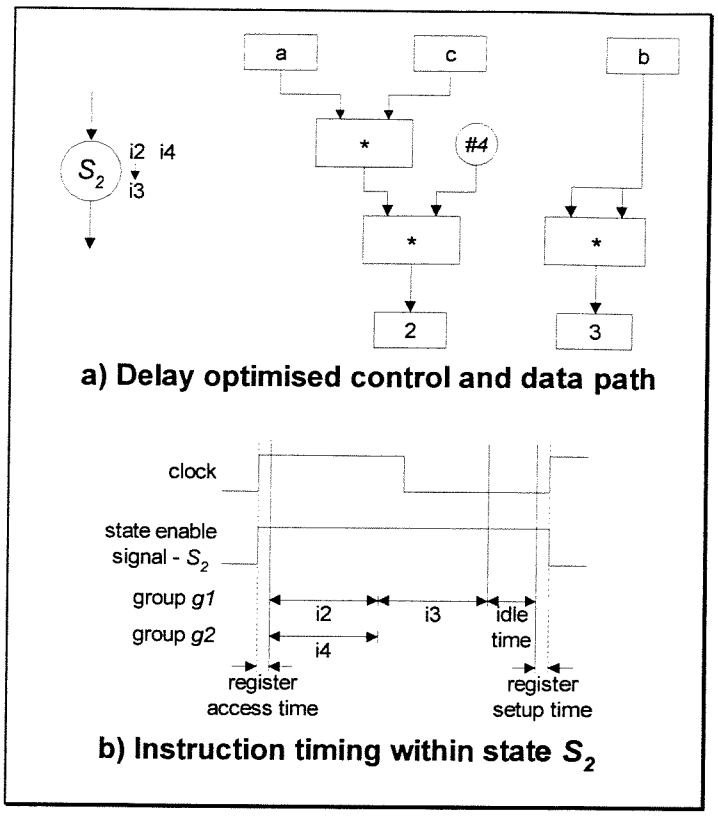


Figure 4.6 Execution of multiple instructions in a single control state

4.3.2 The Data Path Graph

The data path graph describes the network of functional (adders, multipliers etc.), storage (registers), and interconnect (multiplexors) units, which implement the functionality of ICODE instructions. The flow of data through this network is directed via control signals generated from the controller node enable signals, activating register load and multiplexor select inputs (as shown in Figure 4.4). All functional units are assumed to be implemented as purely combinational logic blocks with a delay and area specified in real physical units taken from the module library.

The data path is created directly from the ICODE instructions and variables thus:

- One storage unit (register) is initially created for each ICODE variable. The type of storage and control pins required depend on the operations performed on the variable. For example, a variable which is only reset and incremented ($a := 0$ and $a := a + 1$) is implemented by a counter register with a reset input, thus the add operation does not require a special adder unit. On the other hand, a variable which appears in a multitude of different instructions will be implemented by a basic register, with each operating instruction performed by a separate functional unit connecting to the register input (via a multiplexor). During synthesis, register sharing and bypassing reduces the number of physical storage units required.

- Each ICODE instruction is implemented either by a functional unit for arithmetic and logical operations, or by a set of register interconnections and control signals for instructions such as MOVE, or COUNT. In each case, data path connections are made via nets joining the appropriate units, together with load enables for each of the output registers, which are set according to the activating control state for the instruction. For example, in Figure 4.5 register *3* is loaded by instruction *i4*, thus its load enable input connects to control signal S_4 (the state activation signal). In addition, instructions may be conditional on other signals generated by the data path, which will also control the load enable input. These are the result of merging conditional states as discussed later in section 4.4.1.

  Initially, all inputs and outputs to a functional unit connect directly to registers (see Figure 4.5), however during optimisation, operation chaining bypasses registers directly coupling functional units together as illustrated in the example of Figure 4.6. The control graph and data path are linked together via a set of tags in the form of bi-directional *implementation links* specifying which functional units implement which ICODE instructions (and vice-versa). These enable MOODS to efficiently identify associated elements of either graph.

- Each data path node is a generic functional block performing the appropriate ICODE operation (or operations for ALUs), which is implemented by a specific physical module held within the module library. All characterisation data for a unit, such as area or delay, is obtained via the module library using a unique *module number* which references a particular implementation within the library. The separation of the generic and physical aspects of data path elements enables a high degree of technology independence without

sacrificing the accuracy of performance information. Further details on the operation and implementation of the module library may be found in Chapter 5.

- Control signals between the controller and the data path are represented as boolean logic equations rather than a network multi-level logic gates. This abstracts the control signal generation mechanism, which could be implemented in a variety of ways including random multi-level logic, ROM lookup, or combined with the control graph into a microcoded controller. At present, the signals are output as simple VHDL logic expressions, leaving boolean optimisation and technology mapping up to the low-level logic synthesis tools.

- Multiple connections to a single unit input pin imply the existence of an multiplexor within the data path. These are not explicitly instantiated until the final netlist is output, thereby simplifying the data path structure. Initially, multiple inputs only occur on storage units where a variable is written to by several instructions, however, functional unit and register sharing during synthesis results in additional (implicit) multiplexors being created. Input selection is represented by labels on data path nets (eg. register *r* in Figure 4.5), describing which ICODE instruction (and therefore which control state) activates the input. When a multiplexor is finally created, its select inputs are driven by the enable signals from the appropriate activating control nodes for the instructions.

During optimisation, unit sharing and register bypassing means that the direct correspondence between scheduled time steps and the data path in the initial implementation (see Figure 4.5) is lost. Over time, the net effect of synthesis optimisations tends to render the complete data path graph unreadable for all but the simplest of systems. The approaches taken to tackle this problem are two-fold:

1. The data path may be represented with one block per instruction (rather than per functional unit) where shared units are explicitly identified as such via markers. The resulting graph represents a hybrid of the data path structure and the instruction schedule, and may be split into time steps much as in Figure 4.5. An example can be seen in Figure 4.7a, which shows the first section of the quadratic equation solver when optimised for area (ie. all three multipliers shared using one physical unit). Here, each operator in the data path is implemented by a single shared unit, *DP1*.

2. The second approach shows the true data path structure, reducing the complexity by considering only a localised sub-section, based around a particular set of functional units or registers, as in Figure 4.7b.

A combination of the two views leads to a clearer exposition of the data path structure and data flow, ie. considering both parts of Figure 4.7 together.

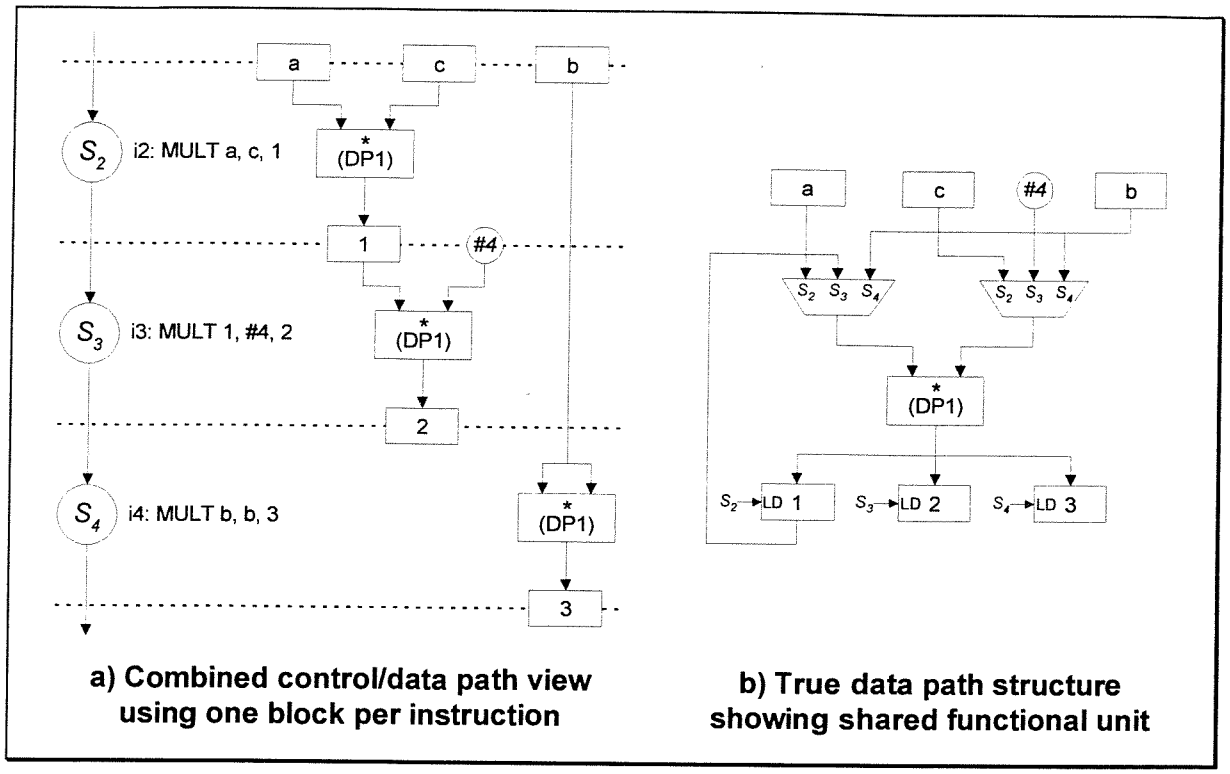


Figure 4.7 Multiple data flow and data path views of shared quadratic multiply

4.4 Transformations

MOODS formulates the synthesis process as the iterative optimisation of a naive maximally serial implementation by means of local semantic-preserving transformations. This allows trade-offs to be made between the various synthesis sub-tasks by simultaneously performing scheduling, allocation and module binding. Throughout the process, the internal representation describes a fully bound implementation, making use of low-level characterisation information from the module library to provide accurate estimates for circuit performance. These figures are used by the optimisation algorithm to guide the selection and targeting of transformations in such a way as to move the implementation through the design space towards the cost objectives specified by the user. The use of accurate low-level figures (delay in nanoseconds, area in square microns) enables MOODS

to take a more sophisticated, and hence less restricted, view of the design space than the traditional unit count and number of control steps measures [26].

At present, MOODS implements a set of fourteen transformations performing a variety of scheduling, allocation and binding optimisations. Each one is *complete* in that the design represents a valid implementation both before and after its application. To provide for a multiplicity of optimisation strategies, a broad set of general transformations is provided. These allow for both design improvement and degradation, necessary to allow hill climbing out of local minima during optimisation (see later). The inclusion of inverse transformations means that at no stage are any prior optimisations irreversible. This overcomes the problems associated with premature binding encountered in other systems which perform synthesis sub-tasks in a fixed sequential order (see Chapter 2).

The selection and application of transformations takes place in four separate phases, illustrated in the flow graph of Figure 4.8 showing the MOODS optimisation loop, which forms the core of the iterative synthesis process. Each stage is implemented by a separate procedure for every transformation, called via *test_trans*, *estim_trans*, and *perf_trans*, to select the appropriate routines according to the data in the *transform data structure*. Thus, implementing additional transformations is simply a matter of creating test, estimate and perform procedures, and adding suitable selection code into *test_trans*, *estim_trans*, and *perf_trans*. The four stages shown in the figure are described below:

1. *Data selection* forms the heart of any optimisation algorithm. Its purpose is to select a transformation, and the portion of the design to which it should be applied. This selection is made in a variety of ways according the particular optimisation algorithm in use. For example, the *simulated annealing* algorithm chooses a random transformation and target area, whereas the *tailored heuristic* applies transformations in a fixed order to every part of the design. The selections made are entered into a transformation data structure which acts as the primary conduit for passing data between the various optimisation procedures.

2. *Testing* - the testing stage examines the transformation data structure to determine the validity of the transformation requested. This ensures that it will not alter the behaviour of the design, and considers such factors as instruction dependency and mutual

exclusivity. Extra information is also added to the transformation data structure for use in the following stages.

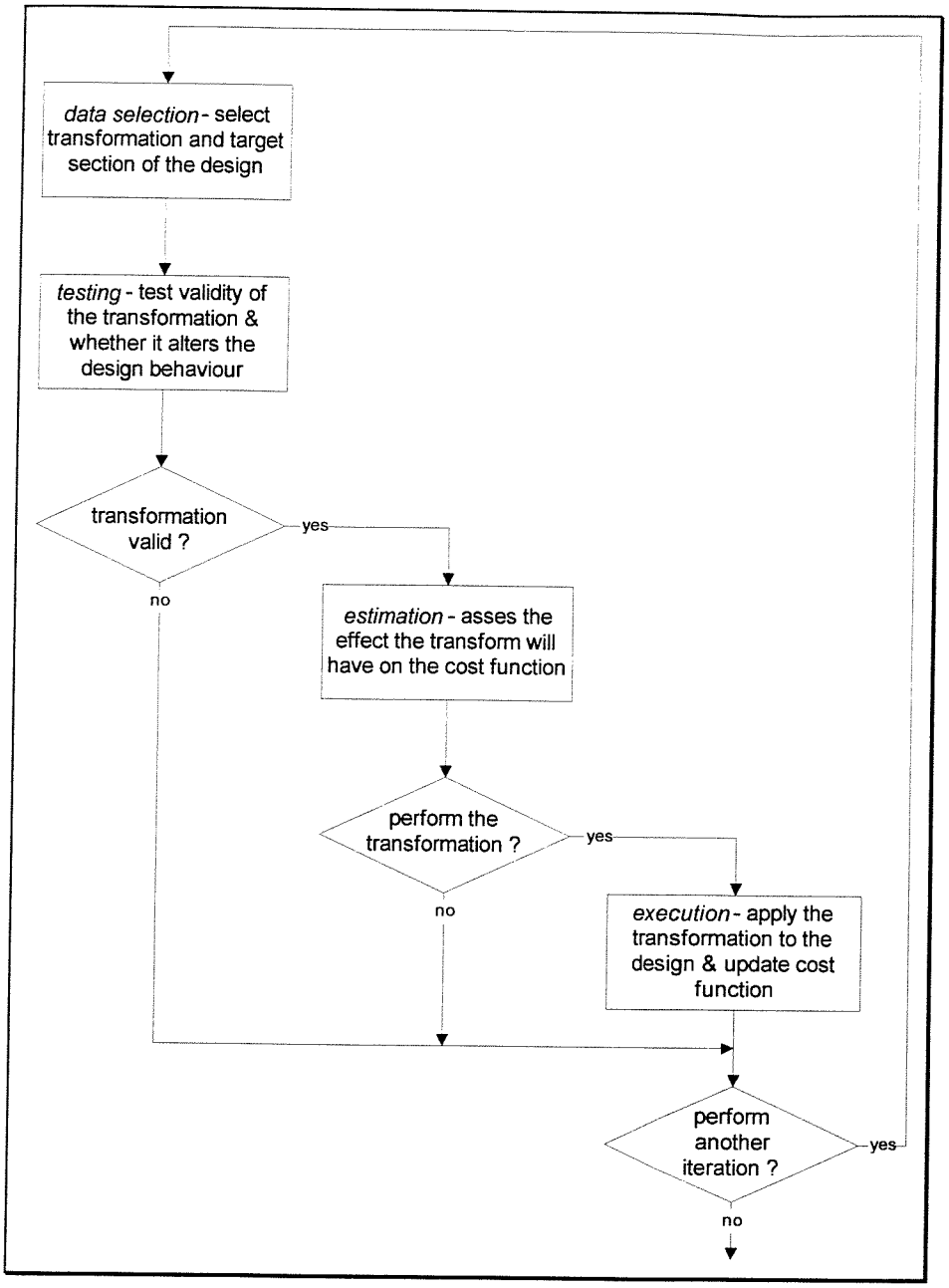


Figure 4.8 The steps to applying transformations in the optimisation loop

3. *Estimation* - assuming the testing stage is successful, the next step is the estimation phase. This plays a vital part in the optimisation process by evaluating the effect of the transformation on the user objectives, without permanently altering the design. In general, this simulates all the changes made to the design by the transformation and calculates the effect of these on the current design performance figure. For example, when estimating the effect on the total area of sharing two functional units, the routine

subtracts the size of both units, adds the size of the combined unit, and also adds in the size of the necessary input multiplexors. The estimation routine returns the calculated figures in the transform data structure, which are then used by the optimisation algorithm to determine whether or not to make the actual changes to the design, ie. perform the transformation.

4. *Execution* - the final step is to actually alter the design data structures to reflection the changes made by the transformation. This then re-evaluates the state of the design, reflecting the effects of the transformation in the cost function (see section 4.5.1).

The following sections describe the main transformations currently implemented, and the tests performed before they may be applied. These fall into two categories: *scheduling transforms*, which apply mainly to the control graph; and *allocation and binding transforms*, responsible for altering the structure of the data path.

4.4.1 Scheduling Transformations

The scheduling transformations perform control graph optimisation by altering the assignment of instructions to control states. There are four basic state merging transformations: *sequential merge*, *parallel merge*, *merge fork and successor* and *group on register*; together with two state splitting transformations to undo the merging: *ungroup on group* and *ungroup to time*; and a final *clock set/multicycling* transformation.

The basic merge transformation is the *sequential merge*. This combines two control nodes in a sequential section (ie. a section composed solely of general nodes) to form a single state, implementing multiple instructions. Figure 4.9 shows the control and data paths before and after an example sequential merge, performed on control states S_1 and S_3. The dependency between these instructions means that the combined state contains a single instruction group with *i3* dependent on *i1*. In order for *i3* to receive the correct input value for variable *a*, the two operations must be chained by bypassing the input register to the adder, which then connects directly to the multiplier output. Note that the register *a* must still be updated for later use, however if after the merge it no longer has any output nets, it will be neglected in the total area calculations and be removed when optimisation is complete. The tests performed prior to execution must check that: a) the instructions being

merged do not share any hardware, or are mutually exclusive; b) there are no dependencies between the top state instructions (*i1* in the example) and those in the intervening nodes leading to the bottom state; and c) the chaining of operations does not result in the state execution time violating any user-specified clock period (the execution for the resulting state is the sum of the multiply and add propagation delays, as illustrated earlier in Figure 4.6).

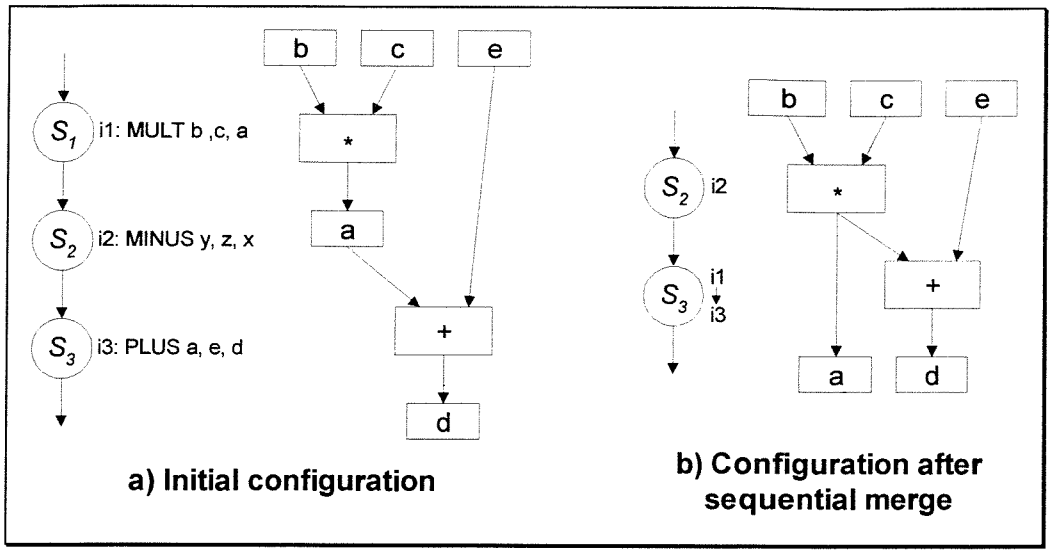


Figure 4.9 Example sequential merge transformation

The *parallel merge* transformation also combines several control states into one, however in this case, all the nodes must be successors of a fork node with the same activation conditions, ie. concurrently executed states. Figure 4.10b shows the result of performing a parallel merge on successors to state S_1 with no condition on their input arcs. The resulting state (S_2) contains the two merged instructions in different instruction groups, which are independent, thus executing concurrently. Note that the removal of the concurrent branches has demoted S_6 from a *collect* to a simple *dot* node. The only test necessary is to ensure that the merged nodes all have a single input arc, with identical activating conditions (ie. they are parallel). Since the states are, by definition, activated concurrently, there is no need to test for hardware sharing or dependencies in the data path.

The next transformation, *merge fork and successor nodes*, combines elements of the previous two. It takes as parameters a fork node and one of its immediate successors, and attempts to move all of the successor instructions into the fork, chaining operators and bypassing registers as required. Figure 4.10c and Figure 4.10d show two examples, the first merging states S_2 and S_4 from Figure 4.10b, and the second states S_2 and S_5 from Figure

4.10c. In both cases, the arcs are translated into a conditional instruction execution within the merged state, indicated as a condition signal in braces following the instruction number, eg. in Figure 4.10d, *i4* is controlled by S_2 and *s42*. Furthermore, the output arcs of the successor must be combined with the input arc condition, and added to the fork node. This may result in multiple arcs between two identical nodes which should be amalgamated into one, such as those created when merging states S_5 and S_2 in Figure 4.10c. In this case, the combining of the two inverse conditions (*s42* and */s42*) produces the single unconditional output arc of Figure 4.10d. Finally, the dot node, S_6, may be completely removed as it executes no instructions and is simply responsible for re-joining the two conditional branches. The tests performed prior to a fork/successor merge must ensure that there is no hardware sharing between instructions in either node, and check that any specified clock period is not exceeded due to operator chaining.

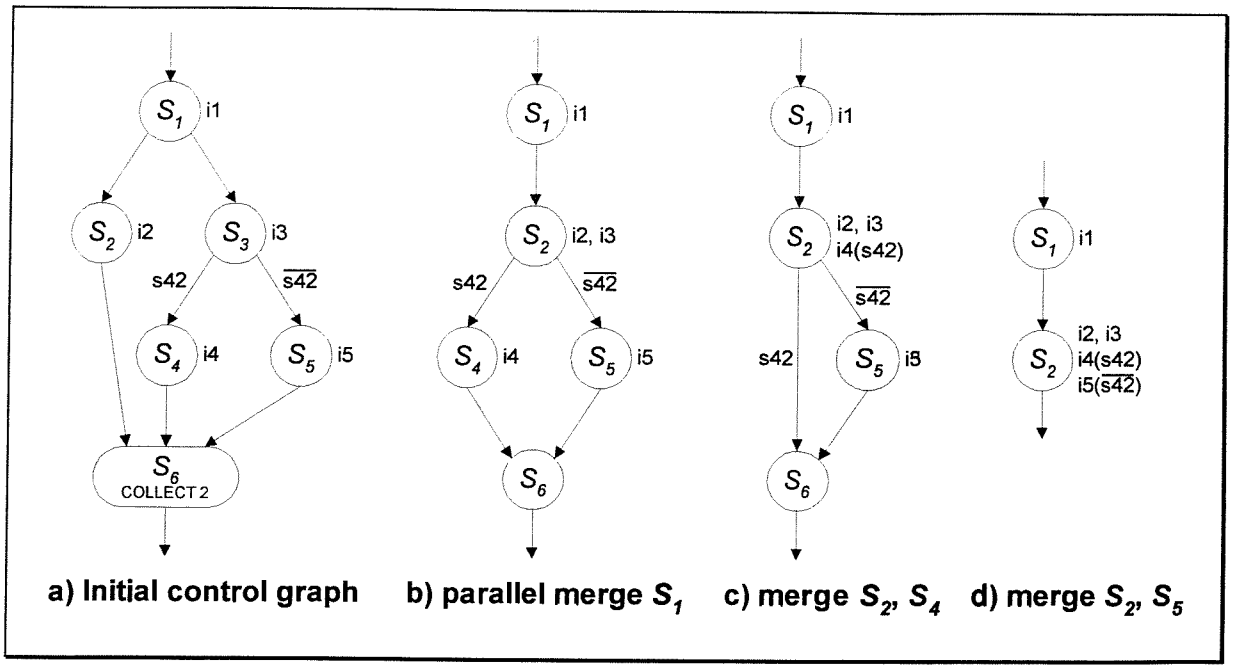


Figure 4.10 Example parallel and fork/successor merge transformations

The final merge transformation, *group instructions on register*, is geared towards removing registers in the data path with one input and one output arc; a common occurrence due to compiler-generated temporary variables (eg. registers *1*,*2* and *3* in Figure 4.5). It takes as parameter a data path register which must have a single input and output net, that is, a register implementing a variable accessed by one reading and one writing instruction. An attempt is then made to merge the group containing the writing instructions into the reading control state. If successful, the intermediate register may be bypassed, effectively removing it from the circuit. For example referring back to Figure 4.9, grouping on register *a* will

merge writing instruction *i1* with reading instruction *i3*, resulting in the removal of the register and the same optimised configuration as illustrated. The tests performed ensure that: a) the register is indeed accessed by only one writing and one reading instruction; b) the writing group may legally be moved to the reading control state (no intermediate dependencies); c) the merged groups do not share any hardware; and d) the clock period is not exceeded.

As discussed earlier, to enable MOODS to climb out of local minima, inverse transformations are implemented to undo the effects of earlier optimisations. Hence the two state-splitting transformations: *ungroup node into groups*, and *ungroup node into time slices*. These two transformations remove instructions from a specified control node, placing them into a number of new nodes inserted into the control graph. Figure 4.11 illustrates a simple example of both operations on a single state containing two groups, of two dependent instructions each. Figure 4.11b shows the *ungroup into groups* transform, which moves a specified instruction group into its own separate control node. In the example, group *g1* is transferred from node S_1 to the new node S_2.

Ungroup into time slices is slightly different in that it takes as parameters a control node and a maximum execution time. The instructions are then separated out into new nodes such that there are no states with an execution time exceeding the period specified; single instructions with a delay greater than this however, will result in a control state exceeding the limit. For example, in Figure 4.11c, ungrouping to a time of 20ns results in the three nodes shown: nodes S_1 and S_2 are both within the 20ns, however since *i3* requires 30ns, S_3 must exceed the delay limit. This problem is tackled by multicycling *i3* over several states, as described in Chapter 5. Two tests must be performed to check whether instructions may be extracted from a node. First, no additional dependencies may be created as a result of the rearrangement of the instruction schedule. For example in Figure 4.11c, if *i3* reads from a variable written to by *i5* then the extraction shown will not be allowed. Secondly, any registers bypassed in the initial grouping of instructions (eg. between *i1* and *i3*) will be re-used when the instructions are separated, they must not therefore be in use during the new control states, which may occur due to register sharing during optimisation.

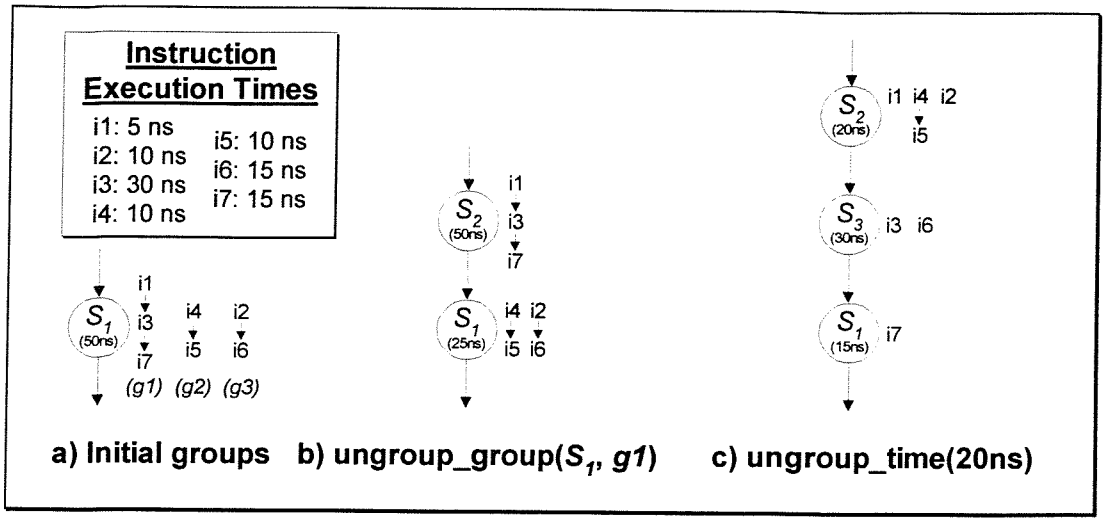


Figure 4.11 Example ungroup transformations

The final scheduling transformation sets the maximum clock period. This is unlike the others in that it is really a global optimisation step, which makes use of the *ungroup to time steps* transformation, forcing all control nodes below a user-specified clock period. Any instructions taking more than the required cycle time (such as *i3* in Figure 4.11c) are *multicycled*, ie. executed over several cycles, through the insertion of extra dummy control nodes. Multicycling is discussed in greater detail in Chapter 5.

4.4.2 Allocation and Binding Transformations

The second group of transformations operates on the data path, performing allocation and binding optimisation, principally concerned with the sharing and unsharing of data path elements. MOODS implements two sharing transformations: *combine functional unit*, and *share registers*; together with their associated inverse unsharing transformations: *uncombine instruction from unit*, *uncombine unit fully*, *unshare variable from register*, and *unshare register fully*.

As its name suggests, the *combine functional unit* transformation is responsible for amalgamating two functional units into one, time-shared between several operations. Control signals derived from the states executing the implemented instructions, are used in conjunction with multiplexors inserted into the data path to select the appropriate inputs, and load the required output registers. Where the two combined units perform different operations, an ALU may be used (module library permitting). Figure 4.12 illustrates the sharing of add and subtract units into a single +/- ALU. Note that although the multiplexors

are shown, they are not explicitly included until the final netlist output stage, instead they are represented as multiple input nets to the central unit, conditional on the appropriate selecting instructions.

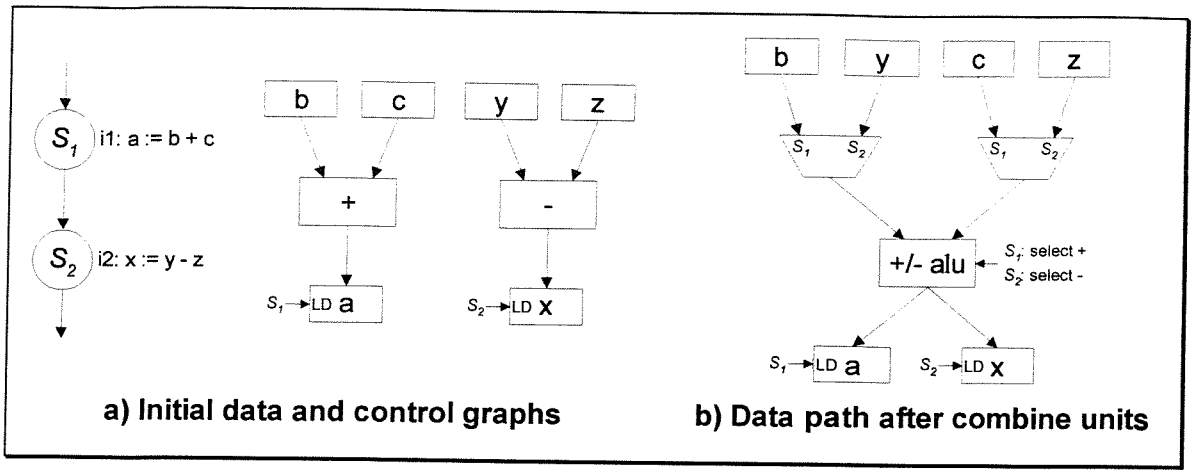


Figure 4.12 Example combine units transformation

The primary mechanism for determining whether units may be combined is to query the module library. This returns a valid module number if an implementation exists capable of performing the set of operations required. Clearly the range of modules available has a significant impact on the sharing possible. In addition to checking for a suitable module, the testing stage must also ensure that the two original units are not concurrently active.

The *share register* transformation is similar in concept to *combine units*, allowing multiple ICODE variables to be stored in a single register. It takes as parameters two data path registers, each of which implements a number of variables. The transformation then attempts to multiplex access in such a way as to realise both sets in a single register. For this to occur, the lifetimes of all the variables under consideration must either be non-overlapping, or must occur only in mutually exclusive conditional branches. This test also takes into account variable persistence around loops and through conditional branches. Note that MOVE instructions, which represent pure data transfers between two variables, may be eliminated by sharing the source and destination registers, thus realising a "self-loading" register and possibly eliminating a control step.

The inverse sharing transformations directly undo their sharing counterparts. *Uncombine instruction from unit* takes a shared functional unit implementing two or more ICODE instructions, and removes one of those instructions into a new unit inserted into the data

path, thereby re-creating the original structure prior to sharing. This makes use of the module library to create a new unit based on the operation being extracted, and not just a copy of the shared unit. The library is also used to select a replacement for the shared unit based on the set of implemented operations minus the removed one. Thus unsharing the MINUS instruction from the ALU of Figure 4.12b, results in exactly the same structure as Figure 4.12a, with a new subtract unit created, and the ALU replaced by an adder.

The equivalent transformation for unsharing registers is *unshare variable from register*, which takes a shared register and removes one of its implemented variables into a new register. Two slightly higher-level transformations, *uncombine unit fully* and *unshare register fully* are also provided. These utilise the former transformations to completely unshare a functional unit or register into separate elements for each constituent operation or variable. Note that no tests are required by any of these since they are all purely structural, having no effect on the behaviour of the circuit.

The *alternative implementation* transformation is the only binding optimisation provided by MOODS. It takes as input a functional data path unit and a reference to an alternative module implementation in the module library, which is used to replace the module currently bound to the specified unit. For example, substituting a carry-lookahead for a ripple-carry adder, will reduce the unit delay, at the cost of area. The alternative module is obtained by querying the module library during the data selection phase of the optimisation algorithm, based on the operations implemented by the target functional unit. This may produce either a range of different implementations, or a single optimised module, depending on the query mechanism employed. From these, a particular implementation must be chosen according to the optimisation algorithm, and the estimated effect on the cost function. For example, the simulated annealing algorithm queries the module library for a list of all possible implementations and from these, chooses a single random module, deciding whether or not to execute the transformation according to the projected change in the cost function. Details of the module library may be found in Chapter 5.

4.5 Optimisation

The flow chart shown earlier in Figure 4.8 depicts the generic form for any iterative improvement-based optimisation algorithm, comprising the four main stages in the selection

and application of transformations. At present, MOODS implements two algorithms: *simulated annealing* [3] and *tailored heuristic* [99] optimisation. Generally, the two main stages in an optimisation algorithm are the initial data selection phase, which entails selecting a local transformation to apply, and the estimation phase which evaluates the global effect of the transformation on the design, and decides whether it should actually be performed. It is the implementation of these two decision-making processes which constitutes the heart of the optimisation algorithms.

4.5.1 The Cost Function

The fundamental metric required by any optimisation algorithm is a measurement of the "goodness" of the design. This is formulated around the calculation of a *cost function* which evaluates a design configuration with respect to the target objectives input by the user, producing a single figure quantifying trade-offs between the various design aspects.

The cost function in MOODS allows the user to specify objectives on area, delay and static power consumption. The ability to specify multiple, possible conflicting, objectives is a unique feature of the MOODS system, allowing the user to explore the complex relationships and trade-offs necessary to synthesise the most suitable implementation. Each objective is specified as a target value and an optimisation priority for a particular design aspect (eg. optimise area, priority 1, target value $10^6 \mu m^2$). The priority determines the order in which targets are optimised, thus priority 1 describes the primary objectives, all of which must be met before any lower priority objectives are considered.

During optimisation, once a transformation has been selected its global effect on the design is evaluated by estimating the change in "energy" of the system. For a single objective, the change in energy is determined thus:

$$\Delta E = \frac{C_{estimate} - C_{previous}}{C_{initial}}$$

where $C_{estimate}$ is the estimated cost after the transformation has been applied (eg. total area or critical path delay), $C_{previous}$ is the current implementation cost before the transformation is applied, and $C_{initial}$ is the cost of the initial implementation, used as a normalising factor to allow the comparison of different design aspects. The average energy change for all primary

objectives forms the main measure by which a transformation is evaluated, thus a negative average energy change implies a general improvement in the design in terms of the target objectives. If all primary objectives are met, ie. all $C_{previous} \leq C_{target}$ and $C_{estimate} \leq C_{target}$, (where C_{target} is the target cost for the objective), then ΔE is calculated from the priority 2 objectives, and so on for the other priorities. Once a value for ΔE is obtained, the optimisation algorithm decides whether to reject the transformation and proceed immediately to the next iteration, or first execute it, and then update $C_{previous}$ for the new implementation.

Over simplistic costing mechanisms, such as considering circuit area purely as the number of functional units, or delay as the number of control nodes in the critical path, lead to an inaccurate representation of the design space, making anything other than general trade-offs difficult [26, 37]. Thus the modelling of circuit characteristics, both at the module and system level, must be accurate enough to ensure that a realistic picture of the state of the design is obtained. In MOODS, low-level characterisation data is fed up from the module library based on the target layout tool. The modelling of this data within the library is of no interest to MOODS itself, however, it must provide the required information accurately enough to ensure the effectiveness of the global models (see below). Methods used may include low-level circuit analysis, such as in Fred [54] and FACE [59], algebraic design space modelling [100, 101], or simple database lookup tables.

MOODS provides global models for each optimisation objective, at present area, delay and static power. The total design area is calculated thus:

$$area = \sum area_{dp} + \sum area_{cg} + \sum area_{i}$$

where $area_{dp}$ and $area_{cg}$ are figures based on the accumulated module areas for all the data and control path units respectively, and $area_i$ encompasses multiplexors and layout wiring costs. At present, both wiring and the cost of the control equations connecting the control and data paths are ignored due to the difficulty of obtaining accurate estimations prior to final logic optimisation and layout. This is currently under investigation through the development of a rough floor-planning tool to be used within the optimisation loop. Power is calculated using an identical method based on module power consumption.

Delay calculation is a somewhat more complex process involving the determination of the maximum required clock cycle and the critical path. Assuming a user-specified clock period has not been set, the clock cycle time is determined by analysing the instructions in each control node. The longest node sets the cycle time, which is multiplied by the critical path length to give a single delay figure. If the user has specified a clock period, this is used in lieu of the longest node time. Node delay is determined from the data path units implementing its constituent instructions as illustrated earlier in Figure 4.6. The calculation is made up from three main components: the instruction input register propagation delay, the functional unit propagation delays, and the output register setup time. Thus, the delay of a node is determined thus:

$$delay_{node} = ipd_{Rmax} + (pdf_{Rmax} \times ic_{OPmax}) + T_{Gmax} + st_{Rop}$$

where ipd_{Rmax} is the inherent propagation delay for the input register R_{max} to the first instruction in the longest group G_{max}; pdf_{Rmax} is the input register *delay factor* and ic_{OPmax} is the total input capacitance driven by R_{max}; st_{Rop} is the setup time for the output register and T_{Gmax} is the total propagation delay for the longest group G_{max}. This propagation delay is calculated by summing the sequential instruction delays within the group where each instruction has a delay:

$$delay_{inst} = ipd_{fu} + pdf_{fu} \times ic_{tot}$$

where ipd_{fu} is the inherent delay of the functional unit implementing the instruction, pdf_{fu} is its propagation delay factor, and ic_{tot} is the total input capacitance driven by the unit. All of the parameters (inherent delays, delay factors, input capacitance and setup times) are available via module library queries.

4.5.2 Simulated Annealing Optimisation

Simulated annealing [102, 103, 104] is a general optimisation technique for minimising a function of many variables, in this case the cost function, where the variables represent the many configurations of the circuit elements (data path nodes, control nodes, etc.). It is derived from the statistical mechanics of annealing in solids where, in order to produce a highly ordered crystalline structure with few defects, a material is annealed: first melted, energising the constituent atoms and rearranging the structure, then slowly cooled, allowing

the system to reach thermodynamic equilibrium, until it freezes in a low-energy state, usually a crystal lattice.

The simulated annealing optimisation algorithm is based on the Metropolis algorithm [105] which fits neatly into the MOODS optimisation loop. The basic method is to select a random transformation (equivalent to the random movement of a particle in the melted solid), and evaluate the resulting change in the cost function, accepting both improving ($\Delta E < 0$) and degrading ($\Delta E > 0$) transformations, where the probability of accepting a degradation is a function of the annealing temperature. The process can be visualised in a *configuration space,* which plots the cost function value for each system configuration. Figure 4.13 shows a simple one-dimensional configuration space with a number of minima. Considering the initial configuration at point *A*, a basic iterative improvement optimisation strategy accepts only downhill (improving) transformations and will generally obtain only a local minimum, point *B*. In order to find the global (deepest) minimum, simulated annealing permits a degree of uphill (degrading) transformations to be applied, allowing the configuration to jump out of local minima into point *C* in the example.

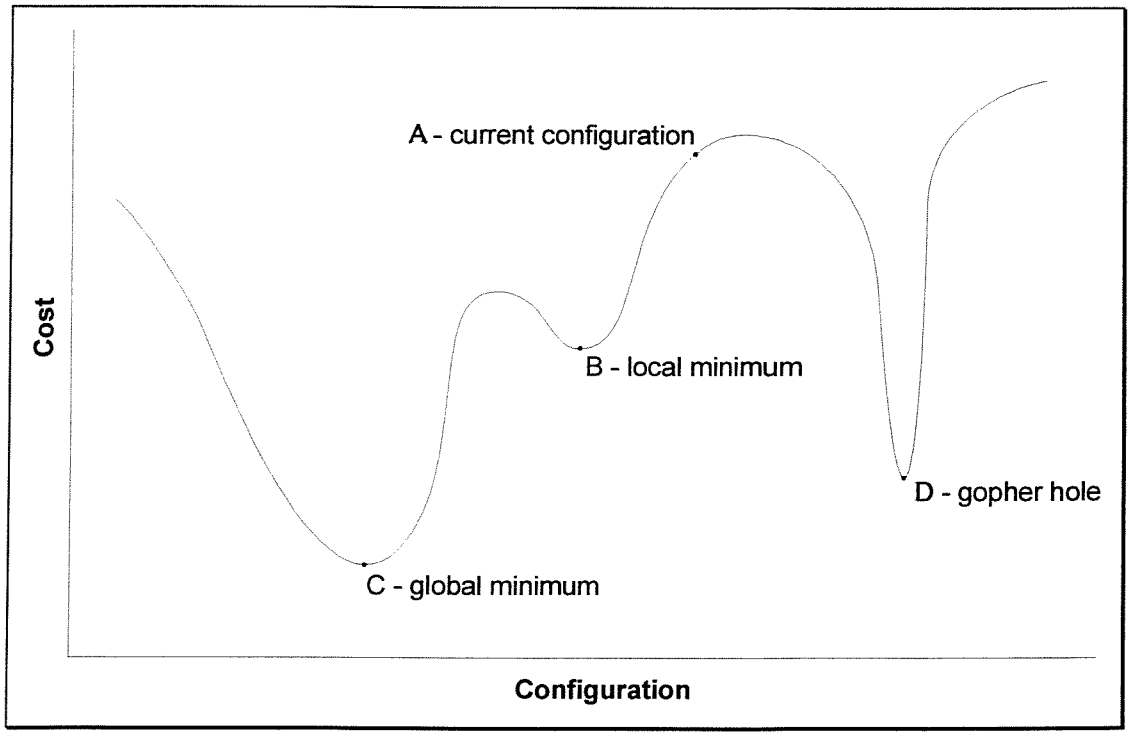


Figure 4.13 A one-dimensional configuration space

Figure 4.14 outlines the algorithm as implemented by MOODS. The user specifies an annealing schedule in terms of the *start* (high) and *end* (low) temperatures, the *number of*

temperature steps and the *number of iterations* to perform per step. These figures form the optimisation iterations during which transformations are selected at random and estimated. The acceptance decision is made thus: accept unconditionally for an improvement, otherwise for a degradation, generate a uniform random number between 0 and 1 and accept if this figure is less than the probability threshold value $e^{-\Delta E/T}$. This ensures that as temperature decreases, the size of the accepted degradations will also decrease. Thus at first, the design will jump easily between the various minima, however, as the temperature drops these becomes more and more difficult to exit and, as it "freezes", the most difficult minimum to exit, the global minimum, should be obtained.

```
for (temp = start; temp >= end; temp *= step) {
  for (i = 0; i < no_its; i++) {
    t = select_transformation();
    if (test_transformation(t)) {
      delta_E = estimate_cost_function(t);
      if (delta_E<0 || rand()<exp(-delta_E/temp))
        execute_transformation(t);
    }
  }
}
```

Figure 4.14 The simulated annealing algorithm

The main advantages of the simulated annealing approach are its abstractness and its ability to find a global minimum. No knowledge of the trade-off mechanisms involved in optimisation is required, instead the process relies entirely on the cost function and transform estimators to encapsulate the design space. This allows complex trade-offs to be made between many conflicting objectives, and permits the inclusion of further objectives (maybe dynamic power consumption or testability) by the addition of a costing mechanism to the cost function, together with the appropriate modelling routines. However, there are a number of disadvantages and pre-conditions which make simulated annealing awkward to use in practice:

- The performance is highly dependent on the landscape of the configuration space: a fairly regular landscape with gradually flowing hills and valleys leading to a deep global minimum is an ideal application for simulated annealing. However a more severe and irregular landscape, populated with deep gopher holes that are difficult to exit (such as point *D* in Figure 4.13), is liable to result in only a local minimum being found.

- Performance is also dependent on the range of transformations provided and the scope of random selections made. These must ensure that the configuration space can be easily traversed, readily jumping into, and out of minima.

- Control of the annealing schedule requires the user to specify several abstract parameters. Not only are these unrelated to any physical measure, they are also difficult to predict in advance, requiring considerable experience to obtain the best solutions.

- Finally, compared to heuristic methods, the simulated annealing approach is slow, especially for larger designs. Its reliance on the generation of random transformations means that many iterations are required for a system to reach equilibrium.

4.5.3 Tailored Heuristic Optimisation

MOODS addresses some of the drawbacks of the simulated annealing approach via the tailored heuristic optimisation algorithm, which is both faster and more user-friendly. The same set of transformations is used, but instead of random selection, they are applied according to a pre-defined regime, guided by an analysis of the design. The algorithm only performs area and delay optimisation, encapsulating knowledge of suitable trade-offs, obtained through an analysis of the effectiveness of each transformation on a range of designs [99].

Two basic routines are employed to optimise delay and area:

1. *compact_CP* performs control graph compaction, utilising the control graph merging transformations to reduce the length of the control path, thus reducing delay and to a lesser extent area.

2. *compact_DP* processes the data path using the data path sharing transformations to optimise area.

Both routines rely on a number of pre-calculated metrics to target the most effective sections of the design for optimisation, akin to the freedom and force measures used in MAHA [39] and HAL [33]:

- The *critical path* is identified to determine the control nodes which most affect circuit delay. These are targeted during delay optimisation, leaving non-critical nodes as fodder for area optimisation.

- A *shareability factor* is calculated for each data path functional unit, quantifying the area saved when maximally sharing it with all other compatible units, ie. all function combinations for which a suitable module exists. The higher the shareability factor, the more potential there is for reducing the data path area through sharing the unit. Obviously, a negative value identifies units unsuitable for sharing.

- Each control node has an associated *share factor*, which is determined from the percentage of the instructions executed by the node possessing a positive shareability factor. Since merging of control nodes reduces the potential for sharing of the activated data path units (due to the extra dependencies resulting from operator chaining), the lower the share factor, the less effect there will be on area minimisation due to node compaction.

- The equivalent of the share factor for the data path is the *critical path factor* which quantifies how close a functional unit is to the critical path. This is defined as the percentage of instructions implemented by the unit that are executed by critical path control nodes. The lower this value, the less effect sharing of the unit has on the potential for delay minimisation, again due to the interdependence between sharing and operator chaining.

The two optimisation routines are designed to optimise a particular main objective (area or delay), while minimising the effect on the secondary objective, therefore compaction of the critical path (*compact_CP*) applies the sequential, fork/successor, and parallel merge transformations to all critical path nodes where, for each transformation, only those nodes with a share factor below a specified threshold value are merged. This process is repeated several times and the share factor threshold increased on each iteration, starting from 0%. Optimisation completes when either the threshold reaches 100%, or the target delay criterion is met. Assuming the latter case, the design should have been optimised in such a way as to allow the maximum possible data path minimisation.

The data path compaction routine (*compact_DP*) is similar to its control path counterpart. Here, the routine attempts to maximally share each functional unit, however only those units with a critical path factor less than a defined threshold are considered. This iterates with the threshold increasing steadily from 0% until it either reaches 100% or the area target is met. This should result in an area optimised design with maximum potential for control graph compaction.

These routines form the major portion of three individual optimisation algorithms performing delay optimisation with a secondary area objective, area optimisation with a secondary delay objective, and combined area and delay optimisation where both have equal priority. Flow charts illustrating the basic operation of each algorithm are shown in Figure 4.15. Taking delay first: the critical path is optimised via *compact_CP* to achieve the delay target; *compact_DP* is then called to minimise the data path within the scheduling constraints already set; and a final stage performs register optimisation and alternate module selection. Register optimisation makes use of *the group instructions on register* transformation to minimise single input and output registers, and *share registers* to combine the rest, while alternate module selection chooses the most suitable module implementation (according to the cost function) for each functional unit.

Area optimisation is almost identical to the above, except for the interchanging of the two compacting routines. Finally, combined area and delay optimisation is achieved by interleaving control and data path compaction within the share factor and critical path factor iterative loops, followed by register optimisation and alternate module selection.

Note that throughout, only downhill (improving) transformations are ever applied, thus there is always the danger of only achieving a local minimum. The controlled ordering and targeting of the transformations goes some way to tackling this problem, however, as with all heuristics, there is no guarantee that the result will be a global optimum for every (or indeed any) design. Results suggest that many systems with a high degree of sequential processing produce results comparable to simulated annealing, with the added benefit of faster execution. The algorithm also provides the user with a good idea of what constitutes a realisable target and may be used as a pre- or post-processing step in conjunction with simulated annealing to further optimise a design. In addition, the application of transformations in a fixed order enables the user to predict, in advance, the scheduling of

instructions. This may be useful when attempting to control the detailed timing of I/O operations and is used successfully in the FPGA design project described in Chapter 7.

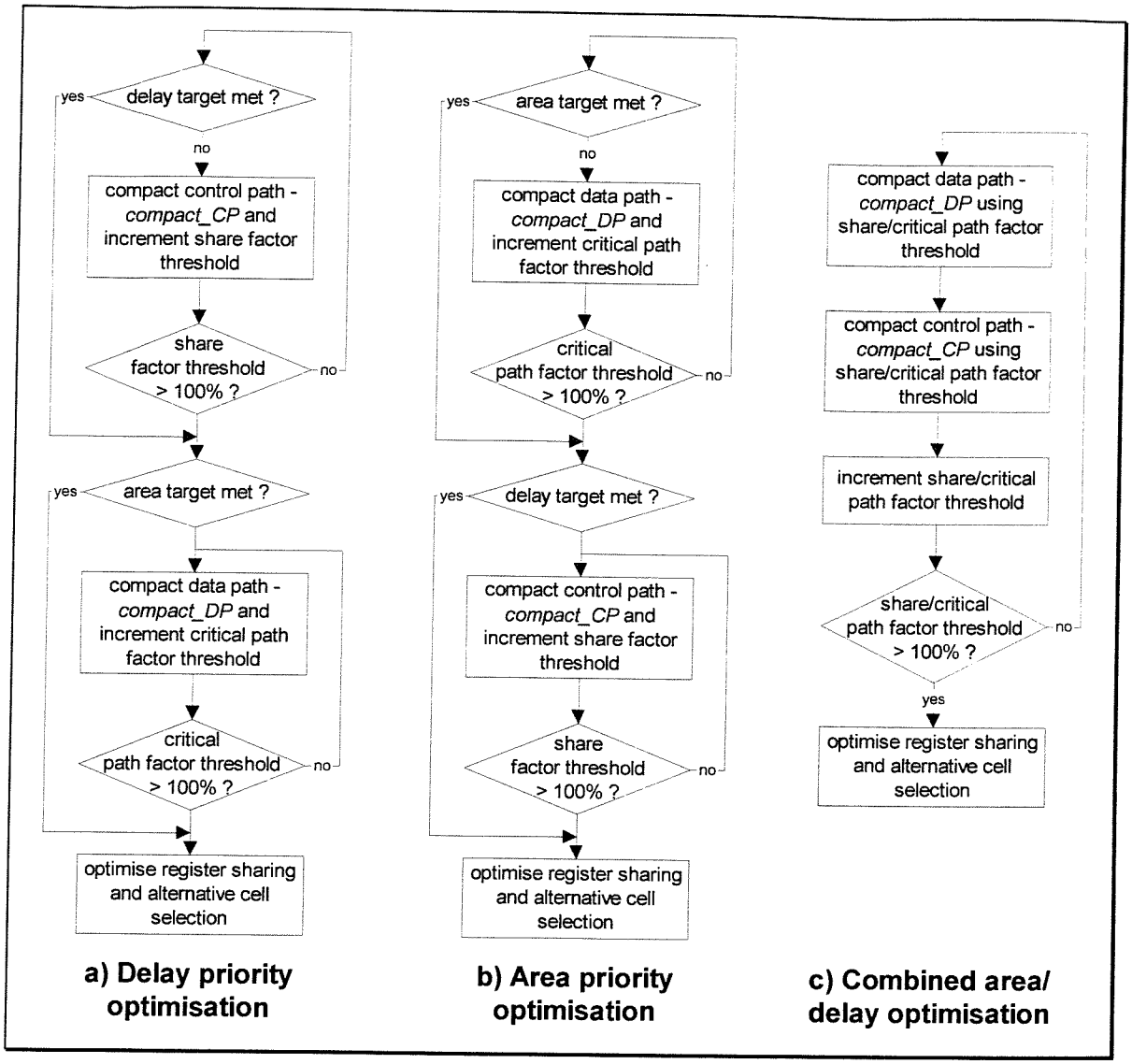


Figure 4.15 Tailored heuristic algorithm flow charts

There are two main disadvantages of the heuristic approach: firstly, its inability to apply degrading transformations can lead to sub-optimal implementations especially where complex interactions occur, such as in the expanded module examples in Chapter 5; secondly, the algorithm only performs area/delay optimisation. The abstractness of simulated annealing allows it to perform multi-dimensional trade-offs between many conflicting objectives without having to understand the complex interactions involved. This is not so for a heuristic method, which must fully encapsulate the interactions between all aspects of the design space in order to perform the most appropriate transformations.

4.6 Summary

The MOODS system allows a designer to synthesise a behavioural description using a variety of tools and optimisation methods. It develops a multitude of structural implementations from a single input description allowing the user to explore trade-offs between several, possibly conflicting, objectives in order to develop the most suitable implementation for particular circumstances.

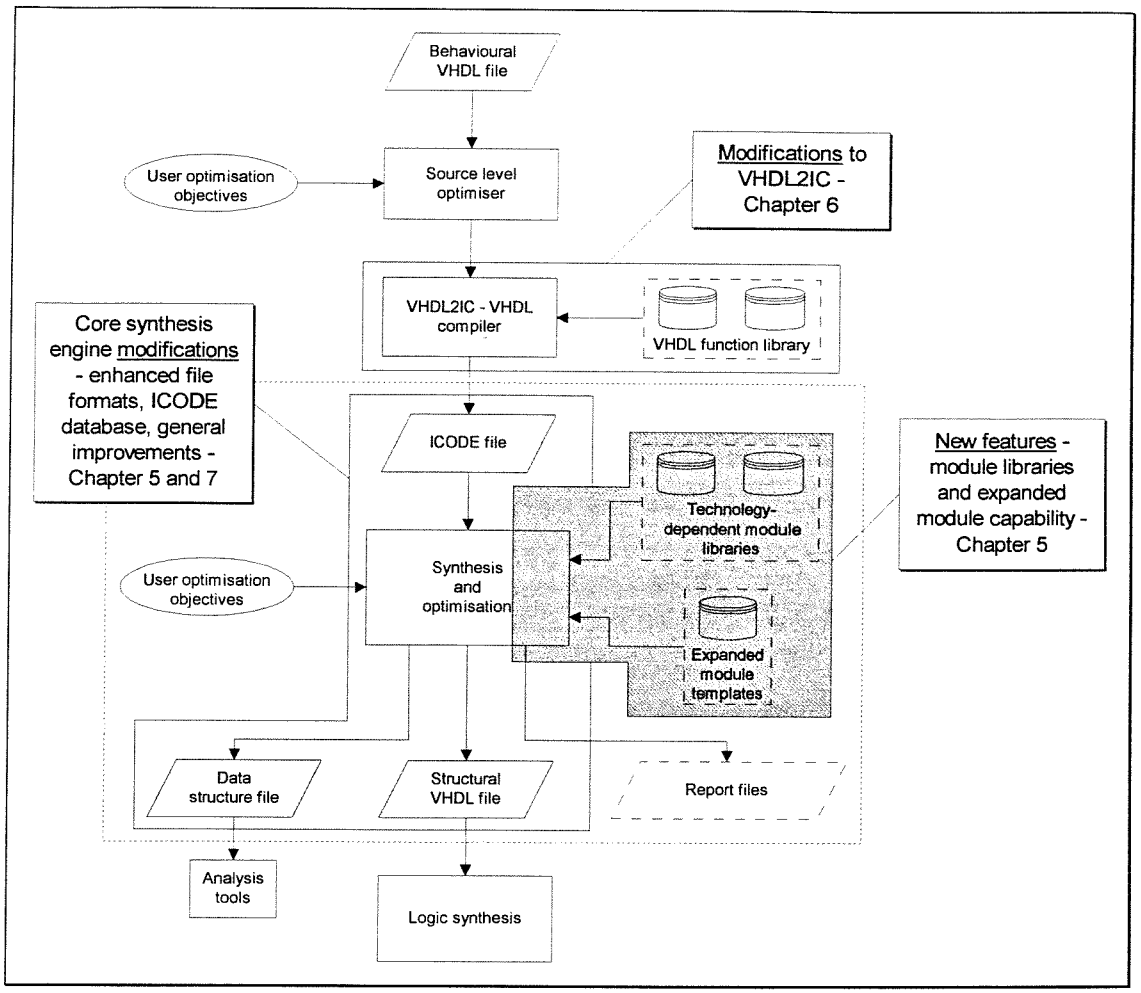


Figure 4.16 Modifications and additions to MOODS

As a research tool, MOODS is continually improved and updated, the source code optimiser being a prime example. The remainder of this thesis describes a number of enhancements, both to the core MOODS synthesis package, and the VHDL compiler. These are identified in Figure 4.16, which reproduces the earlier system block diagram (Figure 4.1), distinguishing between modifications to the original system, and newly added features. In addition, Chapter 7 details the development of a spectrum analyser demonstrator, which represents one of the first examples of physical hardware produced by

the system. This design process provides valuable insights into the real world use of behavioural synthesis, and prompts many further developments.

Chapter 5

Hierarchical Module Expansion

Early synthesis systems considered each data path unit to be implemented by a single combinational module, executed over exactly one clock cycle. Later developments utilised more sophisticated characterisation models allowing operations to be either *chained* together, cascading combinational modules in a single control state, or conversely, *multicycling* long operations over a number of cycles. In both cases, the modules are treated by synthesis as "black boxes" accompanied by area and delay parameters. Any consideration of their internal composition is delegated to the module libraries (see Chapter 3).

MOODS handles modules in a similar manner to other comparable systems, regarding each as a combinational block with delay, area and power parameters. These are modelled using equations developed from characterisation data based on the target low-level layout system. As described in Chapter 4, control graph optimisation results in the chaining of operations due to control state merging. In addition, MOODS implements a limited form of multicycling by means of a user-specified clock period. This allows combinational modules to execute over several control states but suffers from a number of restrictions, described below.

This chapter details the development of *expanded modules*: sequential modules implemented as a mixed structural/behavioural description of sub-modules, which are dynamically expanded within the internal design representation at any point during the synthesis process. They have evolved from the need to resolve the main drawbacks of the original multicycling implementation, the primary concerns being the lack of support for true sequential modules (ie. modules with internal registers and control, as opposed to purely combinational with a delay greater than a single cycle), and the inability of the optimisation transforms to effectively exploit the extra control states created. The

development of a general mechanism for performing *hierarchical module expansion* also facilitates the implementation of a number of additional features. In particular the ability to pre-define local timing relationships within an expanded module, enables the use of so called *macro ports* to easily implement complex interfacing protocols.

The remainder of this chapter is divided into four sections. First, section 5.1 describes the original multicycling implementation and its main weaknesses influencing the development of expanded modules. Section 5.2 describes the underlying principles of expanded modules and *hierarchical module expansion*, considering its operation within the MOODS system, and its effects on the optimisation process. Section 5.3 provides more information on the implementation and integration expansion within the MOODS infrastructure, while section 5.4 details the results of an extensive analysis of its operation and effectiveness on a number of benchmark designs, highlighting the trade-offs and problems encountered. Finally, section 5.5 summaries the chapter and mentions further improvements to the system, some of which are already under development.

5.1 Multicycling

As mentioned above, the original MOODS system possesses a basic multicycling capability based around the specification of a user-supplied clock period. MOODS incorporates in its cost function, a cycle-time objective, which if present, is enforced throughout the optimisation process. Any transformation violating this period will fail its testing stage, ie. control node merging where the chaining of dependent operations results in a total state time greater than the required cycle time. The problem of single instructions with too large a delay is dealt with by extending the instruction into a sufficient number of extra control states inserted into the control graph. These each contain a dummy instruction which refers to the original, but possesses a negative instruction number, tagging it as multicycled. Instruction dependencies and result-loading control signals are all adjusted according to the new schedule. Figure 5.1 illustrates the main use for multicycling: reducing the effect slow modules have on the cycle time, thus reducing the amount of excess delay within the control graph. The figure shows the execution of two instructions with considerably different execution delays. In the non-multicycled example, Figure 5.1a, the control state delay is determined by the longest instruction, resulting in

75ns of excess (unused) time in state 2. After multicycling to a clock period of 75ns, this excess is eliminated, thus reducing the total delay from 300ns to 225ns.

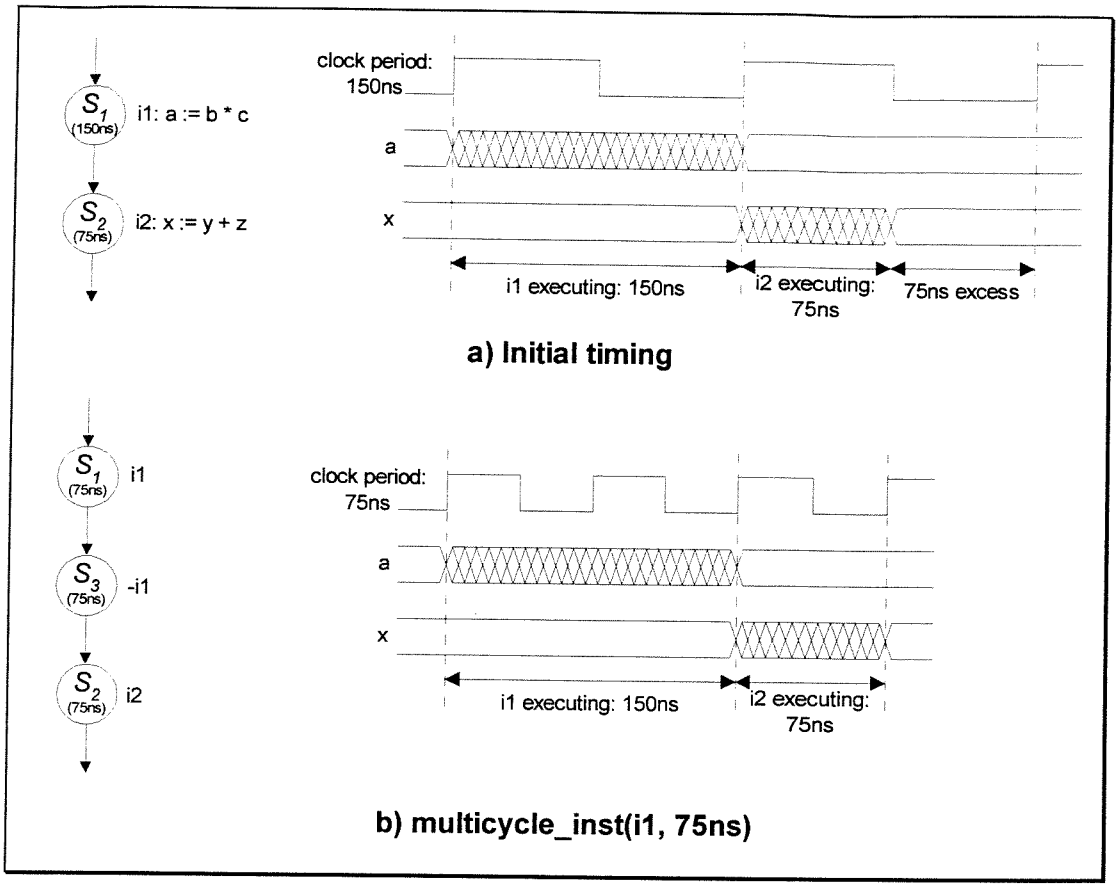


Figure 5.1 Example of instruction multicycling

There are two main drawbacks to this approach:

1. The multicycling process applies only to the control graph. The data path node implementing the instruction must still be a combinational module with a single propagation delay figure, a restriction which misses out on one of the main opportunities afforded by multicycling: utilising sequential (clocked) module implementations to provide greater scope for data path optimisation. This extra dimension can have a significant impact on the range of module implementations available, witness the scope of the topologies described by Theeuwen et. al. [55] (see section 3.1).

2. MOODS has been designed from the outset to consider instructions as executing in a single control state. All testing, estimation, transformation and ancillary routines assume this configuration, thus whenever a transformation test/estimate is performed, any multicycled instructions involved are first un-multicycled, merging all dummy

states back into the original, then re-multicycled once the operation is complete. The re-multicycling step creates a virgin set of control states, each containing a single dummy instruction, thus any previous scheduling optimisations within the original states will be undone. This is illustrated in Figure 5.2 which shows a delay optimised control graph section before, during, and after clock setting. In Figure 5.2a, all instructions occur within a single state with instruction *i1* determining the clock period of 150ns. Setting the cycle time to 50ns forces the dependent instructions *i2* and *i3* into separate states (Figure 5.2b), and requires *i1* to be multicycled by inserting two additional states into the graph (Figure 5.2c). The optimum schedule may then be achieved by merging states S_2 and S_4 producing a final graph of three states, which would, however, immediately be undone when the next transformation is tested.

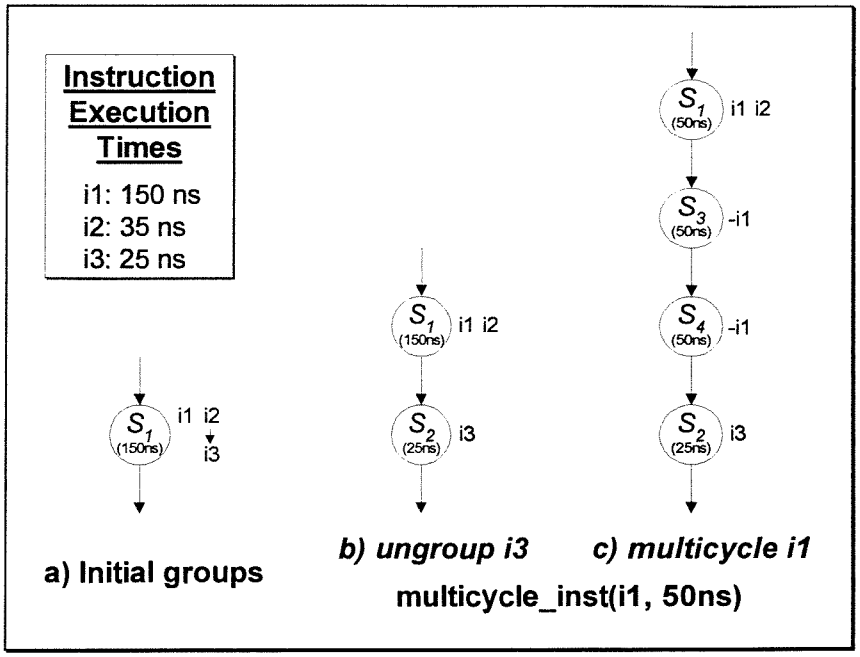


Figure 5.2 Example of inefficient state usage during multicycled optimisation

The net effect may not be too significant on designs with a small number of long instructions, and may well be compensated for by the improvement in clock utilisation, however, more complex situations can result in a substantial waste of control resources, especially where several concurrent instructions are multicycled, forcing them to be effectively serialised.

Neither of these problems is intrinsic to behavioural synthesis but they both require a fundamental change in the design of the system, in case (1), to include the concept of modules with a sub-clock, and in case (2), to consider instructions as taking more than one

control state. The expanded modules described below, work within the confines of the existing system to provide an alternative route for implementing clocked sequential modules.

5.2 Expanded Modules

5.2.1 Sequential Module Implementations

Module expansion exploits the inherent hierarchy of arithmetic and logical operators used in systems such as Fred [54] and BADGE [60, 61] (see section 3.1). In these systems, hierarchical module descriptions provide performance models, and guide the generation of module structure independently from the synthesis process. Module expansion, on the other hand, is based on the concept of utilising this hierarchy in the body of the synthesis loop by expanding the sub-structure of a module into its constituent sub-components within the top-level control and data paths. The process basically involves replacing a given data path unit, and its activating control states, by a sub-control and data path which describes the desired module implementation, effectively flattening the module hierarchy. Figure 5.3 shows a simple example of a 32-bit combinational adder expanded using a serialised implementation, based around the accumulation of an 8-bit partial sum over four cycles.

There are three main factors which influence the effect a module expansion such as this has on a synthesised design:

1. The ability to use sequential implementations for a complex operation facilitates an additional level of trade-off between area and delay. This is achieved, however, without any alterations to the MOODS low-level module model since the smaller constituent sub-modules are all combinational. The hierarchical nature of the description means that additional trade-offs may be made by further expanding sub-modules. For example, the 8-bit adder in Figure 5.3 might be further expanded into a serial implementation based around a 2-bit adder, thus increasing the total addition time from 4 to 16 cycles. Clearly, the depth of expansion must be balanced against both the total delay, and the cost of the extra circuitry (multiplexors and intermediate registers) required.

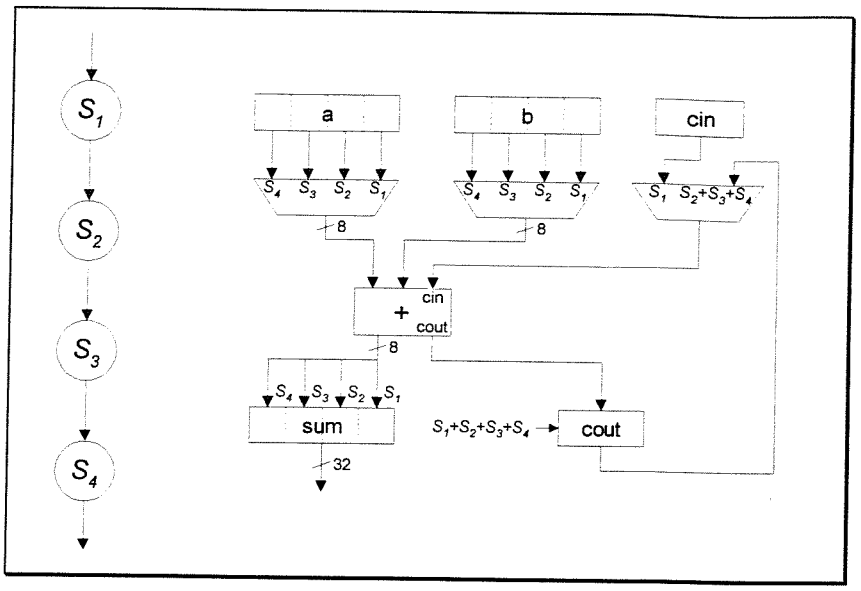


Figure 5.3 Expanded 32-bit serial adder

2. The original single control state is replaced by the controller sub-graph for the expanded module, effecting a finer level of control state granularity. Furthermore, the expansion process generally results in the use of simpler data path units with a shorter delay, consequently the expanded control states will also be shorter than the original. This can have a dramatic effect on the overall speed of the design: by targeting the slowest data path unit (which dictates the maximum control state delay, and hence the minimum allowed clock period) for expansion, both the clock period, and the amount of control state excess time can be decreased. This is similar to the effect of multicycling illustrated in Figure 5.1, except that there is somewhat less control over the exact delay.

3. Because the expanded module sub-graphs are merged seamlessly into the top-level design structure, and are treated identically to all other data and control path elements, further optimisation of the entire design may result in additional improvements through the sharing of sub-modules with other similar data path units. Thus, the area cost of a complex operator may in some cases, be reduced simply to the overhead incurred by the extra control states, interconnect (multiplexors) and intermediate registers from the expanded module. This is used to great effect in the FFT demonstrator described in Chapter 7.

A slightly different approach to the expansion of the 32-bit adder of Figure 5.3 is illustrated in Figure 5.4. Here, the serialised structure is unrolled forming a simple, non-optimal implementation, identical to the initial configuration obtained from a behavioural

description of the sequential adder (see Figure 5.11 for an example behavioural description of a split adder).

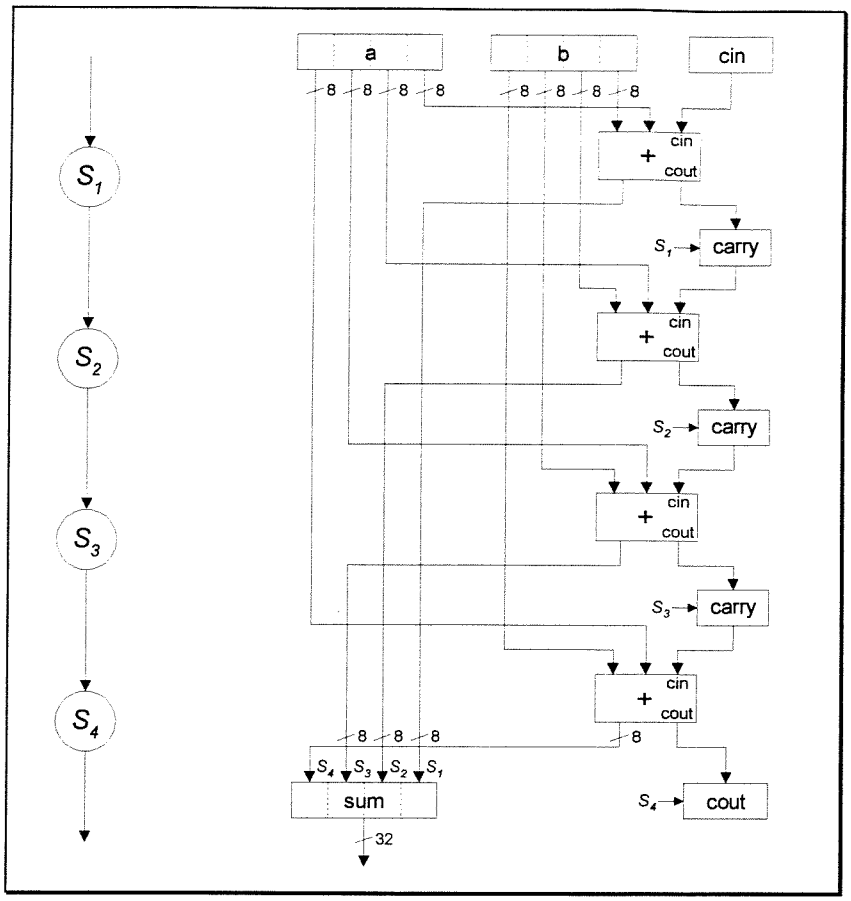


Figure 5.4 Expanded 32-bit split adder

At first sight this approach would appear to be somewhat more inefficient than before, requiring four separate 8-bit adders and carry registers. It is not difficult to see however, that the simple sharing of all the adders, and all the intermediate carry registers results in a data path structure identical to Figure 5.3. This puts the onus on MOODS to draw the maximum benefit from the new configuration through further optimisation, which, while not necessarily the quickest and most computationally efficient method, possesses a number of advantages over the earlier model:

- MOODS will optimise the expanded module in a manner most suited to the target cost objectives. If this requires maximal unit sharing, thus obtaining the original sequential configuration, then so be it. Alternatively, it may be more appropriate to merge the four control states into two, and similarly share the adders, effectively accumulating a 16-bit partial sum over two cycles, as shown in Figure 5.5. It can therefore be seen that *the most important consideration when designing an expanded module is not its structural*

efficiency, but the range of alternative configurations obtainable through further optimisation.

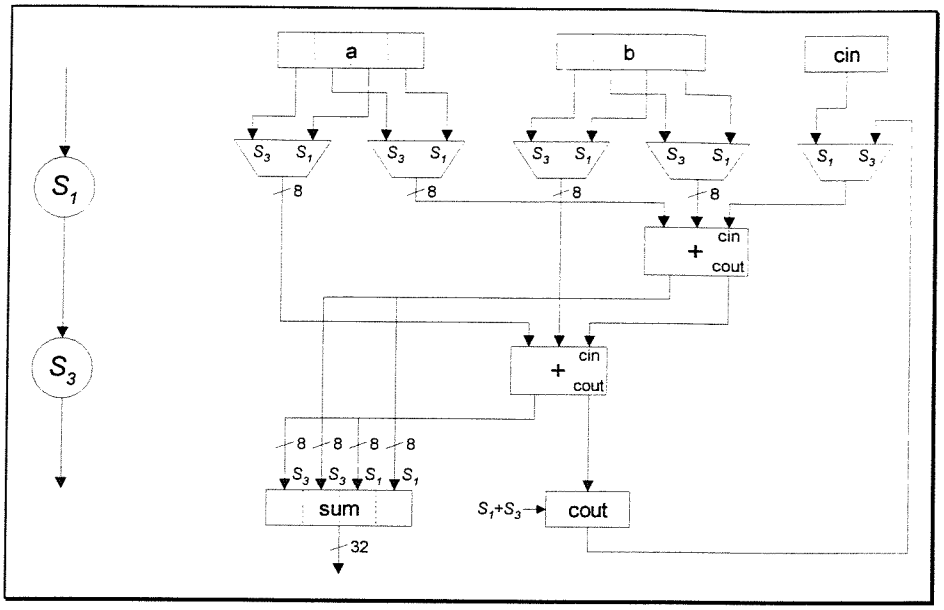


Figure 5.5 Example optimisation of 32-bit split adder

- Feedback loops and unit sharing within the data path of Figure 5.3 make it much more difficult to optimise the sub-modules in the context of the whole design, since it would first be necessary to unshare the adders and registers. This is particularly significant when using the heuristic optimisation algorithm which does not allow these design degradations to be performed.

- Creation of the expanded module template is simply a matter of writing an appropriate behavioural description. Examples of these may be found in section 5.3.2.

On the negative side, the immediate effect of module expansion on the overall delay and area will almost always be an increase, thus the expansion process must be viewed as a transformation which increases the potential for optimisation at a later date. Integration of module expansion within an automatic optimisation algorithm is therefore extremely difficult, as it is not possible to evaluate its global effect until synthesis is complete. This problem is complicated by other considerations such as the depth of hierarchy to expand, which units to target, and at what point in the synthesis process should the expansion occur. These problems are investigated in the later results section (section 5.4).

5.2.2 Further Uses of Expanded Modules

As well as the basic sequential module implementations described above, module expansion may be used in a number of other ways to provide features of considerable import to the user.

- *Pipelining* - there are two forms of pipelining applicable to behavioural synthesis identified in the literature [33]. *Functional pipelining* refers to the process of transforming the top-level design into pipelined stages, the execution of which may be overlapped. SEHWA [65], one of the first systems to tackle pipeline synthesis, partitions an acyclic data flow graph into stages separated by stage latches. This increases the rate at which data may be applied to the top-level design, and is best suited to highly repetitive systems such as digital filters. *Structural pipelining* refers to the use of pipelined modules whose *throughput* is less than the total module delay. These units require modified scheduling algorithms to overlap pipe stage execution according to the required throughput and are most suited to sequences of independent operations. MOODS contains no special consideration of pipelining, however the use of pipelined expanded modules, in conjunction with the standard optimisation algorithms, produces a similar effect. Figure 5.6 demonstrates a scenario where two multiply operations (Figure 5.6a) are expanded using a two-stage pipelined multiplier (Figure 5.6b). Given a suitable cost function assigning equal priorities to both area and delay, optimisation results in the configuration of Figure 5.6c, which is the equivalent of performing the two operations on a single pipelined data path unit, overlapping their execution by one cycle.

- *Macro operators* - expanded modules are also used to implement functions not available in the standard module libraries. These can range from relatively simple individual units, such as the fixed-point multiplier described in Chapter 7, to complete complex function libraries, maybe to support floating point arithmetic and transcendental functions[1]. Since the expanded module templates are not tied to any specific target library, these macro operators effectively form a high-level function library which seamlessly integrates into the system design flow. More details may be found in Chapter 6.

---

[1] This is the subject of a current PhD [110] which exploits the expanded modules described here.

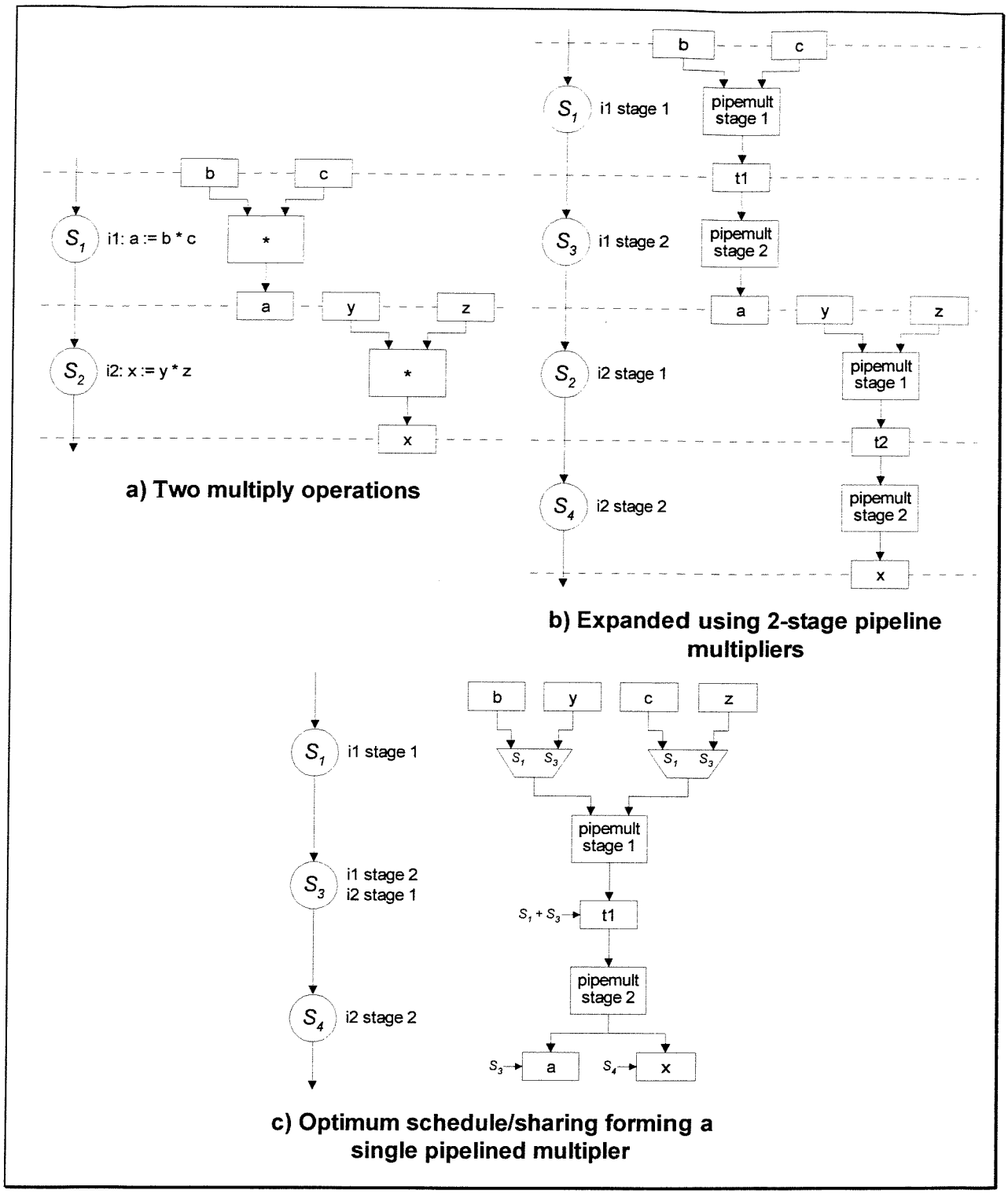


Figure 5.6 Pipelined multiply operations

- *Macro ports* - if module expansion occurs after optimisation, just prior to outputting the design as a structural netlist, the expanded control and data path will be directly implemented in the final configuration. This enables the user to exactly specify the local scheduling of operations within the module on a cycle-by-cycle basis in a similar vein to the behavioural templates [63] discussed in Chapter 3. One of the main

drawbacks of behavioural synthesis is the difficulty in interfacing a synthesised design to the outside world, which generally requires the exact specification of operation timing in I/O protocols (eg. memory access control). By encapsulating the required behaviour in a expanded module template with a fixed, pre-defined operation schedule, complex interfaces can be simplified into a single instruction. Macro ports also allow a designer to use alternative protocols (eg. faster/slower memory I/O), by simply changing the expanded module used. Again, these are discussed in more detail in Chapter 6.

5.2.3 Controlling Module Expansion

The sections above illustrate some of the complex interactions surrounding the use of expanded modules. MOODS is currently configured to investigate these interactions and develop a general strategy to, at least partially, automate the process. Below are described the three main methods with which modules expansion is controlled by the user:

1. The primary mechanism for inserting an expanded module into a design is the *split module* transformation, which replaces a specified data path node by an expanded module implementation. This includes the necessary pre- and post-processing steps to prepare the control and data paths, and ensure the seamless integration of the new expanded sub-structures.

2. *Automatic module expansion* (MOODS command *aob*) expands all data path nodes of a particular functional type wider than or equal to a specified threshold bit-width. For each matching unit the user may either select from a number of suitable expanded module templates, or use a completely automated process. This method exploits the full power of hierarchical expansion, allowing multiple expansions within a single data path unit. Once complete, all instances of the specified unit type will be smaller than the bit-width threshold.

3. *Macro expansion* (MOODS command *aom)* searches the entire data path, expanding each macro operator and macro port in succession, and is always performed prior to creation of the final output netlist. It is also provided as a manual operation, allowing the user to further optimise the design if desired. This can lead to a higher level of integration in the control and data path, allowing additional operations to be scheduled

within the expanded sub-graphs, however it may also alter the pre-defined scheduling in some cases.

Details of the implementation and use of these three processes are discussed in the next section.

5.3 Implementation Details

Module expansion is integrated into MOODS via the *expanded module library*, which is responsible for the management and selection of expanded modules based on a target data path node. This section describes the implementation of both the library and the expanded modules themselves, opening with an overview of initial modifications made to the MOODS system, in order to develop an infrastructure around which the expanded module library is constructed. Following on from this is a discussion of the development and organisation of expanded module definitions (*expanded templates*), together with details of the expansion process and the various user interface routines. Further implementation details, including various file formats may be found in Appendix B.

5.3.1 Infrastructure Development

Prior to the development of the expanded module capability, an extensive range of improvements was carried out on various parts of the system. In addition to general developments, particularly of the estimation and netlist outputting sub-sections, fundamental changes were made to the instruction model, and the data path and module libraries.

ICODE instruction database. The ICODE instructions were originally hard-coded into MOODS itself, requiring extensive modification of the source code to include any new features. In addition, each instruction was classed as having either one or two inputs, with only one output, making them unsuitable for functions such as *add with carry* (3 inputs, 2 outputs), essential for module expansion (witness the earlier examples). To tackle these shortcomings, a new instruction database has been developed with the following features:

- A modular architecture is used which removes all hard-coded non-control instructions from the main system into a central instruction database. All definitions are stored in a text file enabling easy addition of new instructions.

- Instructions can have any number of input and output parameters specified in the database file. This facilitates the inclusion of such instructions as PLUSC (add with carry in and out) and complex macro operators (see Chapters 6 and 7).

- The database allows more flexible I/O parameter width relationships. Originally, the instruction "width" (which defines the associated data path unit width) was determined from the largest input variable, with the output width being either the instruction width, the instruction width plus one, or twice the instruction width. The new database allows any parameter (input or output) to specify the width, and incorporates a more comprehensive range of relationships including fixed widths and $\log_2$(width).

- Enhanced mappings between instructions and data path functions are now supported, allowing several instructions to perform the same basic function. For example, PLUS and PLUSC are both add operators, but they differ in their I/O parameter specification. This also extends to the module libraries which match both the function and pin requirements when selecting modules, thus a module performing an *add* operation with a carry input pin can implement both PLUS and PLUSC instructions, whereas an adder module without a carry pin can only implement PLUS.

Module library. In common with the ICODE instructions, the original cell library was hard-coded into the system. This has been replaced by an object-based module library which manages the storage, selection and modelling of many different *technology libraries* all accessed via a single common interface class, *MModuleLibrary*. *Technology libraries* encapsulate the supported modules and characteristics of any desired low-level layout system. These are all derived from a common base class, *MTechnoLib,* forming a standard library interface that includes access routines for module selection and performance estimation. Libraries may either be hard-coded, or based on the standard generic library which uses a *library database file* and provides performance modelling equations.

The module library itself (*MModuleLibrary*) treats all available technology libraries as a single homogeneous set of modules from which any valid implementation may be

selected, based upon function type and I/O pin requirements supplied by MOODS. This configuration allows several technology libraries to be used for a single synthesised design and enables small libraries of specialised modules (eg. a library of optimised multipliers) to be plugged in and out of MOODS according to the occasion. Indeed at the extreme, every module could be stored in a separate library (although this approach would be rather rash). Figure 5.7 is a block diagram illustrating the relationship between the technology libraries and the top-level module library object.

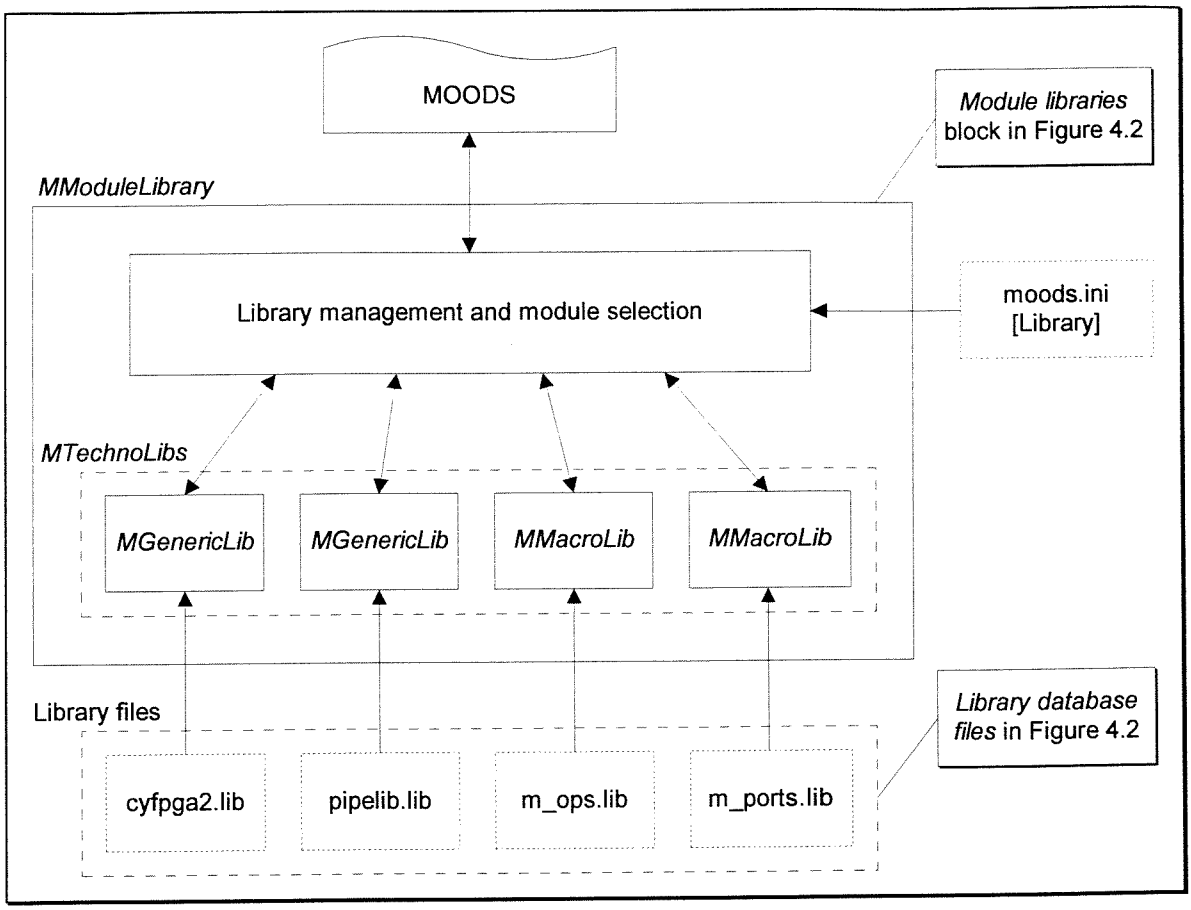


Figure 5.7 Module library block structure

The module library is configured from information in the *technology* sub-section of the MOODS initialisation file, **moods.ini**, illustrated in Figure 5.8. Each line describes the configuration of a technology library starting with the library type, which refers to the particular technology class, followed by a parameter list specific to the library in question. In the example, two generic libraries are included: an FPGA module library stored in file **cyfpga2.lib** containing all the standard modules; plus a specialised set of pipelined multipliers in **pipelib.lib** to augment the basic module set.

```
[Technology]
Cypress FPGA Library  = GenericLibrary, cyfpga2.lib
Pipelined Multipliers = GenericLibrary, pipelib.lib
```

Figure 5.8 Module library configuration in the initialisation file

MOODS requests a module through the *MModuleLibrary* interface, providing a list of functions to implement, a set of I/O pins (together with bit-widths), and a selection parameter. All active technology libraries[1] are searched, and one implementation chosen based on the selection parameter. This may be one of several types: select first implementation, select random, select smallest or select fastest; other parameter types may be readily added at a later date. The library is also able to provide a list of all suitable modules, leaving selection up to MOODS.

All module-specific information is maintained within the library, which returns a unique *module number* as a reference for the selected module. This is used as an identifier whenever MOODS interrogates the library to obtain performance data or other module information.

Extended ICODE format (.xic files). The extended ICODE file format (detailed in Appendix B) is a textual version of the standard binary ICODE file generated by VHDL2IC. The use of a user-modifiable format as direct input to MOODS has a number of advantages over just using behavioural VHDL:

- It is now possible to modify a behavioural description at the instruction level to provide greater control over the synthesis process. Used in conjunction with the ICODE instruction database, this also allows the inclusion of specialised instructions, maybe implemented by macro operators, without having to make any changes to VHDL2IC.

- Manual modification of the ICODE also provides a degree of control over the final scheduling of parts of the design through the inclusion of ICODE PROTECT instructions. These partition a design such that *instructions after the PROTECT are always guaranteed to be scheduled at least one cycle after all the preceding instructions*. For example, without the PROTECT, every instruction in Figure 5.9

[1] A technology library may be temporarily de-activated during synthesis to investigate implementations based on different module sets.

would probably be scheduled concurrently in a single control state. The PROTECT however, forces instructions *i4* and *i5* into a separate state as shown in the example schedule. This approach goes some way to tackling the problems encountered in the behavioural synthesis of VHDL processes, where the language itself provides little control over the detailed timing of instructions due to its simulation-based nature (see section 3.2). PROTECT instructions have proved invaluable in the development of the video processor design described in Chapter 7, which requires exact cycle-by-cycle specification of various I/O signals. They are also used in macro ports to enforce a particular delay during memory accesses (see Chapter 6).

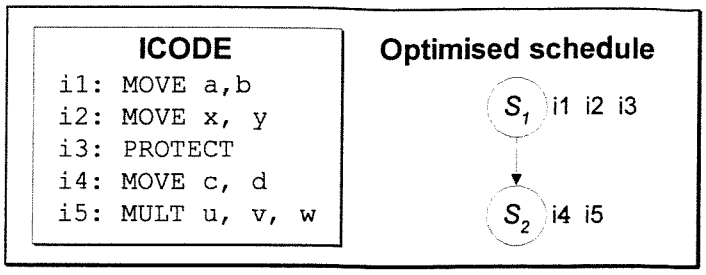


Figure 5.9 The effect of PROTECT instructions on scheduling

- Development of expanded modules is based on the construction of a behavioural description which is then pre-processed by MOODS (see section 5.3.2). VHDL descriptions contain a significant amount of excess baggage (signal shadow registers, wait statements, process loops) unnecessary in an expanded module, which may be removed by manually editing the ICODE file.

- New features for the VHDL front end can be investigated prior to coding them into the VHDL2IC source by manually simulating their effect in the ICODE file. This method has been used to develop macro operators and macro ports.

Design data structure format (.ddf files). The **.ddf** file is a textual equivalent of the original data structure (**.ds**) format. As shown in the system block diagram in Chapter 4 (Figure 4.2), MOODS allows the user to save a snapshot of the internal design representation in a *data structure* (**.ds**) file, containing a full behavioural and structural description of current state of the synthesised design. For reasons discussed later, the MOODS data structures are used as the basis for defining expanded modules, a task clearly suited to the data structure file, however there are a number of drawbacks to using the main **.ds** format:

- The **.ds** files include a substantial amount of information which may be directly derived from the bulk of the data structure, such as instruction dependencies and mutual

exclusion links. In addition, this information is likely to change when module expansion occurs and must therefore be dynamically re-calculated.

- Expanded modules require additional information not currently supported in **.ds** files, such as details of the data path function, bit-width and interface pins supported. It may also, in the future, require extra support for features such as parameterisation, allowing a single generic expanded module description to be automatically adjusted for different bit-widths.

- Finally, the **.ds** file is a binary format, and may not therefore be manually modified - a desirable feature allowing a designer to modify sections of the structure if so desired. A binary format is also more difficult to extend while maintaining backward compatibility.

For these reasons, the **.ddf** format has been developed and forms the basis for all expanded modules, as described below. Syntax details and an example may be found in Appendix B.

5.3.2 Developing Expanded Modules

Expanded modules are defined in an *expanded module template* (**.xmt**) file describing the control and data path sub-graphs comprising the structure and behaviour of the module. As mentioned above, the template is created from a behavioural description, first processed by MOODS and then output in a **.ddf** format file using the "save expanded module" (*sxm*) option. The use of MOODS to create these descriptions has a number of advantages:

- The **.xmt** file describes the control and data paths in a form directly compatible with the internal data structures in MOODS. This simplifies the expansion process ensuring that all the required information (assorted links, etc.) is present.

- Employing the standard MOODS design flow simplifies the whole process, allowing a designer to use the familiar MOODS tools, and to develop an expanded module at any desired level of abstraction.

Figure 5.10 illustrates the typical design flow for the creation of an expanded module template. There are three possible starting points for the description, each at a different level of abstraction. It is entirely possible to develop the template file from scratch,

manually adding each control node, control arc, data path node, variable, instruction etc. - a process akin to entering each binary digit in a machine code program. At a higher level, the behaviour of a module may be described in an ICODE file. The use of extended ICODE allows the designer to manually create the description at the instruction level, providing a degree of control over scheduling through the use of PROTECT instructions. This parallels the development of assembly code in software. Finally, at the top level, behavioural VHDL may be used, and it is this that forms stage one in the design flow.

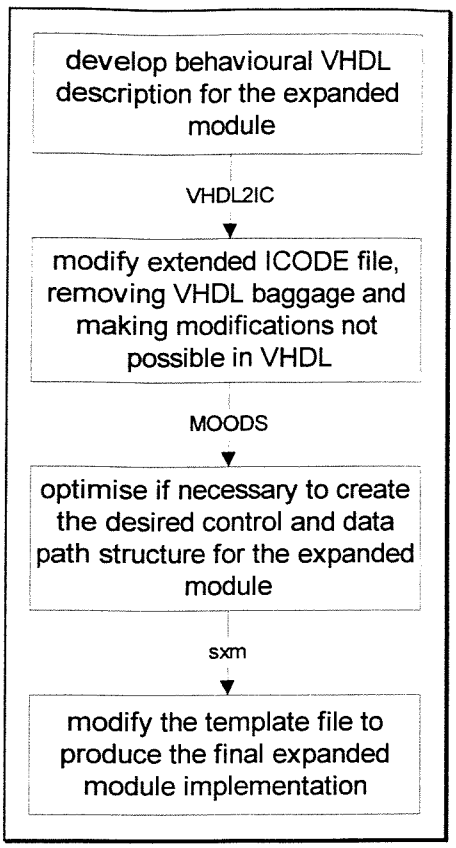


Figure 5.10 Expanded module creation design flow

The operation of an expanded module should be coded as a single VHDL process, in an architecture whose I/O ports match the pins of the combinational module targeted for expansion. The description should follow standard guidelines for behavioural synthesis (see Chapter 7), making as little use as possible of signals, except to output the final result at the end of the operation. The use of VHDL enables rapid development in a familiar language, which is easily simulated to ensure correct module operation. It does however pose a few problems due to its high-level nature:

- Describing the detailed structural topology of an expanded module can be difficult, requiring some knowledge of the likely optimisations performed by MOODS. It can also be a rather protracted approach, compared to a more traditional structural description (ie. a netlist). However, as indicated previously, there are a number of advantages to using a relatively simple description in order to allow MOODS maximum freedom to perform optimisation. The use of VHDL actively encourages this approach.

- There is a considerable amount of extra code associated with the implementation of the VHDL simulation cycle, such as shadow registers for signal assignment, sensitivity lists, process loops etc. (details may be found in section 3.2 and, specifically for MOODS, in Chapter 6). These features are not suitable for an expanded module which

is required to implement a relatively simple logical or arithmetic function, and should involve the minimum necessary overhead in terms of extra control and intermediate registers.

Once complete, the VHDL description is transformed, via VHDL2IC, into an extended ICODE representation which may then be manually modified to provide greater control over the minutiae of the implementation. The main modifications to be made are:

- Where appropriate, signal shadow registers, and any associated ICODE instructions (mainly due to VHDL *wait* statements), should be removed. This may be performed automatically using the **/no_sigs** switch on VHDL2IC which implements all signals as variables (see Chapter 6).

- Expanded modules must have unique, non-exitable, start and end control states, clearly defining where the operation begins and ends. VHDL processes, on the other hand, repeat indefinitely, therefore the process loop, including any implicit *wait* statement and sensitivity list, must be removed (again, see Chapter 6 for details of VHDL2IC).

A few words should also be said about the use of I/O ports and registers. In some cases, it may be necessary to copy the input values into intermediate registers at the start of the code. If the input values are modified during execution (eg. the block multiplier of [55] rotates its inputs), they must first be copied into intermediate registers. This is necessary because, when expanding a module, other operations may concurrently access the same input registers, thus it is essential to maintain their stability over the entire period of its execution. The same is not true of output registers which are guaranteed by the expansion process to only be required before or after module execution (and not during). A more detailed discussion of these topics may be found below in section 5.3.4.

Once the ICODE is complete, it is then loaded into MOODS as a standard behavioural description. The resulting initial control and data paths, implementing the ICODE with one cycle per instruction (see Chapter 4), may then be optimised to obtain the desired schedule and data path structure for the expanded module. It should be noted, however, that further optimisation of a design following module expansion may undo any of these optimisations, hence this task is of most use when creating macro ports, which are only expanded after optimisation. Once a suitable implementation is obtained, it is then saved

as an expanded module template (**.xmt**) in **.ddf** format. The *sxm* command prompts the user for the data path functions implemented by the expanded module, and the range of bit-widths supported. It also allows the I/O ports of the template to be mapped onto the abstract function ports.

The template file created in stage three is suitable for use as an expanded module in MOODS. It is possible, at this point, to make alterations directly to the template structure in order to slightly modify the implementation. However, this is rarely necessary as most requirements can be met either within the ICODE, or through manual manipulation of the design using optimising transformations within MOODS.

As illustrations of this process, there now follow two examples showing several stages in the development of a 32-bit split adder (as used in Figure 5.4) and a 32-bit block multiplier, (based on the architecture described by Theeuwen et. al. [55]). Further examples of macro operator and port development may be found in Chapters 6 and 7.

Figure 5.11 demonstrates the development of a 32-bit adder, split into two 16-bit sequential stages. This illustrates a number of pertinent points:

- One advantage of using VHDL is that it may be simulated during development to ensure correct operation of the expanded module, without having to go through the entire design flow. However, this necessitates the proper use of signals and wait statements, which are not needed in the expanded module template. The resulting overhead in the ICODE file, shown in Figure 5.11b must be manually removed producing the final ICODE description of Figure 5.11c. Notice also the removal of the implicit process loop formed by the loop-back activation of the original final instruction; the final version just ends on the last instruction.

- The lack of support in VHDL for low-level operations, such as add with carry (PLUSC) means that where these are needed, an alternative to the standard VHDL operators must be used. In the example, a special *plusc* procedure is created (definition omitted from Figure 5.11a), the call to which is changed to a PLUSC instruction in the ICODE .

- The final ICODE description is saved, unoptimised, as an expanded module template, with a bit-width of 32 bits. When expanded into a design, a structure similar to that shown earlier in Figure 5.4 is obtained (but only using two stages, as opposed to four).

```
entity add32 is
   port ( inp1, inp2 : in bit_vector (31 downto 0);
          outp : out bit_vector (32 downto 0)
        );
end add32;

architecture behaviour of fixed_point_mult_12_10 is
begin
  add32: process
    -- input word bit slices - 32 to 16 bits
    alias inp1lo : bit_vector(15 downto 0) is
                                inp1(15 downto 0);
    alias inp1hi : bit_vector(15 downto 0) is
                                inp1(31 downto 16);
    alias inp2lo : bit_vector(15 downto 0) is
                                inp2(15 downto 0);
    alias inp2hi : bit_vector(15 downto 0) is
                                inp2(31 downto 16);
    alias outplo : bit_vector(15 downto 0) is
                                outp(15 downto 0);
    alias outphi : bit_vector(16 downto 0) is
                                outp(32 downto 16);

    -- intermediate variables
    variable carry, dummy : bit;
  begin
    -- add 32 bits as 2 cascaded 16-bit adds
    -- assumes the existence of a plusc procedure
    plusc(inp1lo, inp2lo, '0', outplo, carry);
    plusc(inp1hi, inp2hi, carry, outphi, dummy);
    wait for 100 ns;
  end process;
end behaviour;
```

a) VHDL source code (without plusc procedure)

```
PROGRAM   add32  inp1, inp2,  outp

// ***** I/O port declarations
INPORT    inp1  [0:31]
INPORT    inp2  [0:31]
OUTPORT   outp  [0:32]

// ***** variable/register declarations
REGISTER  inp1_tempsigin [0:31]
REGISTER  inp2_tempsigin [0:31]
REGISTER  outp_tempsig   [0:32]
REGISTER  carry          [1:1]
REGISTER  dummy          [1:1]

ALIAS  inp1lo [0:15]  FROM  inp1_tempsigin [0:15]
ALIAS  inp2lo [0:15]  FROM  inp2_tempsigin [0:15]
ALIAS  outplo [0:15]  FROM  outp_tempsig   [0:15]
ALIAS  inp1hi [0:15]  FROM  inp1_tempsigin [16:31]
ALIAS  inp2hi [0:15]  FROM  inp2_tempsigin [16:31]
ALIAS  outphi [0:16]  FROM  outp_tempsig   [16:32]

       MOVE  inp2,  inp2_tempsigin
       MOVE  inp1,  inp1_tempsigin

.add32 MODULEAP  plusc  inp1lo, inp2lo, #0, \
                        outplo, carry
       MODULEAP  plusc  inp1hi, inp2hi, carry, \
                        outphi, dummy

       // ********** wait **********
       MOVE  outp_tempsig, outp
       MOVE  inp2, inp2_tempsigin
       MOVE  inp1, inp1_tempsigin       ACT add32

       ENDMODULE add32
```

b) Directly compiled ICODE (without plusc)

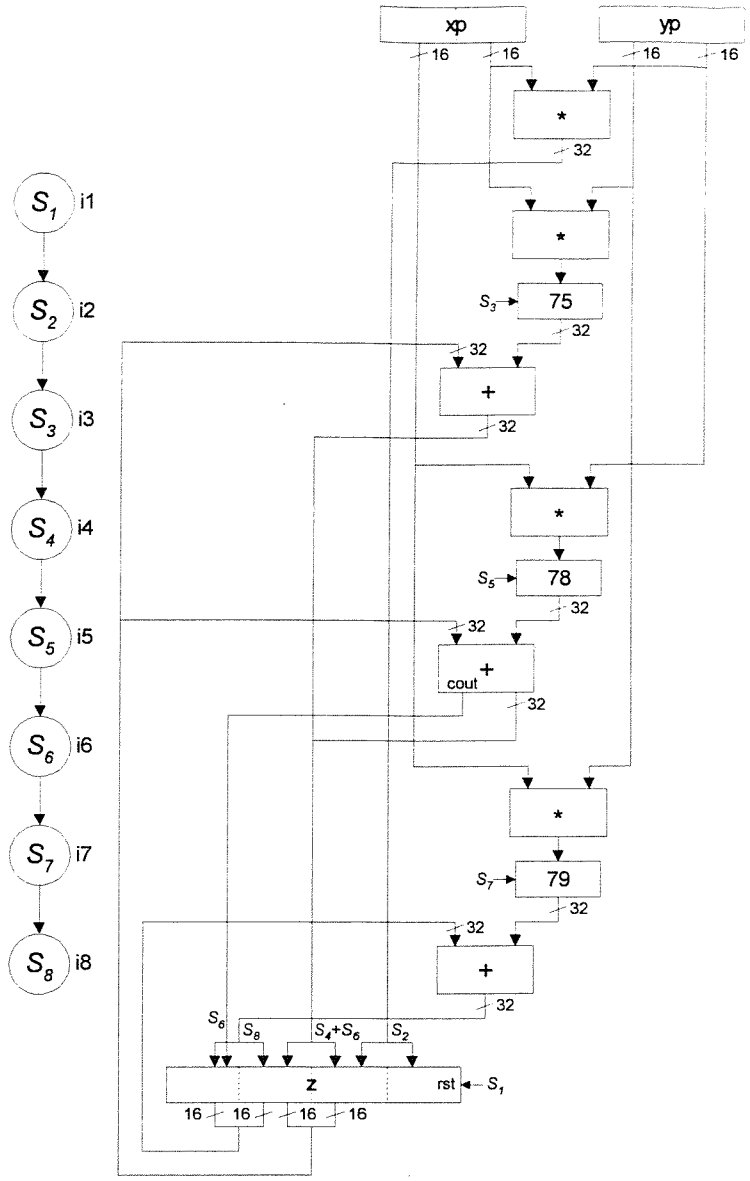
```
PROGRAM   add32  inp1, inp2, outp

// ***** I/O port declarations
INPORT    inp1  [0:31]
INPORT    inp2  [0:31]
OUTPORT   outp  [0:32]

// ***** variable/register declarations
REGISTER  carry  [1:1]
REGISTER  dummy [1:1] // dummy carry out
ALIAS  inp1lo [0:15]  FROM  inp1 [0:15]
ALIAS  inp1hi [0:15]  FROM  inp1 [16:31]
ALIAS  inp2lo [0:15]  FROM  inp2 [0:15]
ALIAS  inp2hi [0:15]  FROM  inp2 [16:31]
ALIAS  outplo [0:15]  FROM  outp [0:15]
ALIAS  outphi [0:16]  FROM  outp [16:32]

// ***** 32 to two 16-bit adders
.add32 PLUSC  inp1lo, inp2lo, #0, \
                      outplo, carry
       PLUSC  inp1hi, inp2hi, carry, \
                      outphi, dummy

       ENDMODULE add32
```

c) Final expanded module ICODE

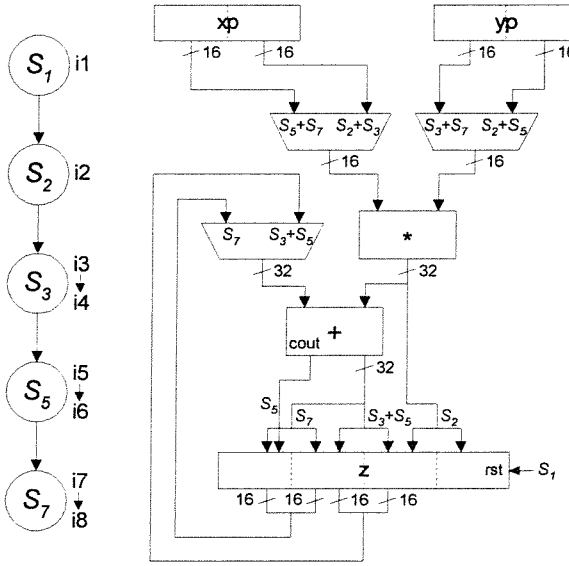
Figure 5.11 Split adder expanded template development

```
PROGRAM  blkmult32  xp, yp, z

// ***** I/O port declarations
INPORT  xp [0:31]
INPORT  yp [0:31]
OUTPORT z  [0:63]

// input block aliases
ALIAS xa [0:15] FROM xp [0:15]
ALIAS xb [0:15] FROM xp [16:31]
ALIAS ya [0:15] FROM yp [0:15]
ALIAS yb [0:15] FROM yp [16:31]

// output register aliases
ALIAS za [0:31] FROM z [0:31]
ALIAS zb [0:31] FROM z [16:47]
ALIAS zc [0:31] FROM z [32:63]
ALIAS c  [0:0]  FROM z [48:48]

.mult MOVE  #0, z

      MULT  xa, ya, za

      MULT  xa, yb, 75
      PLUS  75, zb, zb

      MULT  xb, ya, 78
      PLUSC 78, zb, #0, zb, c

      MULT  xb, yb, 79
      PLUS  79, zc, zc

      ENDMODULE  blkmult32
```

a) ICODE description



b) Initial post-expansion control and data paths



c) Area optimised configuration

Figure 5.12 Block multiplier expanded template development

The block multiplier example of Figure 5.12 shows how a complex module structure may be implemented in a more abstract form in such a way that optimisation after expansion yields the desired topology. The behavioural description in Figure 5.12a is based on the block multiplier architecture of [55] shown in Chapter 3 (Figure 3.2), slightly modified to use aliases in lieu of shift registers. For this example, the ICODE has been created from scratch, thus requiring the module to be synthesised before it can be simulated.

As described in Chapter 3, the block multiplier splits the input words into a number of blocks, each pair of which is multiplied and summed over several cycles. This version splits the 32-bit inputs into two blocks of 16 bits each. The entire operation thus requires four cycles to complete. Figure 5.12b shows the initial control and data path structure obtained immediately after module expansion, where each original instruction requires one cycle. Area optimising this configuration results in maximal sharing of functional units (leaving one multiply and one add), and merging of control nodes, as shown in Figure 5.12c. Comparing this to the original block multiplier structure (Figure 3.2) clearly demonstrates their correspondence, bearing in mind that register rotation is now emulated using multiplexors. Note that although only a fixed block size is used, exploiting hierarchical expansion to further split the 16-bit multiplier using two 8-bit blocks, results in the same structure as a 32-bit multiplier with block size 8 bits. Thus the user may investigate the effects of varying the block size by altering the maximum bit-width threshold (during *aob* expansion).

5.3.3 Template Library Organisation

MOODS interacts with the expanded module templates via the template library. This is responsible for maintaining a database of all available templates, and selecting a suitable one when requested, based on a target data path node to be expanded. Figure 5.13 depicts the library block structure showing its composition, and the relationships between the various templates, the library itself and MOODS.

Each expanded template is created from a single **.xmt** file, and is internally represented by an *ExpandedTemplate* object, holding a complete expanded module description based on a simplified control and data path. Templates are grouped into sets (*ExpTemplateSet*), each containing a number of topologically identical expanded modules, with differing bit-

widths. The selection process chooses, at most, one template from each set, selecting the one with the closest bit-width to the data path node being expanded. Sets incapable of implementing the required data path function are ignored. The resulting list of suitable templates is either returned to MOODS for further processing, or whittled down to a single entry according to a specified selection parameter. Current options are "select a random template", or just "select the first".

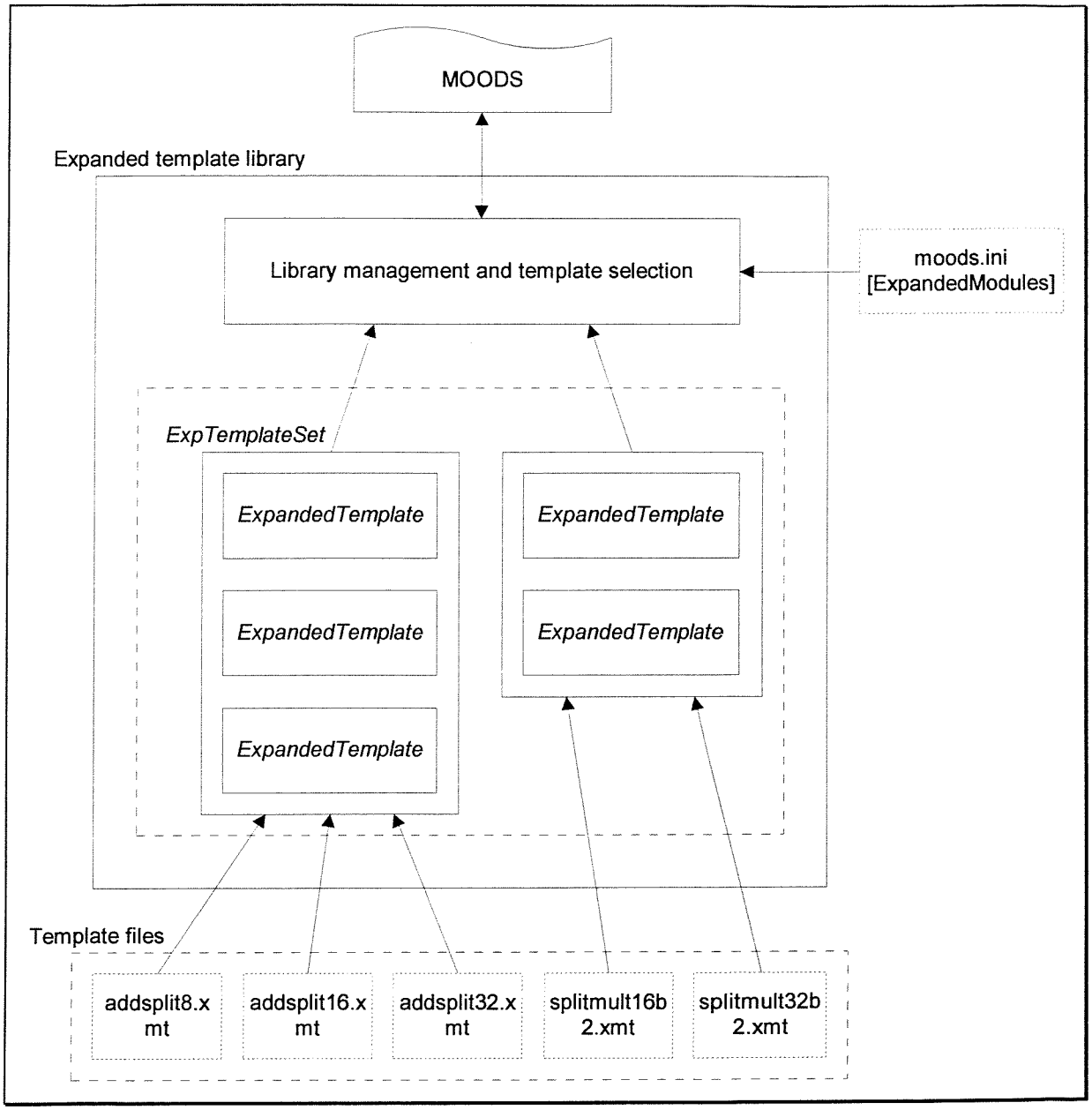


Figure 5.13 Template library block structure

The contents of the library is defined in the *ExpandedModules* section of the MOODS initialisation file, **moods.ini**, a section of which is illustrated in Figure 5.14. This contains one entry per template set, listing its constituent template files. In the example, the following sets are defined: a split adder supporting 8, 16 and 32-bit widths, and a block

multiplier supporting 16 and 32 bits. Note that each template file specifies a range of supported widths, so for example, if **splitmult32b2.xmt** covers from 25 to 32 bits, any module in this range would be expanded using the 32-bit template. The expansion process (examined below) re-sizes all necessary external components to exactly match the I/O widths. Details of the initialisation file options may be found in Appendix B.

```
[ExpandedModules]
SplitAdder    = addsplit8.xmt, addsplit16.xmt, addsplit32.xmt
BlockMultiply = splitmult16b2.xmt, splitmult32b2.xmt
```

Figure 5.14 Template library configuration in the initialisation file

5.3.4 The Expansion Process

Central to the implementation of expanded modules is the *split module* transformation. This operates in a similar manner to the other MOODS transformations, splitting into the three basic stages of Figure 5.15:

1. The testing stage, performed by function *test_split_module_t*, simply checks whether the selected data path node represents a functional unit, and ensures that at least one suitable template exists in the template library.

2. Selection of a template can either be manual or automatic. Manual selection returns to the user a list of all templates compatible with the data path node in question. From this, one is chosen with which to proceed with the expansion. Automatic operation uses the template library selection parameter to return either the first, or a random template.

3. Given a data path node, and a valid expanded template, the *split_module* routine is called to perform the expansion, updating the control and data path graphs to include the new elements.

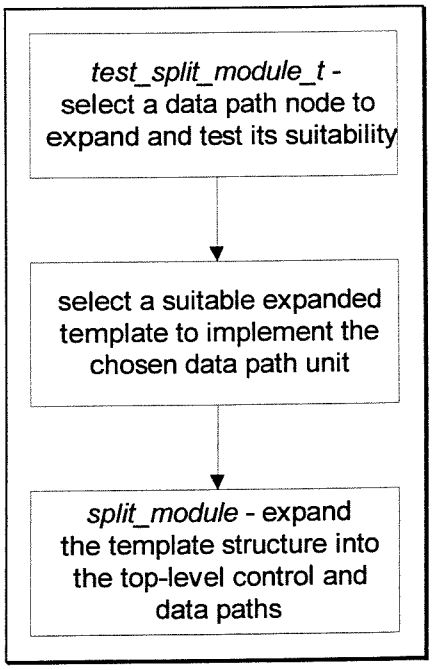


Figure 5.15 Applying the split module transformation

Note that no estimation stage is implemented at present as it will generally result in an increase in both delay and area. This transformation should not be regarded simply as a

means for directly improving the cost function, but as a mechanism for increasing the scope of future optimisation. As is demonstrated in the results, section 5.4, the multitude of synthesis options resulting from this increased scope makes predicting the effect of a module expansion on the final design implementation impossible, without actually synthesising to completion. Estimation, therefore, is of little use.

The transformation routine itself splits into two steps: preparation of the control and data paths, followed by the actual expansion of the expanded template.

Before expansion may proceed, each ICODE instruction implemented by the target data path node must be isolated in its own control state. This has two important effects: a) all instruction dependencies are made explicit in the ordering of control states; and b) the instruction isolation ensures that that no operator chaining occurs, thus the input and output pins of the target data path unit all connect directly to registers (important for maintaining I/O integrity over several cycles). Note that since data path unit sharing results in multiple ICODE instructions being implemented by a single data path node, it is necessary to replace every shared instruction when expanding the template.

Once the data structures have been prepared, *expand_ExpandedModule* is called to realise the expanded template sub-control and data paths within the main design description. The template holds a mirror of the MOODS data structures for implementing the expanded module, ie. all control nodes, control arcs, ICODE instructions, variables, data path nodes, nets and control equations. The task of *expand_ExpandedModule* is to replicate these elements, replacing the original data path unit with a new expanded module structure, customised for the particular target configuration (I/O registers, activation conditions etc.). Data path unit sharing is maintained as the operation creates just one set of expanded data path nodes, but many sets (one per instruction) of expanded control nodes, instructions, variables and data path interconnects (nets).

This is illustrated in Figure 5.16 showing two add instructions implemented on a single shared adder (Figure 5.16a), which is expanded using a two-stage split adder template (as in Figure 5.11). The result is two images of the expanded control structure for each original instruction (S_2, S_5 and S_3, S_6), but only one set of expanded data path nodes. These are joined however, by a set of nets for each expanded instruction, hence the doubling-up of the multiplexing structure in Figure 5.16b.

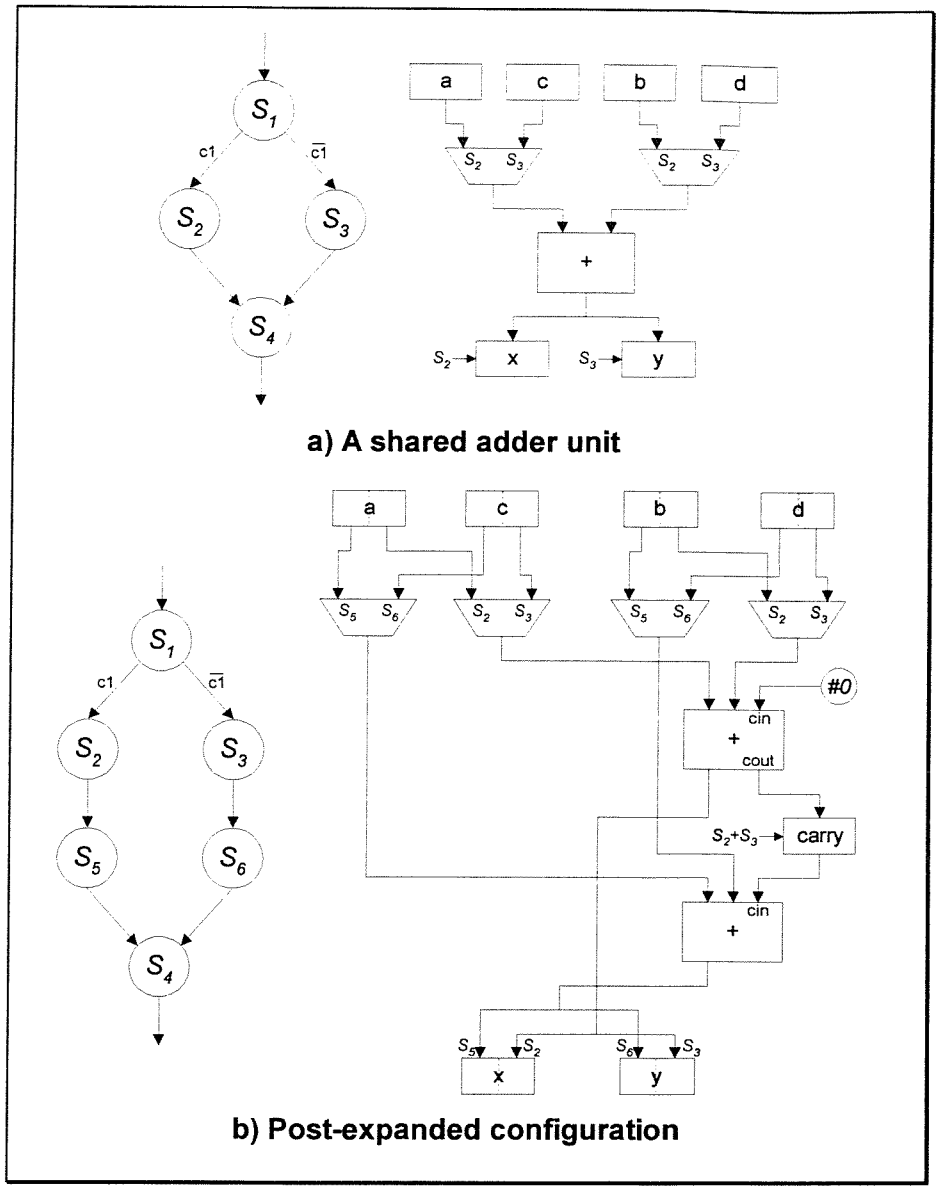


Figure 5.16 Expanding shared data path units

In general terms, expansion is based on the creation of an *ExpandedModule* object which maps each template entry onto a newly created MOODS data structure member. This is a one-to-many mapping where every template element is replicated for each target ICODE instruction (except for data path nodes, only replicated once). When the internal template structure has been constructed, the entire expanded module is inserted into the top-level data and control paths by mapping module I/O ports onto the original I/O registers, and connecting the start and end control nodes to the input and output arcs of the target state. The *ExpandedModule* object keeps track of which design elements implement each part of the template, thus at any point after expansion, it is possible to identify the constituent parts of a particular expanded module, even if they have been extensively optimised

(theoretically allowing the expansion to be undone at a later date, although this is not currently implemented).

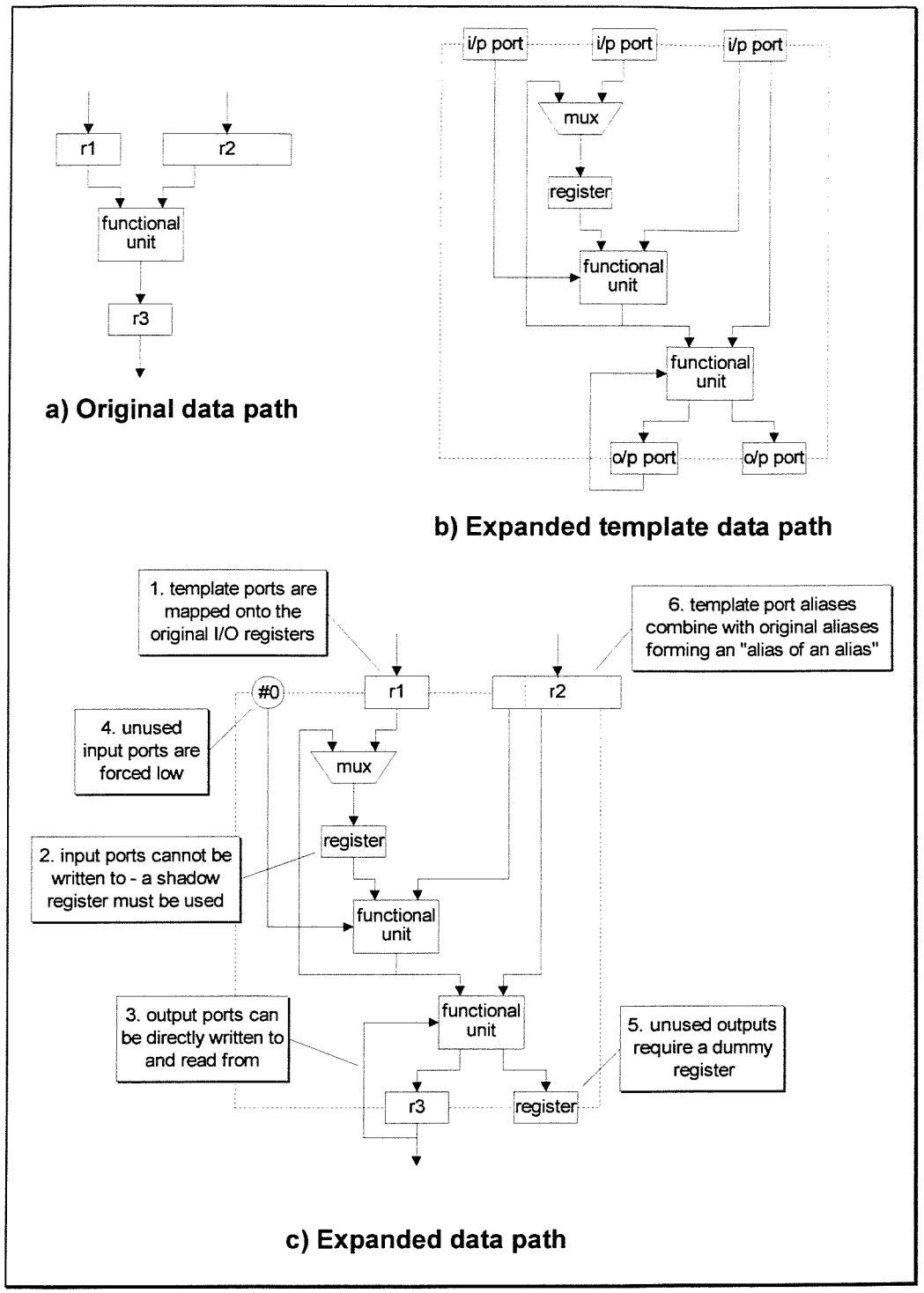


Figure 5.17 Example of data path expansion

Figure 5.17 demonstrates the replacement of a simple data path unit with an equivalent expanded module data path sub-graph. This example illustrates a number of important factors which concern the interfacing of the expanded sub-graph to the I/O registers

connecting to the original unit. These must be carefully considered to ensure the correct operation of the module within a complete system.

As is shown in Figure 5.17c, the I/O ports of the expanded module (Figure 5.17b) are mapped directly onto the I/O registers of the original data path unit (Figure 5.17a); a process akin to passing software procedure parameters by reference. The central issue of concern with this approach is the stability and validity of input and output registers, with respect to both the internal operation of the module, and the rest of the system. Four rules can be identified which must be adhered to, in order to ensure consistent operation after module expansion:

1. *Inputs must remain stable if required throughout the entire execution of an expanded module*. The point of this condition is that no other operation must be able to alter the input registers until the expanded module has either completed execution, or the input values are no longer required. For example, the split adders shown earlier require the input to be valid during each of their control states, whereas the pipelined multiplier (Figure 5.6) reads the inputs only once, at the start of execution. A pathological example is shown in Figure 5.18a. Here, all three instructions may be scheduled in a single state by chaining *i1* and *i2* thus bypassing register *b*. The preparation stage for expanding the adder in *i2* requires the instruction to be isolated in a single control state, resulting in the schedule of Figure 5.18b. This is obtained thus: the dependence of *i2* on the result of *i1* forces *i1* into a preceding state; *i3* on the other hand, must occur after *i2*,

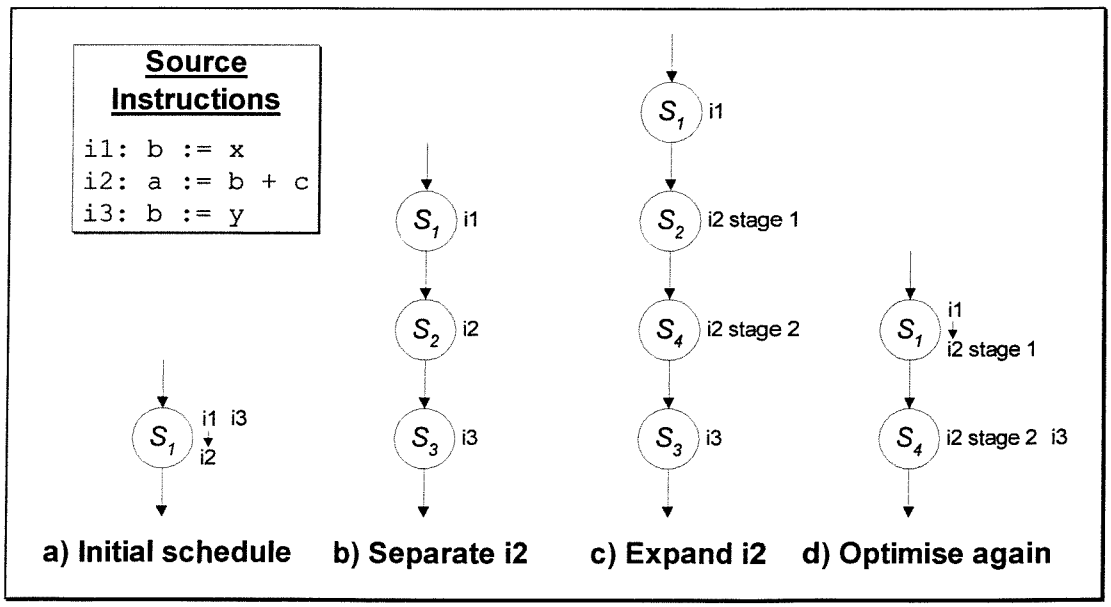


Figure 5.18 Example expanded schedule demonstrating input stability

any other configuration would create a dependency between *i2* and *i3* (on variable *b*) which does not exist. The state after expansion, Figure 5.18c, shows that input *b* remains stable throughout execution of the module. In addition, after further optimisation, the configuration of Figure 5.18d results. This illustrates how MOODS utilises dependencies within the module to obtain a highly efficient schedule: *i1* and *i2 stage 1* are chained, bypassing register *b*; however *i1* will still update the register allowing *i2 stage 2* to read the correct stable value. In addition, *i3* is scheduled concurrently with *i2 stage 2* as *b* is only required to be stable until the end of the state, at which point it is updated with its new value.

2. *Inputs must remain stable if required by concurrent operations external to the module.* This condition requires that the expanded module must not alter the contents of the inputs during execution as they may be required by other concurrently executing operations. An example would be the block multiplier described in Chapter 3, which rotates its inputs on each clock cycle. If this feature is required, a shadow input register must be explicitly provided as in Figure 5.17c, note 2. This behaviour is automatically enforced when creating an expanded template since ICODE input ports may only be read: write access would result in an ICODE parse error.

3. *Outputs must be valid, and remain stable when required by dependent operations.* Sequential modules tend to use output registers to build up a partial result over several cycles before obtaining a final value. The system must, therefore, guarantee that a) the output value is not read until after execution, and b) earlier instructions reading the

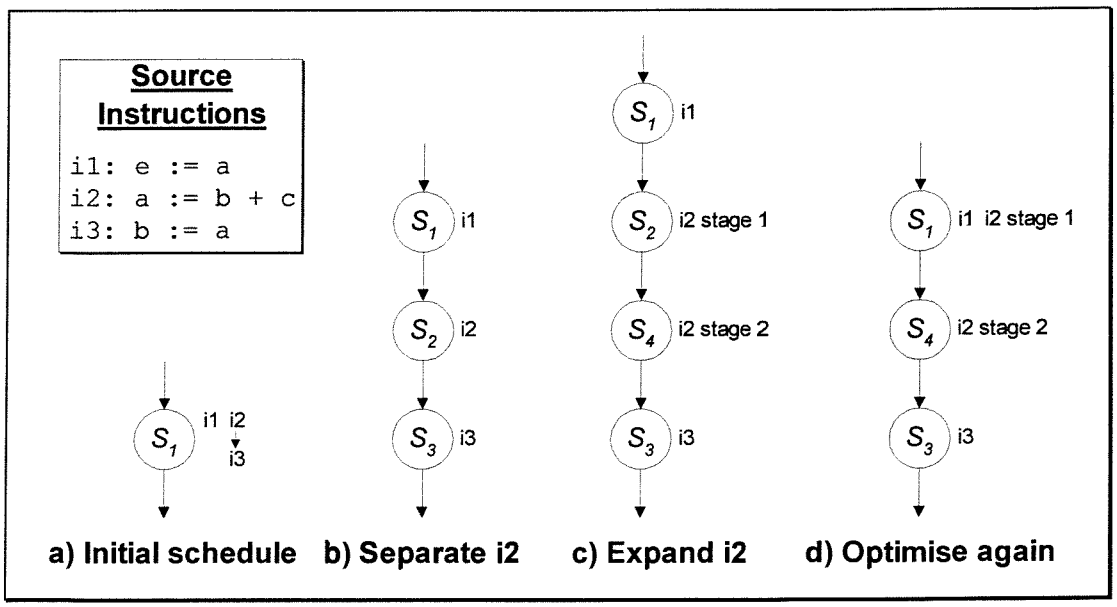


Figure 5.19 Example expanded schedule demonstrating output stability

previous output value cannot be scheduled during (or after) the expanded operation. Considering the example of Figure 5.19a, all three instructions may be scheduled concurrently in a single state with *i2* and *i3* chained due to the dependency on variable *a*. When preparing to expand *i2*, the control graph of Figure 5.19b will be obtained because: *i3* must occur after *i2* as it depends on the result, *a*; and *i1* must occur before *i2* since it requires the value of *a* before *i2* modifies it. Thus, after expansion (Figure 5.19c), the output value *a* is not accessed externally during module execution, and may therefore be used to build up intermediate results. This applies, even after further optimisation (Figure 5.19d): *i1* and *i2 stage 1* may be scheduled concurrently as *a* will not change until the end of the state, however *i3* cannot be chained with *i2 state 2* since *a* is built up in a register over two cycles and cannot therefore be bypassed.

4. *Input and output values must be valid if accessed by operations in concurrent control branches.* Finally, the system must ensure that instructions executing in concurrent control threads do not modify the input registers, or read from the output registers during the operation of an expanded module. There are two possible scenarios: *implicit concurrency* is created by MOODS where two groups of operations are completely independent, while *explicit concurrency* is specified by the user via concurrent VHDL processes (see Chapter 6). The former situation poses no problems for module I/O stability as all operations are guaranteed to be independent. The latter however, allows inter-process communication via common VHDL signals, implemented as global ICODE variables. The concurrency model supported by MOODS (discussed in Chapters 6 and 7) does not force processes into lockstep (as defined by the VHDL simulation cycle), and thus requires explicit synchronisation via handshaking. Since all communicating variables must be updated prior to synchronisation, the stability of both input and output registers is guaranteed. Note that care must always be taken to ensure proper two-way handshaking if reliable and predictable system operation is to be achieved.

Other important points illustrated in Figure 5.17 are:

- Unused input pins are connected to a constant *#0* input (Figure 5.17c, note 4). All expanded modules are written such that any inputs liable to be unconnected (eg. adder carry pins) default to *#0*.

- Unused output pins result in the creation of a dummy register (note 5). This is provided because the output may be used within the expanded module as an intermediate variable (note 3). Following optimisation, any registers without output nets (not read from) are removed, thus if this dummy is not actually required, it will be automatically eliminated.

- Aliasing of ports is common in expanded modules due to their "block" nature. When one of the original ports is also aliased (*r2* in Figure 5.17a), this must be taken into account when creating the input nets within the expanded module (note 6). For example, if the template of Figure 5.17b accesses bits 15 to 8, and 7 to 0 of its input port, and the original unit accesses bits 31 to 16 of *r2*, the expansion process must map the two template aliases onto bits 31 to 25, and 24 to 16 of *r2*.

- If an output register is shorter than the required template port, it is widened to accommodate the necessary bits. This is important as those extra bits may be vital for storing intermediate values within the module. If an input register is too short, then the connecting nets are simply shortened accordingly. Any nets completely outside the input width are connected to a constant #0 input (as in note 4).

5.3.5 Automated Module Expansion

MOODS provides the user with three methods for expanding data path units, introduced in section 5.2.3. The simplest involves the manual operation of the *split module* transformation where the user specifies a single data path node, and the template with which it is to be expanded. This is best suited to minor design modifications or investigating the global effects of a single module expansion. A more automated approach is provided by the *automatic module expansion* (*aob*) and *macro expansion* (*aom*) routines.

The MOODS *aob* command accesses the *automatic module expansion* algorithm which is geared towards the general expansion of arithmetic functional units with their sequential equivalents. It is the only mechanism to exploit full hierarchical expansion of modules allowing multiple levels of expansion within a single module implementation. The basic algorithm is shown in the flow chart of Figure 5.20. This begins by prompting the user for a function type and bit-width threshold forming the main algorithm control parameters.

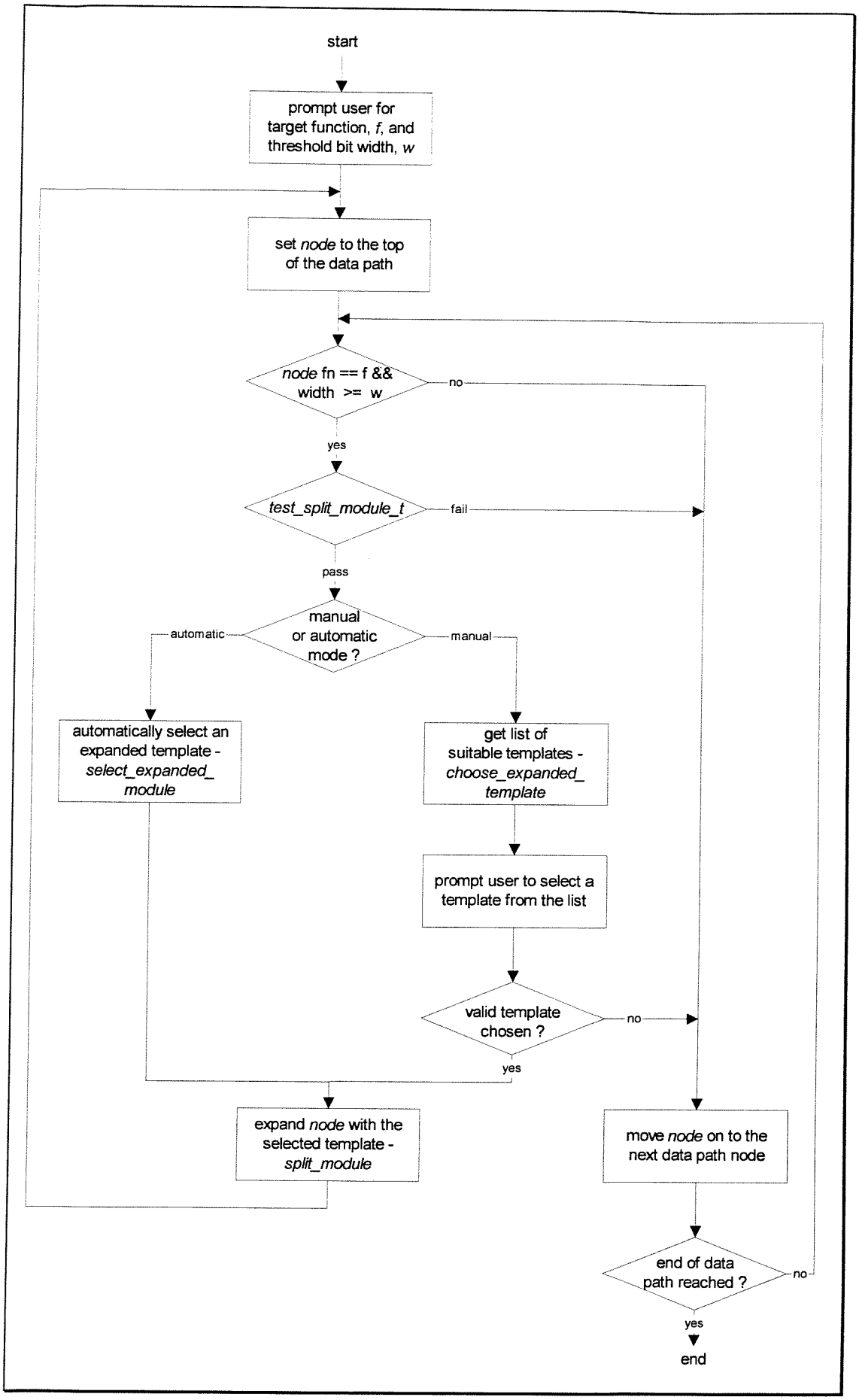


Figure 5.20 Flow chart for the automatic module expansion algorithm

The entire data path is then processed, expanding all nodes which implement the specified function type, with a width greater than or equal to the bit-width threshold. Each time a successful expansion is performed, the process restarts from the top of the data path thereby processing all new nodes created by earlier expansions. The algorithm has two modes of operation: in automatic mode, the first suitable template encountered in the library is used without any further user intervention; in manual mode, the user is prompted to select from a list of suitable templates, the chosen entry being used to expand the node. The user may alternatively choose to skip the current node without performing any expansion, in which case the algorithm then proceeds to the next target node.

The design of the algorithm centres around the requirement for an automated method allowing full control over the expansion of individual units, in order to investigate the effects of the various options available (which template to use, which bit-widths to expand etc.). This is based on the following general observations:

- The behavioural designs encountered contain only a handful of functional units suitable for expansion. The majority tend to be biased toward control applications and therefore contain a large proportion of control logic, interspersed with functional units. This is especially true after optimisation which tends to drastically reduce the number of units of a particular type. Even delay optimised designs contain at most around five instances of suitable functional units. The manual approach, therefore, is entirely feasible under these circumstances.

- The size of typical designs processed by MOODS is further limited when targeting relatively small FPGA devices. Experience with various designs (see Chapter 7) suggests that under these circumstances, maximal functional unit sharing is essential. This limits the number of nodes suitable for expansion to one or two for each function type, thus giving further credence to a manual approach.

- Area optimisation tends to result in maximal sharing of data path units. Designs with a large bit-width range for a particular function type do not make optimum use of either delay or area, since all operations will be implemented by the largest unit. For example, in a design containing 8, 16 and 32-bit multipliers, maximal sharing will use a 32-bit implementation for each operation, which is inefficient for the 8 and 16-bit operations. The *aob* bit-width threshold effectively specifies the maximum permitted width for a given function type, it can thus be used to limit the range of unit sizes in a

design, thereby improving the final utilisation efficiency. So, in the earlier example, the 32-bit multiplier could be split into two 16-bit operations resulting in a final implementation containing one 16-bit multiplier. Of course, the effect this has on the overall cost function will depend on the particulars of the design.

The *macro expansion* routine is based on a simplified version of the *aob* algorithm used to automatically expand all *macro operators* and *macro ports* within a design. It is assumed that a single expanded template exists for any one macro type, thus no user intervention is required and the process operates automatically. This routine may be executed via the MOODS *aom* command, but is always called at the end of optimisation to ensure that no macro modules are left unexpanded.

5.4 Results and Analysis

It should be evident from the earlier discussion that integrating module expansion into the general synthesis design flow is by no means a trivial task. As has been already stated, the module expansion process must be viewed as a mechanism for enhancing the scope for optimisation, rather than an improvement in its own right. This poses the questions: which modules should be expanded, by how much (ie. what bit-width threshold), and at what stage in the optimisation loop? The results presented in this section demonstrate the effect various module expansions have on the optimisation of several behavioural benchmark designs [106, 107]. Analysis of these figures shows how the choices made during optimisation affect the final outcome, and examines the operation of the two main algorithms (heuristic and simulated annealing) and the efficacy of the expansion process in general.

From the main set of benchmarks examined, four have been selected for in-depth analysis. These designs contain a substantial range of functions of various bit-widths, and are largely data-flow (as opposed to control) oriented. Table 5.1 describes the four designs in terms of their source code size, and the type and width of operations required. They comprise: **DHRC2**, implementing the differential heat release algorithm to model the heat release within an internal combustion engine; **FFT2** which calculates a fast Fourier transform using 32-bit integer arithmetic; **diffeq**, a differential equation solver, again using 32-bit integers; and **ellip**, which describes a 16-bit digital elliptical filter. Full VHDL

source listings may be found in Appendix B. Throughout this section, all performance figures are taken directly from MOODS using the **cyfpga2.lib** module library. This models area and delay based on an analysis of the output obtained from the Warp FPGA development system [93]. Table 5.2 lists example area[1] figures for a number of important modules, together with the modelling equations implemented.

Design name	Lines of VHDL	ICODE insts	Operators (quantity x bit-width)								
			plus	Minus	mult	eq	ls	le	ne	lshift	rshift
Dhrc2	112	46	4 × 16 1 × 32	1 × 16 1 × 32	4 × 16 1 × 32		1 × 32		1 × 1		1 × 16 5 × 32
Diffeq	108	36	2 × 32	2 × 32	6 × 32	2 × 1	1 × 32				
Ellip	100	50	26 × 16		8 × 16				1 × 16		
Fft2	157	96	7 × 32	6 × 32	4 × 32		7 × 32	1 × 32	1 × 1	2 × 32	3 × 32

Table 5.1 Benchmark design size and operator usage figures

Bit-width (n)	Register	2 input multiplexor	Ripple-carry adder	Multiplier
8 bit	800 μm^2	240 μm^2	1200 μm^2	13803 μm^2
16 bit	1600 μm^2	480 μm^2	2400 μm^2	55275 μm^2
32 bit	3200 μm^2	960 μm^2	4800 μm^2	221163 μm^2
64 bit	6400 μm^2	1920 μm^2	9600 μm^2	884715 μm^2
Model	100n μm^2	30n μm^2	150n μm^2	216n^2 - 21 μm^2

Table 5.2 Sample module sizes for various bit-widths

Each design has been synthesised using a variety of optimisation configurations featuring different expansions of the main arithmetic modules (multiply, add and subtract), and various positions for the expansion operation within the synthesis design flow. These are compared to the non-expanded implementations based on heuristic optimisation with area or delay at priority one, and targets of zero, ie. the minimum area and delay (non-multicycled) obtainable. Some multicycled optimisations are also included showing the lowest total delay possible through a reduction in the clock period to minimise idle execution time (see Figure 5.1). Tables 5.3 to 5.6 summarise a handful of optimisations for each of the four benchmarks providing a whole host of area and delay figures. Each design is optimised several times, mainly using the heuristic algorithm. At various stages in this process, all modules of a particular functional type and bit-width are expanded (or "split") using the MOODS *aob* command. Particular attention is paid to the difference between

[1] Size in μm^2 is the number of FPGA cells required, multiplied by 100 to give a reasonable dynamic range.

pre-splitting, that is, expanding modules prior to the first optimisation pass, and *post-splitting*, where an initial heuristic optimisation is performed before any expansion takes place. For example, the configuration tabulated in row three (*A2*) of Table 5.3 is obtained by first expanding all multipliers with bit-width threshold 32 bits, followed by heuristic area optimisation; row seven (*A6*) on the other hand, describes a post-split where first heuristic area optimisation is performed, then the multipliers are expanded, all of which is finally followed by a second area optimisation pass.

Each configuration is given a reference code based on the target optimisation type, eg. *A1* and *A7* for the two area optimised configurations above. The total execution time of all the optimisation stages is also provided, based on run times obtained from a P5-100 machine with 32MB memory, running MOODS version 2.6d on Windows NT 4.0. The total area and delay figures reported by MOODS form the basic performance results, together with the percentage change compared to the basic non-expanded implementations, ie. area optimised designs are compared to *A1*, and delay optimised compared to *D1*. Breakdowns of these overall figures show the area in terms of the percentage occupied by functional, storage, interconnect and control components, together with the quantity and bit-width of the major arithmetic modules used; and the delay as the *critical path length*[1] and minimum clock period permitted.

The results are summarised by a number of graphs in Figures 5.21 to 5.32. The first group, Figures 5.21 to 5.24, show sections of the area-delay design space for each design, covering the tabulated design points. Note that in the interests of clarity, the unoptimised implementations are not included. Figures 5.25 to 5.32 compare the area and delay breakdowns of the various configurations for each design.

The following sections discuss a number of pertinent points regarding the results, both generally, and specifically to each design.

---

[1] The critical path length defines the maximum number of control steps, including loop iterations, in a single execution of the design. In the case of VHDL, this is the time taken by one iteration of the longest process.

Optimisation configuration	Ref	CPU (secs)	Area (μm^2)	Δ area cf. A1,D1 (%)	Area Breakdown (%)				Module Usage	Delay (µs)	Δ delay cf. A1,D1 (%)	CPL (states)	Clock (ns)
					Func	Store	Inter	Cont					
unoptimised	N	-	535526	-	87.3	11.7	0.3	0.8	4x(+,16) 1x(+,32) 1x(-,16) 1x(-,32)	10.8	-	42	257.3
area optimised	A1	1	273836	0.0	85.0	8.8	5.6	0.5	1x(+/-,32) 1x(*,32)	6.1	0.0	15	409.5
pre-split (*,32), area optimised	A2	2	117563	-57.1	56.9	24.6	16.9	1.6	1x(+/-,32) 1x(*,16)	5.6	-9.2	19	293.6
pre-split (*,16), area optimised	A3	28	94146	-65.6	26.9	34.1	33.5	5.4	1x(+/-,32) 1x(*,8)	9.8	59.4	51	192.0
pre-split (*,32), (+,32) & (-,32), area optimised	A4	3	118944	-56.6	52.9	25.9	19.2	2.0	1x(+/-,16) 1x(*,16)	6.4	4.2	24	266.6
pre-split (*,16), (+,32) & (-,32), area optimised	A5	34	96622	-64.7	22.1	36.7	35.3	5.8	1x(+/-,16) 1x(*,8)	9.2	50.4	56	165.0
post-split (*,32), area optimised	A6	6	154988	-43.4	43.1	30.0	24.6	2.3	1x(+/-,32) 1x(*,16)	8.0	29.7	35	227.6
post-split (*,16), area optimised	A7	213	144556	-47.2	17.6	34.4	39.8	8.2	1x(+/-,32) 1x(*,8)	22.9	272.6	119	192.3
pre-split (+,32) & (-,32), area optimised	A8	2	270572	-1.2	84.5	9.6	5.2	0.6	1x(+/-,16) 1x(*,32)	6.5	5.8	17	382.4
pre-split (+,16) & (-,16), area optimised	A9	5	275937	0.8	82.2	9.5	7.3	0.9	1x(+/-,8) 1x(*,16)	9.5	54.8	26	365.7
delay optimised	D1	1	382861	0.0	90.7	5.9	3.0	0.3	1x(+,16) 1x(+,32) 1x(-,32) 2x(*,16) 1x(*,32)	3.4	0.0	13	258.3
pre-split (*,32), delay optimised	D2	2	191111	-50.1	70.2	16.8	12.1	0.9	1x(+,16) 1x(+,32) 1x(-,32) 2x(*,16)	3.2	-4.7	18	177.9
pre-split (*,32), delay optimised, delay optimised	D3	2	191211	-50.1	70.1	16.8	12.1	0.9	1x(+,16) 1x(+,32) 1x(-,32) 2x(*,16)	3.0	-10.1	18	167.8
pre-split (*,16), delay optimised, delay optimised, delay optimised	D4	26	111521	-70.9	41.4	29.0	25.3	4.3	1x(+,32) 1x(-,32) 1x(+/-,16) 1x(*,8)	5.5	64.9	48	115.4
pre-split (*,32), delay optimised, delay annealed, delay optimised	D5	8	190043	-50.4	69.3	17.8	11.8	1.0	1x(+,32) 1x(-,32) 1x(+/-,32) 2x(*,16)	2.7	-18.8	19	143.5
pre-split (*,16), various annealing and heuristic optimisations	D6	-	172028	-55.1	47.9	25.2	24.9	2.0	3x(+,16) 1x(+,32) 1x(-,16) 1x(-,32) 3x(*,8)	3.6	7.6	34	106.2
pre-split (*,32), (+,32) & (-,32), delay optimised	D7	2	182861	-52.2	67.1	17.8	13.8	1.4	1x(+,16) 1x(+/-,16) 2x(*,16)	3.6	6.2	25	142.6
post-split (*,32), delay optimised	D8	2	177139	-53.7	72.4	16.4	10.3	0.9	1x(+,16) 1x(+,32) 1x(-,32) 1x(*,16)	3.7	11.1	16	233.3
post-split (*,32), delay optimised, delay annealed (100,0,-50,200)	D9	13	244123	-36.2	74.3	16.5	8.4	0.8	1x(+,32) 1x(-,16) 1x(-,32) 3x(*,16)	2.7	-19.4	19	142.4
post-split (*,16), delay optimised, delay optimised, delay optimised	D10	26	112086	-70.7	38.6	28.8	28.2	4.4	1x(+,16) 1x(+,32) 1x(-,32) 1x(*,8)	5.3	56.6	49	107.3
delay optimised, multicycled (70 ns)	DM	1	384461	0.4	90.3	5.9	3.0	0.7	1x(+,16) 1x(+,32) 1x(-,32) 2x(*,16) 1x(*,32)	2.0	-41.6	28	70.0

Table 5.3 DHRC2 area and delay figures for various optimisation configurations

Optimisation configuration	Ref	CPU (secs)	Area (μm^2)	Δ area cf. A1,D1 (%)	Area Breakdown (%)				Module Usage	Delay (µs)	Δ delay cf. A1,D1 (%)	CPL (states)	Clock (ns)
					Func	Store	Inter	Cont					
unoptimised	N	-	1077470	-	90.4	7.6	1.2	0.8	7x(+,32) 6x(-,32) 4x(*,32)	11.8	-	46	257.3
area optimised	A1	4	320041	0.0	73.8	17.6	7.7	0.9	1x(+/-,32) 1x(*,32)	5.5	0.0	16	342.9
pre-split (*,32), area optimised	A2	13	178638	-44.2	39.4	38.2	20.0	2.4	1x(+/-,32) 1x(*,16)	6.8	23.1	29	232.9
pre-split (*,16), area optimised	A3	353	155396	-51.4	18.6	44.0	30.6	6.8	1x(+/-,32) 1x(*,8)	18.0	228.1	93	193.6
post-split (*,32), area optimised	A4	13	172528	-46.1	40.8	36.3	20.5	2.5	1x(+/-,32) 1x(*,16)	6.8	24.4	30	227.5
post-split (*,16), area optimised	A5	233	149396	-53.3	19.3	41.9	31.4	7.4	1x(+/-,32) 1x(*,8)	18.8	242.2	97	193.6
post-split (*,32), area optimised, split (*,16), area optimised	A6	199	159986	-50.0	18.0	41.3	34.1	6.6	1x(+/-,32) 1x(*,8)	18.0	228.1	93	193.6
delay optimised	D1	4	553814	0.0	83.5	10.7	5.4	0.5	1x(+,32) 1x(+/-,32) 2x(*,32)	3.3	0.0	13	257.3
pre-split (*,32), delay optimised	D2	9	254523	-54.0	53.8	26.0	18.6	1.5	2x(+/-,32) 2x(*,16)	4.5	34.1	27	166.1
pre-split (*,16), delay optimised, delay optimised	D3	290	176061	-68.2	25.6	37.0	31.5	5.9	1x(+,32) 2x(+/-,32) 1x(*,8)	15.1	352.0	91	166.1
post-split (*,32), delay optimised	D4	6	246528	-55.5	52.9	28.1	17.5	1.6	1x(+,32) 1x(+/-,32) 2x(*,16)	4.5	35.2	27	167.4
post-split (*,16), delay optimised	D5	230	166396	-70.0	20.2	39.2	34.3	6.2	1x(+,32) 1x(+/-,32) 1x(*,8)	15.2	355.5	91	167.4
post-split (*,32), delay optimised, various annealing optimisations	D6	-	335648	-39.4	59.1	24.1	15.8	1.0	2x(+,32) 2x(+/-,32) 3x(*,16)	3.1	-6.9	22	141.5
post-split (*,16), various annealing optimisations	D7	-	346719	-37.4	32.8	30.7	34.5	2.0	1x(+,16) 3x(+,32) 1x(+/-,32) 5x(*,8)	7.7	129.5	52	147.6
delay optimised, multicycled (50 ns)	DM	4	558514	0.8	82.8	10.8	5.3	1.1	(1x(+,32) 1x(+/-,32) 2x(*,32)	2.1	-37.2	42	50.0

Table 5.4 FFT2 area and delay figures for various optimisation configurations

Optimisation configuration	Ref	CPU (secs)	Area (μm^2)	Δ area cf. A1,D1 (%)	Area Breakdown (%)				Module Usage	Delay (µs)	Δ delay cf. A1,D1 (%)	CPL (states)	Clock (ns)
					Func	Store	Inter	Cont					
unoptimised	N	-	1410474	-	95.7	3.9	0.2	0.2	2x(+,32) 2x(-,32) 6x(*,32)	5.7	-	22	257.3
area optimised	A1	1	285581	0.0	81.5	14.6	3.5	0.4	1x(+/-,32) 1x(*,32)	3.1	0.0	9	342.9
pre-split (*,32), area optimised	A2	13	167973	-41.2	39.8	36.3	22.0	1.9	1x(+/-,32) 1x(*,16)	7.1	128.7	31	227.7
post-split (*,32), area optimised	A3	11	154253	-46.0	43.3	33.4	21.2	2.1	1x(+/-,32) 1x(*,16)	7.3	136.0	32	227.7
pre-split (*,16), area optimised	A4	440	172951	-39.4	14.6	37.2	40.7	7.4	1x(+/-,32) 1x(*,8)	20.8	574.0	127	163.8
post-split (*,16), area optimised	A5	898	150246	-47.4	16.8	36.5	37.8	8.9	1x(+/-,32) 1x(*,8)	21.6	600.6	132	163.8
post-split (*,32), area optimised, split (*,16), area optimised	A6	416	160661	-43.7	15.8	34.3	41.9	8.0	1x(+/-,32) 1x(*,8)	21.0	579.4	128	163.8
post-split (*,32), (+,32) & (-,32), area optimised	A7	28	182359	-36.1	34.4	32.5	30.0	3.1	1x(+/-,16) 1x(*,16)	11.0	257.8	55	200.7
post-split (+,32) & (-,32), area optimised	A8	1	288077	0.9	79.4	14.7	5.4	0.5	1x(+/-,16) 1x(*,32)	3.8	22.8	12	315.8
pre-split (+,32) & (-,32), area optimised	A9	2	288757	1.1	79.2	14.7	5.6	0.5	1x(+/-,16) 1x(*,32)	4.3	37.7	14	303.6
post-split (18,32_32), area optimised	A10	5	125973	-55.9	53.0	30.7	14.1	2.1	1x(+/-,32) 1x(*,16)	5.9	91.6	26	227.4
post-split (18,16_16), area optimised	A11	117	102291	-64.2	24.7	39.5	26.9	8.9	1x(+/-,32) 1x(*,8)	14.6	374.0	90	162.5
delay optimised	D1	1	963022	0.0	94.2	4.7	1.1	0.1	1x(-,32) 1x(+/-,32) 4x(*,32)	1.5	0.0	5	303.6
pre-split (*,32), delay optimised	D2	1	255965	-73.4	54.4	24.0	20.4	1.2	3x(+,32) 1x(-,32) 2x(*,16)	4.1	170.6	29	141.7
post-split (*,32), delay optimised	D3	5	253368	-73.7	51.3	26.8	20.7	1.2	2x(+/-,32) 2x(*,16)	4.3	180.0	30	141.7
pre-split (*,16), delay optimised	D4	675	211744	-78.0	21.5	29.8	42.8	5.9	2x(+,32) 1x(-,32) 2x(*,8)	9.5	523.7	124	76.4
post-split (*,16), delay optimised	D5	370	232556	-75.9	24.4	28.6	41.6	5.5	2x(+/-,32) 2x(*,8)	11.1	633.6	127	87.7
post-split (*,16), delay optimised, delay optimised	D6	582	242037	-74.9	27.4	27.4	39.9	5.2	2x(+/-,32) 2x(*,8)	9.7	535.8	126	76.6
post-split (*,32), delay optimised, split (*,16), delay optimised, delay optimised	D7	175	243662	-74.7	28.2	28.6	38.8	4.5	1x(+,16) 2x(+/-,32) 2x(*,8)	8.2	442.9	108	76.3
post-split (*,32), (+,32) & (-,32), delay optimised, split (*,16), delay optimised	D8	171	224871	-76.6	17.4	32.8	43.9	5.8	2x(+/-,16) 2x(*,8)	9.9	553.6	130	76.3
post-split (+,32) & (-,32), delay annealed (100,0,-50,200)	D9	12	740656	-23.1	92.2	5.8	1.8	0.2	2x(+,16) 3x(-,16) 1x(+/-,16) 3x(*,32)	2.8	86.4	11	257.3

Table 5.5 Diffeq area and delay figures for various optimisation configurations

Optimisation configuration	Ref	CPU (secs)	Area (μm^2)	Δ area cf. A1,D1 (%)	Area Breakdown (%)				Module Usage	Delay (µs)	Δ delay cf. A1,D1 (%)	CPL (states)	Clock (ns)
					Func	Store	Inter	Cont					
unoptimised	N	-	571178	-	88.5	10.7	0.0	0.8	26x(+,16) 8x(*,16)	6.6	-	47	141.4
area optimised	A1	9	91773	0.0	63.9	24.5	8.6	2.9	1x(+,16) 1x(*,16)	8.6	0.0	27	319.0
pre-split (18,16), area optimised	A2	68	82621	-10.0	20.8	42.7	29.3	7.1	1x(+,16) 1x(*,8)	6.7	-21.7	59	114.3
post-split (18,16), area optimised	A3	69	68061	-25.8	25.2	42.5	23.6	8.7	1x(+,16) 1x(*,8)	6.8	-21.6	59	114.4
post-split (18,8), area optimised	A4	1868	80953	-11.8	8.4	37.7	30.7	23.2	1x(+,16) 1x(*,4)	14.3	66.2	188	76.1
post-split (18,16), target area optimised (90000)	A5	30	72646	-20.8	30.6	42.0	18.8	8.5	1x(+,16) 1x(*,8)	4.7	-45.4	62	75.8
post-split (18,16), delay optimised	A6	18	92740	1.1	38.3	34.6	22.1	5.0	2x(+,16) 2x(*,8)	4.7	-45.9	46	101.2
post-split (18,16), area optimised, split (18,8), area optimised	A7	1992	102493	11.7	6.6	36.0	39.7	17.7	1x(+,16) 1x(*,4)	13.8	60.1	181	76.2
pre-split (18,8), area optimised	A8	3850	92453	0.7	7.4	36.5	36.0	20.2	1x(+,16) 1x(*,4)	14.2	65.0	187	76.0
area optimised, multicycled (20 ns)	AM	9	115673	26.0	50.7	19.5	6.8	23.0	1x(+,16) 1x(*,16)	5.3	-38.2	266	20.0
delay optimised	D1	3	155648	0.0	76.3	14.5	8.2	1.1	3x(+,16) 2x(*,16)	2.4	0.0	17	141.5
pre-split (18,16), delay optimised	D2	16	110064	-29.3	34.7	32.1	29.3	3.9	4x(+,16) 2x(*,8)	3.2	35.0	43	75.5
post-split (18,16), delay optimised	D3	12	107964	-30.6	35.4	34.2	26.6	3.9	4x(+,16) 2x(*,8)	3.2	31.8	42	75.5
post-split (18,8), delay optimised	D4	527	110148	-29.2	15.8	32.8	36.0	15.4	4x(+,16) 2x(*,8)	7.1	193.6	170	41.6
post-split (18,16), area optimised	D5	57	88621	-43.1	24.8	39.8	30.1	5.3	3x(+,16) 1x(*,8)	5.4	123.1	47	114.2
post-split (18,16), delay optimised, split (18,8), delay optimised	D6	266	117028	-24.8	15.9	32.2	39.4	12.5	1x(+,8) 4x(+,16) 2x(*,4)	6.0	148.6	146	41.0
pre-split (18,8), delay optimised	D7	909	128344	-17.5	15.3	29.4	41.9	13.5	4x(+,16) 2x(*,4)	7.0	189.0	173	40.2
delay optimised, multicycled (70 ns)	DM	4	157748	1.3	75.3	14.3	8.1	2.4	3x(+,16) 2x(*,16)	1.5	-36.8	38	40.0

Table 5.6 Ellip area and delay figures for various optimisation configurations

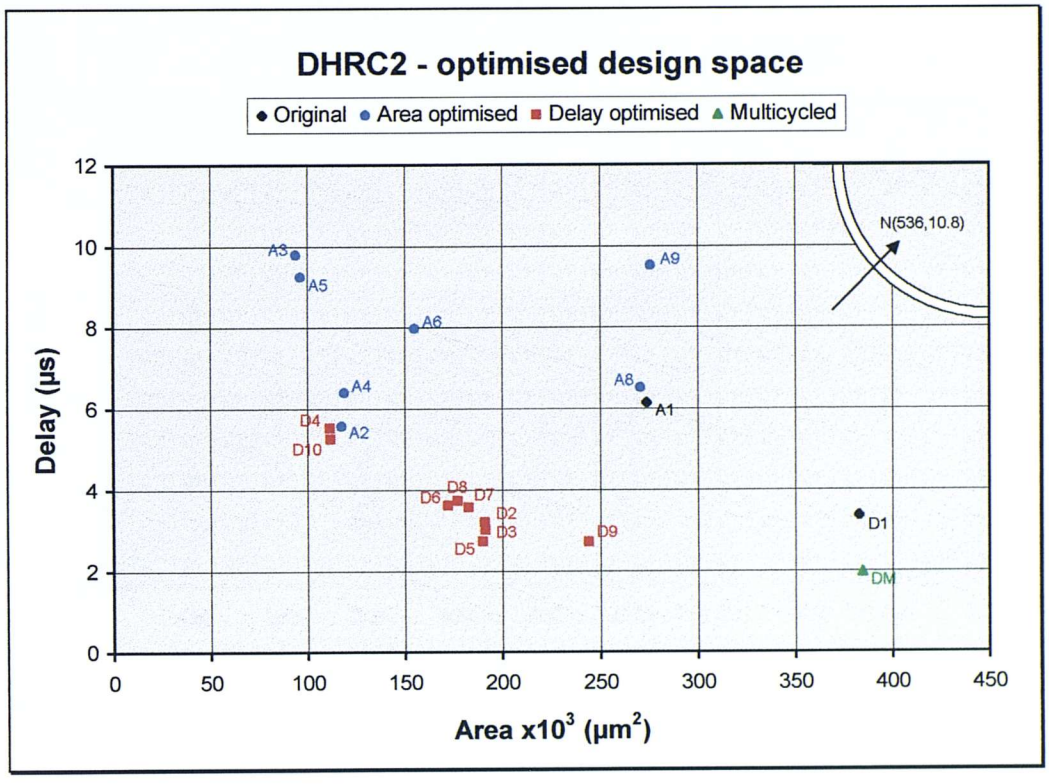


Figure 5.21 DHRC2 expanded design space

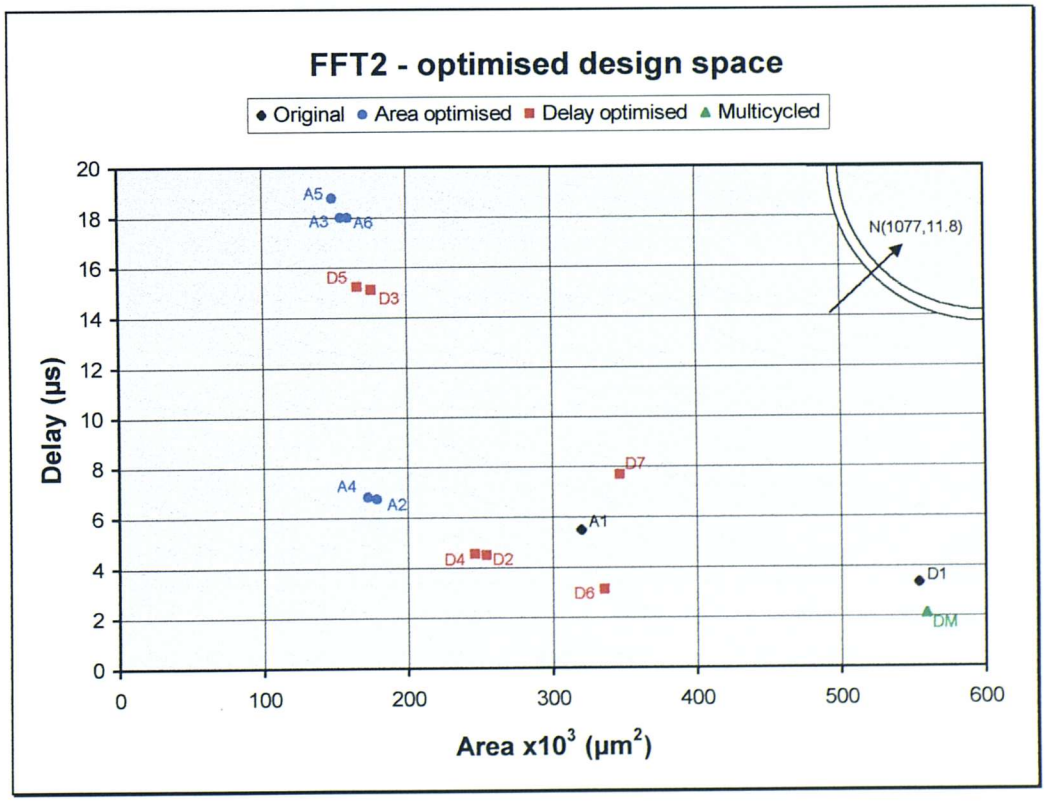


Figure 5.22 FFT2 expanded design space

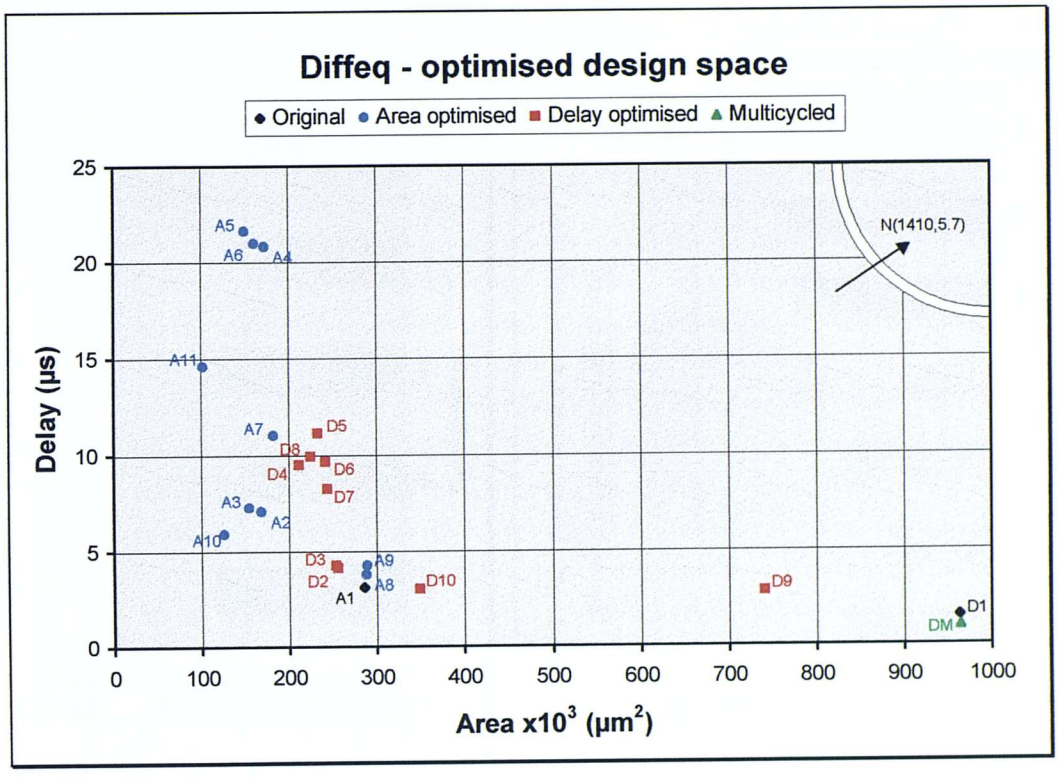


Figure 5.23 Diffeq expanded design space

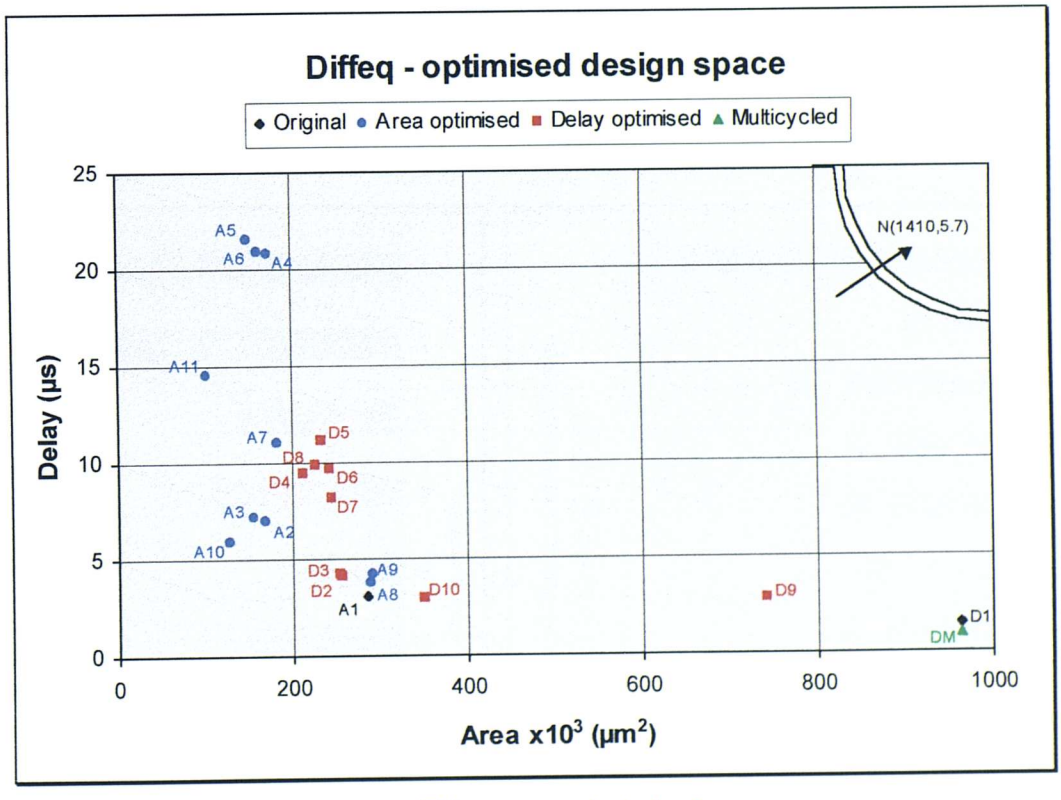
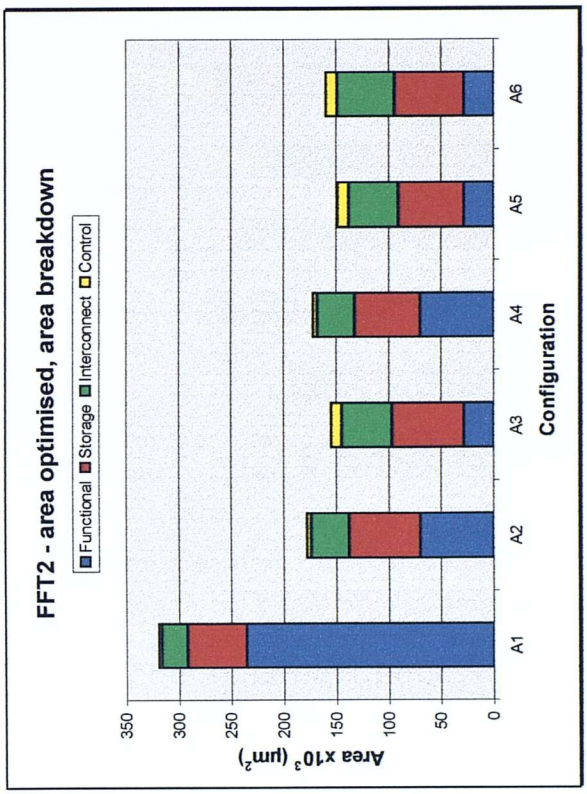


Figure 5.24 Ellip expanded design space



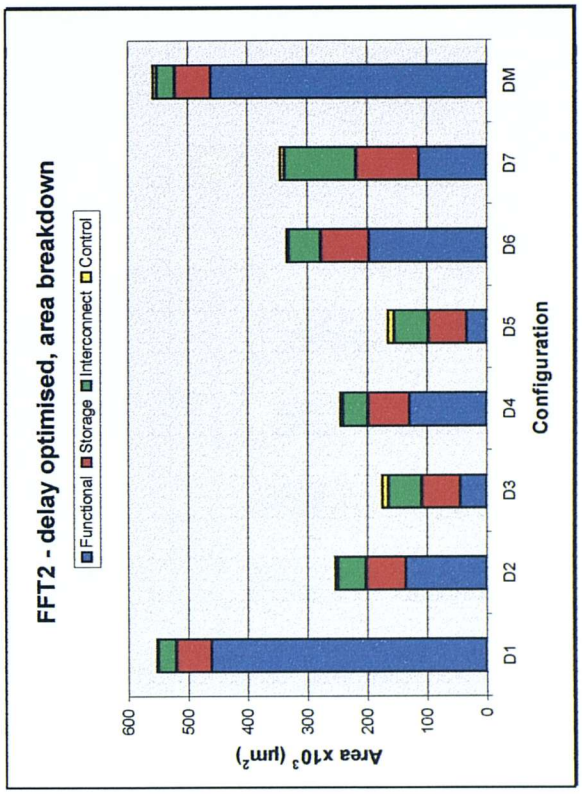
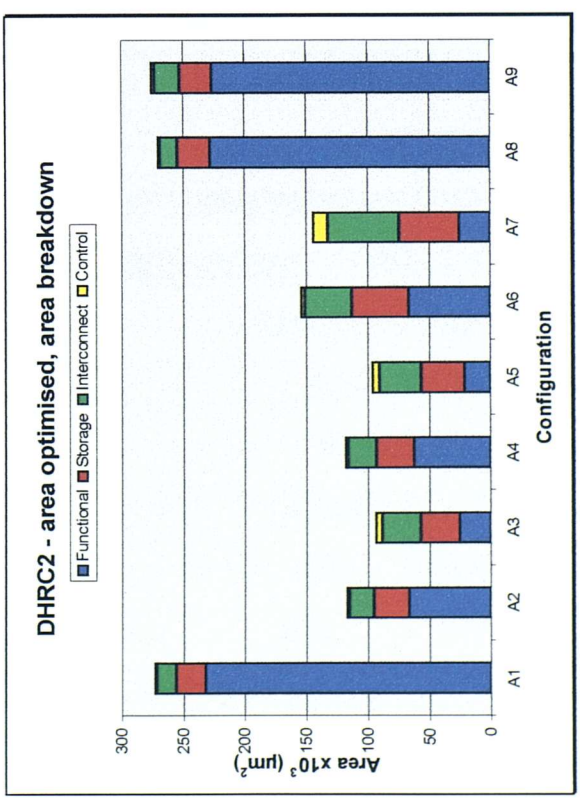


Figure 5.26 Optimised FFT2 area breakdown



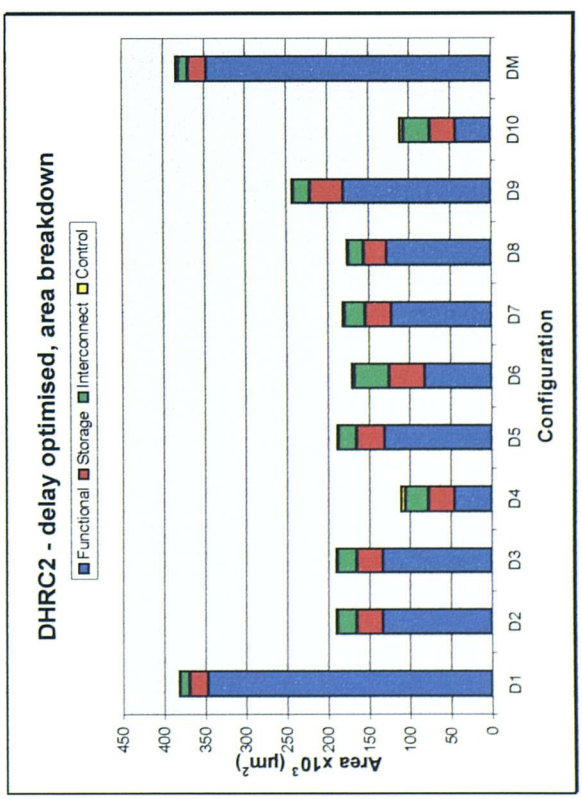
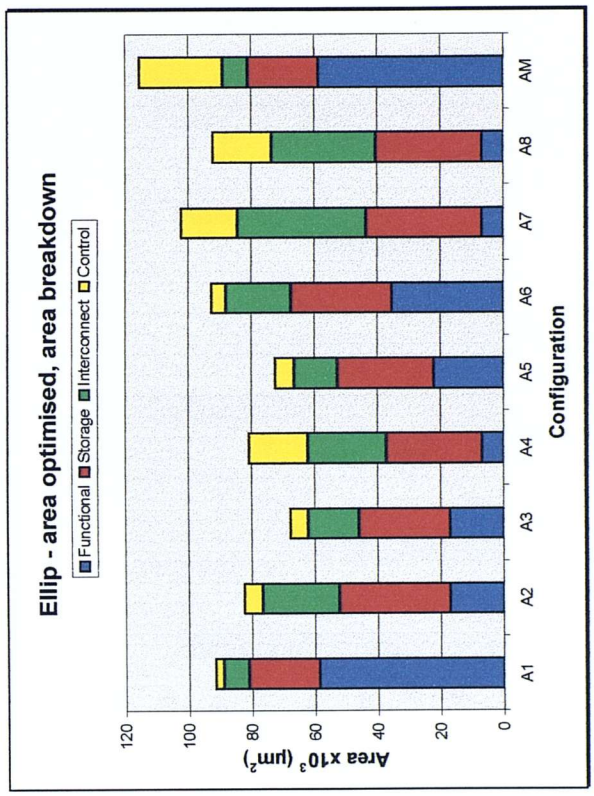


Figure 5.25 Optimised DHRC2 area breakdown



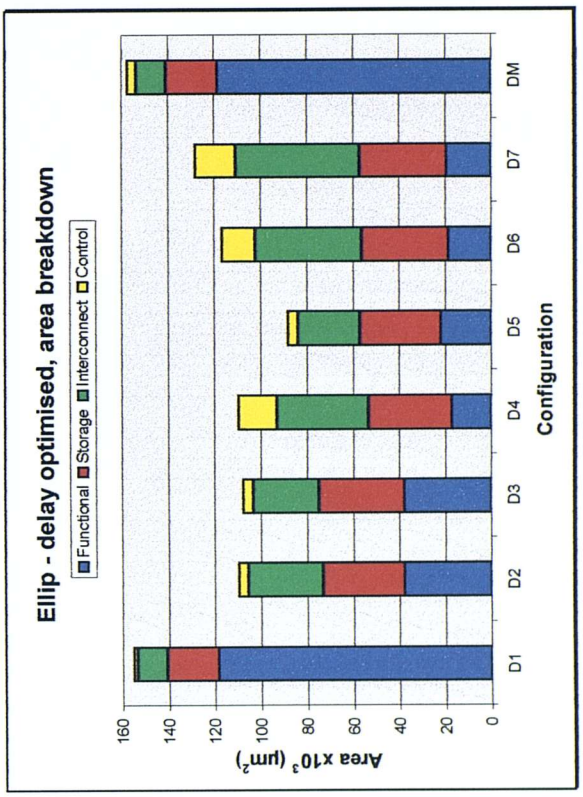
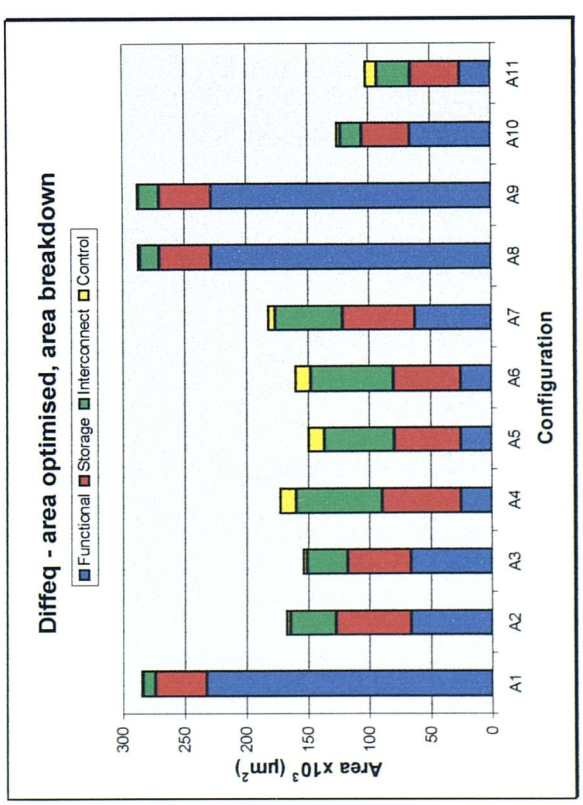


Figure 5.28 Optimised ellip area breakdown



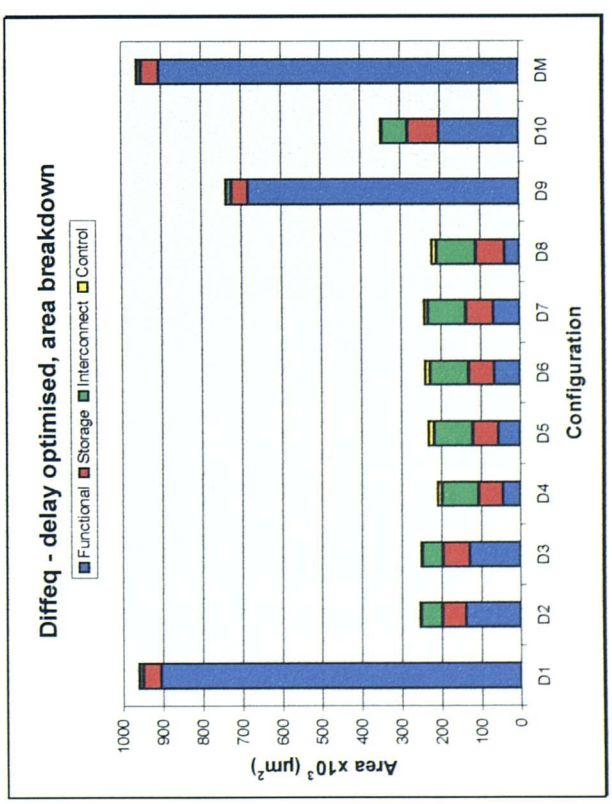
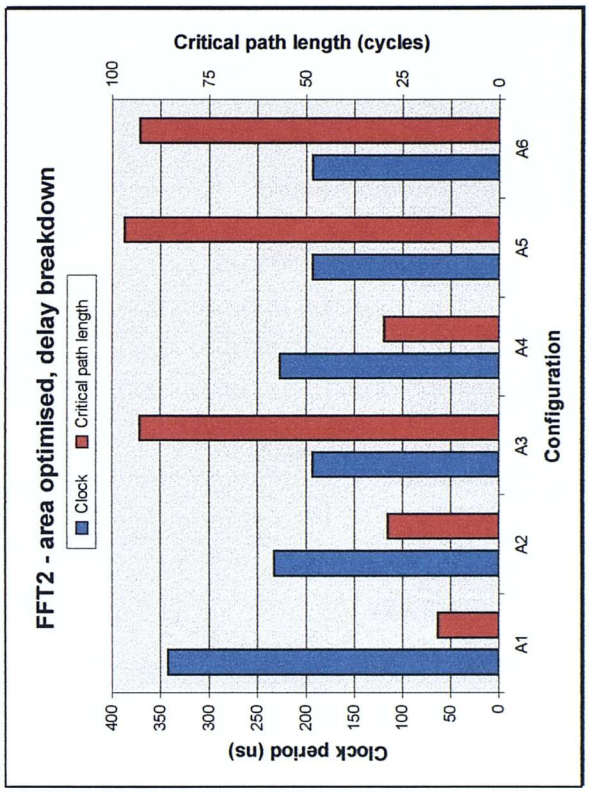


Figure 5.27 Optimised diffeq area breakdown



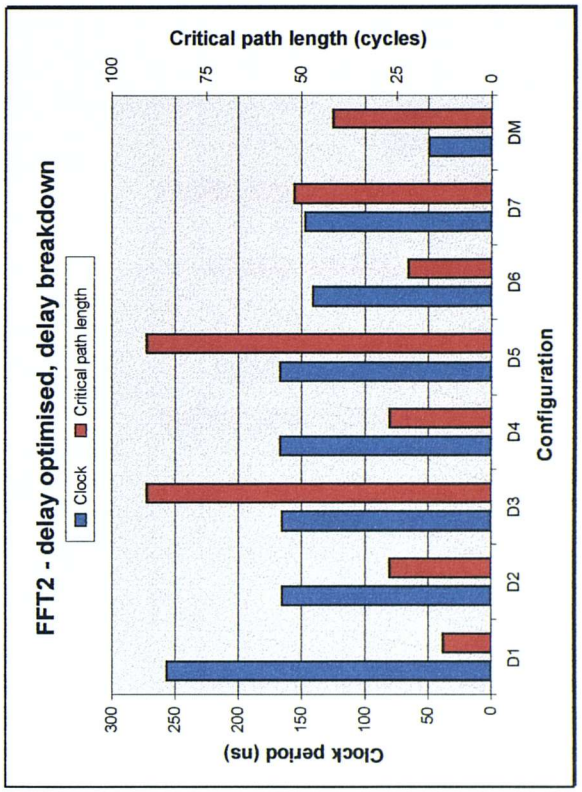
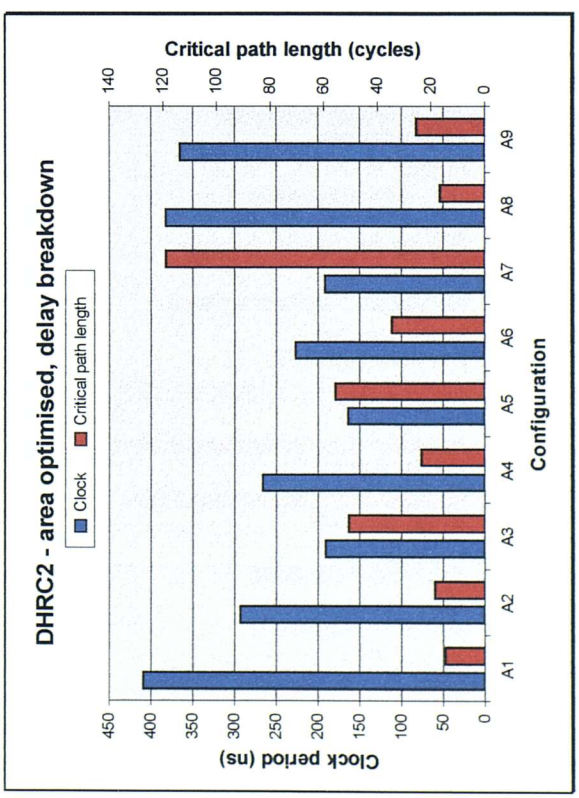


Figure 5.30 Optimised FFT2 delay breakdown



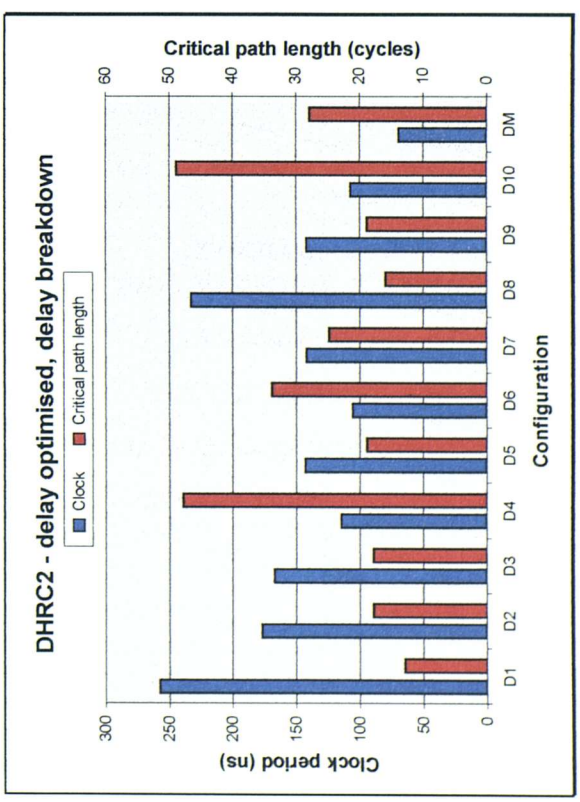
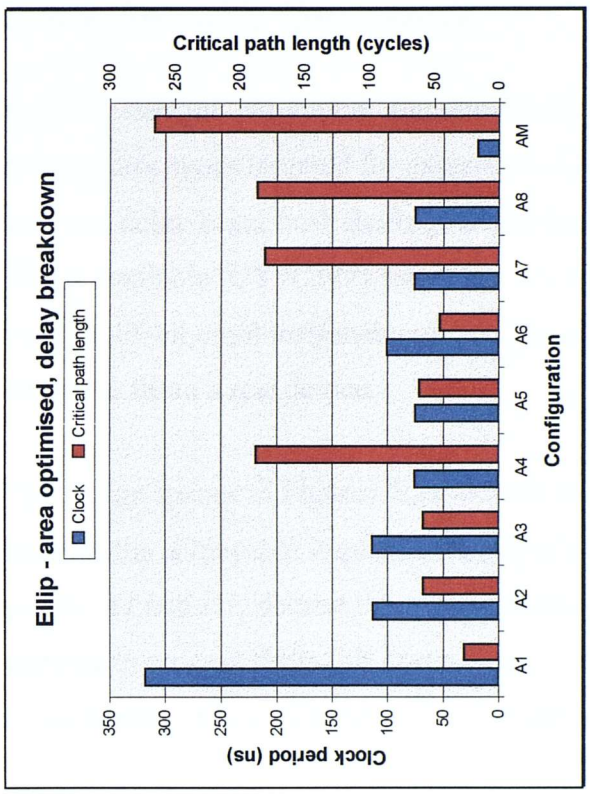


Figure 5.29 Optimised DHRC2 delay breakdown



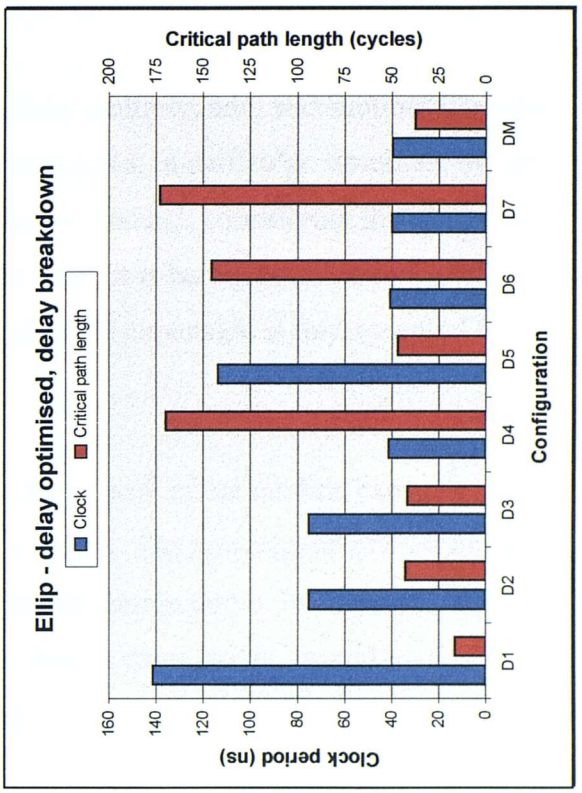
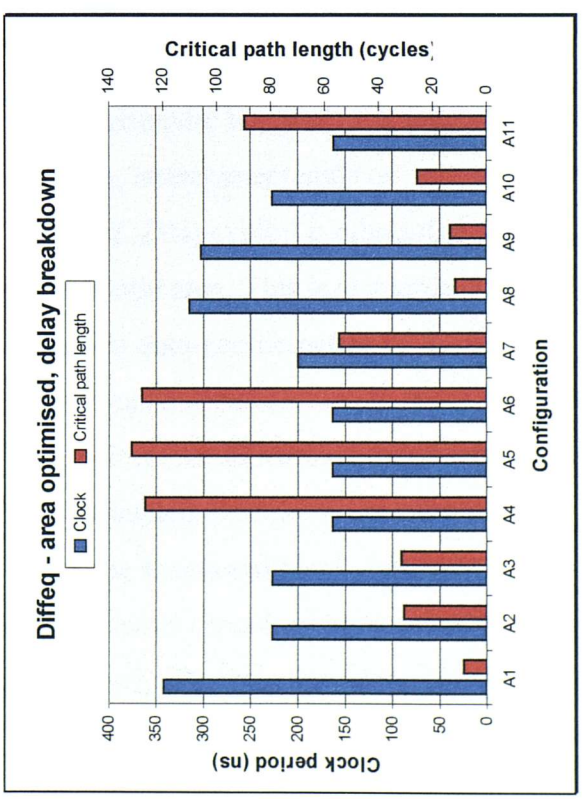


Figure 5.32 Optimised ellip delay breakdown



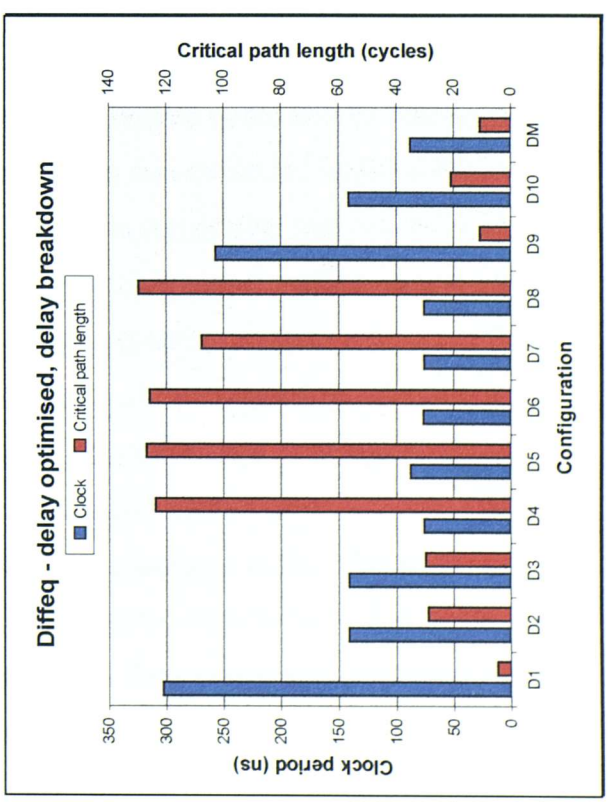


Figure 5.31 Optimised diffeq delay breakdown

5.4.1 Results Overview

In all the designs, the largest functional units are the multiply, add, and subtract modules, are the three types targeted for expansion. The multiplies, in particular, dominate the total area and delay being both the largest and the slowest. Indeed, considering that the largest FPGA available (CY7C387) can hold only 76800μm^2, it is barely possible to include even a single 16-bit combinational unit (55278μm^2); module expansion is thus essential for any design to fit on a real device.

The design spaces in Figures 5.21 to 5.24 show the overall effect module expansion can have on the achievable implementations of each design. The non-expanded area and delay points, *A1* and *D1*, delimit the non-expanded optimal design curve. In each case, the expansion process shifts this curve closer to the design space origin, providing a considerably enhanced range of implementations.

The most obvious characteristic of all the figures, is the dramatic improvement in area of both the area and delay optimised groups, generally, but not always, at the cost of extra delay. In all cases, the non-expanded minimum area figure is reduced from anywhere between 25%, in the case of **ellip** *A3*, to 65% for **diffeq** *A11*. The mechanics of this process can be explained with the aid of the area breakdown graphs (Figures 5.25 to 5.28) which compare the total area for each configuration, broken down into its functional, storage, interconnect and control components. All the non-expanded implementations (*A1, AM, D1, DM*) exhibit a substantial functional unit bias comprising between 65% and 95% of the total area. This is primarily due to the large combinational multipliers and the inherent data-centric nature of the designs. Module expansion reduces the size of the functional component due to the reduced complexity of the constituent sub-units, in return for an increase in storage, interconnect and control requirements, forming the skeleton of the expanded module. As expansion depth increases (ie. smaller bit-width thresholds), so does the functional block, however at some point, the increase in the other three components cancels out any decreases, thus rendering the expansion useless. **Ellip** *A3* and *A4* clearly illustrate this point: the first configuration shows post-split expansion of all 16 bit multipliers achieving a 25.8% decrease in total area, from a 71% decrease in functional area, and a 28%, 100% and 118% increase in storage, interconnect and control area respectively, compared to the non-expanded *A1*. Expanding a further level of hierarchy

with post-split optimisation of all 16 and 8-bit multipliers, results in a less dramatic 11.8% total decrease, this time comprising an 88% drop in functional area, but a 36%, 213% and 596% increase in the other three.

Another, less obvious, effect of expansion is an improvement in the overall clock utilisation. Recall from section 5.1 that the longest functional unit propagation delay determines the minimum clock period, leading to the waste of a considerable amount of excess time within many control states. Multicycling attacks this inefficiency by allowing combinational modules to evaluate over several cycles, thus reducing the clock period and improving the overall delay figure. This is illustrated in the delay breakdown graphs (Figures 5.29 to 5.32) which show how the clock period and critical path length are affected by each optimisation. Comparing the original implementations (*A1* and *D1*) with their multicycled equivalents (*AM* and *DM*), it can be seen that the decrease in clock period is accompanied by an increase in the critical path length; thus multicycling serves to balance out these two components, achieving a more even distribution of functional unit execution time across the available control states. The greater number of control states also affects the area, showing up in the breakdown graphs as an increase in the control size. This is not accompanied by any change in the other area components as multicycling simply modifies the control graph.

Examination of the delay breakdown for the expanded configurations exhibits a similar pattern. Here, the shorter clock period is the result of breaking down the longest functional units into simpler (quicker) sub-modules, where the elongated critical path is due to the additional states comprising the sub-structure of the expanded modules. Unlike multicycling, these changes are generally accompanied by a decrease in the total area, thus it is possible to achieve both a delay and area improvement, such as in **DHRC2** *D2, D3, D5* and *D9*. However, expansion should not be viewed as a mechanism solely for optimising delay as multicycling provides more overall control, and does in general produce better results if the minimum delay is desired at any cost. It does however, provide a number of intermediate implementations balancing improvement in both delay and area, eg. **ellip** *D2* and *D3* in Figure 5.24.

An alternative view of clock utilisation is illustrated in Figure 5.33 which shows the *clock utilisation distribution* for a handful of **ellip** configurations. Here, the percentage of time utilised within each control state is determined and separated into 5% bands; the number of states (as a percentage of the total) falling within each band is then plotted for several optimisation configurations. For optimum clock utilisation, the majority of states should occupy the top 95-100% range, however, as can be seen in Figure 5.33, the original non-expanded configuration (*A1*) falls far short of this target, with over 65% of states having a utilisation of around 60% (ie. wasting 40% of the clock period). The best multicycled implementation (*AM*) is a far more efficient scenario with 90% of states almost 100% utilised. In between these two extremes, appear the expanded implementations demonstrating a varying degree of improvement. Note that Figure 5.33 only shows the utilisation efficiency; better utilisation does not necessarily mean a lower total delay as both the clock period and critical path length may vary, but does indicate how much time is wasted in a given schedule. It is interesting to note that in every configuration there exists at least one state with a very low utilisation. These represent VHDL *wait* statements forming points at which a process is idle, thus no substantial operations will be scheduled resulting in a low utilisation figure.

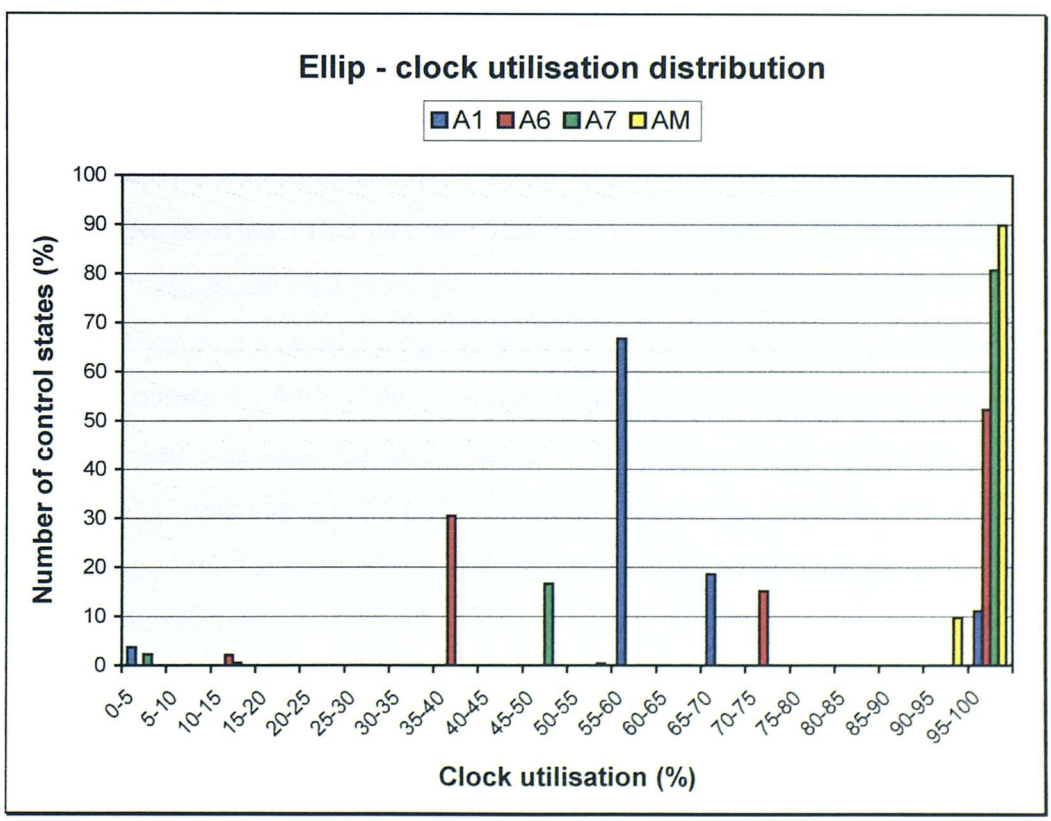


Figure 5.33 Various ellip clock utilisation distributions

The subject of clock and module utilisation is discussed further in the two papers [108, 109] reproduced in Appendix A. These detail work, not reported elsewhere in this thesis, to investigate the possibilities for exploiting the time during which a module is inactive to perform on-line testing.

5.4.2 Further Analysis

While it can be said that in general, module expansion results in a reduction in area, there are a number of factors concerning both the composition of the behavioural description (type and size of operations), and the method of optimisation and expansion (what to expand, when, and by how much), which must be taken into account when synthesising a design. The numerous configurations listed in Tables 5.3 to 5.6 illustrate how many different optimisation options are available, and the dramatic effect they can have on the final outcome.

The first decision to make is which combinational modules should be expanded. The complexity of most designs means that manually expanding individual data path units is impractical. In many cases however, especially when optimising for area, optimisation achieves maximal sharing of the largest units, resulting in a single instance for each operation type, confirmed by the module usage figures for all the *A1* configurations. In most optimised designs there are one or two modules which dominate the implementation area. The four benchmark examples all contain at least one multiplier which is significantly larger than any other unit (see Table 5.2). This must be the first candidate for expansion if any substantial area gains are to be realised. In fact, the multiplier is so large that there is little point in expanding any of the lesser units. For example, in **DHRC2** *A1*, the multiplier accounts for 80% of the total area, whereas the add/subtract ALU takes only 3%. Hence, whereas expanding the multiplier just once reduces the area by 57% (*A2*), expanding the ALU only saves 1.2% (*A8*). This is aggravated by inefficiencies in the MOODS hardware architecture which add a degree of overhead through the inclusion of unnecessary multiplexors (see section 5.4.3).

Once a function has been selected, the next step is to decide on a suitable depth of expansion, ie. what *aob* bit-width threshold to specify. As described above, the expansion process reduces the total area by cutting the size of the functional component, clearly demonstrated in the area breakdown graphs. Taking **DHRC2** Figure 5.25 as an example:

expanding all 32-bit multipliers prior to area optimisation (*A2*) results in an area reduction of 57% compared to non-expanded *A1*. This is achieved with a 71% drop in functional area, counteracted by a 23% increase in storage, interconnect and control. The resulting implementation is less dominated by functional modules, thus reducing the possible effectiveness of further expansions. A two-level expansion therefore (*A3*), splitting all 32-bit and 16-bit multipliers, results in a much less dramatic area reduction to 66% of *A1*. The functional component in this implementation is so small that any further expansions (ie. split 8-bit multipliers) would have little effect on the area, and may even result in an increase due to the added overheads. At this point, where the extra cost of storage, interconnect and control outweighs the reduction in functional area, expansion should cease; assuming of course that delay degradation has not already halted the process. The exact bit-width at which this occurs is dependent upon the target design. Comparing the expansion of 32-bit and 16-bit multipliers in **diffeq** (*A2* vs. *A4* in Figure 5.27), the second level of expansion results in the decrease in functional area being negated by a large increase in the other components, thus making the two-level implementation 3% larger than one-level (split only 32-bit multipliers), not to mention 190% slower. There are two main factors affecting this:

1. The expansion of ICODE instructions sharing a single module requires one set of shared functional units for the expanded sub-modules, but a separate set of interconnects (data path nets) and control for every implemented instruction. Thus the larger number of multiply operations in **diffeq** compared to **DHRC2** (12 as opposed to 6) means that while the former design only benefits from one level of multiplier expansion, the latter shows an area reduction for both one and two-level expansions.

2. As the module bit-width decreases, the proportion of the expanded module structure comprised of interconnect, storage and control increases with respect to the functional area, thereby reducing the effectiveness of later expansions. For example, an area optimised 32-bit split multiplier (splits 32-bit into 16-bit multipliers) has a functional unit composition of 79%, whereas for an 8-bit split multiplier, this figure is only 55%.

An additional consequence of increased expansion depth is a sometimes substantial rise in the processing time required by the optimisation algorithms. This results from the rapid increase in complexity as more sub-modules are expanded, and is often an indication of

proceeding a step too far. For example in **diffeq**, moving from expanding 32 bits in *A3* to 16 bits in *A5* increases the CPU time from 11 seconds to 15 minutes.

The final factor affecting the optimised implementation is the actual optimisation schedule: when to expand and what optimisation algorithm to use. The results in Tables 5.3 to 5.6 show that the positioning of the first expansion with relation to the optimisation steps can have a substantial impact on the final area and delay figures. Put simply this comes down to whether the best results are obtained from pre-splitting or post-splitting. The answer to this question depends to a large extent on the details of the particular design. All of the examples, except for **DHRC2**, produce better area figures when expansion follows an initial heuristic optimisation stage (ie. post-splitting) with little difference in the total delay. Examples include **FFT** *A2* vs. *A4* and *A3* vs. *A5*, and **ellip** *A2* vs. *A3*. For each pre/post-split pair, the final number of functional units is identical, the differences lying in the optimisation of storage and interconnect units. **DHRC2**, on the other hand, shows completely the opposite behaviour, where the pre-split *A2* configuration is smaller than the post-split equivalent *A6*. In fact, this can be considered as a special case which arises due to its use of both 32 and 16-bit multiply operations: initial area optimisation assigns all the multiplies to a single 32-bit module. When this is post-split, each of the original 16-bit operations is expanded as a 32-bit multiply, requiring extra states and enlarged 64-bit output registers (as described in section 5.3.4). When pre-splitting however, all the original 16-bit operations are separate, thus only the single 32-bit multiplier will be expanded resulting in a smaller area and a shorter critical path length. Manually removing the 32-bit operation from the shared multiplier prior to post-splitting solves this problem, producing an identical result to *A2*. In all the other examples, a single bit-width is required for all operations of the same type (see Table 5.1), thus they do not exhibit this inefficiency.

The general difference between the pre- and post-splitting results can be attributed to the increased complexity of the overall design structure as a result of module expansion. This makes it more difficult for MOODS to optimise interconnects and register allocation due to the increased number of combinations possible. The two-stage approach allows MOODS to first optimise at a slightly higher (and thus simpler) level, and then refine this following module expansion. This is borne out by tests on simpler systems in which the differences between the two approaches become insignificant. Note that post-splitting in

several stages does not exhibit the same improvements over performing all module expansion in one step. For example, **FFT** *A6* is obtained by first optimising the non-expanded design, then performing a one-level split of all 32-bit multipliers, then another optimisation, followed by a one-level split of all 16-bit multipliers, and finishing off with a further optimisation. The results obtained are actually worse than *A5* where all 32 and 16-bit multipliers are expanded in one step. This behaviour is unsurprising since once the initial higher-level optimisation is complete, subsequent post-split optimisations are mainly concerned with improving the expanded modules themselves.

Taking all of the above points into consideration, a general approach to design optimisation can be developed thus:

1. Perform standard non-expanded optimisation to investigate the design space and determine minimum feasible area and delay figures.

2. Identify the dominant functional unit type and bit-widths.

3. Perform one-level pre- and post-split optimisations to gauge the possible effects of expansion. If a module is shared among instructions of varying bit-width, it may be necessary to unshare some instructions prior to post-splitting.

4. Repeat the expansion process based on the new dominant functional unit, or with more levels of expansion, until the desired area is obtained, the area starts to increase, or the delay exceeds some maximum allowed limit.

These steps should form the basis upon which a more detailed optimisation configuration can be developed, possibly involving the consideration of some individual modules, and the manipulation of cost function target values. For example, in **ellip** *A5*, the original non-expanded area figure is used as an optimisation target following module expansion in an attempt to exploit the reduced area of the expanded implementation in order to obtain a faster schedule. In this case, the resulting circuit is still 20% smaller than the original, though not as small as *A3* where the target area is zero. The extra area is occupied by a carry-lookahead adder module, in place of the original ripple-carry adder, enabling a faster clock rate to be used, thus reducing the delay by 31% compared to *A3*, and 45% compared to *A1*. This is even better (11%) than the best area optimised, multicycled implementation *AM*, with the additional benefit of a 48% reduction in area.

5.4.3 Problems Encountered

Although the results obtained are generally good, providing substantial area reductions, and in some cases delay improvements as well, the expansion process highlights a number of inefficiencies in the MOODS optimisation algorithms and target hardware architecture, the resolution of which would lead to further improvements.

- The inability of the heuristic optimisation to perform hill-climbing transformations in order to avoid local minima means that there are certain circumstances where the simulated annealing approach is significantly superior. These tend to show up in delay optimisations where the instruction scheduling is of utmost importance; area optimisation tends to achieve maximal sharing, limiting the range of possible schedules. Expansion can significantly increase the complexity of a design, particularly its control graph, making the search for an optimum schedule more difficult. For example, **diffeq** configuration *D2* has almost six times as many control states as the non-expanded *D1*, and *D4*, with two levels of expansion, has twenty five times as many. Best results are often obtained using a combination of the two algorithms, however the user can only achieve this by getting a "feel" for the most suitable approach through several attempts on a particular design. Given the relatively speedy operation of MOODS, where the best results tend to be achieved in optimisation runs ranging from several seconds to a few minutes, this is a perfectly feasible approach. For example, **diffeq** *D10* uses a number of heuristic and annealing optimisation runs following module expansion to achieve a delay reduction of 30%, when compared to the same expansion using just the heuristic algorithm (*D2*).

- A related point of interest is that some implementations require two or three sequential passes though the heuristic algorithm (eg. **FFT2** *D3* and **diffeq** *D6*) before the best delay figure is obtained. Recall from Chapter 4 that the final part of the heuristic process is the selection of alternative module bindings; the need for several passes is brought about when the selection of a different module type enables an alternative schedule to be achieved that improves the overall performance. Since this is the last stage in the algorithm, a further optimisation pass is necessary in order to exploit the new configuration, at the end of which another alternative module binding may require further passes. This just serves to illustrate the problems encountered due to the fixed

order of optimisation in heuristic methods where so many different trade-offs are possible.

- Referring back to section 5.3.4, if the output register connected to a module is not wide enough for the expanded module template, it is automatically enlarged during expansion. This results in a substantial register overhead in some situations. For example, **diffeq** uses 32-bit integers for all variables as both inputs and output to multiply operations. Expanding these modules using a basic 32-bit split multiplier with 32-bit inputs, and a 64-bit output, will add an extra 32 bits to each output register. The solution to this problem is to ensure that a suitable 32 bits in/32 bits out expanded module template is available in the library. The expansion algorithm will then automatically select the template with the closest matching I/O widths, leading to improvements in both the total area (due to the removal of the unneeded registers), and delay (due to less cycles in the expanded template). In the case of **diffeq**, all optimisations other than *A10* and *A11* are performed with only the basic set of templates (the 32-bit multiplier requiring 64 output bits). *A10* and *A11*, however, include expanded modules optimised for 32 bits in/32 bits out, and 16 bits in/16 bits out operation. These two implementations compare favourably with their wider module equivalents, *A10* being 20% smaller and faster than *A2*, and *A11* around 30% smaller and faster than *A5*.

- One final inefficiency stems from the use of multiplexors to implement multiple writes to aliased registers (ie. bit slices). The MOODS hardware model defines each register bit as having an independent load enable input so that when writing to parts of a variable (via an alias), only the relevant bits are loaded. However, multiple writes to the same register generally requires a multiplexor to select the appropriate input, the multiplexor being the same width as the register. Aliases into these input multiplexors are padded out to the full register width with leading and trailing zeroes, while the register load signals determine the aliased block. Inefficiencies tend to occur in expanded modules when there is no overlap between aliases within a register. This occurs in the split adder of Figure 5.4 where each of the four inputs to the *sum* register writes to a non-overlapping set of bits. The structure produced by MOODS however, will include a four-way 32-bit multiplexor on the register input, shown in Figure 5.34, even though this is entirely superfluous. In general use, this limitation is rarely significant as registers tend to be highly shared, but in the context of expanded

modules, the wasted area of the multiplexor means that in some cases, expansion becomes uneconomic. For example, area optimisation of the 32-bit split adder in Figure 5.4 results in the configuration in Figure 5.34. Ignoring the control and carry circuitry, this structure will occupy approximately 4080μm^2 based on the figures in Table 5.2. Compared to the 4800μm^2 of a full ripple-carry adder this represents a saving of around 15%. Removal of the output multiplexor reduces this area to 2160μm^2 ,a reduction of 47%, and an improvement over a full 32-bit ripple-carry adder of 55%.

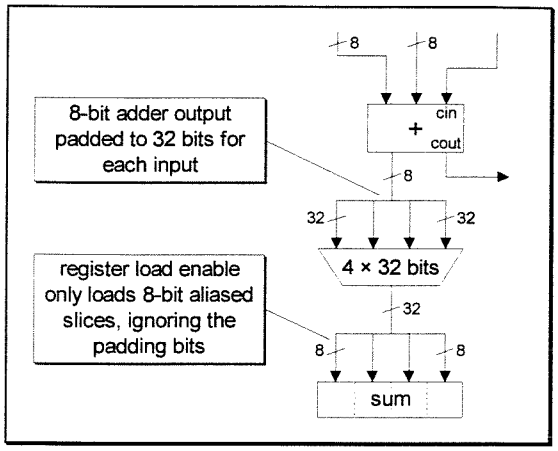


Figure 5.34 Example of inefficient MUX usage

Hence the poor area improvements achieved in the area optimised expansion of adders and subtractors, eg. **diffeq** *A8* and *A9*. Notice however, that where carry-lookahead modules are used, eg. **diffeq** *D9*, the block adder can make a difference, even with the inefficient multiplexing, due to the greater size of the adder units. The solution to this problem is to use a more sophisticated interconnection model together with multiplexor input optimisation for extracting the minimum required bit-width. The enhanced implementation of register bit-slicing is a central requirement for the next version of the MOODS system, currently under development (see Chapter 8).

5.5 Summary

The development of a module expansion capability within the existing MOODS framework enables further optimisation of a behavioural design, primarily as a mechanism for decreasing the area occupied by the final implementation. It enables MOODS to utilise sequential clocked modules in place of the original purely combinational units, allowing much more substantial designs to be developed within the constraints imposed by the limited resources of FPGA devices. Expanding the hierarchical structure of a module

within the top-level control and data paths, allows MOODS to perform inter-module optimisations further reducing the area requirements. The expansion process also tends to reduce the complexity (and hence delay) of the slowest functional modules, thus improving the clock utilisation through a lower minimum clock period.

Results from a number of benchmark designs using a basic set of expanded module templates yield substantial reductions to the original minimum area of between 25% and 65%. In some cases, delay improvements can also be achieved in addition to a reduction in area, due to the shorter cycle time, and the use of faster (and larger) sub-modules which soak up some of the reduced area.

The range of expanded templates employed to date is fairly general, however, as the system is used, the development of more specific enhanced modules, such as the 32 bits in/32 bits out multipliers used in **diffeq** *A10* and *A11*, will further improve the optimisation capabilities of the system. Specialised templates, such as a fixed-point multiplier described in Chapter 7 and the floating-point library currently under development [110], will also enable the creation of a technology-independent high-level function library providing an enhanced behavioural design environment.

The increased complexity of expanded designs does, however, highlight a number of shortcoming in MOODS itself. The provision of improved optimisation algorithms and a more flexible hardware architecture are high priorities in the development of a second-generation synthesis system, discussed in more depth in Chapter 8.

Chapter 6

Enhancements to the VHDL Compiler

The VHDL to ICODE compiler, VHDL2IC, forms the front end to the MOODS system, compiling a subset of VHDL'87 [6] into ICODE, suitable for direct translation into an initial design representation (see Chapter 4). This chapter describes a number of improvements made to the compiler to enhance the usability of the system, and provide the user with greater power and flexibility for the development of complex designs. These include expansion of the supported VHDL subset and extra facilities integrating *macro operators* and *macro ports* with a high-level function library, directly accessible from the VHDL source.

This text is subdivided into three main sections: section 6.1 describes the implementation of a substantially enlarged VHDL subset when compared to the original version of VHDL2IC, covering features similar to those described in Chapter 3; section 6.2 examines the effects of these additions on the size and performance of the optimised designs obtained, and discusses a number of restrictions to the VHDL simulation model peculiar to behavioural synthesis; and finally section 6.3 details the development and use of expanded *macro operators* and *macro ports* within the VHDL design environment, and their optimisation and operation in MOODS itself.

6.1 Enhanced Implementation of VHDL

The original VHDL2IC compiler (version 1.1) supports a limited subset of the VHDL 1987 standard language [6], imposing a number of restrictions on the design style adopted by the user [111]. To enable the development of more complex systems, and allow the synthesis of the benchmark behavioural designs [106, 107], it was necessary to significantly enhance the range of VHDL constructs implemented. Based on the general features used in the

benchmarks, and typical requirements of complex VHDL designs, the following significant (and not so significant) omissions in VHDL2IC v1.1 can be identified:

- The main VHDL *architecture* may only contain one *process*, ie. there is no support for concurrent processes or structural VHDL (*components*).
- No *signals*, other than the main *entity ports*, are allowed, since they are primarily used for connecting structural components and communicating between processes.
- Only one *wait* statement with a *sensitivity list* is allowed in the process, and then it may only be sensitive on special *ctrl* input.
- Procedures may only use *variable* class parameters and, since they must be declared in external packages, cannot access global signals or process variables.
- Various other useful features including: *inout* ports, VHDL *aliases*, signal and variable initialisation, and *exit* and *next* statements are also absent.

Full details of the original VHDL implementation may be found in the user manual "Writing Behavioural VHDL for MOODS Synthesis" [111]. The following sections describe how the above shortcomings are overcome in the enhanced compiler, VHDL2IC v1.5, showing how the various constructs are implemented in ICODE.

6.1.1 Overall VHDL Control Structure

Within a VHDL architecture all top level statements execute concurrently, forming in effect, a structural netlist description. Behavioural (algorithmic) code is implemented in VHDL processes, each of which describes a single concurrent block at the architecture level. Version 1.1 of VHDL2IC restricts an architecture to containing only one process, which must be sensitive on a special *ctrl* input signal, either through the process sensitivity list or using a *wait on ctrl* statement. Execution of each process iteration is controlled solely by means of *ctrl* edges, thereby enforcing a degree of simulation/synthesis correspondence.

This control model has two fundamental restrictions:

1. For almost any substantial design, insisting on the use of a single pre-defined *ctrl* input for process execution control, means that existing designs must be re-written, removing all sensitivity lists, and also requires external generation of the *ctrl* signal to re-create the intended control flow.

2. Many designs naturally split into concurrent communicating blocks, for example the video processor described in Chapter 7 uses three concurrently executing processes: one to access video memory, one to generate the video raster signal, and one to render images into the frame buffer. The lack of support for more than one process precludes this type of architecture, effectively synthesising a single threaded microprocessor for every design.

To address these, and other, shortcomings, the enhanced VHDL2IC implements a more complex control structure including support for parallel processes, and signal and variable initialisation. This, coupled with the full implementation of signals, and unrestricted sensitivity lists, addresses the major omissions from the original version. Figure 6.1 shows the skeleton control structure obtained from a VHDL source comprising two processes with various signal and variable initialisations.

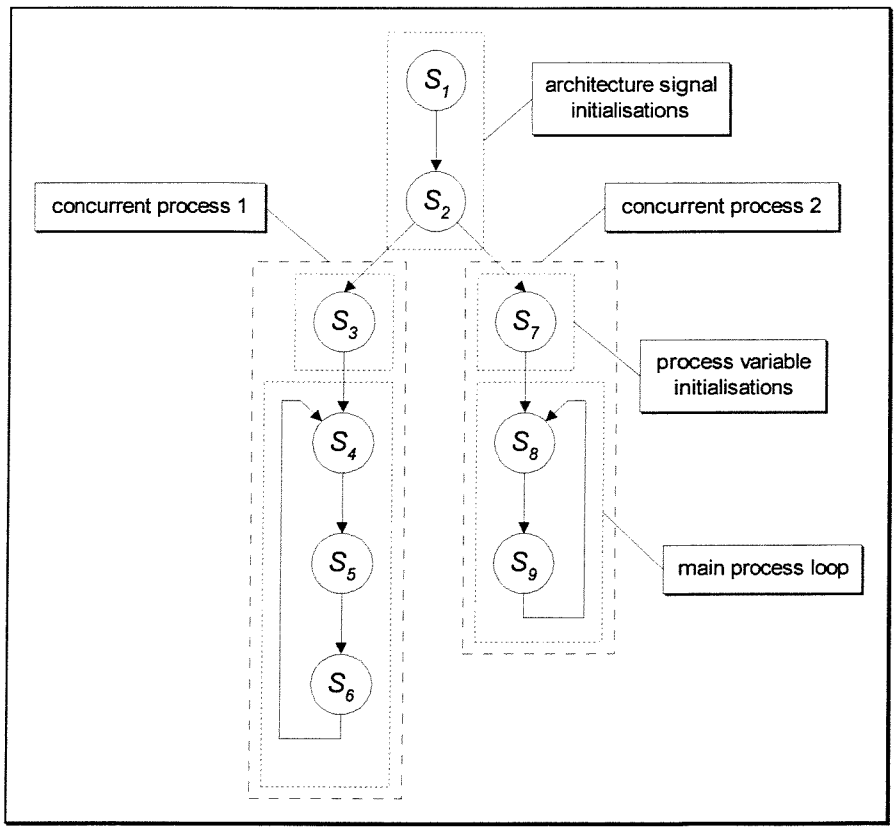


Figure 6.1 Skeleton VHDL control structure

Referring to this figure, the following features can be identified:

- Parallel processes are formed from disjoint free-running state machines forking off the initial line of control. In the diagram, the two processes branch off from fork node S_2 comprising states S_3 to S_6 and S_7 to S_9. Ignoring for the moment the signal and variable initialisations, each process is enclosed by a feedback arc (S_6 to S_4 and S_9 to S_8) which implements the implicit top-level process loop. Thus once entered, a process will repeat ad-infinitum until power off, as defined in the VHDL standard section 9.2. Note that processes are the only statements presently allowed in an architecture. If any other structural elements are required, they must either be enclosed in a single process, or manually added to the structural VHDL output from MOODS (see Chapter 7).

- Liberalisation of the wait statement (see section 6.1.3) and the associated removal of the *ctrl* signal means there is no hard-wired execution control with which to synchronise concurrent processes; instead the user must make use of handshaking with global signals, to perform explicit synchronisation where necessary. This relaxation of the strict VHDL simulation model allows maximum scheduling flexibility, requiring individual processes to only synchronise when required by the designer, and reduces the overall complexity of the control structure. Process synchronisation is discussed in detail in Chapter 7.

- Signal initialisation, (eg. *signal fred : bit_vector(7 downto 0) := "11001011";*) occurs only once, setting the initial value for the signal on a system reset. Since signals are only declared in the architecture, they are all classed as global from the process perspective, thus the signal initialisation states (S_1 and S_2) occur before control forks off into the processes.

- Variables are declared within a process but are still only initialised at reset (the current variable value is maintained across process iterations), thus the first states within a process implement variable initialisations, but are outside the outer process iteration loop.

6.1.2 Signals

VHDL defines an event based simulation cycle where time advances in steps; each step occurs at a wait statement with the sequential code between waits considered to execute in zero time. Signal assignments schedule events to update the value of the signal on the next time step, thus any writes to a signal do not actually happen until a wait statement is encountered, at which point all assignments since the previous wait are updated simultaneously (also see Chapter 3).

Behavioural synthesis systems (as described section 3.2), do not strictly follow this timing model, allowing statements between waits to execute over several cycles. To preserve the behaviour of signals from simulation to synthesis it is not sufficient to implement them simply as single registers, but to instead make use of a more complex deferred assignment model: when compiling VHDL into ICODE, each signal is implemented by one main register, together with an associated shadow register, holding the value to be assigned to the signal on the next simulation time step, ie. at the next wait. Figure 6.2 shows the ICODE implementation of signal *target* using two registers: *target*, holding the current signal value, and *target_tempsig*, holding the next assigned value. Any writes to the signal update *target_tempsig* (instruction *i2*), whereas reads access the main *target* register (*i1* and *i3*), thus even after an assignment, the old value is still returned until a wait statement is encountered, at which point the main register is updated to reflect the new signal value (*i4*).

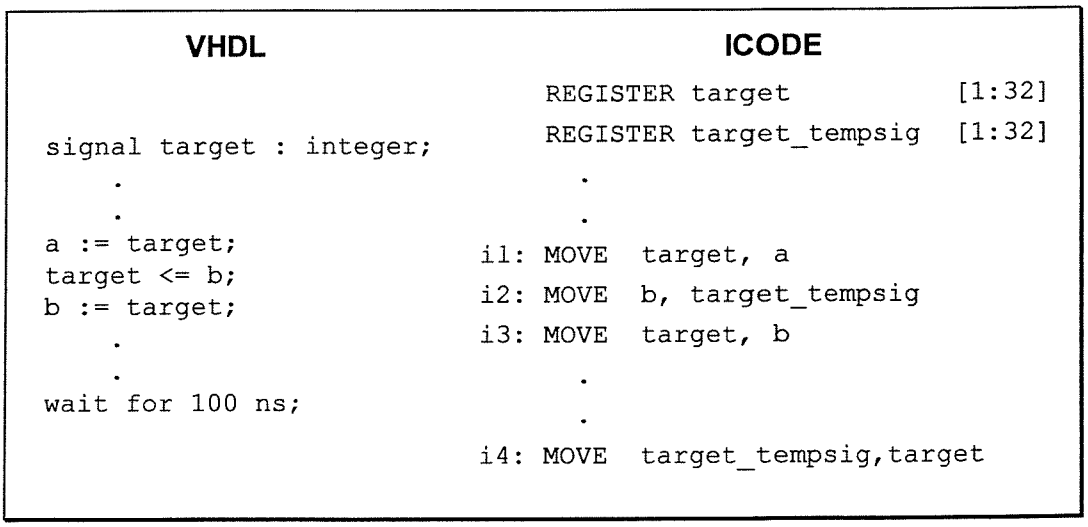


Figure 6.2 Compiling VHDL signals into ICODE

The compiler tracks where within a process signal assignments occur and, when translating a wait statement, only updates those signals modified since the previous wait.

Entity I/O ports are also defined as signals with the addition of a *mode* specifier, determining the direction of data flow though the port (VHDL standard section 1.1.1.2). There are four modes supported by VHDL2IC:

1. Mode *out* ports may only be written to, and exhibit the same deferred assignment behaviour as signals. Thus assignments to an output write to a shadow register, with the actual output port only being updated at the following wait.

2. Ports of mode *buffer* are identical to outputs except that the current output value may also be read internally. Since ICODE output ports are actually registers, allowing write and read access, buffer ports can be implemented identically to out ports.

3. Mode *in* ports, representing read-only inputs, are somewhat different to normal signals. Since, according to the VHDL simulation model, time only passes at wait statements, inputs will only change at these points, thus within sequential code, the input value is guaranteed to be stable. This does not, however, apply to synthesised systems where an input could change at any point during process execution. To ensure the stability of inputs for the duration of a wait-wait section, each has an associated shadow register which is updated with the current port value on termination of a wait. When reading from an input, the shadow register, and not the input port, is accessed. Note the distinction between outputs, updating at the start of a wait, and inputs and the end.

4. Mode *inout* defines bi-directional ports that may be read from and written to, both internally and externally. In practice, inout ports require some mechanism for controlling the flow of data and determining which, of several connecting drivers, represents the true port value, eg. is the input value the last output value or the external driving value ? This is achieved through the use of *resolution functions* to arbitrate between multiple connections to a single signal, however these are not currently supported by VHDL2IC and so the approach taken is to split an inout port into two halves, forming separate in and out mode components. The compiler ensures that writing to the port updates the output half, while reading accesses the input half. If a true tri-state bi-directional port is required, the two halves can be re-combined with a slight modification to the MOODS

structural output according to the requirements of the particular low-level synthesis system used. An example of this is given in Chapter 7.

6.1.3 Wait Statements

The *wait* statement causes all recently assigned signals and ports to update from their shadow registers, and suspends a process until certain conditions are met. Figure 6.3 summarises the syntax of the wait statement as defined in the VHDL standard, section 8.1.

```
wait_statement ::=
  wait [sensitivity_clause] [condition_clause] [timeout_clause];
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name {, signal_name}
condition_clause ::= until condition
condition ::= boolean_expression
timeout_clause ::= for time_expression
```

Figure 6.3 Wait statement syntax

A wait may include up to three different clauses which together determine its termination condition. Put simply, a wait causes a process to hang until at least one signal in the *sensitivity list* changes state and the condition in the *condition clause* is met, or the specified timeout (with respect to the start of the wait) is exceeded. The VHDL to ICODE compiler converts the wait into a sequence of equivalent ICODE instructions, the overall operation of which is depicted in the flow chart of Figure 6.4. This splits into four main sections:

1. The first step is to update all signals and output ports assigned to since the previous wait from their shadow registers. During optimisation, MOODS almost invariably schedules these instructions in a single control state, thus ensuring all signals are updated simultaneously.

2. The wait sensitivity list brings the process to a halt until an event (a change in value) is detected on any one of its constituent signals. In the ICODE version, each signal is copied into a temporary register, which is then continually compared to the current signal value until a change occurs. Note that when checking input ports, the real port must be used, and not its shadow register, which is guaranteed to be stable.

3. The wait condition clause hangs a process until the condition is met. In the presence of a sensitivity list an event must occur on one of the member signals before the condition clause is evaluated. If the result is false, another event must occur before the condition is re-evaluated. Again, references to input ports must be to the port itself and not the shadow register. Note that if the sensitivity list is not present, the condition must change to true before the wait terminates, thus even if the condition is met on entry, the wait will still hang until it first goes false, and then true again. This special case is dealt with by treating the entire condition as a sensitivity list, only exiting when a change to true occurs.

4. Finally, once the wait has terminated, all input port shadow registers are updated before the next block of sequential code executes, thus ensuring accesses to input ports return the input value at the end of the last simulation time step.

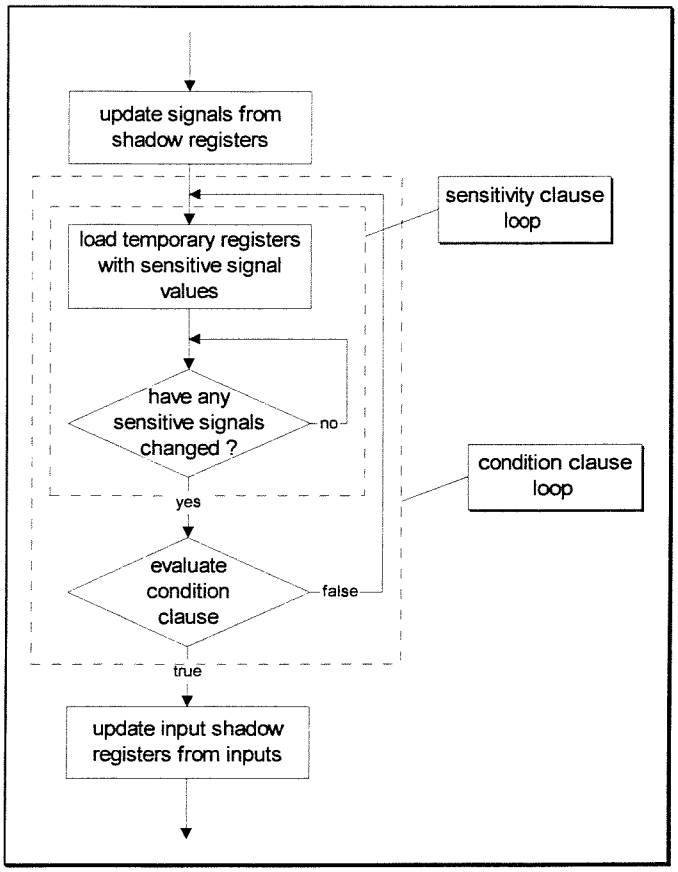


Figure 6.4 Wait statement flow chart

MOODS does not, at present, implement any form of local timing constraints that could be utilised in conjunction with the timeout clause. The exact time values are therefore ignored by the compiler, however, all the output and input signal updates are still performed,

effectively implementing a timeout of zero. Pure wait for timeout statements are most useful for adding approximate cycle delays in the source to emulate the final scheduling (see Chapter 7).

Figure 6.5 brings all the above features together in a single example demonstrating the ICODE implementation of a wait statement with shadow register updates, a sensitivity list and a condition clause.

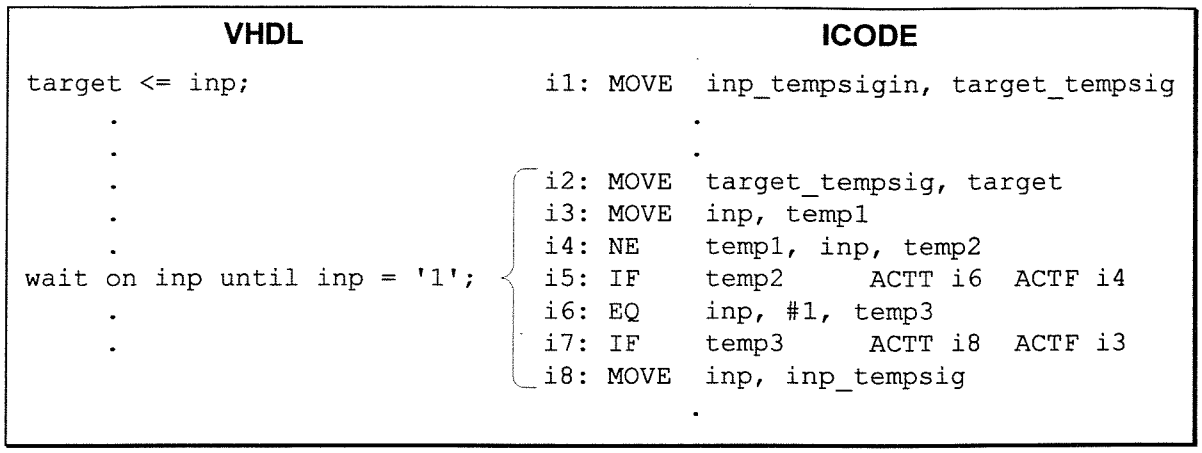


Figure 6.5 ICODE implementation of a wait statement

One feature not implemented in VHDL2IC is a strict interpretation of the VHDL simulation cycle, in particular, the implicit synchronisation of all processes at waits (as in the unrestricted model of [73], described in section 3.2). Recall the VHDL standard specifies that time only passes in wait statements; thus parallel processes are effectively kept in lockstep as they are all guaranteed to enter waits at the same time, and are therefore implicitly synchronised at these points. For example, in the two concurrent processes shown in Figure 6.6, the wait in process *P2* will synchronise alternately with each one in *P1*, irrespective of the complexity of any intermediate sequential statements.

Behavioural synthesis does not naturally follow this strict timing regime (otherwise very little scheduling optimisation would be possible), with sequential code instead executing over a number of clock cycles. Any attempt at implementing the full VHDL process timing model will therefore have to

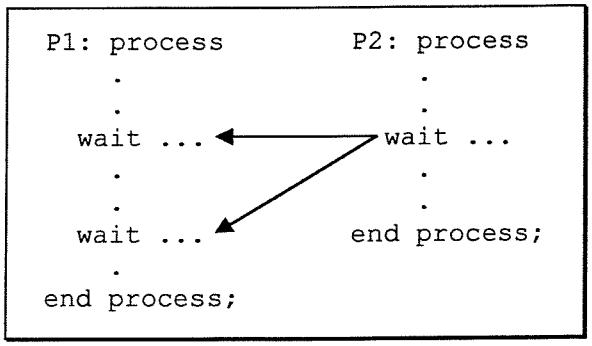


Figure 6.6 Implicit process synchronisation

compromise somewhere along the line. The unrestricted model of the CAMAD system forces concurrent processes to implicitly synchronise by halting execution at the start of a wait until all other processes are in the same state. This may be achieved relatively easily using a single "waiting" flag (1-bit register) per process where entry into a wait asserts the flag but does not allow any further action (ie. updating signals from shadow registers), until all other waiting flags are set. Once the wait has terminated, the flag is cleared.

The main drawback of this approach is that it can lead to processes remaining idle for a large period of time while waiting for the others the synchronise, even if there is no inter-process interaction. The approach taken in VHDL2IC is to allow processes complete independence; synchronisation being left up to the user through the use of handshaking via global signals. This is similar to the restricted model in CAMAD except that the handshaking mechanism is not encapsulated in pre-defined procedures (although there is no reason why it could not be). Requiring synchronisation to be explicitly specified also forces the user to consider process interaction more carefully, and not to rely on apparent timing relationships which may not always apply, especially after synthesis. Chapter 7 shows how synchronisation can be accomplished in practice, and highlights some of the pitfalls that must be avoided to achieve reliable inter-process communication.

On a final note, [83] describes the behaviour of signals external to processes including the implementation of *bus* and *register* kinds, and the use of resolution functions allowing two or more processes to concurrently write to a single signal. The VHDL2IC compiler does not at present support these features, however this has proved to pose little real restriction in practice, and indeed none of the benchmark designs need them either.

6.1.4 Subprograms

VHDL subprograms (functions and procedures) are implemented as separate ICODE modules (not to be confused with data path modules), executed via a MODULEAP instruction. An ICODE description comprises a top-level PROGRAM module (equivalent to the VHDL architecture), which implements all processes, together with a number of smaller modules for each VHDL subprogram. A module also has a number of I/O parameters which are mapped onto top-level registers via the calling MODULEAP instruction. MOODS does not optimise between modules, instead creating a separate

control and data path for each, which are activated by call nodes in the main control path. Subprograms therefore provide a mechanism by which a designer can impose, to some degree, a physical partitioning of the final implementation.

VHDL subprograms can communicate with the rest of the design in three ways:

1. Subprograms have I/O parameters just like their software equivalents. VHDL *in* and *out* mode parameters are mapped directly onto the *inport* and *outport* ports of the ICODE module; while *inout* parameters are split into separate input and output components, connecting in the module call to the relevant reading and writing ICODE variables. For example, Figure 6.7 shows how an inout subprogram parameter connecting to a signal *S* will be split such that the input part reads from the main signal register, while the output writes to the associated shadow register.

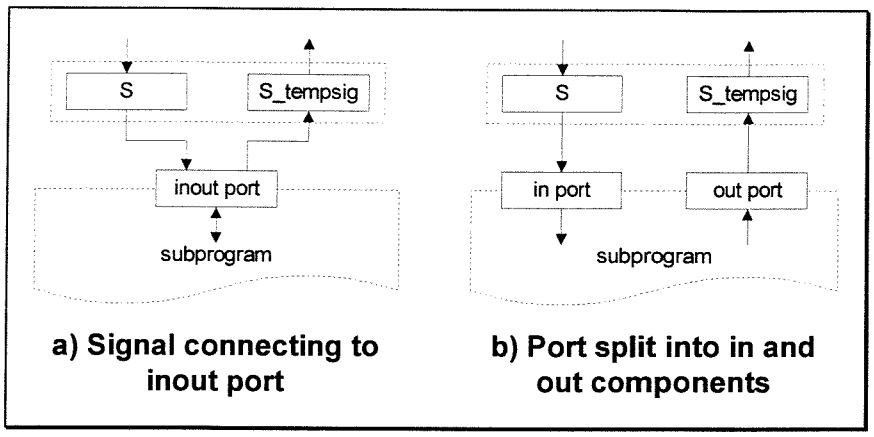


Figure 6.7 Splitting subprogram inout ports into separate in and out parameters

VHDL defines three different *classes* for subprogram parameters whose translation into ICODE is illustrated in Figure 6.8. *Constant* and *variable* are the default classes for *in* and *out* parameters respectively, and specify that the value of the calling signal or variable is copied in and out at the start and end of the subprogram, ie. parameter passing by value. Thus, if a procedure call connects a signal to a variable class output, any output assignments within the procedure will not be assigned to the controlling signal until the procedure has completed. Even then, because subprograms are treated as assignments, the actual register in the procedure call parameter list is the shadow register, the signal itself not being updated until the following wait statement (see Figure 6.8 notes 6 and 8).

Signal class parameters however, connect directly to the calling signal or variable, ie. parameters are passed by reference. Within the subprogram, signal parameters behave identically to normal signals with full deferred assignment requiring local shadow input and output registers (Figure 6.8 notes 1, 2 and 3), which are only updated on a wait (Figure 6.8 notes 4 and 5). At the calling end, because the procedure can update the current signal value without exiting, the signal itself, and not its shadow register, appear in the calling parameter list (Figure 6.8 note 7).

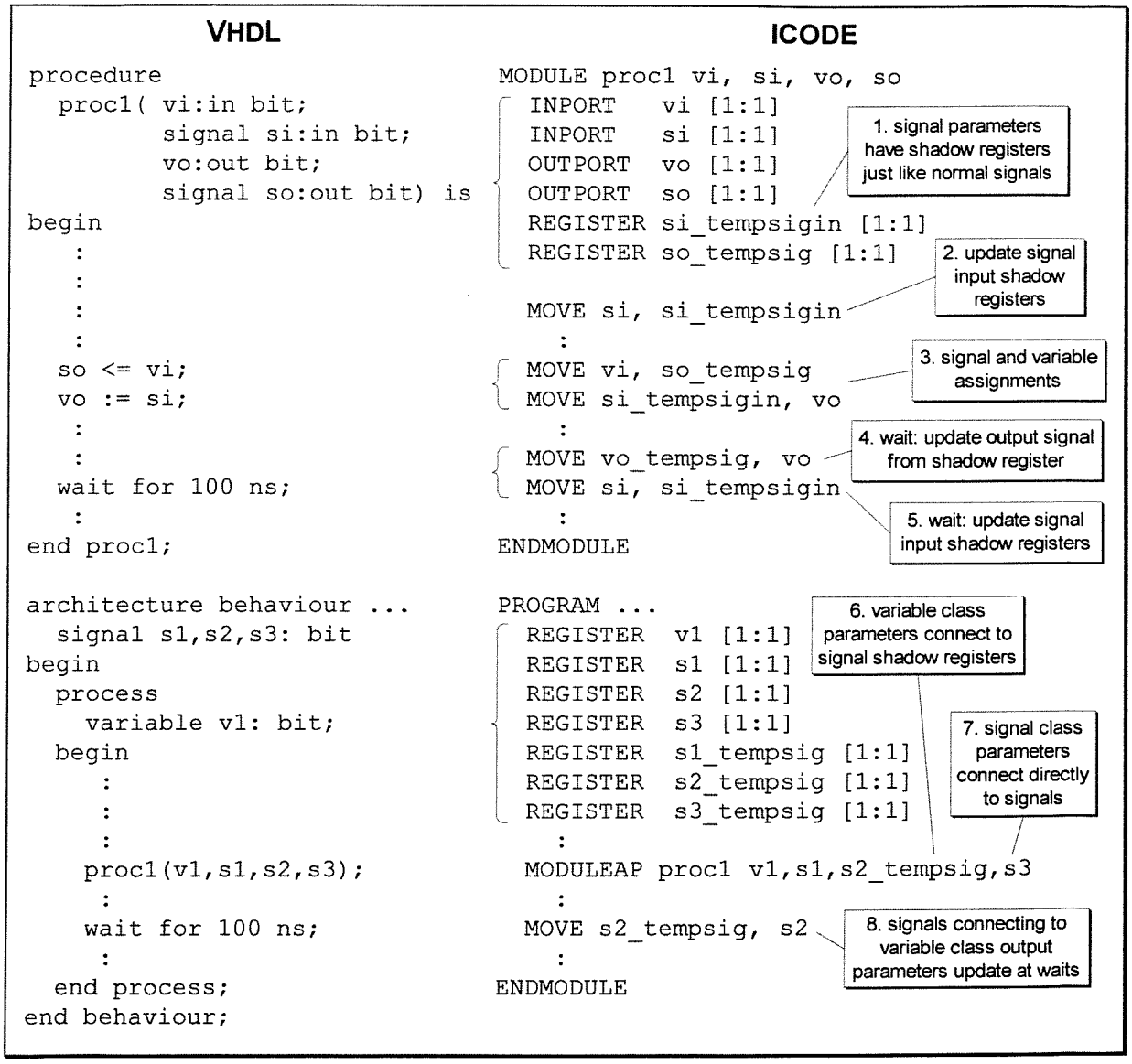


Figure 6.8 Translating VHDL subprogram parameters into ICODE

2. If a subprogram is declared inside an architecture, as opposed to being in an external package, it may directly access all global signals and entity I/O ports. ICODE variables declared in the top-level PROGRAM module are considered global, being visible throughout the entire design, including other modules. Subprograms, therefore,

automatically have access to all signals and entity ports. The only other consideration is that where a subprogram contains a global signal assignment, every wait statement following a call to that subprogram must include an update of the relevant signal from its shadow register.

3. Subprograms defined within a process may directly access process variables (within the scope of the parent process) in addition to the global signals and ports. All process variables are implemented as top-level ICODE variables, with the compiler ensuring that any name clashes between different processes are resolved by preceding the ICODE variable name with the process name. Again, the global nature of PROGRAM variables means that all ICODE modules can automatically see all process variables, however the VHDL parsing stage ensures that only those variables declared in the parent process of the subprogram are accessible.

6.1.5 Aliases

VHDL *aliases* are used to partition a bit vector into several logical sub-blocks, which may then be individually accessed via the alias. These bit slices are useful in a number of situations such as identifying individual status bits in a microprocessor status register, or slicing an instruction into opcode and operand fields. They are also of particular use in expanded modules templates.

ICODE supports aliasing as a native feature, allowing a variable to be declared as a sub-range of a parent. Thus translating VHDL aliases is simply a matter of mapping the VHDL syntax onto its ICODE equivalent, as illustrated in Figure 6.9.

VHDL

```
variable inst : bit_vector(31 downto 0);
alias cond    : bit_vector(3 downto 0)  is inst(31 downto 28);
alias opcode  : bit_vector(3 downto 0)  is inst(24 downto 21);
alias rn      : bit_vector(3 downto 0)  is inst(19 downto 16);
alias rd      : bit_vector(3 downto 0)  is inst(15 downto 12);
alias rop2    : bit_vector(3 downto 0)  is inst(3 downto 0);
alias shift   : bit_vector(7 downto 0)  is inst(11 downto 4);
alias imed    : bit_vector(11 downto 0) is inst(11 downto 0);
```

ICODE

```
REGISTER inst   [0:31]                        // instruction word
ALIAS    cond   [0:3]  FROM inst [28:31]      // condition
ALIAS    opcode [0:3]  FROM inst [21:24]      // operation code
ALIAS    rn     [0:3]  FROM inst [16:19]      // operand 1 register
ALIAS    rd     [0:3]  FROM inst [12:15]      // destination register
ALIAS    rop2   [0:3]  FROM inst [0:3]        // operand 2 register
ALIAS    shift  [0:7]  FROM inst [4:11]       // operand 2 shift
ALIAS    imed   [0:11] FROM inst [0:11]       // operand 2 value
```

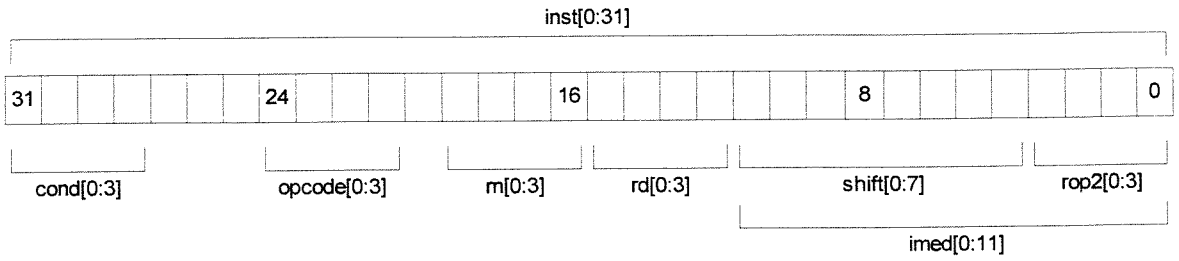


Figure 6.9 VHDL and ICODE aliases

6.1.6 Next and Exit

The final significant enhancement to the ICODE compiler is the inclusion of support for *next* and *exit* statements. These are similar to the standard C *continue* and *break* instructions, allowing a loop to skip immediately to the next iteration (next), or terminate immediately (exit). Both statements are directly implemented as extra arcs on the control graph, as shown in Figure 6.10. Exit is particularly useful as it enables the construction of loops within a single control state ("zero-delay loops"), and also allows nested loops to be created whereby the inner loops do not require an extra cycle for counter re-initialisation. These structures are vital to the operation of the video processor, described in detail in Chapter 7, which requires exact cycle-by-cycle control over the optimised schedule.

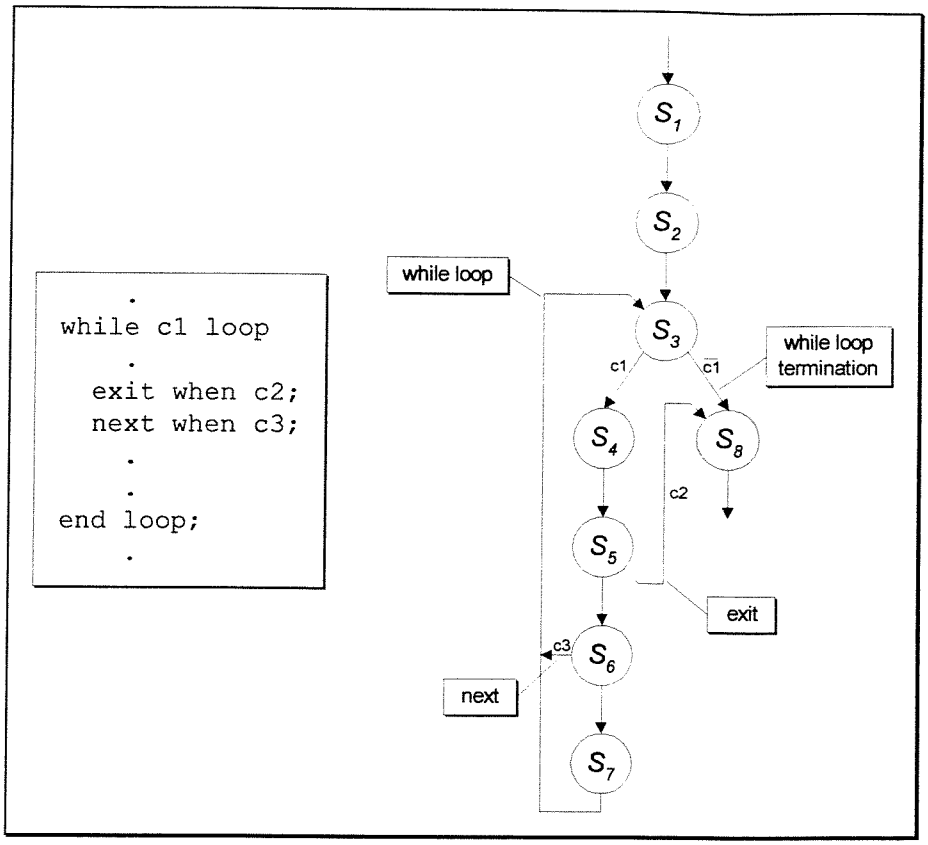


Figure 6.10 Next and exit control structure

6.2 Practical Considerations

The development of a more accurate VHDL representation, including support for parallel processes and deferred signal assignment, allows a designer to make full use of the power and flexibility provided by a concurrent architecture, while maintaining a substantial degree of simulation/synthesis correspondence. The research described in section 3.2 achieves similar goals, however there is generally little consideration of the added costs involved in, for example, adding shadow registers to every port and signal. Furthermore, despite the improved implementation of the VHDL timing model, there are still inherent differences between the simulation of the behavioural source, and the final structural description. The following sections examine some of the implications of casting the VHDL event-based model into the behavioural synthesis domain, on both the cost and performance of the final synthesised implementations.

6.2.1 Costing the VHDL Simulation Model

The inclusion of signal shadow registers and their associated update instructions, together with the extra control involved in complex wait clauses, must have some impact on the area and delay of the final structure. This section compares the results of synthesising a number of designs with and without these extra features in an attempt to quantify the overheads involved.

Table 6.1 lists area and delay figures obtained from heuristic optimisation of a number of behavioural benchmarks [106, 107]. For each optimisation type (area or delay priority) the enhanced VHDL2IC model, using full deferred signal assignment and waits, is compared to a stripped-down implementation, with the signal shadow registers and their associated update instructions removed (obtained using the VHDL2IC **/no_sigs** command line switch).

Results for eight designs are listed, of which seven are benchmarks, and one is the video processor component, **VidProc**, of the project described in Chapter 7. For each one, the table lists the size of the VHDL source and the number of compiled ICODE instructions for both VHDL models, together with area and delay optimised results. These show the figures reported by MOODS after optimisation for the final area, the functional, storage, interconnect and control area breakdowns and the total delay. There is also a rough summary of the makeup of the VHDL source, listing the number and size of the larger (wider than 1-bit) signals, ports and variables, and the total number of processes and wait statements, ie. the main VHDL elements affected by the enhanced model. Note that each set of results is identified by a two-letter code where "AN" represents the area optimised, non-enhanced model; "AE" the area optimised, enhanced model; "DN" the delay optimised, non-enhanced model; and "DE" the delay optimised, enhanced model.

The results are summarised in Figure 6.11 which plots the difference in area and delay between the two models, as a percentage of the non-enhanced version, in other words the cost increase due to the extra registers and instructions.

Design	Lines of VHDL	ICODE insts	Full model insts	Δ insts (%)	Optimised		Area (μm^2)	Area breakdown (%)				Delay (µs)	Δ area (%)	Δ delay (%)
								Func	Store	Inter	Cont			
alu_beh - 2×32 bit in/out ports, 9×25 bit signals, 8×32 bit signals, 2×25bit variables, 12×32 bit variables, 6 waits, 2 processes	121	80	113	41.3	Area	AN	364938	71.4	21.4	6.5	0.7	4.89	7.7	0.0
						AE	393118	66.3	25.7	7.3	0.6	4.89		
					Delay	DN	386694	73.8	21.2	4.7	0.3	2.57	5.0	0.0
						DE	406104	70.3	23.7	5.7	0.3	2.57		
DHRC2 - 7×16 bit in/out ports, 12×16 bit variables, 10×32 bit variables, 1 wait, 1 process	105	46	68	47.8	Area	AN	272266	85.5	8.4	5.7	0.5	5.32	2.3	15.4
						AE	278581	83.5	9.9	6.0	0.5	6.14		
					Delay	DN	382026	90.9	5.6	3.2	0.3	2.84	1.7	18.2
						DE	388496	89.4	6.7	3.6	0.3	3.36		
Diffeq - 5×32 bit in/inout ports, 14×32 bit variables, 4 waits, 1 process	95	39	80	105.1	Area	AN	281909	82.6	12.7	4.3	0.4	3.43	5.3	10.4
						AE	296859	78.4	16.4	4.7	0.4	3.78		
					Delay	DN	963830	94.1	4.7	1.1	0.1	1.82	1.0	49.9
						DE	973428	92.6	6.3	1.0	0.1	2.73		
Ellip - 8×16 bit inout ports, 30×16 bit variables, 2 waits, 1 process	92	43	76	76.7	Area	AN	91873	63.8	24.6	8.6	2.9	8.61	26.5	7.4
						AE	116183	50.5	38.7	8.3	2.5	9.25		
					Delay	DN	159668	75.9	14.2	9.2	0.8	1.84	5.2	15.4
						DE	167978	72.1	17.3	9.7	0.9	2.12		
FFT2 - 6×32 bit in/out ports, 20×32 bit variables, 1 wait, 1 process	155	96	113	17.7	Area	AN	317941	74.3	17.7	7.2	0.8	6.25	4.7	6.3
						AE	332761	71.0	20.7	7.5	0.8	6.64		
					Delay	DN	552104	83.7	10.8	5.1	0.4	3.35	2.7	7.7
						DE	566924	81.5	12.7	5.3	0.4	3.60		
FM8501 - 3×16 bit in/out ports,10×16 bit signals, 6×16 bit variables, 2×32 bit variables, 20 waits, 2 processes	362	328	451	37.5	Area	AN	131985	22.5	30.6	42.9	3.9	2.56	8.0	100.2
						AE	142514	20.0	41.5	31.6	6.9	5.13		
					Delay	DN	162420	24.4	26.6	45.9	3.1	1.28	1.7	71.2
						DE	165179	27.4	36.1	30.8	5.6	2.20		
Tseng - 15×32 bit inout ports, 18×32 bit variables, 4 waits, 1 process	151	67	170	153.7	Area	AN	353192	68.9	27.3	3.5	0.3	3.43	13.8	50.0
						AE	401822	60.5	36.0	3.1	0.4	5.14		
					Delay	DN	354212	69.1	27.2	3.4	0.3	2.32	13.6	22.2
						DE	402542	60.8	35.9	3.0	0.3	2.83		
VidProc - 7×8 bit in/out ports, 3×14 bit in/out ports, 3×8 bit signals, 7×8 bit variables, 23 waits, 3 processes	428	138	347	151.4	Area	AN	41763	32.0	50.5	12.2	5.3	0.72	21.6	55.6
						AE	50768	30.5	48.7	13.8	7.1	1.12		
					Delay	DN	41963	31.8	50.3	12.2	5.7	0.47	28.8	40.4
						DE	54058	31.7	48.5	13.2	6.7	0.66		

Table 6.1 Enhanced VHDL area and delay overheads

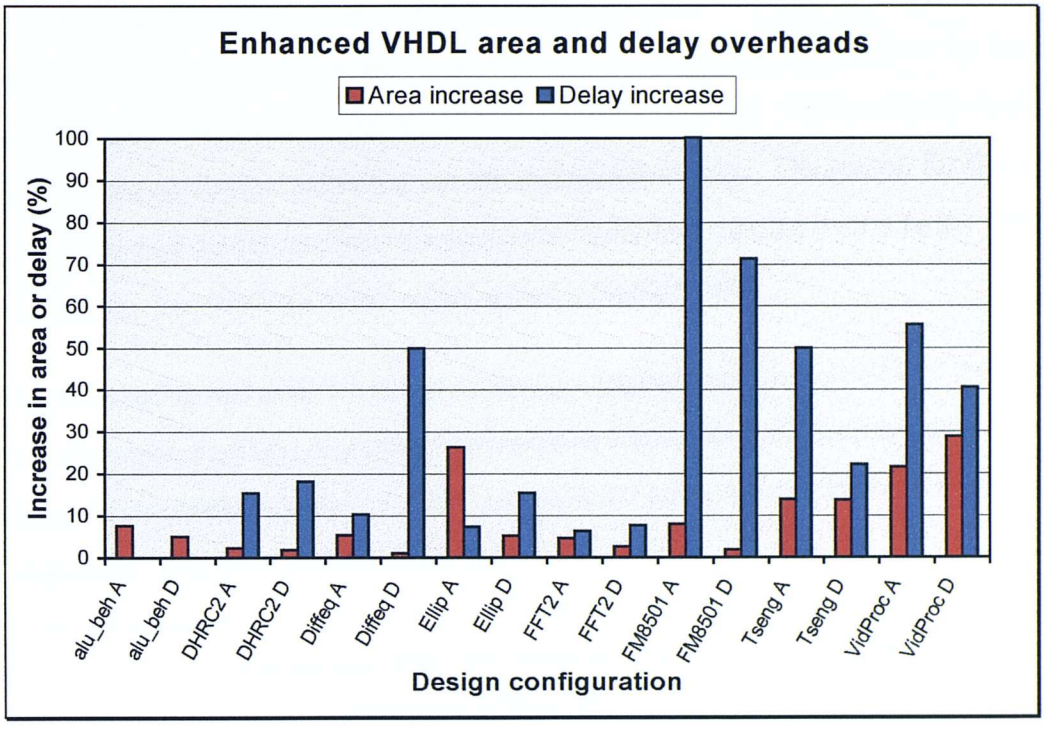


Figure 6.11 Increase in area and delay with deferred signal assignment

The results provide a mixed picture, suggesting that the enhancements can result in anything from a 1% to 30% increase in the final area, and a 0% to 100% increase in the final delay. Clearly, although the added components can have a significant impact, the degree to which they affect the final outcome is highly design dependent.

Considering the effects on area first, Figure 6.12 shows area breakdowns for a number of the designs in both their enhanced and non-enhanced forms. These figures have been normalised such that for each enhanced/non-enhanced pair, the total enhanced area is defined as 100, with both results scaled accordingly. Looking at this graph there appear to be three main factors coming into play:

1. The addition of shadow registers results in an increase in the storage area component. Since a shadow is created for every port and signal in the design, the larger the size and quantity of these, the greater the increase in storage requirements and hence the greater the increase in total area. This is less pronounced in the delay optimised figures as state merging tends to bypass (and therefore eliminate) a large proportion of the added registers.

2. Designs with a high functional component generally show less of an increase than their control-dominated counterparts. Thus **DHRC2** and **diffeq** are much less affected by the

enhancements than **Tseng** and **VidProc**. **FM8501** is an interesting deviation from this general rule as it has the smallest functional component, but does not show the largest area increase. This is due to the high interconnect component, which actually decreases in the enhanced version balancing out the increase in storage. The reason for this is that in the enhanced model, the implementation of variables is spread over a larger number of registers (ie. less sharing), thus requiring fewer multiplexors.

3. The enhanced model adds a number of shadow register update instructions around each wait statement, resulting in a general increase in the number of control states. Thus designs with greater number of waits tend to show an associated increase in the control component, as in **FM8501** and **VidProc**.

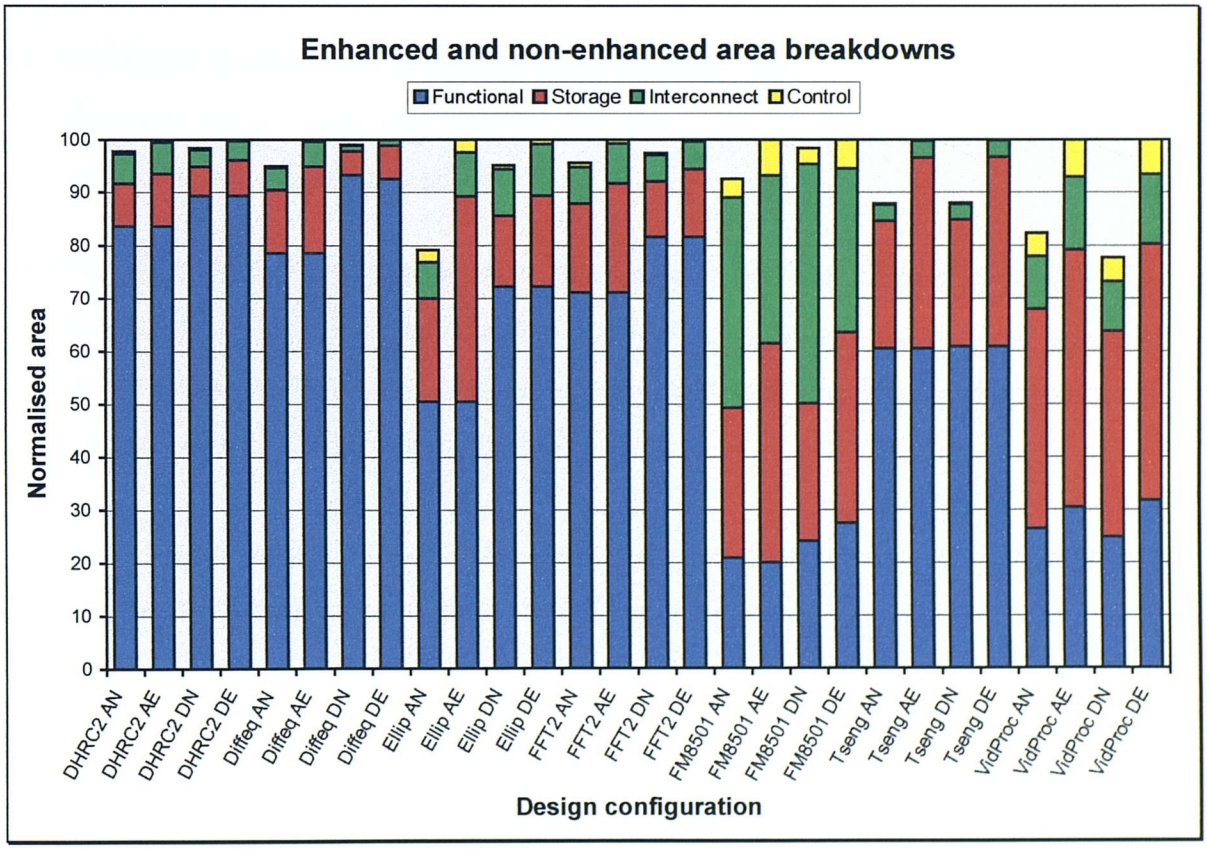


Figure 6.12 Normalised area breakdown with and without deferred signal assignment

As far as delay is concerned, there are two main factors affecting the increase:

1. The added instructions around wait statements increase the critical path length, thus the greater the number of waits, the greater the increase in delay. Note that since all update instructions tend to be scheduled in a single state, the overhead is not highly dependent

on the number or size of signals and ports. Hence the largest increase is visible in those designs with the greatest number of waits, namely **FM8501** and **VidProc**.

2. The increased complexity of the enhanced designs means that there are more possible optimisation configurations which, as demonstrated in the previous chapter, make the synthesis task more prone to inefficiency. This will be affected both by the number of waits and the number of ports and signals.

Thus, a few guidelines can be suggested for attempting to minimise the cost of the enhanced model in new designs:

- Limit ports and signals to the minimum required width and quantity.
- Implement as much code as possible using variables, and assign signals and ports in one block just before a wait, thus ensuring that where possible, shadow registers will optimised away (see next section).
- If an even smaller area is required and the design cannot be optimised any more, consider using the VHDL2IC **/no_sigs** switch but remember: a) outputs and signals will update immediately they are assigned; and b) inputs must remain stable between waits, unless they are definitely not being read.

This final suggestion has serious ramifications for the designer as, not only does it alter the timing, it also effectively re-defines the VHDL standard on signals. In some cases, however, with careful consideration of the implications, the ends may justify the means: as is demonstrated in Chapter 7, this may be the only way of sufficiently shrinking a design to fit on the target device.

6.2.2 Timing in the Enhanced Model

A fundamental problem for all VHDL behavioural synthesis systems is the difference between simulation of the behavioural source and the actual optimised hardware. The price paid for allowing synthesis scheduling flexibility is that the schedule is not pre-defined, thus the circuit timing cannot be exactly determined until synthesis is complete. The problem is tackled in a variety of ways discussed in Chapter 3, generally falling into two camps: those

which restrict the flexibility of the scheduling process, requiring an almost complete timing specification in the source, such as in the CALLAS [69, 82] system; and those that implement some form of local constraints so that only timing-critical sections of the design need be pre-defined, for example CAMAD [73].

For MOODS, expansion of the supported VHDL subset means that timing is no longer determined by an external *ctrl* signal, but is dependent on the scheduling transformations applied during synthesis. For the most part, the user will not be over concerned with the exact scheduling within a process, however there are a number of critical points at which the cycle-by-cycle operation needs to be controlled. These tend to fall into two categories: the relative timing of output transitions (including internal signal writes), for example a memory r/w signal must change after the address and data become valid; and the timing of input transitions (again, including internal signal reads), ie. when is an input allowed to change?

In VHDL, the user defines the relationships between I/O transitions through the use of wait statements, based on the assumption that all outputs change at a wait, and any input changes are detected with a sensitivity list. MOODS attempts to reproduce this behaviour using shadow registers, however there are a number of caveats which must be borne in mind when developing and operating a synthesised system.

Considering the timing of inputs first of all, the primary mechanism in VHDL for detecting transitions is the wait sensitivity list. Figure 6.13 shows the VHDL source and equivalent ICODE for part of a process in which each iteration of a loop is triggered by a rising edge on the *inp* input. The ICODE is based on the enhanced wait model shown earlier in Figure 6.5. When synthesised, the control graph in Figure 6.13a is obtained; notice how the wait is implemented using two states: S_1 sets a temporary variable to the initial value of *inp*, which is then continually checked against current *inp* value in S_2. Once a change is detected, the condition clause is evaluated, either exiting the wait block, or returning back to S_1 to detect another input transition.

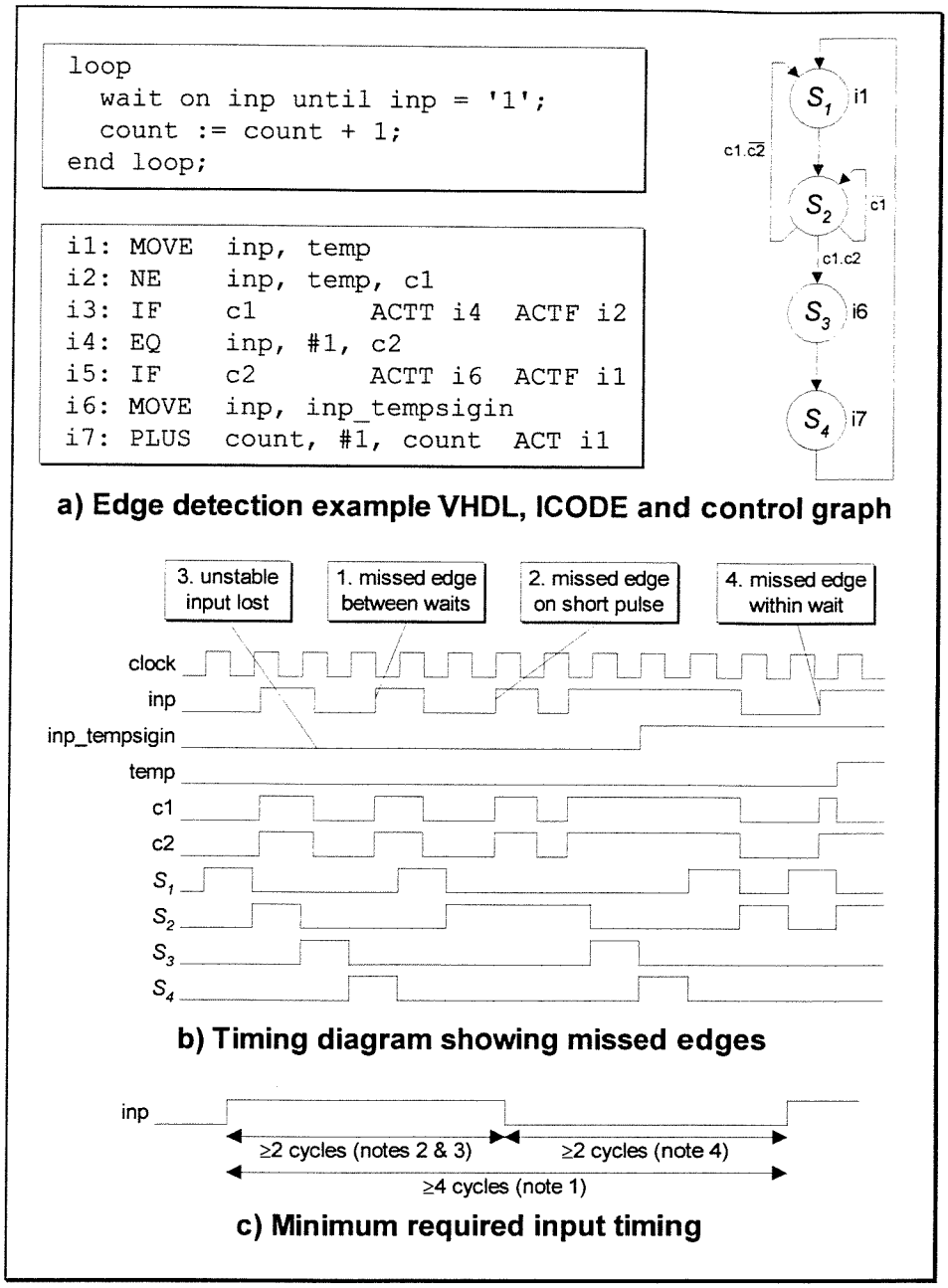


Figure 6.13 Timing of sensitivity list input transitions

The timing diagram in Figure 6.13b illustrates several pitfalls which may trap the unwary user. There are four typical situations highlighted in the diagram, which can result in missed edges:

1. Whereas in the VHDL simulation model the delay between two adjacent waits is considered to be zero, in reality there may be any number of intermediate control states. In Figure 6.13a, ICODE instructions *i6* and *i7* each require a separate state, thus there will always be a delay of at least 2 clock cycles between waits on sequential loop iterations. This means that any activity on the *inp* input during this time will go

undetected, probably resulting in skipped edges. From the user perspective, this delay determines the minimum allowed time between input edges to guarantee detection, in this case 4 cycles (2 for the wait-to-wait delay and 2 for the wait statement itself).

2. Since MOODS synthesises a synchronous design, it is imperative that input pulses are long enough to be latched on a clock edge. This will only be guaranteed by specifying the minimum pulse width to be equal to the clock period.

3. Once the wait has completed, all input ports will update their shadow registers, thus the stability of the changing input must be guaranteed for the cycle following a true evaluation of the condition. This adds an extra cycle onto the minimum pulse width taking it up to 2 cycles.

4. When an edge occurs that does not meet the wait condition clause (ie. instruction *i4* returns false), the temporary variable used for edge detection must be reset to the current *inp* value, requiring an additional clock cycle for S_1. If, however, another edge occurs before the temporary is re-loaded it will be ignored. Thus in the example, to guarantee detection, the minimum allowed negative pulse width must be at least 2 cycles tying neatly in with the previous item, which also requires a degree of stability after a positive edge.

Hence for the example, the minimum input timing to guarantee detection of an edge can be determined, as shown in Figure 6.13c.

The wait schedule described above is fairly typical, however if a long sensitivity list or complex condition is used, more than 2 states may be required, in which case the timings deteriorate even further. For this reason, it is better to use simple wait clauses to achieve the tightest detection loop. Note that these timings also apply when synchronising processes on signal edges, where the designer must guarantee that a process is in a waiting state before the synchronisation signal changes. This is discussed in more depth in Chapter 7.

Now consider the example in Figure 6.14 which shows a possible VHDL fragment for controlling a write to an external static memory. For simulation purposes each wait has a delay of 50ns (ignored during synthesis), which represents the passage of time during one

clock cycle, to produce the desired output timing in Figure 6.14a. The resulting ICODE in Figure 6.14b illustrates the added complexity of the enhanced signal model, with each VHDL assignment effectively requiring two ICODE instructions. Note that since the waits contains no sensitivity list or condition clause their only purpose is to update the outputs and emulate a clock cycle during simulation. The control graph obtained after synthesis, and its associated output timing are also shown in the figure, about which the following observations can be made:

- *Sequential merge* and *group on register* transforms have effectively eliminated the need for the shadow registers, assuming they are not required elsewhere in the design. This goes some way to reducing the overhead imposed by deferred signal assignment, and does not generally damage the timing as merge transformations move the original assignment down the schedule, to the same position as the output update instruction. Thus to ensure maximum elimination of shadow registers, it is best to perform all signal assignments as close to a wait statement as possible.

- The merging of control states, while beneficial in most cases, has also wrecked the desired output timing between the *addr/data* signals (*i1* to *i4*) and the *r/w* falling edge (*i5* to *i6*), all of which have been merged into one state, S_1. To solve this problem the extended ICODE PROTECT instruction is used. Recall from section 5.3.1 that a PROTECT placed at a point in the ICODE prevents any of the following instructions being scheduled concurrently, or before it. Thus if a PROTECT is placed between instructions *i4* and *i5* in the example, the desired timing wlll be achieved, as *i5* and *i6* will no longer be able to be scheduled in S_1. Of course this requires manual editing of the ICODE file after compilation which may not always be desired, therefore a new VHDL2IC **/protect** command-line switch has been added to automatically insert a PROTECT instruction in all waits with a non-zero timeout. Figure 6.14c contains the resulting ICODE, optimised schedule and timing diagram, this time showing the desired behaviour. Since the PROTECT is placed after the signal update instructions, shadow registers may still be removed via state merging transformations. Note that if a wait with a sensitivity list or condition clause is used, the subsequent instructions are guaranteed not to be moved up the schedule due to the dependencies created by conditional control arcs in the wait, thus the PROTECT instruction is unnecessary.

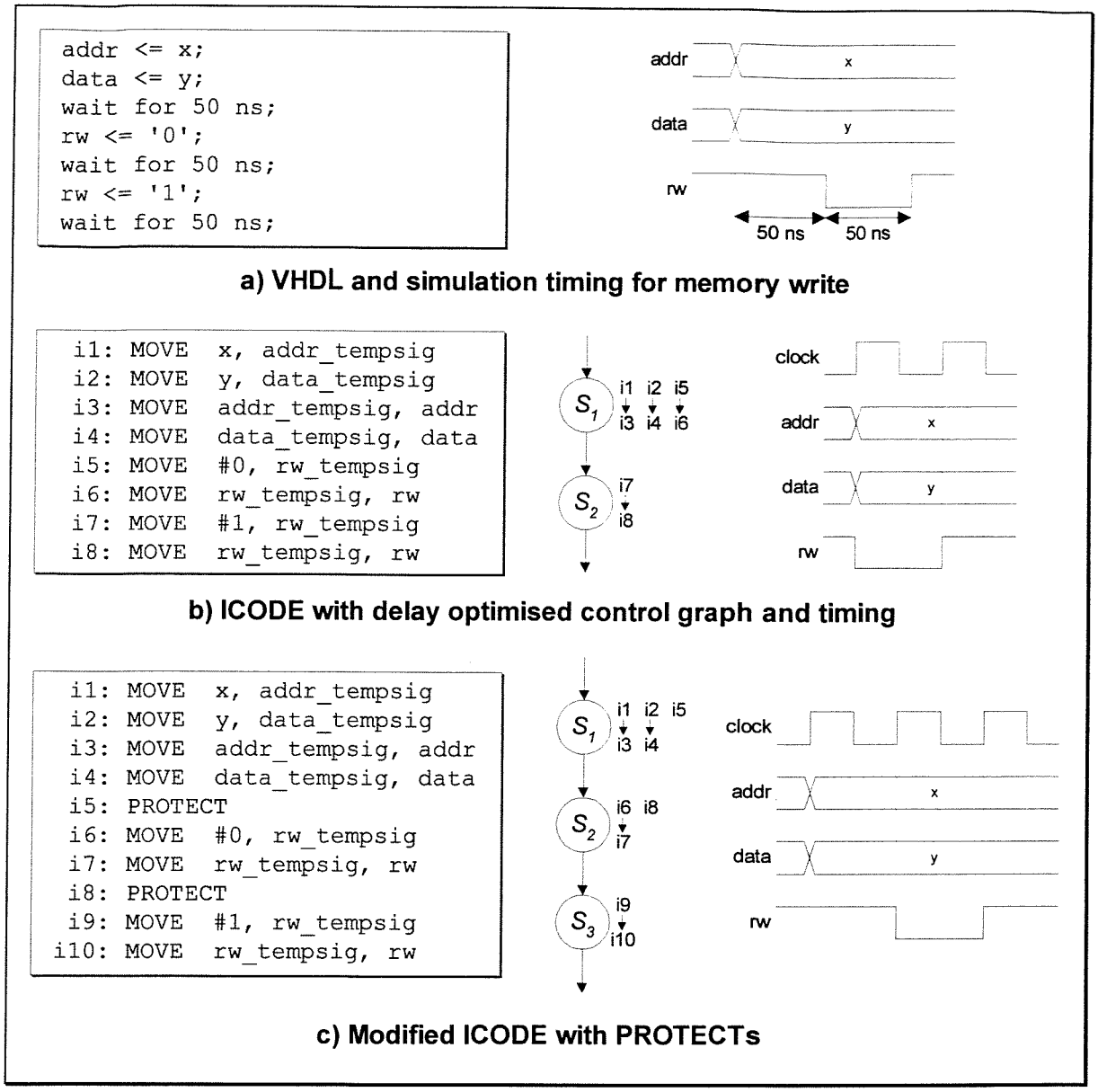


Figure 6.14 Local timing of outputs using waits

Considering each of the above situations when developing new designs should ensure that the synthesised implementation performs as required. In a few cases however, it may be necessary to manually add the occasional PROTECT instruction in order to enforce a particular schedule and so as a final measure, a special "*protect*" VHDL procedure has been added to VHDL2IC, which, when encountered, places a PROTECT instruction at the same point in the ICODE file; manual editing is therefore unnecessary. More details regarding the synthesis of some real designs (including the exact control of cycle-by-cycle timing) may be found in Chapter 7 which discusses the development of a complete system from inception to physical hardware construction.

6.3 Macro Modules

Section 5.2.2 introduces the concepts of *macro operators* and *macro ports*, both of which are implemented via the *macro module* library. Macro modules provide a mechanism for directly instantiating an expanded module from within a VHDL description, by the simple inclusion of a call to a function or procedure. These VHDL subprograms form a high-level function library defined in special *macro_ops* and *macro_ports* packages built into the VHDL2IC library file **packages.vhd**. They are similar in concept to standard libraries, where sets of pre-defined subroutines are provided for inclusion in the behavioural source code, however instead of synthesising the procedure body along with the main design, macro modules are pre-synthesised in the form of expanded module templates. This has a number of advantages over the traditional approach:

- The macro module function may be initially treated as single functional unit during a first pass optimisation run. It can then be further optimised after expansion in a similar manner to post-splitting described in the previous chapter.

- The expanded template defining the macro module may be modified at the ICODE and structural level to implement a more complex and efficient design than permitted by VHDL alone. Inclusion of PROTECT instructions and pre-synthesising the template also enables the final local schedule to be exactly controlled state-by-state. This forms the basis for *macro ports*, which allow strict localised control over the timing of inputs and outputs, useful for complex I/O protocols such as memory access.

- Multiple implementations may be developed for a single macro, which can then be selected during synthesis to investigate the effects of different module topologies.

- Alternate sets of templates can be created for different I/O protocols. For example, by providing one set of macro ports for fast static rams, and another for dynamic memory, different system configurations can be created without altering the design, simply by exchanging the expanded templates used.

6.3.1 Implementation

As mentioned above, macro modules are implemented in the VHDL2IC compiler as pre-defined packages of functions and procedures. These include simulation models for use when developing and testing the behavioural source, which are however ignored during compilation. The compiler does not implement the subprogram call with the usual MODULEAP instruction, but instead translates it into an equivalent ICODE instruction with the same name and parameter list. Figure 6.15 shows example fragments from the various VHDL source and library files for a typical memory access macro port, together with the generated ICODE instructions. Note how the procedure parameters intended to connect to entity ports (which interface to the external memory device) are all declared as signals so that the procedure can directly update the ports themselves, and not just their shadow registers.

Because macro module names are only defined in the library package, new macros may be added simply through their inclusion in the VHDL2IC library file (**packages.vhd**). A side effect of this feature is that the older style binary ICODE file format (**.ic** files) can no longer be used, as it requires all ICODE instructions to be hard-wired into the compiler itself.

Following compilation, the extended ICODE file is loaded into MOODS for synthesis. As this is read, each instruction is matched against an entry in the ICODE instruction database (see section 5.3.1), which defines I/O parameters and the data path function type required for implementation. The module library is then queried for an appropriate data path module to support this function. Thus, each macro module must have an associated entry in the instruction database, together with a suitable module in the module library with which MOODS creates an initial design representation. Whereas most library modules represent a physical component in the target low-level synthesis system, these macro entries are simply dummies, used as place markers by MOODS until the actual macro module expansion occurs. MOODS differentiates between the two types by reserving a block of special function codes for the dummy entries. Referring back to section 5.3.1, the module library encapsulates many separate technology libraries in a single entity, therefore all the macro module dummies may be stored in a special library to be used in conjunction with any desired set of technology-dependent low-level modules (see Chapter 5).

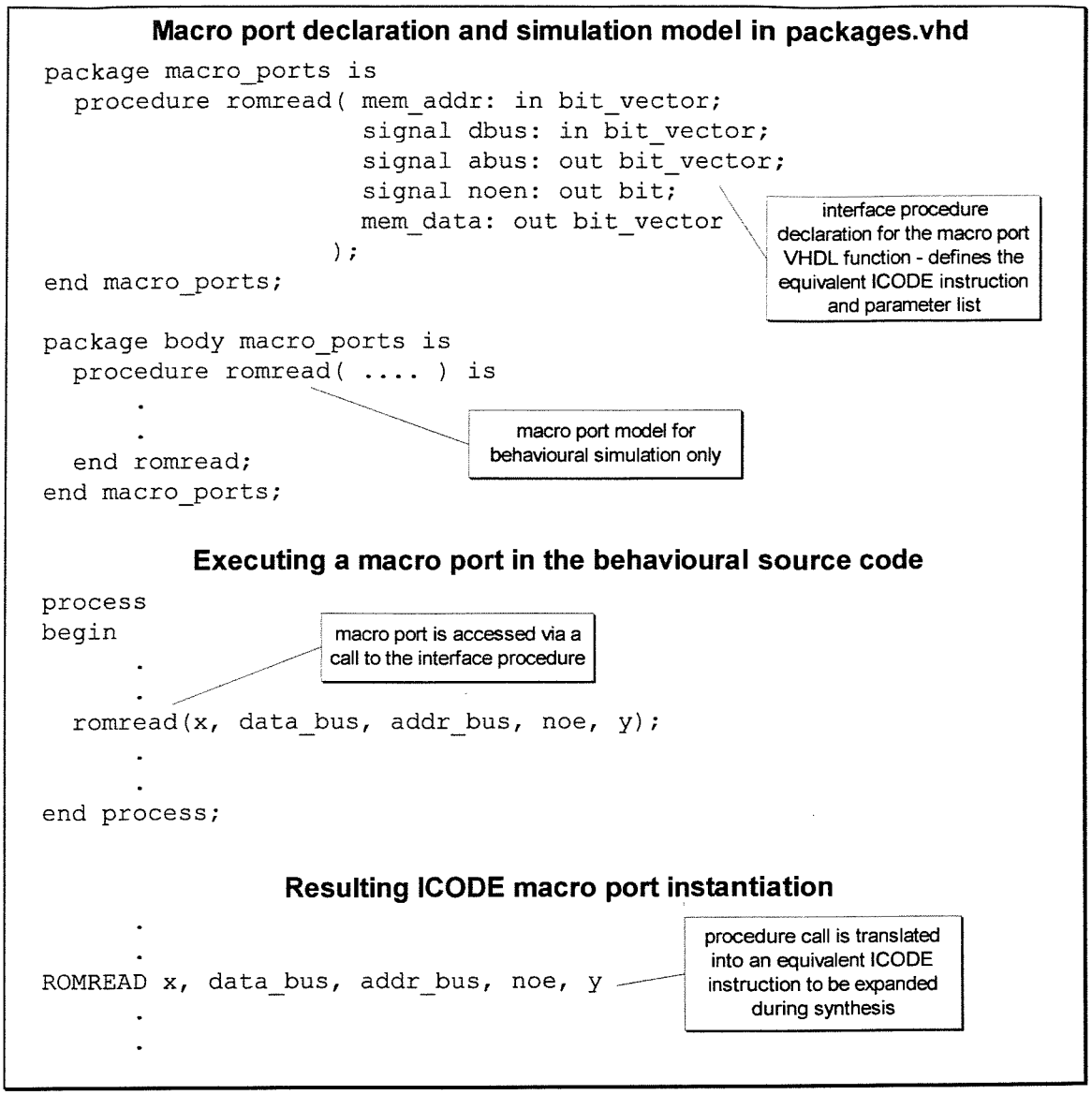


Figure 6.15 Declaring and accessing macro ports in VHDL

Once the initial control and data paths have been created, the design can be optimised in the usual way. At some stage during this process, the macro modules themselves must be expanded by replacing the dummy place-holder modules with their proper expanded implementations. There are four possible ways in which this can be achieved:

1. Expand particular function types using the *aob* command with the appropriate macro module function, following the same methods used for module splitting in Chapter 5.

2. Automatically expand all macro modules in one go using the *aom* command.

3. Expand individual dummy modules by manually applying the *split module* transformation.

4. Allow MOODS to automatically expand all macro modules as a final step prior to outputting the structural netlist

Options 1 to 3 may be performed at any time during the optimisation process, and are mainly intended for expanding macro operators, which may need to be further optimised to achieve the intended structure. This is identical to the module expansion process investigated in the previous chapter, and the same principles regarding how and when to expand apply. Option 4 is designed for macro port expansion where the scheduling of operations within the expanded template is to be preserved. Expanding these modules after optimisation is complete ensures that no alterations are made to the macro port timing, thus guaranteeing their correct operation.

The choice of which option to use depends largely on the type of macro modules being expanded, but there are one or two other factors which should be considered when dealing specifically with macro ports, discussed in the next section. Either way, once MOODS has completed, the resulting structural netlist will be devoid of any references to the original functions and dummy placeholder modules, being entirely composed of components from the low-level technology libraries present.

6.3.2 Macro Port Issues

Unlike macro operators, which generally require optimisation of their sub-structures within the top-level design, macro ports must be unchanged by optimisation, at least as regards scheduling, hence the use of the post-optimisation expansion step. Macro ports, therefore, must be implemented as efficiently as possible within the expanded template, as they cannot rely on any additional MOODS optimisation. In general, most macro ports just involve the transfer of data between inputs and outputs, and do not make use of any significant functional units, thus data path optimisation is not normally of overriding concern. However, the same cannot be said of scheduling, which must be pre-defined within the template to facilitate the required I/O timing.

The easiest way to create a macro port it to first design a behavioural description at the VHDL, or more likely, the ICODE level. This is then manipulated within MOODS to achieve the desired schedule before being saved as an expanded template. To ensure the

correct schedule, the module may be manually optimised state-by-state, however, as will be shown it is preferable to utilise PROTECT instructions partitioning the ICODE into scheduled instruction groups.

SRAM write macro port ICODE source

```
      PROGRAM      sramwrite_a14_d8   mem_addr, mem_data, \
                                      data_bus, addr_bus, \
                                      rnw, noe, fp_noe

      INPORT       mem_addr    [0:13]   // address to write to
      INPORT       mem_data    [0:7]    // data to write
      OUTPORT      data_bus    [0:7]    // ram data_out bus
      OUTPORT      addr_bus    [0:13]   // ram address bus
      OUTPORT      rnw         [1:1]    // ram read/not write
      OUTPORT      noe         [1:1]    // ram output enable
      OUTPORT      fp_noe      [1:1]    // FPGA output enable

 i1:  MOVE         mem_addr,  addr_bus  // set address bus value
 i2:  MOVE         mem_data,  data_bus  // set data bus value
 i3:  MOVE         #1,  noe             // make sure ram is hi_z
 i4:  PROTECT                           // enforce setup time

 i5:  MOVE         #0,  fp_noe          // enable FPGA bus
 i6:  MOVE         #0,  rnw             // ready for write to ram
 i7:  PROTECT                           // enforce 2 cycle
 i8:  PROTECT                           //   write low pulse

 i9:  MOVE         #1,  rnw             // latch data into ram
i10:  PROTECT                           // enforce hold time

i11:  MOVE         #1,  fp_noe          // disable FPGA bus output

      ENDMODULE    sramwrite_a14_d8
```

Macro port optimised schedule and output timing

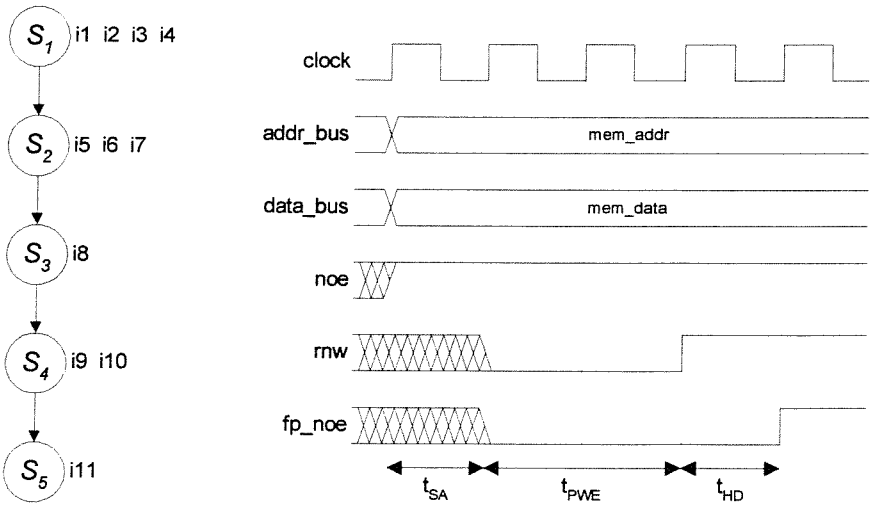


Figure 6.16 Macro port creation

Figure 6.16 shows a typical ICODE description for a *static RAM write* macro port, together with the desired instruction schedule and output timing. The macro takes seven parameters: *mem_addr* specifies the address to which the value of *mem_data* should be written; *data_bus* and *addr_bus* should be mapped onto to the entity ports connecting to the data and address buses of the external memory; *rnw* and *noe* connect through the entity ports to the memory write enable, and output enable control lines; and *fp_noe* controls the direction of the target device (ie. the physical device implementing the synthesised design) tri-state data bus, ie. the top-level port connecting to *data_bus*. The timing of this particular SRAM device requires an *address to write start setup delay* (t_{SA}) of 1 cycle, a *write enable pulse width* (t_{PWE}) of 2 cycles, and a data hold from write end (t_{HD}) time of 1 cycle. In reality of course switching characteristics are quoted as absolute delays that are divided by some minimum allowed clock period to determine the macro port timing. Note that where a 2 cycle delay is required, two adjacent PROTECTs are used. The exact positioning of the PROTECT instructions means that delay optimising the ICODE produces the required schedule, which is then saved as an expanded template ready for inclusion in the VHDL2IC library.

Although in order to guarantee the required local schedule the macro ports should be expanded after optimisation, there are advantages to relaxing this rule a little, and performing a final optimisation pass in between expansion and netlisting. The reason for this approach is illustrated in Figure 6.17 which demonstrates the expansion of a simple *romread* macro port with and without the additional optimisation stage. Figure 6.17a shows the macro port ICODE and the optimised expanded template schedule for reading from a ROM with an access time of less than 1 cycle. Figure 6.17b is an ICODE fragment compiled from a VHDL description featuring a macro port call, and is shown together with its optimised schedule before the macro is expanded. Remember that at this stage the *romread* is treated as a normal data path functional unit, and so may be chained with other instructions in a single control state. Note that if it was instead implemented as an ICODE module call, this would not be possible as these require special *call* control nodes (see Chapter 4).

Expanding the macro port without any post-expansion optimisation results in the schedule of Figure 6.17c. Here the port timing has been maintained, but the expansion process has

forced *i1* and *i3* into additional states resulting in a total delay of 4 clock cycles. Optimising this produces the compressed schedule of Figure 6.17d in which *i1* and *i3* have been merged into the expanded module structure, reducing the delay to 2 cycles. The actual timing of the *romread* however, has not been altered as the PROTECT instruction ensures that the two macro port states do not merge.

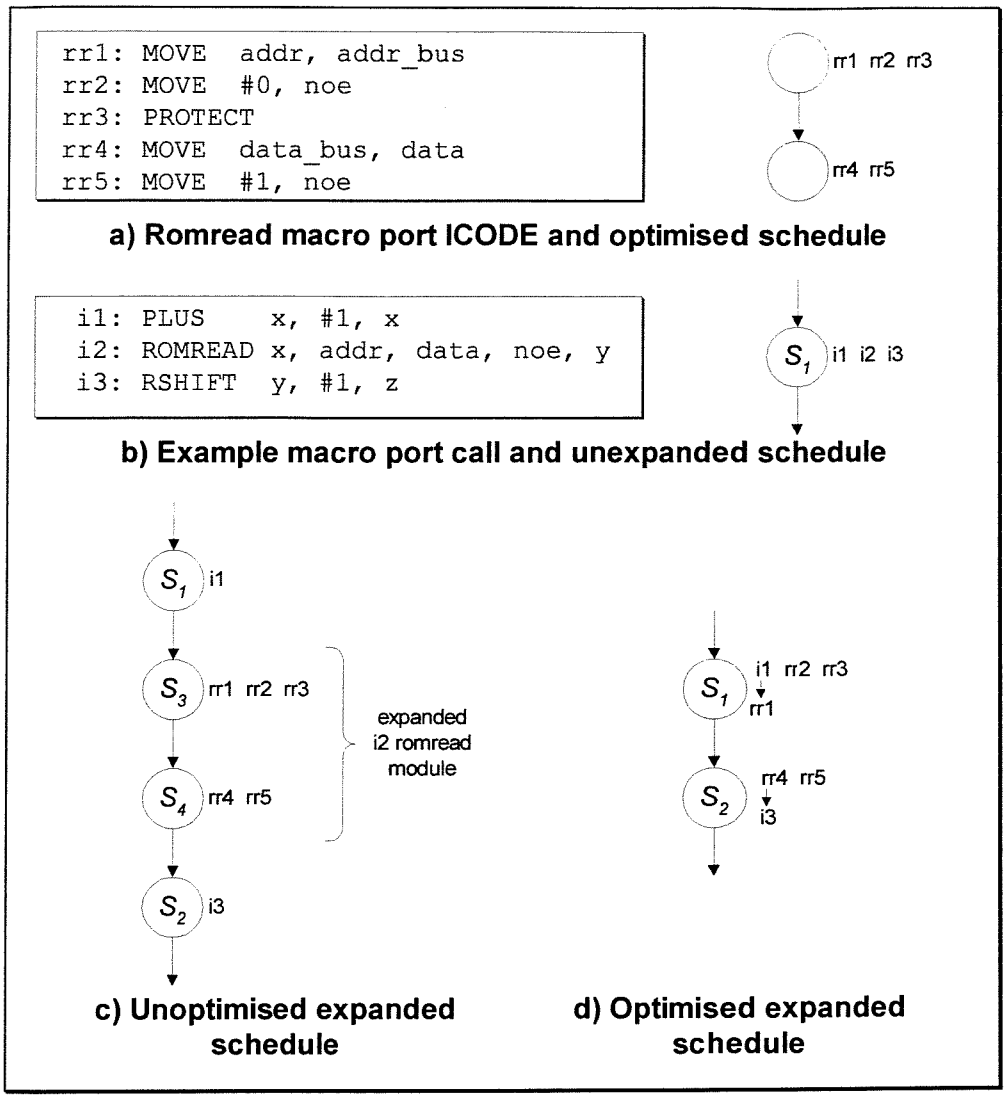


Figure 6.17 Macro port expansion and optimisation

Including this extra optimisation step has two main advantages:

1. The macro port control structure is seamlessly merged into the main control graph, resulting in an efficient schedule while maintaining the desired local timing behaviour. The use of PROTECT instructions to fix the port schedule means that no amount of additional optimisation will violate the timing constraints.

2. The knowledge MOODS has of the constituent instructions of the expanded modules allows it to perform optimisations within the expanded structure. For example, in Figure 6.17d *rr4* is chained to *i3*, thus possibly eliminating the need for the *y* register.

Best results therefore are generally achieved by expanding macro ports just prior to a final optimisation pass. Care must be taken, however, to ensure that all the expanded templates use PROTECTs to determine the macro port schedule, and are not simply the product of manual manipulation at the template optimisation stage. Further discussion concerning the use of macro operators and ports within the confines of an FPGA design environment may be found in Chapter 7.

6.4 Summary

The enhancements to the MOODS VHDL compiler, VHDL2IC, described in this chapter, enable a much wider range of designs to be synthesised, exploiting the full power of VHDL concurrent processes. Coupled with a more accurate implementation of the VHDL simulation cycle, this allows the development of complex designs with a substantial degree of simulation/synthesis correspondence. The system attempts to strike a balance between the desire to synthesise an exact implementation of the simulated timing, and the extra area and delay costs involved. As a final resort, VHDL2IC allows the most expensive features to be bypassed, in order to gain that extra bit of area and performance.

Inclusion of source-level support for macro operators enables the development of a high-level library of complex functions, directly accessible within a behavioural VHDL description. These are both technology and source language independent, and are integrated into the design flow allowing them to be optimised within the structure of the top-level design. Finally, macro ports provide a mechanism for encapsulating the exact timing of complex interface protocols in pre-defined packages accessed with a single VHDL subprogram call.

Many of the features described in this chapter are heavily used in the development of the audio-band spectrum analyser system detailed in Chapter 7 which also investigates the practicalities of synthesising behavioural designs within the limits imposed by relatively small FPGA devices.

Chapter 7

Practical Synthesis using FPGAs

This chapter describes the design, synthesis and final physical implementation of a real-time audio band spectrum analyser. Throughout its life MOODS has never, until now, synthesised a complete system down to the hardware level due to the high cost of small-scale integrated circuit production, relying instead on simulation of the structural output to verify its operation. The continual decrease in the price of sizeable FPGA devices, however, has provided a quick and cheap alternative route down to silicon. The purpose of this project therefore, is to demonstrate the ability of MOODS to synthesise complete systems and at the same time, provide a test case for investigating the problems encountered when attempting complex designs at the behavioural level. It also affords a perfect opportunity to utilise some of the new features described in earlier chapters and to develop ideas for further enhancements.

The sections below detail the workings of the spectrum analyser with a detailed discussion of the design of the system as a whole, and its major sub-components. These serve to illustrate the main problems and solutions in the development of a typical system covering such topics as the synthesis of exact timing behaviour, designing within the confines of FPGA devices, and interfacing to the outside world. The chapter concludes with an examination of the synthesis process in relation to the design requirements, and looks at the performance of the final hardware. Full VHDL source listings and other associated information may be found in Appendix C.

7.1 System Specification and Design

The goal of the project described in this chapter is to develop a device capable of showing on a small screen display, the discrete Fourier transform of an audio signal in real time. The system unit, which can be seen in Figure 7.2, and the photographs in section 7.5, is

based around a 320×256 pixel electroluminescent display (detailed in Appendix C) outputting a continually updated plot of magnitude versus frequency for an audio input signal. This device is mounted in a single unit that also provides a number of controls for altering the display setup:

- *Level adjust* controls the gain of the input signal amplification stage and is used in conjunction with an LED level meter to set the display for maximum displacement without distortion.

- *Pan left* and *pan right* alter the frequencies displayed on the screen, which can cover a maximum of 112 points, forming a window on the full 512 frequency results calculated. The pan controls move this window along the frequency range in 8 point steps allowing all 512 points to be examined.

- *Zoom* is used in conjunction with the pan controls to magnify/de-magnify the display. This alters the width of the frequency bars between 1 and 15 pixels, enabling easier viewing of particular sections of the spectrum.

- *Hold* freezes/releases the current display image on alternate presses, allowing snapshots of a varying input to be taken. While in hold mode, the pan and zoom controls function normally for detailed examination of the whole frequency range.

The inputs provided to the unit are a line-level audio signal and a 15 volt DC power supply, together with a pass-through audio output for monitoring the displayed signal on external speakers or headphones.

7.1.1 System Overview

The system is based around the popular Cooley-Tukey fast Fourier transform (FFT) algorithm [112, 113] applied to a block of 1024 discrete samples, which are taken from a 12-bit analogue to digital converter (ADC) connected to the audio input. Its operation splits into three general stages outlined in Figure 7.1:

1. The input signal is first sampled 1024 times and the data stored in an FFT scratch memory.

2. These samples are then processed by the FFT algorithm to obtain a frequency spectrum split into 512 points, replacing the original samples in memory.

3. Finally, the FFT results are plotted as a frequency spectrum the video display.

These operations repeat, updating the display image at a rate of around ten frames per second, thus obtaining a real-time frequency spectrum.

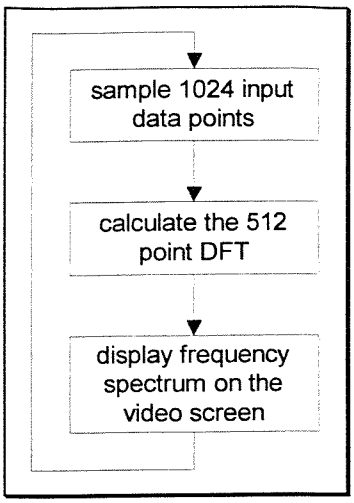
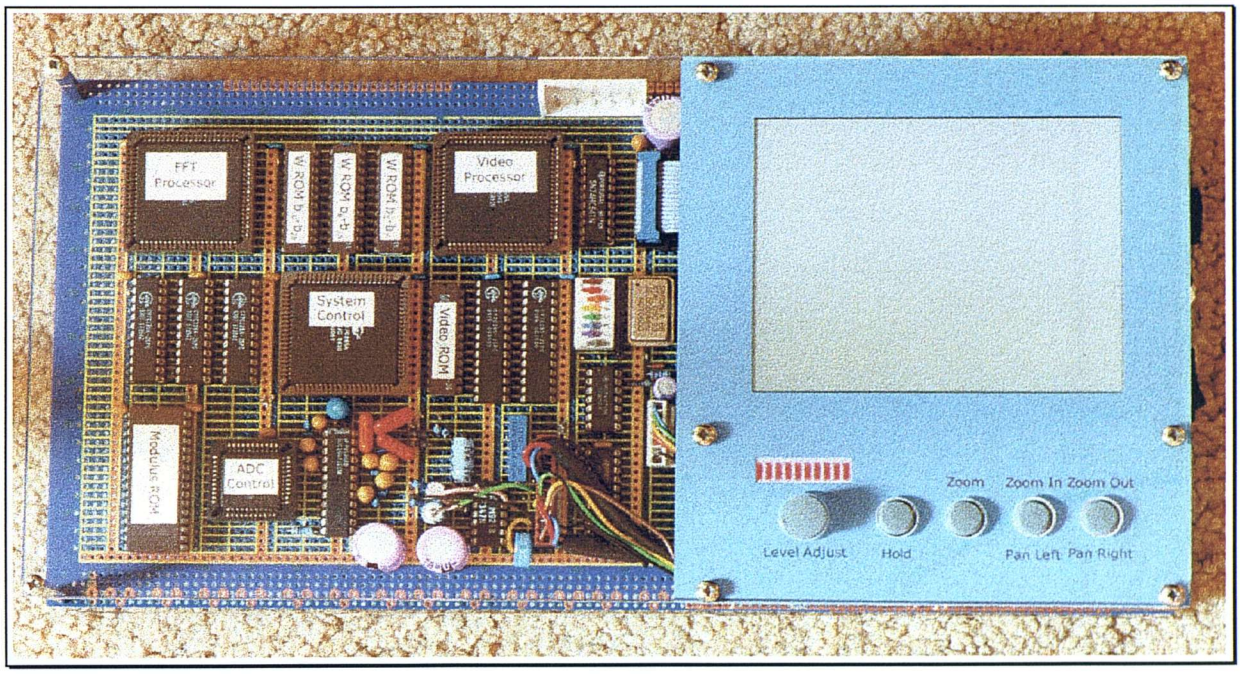
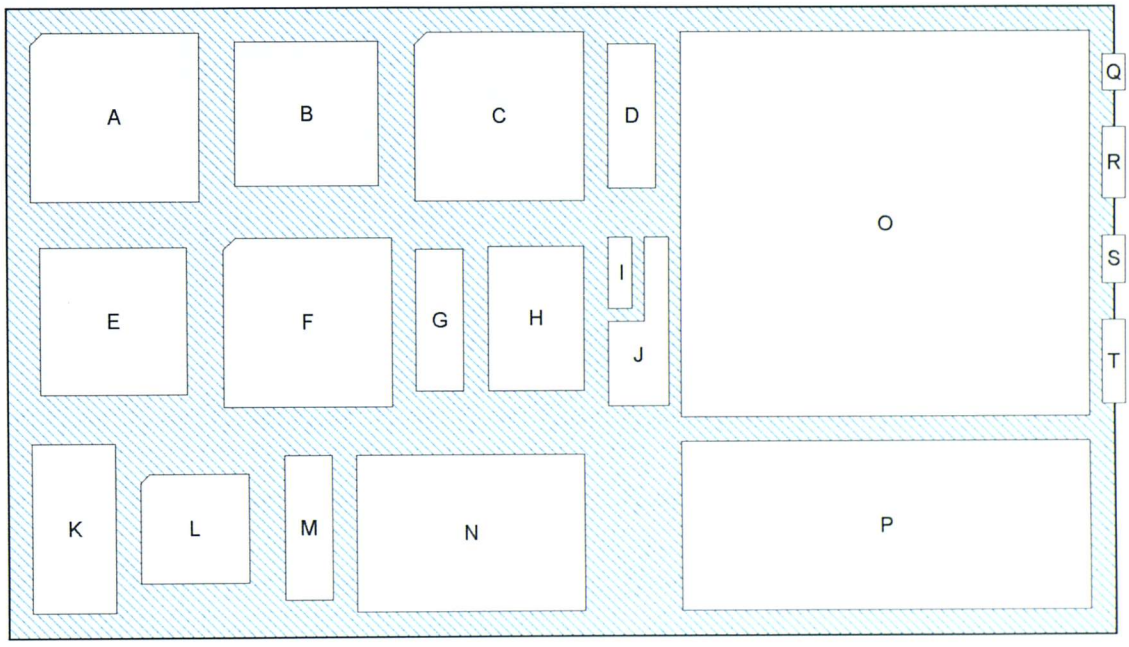


Figure 7.1 Core system flow chart

Figure 7.2 shows a photograph of the final hardware unit, identifying its key components, and their positioning on the system board. The bulk of the design is implemented in three FPGAs synthesised by MOODS, plus a variety of EPROM and RAM memories. The system block diagram of Figure 7.3 maps almost directly (apart from the ADC controller and analogue input) onto the physical unit, and shows how the various components interconnect. This partitioning is largely governed by the limited size of the FPGAs, all of which are 93% to 100% utilised in the final implementation, and the number of I/O pins available on each device (68 on the CY7C385A used here). At the heart of the design is a central controller (**FFTCont**), which implements the three core stages of Figure 7.1, directing the flow of data between the various system components though several shared buses. It is aided by the FFT arithmetic unit (**FFTProc**) performing the necessary complex number manipulations central to the FFT algorithm, and the video processor (**VidProc**), which looks after all the low-level display requirements. The on-board memories comprise: RAM and ROM for the video frame buffer and pixel map resources (character bitmaps etc.); W ROM containing constant data for the FFT calculation, plus video data for the static portion of the display (axes, title etc.); and a Modulus ROM, which contains a lookup table for calculating the modulus of the FFT result complex numbers, scaled to the required screen resolution. Ancillary components include buttons on the control panel, the ADC and ADC controller chip (itself synthesised by MOODS), a video blanking preset (used for setting up video refresh timing), and the video screen itself.





Key

A. FFT arithmetic unit - FFTProc
B. W ROM / Video display data
C. Video processor - VidProc
D. Video signal output buffers
E. FFT data memory
F. Central system controller - FFTCont
G. Video resource ROM
H. Video frame buffer memory
I. Blanking and ADC clock setup
J. System clock and power on reset
K. Complex number modulus lookup table ROM
L. Analogue to digital converter controller - ADCCont
M. Analogue to digital converter - ADC16071
N. Analogue input signal buffer/amplifier
O. Electrolumiscent display - EL320.256-F
P. Control panel
Q. Amplified signal out
R. Analogue signal in
S. 15v @ 1A DC power supply
T. Power on/off switch

Figure 7.2 Spectrum analyser hardware unit and key components

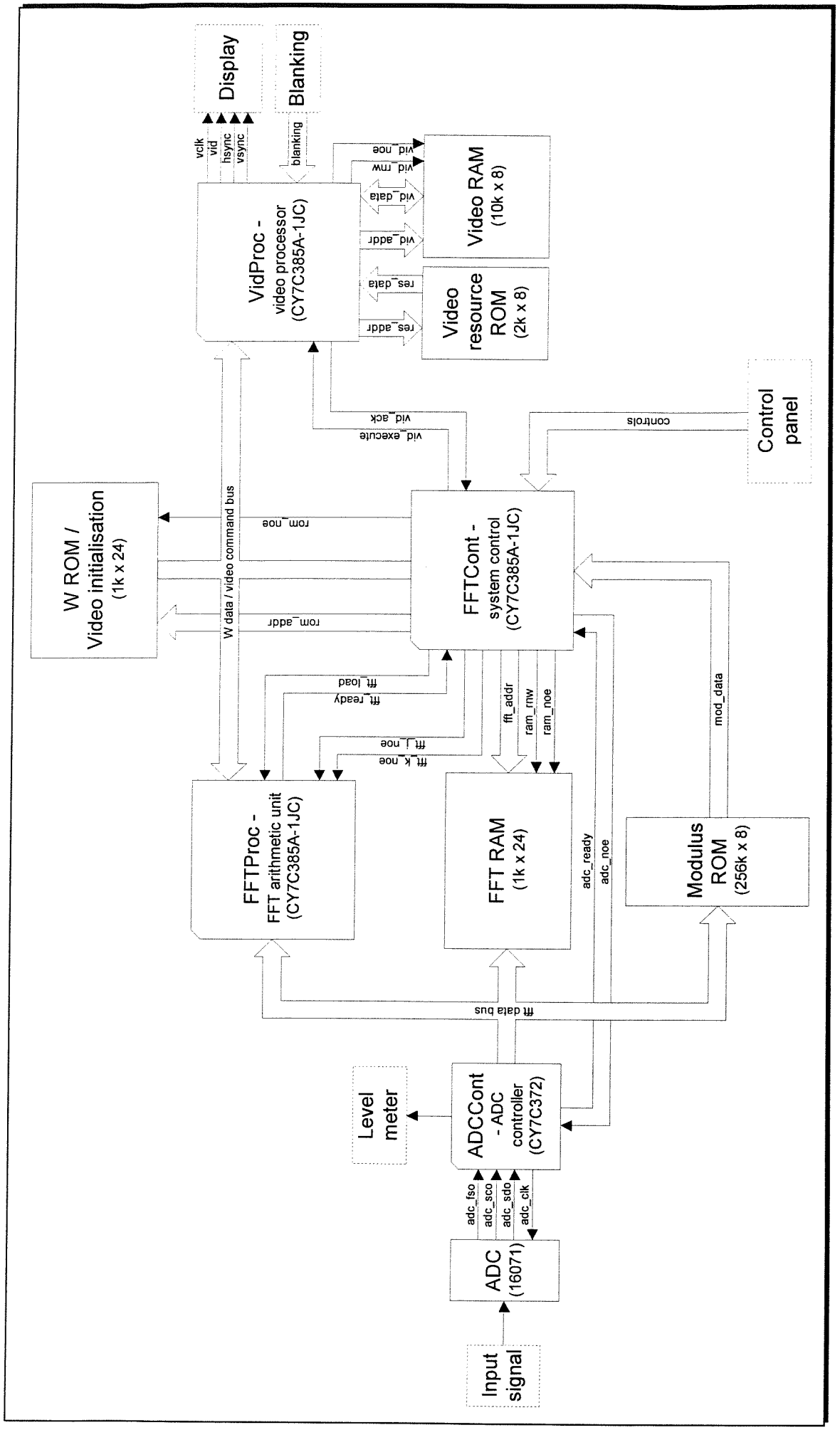


Figure 7.3 Spectrum analyser system block diagram

7.1.2 Design Methodology

As this is the first ever project employing MOODS to produce hardware, there was a substantial amount of preparatory work necessary before embarking on the main design. The creation of technology libraries targeted to the Warp [93] low-level FPGA synthesis tool was a first priority, and included the generation of a full set of library modules to interface between MOODS structural output and the Warp libraries. Other important preliminaries included fixing a number of problems with MOODS itself, particularly in the VHDL outputting stage, and some initial soundings on the size and type of designs possible, within the constraints of the available target devices.

Many of these areas were tackled during the initial design of a video test device primarily intended to investigate the control signal timing required for the display screen. It also provided a test bed upon which to develop techniques for controlling the exact optimised schedule obtained from MOODS, by using a single complex programmable logic device (CPLD) mounted on a small test board connecting to the display. This enabled the physical repercussions of design and optimisation strategies to be investigated with ease by re-programming the device with various different implementations. In many respects, it was also a confidence building step, showing for the first time that MOODS could actually synthesise a fully functioning piece of hardware within the limits imposed by programmable devices. The development of libraries and bug-fixing, however, continued more-or-less throughout the early stages of the project, and included the addition of a number of extra facilities such as macro operators and the ICODE PROTECT instruction.

Once the initial explorations were complete, the libraries developed thus far were augmented to encompass the full set of module functions supported by Warp. This involved the creation of VHDL wrappers to implement the MOODS modules in terms of the LPM library provided by Warp (see below), together with functionally equivalent components for use during simulation of the structural output. The associated library description files (see section 5.3.1) were also created, containing accurate size and delay models for each module derived from low-level synthesis of a set of test designs. Based on these, the final characterisation figures predicted by MOODS show a high degree of correspondence to the actual FPGA results, as discussed in section 7.5.

One point of particular interest is the use within Warp of the "Library of Parameterised Modules" (LPM) standard [114] for its low-level cell library. LPM is a relatively recent initiative designed to harmonise the library interfaces provided by many different logic synthesis tools. The library support developed for this project therefore, should be directly transferable to other target systems without any further modifications, apart from adjustments to the area and delay models.

Before discussing the main steps in the development of the spectrum analyser, a brief overview is given of a general design flow suitable as a starting point for most behavioural designs. Figure 7.4 shows the six main steps in the development of a system from initial conception through to final hardware construction:

1. The first step in any design process is the overall system specification and design, identifying the key elements involved and mapping them onto various high-level blocks (memories, synthesised FPGAs etc.). In most cases, this structure will be refined as development progresses and the constraints of the target devices and algorithms are determined.

2. Initial software emulation of any important algorithms and synthesised sub-blocks is an essential step in all but the simplest of designs. A software development environment allows much more scope for investigating and testing different approaches than behavioural simulation alone, being both faster and more flexible. The designer is able to evaluate the core algorithms, and examine factors such as the numeric range of variables, trade-offs between accuracy and variable size and the typical performance of the final implementation. It also provides a mechanism for obtaining test data for use as input, and to compare against the output from the later simulation steps.

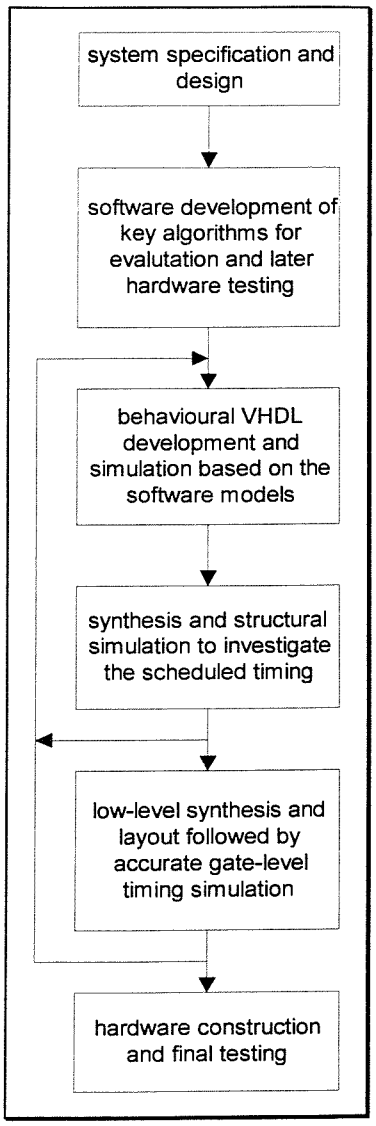


Figure 7.4 General design flow

3. The beauty of behavioural synthesis is that the software emulation code can usually be directly translated into a VHDL equivalent with

minimum effort, and synthesised almost immediately to produce a structural implementation. Most of the additional VHDL design effort thereafter goes into adding internal and external communication (*signals* and *waits*), and enhancing the architecture with features such as concurrent processes. Simulation is used to compare results with the software, and develop the general timing and interface requirements. Once satisfactory descriptions have been obtained, the whole system can be brought together for a full simulation to examine the interoperability of the various sub-blocks, and iron out any problems encountered.

4. Once the VHDL code works as required, behavioural synthesis converts the descriptions into an optimised structural implementation, based on low-level modules. This output may then simulated to obtain the exact scheduled timing behaviour and to determine whether the design still operates within the required parameters. Any shortfalls can be addressed either through different synthesis parameters, or by modifying the behavioural description for the desired size and timing. Again, the entire system should be simulated at this level to check for timing problems, particularly relating to communication and edge detection.

5. The structural description simulated above forms the input to the low-level logic synthesis and hardware layout tools. These will provide true area and delay figures for the design, which may require a certain amount of modification at either of the previous two levels to obtain the final hardware implementation. The layout tool also provides a gate-level VHDL structural description, together with accurate timing models, which may be used to perform a final system simulation. This tends to identify any extreme timing problems and is particularly useful for testing the power-on reset phase where gate-level modelling is crucial.

6. The final and most straightforward step is to create the actual hardware devices and construct the system board. With care and patience, few problems should be encountered, as the extensive system-level simulations above will have hopefully identified most design errors. Construction related problems of course, are not covered by this, so the usual issues (eg. noise, buffering, de-bouncing) must still be addressed.

It must be stressed that the design flow described above only represents a general framework within which to operate. Many of the later steps may require changes to be made to certain elements of the higher-level design in order to re-distribute parts of the

system among the various devices, once final FPGA layout figures are available. The spectrum analyser project in particular required a certain amount of system and behavioural level modification, as the capabilities of MOODS and the capacity of the FPGAs became more apparent. Looking at the design flow of this in more detail, the following stages can be identified:

1. The system was initially partitioned into FFT calculation and video output devices plus the various ROM and RAM resources. As the design progressed, however, it became clear that the size and I/O resources available required the FFT to occupy two devices, hence the system organisation shown in the block diagram of Figure 7.3.

2. *Software development* of the core FFT algorithm proceeded to some extent in parallel with the system design, as the data bit-widths and required memory resources were investigated. Two versions of the FFT were developed, the first being a basic implementation of the algorithm using standard floating-point arithmetic, which was then transformed into a fixed-point integer-based equivalent, making use of the same feature set available in VHDL. The software environment allowed the results of both implementations to be readily compared, and the investigation of a variety of fixed-point configurations and integer bit-widths. Both versions, together with a detailed description of the fast Fourier transform, may be found in Appendix C.

3. *Video processor development and synthesis*:

   3.1 Behavioural development of the video processor was based on the initial test design described earlier, augmented with a hardware rendering capability and video memory arbitration.

   3.2 MOODS synthesis, followed by structural simulation, was critical for obtaining the correct video signal timing, involving substantial modification of the behavioural description.

   3.3 The final logic synthesis stage also had a significant impact on the behaviour, as the original version was some 25% too large, requiring many major and minor alterations before the design would fit. Section 7.2 details the design and operation of the video processor.

4. *FFT and system control development and synthesis.* As has been already mentioned, the original system design partitioned the FFT and system control stages in a single device. At its heart, this was a straight VHDL implementation of the fixed-point FFT software algorithm (even down to the variable names), together with the sampling and results display stages. The first attempt at fitting this on an FPGA revealed that the design was both too large, and required too many I/O pins. Thus, the core arithmetic portion of the FFT algorithm was moved onto a separate slave device and the design/synthesis loop reiterated. These two designs are detailed in section 7.3.

5. *Full system simulation*:

   5.1 A test jig was created to simulate the whole system implementing the block structure in Figure 7.3. This utilises specially designed RAM and ROM components with file I/O capabilities to allow the memory contents to be dumped and retrieved from files, and a complete emulation of the ADC with sample values also read from a file. The correctness of the entire system could therefore be determined by looking at the contents of the various memories and comparing the results to the software emulation. VHDL source for the test jig is contained in Listing C.8 in Appendix C.

   5.2 Since MOODS maintains the I/O port list of the original design in the optimised implementation, the same test jig could be used for both behavioural and structural system simulation simply by changing the underlying components. At this level a complete iteration of the main system loop required about half an hour simulation time[1], compared to a few minutes for the behaviour.

   5.3 Finally, the gate-level descriptions generated by the FPGA layout tool were inserted into the test jig, and the same simulation performed as in the previous two stages. In this case, the time taken for a single system iteration was around two days, making this level of simulation a one-off final confirmation of correctness. In the event, simulation of the MOODS output was generally the most useful as it shows the exact cycle-by-cycle timing behaviour in a reasonable amount of time.

---

[1] Times quoted are taken from a simulation using the Cadence Leapfrog simulator on a Sun Sparcstation 20.

6. The final stage was of course construction and testing of the hardware, which was simply a matter of implementing the interconnection structure in Figure 7.3 on a wire-wrap circuit board. Full system simulation at all levels ensured that the hardware worked first time, the only difficulties arising due to noise on the ADC reference input, and some clock and reset signal drive issues. All of these were tackled with the addition of extra bypass capacitors and buffers.

The following sections describe the three core system components in more detail, illustrating the approaches taken to typical problems encountered when using behavioural synthesis, and MOODS in particular.

7.2 The Video Processor – VidProc

The video processor is, in many ways, the most complex single component of the whole system. It is intended as a general-purpose device, providing a programmable pixel-based display capability, which interfaces directly to a raster-based video display. The design has the following general features:

- All video drive signals are generated internally, only requiring a set of output buffers before connecting to the display. The signal timing is exactly controlled throughout synthesis, and may be adjusted for a range of vertical refresh rates and system clock speeds.

- The display image is stored in an external frame buffer memory (video RAM) with the processor arbitrating between read access, when generating the video signals, and write access to create the displayed image.

- A number of hardware-based drawing commands are provided for building up an image in the frame buffer. These include individual pixel-plotting, horizontal and vertical line drawing, and horizontal and vertical character plotting based on an external character bitmap ROM. All operations interact with the current display image according to one of four plot modes: FILL, SET, CLEAR or XOR pixels.

Appendix C contains a complete listing of the VHDL source for the video processor (Listing C.3), plus a full device pin out and I/O pin description. Within the design, the features listed above fall naturally into three concurrent VHDL processes:

1. *Raster* (source lines 189-307) retrieves data from the frame buffer and outputs it, together with the required synchronisation and clock signals, to the screen.

2. *Render* (lines 331-452) plots a point, line, or character onto the frame buffer according to the video command selected on the *command*, *x* and *y* input buses.

3. *Memory* (lines 81-180) arbitrates between *raster* and *render* memory accesses, ensuring the former process is never held up waiting for the next byte as this would affect the video timing and corrupt the image displayed.

Figure 7.5 is a block diagram of the internal **VidProc** architecture showing the communication between the three processes and the outside world. The external components required are 10k × 8 bits of frame buffer video memory, 2k × 8 bits of character bitmap resource ROM, and a 4-bit "blanking" constant which determines the length of the video horizontal blanking phase. The rest of this section describes the

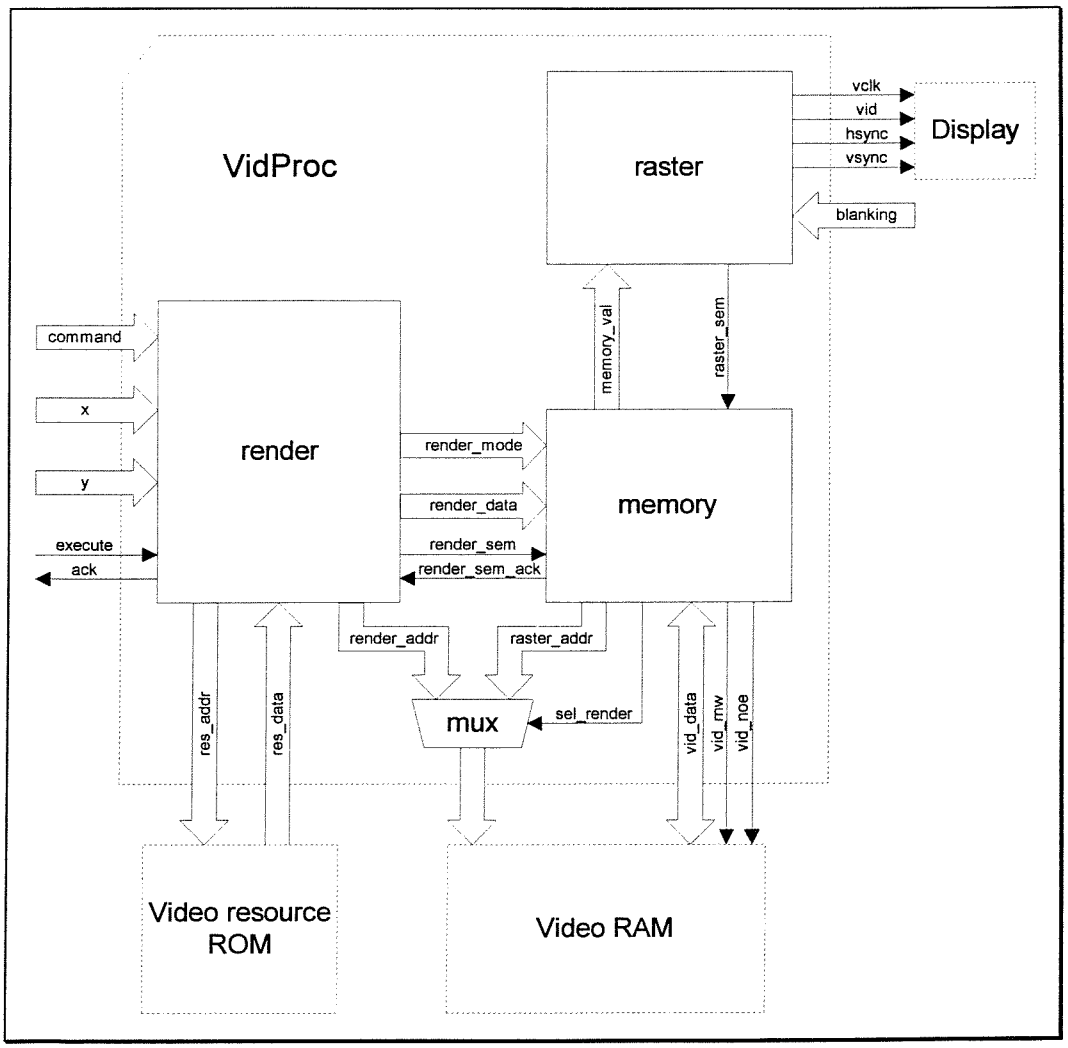


Figure 7.5 Video processor internal block structure

operation of each of these three processes in more detail. Note that throughout this chapter, process names and other elements of the source code are all *italicised*, while all file names and command-line switches are in **bold**.

7.2.1 The Memory Process

The central block in the video processor is the *memory* process, which provides *raster* and *render* processes with separate interfaces to the single-port video memory. This is used in preference to a dual-port memory with true independent access for each process, due to the substantial increase in the cost of both the memory device itself, and more importantly, the extra I/O pins required for two separate address and data buses. In fact, *memory* does more than just emulate a dual-port interface as it encapsulates all the memory access code, providing pipelined read for *raster* and a complete read/mask/write operation for *render*. This has three major advantages over the simpler approach:

1. The communication overhead between processes is reduced: *raster* can assume that the frame buffer has already been read when it needs to retrieve pixel data, and so does not have to request a particular address and then wait for a response; and *render* just passes an address and pixel mask, and does not need to read from memory, update and write back the result, which would require two communications with *memory*.

2. By encapsulating all memory I/O in a single common block, the overall complexity of the design is reduced, requiring less registers and control code.

3. Since all processes execute concurrently, memory accesses can be overlapped with the generation of video signals in *raster*. In effect, this provides free (as far as delay is concerned) memory reads, which, as will be shown later, is essential for correct video signal timing.

Before examining the operation of the *memory* process, first consider the frame buffer memory map in Figure 7.6. This shows how each pixel, specified as an x-y co-ordinate relative to the top left of the display, maps on to a memory location in the frame buffer. Since the video screen is only capable of monochrome operation, 1-bit per pixel is all that is required, thus 8 pixels are packed into a single byte requiring a total of $320 \times 256 \div 8$ bytes (10kB). Note that within a byte, the most significant bit represents the far left pixel, as shown in the diagram.

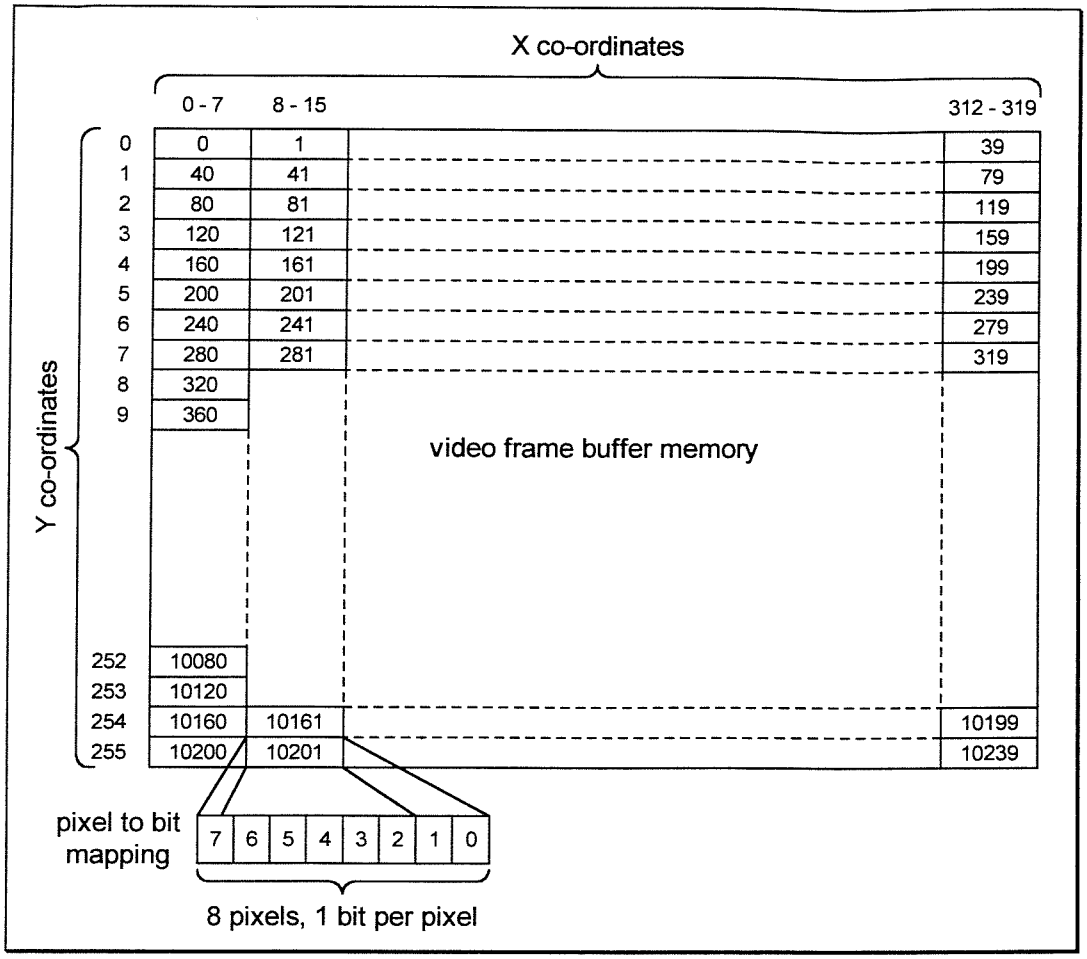


Figure 7.6 Frame buffer memory map

The flow chart of Figure 7.7 illustrates the operation of the *memory* process in more detail, where the line numbers correspond to Listing C.3 in Appendix C. The process is based around a main outer loop (*vram_loop*, lines 103-179), which reads sequential bytes from memory address *raster_addr* into the global *memory_val* signal (lines 112-113), forming the frame buffer read-ahead for the *raster* process. *Raster* simply treats *memory_val* as the next pixel byte for outputting to the screen and, once it has taken its own local copy, signals to *memory* (via the *raster_sem* handshake signal) that the value has been read, and the next location should be retrieved. Thus the two processes are kept in sync, with *memory* tracking the current frame buffer address and value, and *raster* generating the correct video drive signals. Note that when the end of an entire video frame is reached, *raster_addr* is automatically reset to the start of the next frame (line 174) without any further interaction between the two processes.

Within this main loop is another (*idle_loop*, lines 121-166) which iterates, once the read-ahead value has been loaded into *memory_val*, until *raster* signals that the next location

should be read. Under normal operation this loop accounts for every 14 out of 16 cycles, with 2 required for the *raster* read and address update operations. This idle time is used to provide the *render* process with access to the frame buffer. So, within *idle_loop* the *render_sem* global signal is monitored for a change in state (line 122), indicating a request by *render* for a memory update operation. If this occurs, *memory* takes the byte value from the *render_data* signal and masks it, according to the value of *render_mode* (another signal), with the data at memory location *render_addr*. Once completed, the operation is acknowledged via the *render_sem_ack* signal (line 162), allowing *render* to continue. Note that the actual setting of the video address bus to either *render_addr* or *raster_addr* is achieved by setting the *sel_render* output to 1 or 0 respectively. This relies on some modifications to the MOODS structural output discussed later in section 7.4.2.

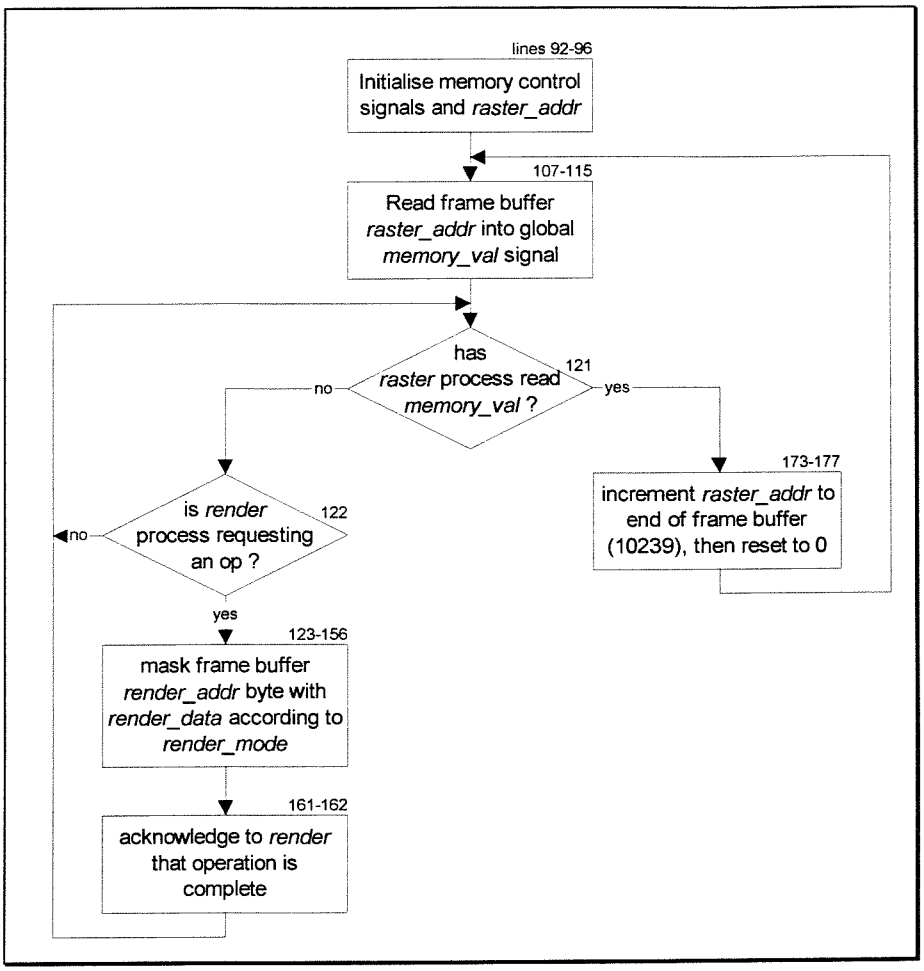


Figure 7.7 VidProc memory process flow chart

7.2.2 The Raster Process

The *raster* process (lines 189-307) takes 8-pixel byte values read by *memory*, and outputs them pixel-by-pixel, together with the appropriate video clock, and horizontal and vertical

sync signals, to create the frame buffer image on the video screen. *Raster* is one section of the system where complete specification of the exact final schedule is essential, as the display requires the pixel data stream (*vid* output) to be synchronised within strict limits to the video clock (*vclk*) and sync pulses (*hsync* and *vsync*). Any unevenness or poor synchronisation will result in a corrupted screen image. Information on how this timing is achieved in practice is provided in section 7.4.3.

Figure 7.8 details some simulation results showing the timing of the key external and internal signals during the output of the first display line (Figure 7.8a), and the horizontal and vertical sync pulses for a whole frame (Figure 7.8b). These results were obtained from a simulation of the synthesised structural description of the optimised video processor, and are an accurate representation of the final hardware performance. The video control signals (*vid, vclk, hsync, vsync*) illustrate the timing requirements of the display device and can be summarised thus:

- A single video frame is built up over 256 horizontal lines, each comprising 320 1-bit pixels.
- Pixels are read from the video processor *vid* output on the rising edge of *vclk*.
- *Hsync* indicates the start and end of a single line of pixels, with the last 320 *vclk* cycles before the *hsync* falling edge being displayed on the screen. *Hsync* must be low for at least 4 *vclk* cycles between each successive line of data.
- A rising edge on *vsync* determines the start of a new display frame, and must occur at least 3µs before the end of the *hsync* pulse for the first line, and at least 100µs after the final *hsync* pulse for the previous frame. The *vsync* period determines the frame refresh rate, which must be less than 72 Hz.

More details concerning the video display may be found in Appendix C which reproduces portions of the device data sheet.

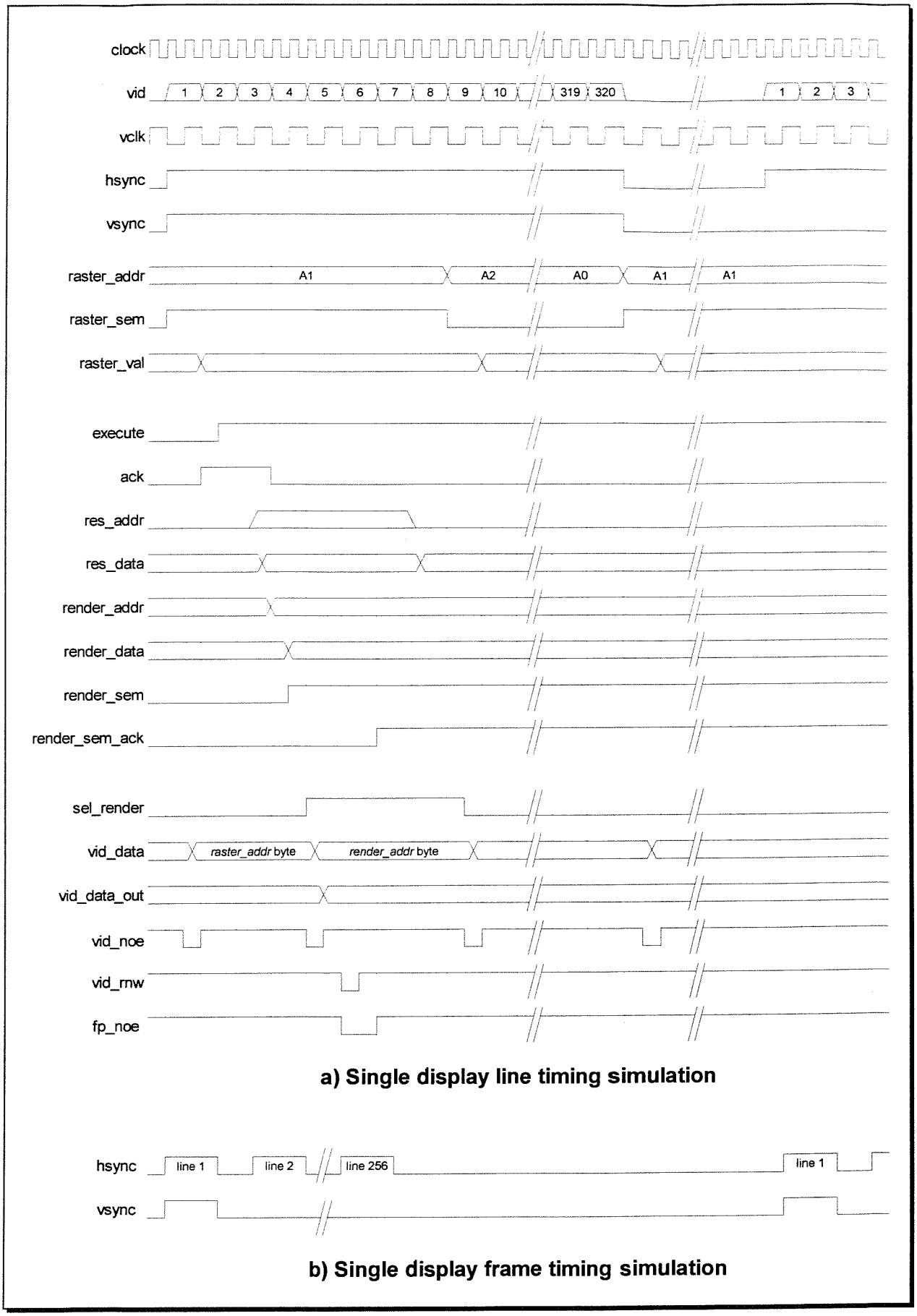


Figure 7.8 VidProc simulated signal timing

The operation of the *raster* process is depicted in the flow chart of Figure 7.9, which also shows the partitioning of operations into control states in the final optimised implementation. The process is based around four nested loops:

1. At the outer level, *refresh_loop* (lines 211-305) is an infinite loop which repeats with each display frame, setting *vsync* high at the very end of the loop (lines 302-303), ready for the start of the next frame. It is used in lieu of the implicit process loop to facilitate the initialisation of the *vsync* output on a power-on reset, as VHDL2IC does not allow port initialisation (or at least not at the time of this project).

2. Next, *vert_loop* (lines 216-296) controls the output of the 256 horizontal lines comprising a frame. In fact the loop counter (variable *line*) ranges from 0 to 259, generating an additional 4 blank lines with no data or *hsync* pulses, to fill out the 100µs required between *hsync* and *vsync* as shown in Figure 7.8b. *Vert_loop* is also responsible for generating the horizontal blanking region (where *hsync* goes low between lines) of at least 4 *vclk* cycles (lines 284-291). Because the *vsync* period is determined entirely by the length of an *hsync* cycle (ie. there is no vertical blanking other than the extra four blank lines), this blanking region effectively determines the vertical refresh rate. To allow for different refresh rates, and provide a degree of system clock speed independence (since different clocks require a different number of blanking cycles to maintain

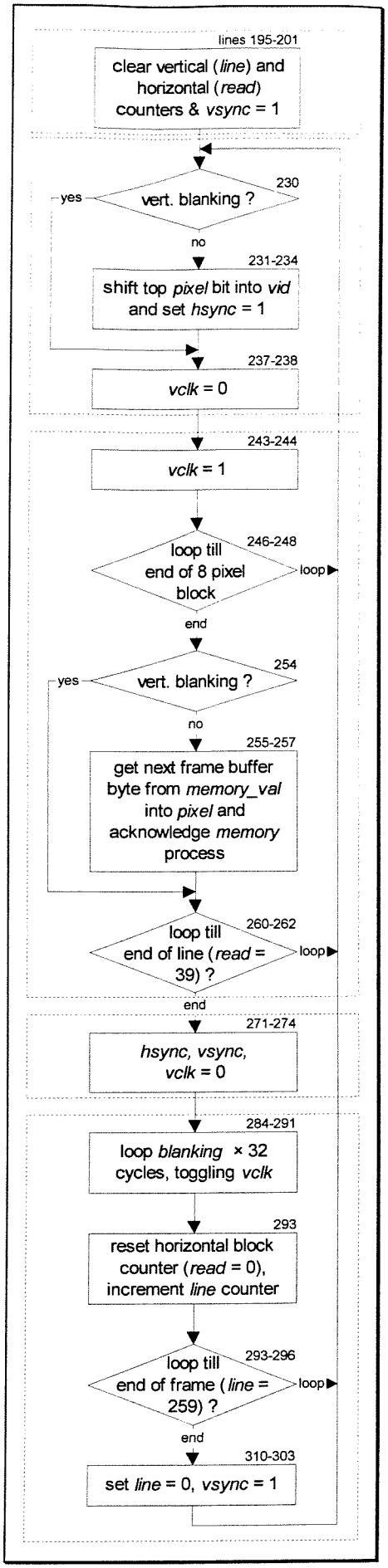


Figure 7.9 VidProc raster process flow chart

the required video timing), the blanking period is adjusted via the 4-bit *blanking* input. Once the 320 pixels for a whole line have been output, *vert_loop* ensures all sync pulses are low (lines 271-274) and then executes a blanking loop (*delay_loop1*, lines 284-291), which iterates through 1 + *blanking* × 32 clock cycles, while continuing to generate *vclk*. It is essential that on exit from this loop, *vclk* is high, ready for the immediate start of the next line, hence the additional cycle in *delay_loop* ensuring an odd number of iterations.

3. *Horiz_loop* (lines 221-262) is responsible for outputting the 320 pixels in a single *hsync* pulse. Because pixel data is stored in memory as 8 pixels per byte, an inner loop (*block_loop*) does the actual outputting, *horiz_loop* just retrieves each successive byte from the *memory* process into the local *pixel* variable, and acknowledges its receipt (lines 255-257). Thus, the loop repeats 320 ÷ 8 times, controlled by the *read* variable. Note that data is only output for lines 0 to 255; if the *line* counter is greater than this, ie. if bit 8 is set, only *vclk* is generated, hence the condition on line 254.

4. Finally, the inner most *block_loop* (lines 226-248) actually outputs a single byte of pixel data serialised on each *vclk* cycle. Referring to the timing diagram of Figure 7.8a, the loop operation can be seen as setting *vid* to a pixel value with *vclk* going low (lines 213-238) and on the next system cycle taking *vclk* high to load the pixel into the display (lines 243-244). The current pixel value is taken from the top bit of the *pixel* variable (line 232), followed by a left shift (line 234), with the loop repeating eight times. Note that data is again only output when *line* is less than 256 leaving only *vclk* cycles for the blanking sections. Also, *hsync* is set high along with *vid,* thus obtaining a sufficient setup time between *hsync* and the *vclk* rising edge (specified in the data sheet as at least 5ns), and automatically ensuring that *hsync* remains low during the horizontal and vertical blanking periods.

One feature of particular note in this process, is the way in which all the iteration and termination logic occurs at the end of the loops. This is vital for the scheduled timing of the final implementation and is discussed in detail in section 7.4.3.

7.2.3 The Render Process

The *render* process (lines 331-452) implements the user interface to the video processor, providing facilities for drawing into the frame buffer to create the required display image. A plotting command is executed by first placing the command code and the necessary parameters on the *command*, *x*, and *y* input ports, and then lowering the *execute* pin. When *render* is next free to process a command, the *ack* output is raised indicating that the inputs must remain stable, during which time various internal variables are initialised according to the required plot action (lines 349-393). *Ack* then goes low to show that the command is being processed and the input data is no longer required. The controlling device may then continue with other operations including setting up the next command which will be automatically loaded as soon as the current plot is completed. The use of level-sensitive (as opposed to edge-sensitive) detection for *execute* (lines 345-347) means that a new command can be requested at any time, other than when *ack* is high, without having to wait for a ready signal before setting up the inputs. The only important timing issue is that, following a command request, *execute* must go high immediately after *ack* goes high to guarantee the command is not processed twice, as some operations only require a single cycle before returning to the detection loop. The relationship between *execute* and *ack* is clearly demonstrated in the timing diagram (Figure 7.8a) which shows the execution of a PLOT POINT command.

Twelve commands are implemented by *render* as summarised in Table 7.1. These split into five groups:

1. Commands 12 to 15 set the render mode. Recall that the *memory* process responds to a request from *render* by masking the value in *render_val* with the contents of memory location *render_addr*. Setting the render mode changes the mask operation used to one of: FILL, SET, CLEAR or XOR. FILL operations overwrite the original memory byte with *render_val*, which is useful for screen clearing or setting up an image from an external bitmap. SET mode superimposes data over the current contents, only overwriting those pixels set in *render_val*, ie. a logical OR operation. CLEAR mode is the opposite of SET, where the pixels set in *render_val* are cleared on screen, ie. a NOT AND operation. Finally, XOR, as its name suggests, performs an exclusive OR operation, thus bits set in *render_val* invert the corresponding screen, enabling effects such as flashing pixels and animation. The render mode is entirely independent of any

other plot operations, allowing any combination of mode and command. For example, to clear the whole screen, CLEAR mode may be used in conjunction with "fill screen with byte 255"; however, the same command using XOR mode will invert the entire display in a single operation.

2. Commands 0 and 1 are the single point MOVE and PLOT commands. Plotting in the video processor is based on the concept of a persistent graphics cursor, where lines and characters are plotted "from the current cursor position", which is automatically advanced to either the end of the line, or the next character position along, after the operation. MOVE and PLOT take x and y values as parameters, and set the cursor to the specified pixel location, with PLOT also plotting a single point there. Note that some commands, such as character plotting, treat the cursor as the whole byte containing the cursor pixel instead of the exact single bit position.

3. Commands 2 and 3 are byte plotting operations where FILL SINGLE BYTE plots the 8 bits of y onto memory location containing the cursor, and FILL SCREEN fills the entire display with byte y. This operation is used in conjunction with CLEAR or SET modes to clear the whole screen.

4. Commands 4 and 5 plot the character with ASCII value y at the current byte-aligned cursor position. The video resource ROM shown in Figure 7.5 contains bitmap images for 256 8×8 pixel characters, addressed by their ASCII value. Both the character plot commands use the y input as the basis for this address, and plot the resulting 8×8 image in the 8 screen bytes down from the current cursor position. Thus, referring back to the frame buffer memory map (Figure 7.6), plotting a character with the cursor at position (5,1) will fill in addresses 40,80,120..320, producing the same result for all horizontal positions from (0,1) to (7,1). This restriction makes the plotting logic much simpler as there is no need to split the character image across several horizontal memory locations. The difference between the two commands is that whereas CHARACTER RIGHT (command 4) moves the cursor on to the next horizontal character position, eg. (8,1) moves to (16,1), CHARACTER DOWN (5) moves the cursor vertically, eg. (8,1) moves to (8,9). In fact, the top 128 characters in the resource ROM are the same standard ASCII character set as the first 128, but rotated by 90° to enable true vertical character plotting, as illustrated by the axis labels in the spectrum analyser photographs (Figure 7.22 and Figure 7.23) and Figure 7.16.

5. The last two commands, 6 and 7, plot vertical and horizontal lines from the cursor position, of length *y*. Vertical lines are plotted on a pixel-by-pixel basis using the offset of the cursor position from its byte origin to obtain a pixel mask image from the bottom eight resource ROM locations. For example, plotting a vertical line from (12,1) will retrieve a mask from ROM address 5, representing the binary value 00001000, since position 12 is the fifth pixel to the right of byte-aligned position 8. This image is then used as the value of *render_val* for plotting each byte along the vertical line. In contrast, horizontal lines only plots whole bytes to avoid the extra logic required for filling in individual pixels from the start and end points to the first byte-aligned position. Plotting a horizontal line from (12,1) therefore, will actually start at (8,1). Furthermore, the length value specifies the number of bytes so the actual line length in pixels should be divided by 8 before executing the command. For pixel-accurate horizontal lines, the PLOT POINT command must be used to fill in the necessary start and end sections.

Code	Operation
0	MOVE cursor to (x,y) position
1	PLOT POINT at (x,y) position and move cursor
2	FILL SINGLE BYTE with y
3	FILL SCREEN with byte y
4	plot CHARACTER y at cursor position, move RIGHT
5	plot CHARACTER y at cursor position, move DOWN
6	plot VERTICAL LINE down from cursor, length y (pixels)
7	plot HORIZONTAL LINE right from cursor, length y (8 pixel blocks)
12	set render mode to FILL
13	set render mode to SET (OR)
14	set render mode to CLEAR (NOT AND)
15	set render mode to XOR

Table 7.1 Video processor command codes

To simplify the generation of command and parameter input codes, a simple command compiler is provided which takes as input a textual list of commands, and produces the required command, x and y values to pass to the video processor.

Figure 7.10 shows a flow chart describing the operation of the *render* process. Like *raster*, it uses an infinite outer loop instead of the implicit process loop, in this case to allow the use of the *next* statement (line 387) to immediately terminate the current iteration and start

again from the top; a much more elegant mechanism than using large conditional *if-then-else* blocks. Each iteration corresponds to the execution of a single command, thus at the head of the loop is the command request detection code (lines 345-347). Note the use of a *while* loop rather than a *wait until execute=0* statement to detect a level and not an edge, so that if *execute* is already low on reaching the detection stage it will be treated as a new command, rather than waiting for a subsequent high to low transition.

Once a command is detected, *ack* goes high (line 351) to ensure input stability during the initial setup stage. Command execution proceeds in two stages: first, the requested command is pre-processed to set up the internal variables (lines 354-400); then the plot operation is performed. All output is based around a single generic loop (*count_loop*, lines 402-432), for both horizontal and vertical plotting. The number of loop iterations, orientation (horizontal or vertical) and plot data bytes are all set up during the pre-processing stage from the plot command. Also at this point, the frame buffer memory location corresponding to the current cursor position is loaded into *render_addr* ready for the first *memory* request. Within the loop itself, each iteration updates the single screen location specified in *render_addr* with the mask value in *render_val*, via a request to *memory* (lines 407-411). This is followed by a section which updates *render_addr* and decrements the *loop_count* variable (lines 413-431) depending on the command type, and also updates the resource ROM address (*res_addr*) for character plots. Note that FILL SCREEN is a special case wherein *loop_count* is ignored, and the loop only terminates when *render_addr* reaches the end of the frame buffer: memory location 10239 (line 418). When the loop has completed, the current cursor position is updated (lines 439-449), again depending on the command type, before the outer loop returns to wait for the next command request.

For full details of the mechanics of plotting, and indeed any other section of the video processor, refer to the Listing C.3 in Appendix C which is heavily commented.

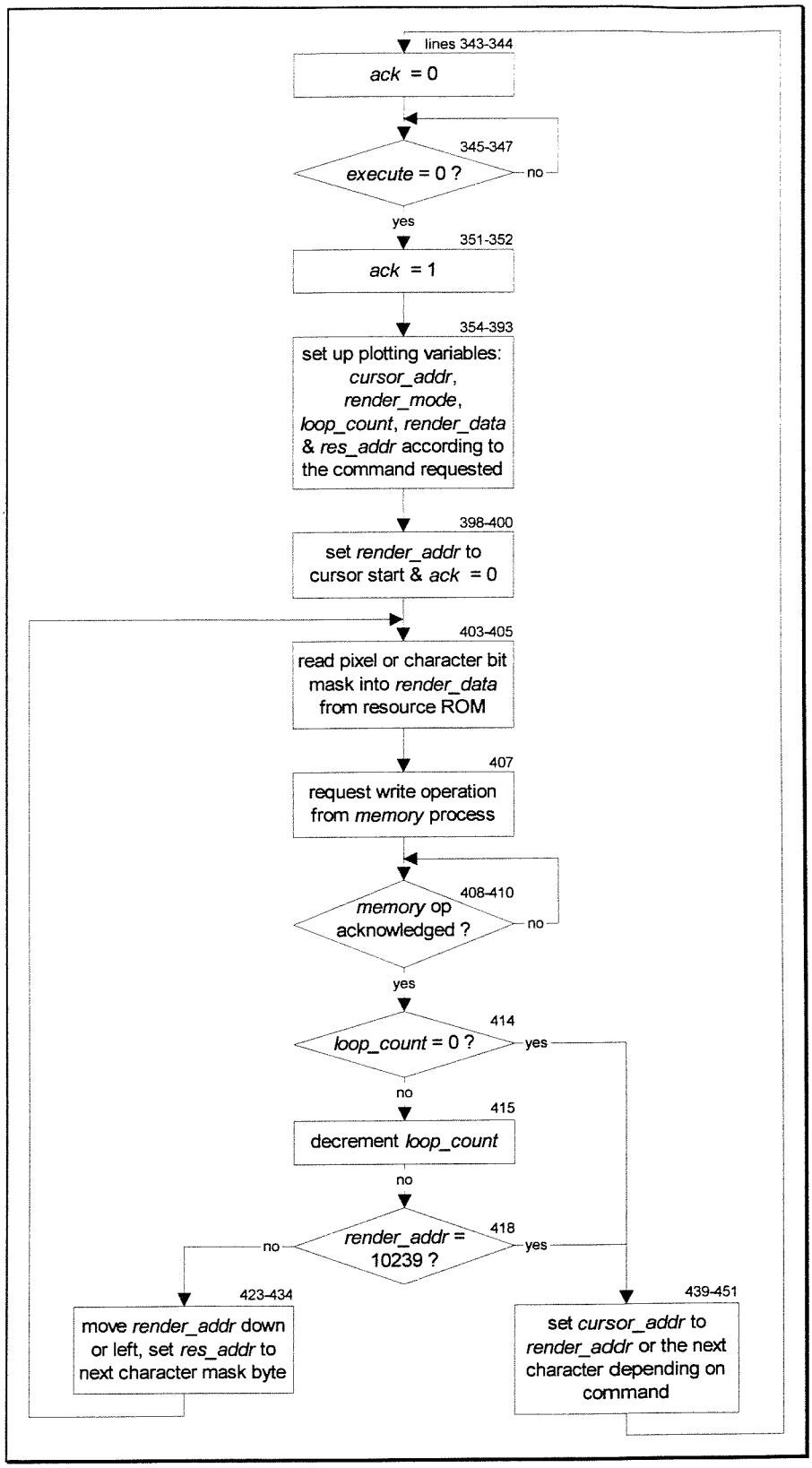


Figure 7.10 VidProc render process flow chart

7.3 System Control and FFT

At the heart of the system is the main controller, **FFTCont**, responsible for sampling, calculating the FFT, and displaying the results using **VidProc**. The flow of data along the two main system buses is directed by the controller via control signals on the other components, as illustrated in the block diagram of Figure 7.3. The design itself, included in Appendix C as Listing C.4, is formed from two concurrent processes:

1. *Controls* (lines 74-124) monitors the four control panel buttons *hold*, *zoom*, *pan left* and *pan right*, setting up various internal signals used to determine the display settings (zoom level, frequency range and hold state).

2. *FFTControl* (lines 154-495) is the central system control process comprising the three main operations from Figure 7.1, with the aid of the FFT arithmetic unit, **FFTProc**, to perform all the complex arithmetic portions of the FFT algorithm.

The following sections discuss each process in more detail.

7.3.1 The Controls Process

Controls is a relatively small process which simply monitors the *controls* input pins, reacting to button presses as they occur. Figure 7.11 is a flow chart illustrating its basic operation. The process is sensitive on the *controls* input by means of an initial *while* loop (lines 80-82) terminating when a change in state is detected, ie. a button is pressed or released. Each of the *left*, *right* and *hold* buttons are then tested and, if they have just been pressed:

- *Left* or *right* alone (ie. without *zoom*) pan the display in the appropriate direction by incrementing or decrementing the *disp_base* signal (lines 90-92, 105-107).

- *Left* or *right* pressed while the *zoom* button is down cause the display to zoom in or out by incrementing or decrementing the *zoom* signal (lines 94-96, 109-112).

- Each press of the *hold* button toggles the state of the *hold_on* signal, ie. inverts it. (lines 118-120).

At the end of the process, *prev_cont* is updated from the current *controls* input, ready for detecting a change on the next process iteration.

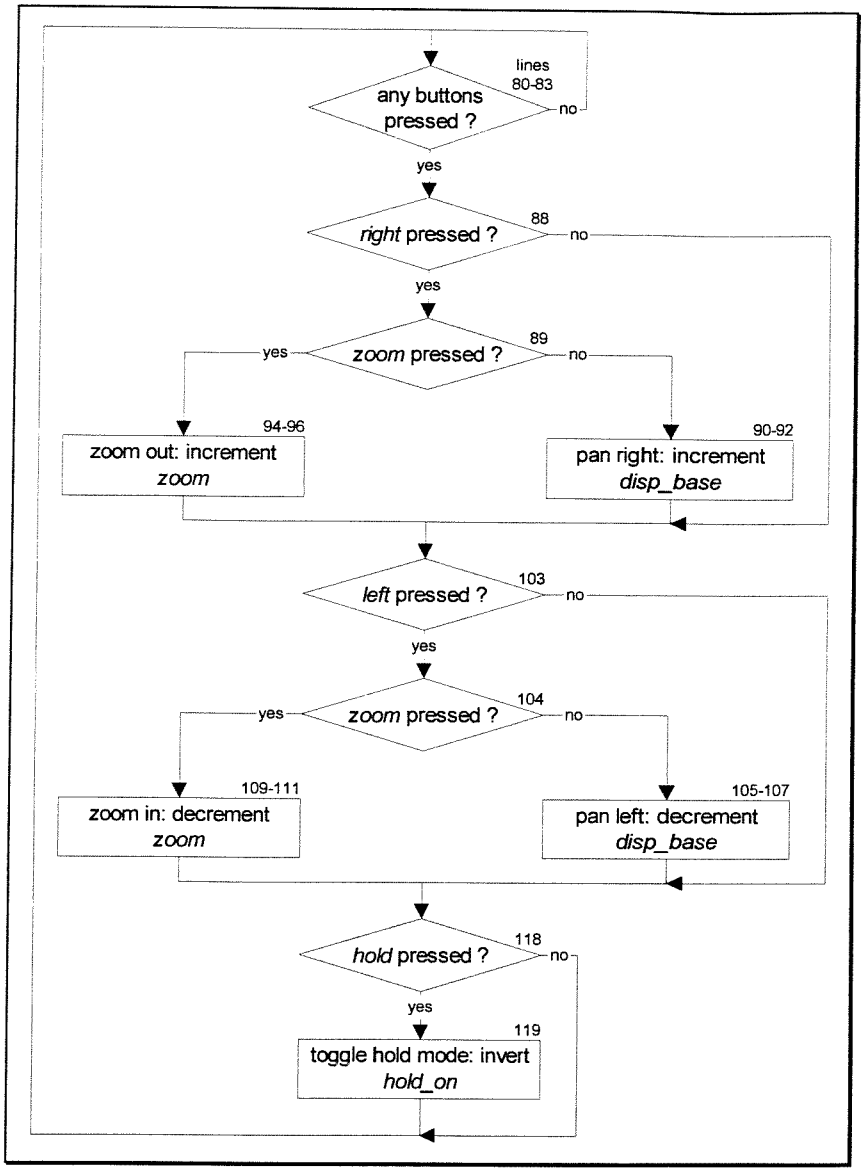


Figure 7.11 FFTControl controls process flow chart

The three global signals modified by *controls* are used within the main *FFTControl* process as parameters to the results display code:

- *Disp_base* holds the address in the FFT memory for the first (far left) frequency value to be displayed. Thus incrementing *disp_base* (*pan right*) moves the window of visible frequencies up the scale, while decrementing it (*pan left*) moves down towards 0Hz. To limit the number of key presses required between the two ends of the frequency range, the actual start address is taken as *disp_base* × 8 (line 398).

- *Zoom* determines the width in pixels of each vertical frequency bar on the display. During the output stage (lines 491-489), frequency bars are build up from several adjacent vertical lines. *Zoom* is used within the output loop to initialise a counter, *zoom_count*, which increments as each line is drawn. When this counter overflows, a one-pixel gap is left to separate the bars, before moving on to the next frequency and re-initialising *zoom_count*. Thus the actual bar width is 16 – *zoom* pixels, giving a total of 15 levels of zoom.

- *Hold_on* determines whether or not the sample/FFT operation is performed. If *hold_on* is low, a new 512 point FFT is calculated and displayed for each *FFTControl* iteration, however, if it is high, the screen image is frozen in "hold on" mode. The *hold* button therefore acts as a hold on/hold off toggle.

One final point about the use of external controls is that all buttons must be properly de-bounced before connecting to the *controls* input. The speed of the final circuit (each iteration of the *controls* process takes around 0.3μs) is such that de-bouncing will not occur automatically.

7.3.2 The FFTControl Process

FFTControl forms the bulk of the controller device, being responsible for performing the core tasks, and co-ordinating all the peripheral components. The flow chart of Figure 7.12 shows how the process splits into five sequentially executed blocks controlling the sampling, calculation and display of the Fourier transform (as described in section 7.1.1). Within each of these blocks, the controller directs data along the two main system buses connecting the various system components (shown in Figure 7.3) as follows:

1. The *W ROM/Video command* bus joins the W ROM data output, the FFT processor, the video processor and the controller video output ports (*vid_command*, *vid_x*, *vid_y*).

    1.1. Static display initialisation drives the bus from the top 512 words of the W ROM (containing video commands for drawing the titles and axes), connecting to the video processor. The ROM address, output enable and **VidProc** execution signals are all generated by the controller.

1.2. The FFT calculation phase again drives the bus from the W ROM, this time with the bottom 512 words connecting to the FFT processor *w_in* port to generate the appropriate $e^{2\pi i/N}$ constant. All addresses are again generated by *FFTControl*.

1.3. The controller itself drives the bus through its *vid_command*, *vid_x*, and *vid_y* outputs when displaying the magnitude versus frequency plot on the screen, making sure that the W ROM data port is first set to a Hi-Z state.

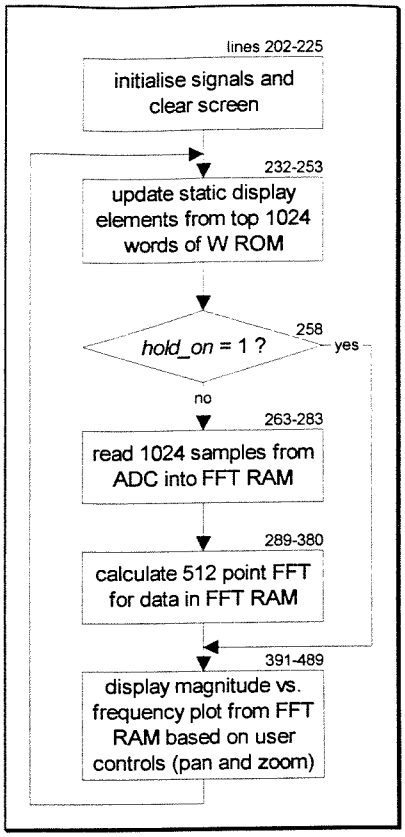


Figure 7.12 FFTControl process flow chart

2. The *FFT RAM* data bus connects the sample output (*data*) on the ADC controller, the FFT processor data input, and the modulus lookup ROM.

2.1. During input sampling, the ADC controller output is enabled thus driving the memory data port. The controller then generates the destination address and read/write control signal to load 1024 samples into the FFT RAM ready for processing.

2.2. When calculating the FFT, the RAM itself drives the bus, connecting to the processor *j_val* and *k_val* inputs. The controller generates the appropriate address, read/write, output enable and **FFTProc** control signals to load values into the processor. Once calculation is complete and the processor *ready* output goes high, the controller then tells **FFTProc** to drive the bus with the calculation results, while it writes them to the appropriate memory locations.

2.3. During the results display block, the FFT RAM data port drives the bus and connects to the modulus lookup ROM address input. The modulus data output is read by the controller (*mod_data* port), which represents the modulus of the complex number currently output by the FFT RAM. Thus as the controller alters the memory address, the result for each frequency point can be read.

The use of multiple function buses to directly connect the various components, in preference to passing all information via the controller, limits the number of controller I/O pins (which would be far too high to fit on a single device) and also reduces the complexity and number of registers required by the design. The main drawbacks are that the system is more complex and therefore susceptible to small errors (eg. two devices concurrently driving a bus), and the controller cannot pre-process data on its way from one point to another.

Looking again at the *FFTControl* flow chart of Figure 7.12, it can be seen that the various stages split into two general groups: the FFT blocks (sample and calculate) and the display code (static and results). The initialisation section simply sets up the various system control signals, and executes and single clear screen command. In addition, a system reset signal is generated (*sys_reset* output) that connects to all the other devices, with the basic power-on reset only applied to the controller itself. This means that on start-up, the main control signals to the various system components are all properly set, thus ensuring that they initialise cleanly, and there is no possibility of starting in an undefined state.

The following two sections discuss the FFT and display blocks in more detail.

7.3.3 Sampling and FFT Calculation

The FFT algorithm implemented in the spectrum analyser is based on the popular Cooley-Tukey algorithm [112, 113, 115], described in detail in Appendix C. The first step in any signal processing application is, of course, the sampling of the input signal. This is achieved using a 16-bit analogue to digital converter[1], controlled by a separately synthesised controller device, **ADCCont** (see Listing C.6 in Appendix C), implemented on a small CPLD. It is a relatively simple device which serves four purposes:

1. Generates a clock signal for the ADC, dividing the system clock by 2 or 4, based on *clk_sel*. The corresponding sample rates are 31.25kHz and 15.625kHz giving a total display range of 0-15.625kHz in steps of 3.05Hz, and 0-7.81kHz in steps of 1.53Hz.

[1] The device used is a National Semiconductor ADC16071 16-bit delta-sigma converter, which generates a 2's complement sample at a rate of CLK/256 where CLK is the ADC clock input. Data is output in serial form as shown in the data sheet extracts in Appendix C.

2. Communicates with the ADC retrieving 16-bit 2's complement serial data for each sampled point.

3. Converts the 16-bit samples into the 12-bit fixed-point format used by the FFT processor (see later), which is appended with a further 12-bit imaginary component to form a full 24-bit complex number. The tri-state output pins on the CPLD are exploited to enable direct connection to the FFT RAM data bus, with the port drive state controlled by the *noe* input driven by the system controller.

4. Tracks the highest value over every 1024 samples, and outputs it as a level on a 10-segment bar-graph display, giving a visual indication of the input signal level.

More details regarding the operation of **ADCCont** may be found in the highly commented source code listing in Appendix C.

Since all control of the sampling frequency and sample data manipulation (converting the serial to a parallel complex number) is done in **ADCCont**, all the main controller sampling block has to do is loop from FFT RAM address 0 to 1023 and, when a new sample is signalled (via the **ADCCont** *ready* output), write the data on the RAM bus into the memory (lines 263-280). The only extra feature is that instead of looping sequentially through the addresses, each sample is stored in bit-reversed address order, ready for immediate processing by the FFT algorithm. This eliminates any additional re-arrangement of the data points prior to the FFT, and also costs nothing as the operation is simply a matter of reversing the direction of the address bits (cross-connecting wires on the FPGA). A special *bit_flip* module was created specifically for this task, discussed in more detail in section 7.4.1.

Once a full 1024 samples are in the FFT RAM, the FFT calculation begins in earnest (lines 289-380). The fundamental operation of the algorithm, including its derivation, is described in Appendix C with a C++ implementation using floating-point arithmetic shown in Listing C.1. In general terms, this algorithm loops through $\log_2 N$ iterations, where N is the number of samples (in this case, 1024, for 10 iterations), combining pairs of complex data points, together with a constant, to form two new values to replace the original pair. At every stage, each of the 1024 data points is updated, leaving, at the end of the whole operation, the discrete Fourier transform of the input samples, with the first 512

memory locations representing the positive frequency components from 0Hz to half the sampling frequency.

For the VHDL implementation it was necessary to convert the original floating point operations into an integer (or rather bit vector) form using fixed-point arithmetic. Due to the limited resources available on the FPGA, data was limited to 12 bits, thus requiring 24 bits for a real/imaginary complex pair. Before writing the VHDL, the original floating-point implementation was modified to use integers in order to investigate various factors such as the range and scaling of the fixed-point calculations. The resulting code is shown in Listing C.2 in Appendix C using the same bit-width and scaling factors as the final VHDL translation. Comparing the floating and fixed-point C++ versions:

- The basic algorithm is identical, differing only in the core data manipulation code which now uses integers.
- To match the capabilities of the VHDL version, only the bottom 12 bits of the integers are used.
- The fixed-point representation locates the binary point between bits 9 and 10, giving a range of -2.0 (integer value 2048) to 1.9990 (2047) where the least significant bit represents 9.77×10^{-4}. See Figure 7.13.
- The initial sample data is assumed to be in the range –1.0 to 1.0, well within the scope of the fixed-point format.
- The result of the integer multiply must be normalised to maintain the same binary point position. Since the operation will have been shifted the point 10 places to the left, normalisation involves a sign-extended (arithmetic) right shift by 10 bits, as illustrated in Figure 7.13.
- Each iteration of the main loop results in a general increase in the magnitude of the data points due to the addition operations in the algorithm (Listing C.2, lines 49-52). In the worst case of a pure DC input signal, this results in a doubling of the maximum data value on every iteration ending at 1024.0 for stage 10. To maintain this range would require a bit-width of 24 bits, so instead, each updated data value is divided by 2 (arithmetic shift right) immediately after calculation (Listing C.2, lines 55-58), thus keeping the final fixed-point number within the range –1.0 to 1.0.Note that although the

maximum result of the operation prior to this scaling is 2.0, which does not quite fit into 12 bits, investigation with the integer C++ code shows that in reality, slight inaccuracies in the least significant bit mean that the actual maximum value is 1.998 (integer 2046), thus the 12 bits are just sufficient. In any case, since the input signal is AC coupled prior to sampling there should be no DC component (except during signal overload), so this worst case will not occur.

- A side effect of the above scaling is that since the number of binary places is the same as the number of iterations (10), the final FFT results will actually be the correctly scaled integer values (ie. no longer fixed-point).

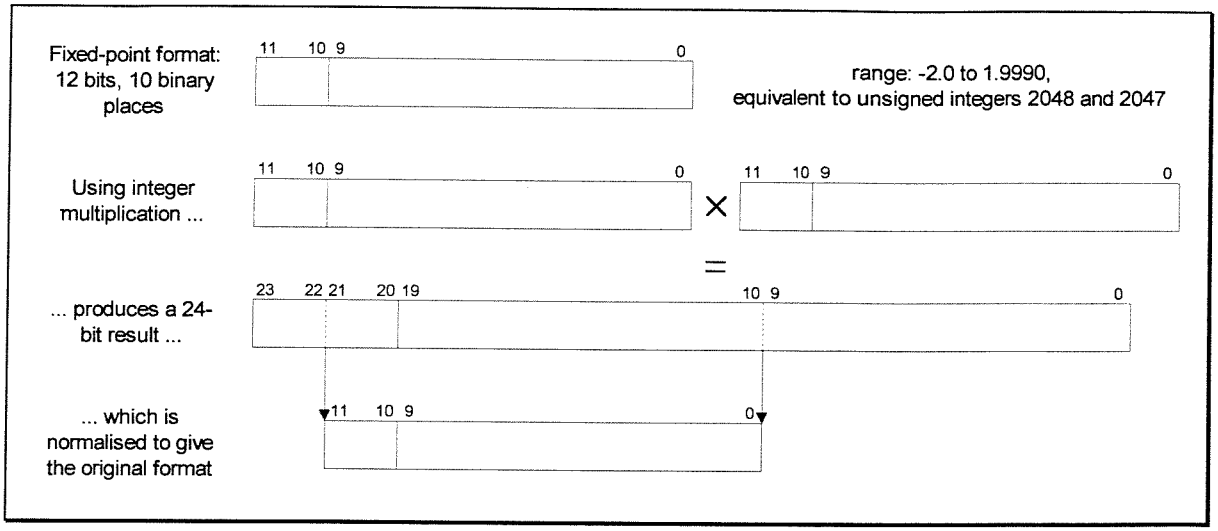


Figure 7.13 Fixed-point number format example

Once the development and testing of the fixed-point C++ code was complete, it was directly translated into an equivalent VHDL implementation. While simulation showed this to operate correctly, it would not fit on a single FPGA device and so was split over two designs with the controlling loops partitioned into the main controller, and all the fixed-point arithmetic transferred into a new FFT processing unit, **FFTProc** (Listing C.5). This also has the advantage of removing the need for the controller to have any interaction with complex data points, which would require a further 24 I/O pins. The use of an external modulus lookup table (the modulus ROM) shrinks the 24-bit values into just 8 pins (*mod_data* input), and additionally scales the result to the desired display resolution needing no further data manipulation in the controller. An added advantage is that the controller is entirely independent of the complex number size and format, thus the same basic system could easily be adapted to use 8 or 16-bit samples, the only changes being to the FFT processor, memory and modulus lookup.

Taking the controller section first, the overall operation of the FFT is illustrated in the flow chart of Figure 7.14. It is the responsibility of this code to provide memory addresses and control signals for each j and k pair of data points to be combined, together with the location in the W ROM of the appropriate $e^{2\pi i/N}$ constant (see the algorithm in Appendix C for details). These addresses are then applied to the FFT RAM and W ROM respectively, and the resulting j and k values loaded into the FFT processor by means of a two-way handshaking protocol (lines 309-320). Note that the **FFTProc** *w_in* input is permanently connected to the *W ROM/Video command* bus as only one value is required per operation. Once the processor has completed the calculation the controller again applies the two j and k addresses to the FFT RAM, and for each one, writes the data from the processor into the memory (lines 335-362). The communication involved in one calculation is illustrated in the timing diagram of Figure 7.15 for which the following points can be made:

- The *ready* output from the FFT processor goes high to signal that the unit is able to accept j and k data for a new calculation. In response, the controller sets up the FFT RAM address (*ram_addr* output) to the location of the j value (including setting the RAM to drive the FFT data bus) and takes the processor *load* signal high telling it to read in the first of a new pair. Two cycles are allowed for the load and then the procedure is repeated for the k value, this

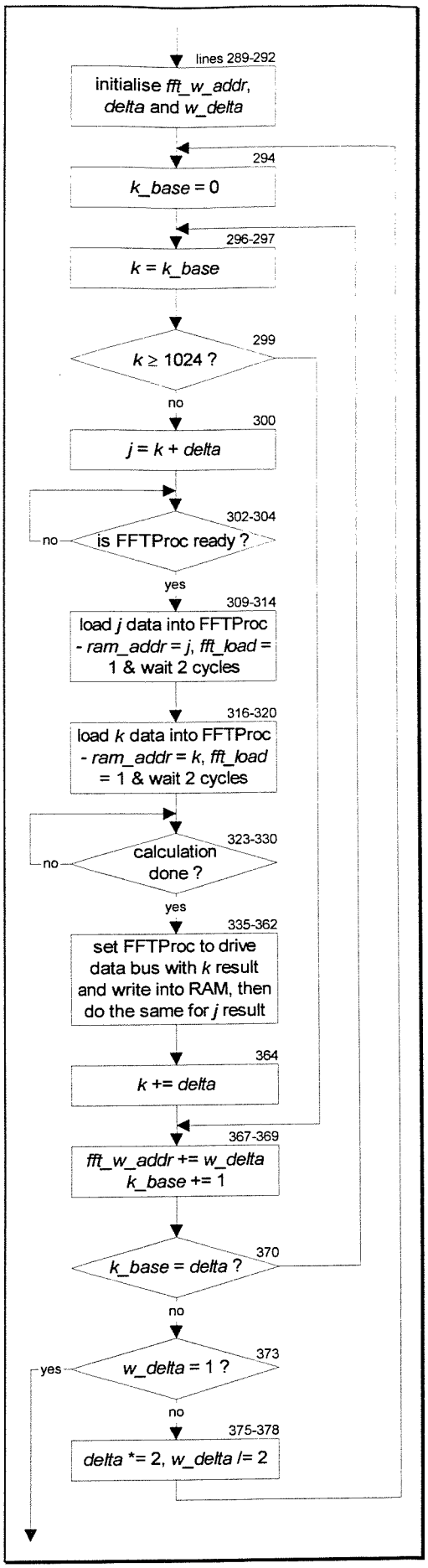


Figure 7.14 FFT algorithm flow chart

time loading on the falling edge of *load*. The processor then takes *ready* low, indicating calculation in progress. Note that whereas the *j* and *k* are stored internally in **FFTProc** and so are not required after loading, *w_in* must be kept stable throughout the calculation.

- When a new pair of *j* and *k* values have been calculated, the *ready* output is taken high ready to accept new data. Within the processor, *j* and *k* values are output (and input) via a common tri-state port connecting to the FFT data bus. The value driving this port is controlled via the *j_noe* and *k_noe* output enable pins, hence to update the result of the calculation in memory, the controller sets up the appropriate addresses and takes the corresponding output enable low, then, after a 1-cycle setup period, writes the data into memory, as shown in the second half of the timing diagram.

The remainder of the controller is simply a VHDL rendition of the looping structure in the FFT algorithm, which generates data point indices to be used as *j* and *k* addresses.

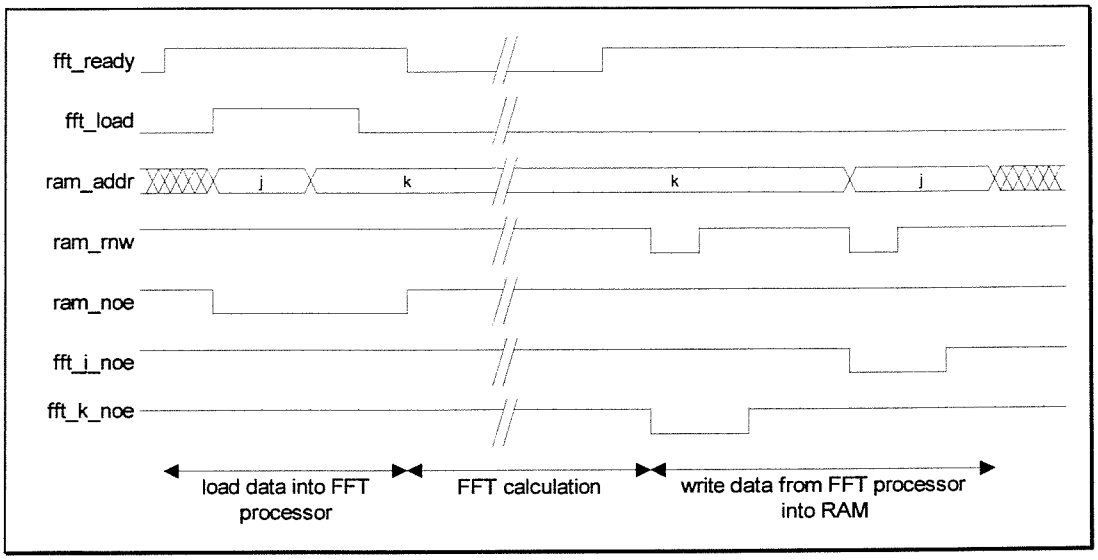


Figure 7.15 FFTCont/FFTProcessor single calculation control timing

The processor part, implemented in **FFTProc**, is actually a fairly short design (see Listing C.5) with the majority of the device being occupied by 24-bit registers and multiply operations. It is these multiplies that are the most interesting part of the processor as they are implemented using a special fixed-point multiplier macro operator, the source code of which is shown in Listing C.7, in Appendix C. This has a number of advantages over the standard combinational module:

- The macro operator implements a Booth multiplier [116, 117] which, once optimised after expansion, requires 27 clock cycles per operation, using a single 12-bit +/- arithmetic unit, and a 26-bit partial product register. Considering this speed penalty in the wider context of the whole design, each FFT outer iteration requires 1024×2 multiplies, thus for the whole 10-stage calculation, $1024 \times 2 \times 10 \times 27 = 55296$ clock cycles are approximately required. Based on a 16MHz clock frequency, this translates into 0.035 seconds which is about the same as the time required for 1024 samples at 31kHz. These figures suggest a display update rate of around 14Hz, not taking into account any of the drawing code, which is sufficient for a real-time display.

- Needless to say, the multiplier requires only a fraction of the FPGA real estate of the combinational equivalent, totalling around $8700\mu m^2$ compared to $31083\mu m^2$ (almost the entire $38400\mu m^2$ area). It also allows the use of a faster clock due to the simpler constituent modules.

- The final result is automatically normalised within the operator to return the required 12-bit fixed-point format (Listing C.7 line 69), thus eliminating the need for explicit normalisation in the processor itself.

- The multiply module provided by the low-level synthesis environment doe not support 2's complement numbers and so cannot be used without modification.

The last operation of the FFT processor before the calculation is complete is to divide the final results by 2, as explained earlier. This is accomplished by simply copying the top 11 bits into the bottom 11 bits (ie. sign-extended right shift, Listing C.5, lines 127-151). Note that in the special case of -9.77×10^{-4} (integer –1), the result is forced to 0 as a basic right shift would have no effect.

On a final note, whereas all the other devices in the system use only unsigned arithmetic modules, the FFT processor and the fixed-point multiplier macro operator use 2's complement arithmetic. As far as VHDL2IC and MOODS are concerned, there is no difference between the signed and unsigned units, the change is only made prior to low-level synthesis (or simulation) by using an alternate set of signed modules, along with the standard MOODS structural output.

7.3.4 Results Display

The final section of the whole system to be considered is the display output code, responsible for sending commands to the video processor to create the visible magnitude versus frequency plot, an example of which is shown in Figure 7.16. Referring back to the *FFTControl* flow chart of Figure 7.12, the display code splits into two sections executing before and after sampling and FFT calculation.

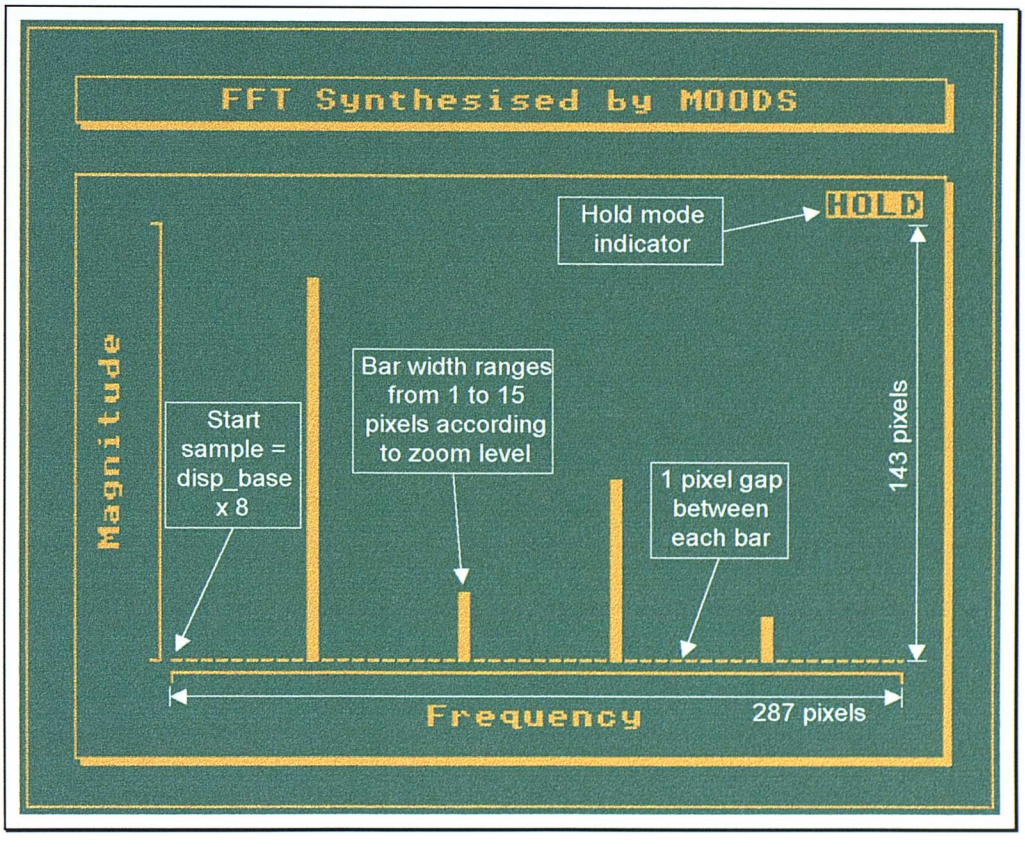


Figure 7.16 Example spectrum analyser display

The first block (lines 240-253) is responsible for creating the static elements of the display, (ie. the titles, axes etc. shown in Figure 7.16), which are stored in the top 512 words of the W ROM. This allows different displays to be created simply by reprogramming the ROM and, more importantly, takes up very little space in the controller. Since the W ROM is 24 bits wide, a complete **VidProc** command, composed of the 4-bit *command*, 9-bit *x* value, and 8-bit *y* value, can be stored in a single location. The controller then simply loops through the addresses and, for each one, tells the video processor to execute the command, waiting until it is acknowledged before continuing on to the next location. The ROM image is created using the video processor command compiler mentioned earlier.

One additional feature is that the top 32 addresses are only executed when in hold mode, enabling the creation of a visual "hold on" indicator as shown in Figure 7.16, with the lower non-hold commands ensuring that this indicator is erased when hold is toggled off. This is the reason why the static elements are updated on every controller iteration rather than just once, as a separate "hold on" loop would take the FPGA over its size limit (its already 96.9% full). The initial clear screen command is hardwired into the controller initialisation step (lines 222-225) so it is not required in the W ROM, thus eliminating any flicker that would result from a clear screen on every update.

Note that throughout the *FFTControl* process, the actual video processor handshaking code is encapsulated in a separate *exec_video_command* procedure (lines 180-195) which accesses the *vid_execute* and *vid_ack* top-level ports. This approach saves some area, since each command execution only uses one *call* control node, instead of the three *general* nodes required without the procedure, saving around 16 control registers in total.

After the screen image has been set up, the input signal is sampled and the Fourier transform calculated. The last step is then the display of these results as a magnitude versus frequency bar graph in the middle of the screen (see Figure 7.16). A fairly simple approach is taken due to the limited hardware resources available, illustrated in the flow chart of Figure 7.17. This revolves around a single loop (*x_loop*, lines 408-489), which moves sequentially from the left to the right of the display window, one pixel column at a time. On each iteration, the video cursor is moved to the top of the column and a black line drawn down to the top of the frequency bar to be plotted. A white (or rather yellow) line is then drawn from this point down to the x-axis thus erasing the previous image and drawing the new one in a single iteration.

The height of the bar is taken from the *mod_data* input connecting to the modulus ROM 8-bit data bus (lines 427-431), which returns the magnitude of the data point in the FFT RAM, scaled to the display window height, in this case 143 pixels. The actual frequency point returned is determined by the memory address output, *ram_addr*, initialised at the start of the loop from the global *disp_base* signal set by the *controls* process (see earlier).

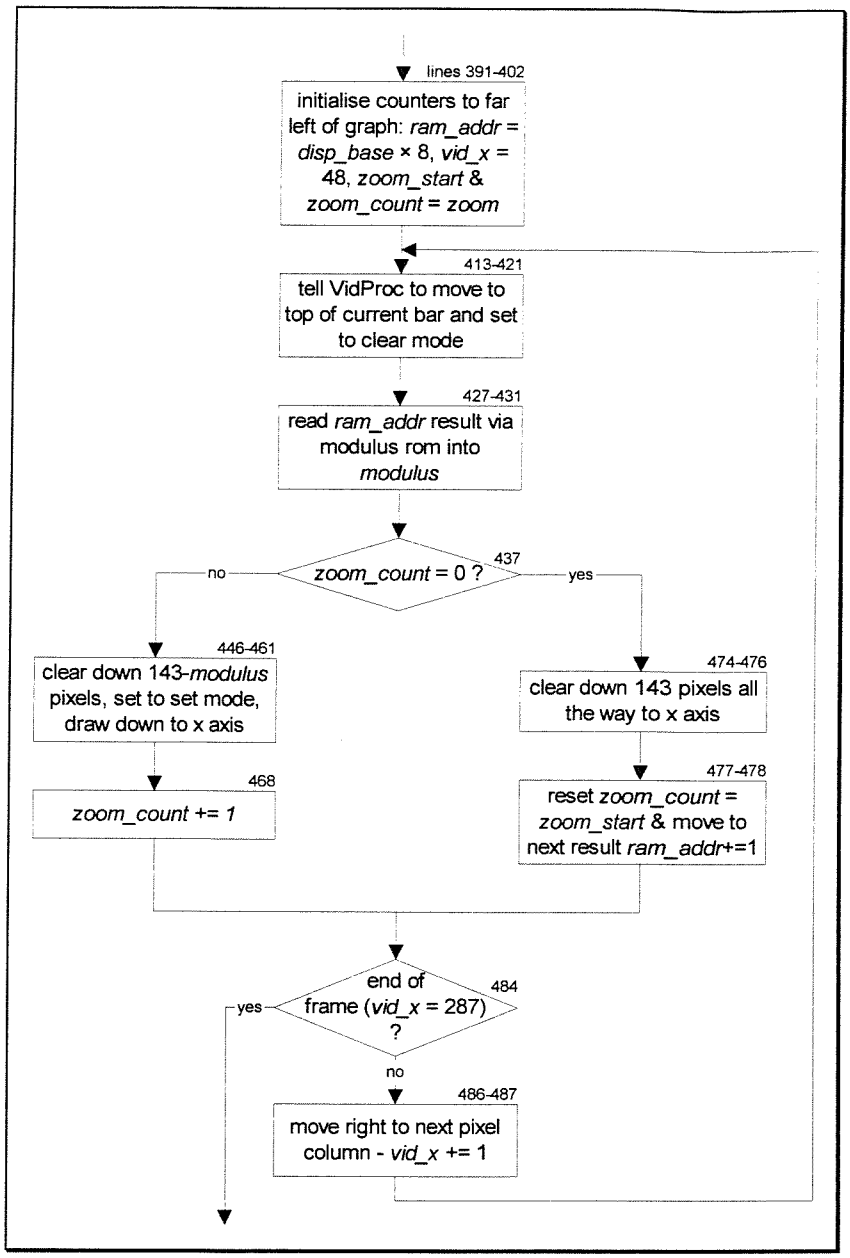


Figure 7.17 Magnitude vs. frequency display flow chart

Within the loop itself there are two possible plotting options: either the magnitude result is displayed, as described above; or a single black line is drawn down to the x-axis as a separator between frequency bars (lines 474-478). Which of these is chosen depends on the value of the *zoom_count* variable, which is initialised to the current *zoom* global value, and is incremented each time a single pixel column is drawn (line 468). When *zoom_count* overflows to 0, the separator bar is drawn, *ram_addr* is incremented, and *zoom_count* is re-initialised from *zoom*. Thus for each frequency point, a vertical bar of width 16 - *zoom* pixels is plotted, together with a single pixel separator. Note that *zoom_count* is actually set from a local copy of *zoom* (*zoom_start*) , ensuring that if a zoom in/out occurs while

outputting, the display does not change zoom level half way through, but waits until the next full screen update.

On a final note, when the modulus ROM value is read (lines 427-431), if *ram_addr* is outside the 512 point result range, the bar height is forced to zero to avoid displaying the first few negative frequency points from RAM locations 512 to 1023.

7.4 Synthesis and Optimisation Techniques

This section highlights several important techniques used at various stages during design and synthesis, to target particular aspects of the final optimised implementation. Although specifically described in relation to the spectrum analyser, these all illustrate general methods applicable to most behavioural designs. The following sections describe four general categories:

1. Special area reduction techniques for use when an area optimised design does not quite fit into the target device.

2. Modifications to structural output generated by MOODS, to implement features such as tri-state I/O ports.

3. Methods used to provide a high degree of control over the cycle-by-cycle timing of the optimised design.

4. Approaches taken to inter-process and external synchronisation and communication, and the possible pitfalls which must be considered.

7.4.1 Area Reduction Techniques

All but the simplest of designs involve some degree of compromise between cost and functionality. In the case of the FPGAs used for this project, the limiting factors are device size and the number of I/O pins available. As has already been mentioned, initial attempts at a system design were far too large to fit in any single device, requiring various sub-sections to be split up into separate video, control and FFT processing functions. Even then, the best implementations were still some 9% to 27% too large, requiring either a substantial reduction in functionality (on top of compromises already made), or some

further modifications to the behavioural descriptions. In the event, a combination of the two approaches managed to fit each design onto a separate FPGA, with final utilisation figures ranging from 93% for the FFT processor, to 100% for **VidProc**, as detailed in Table 7.2 (page 234). These figures were achieved using a variety of the techniques described below:

- Faced with a necessary reduction of around 30%, the first victim to fall was the VHDL deferred signal assignment model, exploiting the VHDL2IC **/no_sigs** switch. This resulted in by far the most significant area decrease of between 15% and 18%; not surprising considering that all three designs are storage and control dominated, thus tending to exhibit a larger shadow register overhead, as described in Chapter 6. Note that although the control portion of **FFTProc** is relatively small, the fact that the multiplier is implemented by a single +/- arithmetic module, and the number of relatively wide registers required for the four 24-bit I/O ports, means that the use of **/no_sigs** still makes a large difference. The graph of Figure 7.24 (in section 7.5) shows the final area optimised figures with and without shadow registers, broken down into functional, storage, interconnect and control components. For both the controller and FFT processor, this step was sufficient to fit both designs on to FPGAs (although a few other savings were made using the general methods described below). The video processor however, was still approximately 4% too large, demanding further attention.

  The drawback of this approach is, of course, the relaxation of the VHDL simulation model, requiring more care to be taken to ensure that the synthesised implementation behaves as originally simulated. This situation is eased by including wait statements immediately after most blocks of signal assignments, coupled with the high level of control exerted over the exact scheduling of most I/O and inter-process communications (see sections 7.4.3 and 7.4.4).

- General techniques for reducing inefficiencies in the source code were also used, particularly in the video processor. These included using smaller = and /= comparators instead of their larger brethren, for example, an 8-bit = comparator occupies around $500\mu m^2$ whereas an 8-bit ≥ comparator requires $1000\mu m^2$. Better still, basing comparison on a single bit, such as in **VidProc** line 231 where instead of "*if line <= 255*" ($1000\mu m^2$), "*if line(8) = 0*" ($100\mu m^2$) is used. These types of modification may seem relatively minor compared to the total design size, however even a single register can make the difference between a design that fits, and one that does not.

Another valuable tool is the VHDL subprogram for encapsulating oft-repeated code segments. Even if this saves only a handful of control registers and interconnects, the gain can be worthwhile. In the case of the system controller, the *exec_video_command* procedure made enough difference, coupled with **/no_sigs**, to get the design onto an FPGA.

- Expanded modules also play a part in decreasing the area requirements of a design. Although not used in this project, all the module splitting methods discussed in Chapter 5 can have a significant impact on area. This also ties neatly in with the use of **/no_sigs** as module splitting trades a decrease in the functional area for an increase in control, registers and interconnect, which is exactly the make-up most affected by **/no_sigs** (as illustrated in the case of **FFTProc**).

  Expanded modules do feature in the spectrum analyser in the form of macro operators implementing the fixed-point multiply and *bit_flip* functions. These are special purpose expanded modules created explicitly for the project but, having been included in the expanded template library, are now re-usable in other designs. In fact, *bit_flip* is not, strictly speaking, a macro operator since it is implemented by a dedicated low-level module in the module library. The VHDL function header is declared as a macro operator but, instead of using a function number from the reserved macro range (see section 6.3.1), the ICODE database assigns to the BITFLIP instruction, a function within the range used by normal combinational modules. A definition for the *bit_flip* module is included in the module library, thus creating a user-defined low-level module. This can be a particularly useful mechanism, enabling a customised, hand-crafted low-level block to be included in the VHDL source code, without requiring any modifications to VHDL2IC or MOODS. In the case of *bit_flip*, what could be a rather inefficient behavioural description (see the *bit_flip* subroutine in the C++ FFT listings in Appendix C), is now a single module which just connects its input to its output reversing the order of bits.

  One feature not used in the spectrum analyser is macro ports. These would seem to be ideal for the memory operations, and were indeed used in some previous incarnations, however, a closer inspection of each of the designs reveals that at no point is a simple memory access used. For example, all accesses to the FFT memory are controlled by the **FFTCont**, but the data is actually generated by the external FFT processor device. The only realistic candidate for a macro port is the *memory* process in **VidProc**. Here

however, the memory address bus is shared between two outputs, *render_addr* and *raster_addr* (discussed in section 7.4.2 below), thus setting up the address is not simply a case of assigning a value to an output, as it is in the macro port, but involves the use a special select signal (*sel_render*). This is not to say that macro ports have no use; they have indeed been exploited in other MOODS projects, but, in a system with complex internal and external bus structures, their generality can make them unsuitable, particularly when space is at a premium.

- As a final resort to squeeze the last few drops of overhead out of a design, a few changes may be made directly to the compiled ICODE file prior to optimisation. This can involve removal of temporary registers, the manual inclusion of special instructions (although the mechanism described for *bit_flip* is more user-friendly), or any other low-level modification not accessible from VHDL.

Original ICODE SWITCHON block

```
SWITCHON select
  CASE #6     ACT sw24_case6
  CASE #0     ACT sw24_case0
  CASE #2     ACT sw24_case2
  CASE #3     ACT sw24_case3
  CASE #7     ACT sw24_case7
  CASE #1     ACT sw24_case1
  DEFAULT     ACT sw24_default
ENDCASE
```

Equivalent DECODE implementation

```
DECODE select
  CASE #0     ACT sw24_case0
  CASE #1     ACT sw24_case1
  CASE #2     ACT sw24_case2
  CASE #3     ACT sw24_case3
  CASE #4     ACT sw24_default
  CASE #5     ACT sw24_default
  CASE #6     ACT sw24_case6
  CASE #7     ACT sw24_case7
ENDCASE
```

CASES must be sequentially re-ordered for DECODE

any gaps in the sequence must be filled, while maintaining the correct behaviour

DECODES do not have a default clause, any unspecified CASES are equivalent to CASE #0

Figure 7.18 Converting SWITCHON to DECODE

One particularly effective modification, used in the video processor, is the conversion of ICODE SWITCHON instructions (generated from VHDL *case* statements) to DECODEs, as illustrated in the code fragments in Figure 7.18. SWITCHON allows an arbitrary number and ordering of choices (within the limits of the selecting variable width), and creates individual comparators for each one. DECODE on the other hand,

requires an ordered sequential list, but is implemented with a single decoder module. Thus in the example, the six 3-bit comparators of the SWITCHON are replaced by a single 3-bit decoder, saving around 935μm^2. Notice how CASEs not present in the SWITCHON (4 and 5) must be included in the DECODE block to fill in the gaps in the sequence. There are three such blocks in **VidProc** which, when converted to DECODES, makes for a total area saving of 3133μm^2 (7.9%), illustrated in the area breakdown graph of Figure 7.24. This is just enough to fit the entire design on a single device.

The drawbacks of this approach are its reliance on a high degree of knowledge of the workings of MOODS and the ICODE, the inconvenience of having to make the modifications after every VHDL2IC re-compile, and the possibility of adding errors to the design. It can, however, be of immense benefit in situations where a design is just slightly too large and all other options have been tried, and is also useful for trying out new features which may later be coded into VHDL2IC; this was how the macros ports and macro operators evolved.

7.4.2 Structural-level Modifications

VHDL2IC and MOODS do not, at present, allow any structural input (ie. VHDL *architecture*-level components), requiring all code to be encapsulated in VHDL processes. There are, however, certain situations where a small amount of structural code can be useful, in which case it may be added into the MOODS output netlist. As a general method, any internal signals intended to connect to structural elements should be implemented in the behavioural description as *entity* ports. This will ensure that the signal and any connecting circuitry is not optimised away during synthesis, on the grounds that the signal is never actually read from. Once a structural description has been created, these dummy ports can be converted into ordinary signals, and the extra structural elements connected into the design.

This mechanism is used to implement two particular features in the spectrum analyser:

1. Recall from Chapter 6 that VHDL2IC splits *inout* mode parameters into separate input and output ports. Once synthesis is complete, these two halves can be recombined together with a direction control signal to implement bi-directional and tri-state I/O ports. A particular example of this is the video RAM data bus in the video processor.

Here, the *vid_data* port has been manually split into input (*vid_data*) and output (*vid_data_out*) halves, but the same effect would be achieved if *vid_data* were defined as mode *inout*. The read/write mode is controlled by the *fp_noe* output which is set high for internally reading from the port (ie. a Hi-Z input), and low for generating an output, as shown in Listing C.3, lines 131-156. Notice that when in input mode, the output half can still be updated (lines 131-140), but the result will only appear on the data bus when *fp_noe* next goes low (on line 148). In the MOODS output description, the *fp_noe*, and *vid_data_out* ports are converted into ordinary signals, and *vid_data* changed from mode *in* to *inout*. All three signals are then combined to form a single bi-directional port using the VHDL code shown in Figure 7.19a. This basically says that the *vid_data* port is driven from *vid_data_out* if *fp_noe* is low, otherwise it is not driven internally at all, being an input. The resulting circuit diagram shows how this is actually implemented in the FPGA.

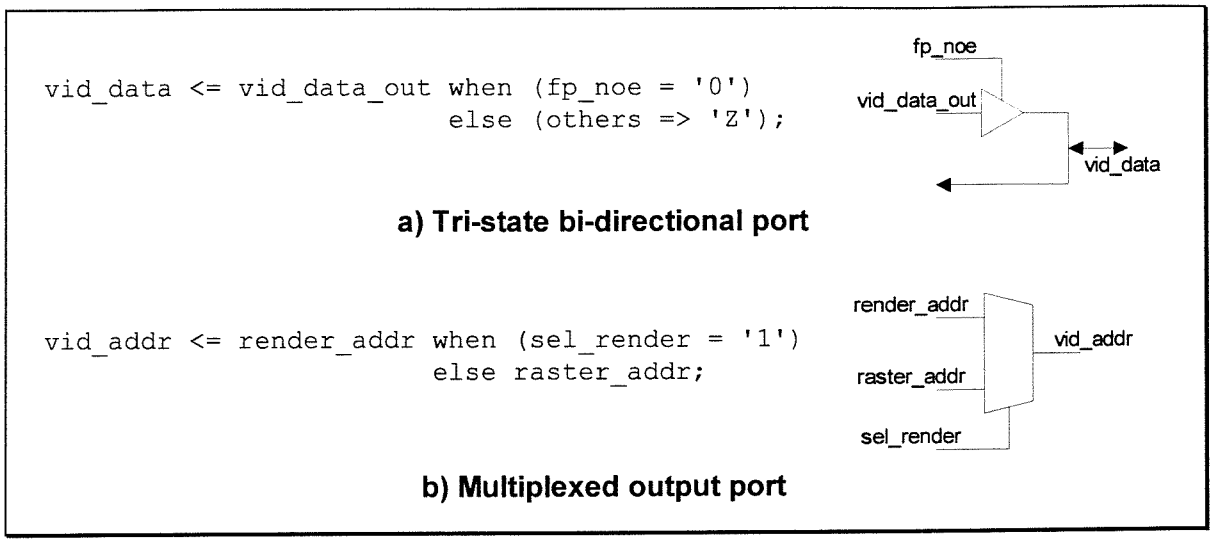


Figure 7.19 Combining ports using structural modifications

2. A variation on this theme can be used to save both registers and pins by multiplexing several ports through a single output. This mechanism is used to combine the two *render_addr* and *raster_addr* outputs in the video processor, saving in the process a 14-bit register (1400μm^2). The basic principle is to combine the two outputs into a single port, with the driving output controlled by the *sel_render* signal, itself a port. After optimisation, all three of these are converted into signals and a new 14-bit *vid_addr* output port is added. They are then all combined as shown in Figure 7.19b resulting in the creation of a multiplexor on the output. This can be clearly seen in the video processor internal block diagram of Figure 7.5. The same behaviour can be achieved in the VHDL source by implementing the two addresses as signals, with a single *vid_addr*

output loaded from either one as necessary, thus requiring three address registers, one for each signal (assuming the use of **/no_sigs**), plus one for the output. The structural approach uses the signal registers themselves in place of the output, thus saving the extra 14 bits.

Although modifying the structural description is labour intensive, and must be repeated after each synthesis run, it is the only way of implementing bi-directional ports, or using any structural components. Future enhancements to VHDL2IC and MOODS however, could quite easily pass any structural code through to the output unaltered, making the whole operation entirely transparent.

7.4.3 Timing and Scheduling Issues

Within each of the three main designs there are several sections where a particular scheduling of operations is required. These tend to be associated with I/O port signal timing, where the relationship between various output transitions is of utmost importance. For example, when loading values into the FFT processor, the controller must ensure each input is stable for at least 2 cycles after the *load* edge (see the timing diagram in Figure 7.15). This delay is guaranteed in the controller by inserting two ICODE PROTECT instructions (see sections 5.3.1 and 6.2.2) immediately after the output assignments (Listing C.4, lines 309-320). Note that the PROTECT is actually added using the VHDL2IC library *protect* procedure, immediately following the *wait for 100ns*. At the time of development, the automatic insertion of PROTECTs in *wait* statements (described in section 6.2.2) had not been implemented, however a brief look at the source listings will show that had it been present, only a few of the manually added *protect* calls would be necessary.

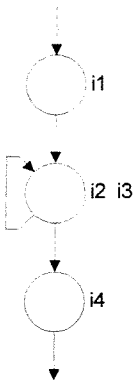
Another particularly useful technique developed for the video *raster* process is the synthesis of "zero-delay loops". Consider the delay optimisation of the simple loop shown in Figure 7.20a. VHDL2IC implements *for* loops using a special counter register combining increment and range test in a single COUNT instruction, where the end of loop test is performed before the increment. Thus in the ICODE of Figure 7.20a, *i3* tests loop variable *i* against the number 3, and if equal, jumps immediately out of the loop to *i4* (following the true activation list, ACTT). If the test is false however, *i* is incremented and control passes back to *i2* (via the false activation list, ACTF). As can be seen in the

optimised control graph, all this occurs in a single control state, thus the entire loop only occupies a single cycle for the loop body, the increment and terminate mechanics may all be considered to take zero delay, hence the term zero-delay loop. Note that throughout this section, deferred signal assignment is ignored, however it will not generally alter the results described.

To implement loops with a non-constant termination state, or an iteration function more complex that just adding 1, a *while* loop must be used. Figure 7.20b shows an equivalent loop to Figure 7.20a implemented using *while*. Here, the resulting optimised schedule does not increment in the same state as the loop body, each iteration instead requiring 2 states, hence halving the *vclk* rate. The reason for this is that the terminate condition (*i2*) is evaluated at the head of the loop, before both the loop body and the increment, thus an extra state must be used to update the *i* register value, before testing at the start of the next iteration. This problem (if indeed it is a problem for a particular situation) can be solved by using the newly implemented VHDL *exit* statement (see section 6.1.6) as shown in Figure 7.20c. The fundamental difference in this case is that the termination condition is evaluated after the loop body but before the increment, thus enabling all the operations to be scheduled concurrently. The final schedule is identical to Figure 7.20a, again implementing a zero-delay loop. Notice how the *exit* and increment effectively form a COUNT instruction as the increment operation (*i5*) is conditional upon the *exit* condition (*temp1*). Also, in Figure 7.20b, *i* must exceed the value of *end_loop* before termination, and may therefore need to be one bit wider, whereas in Figure 7.20c, termination is based on an = condition, with no need for an extra bit.

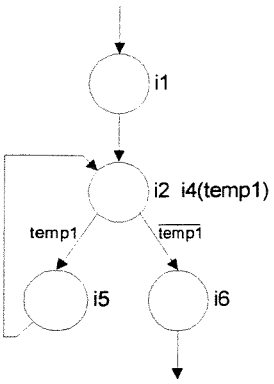
A more complex situation arises when nested loops are used, such as in the four main *raster* loops in **VidProc**. Even using simple *for* loops can create problems, as illustrated in Figure 7.21a. The result of nesting is that while both loops terminate/increment concurrently, at the end of the inner loop, an extra cycle is required for its re-initialisation on the next outer loop iteration. This is potentially more serious than the previous examples as it results in the uneven output waveform shown in the figure, which in the case of the video *vclk* signal, is unacceptable as it will corrupt the display.

```
for i in 0 to 3 loop
  vclk <= NOT vclk;
  wait for 100 ns;
end loop;


i1: MOVE  #0, i
i2: NOT   vclk, vclk
i3: COUNT i, #3        ACTT i4 ACTF i2
i4: ...
```



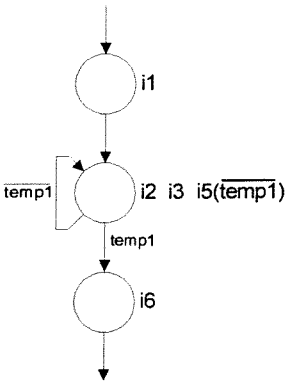
a) Simple optimised *for* loop

```
i := 0;
while i <= end_loop loop
  vclk <= NOT vclk;
  wait for 100 ns;
  i := i + 1;
end loop;


i1: MOVE #0, i
i2: LE   i, end_loop, temp1
i3: IF   temp1         ACTT i4 ACTF i6
i4: NOT  vclk, vclk
i5: PLUS i, #1, i      ACT i2
i6: ...
```



b) 2-state optimised *while* loop

```
i := 0;
loop
  vclk <= NOT vclk;
  wait for 100 ns;
  exit when i = end_loop;
  i := i + 1;
end loop;


i1: MOVE #0, i
i2: NOT  vclk, vclk
i3: EQ   i, end_loop, temp1
i4: IF   temp1         ACTT i6 ACTF i5
i5: PLUS i, #1, i      ACT i2
i6: ...
```

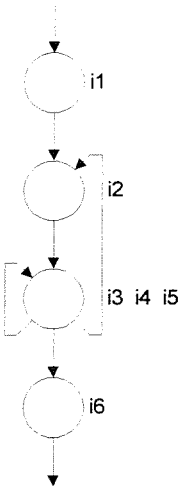


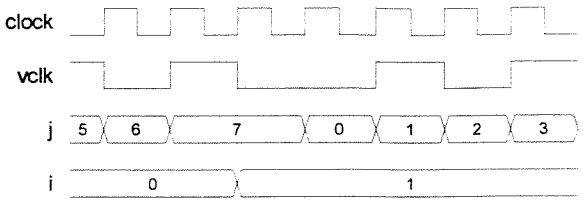
c) Zero-delay optimised *while* loop

Figure 7.20 Examples of zero-delay loops

```
for i in 0 to 3 loop
  for j in 0 to 7 loop
    vclk <= NOT vclk;
    wait for 100 ns;
  end loop;
end loop;
```

```
i1: MOVE  #0, i
i2: MOVE  #0, j
i3: NOT   vclk, vclk
i4: COUNT j, #7        ACTT i5 ACTF i3
i5: COUNT i, #3        ACTT i6 ACTF i2
i6: ...
```

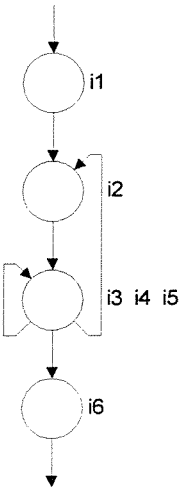


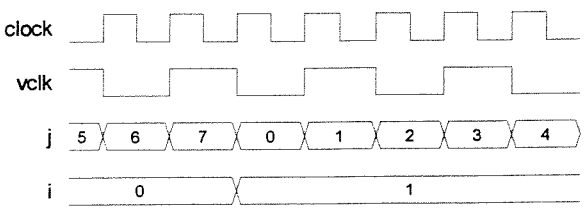


a) Optimised nested *for* loops

```
i := 0;
j := 0;
loop
  loop
    vclk <= NOT vclk;
    wait for 100 ns;
    exit when j = 7;
    j := j + 1;
  end loop;
  exit when i = 3;
  i := i + 1;
  j := 0;
end loop;
```

```
i1: MOVE  #0, i
i2: MOVE  #0, j
i3: NOT   vclk, vclk
i4: EQ    j, #7, temp1
i5: IF    temp1        ACTT i7  ACTF i6
i6: PLUS  j, #1, j     ACT  i3
i7: EQ    i, #3, temp2
i8: IF    temp2        ACTT i11 ACTF i9
i9: PLUS  i, #1, i
i10: MOVE j, #0        ACT  i3
i11: ...
```





b) Optimised nested zero-delay *while* loops

Figure 7.21 Examples of nested zero-delay loops

A solution to the problem again exploits the *exit* statement as shown in Figure 7.21b. The principle here is that the re-initialisation of the inner loop is performed at the same point as the outer loop increment, effectively moving all control code to the bottom of the loop. Optimising this description will concurrently schedule all the update instructions, as well as the inner loop initialisation and main body, resulting in a single cycle, zero-delay loop as shown. The resulting *vclk* output waveform is now perfectly regular.

Both PROTECT and zero-delay loops are used to great effect in the **VidProc** *raster* process, enabling a complex series of operations to be exactly scheduled, with full control over both the relative and absolute timing. This is essential for generating the highly specific output waveforms required to drive the video display. The final partitioning of this process into control states is illustrated in the original flow chart of Figure 7.9 and the timing diagram in Figure 7.8.

7.4.4 Communication and Synchronisation

The final techniques to be considered here are also of great importance in the video processor. Chapter 6 discusses several differences between a behavioural VHDL simulation, and its synthesised implementation. These generally revolve around the way in which MOODS uses several states to accomplish operations occurring instantaneously during simulation.

This is particularly important in relation to the timing of transitions on signals and ports that are detected using a *wait on* sensitivity list, where the fundamental problem, as discussed in section 6.2.2, is that edges can be lost if they occur at certain points, such as between a failed condition clause evaluation and re-entry to the sensitivity list detection loop (as discussed in Chapter 6). While this may not be a problem in some cases (such as when detecting button presses), if a wait statement is used to synchronise two processes, or communicate between two separate devices, a single missed edge can result in deadlock, with both parts waiting for the other to make a move. These types of problem often only show up during simulation of the synthesised structural implementation, and even then, unless all possible situations are considered, there is no guarantee that the design will work reliably in service.

The general approach taken to this problem in each of the spectrum analyser components, is to use level, instead of edge detection, and design synchronisation so that any missed signals will not result in deadlock, just a few erroneous results. Note that this last feature is intended as a fail-safe mechanism; with careful design, it should not be necessary.

By far the most complex part of the project is the video processor, in particular the inter-process synchronisation and command execution mechanisms. Taking the communication between *memory* and *render* processes as a case in point, whenever *render* wants a frame buffer memory location to be modified, it first sets up the required address and byte mask value in the *render_addr* port, and *render_val* signal. Note that these two are only ever written to by this process, thus there is no concern over concurrent signal access. *Render* then requests that *memory* performs the operation by inverting the *render_sem* signal (Listing C.3, line 407), and waiting until *memory* signals completion via an inversion of *render_sem_ack* (lines 408-410), at which point *render* continues.

Rather than using a sensitivity list to detect a request or acknowledge edge, each process maintains a local variable copy of the communicating signal generated by its counterpart. Thus *render_sem* (controlled by *render*) is paired with *render_sem_val* (local to *memory*), and *render_sem_ack* (controlled by *memory*) with *render_sem_ack_val* (in *render*). These variables track the next synchronising value of their paired signal, and instead of waiting for an edge, detection is based on matching the variable and signal value. For the *render* process, this means that the initial *render_sem* request can be made without worrying about whether *memory* is in the middle of another operation, as when it finally comes to check the signal (line 122), the level will match *render_sem_val* and the two processes will synchronise. The same method is used when *memory* reaches the end of the operation and signals to *render* that it is complete, at which point both processes invert their local copies (lines 161 and 411), ready for the next time around. The timing of these operations is illustrated in the original **VidProc** simulation of Figure 7.8, which shows how a single *render* request fits in with the generation of video signals by *raster*.

The same mechanism is also used when *raster* tells *memory* that it has just read the current pixel byte from *memory_val,* and it should now retrieve the next byte from the frame buffer (lines 255-257). Here again, *memory* may be processing a *render* mask operation when this request is made and so would miss an edge, but not in this case as the change will be detected once the current operation completes. The only proviso is that *memory*

must guarantee that the maximum time required to execute a single mask operation is less than the time between *raster_sem* edges. Even if this is not always the case, and a *raster_sem* request is missed, the system will not deadlock but just miss out a couple of pixel blocks. Of course the display will get corrupted, but at least there will be some visual indication of the error, rather than just a blank screen. Needless to say, in the actual hardware, this never occurs as *memory* can actually perform three whole mask operations per *raster* request.

Level detection is also used for most inter-device communication such as the video processor/controller *execute/ack* handshake, or the FFT processor/controller *load/ready* handshake. This is particularly relevant for video commands, as the input data only needs to be stable for a short time (while *ack* is high), after which, although the processor is busy rendering to the frame buffer, a new command can be set up on the inputs, along with the *execute* request. **VidProc** will then respond as and when it can, without further involvement by the controller.

One consequence of using edge detection is that simple wait statements are no longer suitable, instead requiring either a surrounding conditional block:

```
if execute = '0' then
  wait until execute = '1';
end if;
```

Or, as is used in the spectrum analyser designs, a *while* loop:

```
while execute = '0' loop
  wait for 100 ns;
end loop;
```

The reason for this rather inelegant construct is that a single *wait until* statement will only terminate when the condition changes from false to true, even if it is true on entry. This is no good for these designs as the condition may change before it is checked, in which case the detection statement should immediately terminate, hence the use of an *if* or *while*.

7.5 Results and Performance

Once all the various components had been simulated, synthesised, laid out on the FPGAs, and the whole system simulated at the gate level (based on timing models generated by the

FPGA placement/router), the complete spectrum analyser could then be built, as shown in Figure 7.2. Examples of the unit in action, can be found in the photographs of Figure 7.22 and Figure 7.23, which demonstrate the device in full working order. Looking closely at the screen displays, it can be seen that Figure 7.22 is showing the spectrum of a pure 720Hz sine wave input with the unit in "hold" mode, while in Figure 7.23, a 400Hz square wave is displayed, broken up into its constituent harmonics.

Design		Area (μm^2)	FPGA utilisation	Clock (ns)
VidProc	full signal model	48.8	127%	80.0
	/no_sigs	39.9	104%	80.0
	/no_sigs and DECODES	36.7	96%	80.0
	final FPGA figures	38.4	100%	91.5
FFTCont	full signal model	41.8	109%	60.3
	/no_sigs	35.6	93%	55.1
	final FPGA figures	37.2	97%	76.5
FFTProc	full signal model	43.7	114%	52.6
	/no_sigs	36.2	94%	52.6
	final FPGA figures	35.9	93%	77.2

Table 7.2 Design size and clock rate figures

Table 7.2 details a number of interesting figures for the size and performance of the final optimised designs, before and after FPGA synthesis. A comparison of the area and clock rate reported by MOODS with the actual low-level figures from the layout tool, reveals a relatively high degree of accuracy for the area (ranging from within 0.8% to 4.4% of the actual figures), but not such realistic results for the clock rate (13% to 37% too small). These area figures are included in the area breakdown graph of Figure 7.24.

The area accuracy reflects the fact that the low-level synthesis tools perform very little logic optimisation, with each module occupying more or less the same number of FPGA cells as modelled by the MOODS module library. As is often the case, this depends to some extent on whether a design is control or data flow dominated, as the larger the proportion of complex combinational modules, the more scope there is for low-level optimisation. Since all the components in the spectrum analyser are register and control dominated, the scope for optimisation is reduced, hence the good area accuracy.

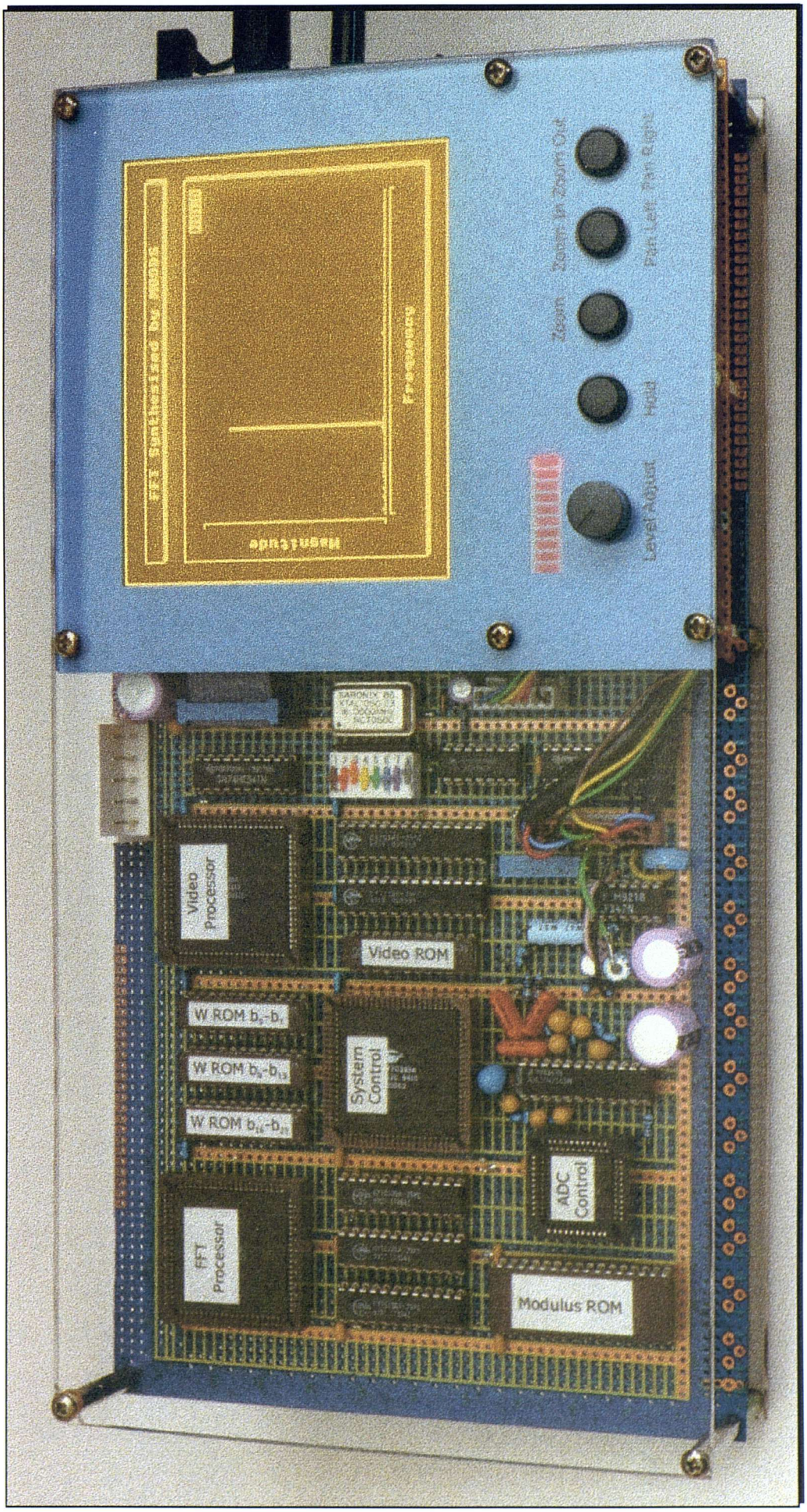


Figure 7.22 The spectrum analyser with a 720Hz sine wave input signal

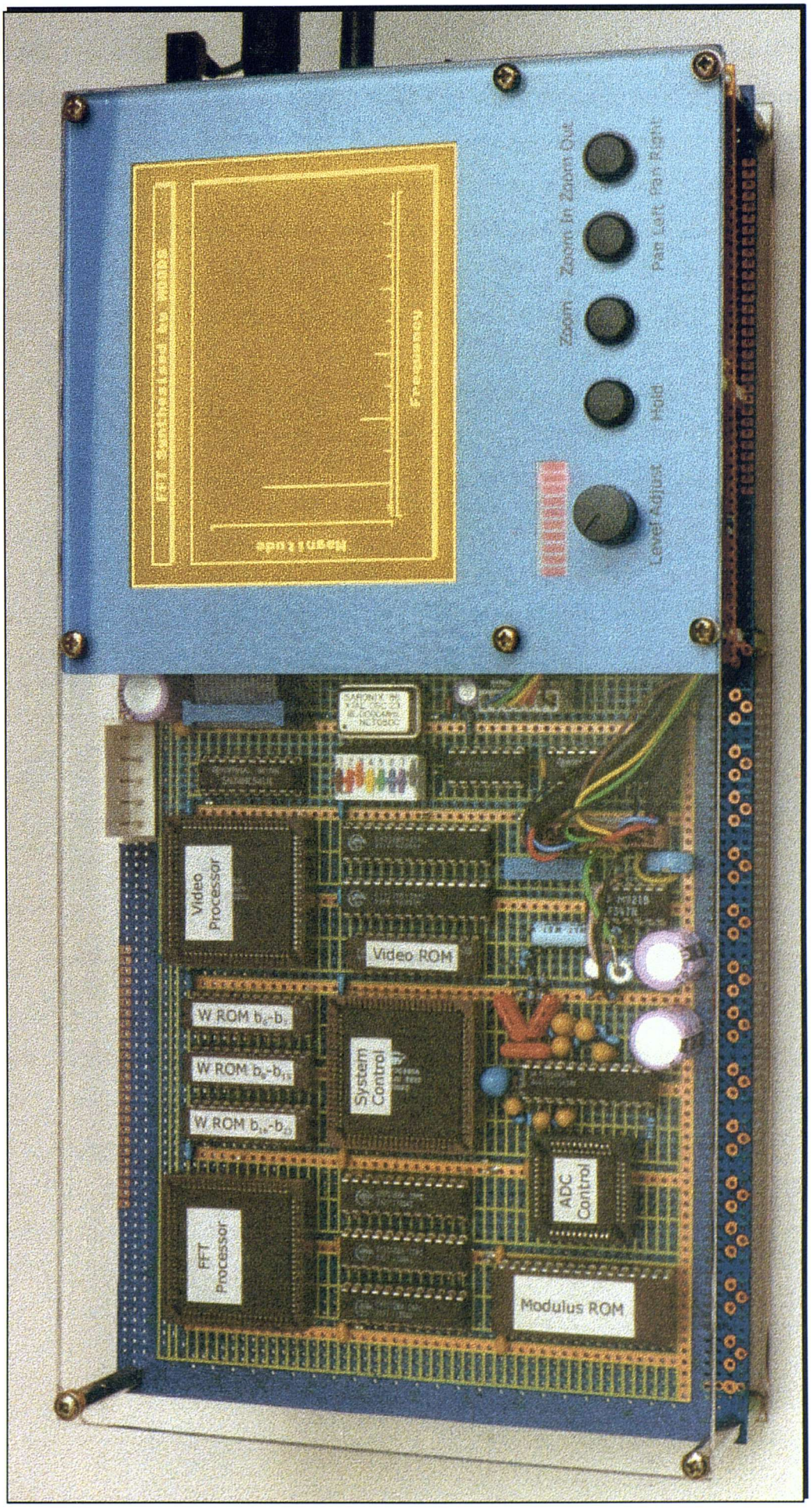


Figure 7.23 The spectrum analyser with a 400Hz square wave input signal

As far as clock rate is concerned, a brief look at the delay breakdowns provided by the layout tool reveals that the length and fan-out of interconnecting nets has a substantial effect on propagation delays. These factors are not, at present, taken into account in the FPGA module library, so it is unsurprising that the final figures do not match. In addition, the large size of the designs limits the ability of the placement tool to position communicating components close together. The result is that some nets extend almost the complete length and breadth of the device further degrading the delay.

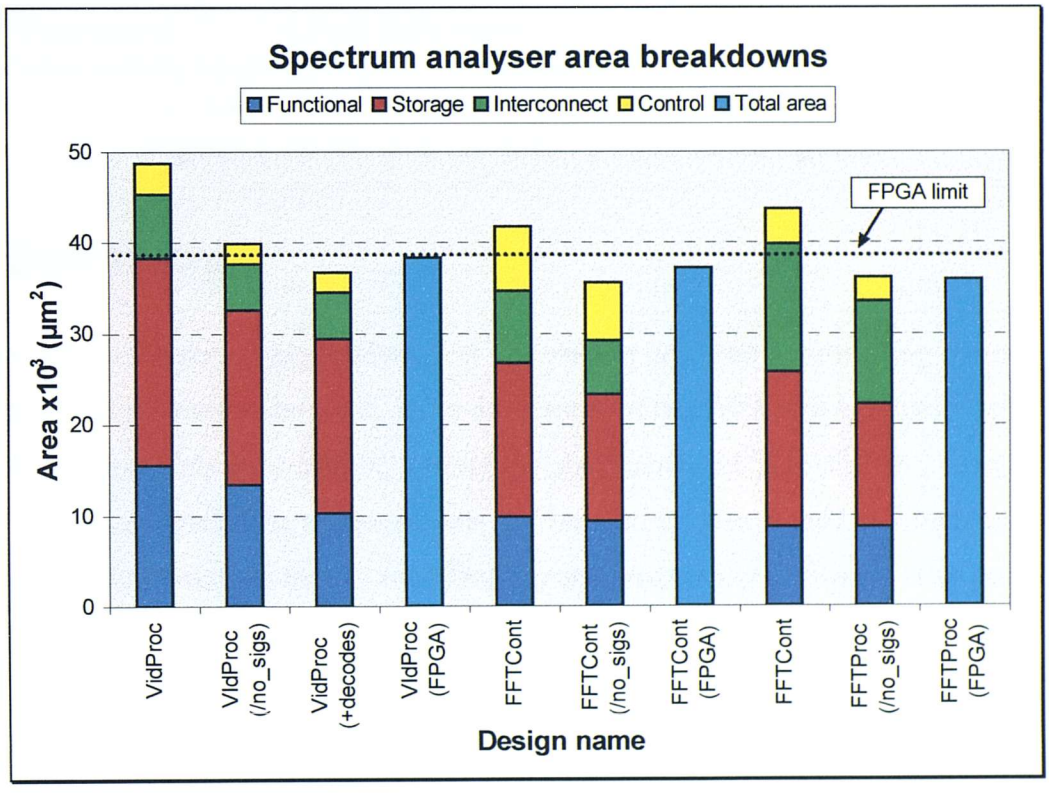


Figure 7.24 VidProc, FFTCont and FFTProc optimised area breakdowns

On a final note, Figure 7.25 details some performance specifications for the completed unit, clocked at 16MHz, which all tally with estimates predicted at the design stage. At no point during hardware construction were any major problems encountered, with everything working more or less first time. The only necessary adjustments concerned cleaning up and buffering the clock signal, and reducing the noise on the ADC voltage reference input to an acceptable level.

Spectrum Analyser Performance	
Clock rate:	16MHz
Sample rate:	clock÷512 or 1024, giving 31.25kHz or 15.625kHz switchable
Frequency display:	512 points from 0 to 15.625kHz @ 30Hz per point, or 0 to 7.8kHz @ 15.26Hz per point
Display update rate:	11Hz (ie. 11 sample/FFT/display passes per second)
Video Processor Performance	
Refresh rate:	62Hz (switchable from 72Hz down to 55Hz)
Vertical lines:	28400 pixels per video frame
Characters:	3550 characters per video frame
Clear screen:	2.8 per video frame
Video memory bandwidth required for rasterisation:	12%

Figure 7.25 Various hardware performance figures

7.6 Summary

The spectrum analyser project described in this chapter has been a resounding success from many perspectives. Above all, it has demonstrated that MOODS is actually capable of synthesising a complex system in a relatively short period of time (the total design time for the project, not including enhancements and bug fixing, was around two months), and has also provided a vehicle for the identification and eradication of bugs in all parts of the system.

Experience gained from this, and other, design exercises has helped develop a number of general techniques for writing VHDL for behavioural synthesis, and identifying danger points of particular concern. It has also suggested a wide range of possible enhancements to the system, particularly regarding the front-end compiler, and features geared specifically toward building inter-communicating systems. Some enhancements have already been added during, and after the project, and are presently being exploited in other design exercises. Chapter 8 discusses some of possible future enhancements in more detail, along with other general additions to the system.

Chapter 8

Further Work

Experience gained from using MOODS, both as a developer of additional features, and as a behavioural circuit designer, has suggested a range of possible future improvements to all parts of the system, described in this chapter.

8.1 VHDL Compiler Issues

The enhancements to VHDL2IC described in Chapter 6 provide the user with a much more complete VHDL implementation than was previously available. However, there are still a number of outstanding issues which should be addressed, with the long-term aim of supporting a greater subset of the language.

8.1.1 Improved Signal Model

With the inclusion of deferred assignment, the operation of signals at the process level has been brought into line with the VHDL simulation-cycle specification. Little attention is paid, however, to their implementation external to processes (as described in [83], see section 3.2), in particular the use of *resolution functions* to arbitrate between concurrent signal writes. This is an area that could usefully be enhanced as it is important, both for inter-process communication, and the implementation of buses and shared ports (see section 7.4.2).

Additional improvements can also be made to further enhance the signal model within processes, allowing the use of attributes (VHDL standard section 14) for detecting various conditions. DSS

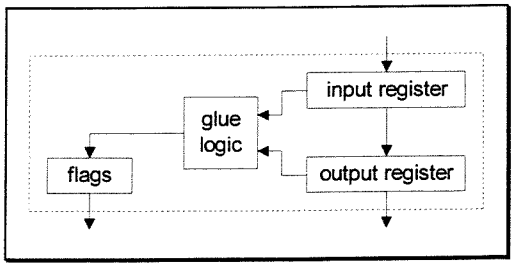


Figure 8.1 DSS signal model

[72], for example, utilises a similar deferred assignment mechanism to MOODS, and includes additional logic to generate flags for *'STABLE* and *'QUIET* attributes as illustrated in Figure 8.1, reproduced from [72]. Improvements to this structure can yield other attributes, such as the often used *'EVENT* for detecting signal edges.

8.1.2 Synchronisation and Waits

It should be clear from the discussions in Chapter 7 that the whole subject of process synchronisation and *wait* edge detection is of utmost importance when designing complex systems. One of the main concerns is the possibility of missing signal edges, either when between waits or within the structure of the wait itself, after a false condition evaluation (see section 6.2.2).

In the control model implemented in MOODS, processes do not implicitly synchronise at wait statements as per the VHDL standard, thus a handshake signal from one process, may change state while another is between waits (hence the use of levels in the spectrum analyser). The CAMAD [73] and DSS [72] systems solve this problem by forcing signal updates to occur concurrently across all processes, halting control flow at a wait until all processes are waiting. Such an approach could easily be taken in MOODS through the inclusion of a "waiting" flag, as described in section 6.1.3. The choice between this implicit synchronisation, and the more flexible current method should be left up to the user.

The problem of skipping edges while inside a wait may also be tackled in this way, however, a more general approach could involve the creation of a special "wait" module, to ensure that the operation is atomic. This has the added benefit of working with the non-deferred assignment model (VHDL2IC **/no_sigs**) and actually reduces the amount of hardware required. Figure 8.2 shows a possible configuration for detecting a signal edge and condition for a *wait on ... until ...* statement. Compared to the current equivalent (shown in Figure 6.11), this method compresses the entire operation into a single control state, thereby eliminating the possibility of a missed edge once in the wait. The key here is the use of an asynchronous structure that monitors both the output and input to a register. This is not possible at the ICODE level, which, due to its synchronous-only nature must use a "previous value" register.

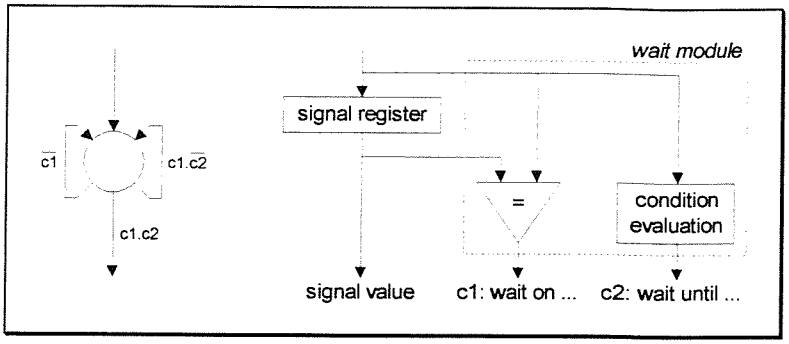


Figure 8.2 Asynchronous wait module

8.1.3 General Improvements

There are many other improvements that can usefully be made to the VHDL compiler further expanding the range of options available to the designer. These include numerous small enhancements such as the implementation of attributes, and the inclusion of a floating-point capability (an integral part of the latest VHDL standard [7]). This latter example is of particular relevance as floating-point operators tend to be rather large, and certainly do not lend themselves to a purely combinational implementation. Using expanded modules (in the form of macro operators) for these units however, should enable them to be heavily inter-optimised within MOODS, and provide for a range of different algorithmic implementations. The design of such a floating-point library is the topic of a recently embarked upon PhD [110].

Another feature high of the list of priorities is the inclusion of some degree of support for structural VHDL constructs, such as those described in section 7.4.2. At the simplest level, this need only involve the separation of structural (architecture) and behavioural (processes) portions of the source code, with all communicating signals converted to ports prior to synthesis. The two parts can then be re-combined in the synthesised structural netlist following behavioural optimisation. This is, in effect, automating the mechanism described in 7.4.2 and relies on the low-level synthesis system accepting all the necessary structural constructs. A more comprehensive approach that does not require the layout system to understand VHDL is depicted in Figure 8.3. Here, the two portions are separated at the input stage as before, however the structure is converted into a netlist comprising the same technology-dependent modules (and maybe even expanded modules) utilised by MOODS. The resulting structural descriptions are then combined in a final linking stage ready for placement and routing.

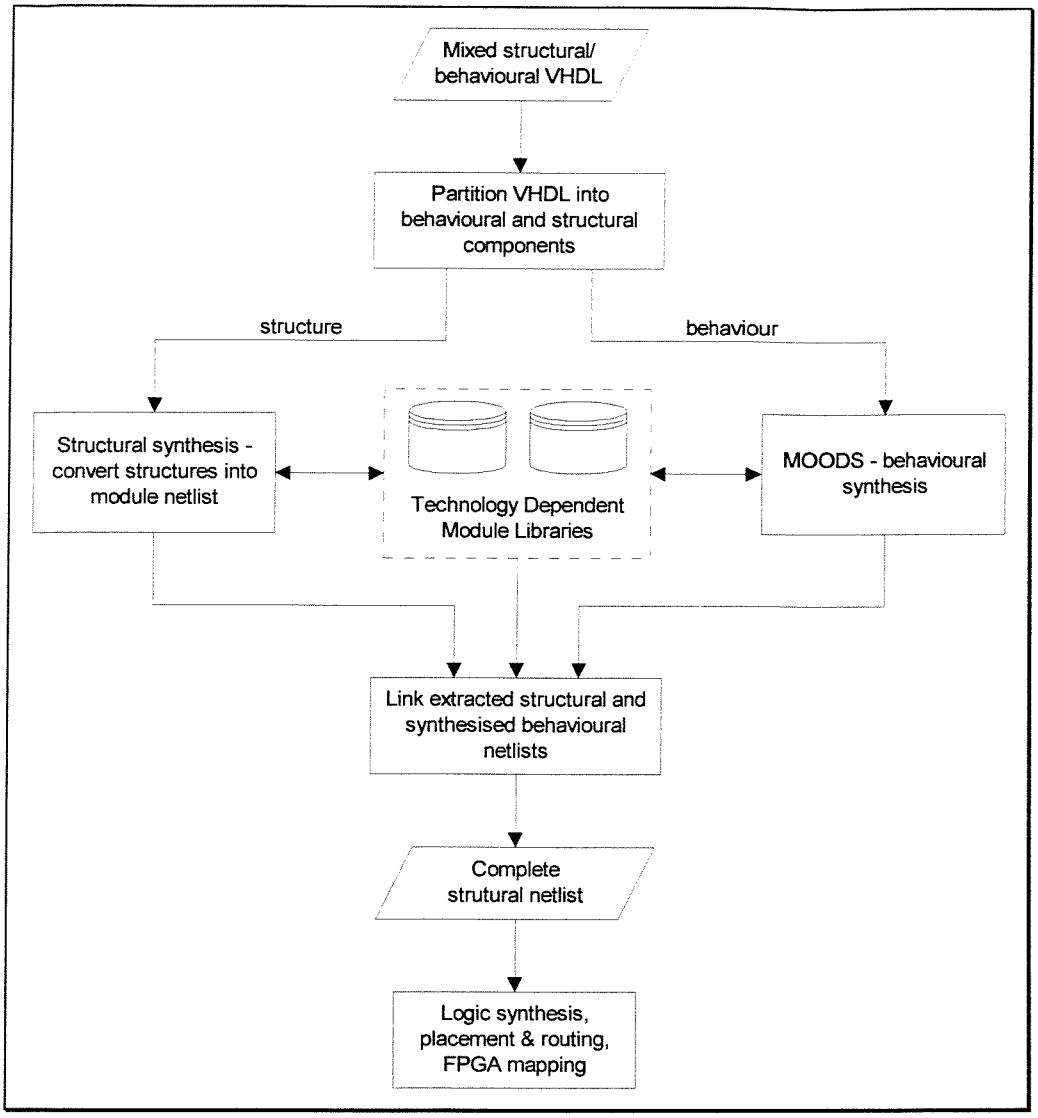


Figure 8.3 Combined structural and behavioural synthesis environment

8.2 MOODS Synthesis Issues

Although the previous chapter has shown that MOODS is perfectly capable of dealing with the complexities of a real system design, there are still a number of shortcomings in the present system that need to be addressed if it is to become more widely used. Many of these problems are currently being tackled in the development of a new version of the system, based on similar principles but involving significant re-engineering of the core synthesis engine. This is intended as a modular, expandable system, suitable both for further research projects, and commercial exploitation. Looking at the main enhancements in relation to the present system, a number of categories can be identified, which are discussed below.

8.2.1 Improvements to Expanded Modules

There are three main areas in which the implementation and use of expanded modules could be improved:

1. The range of expanded templates developed to date is sufficient for the benchmark and project designs so far encountered. However, they do not by any means form a comprehensive set, both in terms of the module widths and function types supported. This situation should gradually improve as more and more design projects are carried out requiring a greater variety of expanded modules. Indeed, the development of the fixed-point multiplier and memory access macro ports for the spectrum analyser project, has already begun this process.

2. One of the main reasons why the expanded module library needs regular enhancement, is that a specific template is required for each particular combination of input and output register widths. This could be simplified through the inclusion of some form of parameterisation in the template definition, thus instead of using absolute bit-widths, the internal data path of the module would be specified in terms of generic parameters. The problem with this approach is that the template design process becomes much more complex than simply writing a behavioural description and synthesising, especially if a change in parameter value requires the template structure to be modified. For example, if only the bottom 16 bits of a 16-bit multiplier result are required, not only can the output register be shortened, but also the last quarter of the operation, corresponding to the top 16 result bits, can be eliminated. This form of parameterisation would also be useful for macro ports, allowing the various setup and access delays (ie. the number of PROTECT instructions) to be varied depending on the port timings required.

3. The automated module expansion features described in Chapter 5, were primarily designed as an aid to the investigation and development of the effects of module expansion. The results in that chapter illustrate how complex the interactions between expansion and optimisation are, requiring a considerable amount of time and effort to be spent in order to obtain the best results. Ideally, some form of more automated approach would be preferable, making the decisions on where, when and how to expand, based on the target objectives specified by the user. Because expansion by itself does not actually improve the cost function however, it is very difficult to make any quantitative

evaluation of its suitability until after optimisation. One approach could be to implement an *unsplit_module* transform, which re-combines the disparate parts of a particular module after it has been expanded and partially optimised. It could then be replaced with an alternative version including, possibly, the original combinational module. This would allow module expansion to become an integral part of the simulated annealing optimisation process, where *split_module* is considered a degrading transformation, which may be undone at a later date if it proves to be unsuitable. The main problem with this is that it would almost certainly have a significant detrimental effect on the optimisation time due to the substantial effort involved in re-arranging the design structure when bringing a module back together. A more realistic approach could involve automatic analysis of the design, in order to identify the largest units and clock-speed bottlenecks. These could then be targeted for expansion either automatically, using a rule-based process, or via user interaction.

8.2.2 Module-level Issues

The improvements made to the module libraries, and additional flexibility provided by expanded modules, have substantially enlarged the region of the design space covered during optimisation. However, MOODS does impose a number of restrictions, built into the core data structures at the lowest level, which can only sensibly be addressed during a full review of the system:

- The poor match between estimated and actual delay figures encountered in section 7.5 is a direct result of simplifications in the way MOODS calculates delay external to the modules. The main problem is that the global modelling calculations performed by MOODS (see section 4.5.1) do not take account of layout issues, and were not designed with the highly restrictive, and device dependent, architecture of FPGAs in mind. What is required is a complete separation of the modelling sections, based around a modular structure that would allow various technology-dependent alternatives to be used. This mirrors the enhancements made to the standard module libraries, as discussed in section 5.3.1. Once separated, more sophisticated algorithms can be readily added, to include such factors as the length of interconnects (via a rough placement/routing stage) and the area and delay characteristics of the particular target technology.

- Part of the motivation for development of module expansion capabilities was the poor original implementation of multicycling (described in section 5.1). This stems from the fundamental assumption that all modules are purely combinational blocks, and only execute in a single control state. Any re-design of the system must include full support for multicycling, without the scheduling problems associated with the current implementation. At the same time, the delay measure associated with the execution time of a module should be enhanced to include sequential designs (with an independent internal controller), requiring several clock cycles and an associated minimum cycle time.

8.2.3 General Improvements

On top of the above problems, there are a whole range of general improvements which could be made to enhance the core operation of the system. These include modifications to the data structures and hardware model to reduce inefficiencies, and improvements to the optimisation algorithms to obtain better implementations, in less time.

The main data structures used to internally represent the design throughout the synthesis process are highly complex, and, having been developed by various individuals over a number of years, are in need of some attention. Two main areas of concern are:

1. Inefficiencies in the implementation of bit-slicing (aliases) mean that MOODS considers a write to any part of a register as a write to the whole register. Thus, even though each bit has an individual load enable, non-overlapping ranges cannot be concurrently updated. A related problem is the over use of multiplexors on aliased inputs described in 5.4.3, which can add a substantial area overhead to the final implementation.

2. The inclusion of static data in the main control and data paths means that some information is continually re-calculated, even though it never changes. Examples include the generation of instruction dependency links, and of mutually exclusive subsets. By removing these elements from the core data structures, a pre-processing analysis stage can be performed at the start of synthesis to determine all the required static information in one step.

Another key area in which improvements can be made is the optimisation algorithms, especially the heuristic one, which, in Chapter 5, has shown itself to be far from perfect. Much research has been done on all kinds of synthesis algorithms (see Chapter 2), many of which could be adapted for use in MOODS. This research has tended to concentrate on constructive, rather than iterative algorithms, however it would appear from the results in Chapter 5 that some form of hybrid approach, involving an initial heuristic, followed by simulated annealing optimisation, may produce the best results.

Enhancements to the set of transformations could also be beneficial to the optimisation process. One particular feature which would immediately pay dividends, is the inclusion of a transform to translate particular patterns in the control and data path, into a more optimal form. MOODS already does this to a limited extent by converting all multiplies and divides by a power of 2, into left and right shifts. Unfortunately, this is hard-coded into the initial data structure creation routines, making inclusion of other optimisations (eg. converting shifts to aliases), rather involved. The development of a rule-based pattern-matching transform, would allow this form of optimisation to be easily implemented. In fact, the VHDL source-level optimiser already performs similar operations, however it is only a pre-processing stage and so cannot make trade-offs based on technology-dependent module characteristics.

Finally, one of the most serious omissions is the lack of support for local timing constraints. Although the use of PROTECTs to force a desired schedule goes some way to alleviating this, it is still not possible to specify any form of absolute timing relationship between instructions. The approach taken by CAMAD [20] allows maximum and minimum delays to be specified between arbitrary points in the source code, adding extra "timing control" arcs to the control graph, which specify scheduling constraints. This would fit quite neatly into the internal design representation, and could be used as a mechanism for locking groups of instructions into a rigid schedule, in a similar vein to the behavioural templates [63] discussed in Chapter 3. The inclusion of setup and access parameters in macro ports would then simply involve setting the constraints on timing control arcs.

8.2.4 User Interface Enhancements

MOODS currently sports a relatively crude user interface. Although this has recently been augmented with a Microsoft Windows front end, it still only provides the most basic of access to the design representation and optimisation algorithms. Experience gained during the spectrum analyser project has highlighted several target areas that would benefit from further work:

- Although behavioural synthesis is intended to take much of the low-level detail out of digital design, the user will still want to know how vital parts of a system have been synthesised, especially where resources are strictly limited. The development of a graphical system for interacting with the internal design structure would allow the user to navigate around the control and data paths, examining the synthesised implementation in more detail. Other facilities could include manual application of transformations via a point-and-click style selection process, and back annotation of implementation data to the VHDL source so that, for example, selecting a particular data path node highlights the operations and variables it implements, and vice-versa.

- General improvements to the usability of the optimisation algorithms are a high priority if the system is to be used by non-expert designers. The simulated annealing algorithm requires the specification of a number of abstract control parameters (start and end temperature, step size etc.). Furthermore, the most effective configuration is highly design dependent and may also require several passes with different parameter sets. Given a certain amount of design analysis, it should be possible to automatically estimate a reasonable range of values with which to optimise. This process could be repeated over several runs and the best results returned. Ideally, the simplest form of user control would be an "effort" setting, determining how long the algorithm should run for.

- One of the most frustrating features of the simulated annealing process is the way in which the best implementation in an optimisation run is not necessarily the final one obtained. A most useful facility, therefore, would be the ability to remember the best design configuration, and return to it when optimisation is complete. This would allow the user to perform many different optimisation passes, without worrying about degradations losing a good design.

Chapter 9

Conclusion

The work described in this thesis has transformed MOODS from an interesting research system into a practical tool capable of synthesising substantial designs with reduced time and effort. It has attempted to address some of the real-world problems encountered when using a high-level synthesis system in the highly restricted environment of Field Programmable Gate Arrays, and other programmable logic devices.

The expanded module capability detailed in Chapter 5 provides MOODS with the ability to exploit hierarchically-defined sequential modules, allowing the sub-structure of a complex operator to be dynamically expanded within the bulk of the top-level design. The resulting expanded structure presents MOODS with improved opportunities for inter-module optimisation through the sharing of common sub-components between the expanded modules and the rest of the design. Improvements in the size of the smallest optimised implementation for a number of benchmark designs show anything from a 26% to 65% decrease; in some cases accompanied by a reduction in total delay and minimum clock period.

The organisation of the expanded module templates enables the creation of a technology-independent high-level function library for implementing operations not normally available in the standard low-level module set. These can include simple hand-crafted structures designed to overcome inefficiencies in a purely behavioural description, or large complex functions, such as floating-point operations, which are only feasible through the exploitation of inter-module optimisation.

Expanded modules also provide the ability to hand-optimise the local scheduling of operations within their sub-structure. They may then be expanded after the main optimisation process to facilitate the use of a fixed-timing I/O sequence, such as for

external memory access. This goes some way to providing a high-level interface library to encapsulate complex I/O protocols in a single behavioural-level function call.

Enhancements made to the VHDL compiler front-end have enabled a much greater subset of the language to be utilised, along with a reduction in the restrictions imposed on the required design style. All the additions have been made taking full account of the behavioural effects of the VHDL simulation cycle, however this has been shown to result in a typical area and delay overhead in real-world designs of around 20%. Facilities are therefore provided for a moderate relaxation of the VHDL model in situations where this overhead is unacceptable. Full integration with the expanded modules is also included, through the use of special *macro operator* and *macro port* functions, which provide the user with transparent access to the high-level expanded module library.

All the above features are brought together in the design and construction of a real-time spectrum analyser, intended as a test-bed upon which to investigate and develop the integration of MOODS into a complete behavioural design flow. This illustrates many useful techniques geared to developing complex and reliable systems within the MOODS environment, and represents the first of a whole range of design projects that are now regularly exercising the capabilities of the system.

In conclusion, the enhancements made to the MOODS synthesis system have enabled it to become a useful tool for the design of small to medium-sized circuits. However, there is still much work to be done on improvements both to the core code, and the oft-neglected user interface. The development of a second-generation system based on the principles developed in MOODS, should help achieve the ultimate goal of a truly user-friendly behavioural synthesis environment, ideal for the rapid development of digital systems.

Appendix A

Papers

The two papers contained in this appendix, "On-Line Testing of Statically and Dynamically Scheduled Synthesized Systems" [108] published in IEEE Transactions on Computer-Aided Design, and "On-Line Test of Synthesized Designs Exploiting Latency Analysis" [109] currently in preparation, detail associated work that is not contained elsewhere in this thesis. These papers describe studies into the possibility of adding an on-line test capability to a synthesised system, exploiting the time during which a unit is normally inactive to apply a set of tests, thus providing a continuously updated indication of the health of the system.

IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 16, NO. 1, JANUARY 1997 47

On-Line Testing of Statically and Dynamically Scheduled Synthesized Systems

Andrew D. Brown, *Senior Member, IEEE,* Keith R. Baker, and Alan J. C. Williams

Abstract—Most digital systems at some time during use have areas (modules) that are "dead" in the sense that they do not contain valid data, i.e., the data that was processed or generated by that area has been passed on to a subsequent stage and will not be required (read) again. In a synthesized system, where the flow of data is determined explicitly by an on-chip (synthesized) controller, the question of which areas will be dead or not (and when) is known in advance. There are areas and times when the "use" is data-dependent, but then the use is known to the controller at that time. This deadtime can be exploited to run a test pattern (either complete or in part) through the unused area, thereby giving the ability to continuously monitor the "health" of the overall system with very little (sometimes zero) impact on the processing capability. This has obvious applications in situations where reliability is a concern. There exist systems where an area is so heavily used that it is impossible to perform any testing at a serious rate; in this case the area may either be partially tested (or tested at a lower rate) or the processing of "real" data periodically halted to allow a more thorough test to take place with concomitant throughput degradation.

This paper describes a behavioral synthesis system that can detect and exploit dead areas for automatic testing. Pertinent aspects of the controller are described, and a number of dead area statistics (including "test throughput") generated from real designs are reported.

Index Terms—Synthesis, test.

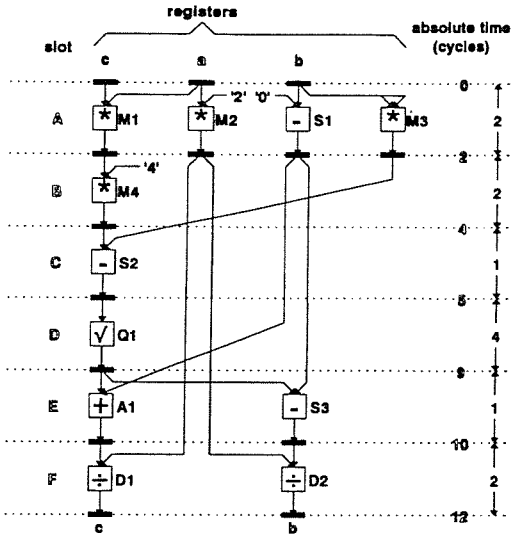


Fig. 1. The possible dataflow representation of a quadratic equation solver.

I. Introduction

THE COMPLEXITY increase seen by ASICS over the last few years is comparable to that experienced by very large scale integration (VLSI) a decade ago. This increase is not only due to advances in processing technology but also to the use of DA tools enabling predefined cells to be "glued" together. The use of synthesis tools has brought an added dimension to the complexity of digital systems: the internal architecture of synthesized circuits is largely opaque to the designer.

The testing of these increasingly complex systems is a vital part of the design process, and issues of testability require consideration at all levels. Techniques such as hierarchical testing, partial scan paths, and the analysis of high-level descriptions to determine hard to detect (HTD) faults and their correction through test statement insertion can help to enhance the testability of designs early in the design process [1], [2].

Manuscript received June 7, 1995; revised October 15, 1996. This work was supported in part by EPSRC Grant Number GR/K00752. This paper was recommended by Associate Editor, R. Camposano.

The authors are with the Design Automation Group, Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, Hants., U.K.

Publisher Item Identifier S 0278-0070(97)01277-3.

Other relevant work in the area incorporates scan and partial scan paths into the overall penalty function calculations [3], efficient transforms for the inclusion of area-efficient scan paths [4], [5], and speed-efficient testing regimes [6].

For critical systems, the testing problem becomes an even greater design issue. To ensure their continual successful operation, critical systems require frequent testing and/or the use of redundancy at the component or system level. The testing of a system conventionally requires part or all of the system to be temporarily taken out of service while a built-in self-test (BIST) or external test procedure takes place. Even though methods such as adding deflection operations to the control and dataflow graph to reduce scan paths [4] improve out-of-service testing, this approach is usually unsatisfactory where the system downtime can jeopardize the function of the system.

The approach described in this paper utilizes the idle time of units in a synthesized synchronous system in a way that enables individual units to be tested while the system remains in operation. This approach provides an almost continuous (discrete, granular) indication of the condition of the sys-

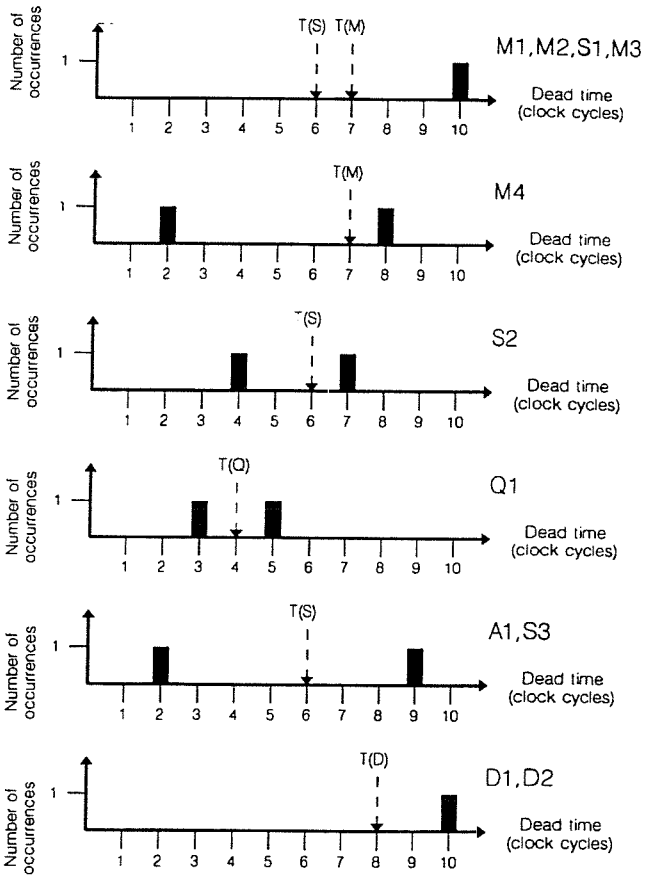


Fig. 2. Dead area profiles derived from Fig. 1.

tem without the overhead of excessive redundancy or the inconvenience of taking the system out of service.

The purpose of this paper is to describe and evaluate only the testing *infrastructure*. The nature and derivation of the test vectors and coverage of the tests is beyond the scope of this paper, but it should be noted in passing that a high stuck-at-fault coverage does not necessarily imply a corresponding defect coverage. The techniques described here are not limited to tests based on the stuck-at model. They apply equally well to functional and exhaustive tests.

Fault *tolerant* synthesis [7], [8] introduces orthogonal complexity into the issue and is not considered here.

II. The Synthesis System

The synthesis system, on which this project is based, is described in detail elsewhere [9], [10]. It is a high-level behavioral synthesis system that optimizes a design with respect to a combination of user specified constraints on silicon area and speed. The optimization is performed by applying a series of local reversible transformations (similar to those used in other systems such as CAMAD [11]) to the control and datapath graphs under the general guidance of a simulated annealing algorithm[1] [12], [13]. The design space has as many dimensions as the user optimization criteria; for example, area, speed, power dissipation, and "testability." The space, *defined* by the user-supplied behavioral description, is discrete, degenerate, and highly irregular. Steepest descent and related algorithms perform extremely badly on problems of this nature, notwithstanding the observation that the concept of adjacency in a discrete space is poorly defined. Details of the algorithm, cooling schedule, and termination criteria may be found in [9] and [10].

[1] Recently, an optimization technique using "randomized branch and bound steepest descent" has been reported [3]. This may be considered similar to a simulated annealing algorithm with a position-sensitive bias on the random number generator.

TABLE I
PROPERTIES OF REAL CIRCUITS FOR DEAD AREA ANALYSIS

Circuit	Name	Description	Instruction count[1]	Optimised for	Functional units	Virtual functional units	Area (μ^2)	Clock cycles[2]	Time (ns)
A	CMPLX	Complex number ALU	21	speed	8	17	21k	4	894
				area	4	17	17k	7	2162
B	CHIP2	Boolean optimiser	632	speed	229	275	1.5M	58	6820
				area	180	275	1.5M	75	12682
C	CBM2	Cascaded Booth multiplier	48	speed	38	47	19k	18	786
				area	36	47	14k	17	833
D	PIP	Polygon inline processor	123	speed	68[3]	68	28k	43	1737
				area	68	68	29k	43	1887

Footnotes:

1: I-code instructions

2: For these implementations, the clock cycle length is controlled by the length of the longest operation within the unit

3: This circuit is composed solely of small units:
53 == comparators
4 <= comparators
8 AND gates
2 adders
1 != comparator

The input design is specified at a behavioral level using VHDL, and the design is implemented as a datapath plus controller architecture which is output as a netlist of parameterizable cells in structural VHDL. A variety of implementations can be generated from a single design specification by varying the target optimization criteria.

III. DEAD AREA ANALYSIS

Accepting that a certain temporal "slack" may now be acceptable in a synthesized design to allow time for testing somewhat warps the penalty function values associated with each optimization transform. For example, without testing, in a design having two adders ($A1$, $A2$) only used in separate temporal intervals, it makes sense to multiplex access to a single physical adder [$A1 \equiv A2$]. The overhead is simply the cost of the multiplexers. Including the dimension of testability into the design space complicates the situation, as using two physical adders means that the first can more easily be tested in its idle or *dead* time while the second is in use. Note that this is not simply device redundancy. The overhead is now

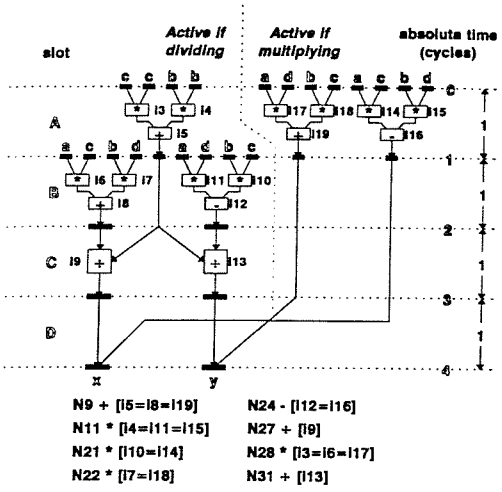


Fig. 3. Datapath of circuit A, optimized for speed.

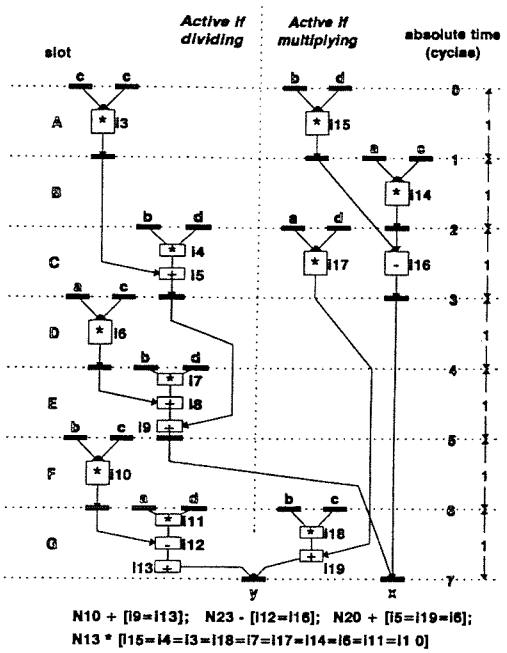


Fig. 4. Datapath of circuit A, optimized for area (compare with Fig. 3).

the extra adder, but the multiplexer has gone, and the test objective is more easily met.

3.1 An Example

Consider the rather contrived example of Fig. 1: the design takes data input from three registers a, b, and c and produces the roots to the quadratic equation $ax^2 + bx + c = 0$, with the results appearing in registers b and c. The figure shows a possible dataflow implementation of the system at some early point in the synthesis process. The vertical axis indicates processor slots which are not in general of equal length and each column in this example is a common register element.

3.1.1 The Dead Area Profile: From the dataflow graph of Fig. 1, we can trivially construct the *area activity intervals:* $M1$; $M2$; S; $M3$: (0–2); $M4$: (2–4); $S2$: (4–5); $Q1$: (5–9); $A1$; $S3$: (9–10); $D1$; $D2$: (10–12). These simply show the *times* at which various modules in the design are in use. We can then construct the *dead area profile* for all the operators in the design. This may be likened to the inverse of the lifetime of a variable, but applied to operators rather than variables. Assuming that all the operator instances in Fig. 1 are mapped onto different physical devices, we get the profiles shown in Fig. 2. These show the distribution of *time* in which each particular device is dead (idle) and can therefore be tested. For example, $Q1$ is unused for slots A, B, and C (i.e., the first $2 + 2 + 1$ clock cycles) and slots E and F (i.e., the last $1 + 2$ clock cycles). The profile, therefore, shows one dead area of five cycles in length and one of three. If we associate with each type of operator a *testing threshold, T()*, the number of clock cycles required to test the operator, we are in a position to assess how well this particular design meets the "testability" criteria set by the user. Looking at the profiles of Fig. 2, we see that *every* instance of *every* operator has at least one dead area above the appropriate threshold, and so the design may be considered completely testable in the sense that we have the *time* to test each operator at least once inside the normal execution of the synthesized design.

Applying a "conventional" transformation allows us to see how this new dimension interacts with the existing two.

If we move operator $M3$ to slot B, we can map ($S1$, $S2$, and $S3$), ($M1$ and $M4$), and ($M2$ and $M3$) onto the same physical operators: $[S1 \equiv S2 \equiv S3]$, $[M1 \equiv M4]$, and $[M2 \equiv M3]$, respectively. The profiles change as a consequence of this transformation. The maximum of $[S1 \equiv S2 \equiv S3]$ now falls below the $T(S)$ threshold, and so there are insufficient contiguous clock cycles to test the operator during the normal course of operation of the design. This, of course, does not necessarily cause rejection of the design. It may be possible to perform a number of partial subtests or to increase the overall execution costs in order to get the test in, or indeed, to omit the test altogether, if the "testing" criteria is weaker than other constraints. Note that $[M1 \equiv M4]$ and $[M2 \equiv M3]$ still have profile components above $T(M)$.

3.2 Design Statistics

This section contains a discussion of the dead area analysis of a number of real circuits. Complexity precludes schematics, but the salient details of each circuit are given in Table I.

TABLE II
TEST VECTORS REQUIRED BY THE UNITS USED IN THE DESIGNS OF TABLE I

	Number of bits	Vectors	% coverage	Faults covered	Missed	Used in design
Adder	9	9	98.2	166	3	C,D
	10	10	100	270	0	B,D
	16	16	100	316	0	A
	32	55	98.4	636	10	B
Subtractor	10	10	98.5	202	3	B
	16	17	100	328	0	A
	32	22	99.6	668	3	B
Comparator	1	3	100	10	0	D
	3	6	100	26	0	B
	6	11	100	58	0	D
	9	12	100	92	0	D
	10	14	100	104	0	B,C
	32	35	98.5	324	5	B
Up counter	8	23	98.7	238	3	D
Multiplier	8	58	89.6	5945	693	A,B

During each clock cycle, the controller activates a set of datapath units, thereby implementing instructions in the original high-level description. At the end of each clock cycle, the instruction results are loaded into registers. The "register load" instructions give the data from which the necessary statistics and results are derived.

3.2.1 Deterministic Dataflow: Circuit *A* contains 21 instructions and operates as either a complex number divider or multiplier depending on an external signal. The datapath is thus configurable. For either operation a subset of the datapath units will be active. Fig. 3 shows the datapath of circuit *A*, optimized for speed, with no area constraints. The controller path length is four cycles, and the corresponding dead area profiles for each unit type are obvious from the figure. The profile for each unit changes depending on whether the circuit is configured for multiplying or dividing. The profiles for the multiply and divide unit types (these may be trivially derived from Fig. 3) depict more than one physical unit where individual dead areas are accumulated. Ideally the profile for each individual unit should be separately shown. However, in this case the deadtimes for similar unit types are equivalent.

Fig. 4 represents the same design, where the design has been optimized for area. For each operator type, the units of Fig. 3 have been merged into single units to conserve area. Here, to accommodate unit sharing the controller path length has increased to seven cycles.

Each unit may have one or more dead areas. In order for a unit to be completely tested, the *longest* dead area is the most significant. The longest dead area profile is constructed using only the longest deadtime for each physical unit. Although the cumulative profiles lose some information about individual units, they give an overall *indication* for the testable state of the design and are useful for illustration within this paper.

To test the 16-bit adder used in circuit *A* takes 16 test vectors [16]. A summary of the number of test vectors required to test some of the units occurring within the designs of Table I is given in Table II. For a complete test of these units, assuming

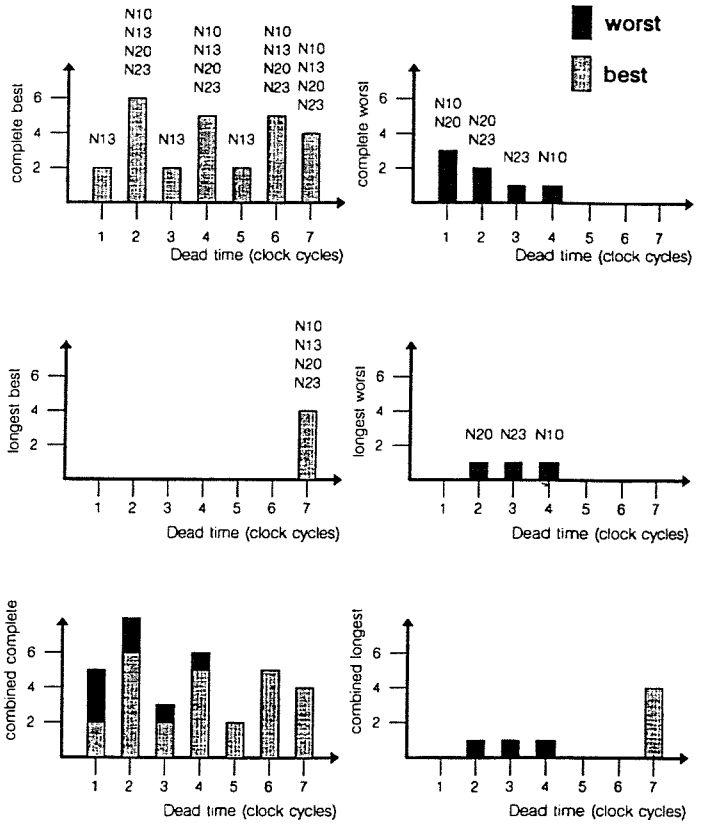


Fig. 5. Probabilistically derived dead area profiles for all functional units in circuit A, optimized for area.

that the application of each test vector takes one clock cycle, 16 dead cycles are required for the adder. The dead area profile shows there is insufficient deadtime to completely test a unit in one dead area or even one controller cycle. Within the framework outlined, therefore, circuit A cannot be tested properly without incurring a speed penalty, which is the extra cycles required to complete the test.

3.2.2 Nondeterministic (Probabilistic) Dataflow: We have so far looked only at dataflow graphs where activation of units is deterministic. However, in the majority of synthesized designs, a significant fraction of the dead area profiles will be data-dependent, a consequence of conditionals in the high-level description. This leads to circuits containing units that do not have clearly defined activities for nondeterministic operations. Without knowledge of the exact data that will be passing through the system, we cannot precisely calculate the dead area times. However, we can make some assumptions and derive best/worst case times and corresponding profiles between which the actual profiles will fall.

The *worst* case dead area times are calculated assuming that a unit always executes all of its assigned instructions, whereas the *best* case dead area times are calculated by assuming that only *one* of a units probabilistically executed instructions is executed each controller cycle. This results in $n + 1$ different dead area profiles where n is the number of possible instructions: one for each probabilistically executed instruction, and one where none are executed. For example, the adder (N20) of Fig. 4 has a worst case dead area of two where each add operation is assumed to be executed. The best case dead area times where each instruction is assumed probabilistic are two, and four for the first instruction [i5], four and two for the second instruction [i8], six for the third [i19], and seven for none. The best and worst dead area times for all units of circuit A, their complete and longest dead area profiles, and combined best/worst profiles are shown in Fig. 5 for the area optimized design of Fig. 4. Comparing the actual dead area profiles corresponding to Fig. 4 with the best and worst case profiles of Fig. 5 illustrates that the real profiles lie between, and are similar to, the combined best/worst profiles.

3.2.3 Further Statistics: The testability situation eases with increasing design size and varies with design data or control domain dominance. For a 32-bit subtracter within circuit B,

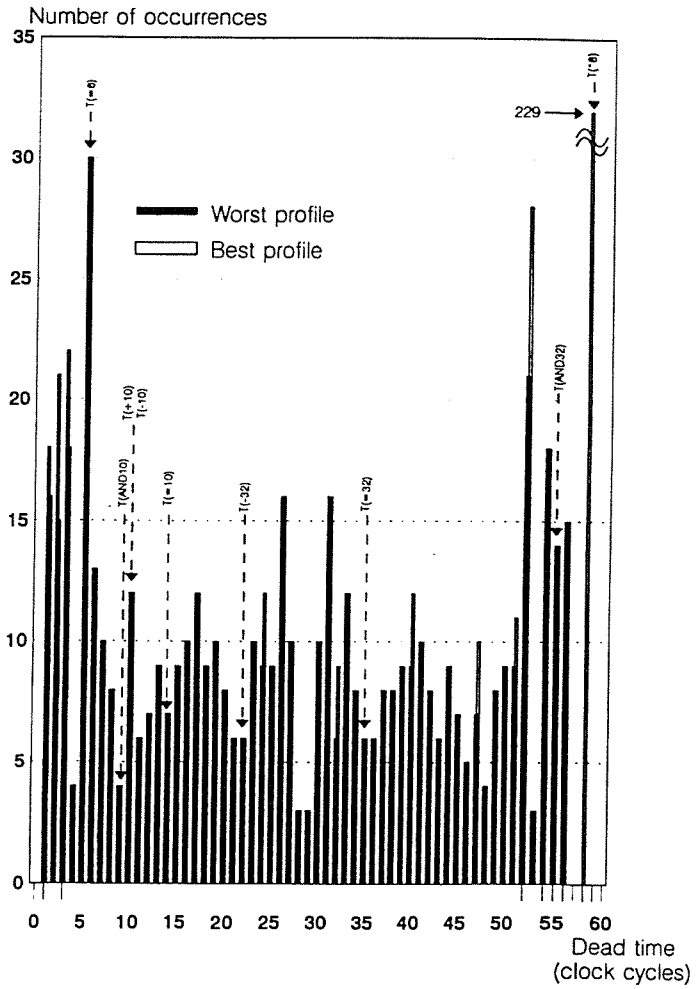


Fig. 6. Complete dead area profile for design B, optimized for speed.

optimized for speed, the controller cycle time is 58 clock cycles, giving dead areas of six, two, six, ten, and 30 cycles. To completely test the 32-bit subtracter 22 test vectors are required [16], so here it is possible to test the unit within a single dead area, i.e., within a single controller cycle with no overall speed overhead.

In addition to increasing dead area with design size, the number of test vectors required to test a unit for 100% coverage increases (almost) logarithmically with increasing bit width. This *increases* the possibility of performing a test within a given dead area without incurring a speed penalty.

Each physical unit in a design gives rise to both long and short deadtimes due to the relatively small number of instructions executed by each unit compared to the length of the controller cycle. This can be seen by the greater number of dead areas at both small and large values. In the larger design B (illustrated by the complete dead area profile of Fig. 6), there is clearly a reflection in the graph through the mid-dead area point. This indicates that many of the units only execute a few instructions resulting in dead areas such as t and $\mathrm{cl} - t$, where t is the controller clock cycle where the instruction is executed, and cl is the controller length. This is demonstrated in longest dead area profile of Fig. 7, where only the longest deadtime for each unit is included in the histogram, which results in a figure similar in shape to the right half of Fig. 6. The reason why many units only implement one instruction becomes clear from inspection of the design statistics: 76% of the functional units in design B are simple gates such as OR, AND and comparators; the remaining 24% are subtracters, adders, and shifters. Real unit data on area and delay are used

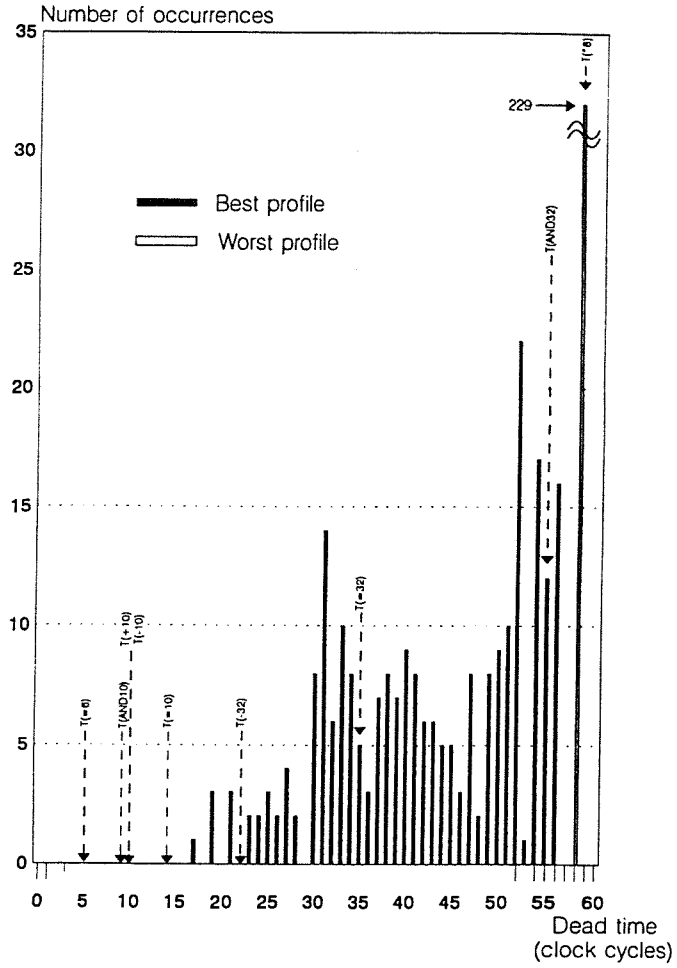


Fig. 7. Longest dead area profile for design B—compare with Fig. 6.

during optimization of the design, and for the simple gates it is just not cost effective to share the majority of units due to the added expense of multiplexing their inputs, so a simple gate is rarely used to implement more than one instruction. A similar result is found for other designs (for example circuits C and D in Table I).

The profile of design D shown in Fig. 8 is chosen as representing an extreme case where each functional unit executes only one instruction. The clustering of dead area times at the ends of the profile is due to the combined effects of probabilistic instructions, few instructions requiring functional units compared with those requiring registers, and optimization transforms biased toward ASAP [14] compaction of instructions within the control graph.

3.2.4 Loops: The deadtimes presented here are calculated by assuming that no loops exist in the controller (which is not generally the case), and the addition of loops to the system will vary the dead area times. The analysis of dead area times taking into account controller loops is a complex procedure. A controller loop will add additional dead areas for each loop iteration. Duplication of dead areas within the loop and the addition of longer dead areas from the end of the loop back to its beginning (loop internal concatenation) will further complicate matters. The dead areas that have been calculated will remain for loops that are not executed, and dead areas may only be lengthened where the end and start of a loop join. To summarize, the addition of loops can *only* skew the profile to the right (lengthy end) of the profile abscissa.

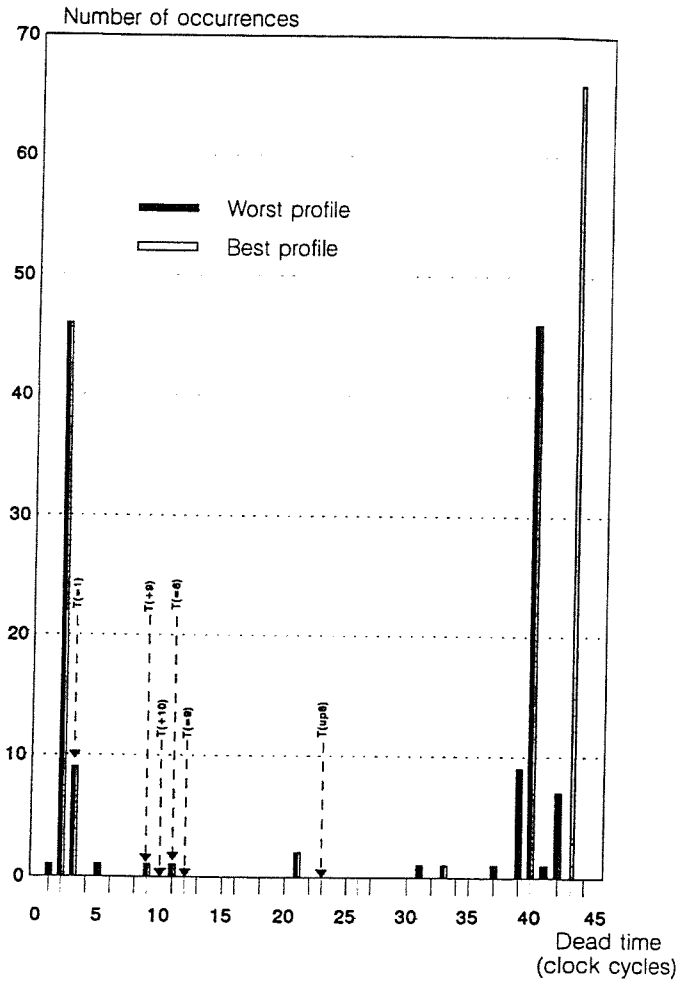


Fig. 8. Longest dead area profile for design D, optimized for speed.

IV. The Controller Subsystem

If the design is small or highly data-domain dominant, and complete testing is required with no timing impact, then some units must be tested over a *number* of dead areas and controller cycles. The test controller is, therefore, required to coordinate the testing of units and to track the progress of each test sequence, halting and resuming the testing between the units live (active) times. The test controller must act on control signals from the main controller, and in order to stop and start the test sequence, intermediate test data must be stored.

Depending on the overall test rate requirement, parallel unit testing can be performed at the expense of test controller complexity and consequent implementation area. Ideally, the test hardware should be as simple as possible to minimize the occurrence of faults within it—bear in mind that the test controller itself incurs an area cost. An increase in the test rate can be achieved at little controller expense by utilizing all unit dead areas rather than just the biggest. Additionally, the test rate can be increased further by carefully ordering the unit testing sequence to reduce the idle time of the test controller itself. Test controller strategies such as a partially distributed control architecture employing reconfigurable scan chains and test registers [17] can be used.

Once the test requirements are defined in terms of dataflow, the testing functional units can of course be optimized alongside the rest of the circuit. The optimizer will never compromise the tests per se, as the *result* of the tests is always required by the output of the system. This is a complex area and will be covered in detail in a later paper.

4.1 *Optimization for* Contiguous *Dead Area*

The synthesis optimization system uses a simulated annealing algorithm to search a multidimensional design space [9], [10]. The abstract nature of simulated annealing allows it to be used for optimization with respect to any criteria that can be assigned a quantifiable penalty function. To enable a less complex (and hence smaller and more reliable) test controller to be constructed, it is naturally desirable to perform a complete unit test within a single dead area of that unit. The cost function for the optimization algorithm is thus extended to include a measure of the lengths of the longest dead area for each functional unit. By aiming to skew the profile toward the "lengthy" end, there is a greater probability that unit tests can be completed within single dead areas, thereby simplifying the test controller.

V. Test Strategy

The actual testing mechanism for a given unit is relatively unimportant (within the context of this paper) as far as the dead area testing approach is concerned. A unit may be tested using any of the established *design for testability* or *built-in testing* techniques [19], [20]. Stored test vectors are unsuitable for ASIC applications due to the excessive chip area taken by the ROM. External ROM is possible and would allow easy access for updating test vectors, but there is an additional test controller and/or IO pin cost in reading the external data. Alternatively, a feedback shift register (FSR) [19], [21] can be used to generate either an exhaustive or random set of test vectors. The circuit response can then be analyzed using signature analysis and the result compared with the stored fault-free signature. As with the previous section, it is possible to associate a penalty figure with each of these approaches and to include it in the annealing process.

VI. Concluding Remarks

The justification and economic considerations of testing at various levels are well known and need not be reiterated here. Most on-line test techniques are applied by intermittently putting the system into a "test" mode while in service. The system described here gives a *real-time continuous built-in test capability* (confidence indication) to a synthesized circuit with almost *no* speed and little area overhead, and certain pathological circuits require some speed overhead. Further work is required on certain aspects of the system, for example, the extension of dead area analysis to include controller loops. Some controller loops will be data dependent thus complicating the test controller still further. Dead area analysis can be refined and made more accurate when analyzing probabilistic operations by taking into account linked probabilistic events. For example, when generating the best case dead area times of Fig. 5 it was assumed that each instruction is executed with independent probability. However, the instructions are all associated with the complex divide operation of the circuit and are, therefore, dependent on the same probabilistic condition, resulting in only the deadtime of seven cycles. In this case the same best deadtime occurs, however, this will not always be the case.

The strategy described involves only functional units; registers and interconnects are not tested.[2] To increase overall confidence further and test these structures, the techniques described here may be used to test all adjacent *pairs* of functional units plus their connecting structures. In this way, *complete* coverage of the functional units, interconnects, and registers is assured, but the cost in terms of area and probably speed is high.

Acknowledgment

The authors gratefully acknowledge the assistance of TransEDA Ltd., for the use of the TransTest [16] test coverage system used in the preparation of this paper.

References

[1] V. Chickermane, J. Lee, and J. H. Patel, "Addressing design for testability at the architectural level," *IEEE Trans. Computer-Aided Design*, vol. 13, no 7, pp. 920–934, July 1994.
[2] C. H. Chen, T. Karnik, and D. G. Saab, "Structural and behavioral synthesis for testability techniques," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 6, June 1994.
[3] M. Potkonjak, S. Dey, and R. K. Roy, "Considering testability at behavioral level: Use of transformations for partial scan cost minimization under timing and area constraints," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 5, pp. 531–546, May 1995.
[4] S. Dey and M. Potkonjak, "Transforming behavioral specification to facilitate synthesis of testable designs," in *Proc. 1994 Int. Test Conf.*, IEEE cat 94CH3483-5, pp. 184–193.
[5] M. Potkonjak, S. Dey, and R. K. Roy, "Behavioral synthesis of area efficient testable designs using interaction between hardware sharing and partial scan," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 9, pp. 1141–1154, 1995.
[6] W. B. Jone and C. A. Papachristou, "A coordinated circuit partitioning and test generation method for pseudo-exhaustive testing of VLSI circuits," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 3, pp. 374–384, 1995.
[7] A. Orailoglu and R. Karri, "Defect tolerant layout synthesis," *Int. J. Electron.*, vol. 76, no. 6, pp. 1121–1133, 1994.
[8] A. Orailoglu and R. Karri, "Synthesis of fault-tolerant and real-time microarchitectures," *J. Sys. Softw.*, vol. 25, no. 1, pp. 73–84, 1994.
[9] K. R. Baker, A. J. Currie, and K. G. Nichols, "Multiple objective optimization in a behavioral synthesis system," *Proc. Inst. Elect. Eng.*, vol. 140, no. 4, pp. 253–260, Aug. 1993.
[10] K. R. Baker, A. D. Brown, and A. J. Currie, "Optimization efficiency in behavioral synthesis," *Proc. Inst. Elect. Eng.*, vol. 141, no. 5, pp. 399–406, Oct. 1994.
[11] Z. Peng and K. Kuchcinski, "Automated transformation of algorithms into register-transfer level implementations," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 2, pp. 150–168, Feb. 1994.
[12] R. A. Rutenbar, "Simulated annealing algorithms: An overview," *IEEE Circuits and Devices Mag.*, pp. 19–26, Jan. 1989.
[13] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 13, 1983.
[14] K. R. Baker, "Multiple objective optimization of data and control paths in a behavioral silicon compiler," Ph.D. dissertation, Univ. Southampton, Southampton, Hants., U.K., 1992.
[15] M. Potkonjak and J. Rabaey, "Optimizing resource utilization using transformations," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 3, Mar. 1994.
[16] *TransTest User Guide*, Version 1.0, TransEDA Ltd., 1992.
[17] D. Mukherjee, M. Pedram, and M. Breuer, "Control strategies for chip-based DFT/BIST hardware," in *Proc. Int. Test Conf.*, Oct. 1994, pp. 893–902.

[2]However, an extension of this technique, "inversion testing," forces into being and then tests closed loops of the dataflow graph, which *includes* functional units, registers, and interconnects.

[18] K. R. Baker and A. D. Brown, "Incorporation of testability into a behavioral synthesis system," submitted to *IEE Proc. Circuits, Devices and Systems*, submitted for publication.
[19] B. R. Wilkins, *Testing Digital Circuits: An Introduction*. London, U.K.: Van Nostrand Reinhold, 1986.
[20] R. Gage, "Structured CBIST in ASICS," in *Proc. Int. Test Conf.*, Oct. 1993, pp. 332–338.
[21] T. Damarla and A. Sathaye, "Applications of one-dimensional cellular automata and linear feedback shift registers for pseudo-exhaustive testing," *IEEE Trans. Computer-Aided Design*, vol. 12, no. 10, pp. 1580–1591, Oct. 1993.

Andrew D. Brown (M'90–SM'96) was born in the United Kingdom in 1955. He received the B.Sc.(Hons.) degree in physical electronics and the Ph.D. degree in microelectronics from Southampton University, Southampton, Hants., U.K, in 1976 and 1981, respectively.

He was appointed Lecturer at Southampton University, Southampton, Hants., U.K, in 1981, Senior Lecturer in 1989, and Reader in 1992. He was a Visiting Scientist at IBM, Hursley Park, U.K., in 1983 and a Visiting Professor at Siemens at NeuPerlach, Munich, Germany, in 1989. He is currently head of the Design Automation Group, Electronics Department, Southampton University. The Group has interests in all aspects of simulation, modeling, synthesis, and testing.

Dr. Brown is a Fellow of the IEE, a Chartered Engineer, and a European Engineer.

Keith R. Baker received the B.Sc. degree in electronic systems engineering from the University of East Anglia, U.K., in 1988 and the M.Sc. degree in microelectronics systems design and the Ph.D. degree from Southampton University, Hants., U.K., in 1989 and 1992, respectively.

He was a Research Assistant at Southampton University, Southampton, Hants., U.K. while completing the Ph.D. degree there. He is currently a Teaching Fellow, Southampton University. His research interests are in high-level synthesis, behavioural hardware description languages, and the continued development of the in-house synthesis suite.

Alan J. C. Williams received the M.Eng. degree in information engineering and the Ph.D. degree in behavioral synthesis from Southampton University, Southampton, Hants., U.K., in 1992 and 1997, respectively.

From 1992 to 1993 he was with Philips Research Laboratories, Redhill, U.K. He is currently with the Design Automation Group, Southampton University, Southampton, Hants., U.K. His research interests include novel synthesis techniques and other areas of computer-aided design.

On-Line Test of Synthesised Designs Exploiting Latency Analysis

A.C. Williams, A.D. Brown, M. Zwolinski, K.R. Baker

1. Introduction

The use of synthesis in the design of complex digital systems is becoming more widespread. As the complexity of systems increases, so the abstraction of synthesis to higher levels will follow. The designer is concerned with the high-level specification and whether the implementation meets certain design criteria such as speed, area, power consumption and testability. The underlying sub-circuits that make up the detailed *structural* implementation are hidden.

An important aspect of any hardware system is the inclusion of some form of testability. This is a post-synthesis task which adds the structures appropriate to the structural implementation to provide the level of testability required by the designer. The test structures consist of some form of partial scan path and perhaps built-in self-test (BIST). Scan path structures allow testing at manufacturing time. BIST additionally allows a test to take place at system initialisation time. Safety critical systems, which cannot easily be taken off-line, may also require a real-time indication of their 'healthiness', so that should a fault occur, redundant circuits (possibly at the chip or board level) can be immediately switched in, and the faulty device identified and replaced without any operational disruption.

The use of on-line testing [1], that is testing the system whilst it is in operation, requires additional processing capability and may have a performance implication for the system. Synthesised synchronous systems usually have periods of time (both whole and partial clock cycles) during which processing units are inactive. These latent periods may be exploited in order to provide on-line testing capabilities, without necessarily impacting upon normal system performance.

This paper details the methods and results of an investigation into the feasibility of adding on-line testability to synthesised designs such that design units are tested during their latent

periods. The study examines the testability of a number of benchmark designs synthesised by the MOODS synthesis system [2, 3]. Both area and speed efficient implementations are analysed using the methods described and their relative impact on testability determined. In addition, techniques to improve the overall on-line testability of a design are considered together with their impact on the user's optimisation objectives.

The rest of this paper is organised as follows: section 1.1 describes the basic requirements for on-line testing and some assumptions and simplifications made for this study. This is followed by an introduction to the analysis of a design demonstrating the basic principles involved and defining a number of important terms used throughout the investigation. This section concludes with a detailed description of the practical methods used to analyse real designs synthesised by MOODS. The methods developed here are then used to examine the on-line testability of a number of benchmark designs with results described for the various initial and optimised implementations. This includes an investigation into how the designs may be further optimised to improve their on-line testability and the trade-offs necessary to achieve this.

1.1 On-Line Testing

On-line testing requires a system to be tested concurrently with normal operation. This may be done with some external monitoring system, but here we assume that part of the system may be tested using any latent periods created when units are not processing data required for normal operation [1]. These latent periods depend upon the scheduling and sharing of units within an architecture as well as the type of unit, and thus vary between different implementations of a single design. Within a distributed data and control path architecture, such as that generated by the MOODS system, the data path units typically constitute the major part of any implementation. For the purpose of this study only the functional units in the data path are considered. (This notwithstanding, the analysis methods could equally be applied to other parts - registers and interconnect - of the design.)

The testability of a given unit in a larger design is dependent on a number of parameters: the unit type, the length and fragmentation of the latent periods, the testing strategy and the test controller. A test controller is required to test each unit over one or more of its latent periods according to the selected testing strategy. If more than one period is used then the

tests must be halted when the unit is required to process real data, thus complicating the controller architecture. There is clearly some trade-off between test controller complexity and the throughput of tests applied to units.

There are a number of testing strategies that might be employed; for example, pseudo-random test vectors and signature analysis or stored test vectors and responses. Each approach has its advantages and disadvantages and each takes a different time to perform its complete test. It may be the case that a strategy good for one unit may not be as suitable for a unit of a different type. This study confirms that on-line testing using design latency is practical; the investigation of different testing strategies and test controller methodologies as applied to various unit types will be the subject of further research.

In order to demonstrate that on-line test using latent periods is achievable, some assumptions on the approach to testing individual units must be made. For the purpose of this paper it is assumed that an optimal set of test vectors exists capable of testing any particular unit for the required single-stuck-at-fault coverage. These are generated externally using a commercial test vector generation tool [4]. The tests will be applied by the controller at a rate of one per system clock cycle, thus it will take n cycles to fully test a given unit where n is the number of test vectors. Exhaustive testing using pseudo-random sequence generators could equally be used, but for the purposes of this study, analysis of extremely long sequences would have been needed. It is also assumed that the test controller is of a relatively simple design which must completely test a unit within a single latent period. The effect of these assumptions on the results and hypotheses thus formulated will be discussed in the results section.

2. Latency Analysis

Section 2.1 describes the principles of latency analysis and the obstacles that must be overcome in order to obtain accurate results. The discussion describes latency analysis in the context of the architecture generated by the MOODS synthesis system. This is followed by a detailed description of the methods and algorithms implemented in an analysis tool used to determine latency and testability figures.

2.1 Design Latency Analysis

The MOODS synthesis system generates a distributed architecture containing a control unit which organises the flow of data through a data path. The control path consists of a single bit register for each state joined together by arcs through which tokens are conditionally passed, in a similar manner to that of Petri-nets. It comprises a combination of parallel and conditional branches, nested loops and sub-program calls. Each state may conditionally activate any number of functional units (adders, multipliers etc.) in the data path which are interconnected by nets directing data through the system.

An initial behavioural description is translated from VHDL source to an unoptimised register-level implementation whereby each instruction occupies a single control state, and each data path unit executes one instruction, the results of which are stored in registers at the end of each clock cycle. During optimisation both the control and data path graphs are transformed so that the user's objectives for constraints such as area and speed are met. This results in control states conditionally activating many data path units, each of which may be used to implement a number of instructions.

The purpose of latency analysis is to calculate the distribution of inactive clock cycles for a given data path unit. Armed with this information and the time required to perform a complete test on the unit, a figure may be obtained describing its on-line testability. This section explains the basis behind the various terms and figures used for latency analysis and on-line testability.

Figure1a shows a simple control graph consisting of a sequential section containing four states linked by arcs. During execution control may exist in any one of the four states and follows the sequence of states, repeating via the feedback arc from state 4 to 1. Each data path node implements a number of instructions executed from one or more control states. The set of control states that activate a particular data path unit is called the target set. In this example we define the target set to be only state 3, which activates the data path unit being analysed. From inspection of the graph it can be seen that this unit is inactive during states 1, 2 and 4. A latent vector is defined as a group of arcs (state transitions), none of which are incident on a member of the target set, except that the final state may terminate on a member of the target set. Thus in Figure 1a, there exist latent vectors {(4,1)},

{(4,1),(1,2)}, {(4,1),(1,2),(2,3)}, {(1,2)}, {(1,2),(2,3)} and {(2,3)}. The length of a latent vector is the latent period.

If, for the sake of illustration, a full test (sufficient to provide the desired fault coverage) for the unit active in state 3 requires 3 cycles, then the test can only be completed when started from state 4, that is the test may only be applied during the latent vector {(4,1),(1,2),(2,3)}. At any moment in time, for this example, there is an *equal probability* of being in any one of the four states; thus the state probability for each state is 0.25. Since a full test can only be started from state 4, the overall probability of being able to fully test the unit at any given moment is 0.25. Alternatively, if the test only takes 2 cycles to complete, the probability increases to 0.5 as there are two possible latent vectors in which a full test may be applied: {(4,1),(1,2)} and {(1,2),(2,3)}. This figure is termed the test completion probability and is defined as the probability that, at any arbitrary time, a test may be started and completed before the data path unit is activated.

Figure 1b shows a more complex example that includes a conditional branch. This type of structure occurs as a consequence of *if*, *for* and *while* statements. States 2 and 3 are mutually exclusive, their activation depending upon the condition of the preceding state 1. Assuming (for now) there is an equal probability of choosing either alternative, the *arcs* can each be assigned an arc probability of 0.5. The presence of such conditional branches can have a profound effect on the latent vectors present. As an illustration, consider the latent vectors starting at state 1. Here, the shortest latent periods are obtained from vectors {(1,2)} and {(1,3)} each of which has a path probability of 0.5, there being an equal chance of branching one way or the other. Walking either of these paths will eventually return us to state 1, where there is again a 0.5 chance of taking either branch. (Thus the chances of ending up at the state in the target set (state 3) will be the asymptote of $\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots$, i.e. 1, which is what we would intuitively expect.)

A latent vector's path probability describes the probability of obtaining a particular vector given that the vector's start state is currently active. However, to determine the probability of obtaining the vector at any point in time, the state probability of the start state must be

taken into consideration. Referring to Figure 1b, the state probability for state 1 is 0.286[1], therefore the latent vector {(1,2),(2,5)} (path probability 0.5) is said to have a vector probability (vector probability × state probability) of 0.143. Whenever a vector includes the branch of state 1 it is divided into two paths, one down the left side that will continue, and one down the right side that will end at state 3. Each time this happens the vector path probabilities are halved thus, in this example, an infinite number of latent vectors with exponentially decreasing path probabilities are obtained.

The behaviour of individual data path units can be examined using the units' dead vectors. These are defined as latent vectors that start from a particular control state, and end on a member of the target set. Thus the set of dead vectors is a subset of the set of latent vectors. The dead vector probability for a vector of length *n* is the probability that at any given moment in time, a member of the target set is exactly *n* clock cycles away from a particular control state. These vectors are encapsulated in the complete dead vector probability histogram, which plots the sum of all dead vector probabilities for a particular functional unit against the dead vector length. The dead vector length gives us the time available to complete one or more tests.

In order to investigate the on-line testability of real designs using data obtained from the methods above, three basic graphs are plotted. The first details the distribution of dead vectors for an individual data path unit by plotting the complete dead vector probability histogram. This shows how the individual vectors are affected by the control structure and the particular scheduling of unit activations within the control graph. A more general guide to the on-line testability of a particular unit is given by the second type of graph which plots the unit's test completion probability against the test time required. Finally, to illustrate the

---

[1] Let the probability of being in state *i* be P_i. Then from Fig. 1b, we see that $P_2 = P_1/2$, $P_3 = P_1/2$, $P_3 = P_4$, $P_5 = P_1$ and $P_1 + P_2 + P_3 + P_4 + P_5 = 1$. Inverting

$$\begin{bmatrix} \frac{1}{2} & -1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

gives us $[P_1,P_2,P_3,P_4,P_5]^T = [0.286,0.143,0.143,0.143,0.286]^T$

overall on-line testability of an entire implementation, a set of sample test times is used to determine the test completion probability for each unit in an implementation, and the results plotted on a histogram. It should be emphasised that this probability does not refer to a unit's fault coverage which is defined by the test vectors, it only represents how often a unit may be fully tested while the system is active.

2.2 Latency Calculation Methods

In order to obtain the above results for real synthesised VHDL designs, the output from the MOODS system is analysed using a dedicated post-processing tool. This includes a number of functions that operate on the control and data graph structures allowing the user to perform a detailed investigation of the optimised design. The data produced includes dead vectors and test completion probability versus test time for any given data path unit, together with combined testability results for all or part of the entire design.

2.2.1 Markov Chains

For the purposes of this discussion, the controller is a deterministic state machine. The control unit of a synthesised implementation can be described by the complete set of control graph states, S. As the unit executes, the flow of control through the graph is a discrete time stochastic process [5] defined by the set of variables:

$$\{X_t \in S \mid t = 0,1,2,...\}$$

where X_t represents the current state (not set of states) t cycles from the start of the process. The passage from state to state through the control graph is described by a set of instances of X_t where decisions made at forking states are independent of the past history of X_t. This property defines the process as a Markov chain [6].

From any given state i, the probability of moving to state j on the next clock cycle is defined as the transition probability:

$$p_{ij} = P(X_{m+1} = j | X_m = i) \text{ where } i, j \in S \quad (1)$$

This corresponds to the arc probabilities described in the previous section. Using these probabilities it is possible to fully determine the evolution of the control process as time

progresses. In addition, the n-step transition probability of moving from state i to state j in n cycles can be defined by:

$$p_{ij}^{(n)} = P\left(X_{m+n} = j \middle| X_m = i\right) \tag{2}$$

The states of a Markov chain can be classified according to a number of different properties:

- An irreducible class is made up from states which form a set C where $\forall\ i, j \in C, \exists$ integers $m, n > 0 \mid p_{ij}^{(m)} > 0, p_{ji}^{(n)} > 0$ (i.e. C is a connected graph). This can be viewed as a set of states which, once entered, will either never be exited or, having been exited will never be re-entered.

- A state is aperiodic if it can return to itself, i.e. $p_{ii}^{(n)} \geq 0$ for $n > 0$ and n is finite, where $p_{ii}^{(n)} = 0$. If the state has a constant return period it is periodic.

- The first return time, T_{ii}, is defined as the number of steps between any two successive entries into a state i.

- A transient state is one for which there is a finite probability that it may never be re-entered.

- A positive-recurrent state recurs infinitely often, and has a finite return time.

2.2.2 Latent Vector Calculation

The Chapman-Kolmogrov equation describes the evolution of a Markov chain as a matrix multiplication based upon the transition matrix:

$$\mathbf{P}^{(n+m)} = \mathbf{P}^{(n)}\mathbf{P}^{(m)} \quad \text{for } m, n \geq 0 \tag{3}$$

By setting m to 1, an iterative method for evaluating each of the n-step transition probabilities from the start of the process is obtained:

$$p_{ij}^{(n+1)} = \sum_{k \in S} p_{kj}^{(n)} p_{ik}^{(1)} \quad \text{for } n = 0,1,2.... \tag{4}$$

This relationship forms the basis for the calculation of latent vector probabilities. A latent vector of length n is defined as an n-step transition starting from an arbitrary control state, which does not activate a particular target data path unit during any of its n steps, except

perhaps the final one. Each of the target units may be activated by a number of states in the control graph which make up the target set, D. Thus, the probability of obtaining a latent vector of length n from a starting state i, to an end state j can be defined as:

$$V_{ij}^{(n)}(D) = P\left(X_{m+n} = j,\ X_m, X_{m+1}, \ldots, X_{m+n-1} \notin D \,\middle|\, X_m = i\right) \quad \text{where } i, j \in S \text{ and } D \subset S \tag{5}$$

Comparing this result to equation 1 it can be seen that the one-step latent vector $V_{ij}^{(1)}$ is directly related to the transition probability $p_{ij}^{(1)}$:

$$V_{ij}^{(1)} = \begin{cases} 0 & \text{if } i \in D \\ p_{ij}^{(1)} & \text{if } i \notin D \end{cases} \tag{6}$$

From this, equation 4 yields an iterative method for determining the latent vector probability:

$$V_{ij}^{(n+1)} = \sum_{k \in S} V_{kj}^{(n)} V_{kj}^{(1)} \quad \text{for } n = 0,1,2,\ldots \tag{7}$$

Thus to calculate latent vector probabilities it is first necessary to modify the one-step transition matrix $\mathbf{P}^{(1)}$ so that all arc probabilities leaving the target states D are set to zero. This results in the one-step latent vector matrix $\mathbf{V}^{(1)}$. Calculating the multi-step latent vectors is then simply a matter of iteratively applying equation 7 via a matrix multiplication as in equation 3.

In order to be able to relate the latent vector probabilities for different start states to each other, it is necessary to weight the vectors according to the probability of being in each particular start state. This is achieved by taking into account the steady state probability described below.

2.2.3 Steady State Probability Calculation

π_j is the steady state probability of being in state j at any given moment. This can also be interpreted as the asymptotic transition probability for progressing from state i to j:

$$p_{ij}^{(n)} \to \pi_j \geq 0 \text{ as } n \to \infty \forall i, j \in S \tag{8}$$

which must be independent of the starting state i if π_j is to have a stationary value. For this to be true, the Markov chain must be both irreducible and aperiodic. These conditions are defined by Kolmogrov's theorem, which also states that in addition to the above properties, a positive-recurrent Markov chain will have a unique solution to the steady state equation:

$$\pi = \pi \mathbf{P}^{(1)} \tag{9}$$

The steady state probabilities for the entire Markov chain can therefore be calculated by determining the solution to the simultaneous linear equations:

$$\pi_j = \sum_{k \in S} \pi_k p_{kj}^{(1)} \quad \text{where} \quad \sum_{j \in S} \pi_j = 1 \tag{10}$$

A Markov chain having the above properties is said to be ergodic. This describes a process in which it is possible to reach any state from every other state. In order to satisfy these conditions for the design implementations synthesised by MOODS, it is necessary to pre-process the control graph before it is analysed. Consider the general control graph of Figure 2 showing a typical controller synthesised from a VHDL input in a skeletal form. The graph splits into three main blocks. The first section contains the main design initialisation states, which are executed only once on a system or power-up reset. These represent transient Markov states that are never re-entered, that is they have an infinite first return time. The graph then splits into a number of concurrent blocks corresponding to the main VHDL parallel processes, which are considered to execute completely independently of one another thus forming a number of irreducible classes. Each process is comprised of an initialisation block, again executed only on a reset, together with a main section that loops ad-infinitum. This is the main functional block which forms a positive-recurrent Markov chain. Since the initialisation stages are all transient states, they have steady state probabilities of zero; the probability analysis therefore proceeds by applying the steady state equation separately to each main process loop. This makes practical sense since any dead vectors which start in an initialisation stage will only occur once on a reset and will be of no use in the design of a system that is continuously tested.

2.3 Dead Vector Calculation

The final stage in the calculation of the complete dead vector probability $H^{(n)}(D)$ for vectors of length n is to combine the above methods into a single algorithm. The dead

vector probability for a vector of length n starting from a particular state i, is defined as the sum of all possible latent vectors of length n, $V_{ij}^{(n)}$, which start at that state and end on any member of the target set D. To calculate the complete dead vector probability, the weighted sum of dead vectors for all possible start states is used:

$$H^{(n)}(D) = \sum_{i \in S} \pi_i \sum_{j \in D} V_{ij}^{(n)} \tag{11}$$

Note that from the definition of $V_{ij}^{(n)}$, if $i \in D$, $V_{ij}^{(n)}$ is zero.

$H^{(n)}$ is evaluated for a range of lengths to construct a dead vector histogram for any particular functional unit (see results section). In order to determine the probability of being able to test a unit given a minimum test time t, it is necessary to take into account all dead vectors of length t and above. This is obtained by integrating $H^{(n)}$ to determine the units test completion probability $T^{(t)}(D)$:

$$T^{(t)}(D) = \sum_{n=t}^{n \to \infty} H^{(n)}(D) \tag{12}$$

Theoretically, the sum of all the dead vector probabilities should be 1. However, since dead vectors starting from a target state will have a probability of 0%, due to the modifications made to the transition matrix to build $\mathbf{V}^{(1)}$, zero length dead vectors do not register in the results. Thus the actual sum of the dead vectors will be $\sum_{i \notin D} \pi_j$. Equation 12 then becomes:

$$T^{(t)}(D) = \sum_{i \notin D} \pi_i - \sum_{n=1}^{t} H^{(n)}(D) \tag{13}$$

It is now a relatively simple task to combine all the above information into a single dead vector calculation algorithm to construct a histogram of $H^{(n)}(D)$ for a range of n:

```
Calculate steady state probabilities, πj;        // equation 10
Build transition matrix from control arc probabilities, P(1);
Determine target set D for the test data path unit;
Process P to build V for target set D;           // equation 6
Iterate dead vector length n=1,2,3,...
{
  Evaluate next state transition matrix V(n);    // equation 7
  Calculate H(n)(D);                             // equation 11
}
```

These results can then be processed to obtain a test completion graph.

2.4 Constraints and Simplifications

In order to simplify the design of the analysis tools described above, a number of constraints and simplifications are imposed upon the implementations which can be analysed.

There are two main restrictions placed on the control implementation structures. First, there may be no VHDL sub-programs and second, there can be no parallel executed control states, other than the basic set of top-level processes.

Sub-programs cannot be used in the current system as they are implemented using a separate control process, which is executed in place of the sub-program call. Since each call occupies a single control state, a state which is actually active for several clock cycles would be considered to occupy just one cycle. A solution to this would be to expand each sub-program call in the main control graph into its constituent states before building the transition matrix.

During synthesis certain sections of a control graph may be parallelised in order to speed up the implementation. These sections re-combine at a *collect* control state which may wait for the longer parallel branches to complete before continuing with the main process. This means that a collect state can actually be active for more than one cycle. The Markov analysis described above however can only cope with single step control states, and while it would be possible to modify the collect state to include an implicit loop back to itself, this would not provide an accurate representation of the system's behaviour. A more suitable solution is to transform the parallel sections into the standard sequential format where a new state would be created to represent the set of original states active at any given point of execution.

Since no explicit consideration is taken of the input signals applied to the system, any data-dependent features must be ignored. An attempt has been made to quantify this via manipulation of the arc probabilities to favour a worst case situation. For example, VHDL *wait* statements are implemented by way of a state that continually loops back to itself until

a particular event occurs, usually a change on an input signal. In this situation, the arc probabilities are biased such that the probability of dropping through the statement is much greater than that of looping back to continue waiting. In addition, inter-process data dependencies which might be used to synchronise or communicate between processes are ignored. Again, this simplification is likely to skew the results a little towards the worst case scenario since the time a process spends waiting for synchronisation signals will not be included in the dead vectors.

None of the problems avoided by the restrictions described above are insoluble, but their practical impact is low and they do not materially affect the conclusions of the study. The data-dependent simplifications are much more difficult to quantify without performing detailed simulation based upon a range of typical input vectors. However, the use of worst case values for the transition probabilities ensures that 'real-life' testability of an implementation will, if anything, be better than the figures calculated.

3. Experimental Results

The MCNC High Level Synthesis Design benchmarks feature a number of synthesisable VHDL designs [7, 8]. Several of these have been synthesised and analysed using the methods described in the previous sections. Two of the designs, a Fast Fourier Transform processor (FFT) [7] and a RISC Microprocessor (FRISC) [8] are discussed in detail in this section.

The designs are investigated in their initial and optimised implementations, the optimisation objectives being area and delay. The results show full test completion probability histograms for each implementation based upon a set of sample test vectors for each of the main functional units. The figures used were generated by the TransTest package [4] and are listed in Table 1. In addition, a number of specific units are examined in detail to illustrate the effects that the different optimisation strategies have on the latent vectors available for testing. These are displayed using the complete dead vector probability histograms and by plotting the test completion probability against test time curves for the individual units concerned. By associating the results with the implementations' control graphs, the main factors affecting on-line testability are identified. This information is then

used to demonstrate how re-synthesis may be applied in order to improve the systems' on-line testability, and what effects any changes have on the user's initial optimisation criteria.

DP unit type (number)	Bit width	Number of test vectors	Fault coverage (%)
Add (14)	32	22	98.4
Minus (15)	32	22	99.6
	64	30	99.6
Rshift (32)	16	16	N/A
Comparators (20-25)	1	3	100
	16	18	100
	32	35	100

Table 1 Functional unit test vectors

The results below are split into two sections. The first examines the FFT design in an attempt to determine the factors affecting the on-line testability of the individual units and pinpoint areas for possible improvement. These ideas are then extended in the second section which details the FRISC processor, presenting some methods that may be used to improve the design and investigating the effect these changes have on the user's optimisation objectives.

3.1 Design On-Line Testability - FFT Processor

The Fast Fourier Transform contains around 140 lines of behavioural VHDL code, which was processed by MOODS to synthesise both area and delay optimised implementations. The resulting control graphs are shown in Figure 3. Both of these have the same general structure formed from the two main while loops in the VHDL description. Within this framework, there are a number of significant differences, the most obvious of which is the smaller number of states in the delay optimised version. This results in a shorter control path and hence a quicker overall implementation. In addition the maximum state delay, that is the longest propagation delay for a single control state, is shorter allowing a higher maximum clock speed. This improved performance is achieved by sacrificing area in order to increase the component count and enable a different scheduling scheme.

There are two main mechanisms involved in the optimisation process: unit sharing and graph compaction. Unit sharing takes two data path units, each activated by one control

state, and combines these into a single unit with multiplexed inputs activated by a number of control states. This results in a reduction in area, but there may be a speed penalty to pay due to increased complexity resulting from the use of input multiplexors. In addition, combining, for example, a 16-bit and 32-bit adder into a shared 32-bit unit will significantly slow down the smaller instruction and may affect the maximum clock speed permitted depending on the schedule. Graph compaction utilises the slack time within a state, that is the difference between the state delay and the clock period to combine two adjacent control states into one, thereby shortening the control path.

Both of these mechanisms have a profound effect on the length and distribution of dead vectors. Figure 4 shows histograms of the test completion probability for each data path unit in the area and delay optimised implementations based upon the sample test vectors. These results serve to illustrate a number of important factors influencing the on-line testability of an implementation. It should be noted that comparisons may only be drawn between the full set of units of the same type since each unit implements a different subset of instructions. For example, in the area optimised design (Figures 3b and 4b) unit 29 implements all six of the system's subtract instructions, whereas the delay optimised version (Figures 3a and 4a) spreads these across two subtractors, units 29 and 30.

The units in each histogram split into two general groups: arithmetic units and comparators. During optimisation none of the comparators will be shared since, for the technology library used, the additional hardware cost involved in sharing a unit is greater than the cost of a separate comparator. Comparing the results of the two optimisations it can be seen that the comparators in the area optimised implementation have a generally higher test completion probability than those in the delay optimised version. This is attributed to the greater number of control states (and hence longer execution time) as shown in Figure 3b. In contrast, the arithmetic units in this implementation are much more heavily shared (in fact there is only one instance of each type), thus the test completion probability for these units is lower than in the delay optimised implementation. The results for the two types of unit show that both increased sharing and decreased graph length conspire to limit the on-line testability of a synthesised design.

The factors influencing the on-line testability of the two implementations can be further understood through a detailed examination of a single operation, in this case the subtract

instructions described above. The control states that activate subtract units are shaded in the control graphs of Figure 3, and are analysed to determine the dead vector histograms, shown in Figure 5. The graphs show a relatively smooth reduction in dead vector probability as the vector length increases. This is typical of designs containing several mutually exclusive branches, which result in a continual increase in the number and length of dead vectors obtained each time control passes around the main loop. From the dead vector histograms corresponding graphs of test completion probability versus test time can be obtained. These are combined in Figure 7 which clearly illustrates the difference between the three units, in particular the more gradual curve for delay optimised unit 30. Considering the control graphs it can be seen that this difference is due to the localised nature of the states activating functional unit 30 (48, 58 in Figure 3a), which maximises the length of the dead vectors obtained. In addition, the unit only appears in one of the two main mutually exclusive branches. Therefore any vector paths following the right hand branch will not activate the unit until control loops back to state 8 resulting in substantially longer dead vectors. In contrast, functional unit 29 in both the delay and area optimised implementations has executing states scattered fairly evenly throughout the control graph (shown shaded in Figure 3). This leads to greater fragmentation of the dead vectors and hence a lower test completion probability. The application of these observations to improving the on-line testability of an implementation is discussed later.

3.2 Design Optimisation - FRISC Microprocessor

The FRISC microprocessor is a larger design of around 300 lines of behavioural VHDL. This is a relatively simple RISC processor based around a VHDL *case* construct which performs the instruction decoding together with a small amount of reset and interrupt request code. Unlike the FFT, the synthesised implementations have a much more linear control structure centred around the main *case* control state (55 in Figure 7) from which emerges a large number of mutually exclusive branches. The figure shows only the area optimised control graph as both implementations are almost identical, the difference in characteristics being mainly due to unit sharing and scheduling.

Figure 8 shows the test completion probability histograms for the area and delay optimised implementations. Comparators are not included as they would obscure the main functional

units and being unshared, they all have a test completion probability of nearly 100%. Considering the delay optimised results first it can be seen that the majority of the units are highly testable having test completion probabilities ranging from 33% to 98%. This is mainly due to the case statement, which results in a large number of mutually exclusive branches. The worst unit, with a probability of 0.4% is the adder in the first column. This poor figure can be attributed to the sharing of the unit, which is activated in almost all states in the top half of the graph (that is, states 14, 24, 28, 39, 49 and 54). The figure is further degraded by the lack of any mutually exclusive branch structures with a unit activation on only one side, thus forcing the adder to be activated on each iteration of the main loop. The effect of intensive sharing can be seen in the area optimised result (Figure 8b). Here 100% sharing (ie. all functional units of each type are mapped onto one physical unit) results in a design with only three different arithmetic units. Although this substantially reduces the *size* of the implementation it has also has a significant effect on the on-line testability. The sharing of large numbers of units within the mutually exclusive branches of the *case* structure reduces the length and number of dead vectors available for performing tests.

In order to improve the on-line testability of the implementations a certain amount of re-synthesis can be applied. This involves moving *away* from the user-specified goals on area and delay, to allow a greater testability to be realised. Clearly the dominating factor in the above design is the unit sharing. Considering the untestable adder (plus 14 in Figure 8a), it is possible to split this unit up into its constituent instructions each implemented by a different data path unit. The effect of unsharing this adder on both area and delay optimised test completion probability histograms is shown in Figure 9. The delay optimised graph is now significantly improved with each unit having at least an 18% probability of being tested. The price to be paid is of course the increase in area taken up by the new units. In the case of the delay optimised implementation the total area (including storage units, ports and interconnects) only increases by 13%. This may not be too much of a problem however examining the new area optimised histogram, it is clear that the number of units has increased out of all proportion to the rest of the design. In fact it has resulted in a 75% increase in total area, and a 250% increase in the area of the functional units.

The problem is now how to improve the area optimised implementation's on-line testability without having too severe an effect on its total size. As mentioned in the FFT results, the best testability figures are obtained by localising the use of a particular unit, and taking advantage of mutually exclusive branches to increase the length of the dead vectors. Examination of the control graph (Figure 7) suggests a number of possible ways of clustering unit activations together. Considering the adders in the design, a suitable partitioning may be to group together those units activated by states 14-39, 49-55, 64-70 and 91-164. This grouping also brings together the 16-bit and 32-bit units without forcing a 16-bit add instruction into a 32-bit adder. A similar exercise can be performed on the other functional units and the new test completion probability histogram calculated, as shown in Figure10. This new graph describes an implementation with a considerably improved on-line testability when compared to the original (Figure 8b), however the increase in total area is now a more acceptable 20%, with the increase in functional unit area being around 50%. It should be remembered that the test completion probability refers to how often a unit may be fully tested, as opposed to the fault coverage obtained. Thus the figures of around 30 to 50% should be perfectly acceptable in many situations.

In some cases it may not be possible to obtain the desired figure even with all units fully unshared. Considering the control graph of Figure 7, the adder activated in state 55 has a maximum test completion probability of around 18%. This is not an unreasonable figure but if it must be improved the only solution is to modify the control graph and include some extra states[2] to lengthen the dead vectors. This is a more complex task than sharing and unsharing data path units, however its effect can be demonstrated by considering the results for the unoptimised design shown in Figure 11. This has a considerable number of extra control states which, together with the lack of any sharing, produces a highly testable implementation. Clearly this does not represent an optimal design with respect to area or delay, however by targeting the relevant sections in the control graph, an improvement in the on-line testability is possible without sacrificing too much performance. It may in fact be possible to make use the extra states in order to re-schedule units in order to decrease the maximum state delay (and thus increase the maximum clock speed).

---

[2] Ostensibly these would be dummy states, but given that we have to have them to meet the testability requirements, the synthesis system *may* be able to do something useful with them.

4. Final Remarks

This analysis has concentrated exclusively on the functional units that make up a design; we should not lose sight of the fact that registers and interconnects can also develop faults. Furthermore, the difference between conditional and unconditional execution of units needs extremely careful analysis; where the probability of execution is data-dependent and highly skewed the strategy employed here will not deliver sensible results. We also note that the results are dependent on the test strategy used - further interaction between units must exist if a single test controller employed.

The work described here has taken as a starting point a synthesised design that has been optimised for area and/or delay, and has attempted to quantify the on-line testability of the design. Ideally, the testability requirements should be factored into the overall process at a higher level: the user should be able to specify the testability alongside the other optimisation criteria, so that MOODS can make trade-offs between the *three* criteria on an equal footing.

Acknowledgements

The authors gratefully acknowledge the assistance of TransEDA Ltd., for the use of the TransTest [4] test coverage system used in the preparation of this work.

References

1. A.D. Brown, K.R. Baker, A.C. Williams, "On-Line Testing of Statically and Dynamically Scheduled Synthesized Systems", IEEE Transactions on Computer-Aided Design, Vol. 16, No. 1, January 1997, pp. 47-57

2. K.R. Baker, A.J. Currie and K.G. Nichols, "Multiple objective optimisation in a behavioural synthesis system", IEE Proceedings-G, 140, no. 4, August 1993, pp 253-260.

3. K.R. Baker, A.D. Brown and A.J. Currie, "Optimisation Efficiency in Behavioural Synthesis", IEE Proceedings-G, 141, no. 5, October 1994, pp 399-406.

4. "TransTest User Guide", version 1.0, TransEDA Ltd., 1992.

5. P.G. Harrison and N.M. Patel, "Performance Modelling of Communication Networks and Computer Architectures", Addison- Wesley, 1994, pp 81-163. ISBN 0-201-54419-9.

6. D. Freedman, "Markov Chains", Springer-Verlag 1983.

7. Microelectronics Centre of North California, "High Level Synthesis Workshop Benchmarks", 1989, 1991.

8. P.R. Panda and N. Dutt, "1995 High Level Synthesis Design Repository", University of California, Irvine, Technical Report #95-04, 7 February 1995.

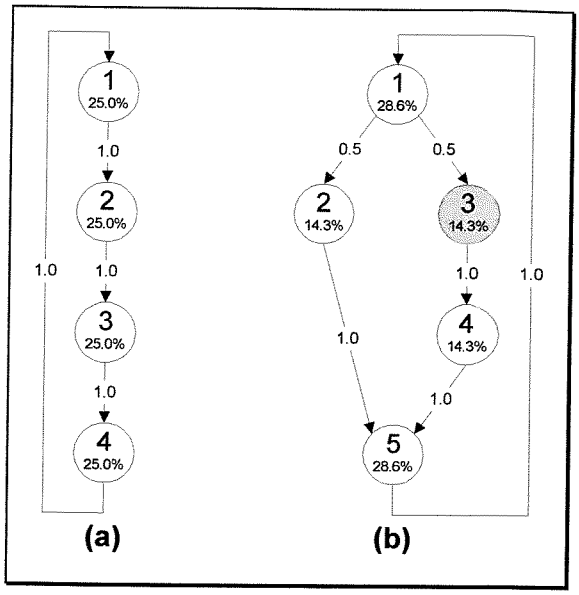


Figure 1 Example control graphs

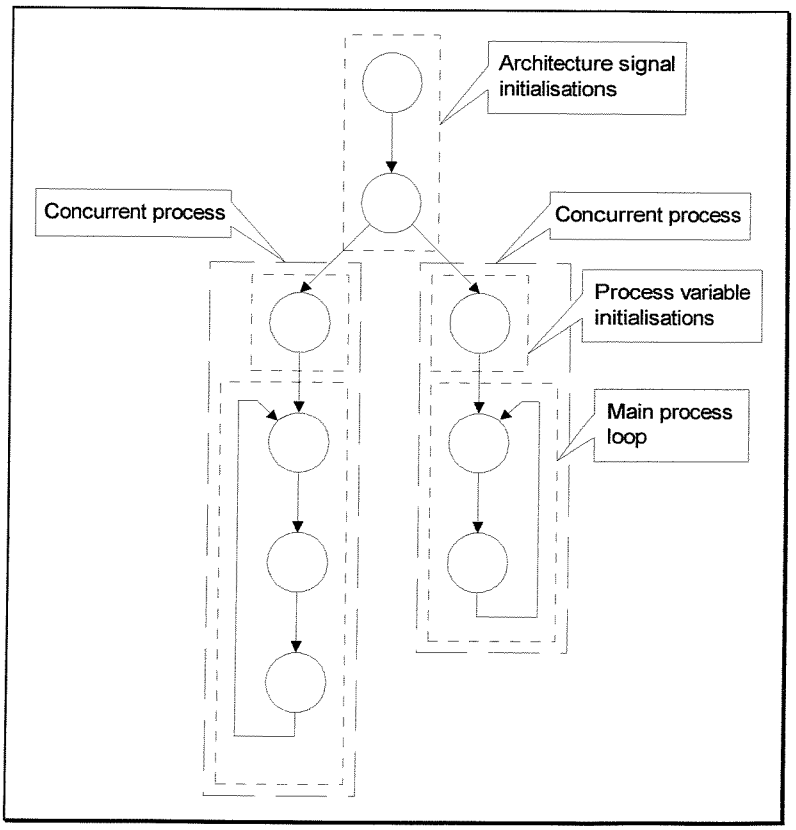


Figure 2 Skeleton synthesised VHDL control graph

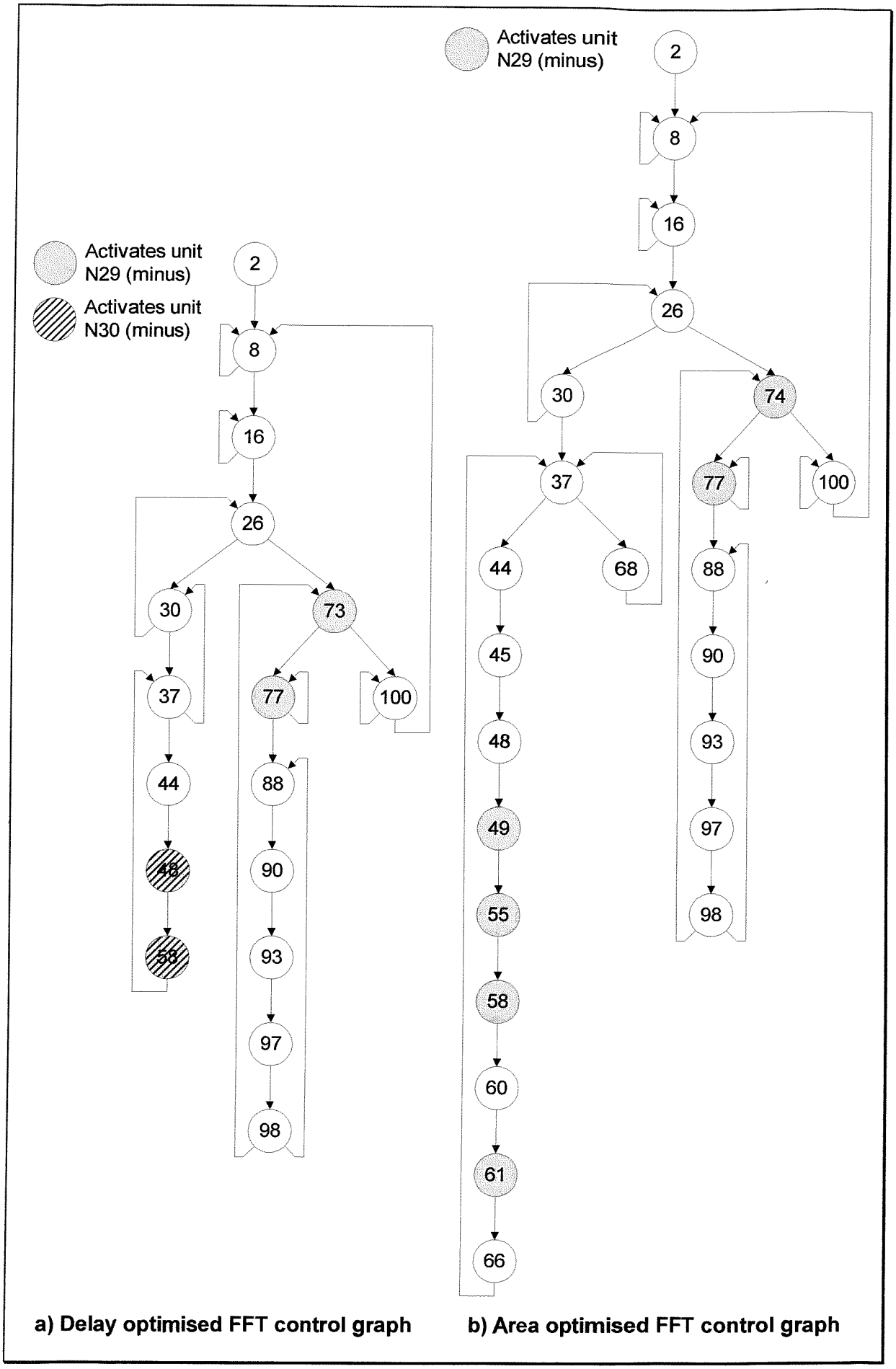


Figure 3 Area and delay optimised FFT control graphs

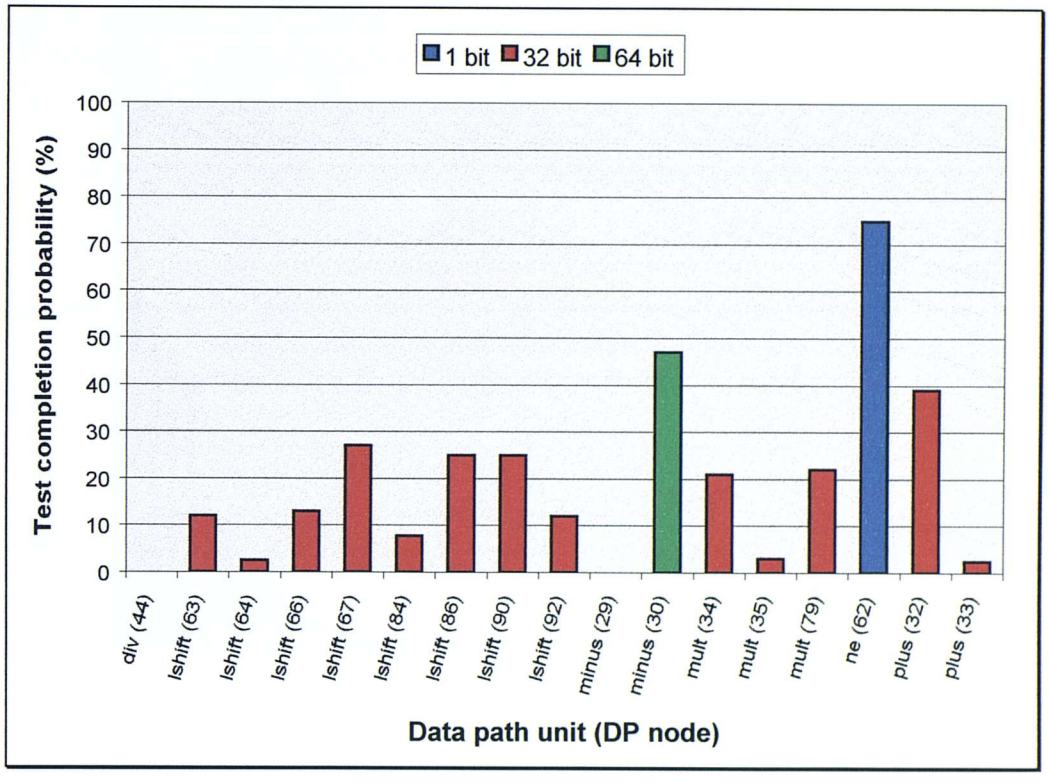


Figure 4a Delay optimised FFT test completion probability per unit

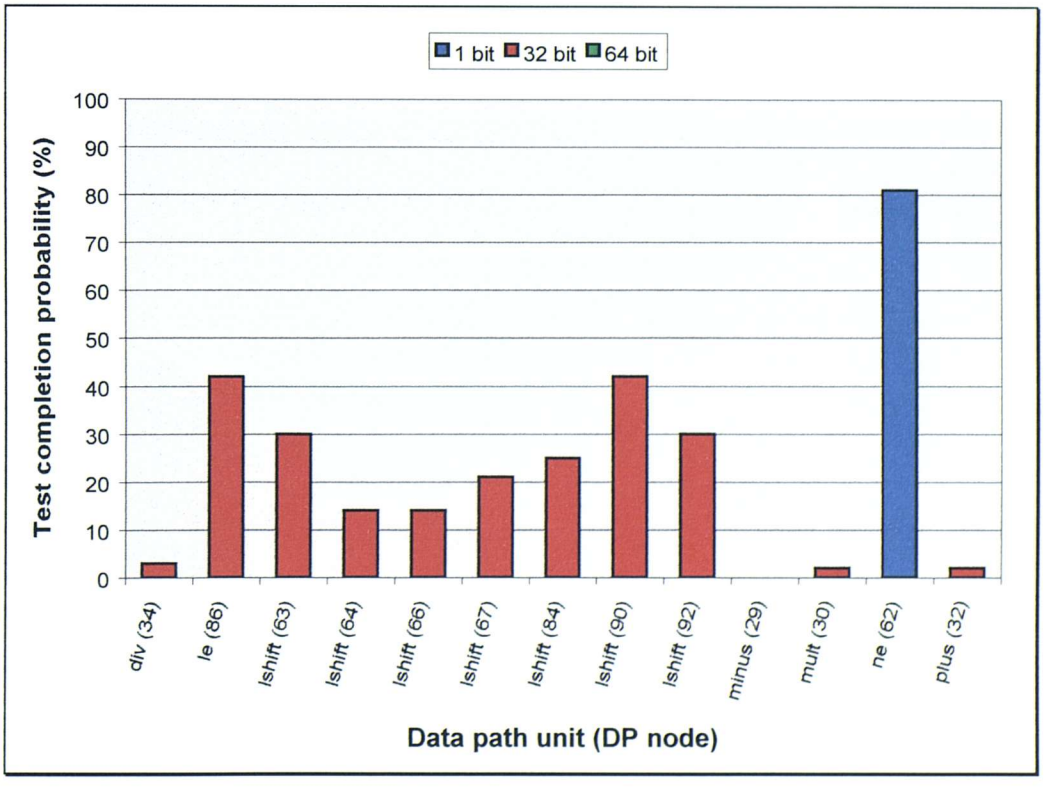


Figure 4b Area optimised FFT test completion probability per unit

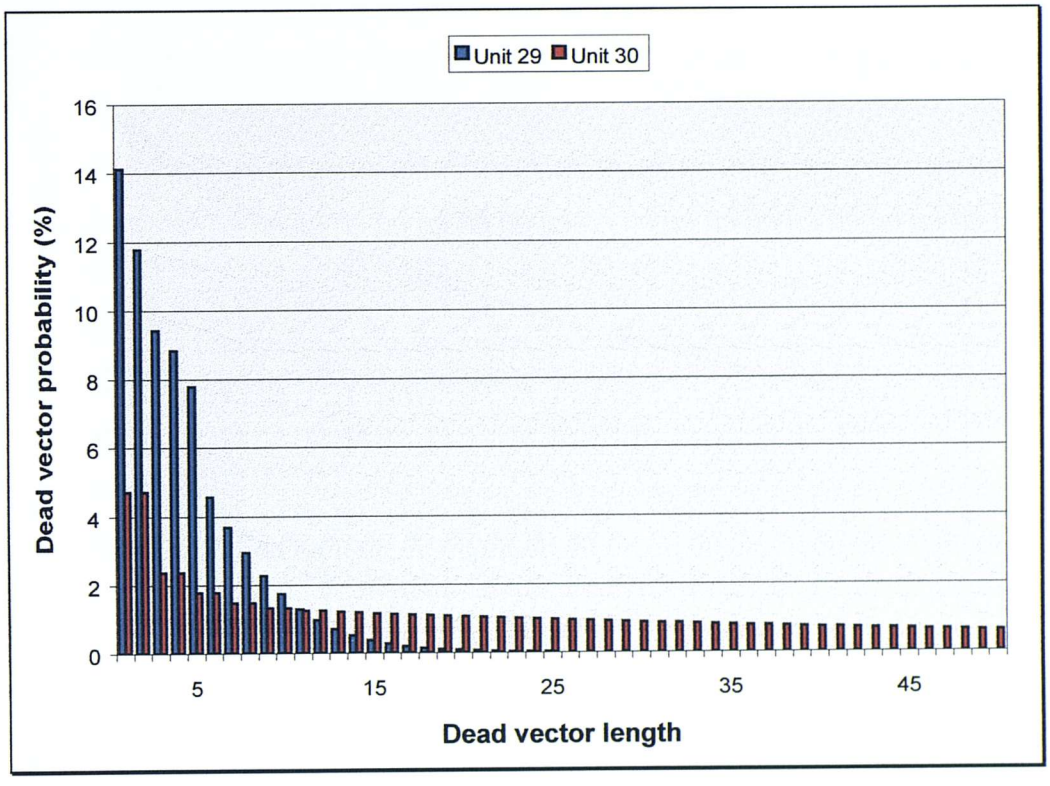


Figure 5a Delay optimised FFT subtractor dead vector probabilities

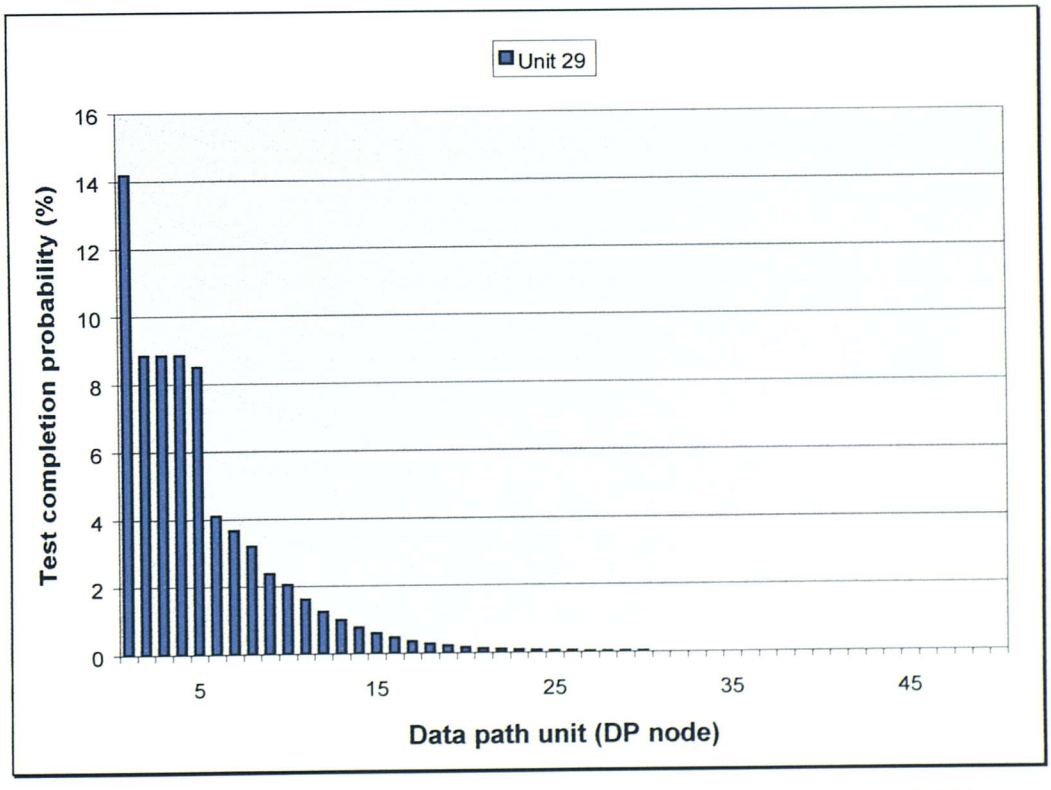


Figure 5b Area optimised FFT subtractor dead vector probabilities

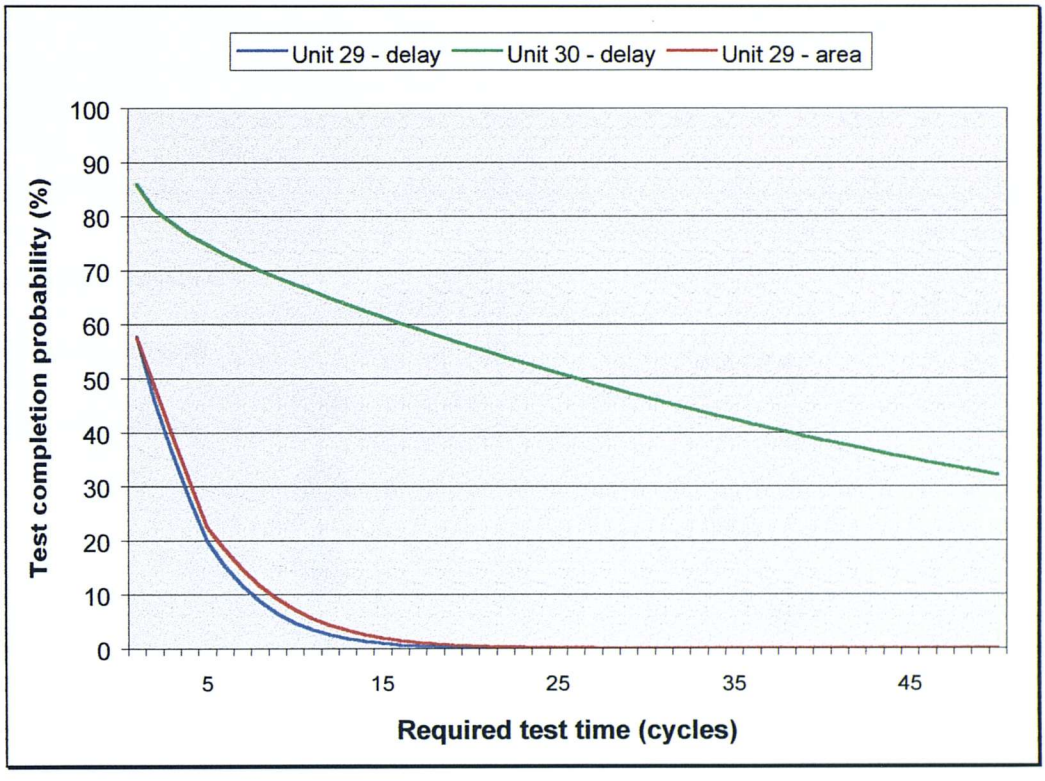


Figure 6 Combined FFT subtractor test completion probability vs. test time

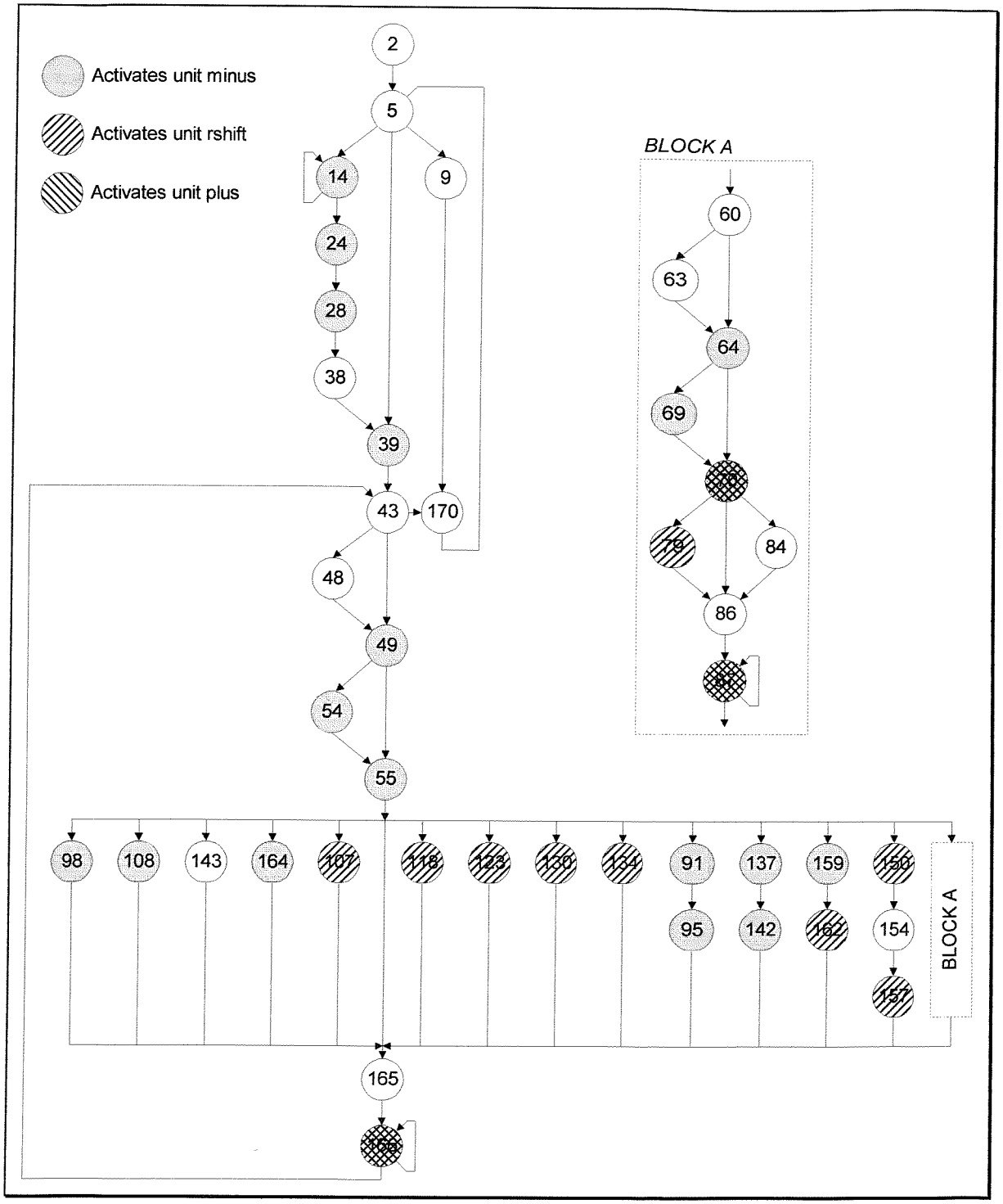


Figure 7 Area optimised FRISC control graph

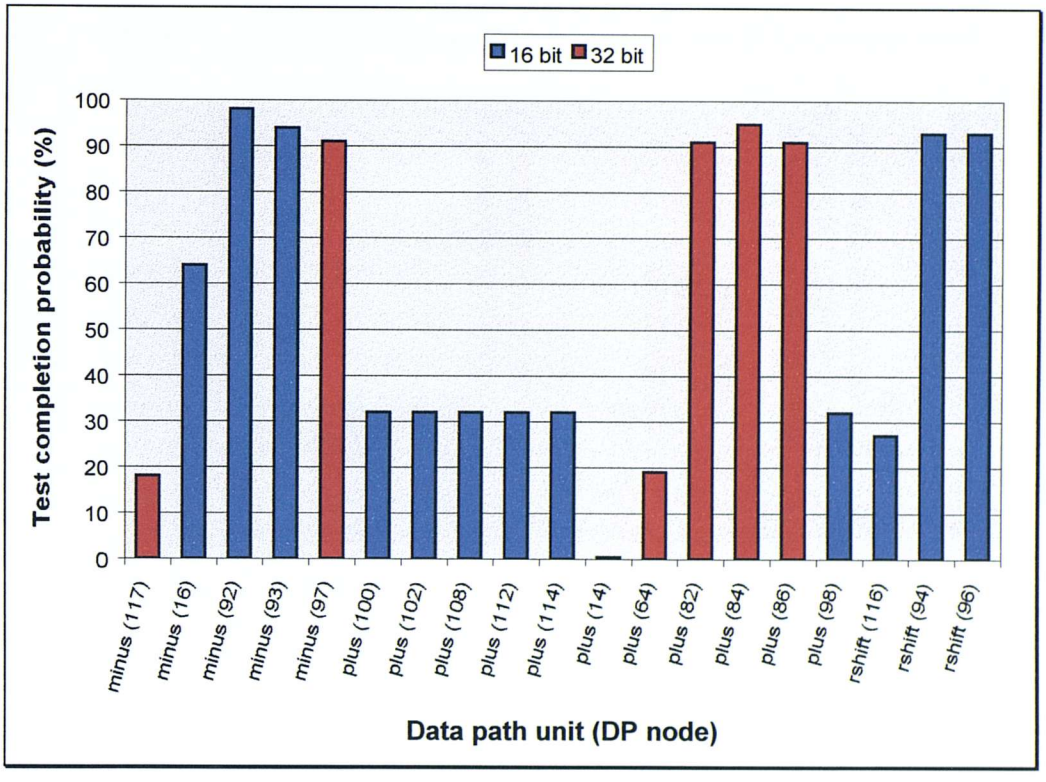


Figure 8a Delay optimised FRISC test completion probability per unit

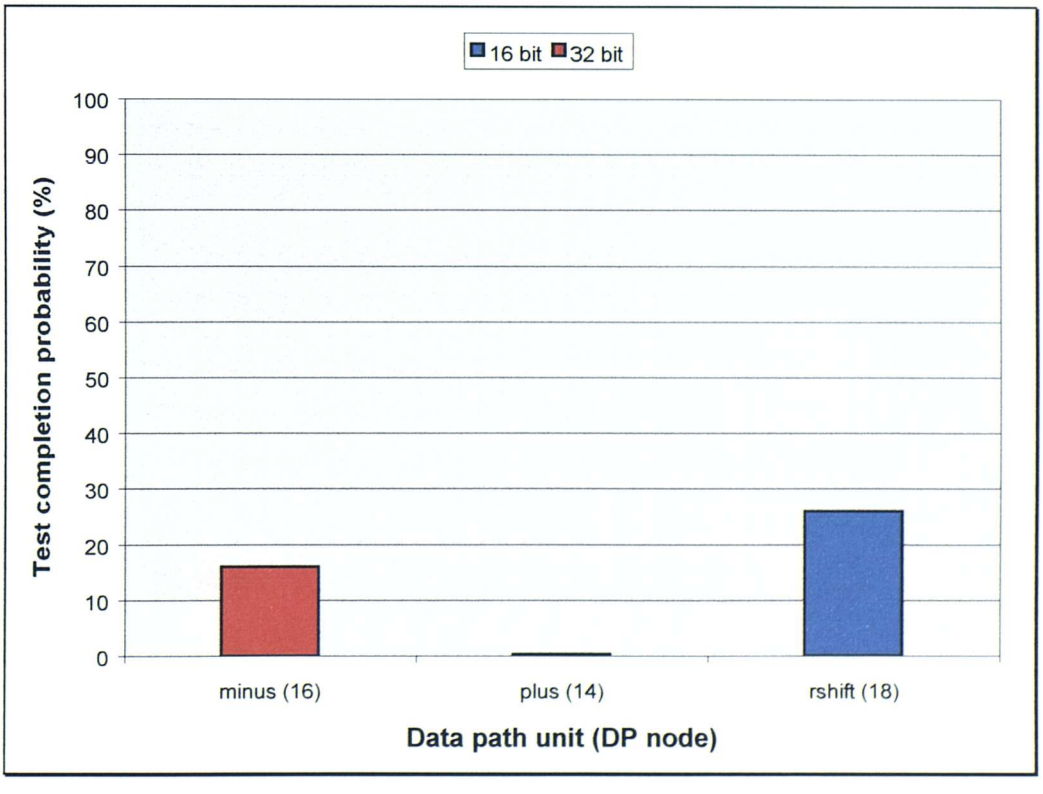


Figure 8b Area optimised FRISC test completion probability per unit

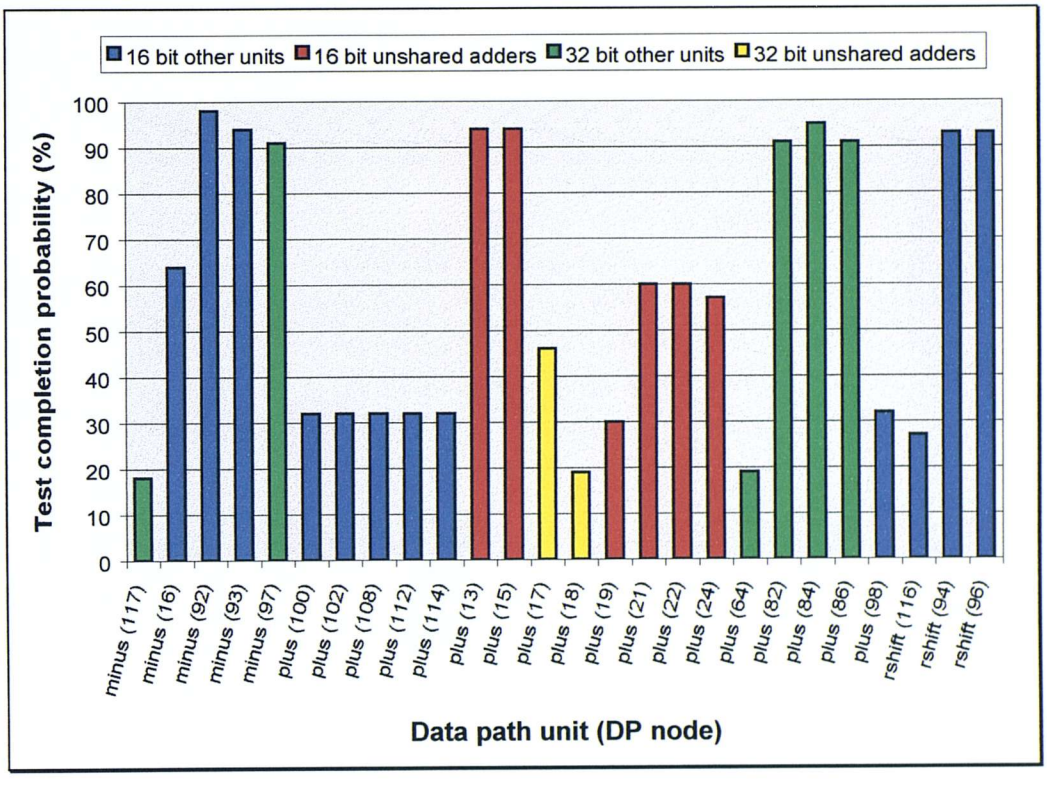


Figure 9a Delay optimised FRISC test completion probability per unit with unshared adder

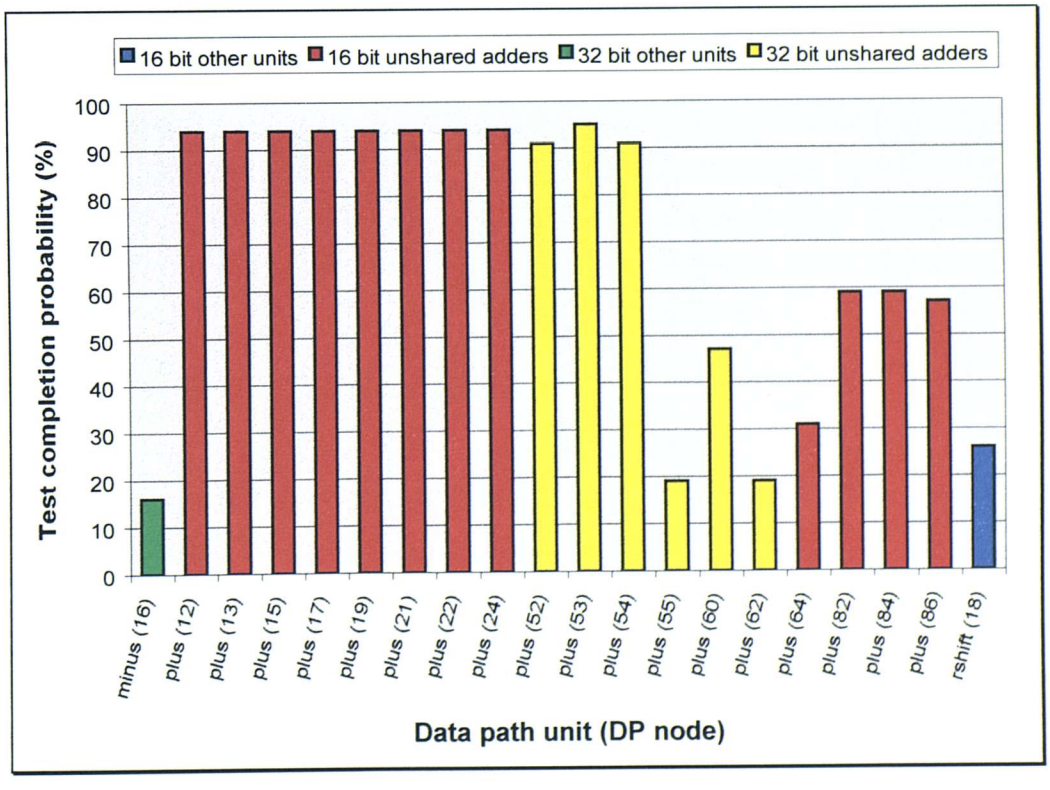


Figure 9b Delay optimised FRISC test completion probability per unit with unshared adder

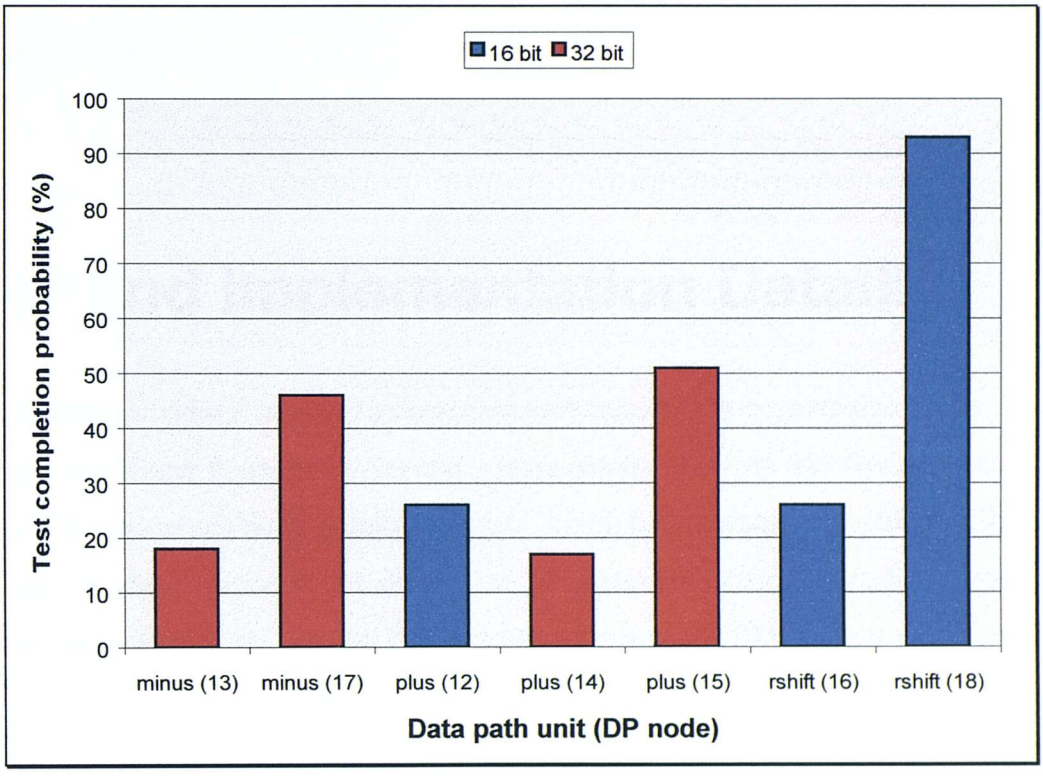


Figure 10 Area optimised FRISC test completion probability per unit optimised for online testing

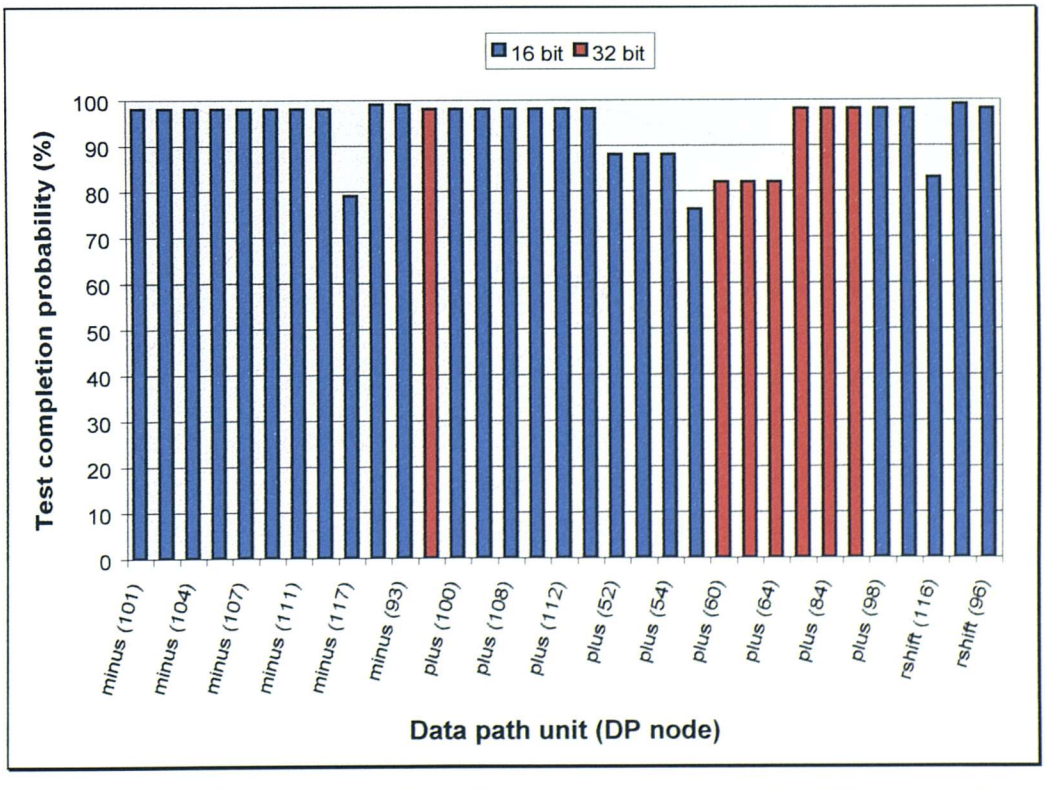


Figure 11 Unoptimised FRISC test completion probability per unit

Appendix B

User and Implementation Details

This appendix provides a range of information pertaining to the expanded modules developed in Chapter 5, and some general system details. It is split into five sections. Section B.1 summarises the expanded template design flow discussed in detail in Chapter 5, providing a quick reference for the design and integration of expanded modules, macro operators and macro ports. Section B.2 lists the behavioural VHDL source code for the four benchmark examples analysed in Chapter 5, while section B.3 outlines a number of file formats created during the development of the expanded module capability, namely the extended ICODE (**.xic**), the design data format (**.ddf**) and the ICODE instruction database, along with a description of the options available in the MOODS initialisation file (**moods.ini**). Section B.4, lists command line switches added to MOODS and VHDL2IC since the MOODS user manual [119] was written. Finally, section B.5 describes the MOODS and VHDL2IC source directory structure including a brief summary of the contents of each source file.

B.1 Creating and Using Expanded Modules

This sections summarises the various steps in the creation of expanded module templates, their integration into the system, and their use within MOODS to expand operators in a behavioural design.

B.1.1 Creating Expanded Template Files

There are four stages in the creation of an expanded module template (**.xmt**) file shown in Figure B.1 (described in detail in section 5.3.2):

1. Develop a single VHDL process describing the expanded module in as simple a form as possible to allow for post-expansion optimisation. Compile this using VHDL2IC with the **/no_sigs** switch to remove all signal shadow registers.

2. Modify the extended ICODE file (or omit the first stage and write the ICODE from scratch), removing any extraneous elements, in particular the implicit VHDL process loop.

3. Load the ICODE file into MOODS and optimise if necessary to obtain the desired expanded module structure. This need only be done for macro ports, which may be expanded after optimisation; most expanded modules should rely on MOODS to optimise their structure after expansion. The expanded template file (**.xmt**) is created using the *sxm* command in MOODS. Figure B.2 shows a typical sequence of events when saving a template, with user-entered elements shown in bold.

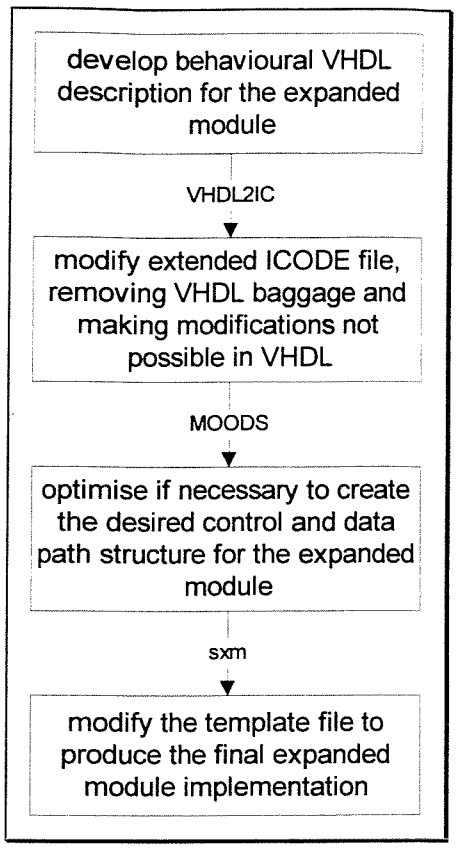


Figure B.1 Expanded module creation design flow

```
MOODS "addsplit32" --> sxm

  Expanded module template "addsplit32"
    DP pin "input1"     -> I/O port variable "inp1"
    DP pin "input2"     -> I/O port variable "inp2"
    DP pin "output1"    -> I/O port variable "outp"
  Enter template module function (name/number) : f_add
  Enter template maximum valid primary width : 32
  Enter template minimum valid primary width (32) : 25
    ----> creating expanded module template file (ACW) addsplit32.xmt

MOODS "addsplit32" -->
```

Figure B.2 Saving an expanded template

MOODS maps each top-level port onto an equivalent low-level module pin, defining how the expanded structure will connect into the data path when expanding a data path unit, thus in the example, ports *inp1* and *inp2* are the two module input pins, and *outp* is the single output. Additional details are obtained from the user to define the module

function and the range of operator widths to which it is targeted. In Figure B.2, the expanded module implements an add function suitable for expanding 25 to 32-bit adders.

4. The final stage involves minor modifications to the template file itself. This will rarely be necessary as most structural modifications can be made using MOODS transformations in stage 3, however there may be occasions where it is desirable to adjust the template header information (entered in the *sxm* command above), or investigate the effects of minor alterations to the module structure without going through the complete design flow.

B.1.2 Integrating Templates into MOODS

Once a template file has been created, it can then be integrated into the current MOODS configuration, enabling it to be accessed during optimisation. The expanded template library loads in template files described in the *ExpandedModules* section of the MOODS initialisation file (**moods.ini**), as illustrated in Figure B.3. Template files must reside in directories in the MOODS library path, specified by the *MOODS_LIB* environment variable. This also defines the location of other MOODS support files including the module and ICODE instruction databases and the VHDL2IC function library.

```
[ExpandedModules]
; Split operator expanded templates
SplitAdder       = addsplit8.xmt, addsplit16.xmt, addsplit32.xmt
SplitSubtractor  = subsplit8.xmt, subsplit16.xmt, subsplit32.xmt
SplitMultiply    = splitmult16b2.xmt, splitmult32b2.xmt

; Macro ICODE cells for macro ports and macro operators
RomRead Macro A  = romrd11_8a.xmt, romrd14_8a.xmt, romrd8_24a.xmt
FixedPointMult   = fpmult12_10.xmt, fpmult16_14.xmt
```

Figure B.3 Template library configuration

Several templates must be created for various bit-width ranges if complete coverage for a particular operation type is required. An entry should be provided in **moods.ini** for each different module topology, listing the set of template files for the supported bit-width ranges. In Figure B.3 five sets are defined, three of which implement split operators, with the remaining two describing a *RomRead* macro port, and a fixed-point multiplier.

This is all that is necessary to utilise templates designed to expand already existing low-level modules (adders, subtractors etc.), however, when creating a new macro operator or macro port type, three other steps are required for full integration into the design flow:

1. To create a new operator type, an associated ICODE instruction must be added into the ICODE database file (**insts.icd**), describing the instruction name, number, and function type, and its I/O parameter specification (see section B.3.3 for details of the instruction database file). The function type specifies the low-level module function required to implement this particular instruction and must match the module function entered when creating the template files (Figure B.2).

2. A suitable low-level module library entry must also be created, describing a combinational implementation of the operator for use in the initial data path created by MOODS. This need not have any physical realisation and will therefore only act as a placeholder in the data path until expansion occurs. It is possible, however, to create an associated structural implementation, thus effectively adding a user-defined low-level cell. Note that library entries with no corresponding combinational module should be marked as such via a special module status flag, enabling MOODS to identify placeholder modules and automatically expand them before outputting the final structural netlist.

3. The final step is to add a VHDL function header into the VHDL2IC library file (**packages.vhd**) enabling the macro to be accessed directly from a behavioural VHDL source description. Functions added to the *macro_ops* and *macro_ports* packages are translated into equivalent ICODE instructions, rather than compiling them as separate ICODE sub-modules, with the same ICODE parameter list as defined by the function header. The library can also include a simulation model of the operator in the package body for use during design development.

B.1.3 Expanding Modules During Optimisation

During the synthesis of a behavioural description there are three mechanisms available for expanding data path modules:

1. The MOODS *aob* command starts the automatic module expansion procedure, designed for the general hierarchical expansion of arithmetic modules and macro operators. This supports two modes of operation: *automatic* expands all modules using the first suitable entry in the template library without any further user interaction; and *prompted* provides the user with a list of the valid templates for each expansion, allowing a manual selection to be made. Figure B.4 shows an example session during which the multipliers in a design comprising one 16 and one 8-bit operator are expanded. The procedure first prompts the user for a function type and a minimum bit-width, which specify the data path unit type and the smallest unit to expand. It then asks whether prompted or automatic mode is required and proceeds with the expansion. In the example, automatic mode is chosen, which first splits the single 8-bit multiply into four 4-bit units and the 16-bit multiply into four 8-bit units. These new 8-bit multipliers are then hierarchically expanded into 4-bit multipliers, thus at the end of the procedure the maximum multiplier width will be 4 bits.

```
MOODS "test" --> aob

   Enter DP function type to split (name/number) [all]: f_multiply
   Enter minimum cell width to split [0]: 8
   Prompt on each expansion (y/n) [no] ? n

    ----> Expanding DP node 13 with template split_mult_8b2
    ----> Expanding DP node 14 with template split_mult_16b2
    ----> Expanding DP node 27 with template split_mult_8b2
    ----> Expanding DP node 28 with template split_mult_8b2
    ----> Expanding DP node 30 with template split_mult_8b2
    ----> Expanding DP node 32 with template split_mult_8b2

      --- COST FUNCTION VECTOR ---
      CRITERION | area (sq um)   | T delay (ns)
      PRIORITY  |       1        |       2
      INITIAL   | 87878.0 sq um  |  1696.2 ns
      TARGET    |     0.0 sq um  |     0.0 ns
      PREVIOUS  |     0.0 sq um  |     0.0 ns
      PRESENT   |141860.0 sq um  |  2114.7 ns
      ESTIMATE  |     0.0 sq um  |     0.0 ns

MOODS "test" -->
```

Figure B.4 Automatic hierarchical module expansion

2. The *aom* command accesses the automatic macro expansion routine, which automates the above process for all modules marked in the module library as macros, ie. all those

that do not have a physical combinational implementation. This method does not require any further information and simply expands all as-yet unexpanded macro modules. Note that *aom* is also executed automatically prior to outputting the structural netlist, thus ensuring that no dummy placeholder modules are present in the final implementation. This, however, does not allow for any post-expansion optimisation steps.

3. Finally, data path units may be individually expanded via manual application of the *split module* transformation, accessed through the standard manual transformation process described in the MOODS user manual [119]. This prompts for a single data path unit and then lists suitable templates from which one can be selected for expansion.

B.2 Benchmark VHDL Listings

Listing B.1 DHRC2

```
1  -- VHDL description of Differential Heat Release Computation
2  --
3  -- Author: Loganathan Ramchandran, University of California, Irvine
4  --           Translated from Silage description [CaSv93]
5  --
6  -- Last Modified: 27 Jan 95  By Preeti R. Panda
7  --                          (removed VSS specific constructs)
8  --
9  -- Verification Information:
10 --
11 --                   Verified     By whom?           Date        Simulator
12 --                   --------   ------------       --------    ------------
13 --  Syntax             yes      Preeti R. Panda     27 Jan 95    Synopsys
14 --  Functionality      No
15
16 entity DHRC is
17   port (  go       : in bit;
18           a0_in    : in bit_vector(15 downto 0);
19           a1_in    : in bit_vector(15 downto 0);
20           a2_in    : in bit_vector(15 downto 0);
21           p339_in  : in bit_vector(15 downto 0);
22           p340_in  : in bit_vector(15 downto 0);
23           p_in     : in bit_vector(15 downto 0);
24           HOST_OUT : out bit_vector(15 downto 0)
25        );
26 end DHRC;
27
28
29 architecture aDHRC of DHRC is
30   subtype Memory is ram_16;
31 begin
32   process
33     variable V    : Memory(468 downto 0);
34     variable P    : Memory(468 downto 0);
35     variable DV   : Memory(468 downto 0);
36     variable BV1  : Memory(468 downto 0);
37
38     variable A0 : bit_vector(15 downto 0);
39     variable A1 : bit_vector(15 downto 0);
40     variable A2 : bit_vector(15 downto 0);
41
42     variable H1 : bit_vector(31 downto 0);
43     variable H2 : bit_vector(15 downto 0);
44     variable H3 : bit_vector(15 downto 0);
45     variable H4 : bit_vector(31 downto 0);
46     variable H5 : bit_vector(15 downto 0);
47     variable H6 : bit_vector(15 downto 0);
48     variable H7 : bit_vector(15 downto 0);
49     variable H8 : bit_vector(31 downto 0);
50     variable H9 : bit_vector(15 downto 0);
51
52     variable S  : bit_vector(31 downto 0);
53     variable S1 : bit_vector(31 downto 0);
54     variable S2 : bit_vector(31 downto 0);
55
56
57     variable CV : bit_vector(15 downto 0);
58     variable DP : bit_vector(15 downto 0);
59
60     variable D  : bit_vector(15 downto 0);
61     variable I  : integer;
62     variable J  : integer;
63     variable K  : integer;
64     variable temp : integer;
65   begin
66
67     wait on go ;          -- start when go edge occurs
68
69     a0 := a0_in;
70     a1 := a1_in;
71     a2 := a2_in;
72     p(339) := p339_in ;
```

```
 73      p(340) := p340_in ;
 74
 75      i := 340;
 76      while (i < 468) loop
 77        j := i + 1;
 78        k := i - 1;
 79        p(j) := P_IN;
 80        d  := p(j)  - p (k);
 81        dp := rshift (d,2);
 82        h1 := a0 * V(i);
 83        h2 := rshift(h1,2);
 84        h3 := a1+ h2;
 85        h4 := h3 * P(i);
 86        s2 := dp * v(i);
 87        h7 := rshift(s2,4);
 88        h5 := rshift(h4,4);
 89        cv := h5 + a2;
 90        s1 := p(i) * dv(i);
 91        h6 := rshift(s1,4);
 92        s  := h6 + h7;
 93        h8 := s * cv;
 94        h9 := rshift(h8,4);
 95        bv1(i) := h9 + h6;
 96
 97        host_out <= bv1(i);       -- update output
 98        wait for 100 ns;
 99
100        i := i + 1;
101      end loop;
102
103    end process;
104  end aDHRC;
```

Listing B.2 FFT2

```
1  -- VHDL description of Fast Fourier Transform design example
2  --
3  -- Author: Loganathan Ramchandran, University of California, Irvine
4  --
5  -- Last Modified: 27 Jan 95  By Preeti R. Panda
6  --                          (removed VSS specific constructs)
7  -- Verification Information:
8  --
9  --                   Verified     By whom?           Date        Simulator
10 --                   --------   ------------       --------    ------------
11 --  Syntax             yes      Preeti R. Panda     27 Jan 95   Synopsys
12 --  Functionality      No
13
14 -- Modified for MOODS by acw
15
16 entity FFT is
17   port ( go            :   in bit;              -- start it all off
18          M             :   in integer;          -- number of points
19          N             :   in integer;          -- 2 ** number of points
20          sig_real_inp  :  in integer;
21          sig_imag_inp  :  in integer;
22          sig_real_outp :  out integer;
23          sig_imag_outp :  out integer
24        );
25 end FFT;
26
27
28 architecture ARCH_TIMER of FFT is
29       subtype Memory is RAM_8(0 to 1023);              -- MOODS memory cell
30 begin
31   main_p: process
32     variable    let     : integer;
33     variable    windex  : integer;
34     variable    wptrind : integer;
35     variable    i,j,l,k : integer;
36     variable    lower   : integer;
37
38     variable    SigReal : Memory;
39     variable    SigImag : Memory;
40     variable    WReal   : Memory;
41     variable    WImag   : Memory;
42
43     variable    Wptr_Real : integer;
44     variable    Wptr_Imag : integer;
45     variable    xuReal : integer;
46     variable    xuImag : integer;
47     variable    TempReal : integer;
48     variable    TempImag : integer;
49     variable    TmReal : integer;
50     variable    TmImag : integer;
51     variable    xlReal : integer;
52     variable    xlImag : integer;
53     variable    xmReal : integer;
54     variable    xmImag : integer;
55
56     variable    temp1 : integer;
57     variable    temp2 : integer;
58
59   begin
60
61     wait on go;              -- wait for go edge
62
63     i := 0;                  -- load all the signal values
64     while(i < N) loop
65       SigReal(i) := sig_real_inp;
66       SigImag(i) := sig_imag_inp;
67     end loop;
68
69
70     let := N;
71     windex := 1;
72     wptrind := 0;
```

```
73
74      l := 0;
75      while (l < M) loop
76
77        let  := rshift(let,1);  --  / 2;
78        j := 0;
79
80        while (j < let) loop
81          Wptr_Real := WREAL(wptrind);
82          Wptr_Imag := WIMAG(wptrind);
83          i := j;
84          while (i < N) loop
85            xuReal := SigReal(i);
86            xuImag := SigImag(i);
87
88            lower := i + let;
89            xlReal := SigReal(lower);
90            xlImag := SigImag(lower);
91
92            TempReal := xuReal + XlReal;
93            TempImag := xuImag + XmImag;
94
95            SigReal(i) :=  TempReal;
96            SigImag(i) :=  TempImag;
97
98            TmReal := xuReal - XlReal;
99            TmImag := xuImag - XmImag;
100
101           temp1 := TmReal * Wptr_Real;
102           temp2 := TmImag * Wptr_Imag;
103           TempReal := temp1 - temp2;
104
105           temp1 := TmReal * Wptr_Imag;
106           temp2 := TmImag * Wptr_Real;
107           TempImag := temp1 - temp2;
108
109           SigReal(lower) := TempReal;
110           SigImag(lower) := TempImag;
111
112           i := lshift(let,1);  -- * 2;
113         end loop;
114
115         wptrind := wptrind + windex;
116         j := j + 1;
117       end loop;
118
119       windex := lshift(windex,1);  -- * 2
120       l := l + 1;
121     end loop;
122
123     j := 0;
124     i := 1;
125     while (i < (n-1)) loop
126       k := rshift(n,1);            -- / 2;
127
128       while (k <= j) loop
129         j := j - k;
130         k := rshift(k,1);          -- / 2;
131       end loop;
132
133       j := j + k;
134       if (i < j) then
135         tempReal  := SigReal(j);
136         tempImag  := SigImag(j);
137         SigReal(j) := SigReal(i);
138         SigImag(j) := SigImag(i);
139         SigReal(i) := tempReal;
140         SigImag(i) := tempImag;
141       end if;
142       i := i + 1;
143     end loop;
144
145     i := 0;                  -- load all the signal values
146     while(i < N) loop
```

```
147         sig_real_outp <= SigReal(i);
148         sig_imag_outp <= SigImag(i);
149       end loop;
150
151       wait for 100ns;           -- force outputs to update
152
153     end process main_p;
154   end  ARCH_TIMER;
```

Listing B.3 Diffeq

```
1  ------------------------------------------------------------------------
2  ------------------------------------------------------------------------
3  --     File Name     : diffeq.v
4  --     Author(s)     : P. Sridhar
5  --     Affiliation   : Laboratory for Digital Design Environments
6  --                     Department of Electrical & Computer Engineering
7  --                     University of Cincinnati
8  --     Date Created  : June 1991.
9  --     Introduction  : Behavioral description of a differential equation
10 --                     solver written in a synthesizable subset of VHDL.
11 --     Source        : Written in HardwareC by Rajesh Gupta, Stanford Univ.
12 --                     Obtained from the Highlevel Synthesis Workshop
13 --                         Repository.
14 --
15 --     Modified For Synthesis by Jay(anta) Roy, University of Cincinnati.
16 --     Date Modified       : Sept, 91.
17 --
18 --     Disclaimer    : This comes with absolutely no guarantees of any
19 --                     kind (just stating the obvious ...)
20 --
21 --      Acknowledgement : The Distributed Synthesis Systems research at
22 --                        the Laboratory for Digital Design Environments,
23 --                        University of Cincinnati, is sponsored in part
24 --                        by the Defense Advanced Research Projects Agency
25 --                        under order number 7056 monitored by the Federal
26 --                        Bureau of Investigation under contract number
27 --                        J-FBI-89-094.
28 --
29 ------------------------------------------------------------------------
30 ------------------------------------------------------------------------
31
32 entity diffeq is
33   port(aport, dxport : in integer;
34        xport, yport, uport : inout integer;
35        reset, ready : in bit;
36        nxt, over : out bit );
37 end diffeq;
38
39 architecture diffeq of diffeq is
40
41 begin
42   main_proc:process
43     variable three, five, a, x, y, dx, u : integer;
44     variable u1, u2, u3, u4, u5, u6, y1 : integer;
45   begin
46
47     if reset = '1' then
48       x := 0;
49       y := 0;
50       u := 0;
51       dx := 0;
52       a := 0;
53       three := 3;
54       five := 5;
55     else
56       wait on ready until ready = '1';
57
58       x := xport;
59       y := yport;
60       u := uport;
61       dx := dxport;
62       a := aport;
63
64       while x < a loop
65         u1 := u * dx;
66         u2 := five * x;
67         u3 := three * y;
68         y1 := u * dx;
69         x := x + dx;
70         u4 := u1 * u2;
71         u5 := dx * u3;
72         y := y + y1;
```

```
73          u6 := u - u4;
74          u := u6 - u5;
75
76          xport <= x;
77          yport <= y;
78          uport <= u;
79          nxt   <= '1';
80          nxt   <= '0';
81          wait for 100 ns;    -- update outputs
82        end loop;
83
84        over <= '1';          -- signal completion
85        wait for 100 ns;
86
87        over <= '0';
88        wait for 100 ns;
89      end if;
90
91    end process;
92  end diffeq;
```

Listing B.4 Ellip

```
1  ------------------------------------------------------------------------
2  ------------------------------------------------------------------------
3  --     File Name     : ellip.v
4  --     Author(s)     : Rajiv Dutta
5  --     Affiliation   : Laboratory for Digital Design Environments
6  --                     Department of Electrical & Computer Engineering
7  --                     University of Cincinnati
8  --     Date Created  : September 1991.
9  --     Introduction  : Behavioral description of the Elliptic Wave
10 --                     Filter, written in a synthesizable subset of
11 --                     VHDL.
12 --     Source        : P. Paulin and many others used this example
13 --                          in their papers. Was a benchmark at the
14 --                          Highlevel Synthesis Workshops.
15 --
16 --     Disclaimer    : This comes with absolutely no guarantees of any
17 --                     kind (just stating the obvious ...)
18 --
19 --      Acknowledgement : The Distributed Synthesis Systems research at
20 --                        the Laboratory for Digital Design Environments,
21 --                        University of Cincinnati, is sponsored in part
22 --                        by the Defense Advanced Research Projects Agency
23 --                        under order number 7056 monitored by the Federal
24 --                        Bureau of Investigation under contract number
25 --                        J-FBI-89-094.
26 --
27 ------------------------------------------------------------------------
28 ------------------------------------------------------------------------
29
30 entity elliptic is
31    port(inp,sv2,sv13,sv18,sv26,sv33,sv38,sv39 : inout bit_vector(15 downto 0);
32         over : out bit);
33 end elliptic;
34
35 architecture elliptic of elliptic is
36
37 begin
38   P1: process
39     variable outpi, sv18i, sv38i : bit_vector(15 downto 0);
40     variable op3, op32, op12, op20, op25, op21, op24 : bit_vector(15 downto 0);
41     variable op19, op27, op11, op22,op29, op9, op30 : bit_vector(15 downto 0);
42     variable op8, op31, op7, op10, op28, op41, op6 : bit_vector(15 downto 0);
43     variable op15, op35, op40, op4, op16, op36 : bit_vector(15 downto 0);
44
45     -- instruction.Execution
46   begin
47     over <= '0';
48     wait for 100 ns;
49
50     op3 := inp + sv2;
51     op32 := sv33 + sv39;
52     op12 := op3 + sv13;
53     op20 := op12 + sv26;
54     op25 := op20 + op32;
55     op21 := op25 * 2;
56     op24 := op25 * 2;
57     op19 := op12 + op21;
58     op27 := op24 + op32;
59     op11 := op12 + op19;
60     op22 := op19 + op25;
61     op29 := op27 + op32;
62     op9 := op11 * 2;
63     sv26 <= op22 + op27;
64     op30 := op29 * 2;
65     op8 := op3 + op9;
66     op31 := op30 + sv39;
67     op7 := op3 + op8;
68     op10 := op8 + op19;
69     op28 := op27 + op31;
70     op41 := op31 + sv39;
71     op6 := op7 * 2;
72     op15 := op10 + sv18;
```

```
73      op35 := sv38 + op28;
74      outpi := op41 * 2;
75      op4 := inp + op6;
76      op16 := op15 * 2;
77      op36 := op35 * 2;
78      sv39 <= op31 + outpi;
79      sv2 <= op4 + op8;
80      sv18i := op16 + sv18;
81      sv18 <= sv18i;
82      sv38i := sv38 + op36;
83      sv38 <= sv38i;
84      sv13 <= op15 + sv18i;
85      sv33 <= sv38i + op35;
86
87      over <= '1';
88      wait on inp;              -- update outputs and wait for new input
89    end process;
90
91  end elliptic;
```

B.3 File Formats

This section describes the format of three file types described in section 5.3.1, along with a summary of the various parts of the MOODS initialisation file (**moods.ini**).

B.3.1 Extended ICODE Format

The extended ICODE format (**.xic**) is a textual version of the original binary ICODE file described in the MOODS user manual.

ICODE represents the behaviour of a system at the register transfer level described as a number of non-recursive *modules* with the top level identified by a special *program* module. Each has an I/O parameter list allowing data to be passed into and out of a module, akin to software procedure parameters (passed by reference). A module comprises a set of ICODE *processes* describing a single ICODE *instruction* together with an associated *activation list*, which specifies the processes to execute once the current one has terminated. Thus the flow of execution through the ICODE is determined by the passage of activations from process to process. All instructions operate on ICODE *variables*, which are either explicitly declared in the module header, or are temporary variables. Explicit variables implement I/O ports, registers, aliases and memory blocks and are specified in terms of their bit-width and implementation (register, memory or alias), whereas temporary variables are not declared, thus their width is inferred from their connectivity.

In an extended ICODE file, module definitions split into two parts: the declaration part, which declares the module I/O ports and explicit variables; and the process part, listing the set of processes defining its operation. Each process contains an optional process label (for referring to the process in activation lists), an instruction and parameter list, an activation list (which if omitted defaults to the following process), and an optional *info* field containing application-specific data (such as source code line numbers, or execution probability values).

Instructions are generally of the form:

```
INSTRUCTION <input variable list>, <output variable list>
```

where the number of parameters is defined in the ICODE instruction database (see section B.3.3).

There are also a number of special instructions used to direct the flow of execution:

- IF and IFNOT instructions allow execution to conditionally branch along either the true (ACTT) or false (ACTF) activation list depending on the value of a single input variable.
- SWITCHON and DECODE test a single input variable against several possible choices (CASE), each of which contains a separate activation list.
- COUNT instructions operate on special counter variables allowing the variable to be incremented (or decremented for a down counter), and the value tested to determine which of the true or false activation lists should be taken. Note that the testing actually occurs before the increment.
- Concurrent threads of execution, initiated from an activation list with more than one target process, are re-combined with a COLLECT instruction. This takes as a parameter a single value specifying how many activations the COLLECT should receive before terminating.

Modules other than the top-level program are executed via a MODULEAP instruction, which takes as parameters a sub-module name and a list of variables to connect to the module I/O parameters. The MODULEAP activates the first process in the called module, only terminating when the appropriate ENDMODULE instruction has been reached. Listing B.5 shows an example ICODE file illustrating all of these features, and is followed by a complete language grammar in BNF format.

Listing B.5 Example extended ICODE file

```
 1 // top-level program definition
 2 PROGRAM        doverylittle  select, a, b, c, d, o
 3
 4 // top-level I/O port declarations
 5 INPORT         select      [1:1]
 6 INPORT         a           [0:1]
 7 INPORT         b           [0:1]
 8 INPORT         c           [0:1]
 9 INPORT         d           [0:1]
10 OUTPORT        o           [0:3]
11
12 // top-level variable/register declarations
13 REGISTER       choice      [0:1]
14 COUNTER        i           [1:4]          // an up counter
15
16 .start         EQ          #0, #1,  12
17                IF          12             ACTT if_true0  ACTF if_false0
18 .if_true0      MODULEAP    make_choice    a, b, choice   ACT endif0
19 .if_false0     MOVE        a,  choice
20
21 .endif0        MOVE        #0,  i
22
23 .countup       SWITCHON    choice
24                  CASE #1                  ACT case1a, case1b     {pt:0.33}
25                  CASE #0                  ACT case0a, case0b     {pt:0.33}
26                  DEFAULT                  ACT case_default       {pt:0.33}
27                  ENDCASE
28 .case0a        MULT        a, b, 17       ACT collect0
29 .case0b        MULT        c, d, 18
30 .collect0      COLLECT     2
31                PLUS        17, 18,  19
32                PLUS        19, i,  o      ACT endofchoices
33
34 .case1a        MULT        a, c,  14      ACT collect1
35 .case1b        MULT        b, d,  15
36 .collect1      COLLECT     2
37                PLUS        14, 15,  16
38                PLUS        16, i,  o      ACT endofchoices
39
40 .case_default  MOVE        i,  o
41
42 .endofchoices  COUNT       i, #15         ACTT start  ACTF countup
43
44                ENDMODULE   doverylittle
45
46
47 // make_choice sub-module definition
48 MODULE         make_choice  i1, i2,  o
49
50 // ***** I/O port declarations
51 INPORT         i1          [0:1]
52 INPORT         i2          [0:1]
53 OUTPORT        o           [0:1]
54
55                GR          i1, i2,  20
56                IF          20             ACTT if_true1  ACTF if_false1
57 .if_true1      MOVE        #3,  o         ACT endofmodule
58 .if_false1     MOVE        #0,  o
59
60 .endofmodule   ENDMODULE   make_choice
```

Extended ICODE grammar in BNF format

Extended ICODE description ::=
program_declaration
{ submodule_declaration }

act_list ::=
*label*_name { ',' *label*_name}

actf_list ::=
ACTF act_list

actt_list ::=
ACT act_list
| **ACTT** act_list

alias_declaration ::=
ALIAS *alias_var*_name *alias*_range **FROM** *parent_var*_name *var_sub*_range

collect_count ::=
integer

collect_inst ::=
COLLECT collect_count [info]

conditional_inst ::=
conditional_inst_name *cond*_var actt_list actf_list [info]

conditional_inst_name ::=
IF | **IFNOT**

constant ::=
'#' integer

count_inst ::=
COUNT *counter*_var ',' *increment*_constant ',' end_term actt_list actf_list

counter_declaration ::=
counter_type *counter*_name *counter*_range

counter_type ::=
COUNTER | **COUNTDN**

declaration ::=
io_port_declaration
| variable_declaration

declaration_part ::=
{ declaration [info] }

decode_inst ::=
DECODE *decode*_var [info]
{ **CASE** constant actt_list [info] }

general_inst[1] ::=
general_inst_name io_list [actt_list] [info]

general_inst_name ::=
EQ | **LS** | **LE** | **NE** | **GE** | **GR**
| **NOT** | **AND** | **OR** | **XOR**
| **NEG** | **PLUS** | **MINUS** | **MULT** | **DIV**
| **LSHIFT** | **RSHIFT** | **ROL** | **ROR**
| **MOVE** | **SETTRUE** | **HIGHZ**

index ::=
integer

info ::=
['{' { info_specifier ':' info_value } '}']

instruction ::=
general_inst
| memory_inst
| conditional_inst
| switch_inst
| decode_inst
| collect_inst
| moduleap_inst

[1] General instructions are defined in the ICODE instruction database and may be enhanced as required.

integer ::=
'**%**'*binary*_integer | '**&**'*octal*_ integer | *decimal*_ integer | '**$**'*hex*_ integer

io_list ::=
term { ',' term }

io_port_declaration ::=
io_port_type *io_port*_name *io_port*_range

io_port_type ::=
INPORT | **OUTPORT**

memory_inst ::=
memory_read_inst | memory_write_inst

memory_read_inst ::=
MEMREAD *memory*_var_name '[' *address*_term '],' *read*_var_name [info]

memory_write_inst ::=
MEMWRITE *write*_term ',' *memory*_var_name ',[' *address*_term ']'

moduleap_inst ::=
MODULEAP *module*_name io_list [info]

process ::=
['.' *label*_name] instruction

process_part ::=
{ process }

program_declaration ::=
PROGRAM *program*_name io_list [actt_list] [info]
declaration_part
process_part
ENDMODULE [*program*_name] [info]

ram_declaration ::=
RAM *ram_var*_name *data*_range **ADDRESS** *address*_range

range ::=
'[' *low_bit*_index ':' *high_bit*_ index ']'

register_declaration ::=
REGISTER *var*_name *var*_range

rom_declaration ::=
ROM *rom_var*_name *data*_range **ADDRESS** *address*_range

submodule_declaration ::=
MODULE *module*_name io_list [actt_list] [info]
declaration_part
process_part
ENDMODULE [*module*_name] [info]

switch_inst ::=
SWITCHON *switch*_var [info]
{ **CASE** constant actt_list [info] }
DEFAULT actt_list [info]

term ::=
constant | var

temporary_variable ::=
integer

var ::=
*var*_name | temporary_variable

variable_declaration ::=
register_declaration
| alias_declaration
| ram_declaration
| rom_declaration
| counter_declaration

Additional notes

- Each entry is considered to occupy one line unless extended using '\'.
- Comments may be included using the standard C++ delimiter '//'.

- Names may include any unreserved combination of alphanumeric characters (not including ICODE delimiters), reserved names being the ICODE keywords.

- General instructions are defined in the ICODE database, which also specifies the exact format of their parameter lists.

- Variables declared in the program module are visible throughout the entire design (unless overridden by a local declaration), sub-module variables are local to the sub-module.

- CASES in DECODES must be in sequential ascending order with no gaps within the sequence. Any missing cases default to the first choice.

- ROM declarations will include a bitmap field in the future.

- *Info* entries may contain any form of application-dependent information such as source line numbers, variables etc. Syntactically, everything within the braces is ignored. In MOODS, info records specify instruction activation probabilities ("**pt**", "**pf**") and loop iterations ("**its**").

B.3.2 Design Data Format

The design data format (**.ddf**) is a textual version of the original MOODS data structure (**.ds**) format, with several additions intended for creating expanded module templates (**.xmt**). A **.ddf** file stores the state of the internal data structures representing a design such that they can be exactly reproduced at a later date, ie. it stores a snapshot of the internal state of MOODS.

The file is a hierarchical collection of blocks, each of which represents some element of the data structure. At the outer level, the file splits into three block types:

1. *Module* blocks describe the instructions and control structure for each ICODE module in the design. There may be several modules in a description but one of these must represent a top-level program module.

2. The *data_path* block stores the entire data path structure for all the ICODE modules, thus there must only be one such block per file.

3. The *condition* block describes the boolean equations that link the control and data paths, again only one is allowed per file.

Note that at present, no low-level module library information is stored, thus a module must be created for each data path unit as the file is read and the data structures created. This will be tackled in the future with the inclusion of a *library* block, however omitting low-level information means that a degree of technology independence is provided, which is essential for expanded module templates.

Within each block, the contents of the data structure is described as a set of parameters and sub-block instantiations representing data structure fields, and linked lower level sub-structures. Listing B.6 shows an example of a simple **.ddf** file describing a design that multiplies either inputs *a* and *b*, or *c* and *d*, depending on the value of the *sel* input. When optimised this produces an implementation with one multiplier, shared between two instructions as illustrated in the example (lines 28-29).

Syntactically each block is defined by a type identifier and a number of type-dependent parameters, followed by its constituent parameters and sub-blocks. Parameters are described with a type identifier followed by a type-dependent parameter specification. For example in Listing B.6, the control graph is described by the *control_path* block (line 21), with node *c2* being both the start and end node for the ICODE module. This contains a single control node specification block describing a *general* control node (number *c2*), which executes an instruction group of three instructions and has a number of control node parameters defining the node activation signal, the state probability (which in this case has not yet been set), and a single feedback control arc. Note how the two multiply instructions (lines 28 and 29) are executed conditionally on signals *s9* and *s11* (defined on lines 114 and 115), with an equal probability of choosing either one.

Exact details of the various parameters may be implied from the MOODS data structures [3] and are also described in the comments in Listing B.6. Further syntax details are provided by the language grammar in BNF format after the listing.

Listing B.6 Example design data format file

```
module 1                                    // module number 1
{
  header {                                  // module header instruction
    instruction i1 {
      icode : program test1;                // ICODE: PROGRAM test1
      input : sel, a, b, c, d;              // input parameter list
      output : o;                           // output parameter list
      prob : 0;                             // inst probability - never for header
    }
  }
  variables                                 // module variable declarations
  {
    port v1 : sel[1:1] is u1;               // v1 is I/O port sel, in dp node u1
    port v2 : a[0:15] is u2;
    port v3 : b[0:15] is u3;
    port v4 : c[0:15] is u4;
    port v5 : d[0:15] is u5;
    register v6 : o[0:31] is u6;
    temp v9 : var_s7[0:0] is u7;
  }
  control_path  c2 : c2                     // control graph, start and ends on c2
  {
    general c2 {                            // control node number c2 definition
      signal : s8;                          // control activation signal
      prob : 0;                             // activation probability (not set)
      group g4 {                            // executing instruction groups
        instruction i2 : eq sel, #1, var_s7;
        instruction i5 : mult c, d, o when s11 prob 0.5; // conditional on s11
        instruction i4 : mult a, b, o when s9 prob 0.5;  // conditional on s9
      }
      feedback a1 : c2;                     // control arc back to c2
    }
  }
}

data_path                                   // the data path
{
  port u1 {                                 // a port unit, dp node u1
    width : [1:1];                          // one bit wide
    net n8 {                                // one output net connecting to u8
      source : output1[1:1] is v1 on i2;
      destination : input1[0:0] @ u8 is v1 on i2;
    }
  }
  port u2 {
    width : [0:15];
    net n9 {
      source : output1[0:15] is v2 on i4;
      destination : input1[0:15] @ u11 is v2 on i4;
    }
  }
  port u3 {
    width : [0:15];
    net n10 {
      source : output1[0:15] is v3 on i4;
      destination : input2[0:15] @ u11 is v3 on i4;
    }
  }
  port u4 {
    width : [0:15];
    net n11 {
      source : output1[0:15] is v4 on i5;
      destination : input1[0:15] @ u11 is v4 on i5;
    }
  }
  port u5 {
    width : [0:15];
    net n12 {
      source : output1[0:15] is v5 on i5;
      destination : input2[0:15] @ u11 is v5 on i5;
    }
  }
```

```
73    storage u6 {                          // a register, dp node u6
74      width : [0:31];                     // 32 bits wide
75      control : load_en[0:31] is v6 on i5; // stores value of var v6 on inst i5
76      control : load_en[0:31] is v6 on i4; // stores value of var v6 on inst i4
77    }
78    storage u7 {
79      width : [0:0];
80      control : load_en[0:0] is v9 on i2;
81      net n13 {
82        source : output1[0:0] is v9;
83        destination : s9 is v9;
84      }
85    }
86    functional u8 {                       // a functional unit, dp node u8
87      width : [0:0];                      // 1 bit wide
88      instruction : i2;                   // implements instruction i2 (EQ)
89      net n14 {                           // one output net to var v7, inst i2
90        source : output1[0:0] is v9 on i2;
91        destination : input1[0:0] @ u7 is v9 on i2;
92      }
93    }
94    functional u11 {                      // a functional unit, dp node u11
95      width : [0:15];                     // 16 bits wide
96      instruction : i4, i5;               // implements instructions i4 and i5 (*)
97      net n16 {                           // output net to register u6, var v6, i5
98        source : output1[0:31] is v6 on i5;
99        destination : input1[0:31] @ u6 is v6 on i5;
100     }
101     net n15 {                           // output net to register u6, var v6, i4
102       source : output1[0:31] is v6 on i4;
103       destination : input1[0:31] @ u6 is v6 on i4;
104     }
105   }
106   net n0 {                              // one non-output net to u8 on i2
107     source : #1 on i2;                  // #1 input for sel comparator
108     destination : input2[0:0] @ u8 on i2;
109   }
110 }
111
112 condition_list                          // condition equations
113 {
114   signal s9 : var_s7 on n7;             // result of i2 comparison, controls i4
115   signal s11 : /s9 on n6;               // inverse of s9 controlling i5
116 }
```

Design data format grammar in BNF format

Design Data Description ::=
    module_list
    data_path
    condition_list

activation_definition ::=
    activate | **feedback** arc_number '**:**' control_number [**when** condition] '**;**'

alias_declaration ::=
    alias variable_number '**:**' *alias_variable*_name *alias*_range **from** *variable*_name
        *variable_sub*_range **is** unit_number '**;**'

arc_number ::=
    ['**a**']integer

condition ::=
    signal_number
    | signal_specification

condition_list ::=
    condition_list '{'
        { signal_definition }
    '}'

constant ::=
    '#'integer

control_block ::=
    control_path '{'
        { control_node_definition }
    '}'

control_node_definition ::=
    control_type control_number control_specification

control_number ::=
    ['**c**']integer

control_parameter ::=
    loop_its ':' integer ';'
    | **prob** ':' float ';'
    | **signal** ':' signal_number ';'

control_specification ::=
    '{'
        { control_parameter }
        | { group_definition }
        | { instruction_definition }
        | { activation_definition }
    '}'

control_type ::=
    general
    | **fork**
    | **collect**
    | **conditional**
    | **call**
    | **dot**

counter_declaration ::=
    counter | **countdn** variable_number ':'
        *counter_variable*_name *variable*_range **is** unit_number ';'

data_path ::=
    data_path '{'
        { unit_definition }
        | { net_definition }
    '}'

group_definition ::=
    group group_number '{'
        { instruction_definition }
    '}'

header_definition ::=
    header '{'
        { *header*_instruction_definition }
    '}'

```
instruction_definition ::=
        instruction instruction_number instruction_specification

instruction_number ::=
        [ 'i' ]integer

instruction_parameter ::=
        icode ':' icode_instruction ';'
        | input ':' input_variable_list ';'
        | output ':' output_variable_list ';'

instruction_specification ::=
        '{' { instruction_parameter } '}'
        | ':' icode_instruction IO_variable_list ';'

memory_declaration ::=
        ram | rom variable_number ':' memory_variable_name width_range
                        address address_range is unit_number ';'

module_definition ::=
        module module_number '{'
                header_definition
                variable_block
                control_block
        '}'

module_list ::=
        { module_definition }

module_number ::=
        [ 'm' ]integer

net_definition ::=
        net net_number net_specification

net_end ::=
        variable_name slice_range @ unit_number | constant | signal_number
                        [ is variable_number ] [ on instruction_number ]

net_number ::=
        [ 'n' ]integer
```

net_parameter ::=
source : *source*_net_end ';'
| **destination :** *destination*_net_end ';'

net_specification ::=
'{' { net_parameter } '}'

port_declaration ::=
port variable_number ':' *port_variable*_name *variable*_range **is** unit_number ';'

register_declaration ::=
register variable_number ':'
*register_variable*_name *variable*_range **is** unit_number ';'

signal_definition ::=
signal signal_number ':' signal_specification ';'

signal_number ::=
['**s**']integer

signal_specification ::=
boolean_expression

temp_declaration ::=
temp variable_number ':' *temp_variable*_name *variable*_range **is** unit_number ';'

unit_definition ::=
unit_type unit_number unit_specification

unit_number ::=
['**u**']integer

unit_parameter ::=
width ':' integer ';'
| **cell** ':' integer ';'
| **boolean** ':' boolean_expression ';'

unit_specification ::=
'{' { unit_parameter } '}'

```
unit_type ::=
        storage
        | functional
        | boolean
        | interconnect
        | port
        | memory

variable_block ::=
        variables '{'
                { variable_declaration }
        '}'

variable_declaration ::=
        port_declaration
        | register_declaration
        | temp_declaration
        | alias_declaration
        | counter_declaration
        | memory_declaration

variable_number ::=
        [ 'v' ]integer
```

B.3.3 The ICODE Instruction Database - insts.icd

All non-control ICODE instructions are defined in the ICODE instruction database file, **insts.icd**, stored in a directory on the library search path. This contains an entry for each ICODE instruction defining:

- The instruction name, as referred to in the extended ICODE file.
- An instruction number, used in the binary ICODE file and as an instruction type identifier in the MOODS data structures.
- A data path module function number describing the function type required to implement the instruction in a low-level module.
- The instruction I/O parameter specification defining the number and size of the various input and output parameters.

Figure B.5 shows an extract from the database file defining nine instructions where an instruction definition occupies a single line, and comments are included using a preceding semicolon. The first three of the above parameters are specified straightforwardly in the first three fields, thus in the example, the PLUS instruction is defined as having instruction number 14, and data path function number 14 (ie. add). The PLUSC instruction on the other hand has the same function number but a different instruction number, thus the two instructions may be shared in a single data path unit, assuming that it provides suitable I/O pins.

Specification of the instruction I/O parameters is a two step process. First, the number of inputs and number of outputs is specified. Thus for PLUS there are 2 inputs and 1 output, while PLUSC has 3 inputs and 2 outputs. Next, the width of each parameter is specified in terms of its relationship to the instruction width. An instruction is considered as having a "primary width" generally (though not always) determined from the input parameter widths. For example, the primary width of an adder with 16-bit inputs will be 16 bits. A parameter width may be one of four different types:

1. *Primary* parameters (type *p*) define the primary width of the instruction from the connecting variables. If more than one primary is specified, the largest width is used. In the case of PLUS, both inputs are primary thus an instruction with both a 16 and an 8-bit input will have a primary width of 16 bits.

2. *Fixed* parameters (type *f* <*n*>) are always of the same width specified by the value *n*. For example, PLUSC has two primary inputs and a fixed 1-bit input for the carry in.

3. *Independent* parameters (type *i*) may be of any width being determined from the connecting variable. The ROMREAD instruction, for example, has a primary width specified as the data bus input, with an independent address width.

4. *Dependent* parameters (type *d* <*n*>) are a function of the primary width. The function type is specified by the value *n*, which at present supports three relationships: primary width, primary width plus 1, and twice the primary width. The output of the MULT instruction therefore, specified as type *d 3*, is twice the input (primary) width. Note that the dependent width is mainly used for inferring the size of temporary output variables; in the event of the connecting variable having a different width, this will be used in preference to the dependent value, thus a MULT instruction using only 32-bit I/O variables will have an output width of 32, and not 64 bits.

```
; ICODE instruction database file - insts.icd

plus      14  14  2 1   p p  d 2    ; add, two inputs, one output
minus     15  15  2 1   p p  d 2    ; subtract, two inputs, one output
mult      18  18  2 1   p p  d 3    ; multiply

; new icode instructions
plusc     150 14  3 2   p p f 1  d 1 f 1  ; add with carry in and out
minusc    151 15  3 2   p p f 1  d 1 f 1  ; minus with carry in and out

; user icode instructions for hand-crafted modules
bitflip   161 10011 1 1   p  d 1          ; bit reversing

; icode macro instructions/ports
romread   100 10000   2 3   i p  i f 1 d 1 ; addr, dbus, abus, noe, data
fpmult    160 10010   2 1   p p  d 1       ; fixed-point multiply
```

Figure B.5 The ICODE instruction database file

B.3.4 The MOODS Initialisation File - moods.ini

All configuration data required by MOODS is taken from a initialisation file, **moods.ini**. This is based on the standard Microsoft Windows **.ini** file format separating the file into a number of named sub-sections, within which are defined a set of parameters as illustrated in Figure B.6. MOODS includes a set of access functions enabling new sections and parameters to be included with new system features, thus providing a central repository for all configuration data. Note that undefined parameters and section names are simply ignored allowing for backwards compatibility.

At present, three sections are defined:

1. *Default* specifies general MOODS initialisation parameters where: *path* lists the default library search path, which is appended onto the path specified in the *MOODS_LIB* environment variable; and *options* lists default MOODS command line parameters used in addition to any specified on the true command line. In the example of Figure B.6 therefore, the default path is set up to search for files in D:\MOODS\LIBRARY and D:\MOODS\LIBRARY\EXPMODS; and the default command line options are set to **/std_logic** (use the VHDL *std_logic* type for bit vectors in the structural output), **/prn_al** (output a list of control arcs in the **.cg** log file), and **/ext_ic** (read in ICODE in the extended ICODE format (**.xic**) and not the original binary **.ic** format). Details of command line options may be found in the MOODS user manual [119].

2. *Technology* defines the technology libraries that make up the set of low-level modules stored in the central MOODS module library. Each technology library is described by a parameter specifying a library name (for user identification purposes), a library type, and a further set of type-dependent parameters. In the example, two technology libraries are defined both of which are based on the *GenericLibrary* type, requiring a single parameter that specifies the library database file. In this case the "Cypress FPGA Library" is defined in **cyfpga2.lib** while the "Macro operators" library is defined in **macro_ops.lib**.

3. *ExpandedModules* lists a set of template files to be loaded into the expanded template library, as described in section B.1.2.

```
[Default]
; Default search path and cli options
Path = D:\MOODS\LIBRARY;D:\MOODS\LIBRARY\EXPMODS
Options = /std_logic /prn_al /ext_ic

[Technology]
; This is the list of technology libraries - any number >1 is allowed
; format is <Library name>=<Type>, <Parameters dept. on lib. type>
Cypress FPGA Library = GenericLibrary, cyfpga2.lib
Macro operators = GenericLibrary, macro_ops.lib
```

Figure B.6 An extract from the MOODS initialisation file

B.4 New Command Line Switches

The MOODS user manual [119] lists a whole host of command line options for configuring various details of the execution of the system. These include such features as controlling the amount of information in the report files, and adjusting various aspects of the optimisation routines. During the course of the work described in this thesis, a number of additional switches have been incorporated into both MOODS and VHDL2IC, and are summarised in Table B.1 and Table B.2 below.

Switch	Description
/ext_ic	Read the behavioural description from an extended ICODE file. This automatically adds a **.xic** extension onto the specified design name.
/idle_info	Incorporate additional information in the **.daf** report file regarding the activity of data path units. This is used during latency analysis.
/ini <filename>	Specify an alternative MOODS initialisation file to the default **moods.ini**.
/no_sigs	Used in conjunction with the equivalent VHDL2IC switch, this allows the registers directly connected to inputs to be bypassed.
/share_all	Enables full register sharing as opposed to only sharing temporary registers.
/std_iogic	Uses the *std_logic* type for all signals in the output netlist in preference to *bit_vector*
/use_work	Creates the output netlist assuming that the low-level module library components are compiled in the VHDL *work* library, as opposed to the *moods* library.
/vhdl_debug	Adds ports to the output netlist replicating the control node activation signals to enable external tracking of the currently active state during operation.

Table B.1 New MOODS command line switches

Switch	Description
/no_collect	Do not create COLLECT instructions in complex arithmetic expressions.
/no_sigs	Disable the use of signal shadow registers.
/protect	Include an implicit PROTECT instruction in all pure *wait for timeout* statements possessing a positive timeout value.

Table B.2 New VHDL2IC command line switches

B.5 Source Directory Description

This section lists the structure and contents of the MOODS and VHDL2IC source directories. Figure B.7 shows the hierarchy of a standard MOODS source installation comprising eight top-level directories:

1. The *MOODS* directory holds the complete C++ source code for the main MOODS synthesis engine, with the module and technology libraries in the *TechLib* subdirectory, and the expanded modules and related functions in *ACW*. The individual source files in these directories are summarised below in Tables B.4 to B.7 .

2. *VHDL2IC* holds the source code for the VHDL2IC compiler. The VHDL2IC source files are described below in Figure B.3.

3. *PARSER* is the VHDL source-level optimiser.

4. *MOODSGUI* is the MOODS graphical user interface for Microsoft Windows.

5. *MOODSDAD* is the dead area analysis tool used to analyse the activity of data path units for the latency analysis paper [109] in Appendix A. The *WinDAD* subdirectory is a Windows front-end for this tool.

6. *Library* holds all the standard MOODS library files including **moods.ini, insts.icd,** technology library files, low-level structural modules, simulation models and the VHDL2IC function libraries. The *ExpMods* subdirectory holds all the expanded template files, with their ICODE source in the *XIC* subdirectory.

7. *Bin* is the location for all the compiled executables comprising the entire system.

8. *Etc* is a repository for various extra files including old module libraries, information files and any other useful items.

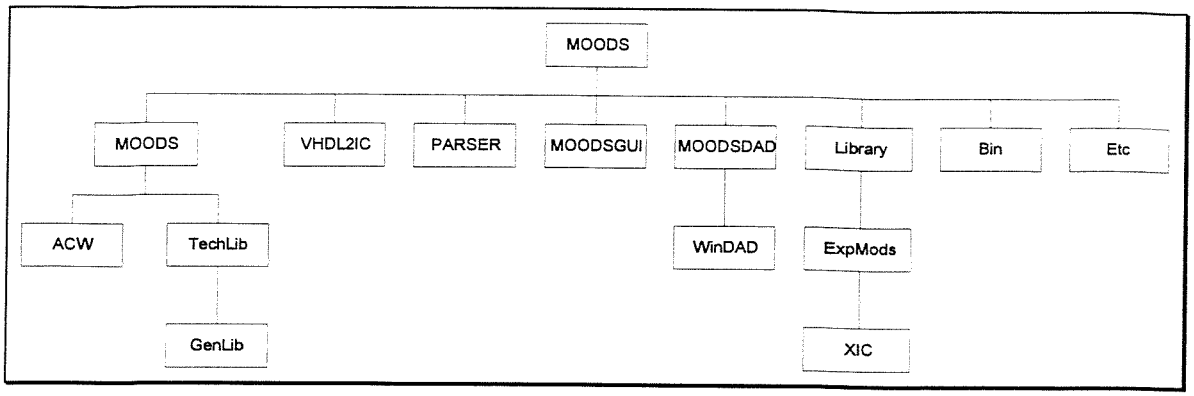


Figure B.7 MOODS source directory tree

Filename	Description
cbasics.cpp	Basic data structure manipulation routines
fns1.cpp	General routines - string and file manipulation
icnames.cpp	ICODE number/name conversion routines
lexic.cpp	VHDL lexical analyser routines
out_ic.cpp	Binary ICODE file outputting routines
out_xic.cpp	Extended ICODE file outputting routines
parse.cpp	VHDL parsing routines
pprint.cpp	Routines for printing the parsed VHDL data structures to a file for debugging
pprintic.cpp	Procedures for outputting the ICODE in a human readable form (pre-XIC)
vconv1.cpp	VHDL to ICODE conversion routines
vhdl2ic.cpp	Top-level VHDL2IC source
fn_prof.h	Function prototypes for all files
headers.h	Includes all the necessary headers for compilation
idefs.h	ICODE definitions - instruction numbers
istructs.h	ICODE data structures
pdefs.h	VHDL parser definitions - lexical tokens etc.
pstructs.h	VHDL parser data structures
system.h	General system-specific definitions - directory separator, version info etc.
makefile.linux	VHDL2IC makefile for GCC 2.7.2 under linux
vhdl2ic.ide	Borland C++ version 5 project file for VHDL2IC

Figure B.3 VHDL2IC directory source file descriptions

Filename	Description
Basics.cpp	Functions for general use in MOODS
Cost_fn.cpp	Procedures to set up and calculate the cost function
Dump_ds.cpp	Dump data structures to a **.ds** file
Estimate.cpp	Transform estimation procedures
Examine.cpp	Functions for interactively examining the data structures
Find_cp.cpp	Routines for determining the critical control path
Find_mfb.cpp	Routines for identifying feedback arcs in the control graph
Fns1.cpp	General utility functions specific to MOODS
Fns2.cpp	General utility functions - file and string manipulation - not MOODS specific
Gen_cont.cpp	Procedures for generating controller to data path control signals
Gen_dp.cpp	Data path generation based on an initial instruction schedule
Gen_igr.cpp	General control graph generation procedures
Gen_igr_ic.cpp	Binary ICODE file reader and initial control graph generation routines
Gen_inte.cpp	Post-optimisation interconnect generation routines
Graph_ma.cpp	General graph manipulation routines
List_man.cpp	General graph manipulation routines - lower level than graph_ma.cpp
Moods.cpp	Top-level MOODS program
Moodsfns.cpp	Top-level functions - decode cli switches and do command processing
Names.cpp	Routines to convert between reference numbers and text strings
Optim.cpp	Manual and simulated annealing optimisation routines
Perform.cpp	Performance calculation routines
Printlis.cpp	List and data structure printing routines
Sequence.cpp	Tailored heuristic optimisation
System_i.cpp	System-specific general routines - user I/O, memory allocation etc.
Tidy_ds.cpp	General routines for setting up and tidying data structures
Trans_te.cpp	Transformation test routines
Transfor.cpp	Transformation application routines
Vhdl_out.cpp	Routines for outputting a VHDL structural netlist from data structures
Defs1.h	General MOODS pre-processor definitions - file extensions, cli switches etc.
Fn_prof.h	MOODS directory function prototypes
Fns1.h	Function prototypes for fns1.cpp
Fns2.h	Function prototypes for fns2.cpp
Gendefs.h	General non system-specific definitions - macros, buffer lengths etc.
Includes.h	Includes all the necessary headers for MOODS compilation
Structs1.h	Data structure definitions
System.h	General system-specific definitions - directory separator, version info etc.
System_i.h	Headers for system_i.cpp
Makefile.linux	MOODS makefile for compilation under Linux with GCC 2.7.2 or later
Moods.ide	MOODS project file for Borland C++ version 5

Figure B.4 MOODS directory source file descriptions

Filename	Description
Acw1.cpp	Routines for inserting expanded modules into the data and control path
Aw_trans.cpp	Top-level expanded module test, estimate and transformation routines
Ddf_lex.cpp	Design data format lexical analyser class
Dump_ddf.cpp	Output data structures in **.ddf** format
Exp_template.cpp	Expanded template library class
Gen_igr_xic.cpp	Generate initial control graph from an extended ICODE file
Icode_db.cpp	ICODE instruction database class
Label_list.cpp	General extended ICODE label management class
Xic_lex.cpp	Extended ICODE lexical analyser class
Xic_parse.cpp	Extended ICODE parser, used by gen_igr_xic.cpp
Acw1.h	Headers for acw1.cpp
Ddf_lex.h	Design data format lexical analyser class header
Exp_template.h	Expanded template library class header
Icode_db.h	ICODE instruction database class header
Label_list.h	Label_list.cpp class header
Strtok1.h	Replacement class for C strtok routine with nesting and garbage collection
Xic_lex.h	Extended ICODE lexical analyser class header

Figure B.5 MOODS\ACW directory source file descriptions

Filename	Description
Mod_lib.cpp	Module library class - MmoduleLibrary
Tech_lib.cpp	Technology library abstract base class - MTechnoLib
Techno_i.cpp	Routines to interface between older MOODS C code and the new C++ module libraries
Units.cpp	Area, delay and power unit conversion and data entry routines
Boderror.h	General error exception class
Dynarr1.h	General dynamic array class template
Fixarr1.h	General fixed array class template
Lnklist1.h	General linked list class template
Mod_lib.h	MModuleLibrary class header
Tech_def.h	MTechnoLib class header
Tech_lib.h	General technology library header for accessing all module library routines
Techno_i.h	Function prototypes for techno_i.cpp
Units.h	Function prototypes for units.cpp

Figure B.6 MOODS\TechLib directory source file descriptions

Filename	Description
Gen_db.cpp	Generic library database core functionality - database management
Gen_ed.cpp	Generic library technology file editor class - database creation
Gen_lib.cpp	Generic technology library class MGenericLib - based on MTechnoLib
Genedit.cpp	Generic library technology file editor tool
Gen_db.h	Generic library database class header
Gen_ed.h	Generic editor class header
Gen_lib.h	MGenericLib class header
Gen_ed.ide	Library file editor project file for Borland C++ version 5
Makefile.linux	Library file editor makefile for GCC 2.7.2 under Linux

Figure B.7 MOODS\TechLib\GenLib directory source file descriptions

Appendix C

Additional Spectrum Analyser Details

This appendix contains additional information regarding the audio band spectrum analyser discussed in Chapter 7. It is organised in three sections: section C.1 describes the Cooley-Tukey Fast Fourier Transform algorithm central to the spectrum analyser; section C.2 provides pin-out data for each of the four main synthesised devices, together with data-sheet extracts for the video display and analogue to digital converter used; and section C.3 contains full behavioural VHDL source listings, and the initial C++ FFT implementations.

C.1 The Fast Fourier Transform

The fast Fourier transform (FFT) algorithm implemented in the spectrum analyser **FFTCont** and **FFTProc** designs is based on the popular Cooley-Tukey algorithm [112, 113, 115] which calculates the discrete Fourier transform (DFT) from N data samples in time $O(N \log_2 N)$. This is considerably more efficient than the direct implementation of the DFT which takes $O(N^2)$ to execute. It has the added advantage of being relatively simple and requires no additional storage other than the basic sample memory, which is transformed by the algorithm into its DFT.

Consider the discrete Fourier transform of N sampled data points f_j:

$$F_k = \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \qquad 0 \le k < N \qquad (1)$$

where $i = \sqrt{-1}$. This can be re-written as the sum of two DFTs formed from the even and odd elements of the original data set:

$$F_k = \sum_{j=0}^{\frac{N}{2}-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{\frac{N}{2}-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \qquad 0 \le k < N \qquad (2)$$

Now defining $W^k = e^{2\pi i/N}$ gives:

$$F_k = \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \qquad 0 \le k < N \qquad (3)$$
$$= F_k^e + W^k F_k^o$$

where F_k^e is the k[th] component of the DFT, length $N/2$, of the even components from the original data set f_j, and F_k^o the equivalent for the odd components. Notice that F_k^e and F_k^o are periodic in k with length $N/2$, whereas W^k repeats every N, however, using:

$$W^{k+N/2} = -W^k \qquad (4)$$

equation 3 becomes:

$$F_k = F_k^e + W^k F_k^o$$
$$F_{k+N/2} = F_k^e - W^k F_k^o \qquad 0 \le k < N/2 \qquad (5)$$

Now, this operation can be recursively applied to determine F_k^e and F_k^o:

$$F_k = F_k^{ee} + W^k F_k^{eo} + W^k \left(F_k^{oe} + W^k F_k^{oo}\right)$$
$$F_{k+N/2} = F_k^{ee} + W^k F_k^{eo} - W^k \left(F_k^{oe} + W^k F_k^{oo}\right) \qquad 0 \le k < N/2 \qquad (6)$$

where F_k^{ee} is formed from the even components of F_k^e, F_k^{eo} from the odd components of F_k^e, and so on. Restricting N to be an integer power of 2 allows this process to be repeated all the way down to the one-point transforms of length 1 which are simply the input data. Thus, each sampled data point maps onto some combination of even/odd subdivisions, ie. $F_k^{eeoee\ldots oee} = f_n$ for some n. To determine the nature of this mapping consider that the successive even/odd separations class the data according to each successive least significant bit of the sample index. So, considering a small example set, of say 8 points, the first application of equation 5 forms an even set from points with bit 0 clear (ie. 0,2,4,6) and an odd set from points with bit 0 set (1,3,5,7). At the next level down the subdivision is based on bit 1, so for example *ee* comprises points 0 and 4 (bits 1 an 0 clear), while *oe* is points 2 and 6 (bit 1 clear, bit 0 set). In general therefore, when the sets finally divide into a single data point, the actual sample index corresponding to a given even/odd ordering is obtained by setting *e* to 0, and *o* to 1, and reversing the bit pattern. Table C.1 shows the mappings for each of the 8 points in the example set.

By rearranging the input data into bit-reversed order, each stage in the iterative calculation of the FFT, starting from the one-point transform, simply needs to combine adjacent points to form the two-point transform via equation 5. The operation of the algorithm is most clearly illustrated with the aid of a signal flow graph to show which data points combine together over each successive iteration. Figure C.1 is such a graph for the 8 point data set. Here the algorithm starts from the far left, combining adjacent pairs to form the two-point transforms. These are then paired together forming a pair-of-pairs, which are combined with their adjacent pair-of-pairs to form the three-point transform, and so on.

Subdivision	Binary	Bit reversed
eee	000	0
eeo	001	4
eoe	010	2
eoo	011	6
oee	100	1
oeo	101	5
ooe	110	3
ooo	111	7

Table C.1 8 point bit reversal

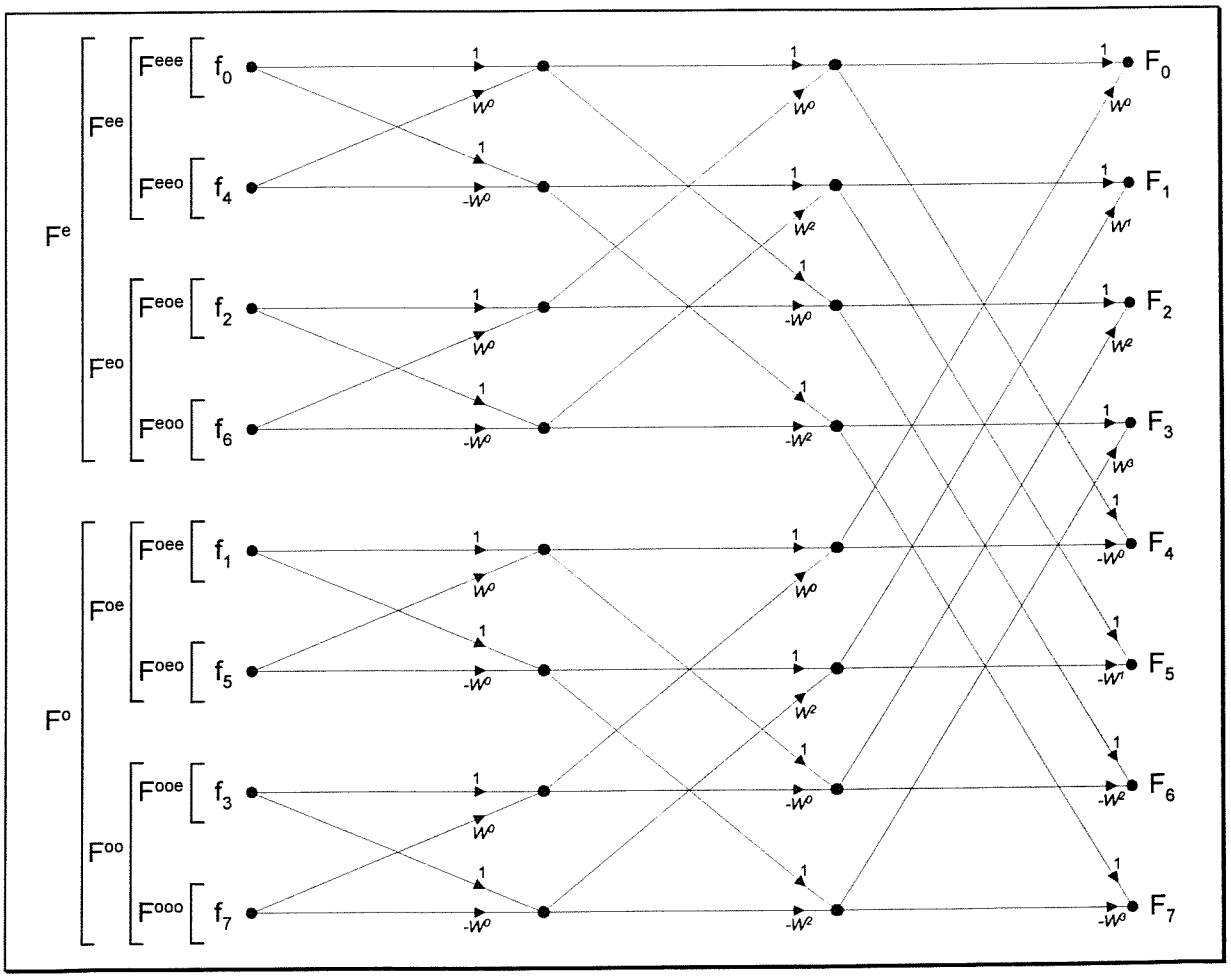


Figure C.1 Signal flow graph for 8 point FFT calculation

With the aid of this graph, an iterative FFT algorithm can be developed based on the following observations:

1. At the top most level, the data set subdivides $\log_2 N$ times along the horizontal axis forming an outer loop on l, where l ranges from 0 to $\log_2 N - 1$.

2. On each iteration the data is divided into even and odd sets of 2^{l+1} points.

3. Within each of these sets, data pairs k and $k + \Delta_k$ are combined according to equation 3 where $\Delta_k = 2^l$ and k loops through elements 0 to $\Delta_k - 1$ of the set.

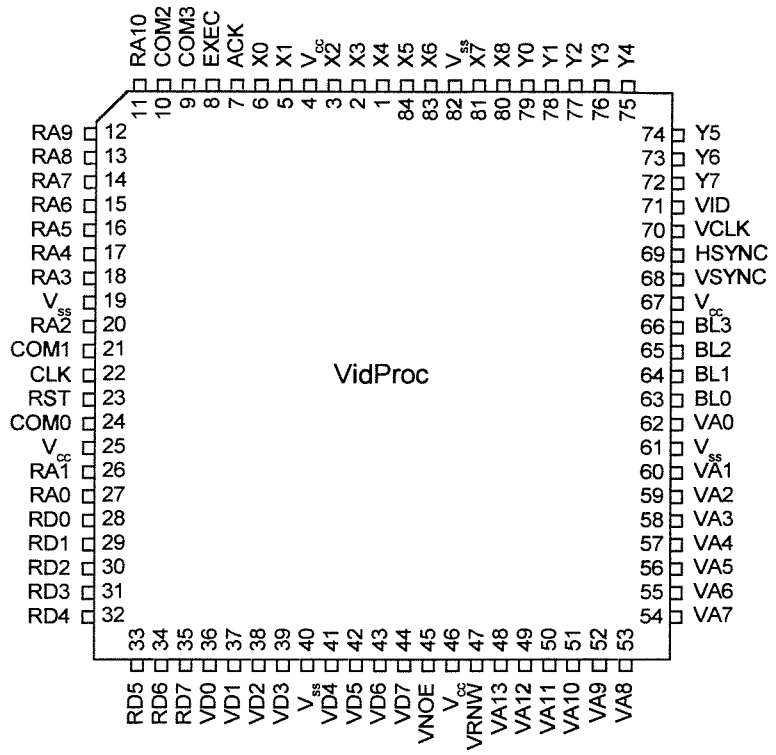
4. In conjunction with item 3, as k increases, W loops from 0 to $N/2 - 1$ in steps of $\Delta_W = 2^{(\log_2 N - 1) - l}$.

These are all brought together in the algorithm shown in pseudo-code form in Figure C.2. A full C++ version of this algorithm using double precision floating point arithmetic is given in Listing C.1. This forms the basis for the fixed point integer implementation in Listing C.2, which itself is used as the test bed for development of the VHDL version used in the spectrum analyser.

```
for (l=0; l<N; l++)
{
  Δ_W = 2^(log2N-1) - l;
  Δ_k = 2^l;
  W_idx = 0;
  for (k_base=0; k_base<Δ_k; k_base++)
    for (k=k_base; k<N; k+=2Δ_k) {
      j = k + Δ_k;
      combine k, j, W_idx;
      W_idx += Δ_W;
    }
}
```

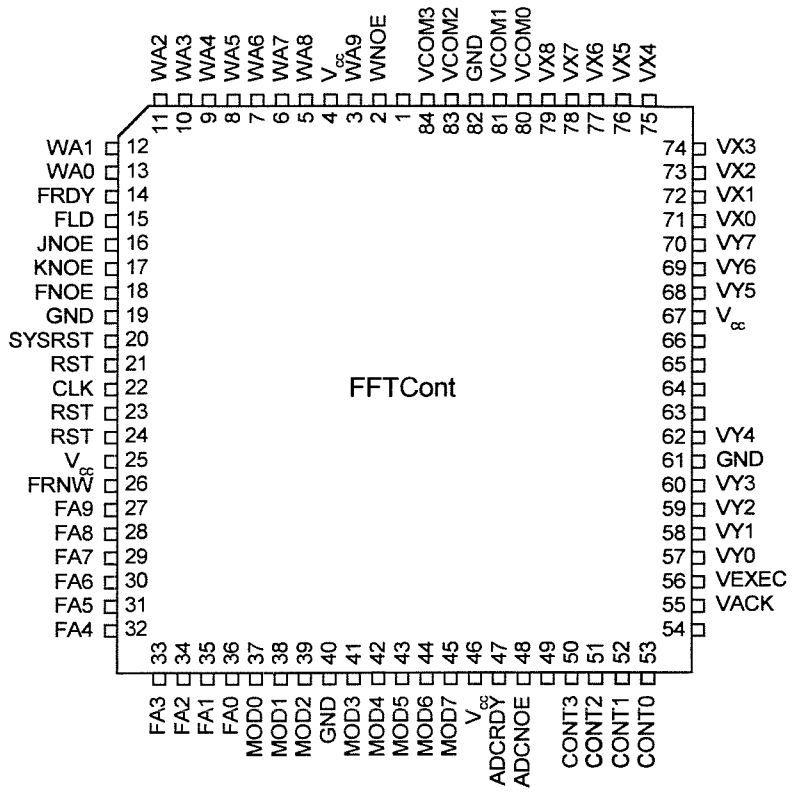
Figure C.2 Pseudo-code for FFT algorithm

C.2 Device Pin-outs and Data Sheets



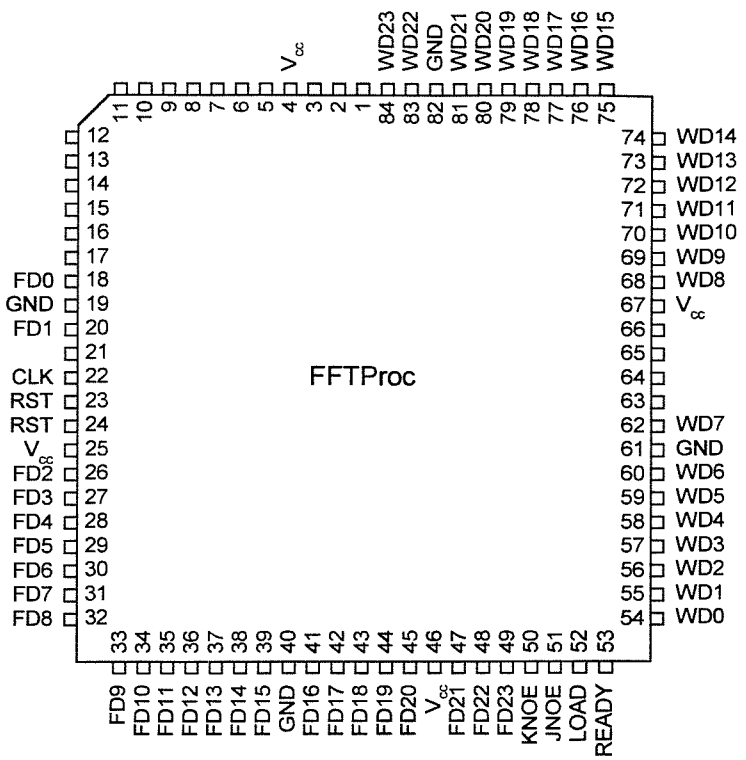
Pin name	Pin number	Description
CLK	22	System clock
RST	23	System reset
X8-X0	80-81,83-84,1-3,5-6	Video command x co-ordinate
Y7-Y0	79-72	Video command y co-ordinate and data value
COM3-COM0	9-10,21,24	Video command code
EXEC	8	Video command execute request
ACK	7	Video command acknowledge/executing
VA13-VA0	48-60,62	Video memory address bus
VD7-VD0	44-41,39-36	Video memory data bus
VNOE	45	Video memory output enable signal
VRNW	47	Video memory read/write signal
RA10-RA0	11-18,20,26-27	Resource ROM address bus
RD7-RD0	35-28	Resource ROM data bus
BL3-BL0	66-63	Display horizontal blanking adjust
VID	71	Display video signal drive
VCLK	70	Display video clock drive
HSYNC	69	Display horizontal sync drive
VSYNC	68	Display vertical sync drive

Figure C.3 Video processor pin diagram



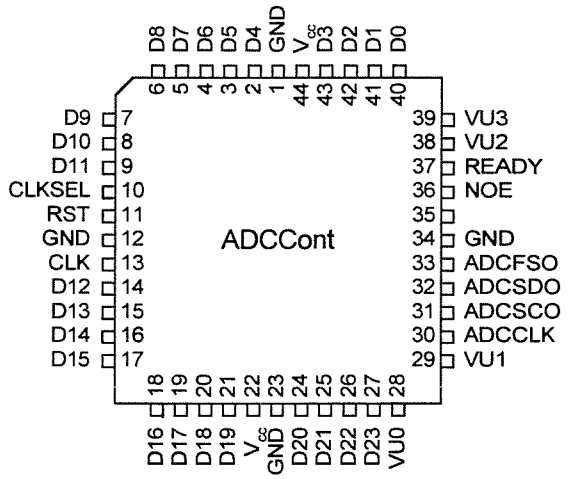
Pin name	Pin number	Description
CLK	22	System clock
RST	21,23-24	Power on reset
SYSRST	20	System reset output
FA9-FA0	27-36	FFT RAM address bus
FNOE	18	FFT RAM output enable signal
FRNW	26	FFT RAM read/write signal
FRDY	14	FFT processor ready output
FLD	15	FFT processor load data signal
JNOE	16	FFT processor j output enable signal
KNOE	17	FFT processor k output enable signal
WA9-WA0	13-5,3	W ROM address bus
WNOE	2	W ROM output enable signal
MOD7-MOD0	45-41,39-37	Modulus ROM data bus
VCOM3-VCOM0	84-83,81-80	Video processor command drive
VX8-VX0	79-71	Video processor x co-ordinate drive
VY7-VY0	70-68,62,60-57	Video processor y co-ordinate drive
VEXEC	56	Video processor execute request
VACK	55	Video processor acknowledge
ADCRDY	47	ADC controller ready
ADCNOE	48	ADC controller output enable drive
CONT3-CONT0	50-53	Control panel buttons

Figure C.4 System controller pin diagram



Pin name	Pin number	Description
CLK	22	System clock
RST	23,24	System reset
FD23-FD0	49-47,45-41,39-26,20,18	FFT RAM data bus
JNOE	51	j output enable
KNOE	50	k output enable
WD23-WD0	84-83,81-68,62,60-54	W ROM data bus
LOAD	52	Load data request
READY	53	Processor ready/calculating signal

Figure C.5 FFT processor pin diagram



Pin name	Pin number	Description
CLK	13	System clock
RST	11	System reset
READY	37	Sample ready signal
D23-D0	27-24,21-14,9-2,43-40	Sample data bus drive
NOE	36	Sample output enable
ADCFSO	33	ADC frame sync out
ADCSDO	32	ADC serial data out
ADCSCO	31	ADC serial clock out
ADCCLK	30	ADC clock drive signal
CLKSEL	10	ADC clock frequency select
VU3-VU0	39-38,29-28	Level meter value drive

Figure C.6 ADC controller pin diagram

Figure C.7 EL320.256-F6 electroluminescent display data sheet extracts

EL320.256–F6 and –FD6

320 x 256 Pixel
Low Power Electroluminescent Display

Product Profile

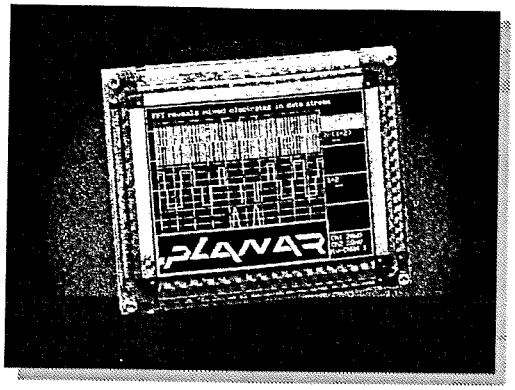
Operations Manual

The EL320.256-F Series displays are low power, rugged, electroluminescent (TFEL) displays which replace the bulky CRT in control and instrument product designs. They feature integral DC/DC converter, and their compact dimensions save space that can allow addition of features or reduction in overall size. They are designed to function in extreme environments, and their crisp display is viewable under most lighting conditions at wide viewing angles. Their ease of installation reduces system integration costs.

The EL320.256-F6 and –FD6 are 320 column by 256 row flat panel displays with a resolution of 80 dots per inch. The pixel aspect ratio is 1:1. The digital flat panel interface is designed to match the needs of most systems. The display may be driven at frame rates up to 75 Hz.

The EL320.256-F Series displays require +5 V and +11 to +30 V (Vcc1, Vcc2) power and four basic signals to operate:

1. Video Data or pixel information (VID)
2. Video Clock, pixel clock, or dot clock (VCLK)
3. Horizontal Sync (HS)
4. Vertical Sync (VS).

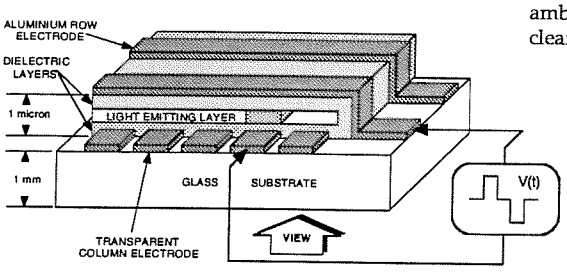


The EL glass panel is a solid-state device with a thin film luminescent layer sandwiched between transparent dielectric layers and a matrix of row and column electrodes. The row electrodes, in back, are aluminium; the column electrodes, in front, are transparent. The entire thin film device is deposited on a single glass substrate. The glass panel is mounted to an electronic circuit assembly board (ECA) with an elastic spacer. The ECA's are connected to the EL glass panel with soldered lead frames. The result is a flat, compact, reliable and rugged display device.

The EL320.256–FD6 display includes a dark ICE (Integrated Contrast Enhancement) background in the display glass. ICE background significantly improves the luminance contrast of the display in bright ambients. ICE also removes the halo around the lit pixels in dark ambient making the appearance of each pixel crisp and clear.

EL Technology

The display consists of an electroluminescent glass panel and two mounted circuit boards with control electronics.



In the EL320.256-F Series, the 320 column electrodes and 256 row electrodes are arranged in an X-Y formation with the intersecting areas performing as pixels. Voltage is applied to both the correct row electrode and the correct column electrode to cause a lit pixel. Operating voltages required are provided by an integral DC/DC converter.

PLANAR®
The Definition of Quality ®

Electrical Characteristics

Connector Layout

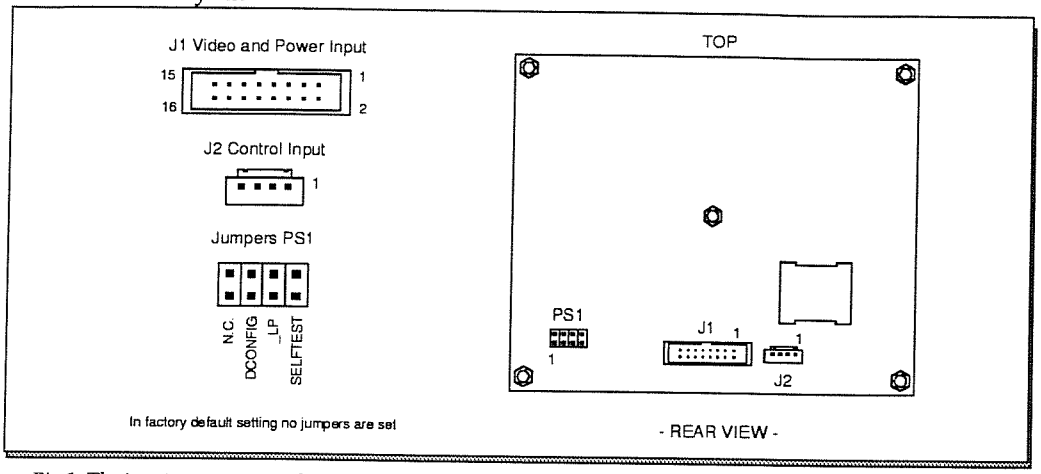


Fig 1. The input connectors and programmable jumpers on EL320.256–F6 and –FD6 displays.

Input Connectors and Programmable Jumpers

Pins	Signal	Symbol	Description
J1 (Data / power input connector)			
1, 2	Voltage	Vcc2	Supply voltage (+11...30 V) converted to required internal high voltages.
3, 4	Voltage	Vcc1	Supply voltage (+5 V) for the logic.
5	Enable	_ENABLE	Display operation is enabled when LOW or left disconnected.
6, 8, 10			
12, 14, 16	Ground	GND	Signal and power return.
7	Two bit data	TVID	Odd column data input for optional two bit parallel mode. See page 3.
9	Vertical Sync	VS	The vertical sync signal VS controls the vertical position of the picture. The topmost row displayed is the first HS HIGH time ending after the rising edge of the VS.
11	Horizontal Sync	HS	The horizontal sync signal HS controls the horizontal position of the picture. The last 320 pixels before the fall of HS are displayed.
13	Video Clock	VCLK	The VCLK signal shifts data present on the VID and TVID lines into the display system. VCLK is active on the rising edge.
15	Video Data	VID	Signal that supplies the pixel information to the system. Even pixel data for two pit parallel mode. See page 3.
J2 (Control input)			
1	Luminance	LCb	Brightness control inputs a and b. If left disconnected, luminance is at its maximum level. See brightness control on page 3
2	Luminance	LCb	
3	Ground	GND	Signal return. Same as GND in connector J1.
4	Low Power	_LOWPOW	If pulled LOW display is in Low Power Mode. Display has its normal brightness if HIGH or left disconnected. See page 3
Pinstrip PS1			
1		NC	No Connection
2	Two bit parallel	DCONFIG	The video data is input two pixels per video clock using VID and TVID if jumper is set.
3	Low Power	_LP	Low Power Mode is selected when jumper is set. This function overrules _LOWPOW control input.
4	SELFTEST		When set, video data input in VID and TVID is displayed asynchronously.

■ Connectors

J1	16-pin header	ODU 511.066.003.016 or eq.
	mating	ODU 517.065.003.016 or eq.
J2	4-pin header	Hirose DF1–4P–2.5 DSA or eq.
	mating	Hirose DF1–4S–2.5 R 24 or eq.
	protector	Hirose DF1–4A 1.33

■ Control basics

The EL panel has 320 transparent column electrodes crossing 256 row electrodes in an X-Y fashion. Light is emitted when an AC voltage is applied at a row-column intersection. The display operation is based on the symmetric, line at a time data addressing scheme which is synchronized by the external VS, HS, and video clock input signals. The signal inputs are HCT compatible with 100 Ω series resistors.

■ Power Input

The input voltages needed are the +5 V input (Vcc1) for the logic and the +11...+30 V input (Vcc2) for the DC/DC converter generating all internal high voltages.

Display Features

■ Low Power Mode

The power consumption of the display is possible to be reduced typically to 3 W by using the low power Mode. This mode is selected either with _LP jumper (PS1/3) or temporarily with _LOWPOW control input (J2/4). The _LP jumper overrules the control input.

Low Power Mode is selected when _LP jumper is set or _LOWPOW input is pulled LOW. When _LP is open and _LOWPOW is HIGH of left disconnected, the display has its normal brightness. This function slightly reduces the contrast and average brightness of the display.

■ Two-Bits-Parallel

For reduction of data clock frequency, it is possible to input the data of two pixels per pixel clock. This feature is selected with DCONFIG jumper (PS1/2). If the jumper is set, data for even columns is input in VID and data for odd columns is input in TVID. If jumper is open, data is input normally to VID only.

■ Brightness Control

The brightness of the display can be adjusted from below 10 % up to full brightness by a 50 kΩ external logarithmic potentiometer between LCa and LCb control inputs (J2/1 and /2). The control function is achieved by sinking a small current from LCa to LCb (when open, the voltages are at 5 V and 0 V respectively).

If the two inputs are left disconnected, the brightness is at its maximum level.

■ Self Test

The operation of the display can be easily tested using the two self test features:

When SELFEST jumper (PS1/4) is set, the video data at VID and TVID are displayed asynchronously without the use of any timing signals.

When only supply voltages without any video data are input, the display starts scanning with all pixels on except the leftmost half of the topmost row.

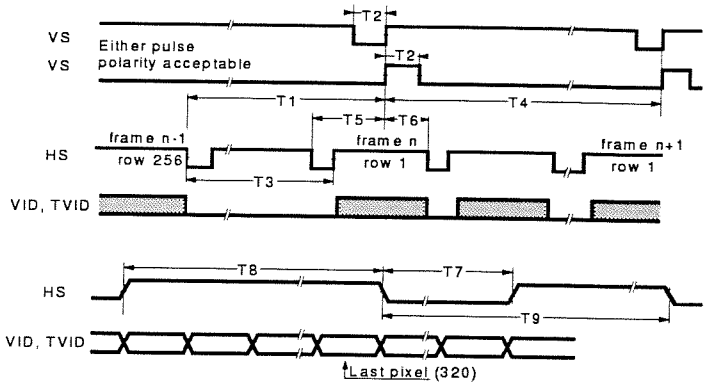
■ Input specifications

Parameter	Symbol	Min.	Typ.	Max.	Absolute min./max.
Logic input HIGH		2 V	—	Vcc1	Vcc1 + 0.5 V Abs. max.
Logic input LOW			—	0.8 V	−0.5 V Abs. min.
Supply voltage	Vcc1	4.75 V	5.0 V	5.25 V	6.0 V Abs. max.
Supply current at 5 V	Icc1	—	0.1 A	0.2 A	
Supply voltage	Vcc2	10.8 V	—	30 V	33 V Abs. max.
Supply current at 12 V	Icc2	—	0.3 A	0.6 A	
Supply current at 12 V (Low Power)	Icc2	—	0.2 A	0.4 A	
Power consumption			4 W	8.2 W	
Power consumption (Low Power)			3 W	5 W	

Operating conditions: Ambient temperature 25°C.

Note: Absolute maximum ratings are those values beyond which damage to the device may occur. The minimum and maximum specifications in this Operations Manual should be met, without exception, to ensure the long-term reliability of the display. Planar does not recommend operation of the display outside these specifications.

■ Timing Characteristics

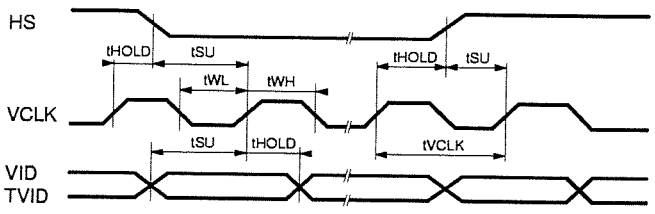


	min	max	unit	Note
T1 Vertical Front Porch	100		µs	1
T2 VS HIGH/LOW time	30		ns	2
T3 Vertical Blank	70		µs	
T4 Vertical Period	256		tHS	
VS frequency		75	Hz	

	min	typ	unit	Note
T5 HS setup to VS	1		µs	3
T6 HS hold from VS	3		µs	
T7 HS Low Time	4		tVCLK	
T8 HS High Time	320	320	tVCLK	4
T9 HS period (tHS)	51		µs	

Notes:
1. This time is needed to display the last row and to initiate the following frame.
2. Only rising edge is used.
3. 2 tVCLK minimum
4. The number of VCLK pulses during HS high time must be even.
   Video clock VCLK must be kept running continuously.

■ Setup and hold timing



symbol		min.	max.
tSU	HS, VID, TVID setup to VCLK	5 ns	
tHOLD	HS, VID, TVID hold from VCLK	8 ns	
tWL	VCLK low width	16 ns	
tWH	VCLK high width	16 ns	
tVCLK	VCLK period	40 ns	
	VCLK frequency		25 MHz

National Semiconductor

February 1995

ADC16071/ADC16471
16-Bit Delta-Sigma 192 ks/s Analog-to-Digital Converters

General Description

The ADC16071/ADC16471 are 16-bit delta-sigma analog-to-digital converters using 64 × oversampling at 12.288 MHz. A 5th-order comb filter and a 246 tap FIR decimation filter are used to achieve an output data rate of up to 192 kHz. The combination of oversampling and internal digital filtering greatly reduces the external anti-alias filter requirements to a simple RC low pass filter. The FIR filters offer linear phase response, 0.005 dB passband ripple, and ≥90 dB stopband rejection. The ADC16071/ADC16471's analog fourth-order modulator uses switched capacitor technology. A built-in fully-differential bandgap voltage reference is also included in the ADC16471. The ADC16071 has no internal reference and requires externally applied reference voltages.

The ADC16071/ADC16471 use an advanced BiCMOS process for a low power consumption of 500 mW (max) while operating from a single 5V supply. A power-down mode reduces the power supply current from 100 mA (max) in the active mode to 1.3 mA (max).

The ADC16071/ADC16471 are ideal analog-to-digital front ends for signal processing applications. They provide a complete high resolution signal acquisition system that requires a minimal external anti-aliasing filter, reference, or interface logic.

The ADC16071/ADC16471's serial interface is compatible with the DSP56001, TMS320, and ADSP2100 digital signal processors.

Key Specifications

■ Resolution	16 bits
■ Total harmonic distortion	
48 kHz output data rate	−94 dB (typ)
192 kHz output data rate	−80 dB (typ)
■ Maximum output data rate	192 kHz (min)
■ Power dissipation	
— Active	
192 kHz output data rate	500 mW (max)
48 kHz output data rate	275 mW (max)
— Power-down	6.5 mW (max)

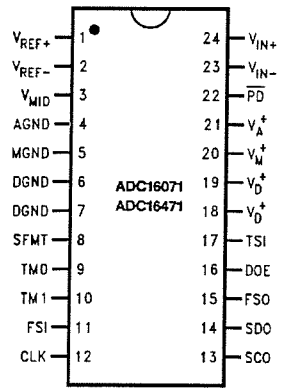
Key Features

- Voltage reference (ADC16471 only)
- Fourth-order modulator
- 64 × oversampling with a 12.288 MHz sample rate
- Adjustable output data rate from 7 kHz to 192 kHz
- Linear-phase digital anti-aliasing filter:
  - 0.005 dB passband ripple
  - 90 dB stopband rejection
- Single +5V supply
- Power-down mode
- Serial data interface compatible with popular DSP devices

Applications

- Medical instrumentation
- Process control systems
- Test equipment
- High sample-rate audio
- Digital Signal Processing (DSP) analog front-end
- Vibration and noise analysis

Connection Diagram



TL/H/11454–2

Ordering Information

Part No.	Package	NS Package No.
ADC16471CIN	24-Pin Molded DIP	N24C
ADC16471CIWM	24-Pin SOIC	M24B
ADC16071CIN	24-Pin Molded DIP	N24C
ADC16071CIWM	24-Pin SOIC	M24B

TRI-STATE® is a registered trademark of National Semiconductor Corporation.

RRD-B30M75/Printed in U. S. A.

ADC16071/ADC16471 16-Bit Delta-Sigma 192 ks/s Analog-to-Digital Converters

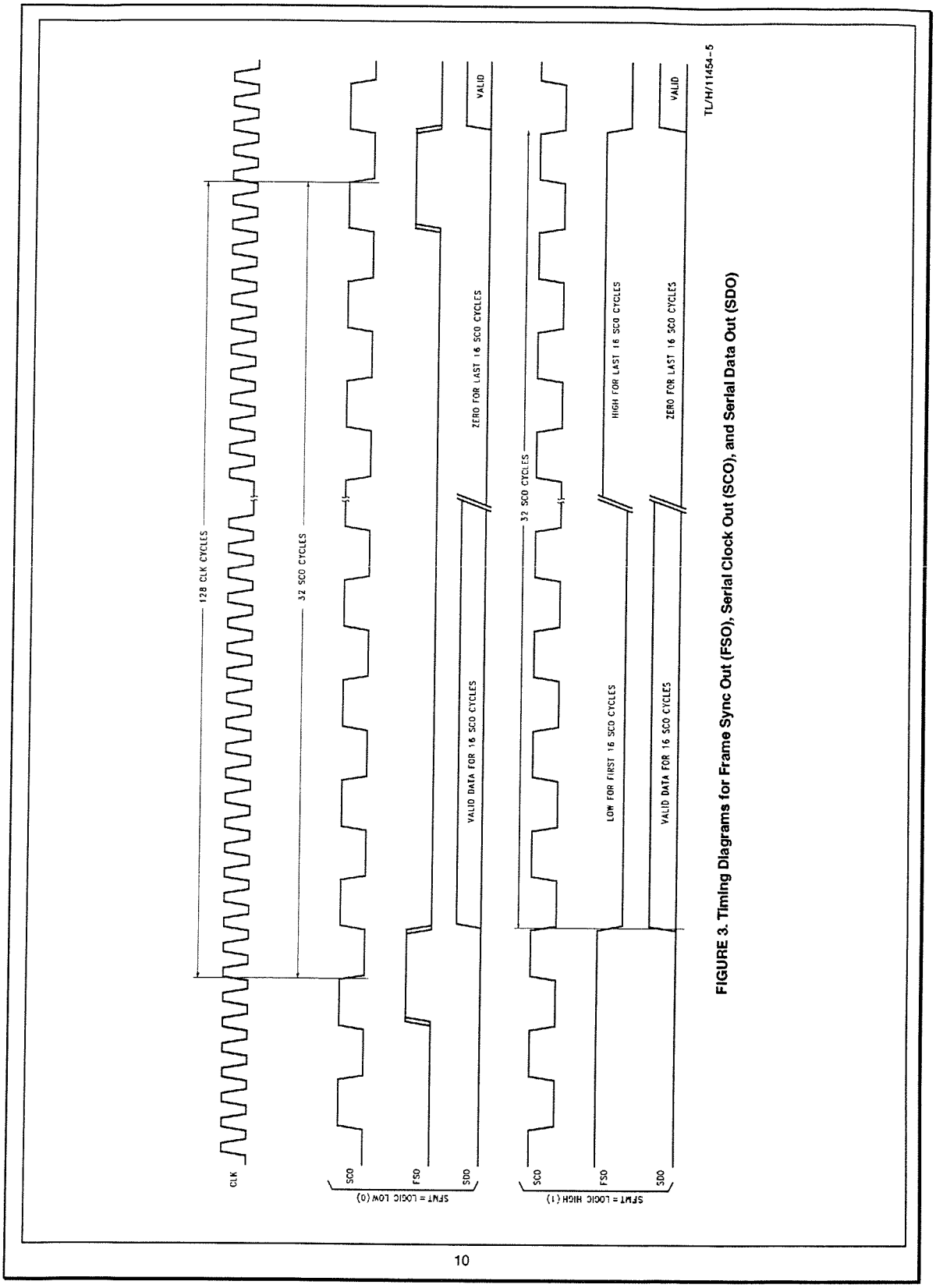


FIGURE 3. Timing Diagrams for Frame Sync Out (FSO), Serial Clock Out (SCO), and Serial Data Out (SDO)

C.3 Source Code Listings

Listing C.1 Fast fourier transform using double precision aritmetic in C++

```
1  void fft(double data[], int n)
2  /* Perform FFT with Cooley-Tukey algorithm using double precision floating
3     point. n is log2 the no. of complex pairs in data, ie. data[0..2*(2^n)-1],
4     where data[i] is real, data[i+1] is complex. On exit, data holds the
5     resulting DFT data. */
6  {
7    int nn=1<<n;                                  // number of points
8
9    // Perform bit reversal on all elements - assumes nn is form 2^n
10   for (int i=0; i<nn; i++) {                    // process whole array
11     int ir = bit_flip(i,n);                     // reverse i bits
12     if (ir > i) {                               // if it hasn't swapped, then swap
13       swap(data[2*i],data[2*ir]);               // note - array is re,imag
14       swap(data[2*i+1],data[2*ir+1]);
15     }
16   }
17
18   // Set up W array of W0 to W(nn/2) - but complex so twice the size
19   double *W_array=new double[2*nn];
20   for (int i=0; i<nn; i++) {
21     W_array[2*i] = cos(6.28318530717959*i/nn);
22     W_array[2*i+1] = sin(6.28318530717959*i/nn);
23   }
24
25   // Now do the FFT calculation using Cooley-Tukey algorithm
26   int delta = 1;                                // combining pair difference
27   int W_delta = nn >> 1;                        // increase in W index per block
28   for (int l=0; l<n; l++)                       // loop the FFT stages
29   {
30     int W_idx = 0;
31     for (int k_base=0; k_base<delta; k_base++)
32     {
33       double Wr = W_array[2*W_idx];
34       double Wi = W_array[2*W_idx+1];
35       for (int k=k_base; k<nn; k+=2*delta)      // loop combining pairs
36       {
37         int j = k + delta;                      // combine k, k+delta
38
39         // complex multiply W*data[j]
40         double tempr = Wr*data[2*j] - Wi*data[2*j+1]; // Re(W*j)
41         double tempi = Wr*data[2*j+1] + Wi*data[2*j]; // Im(W*j)
42
43         // calculate new data[k] and data[j] values
44         data[2*j] = data[2*k]-tempr;            // j = k - W*j
45         data[2*j+1] = data[2*k+1]-tempi;
46         data[2*k] += tempr;                     // k = k + W*j
47         data[2*k+1] += tempi;
48       }
49       W_idx += W_delta;
50     }
51     delta <<= 1;                                // delta = 1,2,4,8...
52     W_delta >>= 1;                              // W_delta = ..16,8,4...
53   }
54   delete[] W_array;                             // delete the W array
55 }
56
57 int bit_flip(int i, int n)
58 /* Reverse the order of n bits in i. */
59 {
60   int ir=0;                                  // i reversed result
61   int j=1;                                   // bottom bit (for i)
62   int m=1<<(n-1);                            // top bit (for ir)
63   while (m && j<=i) {                        // reverse i bits into ir
64     if (i & j) ir+=m;
65     m>>=1;                                   // move ir bit down
66     j<<=1;                                   // move i bit up
67   }
68   return ir;
69 }
70
71 void swap(double &a, double &b)
72 /* Swap the contents of a and b. */
```

```
73 {
74   double temp=a; a=b; b=temp;
75 }
```

Listing C.2 Fast fourier transform using fixed point aritmetic in C++

```
#define NORM 10                                 // bits to rhs of binary point

void fft_int(int data[], int n)
/* Perform FFT with Cooley-Tukey algorithm using fixed point integer maths.
   n is log2 the no. of complex pair in data, ie data[0..2*(2^n)-1], where
   data[i] is real, data[i+1] is complex. On exit, data holds the resulting
   DFT data.

   Fixed point format is determined by NORM which is the number of bits to
   the right of the binary point. data is assumed to be already scaled. */
{
  int nn=1<<n;                                  // number of points

  // Perform bit reversal on all elements - assumes nn is form 2^n
  for (int i=0; i<nn; i++) {                    // process whole array
    int ir = bit_flip(i,n);                    // reverse i
    if (ir > i) {                               // if it hasn't swapped, then swap
      swap(data[2*i],data[2*ir]);               // note - array is re,imag
      swap(data[2*i+1],data[2*ir+1]);
    }
  }

  // Set up W array of W(0) to W(nn/2) - but complex so twice the size
  int *W_array = new int[nn];
  for (int i=0; i<nn/2; i++) {
    W_array[2*i] = (int)(((double)(1<<NORM)) * cos(6.28318530717959*i/nn));
    W_array[2*i+1] = (int)(((double)(1<<NORM)) * sin(6.28318530717959*i/nn));
  }

  // Now do the FFT calculation using Cooley-Tukey algorithm
  int delta = 1;                                // combining pair difference
  int W_delta = nn >> 1;                        // increase in W index per block
  for (int l=0; l<n; l++)                       // loop the FFT stages
  {
    int W_idx = 0;
    for (int k_base=0; k_base<delta; k_base++)
    {
      int Wr=W_array[2*W_idx];
      int Wi=W_array[2*W_idx+1];
      for (int k=k_base; k<nn; k+=2*delta)  // loop combining pairs
      {
        int j = k + delta;                      // combine k, k+delta

        // complex multiply W*data[j] - normalised to fixed point width
        int tempr = asr(Wr*data[2*j],NORM) - asr(Wi*data[2*j+1],NORM);
        int tempi = asr(Wr*data[2*j+1],NORM) + asr(Wi*data[2*j],NORM);

        // calculate new data[k] and data[j] values
        data[2*j] = data[2*k] - tempr;          // j = k - W*j
        data[2*j+1] = data[2*k+1] - tempi;
        data[2*k] += tempr;                     // k = k + W*j
        data[2*k+1] += tempi;

        // reduce the range of data points to -1.0->1.0
        data[2*j] = asr(data[2*j],1);
        data[2*j+1] = asr(data[2*j+1],1);
        data[2*k] = asr(data[2*k],1);
        data[2*k+1] = asr(data[2*k+1],1);
      }
      W_idx += W_delta;
    }
    delta <<= 1;                                // delta = 1,2,4,8...
    W_delta >>= 1;                              // W_delta = ..16,8,4...
  }
  delete[] W_array;                             // delete the W array
}

int bit_flip(int i, int n)
/* Reverse the order of n bits in i. */
{
  int ir=0;                                  // i reversed result
  int j=1;                                   // bottom bit (for i)
```

```
73    int m=1<<(n-1);                          // top bit (for ir)
74    while (m && j<=i) {                      // reverse i bits into ir
75      if (i & j) ir+=m;
76      m>>=1;                                 // move ir bit down
77      j<<=1;                                 // move i bit up
78    }
79    return ir;
80  }
81
82  int asr(int a, int shift)
83  /* Arithmetic shift right- shift a right by shift places extending the sign. */
84  {
85    if (a == -1) return 0;                   // smaller than -1 = 0
86    for (int i=0; i<shift; i++) {            // shift and extend
87      a>>=1;
88      if (a & 0x40000000) a |= 0x80000000;   // extend top bit (32 bits assumed)
89    }
90    return a;
91  }
92
93  void swap(int &a, int &b)
94  /* Swap the contents of a and b. */
95  {
96    a=a^b; b=a^b; a=a^b;
97  }
```

Listing C.3 VidProc.vhd - Video processor

```
1  ------------------------------------------------------------------------------
2  -- VidProc.vhd - Video processor for Planar VFD     Alan Williams, October 1996
3  --
4  -- Part of the audio band spectrum analyser PhD. project.
5  ------------------------------------------------------------------------------
6  -- Implements a complete video subsystem for a bit mapped display on a Planar
7  -- VFD device. Generates the necessary video data and sync signals from data
8  -- stored in a frame buffer.
9  --
10 -- Also implements a rendering system featuring: direct to video memory, pixel
11 -- plotting, horizontal and vertical lines, horizontal and vertical characters
12 -- and screen fill/clear. All rendering can be done using 4 modes to control
13 -- how the frame buffer is updated: replace, OR, clear, XOR.
14 --
15 -- More details can be found in PhD thesis chapter 7 and appendices.
16 ------------------------------------------------------------------------------
17 -- Modifications to the ICODE:
18 --    . Convert all three switch blocks to decodes - any missing choices should
19 --      be set to the default
20 --
21 -- Modifications to optimised structural VHDL (see file "bits.vhd"):
22 --    . Convert vid_data and vid_data_out ports into tri-states controlled
23 --      by fp_noe.
24 --    . Combine raster_addr and render_addr into a single vid_addr port with
25 --      the driver controlled by sel_render.
26 ------------------------------------------------------------------------------
27
28 --use work.icode_ops.all;                              -- for simulation only
29 --use work.macro_ops.all;
30
31 entity VideoProcessor is
32   port( x           : in bit_vector(8 downto 0);       -- x and data input
33         y           : in bit_vector(7 downto 0);       -- y input
34         command     : in bit_vector(3 downto 0);       -- command input
35         execute     : in bit;                          -- execute command
36         ack         : out bit;                         -- command executing
37
38         raster_addr : buffer bit_vector(13 downto 0); -- raster video address
39         render_addr : buffer bit_vector(13 downto 0); -- render video address
40         sel_render  : out bit;                         -- render/raster address
41         fp_noe      : buffer bit;                      -- FPGA vid_data IO drive
42
43         vid_data    : in bit_vector(7 downto 0);       -- video memory data bus
44         vid_data_out : buffer bit_vector(7 downto 0); -- out of vid_data bus
45         vid_rnw     : out bit;                         -- video memory rnw
46         vid_noe     : out bit;                         -- video memory noe
47         blanking    : in bit_vector(3 downto 0);       -- horiz. blanking time
48
49         res_addr    : buffer bit_vector(10 downto 0);  -- resource ROM address
50         res_data    : in bit_vector(7 downto 0);       -- resource ROM data
51
52         vclk        : buffer bit;                      -- screen video data
53         vid         : out bit;                         -- screen video clock
54         hsync       : out bit;                         -- screen horizontal sync
55         vsync       : out bit                          -- screen vertical sync
56       );
57 end VideoProcessor;
58
59 architecture behaviour of VideoProcessor is
60   ----------------------------------------------------------------------------
61   -- signals used for inter-process communication/synchronisation
62   ----------------------------------------------------------------------------
63   signal raster_sem : bit := '1';                -- raster ack read value
64   signal memory_val : bit_vector(7 downto 0);    -- memory value for raster
65
66   signal render_sem : bit := '0';                -- render request write
67   signal render_sem_ack : bit := '0';            -- memory ack render write
68   signal render_mode : bit_vector(1 downto 0) := "00";     -- render mode
69   signal render_data : bit_vector(7 downto 0);   -- out data from render
70
71 begin
72
```

```
73  ------------------------------------------------------------------------
74  -- Pipelined multi-process video RAM I/O -
75  --   Updates frame buffer for render process and also retrieves next byte
76  --   for raster process. Multiplexes memory access between the two giving
77  --   priority to raster by reading ahead one byte. Any remaining memory
78  --   bandwidth is free for use by render.
79  ------------------------------------------------------------------------
80
81  memory: process
82    ----------------------------------------------------------------------
83    -- variables used to track next semaphore states (avoids wait on)
84    ----------------------------------------------------------------------
85    variable raster_sem_val : bit := '1';        -- next request raster_sem
86    variable render_sem_val : bit := '1';        -- next request render_sem
87
88  begin
89    ----------------------------------------------------------------------
90    -- initialise memory controls safely for once only (on reset)
91    ----------------------------------------------------------------------
92    fp_noe <= '1';
93    vid_rnw <= '1';
94    vid_noe <= '1';
95    raster_addr <= convert_int2bv(0,14);         -- start raster address
96    wait for 0 ns;
97
98    ----------------------------------------------------------------------
99    -- infinite loop - pipelined VRAM data retrieval of each VRAM byte for
100   -- the raster process. Once data is retrieved, time is given over to
101   -- render operations until raster has acknowledged the read
102   ----------------------------------------------------------------------
103   vram_loop: loop                              -- loop VRAM data for raster
104     --------------------------------------------------------------------
105     -- get the next byte from VRAM to output to screen (via raster process)
106     --------------------------------------------------------------------
107     sel_render <= '0';                         -- video_addr is raster_addr
108     vid_noe <= '0';                            -- enable VRAM output
109     wait for 100 ns;                           -- wait 1 cycle
110     protect;                                   -- insert ICODE PROTECT inst
111
112     memory_val <= vid_data;                    -- load up output value
113     vid_noe <= '1;                             -- VRAM output HiZ
114     wait for 100 ns;                           -- wait 1 cycle
115     protect;                                   -- insert ICODE PROTECT inst
116
117     ------------------------------------------------------------------
118     -- wait until raster process has read the value, in the meantime monitor
119     -- render_sem for render access to memory
120     ------------------------------------------------------------------
121     idle_loop: while raster_sem /= raster_sem_val loop
122       if render_sem = render_sem_val then      -- if render requests I/O....
123         sel_render <= '1';                     -- video addr is render_addr
124         vid_noe <= '0';                        -- read the value
125         wait for 100 ns;                       -- wait 1 cycle
126         protect;                               -- insert ICODE PROTECT inst
127
128         ----------------------------------------------------------------
129         -- mask retrieved value with vid_data_out according to render_mode
130         ----------------------------------------------------------------
131         case render_mode is
132           when "11" =>                         -- XOR mask bits
133             vid_data_out <= vid_data XOR render_data;
134           when "10" =>                         -- clear mask bits
135             vid_data_out <= vid_data AND NOT render_data;
136           when "01" =>                         -- set mask bits
137             vid_data_out <= vid_data OR render_data;
138           when "00" =>                         -- fill 8 pixels
139             vid_data_out <= render_data;
140         end case;
141
142         vid_noe <= '1';                        -- HiZ video data bus
143         wait for 100 ns;                       -- wait a cycle
144         protect;                               -- insert ICODE PROTECT inst
145
146         -- write vid_data_out to video RAM
```

```
147           vid_rnw <= '0';                          -- write to video
148           fp_noe <= '0';                           -- drive video bus from FPGA
149           wait for 100 ns;                         -- pulse rnw - 1 cycle
150           protect;                                 -- insert ICODE PROTECT inst
151
152           vid_rnw <= '1';                          -- write the data
153           wait for 100 ns;                         -- data hold - 1 cycle
154           protect;                                 -- insert ICODE PROTECT inst
155
156           fp_noe <= '1';                           -- FPGA release video bus
157
158           ------------------------------------------------------------------
159           -- acknowledge operation complete and prepare for next semaphore
160           ------------------------------------------------------------------
161           render_sem_val := not render_sem_val;  -- track render_sem_out change
162           render_sem_ack <= not render_sem_ack;  -- tell render operation done
163         end if;
164
165         wait for 100 ns;                           -- update and wait a cycle
166       end loop idle_loop;
167
168       raster_sem_val := not raster_sem_val;      -- flip state to track raster
169
170       ------------------------------------------------------------------------
171       -- increment rast_addr or reset and start again if the end of the frame
172       ------------------------------------------------------------------------
173       if raster_addr = convert_int2bv(10239,14) then
174         raster_addr <= convert_int2bv(0,14);
175       else                                       -- else increment rast_addr
176         raster_addr <= raster_addr + convert_int2bv(1,14);
177       end if;
178       wait for 100 ns;                           -- update and wait a cycle
179     end loop vram_loop;
180   end process;
181
182   ------------------------------------------------------------------------------
183   -- Rasterising process from video RAM to screen -
184   --   Continually retrieves pixel data from memory process and outputs in to
185   --   the display pixel by pixel while generating hsync and vsync and blanking
186   --   pulses etc.
187   ------------------------------------------------------------------------------
188
189   raster: process
190     ----------------------------------------------------------------------------
191     -- Variables - note initialisation of counter vars for zero-delay loops
192     ----------------------------------------------------------------------------
193     variable pixel  : bit_vector(7 downto 0); -- pixel data for serialising
194     variable pix_no : integer range 0 to 7 := 0;    -- pixel shift counter
195     variable read   : bit_vector(8 downto 0) := "000000000";  -- 0 to 300
196     variable line   : bit_vector(8 downto 0) := "000000000";  -- 0 to 260
197
198   begin
199
200     vsync <= '1';                                -- start vsync for first ever
201     wait for 100 ns;
202
203     ----------------------------------------------------------------------------
204     -- loop forever outputting VRAM serially together with hsync with horiz
205     -- data (320 pixels) and vsync on the first line of 256. Also a further 4
206     -- lines of blanking (must be >100us) with no hsync or vsync.
207     -- NOTE: the use of nested zero delay loops: counter initialisation is
208     --   performed at the end of the loops so that there is no delay at the
209     --   top, hence the strange positioning of loop initialisers.
210     ----------------------------------------------------------------------------
211     refresh_loop: loop
212
213       ------------------------------------------------------------------------
214       -- loop 256 + 4 lines vertically with vsync high for 1st horizontal
215       ------------------------------------------------------------------------
216       vert_loop: loop                          -- loop horizontal lines (256)
217
218         ----------------------------------------------------------------------
219         -- loop 320 horizontal lines with hsync high, plus blanking (hsync low)
220         ----------------------------------------------------------------------
```

```
221         horiz_loop: loop                          -- loop blocks in 1 line (40)
222
223           ------------------------------------------------------------------
224           -- loop 8 pixel bits output one per vclk (= moods clock/2)
225           ------------------------------------------------------------------
226           block_loop: loop                        -- loop pixels in a block
227
228             ------------------------------------------------------------------
229             -- video cycle 1: output new pixel data on vclk falling edge
230             ------------------------------------------------------------------
231             if line(8) = '0' then                 -- line >255 => blanking
232               vid <= pixel(7);                    -- output the next pixel
233               hsync <= '1';                       -- always hsync
234               pixel(7 downto 1) := pixel(6 downto 0);   -- shift left by 1 bit
235             end if;
236
237             vclk <= '0';                          -- vclk falling edge
238             wait for 100 ns;                      -- wait a cycle
239
240             ------------------------------------------------------------------
241             -- video cycle 2: vclk rising edge - screen latch in pixel data
242             ------------------------------------------------------------------
243             vclk <= '1';                          -- vclk rising edge
244             wait for 100 ns;                      -- wait a cycle
245
246             exit when pix_no = 7;                 -- loop 8 pixels
247             pix_no := pix_no + 1;
248           end loop block_loop;                    -- zero delay loop
249
250           ------------------------------------------------------------------
251           -- end of video cycle 2, pixel 8: get next pixel block to output
252           ------------------------------------------------------------------
253           pix_no := 0;                            -- reset pixel counter
254           if line(8) = '0' then                   -- line >255 => blanking
255             pixel := memory_val;                  -- get next value from memory
256             raster_sem <= not raster_sem;         -- acknowledge read
257             wait for 0 ns;                        -- no delay allowed
258           end if;
259
260           exit when read = convert_int2bv(39,9); -- loop 40 blocks (320/8)
261           read := read + convert_int2bv(1,9);
262         end loop horiz_loop;                      -- zero delay loop
263
264         -- force the above loop to take just 1 cycle, otherwise the last bit
265         --   merges with the lines below requiring 2 cycles
266         protect;                                  -- insert ICODE PROTECT inst
267
268         ------------------------------------------------------------------
269         -- end of line, hsync & vsync go low with vclk falling edge
270         ------------------------------------------------------------------
271         vclk <= '0';                              -- vclk falling edge
272         hsync <= '0';                             -- end of line
273         vsync <= '0';                             -- end of vsync
274         wait for 100 ns;                          -- wait a cycle
275
276         ------------------------------------------------------------------
277         -- hsync period must be >53us. Video above takes 320*2 system cycles,
278         -- = 40us@16MHz so need ~15us padding. 301*sys_clk ~ 18us@40MHz.
279         -- NOTE: ODD number of iterations is essential so vclk exits high ready
280         --       for falling edge at the start of the next line
281         -- This now uses an input to set the top 4 bits of the count allowing
282         -- a range from 32-480 (step 32)
283         ------------------------------------------------------------------
284         read := convert_int2bv(0,9);
285         delay_loop1: loop                         -- horizontal blanking loop
286           vclk <= not vclk;                       -- one vclk edge per iteration
287           wait for 100 ns;
288
289           exit when read(8 downto 5) = blanking; -- 0 to 300 => vclk=1 on exit
290           read := read + convert_int2bv(1,9);
291         end loop delay_loop1;                     -- zero delay loop
292
293         read := convert_int2bv(0,9);              -- reset horiz block counter
294         exit when line = convert_int2bv(259,9);  -- 255 line + 4 blanking = 259
```

```
295         line := line + convert_int2bv(1,9);
296       end loop vert_loop;                          -- zero delay loop
297
298       ---------------------------------------------------------------------
299       -- end of frame - reset line counter and set vsync high for new line 1
300       ---------------------------------------------------------------------
301       line := convert_int2bv(0,9);                 -- reset line counter
302       vsync <= '1';
303       wait for 0 ns;                               -- no delay, get on with it
304
305     end loop refresh_loop;                         -- zero delay loop
306
307   end process;
308
309   -------------------------------------------------------------------------
310   -- Rendering process -
311   --   responds to x,y,command,execute to render into the frame buffer.
312   --   To control, data values are set on x,y with a command set on command.
313   --   If ack is low, a rising edge on execute will start the process. When
314   --   ack goes high, the processor is busy but the input values may be changed
315   --   safely. Once ack goes low again another command may be executed.
316   -- Commands :
317   --   0000 - move cursor to (x,y) position
318   --   0001 - plot point at (x,y) position and move cursor
319   --   0010 - fill single byte with y
320   --   0011 - fill screen with byte y
321   --   0100 - plot character y at cursor position, move right
322   --   0101 - plot character y at cursor position, move down
323   --   0110 - plot vertical line down from cursor, length y (pixels)
324   --   0111 - plot horizontal line right from cursor, length y (8 pixel blocks)
325   --   1100 - set render mode to fill
326   --   1101 - set render mode to set (OR)
327   --   1110 - set render mode to clear (NOT AND)
328   --   1111 - set render mode to XOR
329   -------------------------------------------------------------------------
330
331   render: process
332     -----------------------------------------------------------------------
333     -- variables store last cursor position for line drawing, plotting etc.
334     -----------------------------------------------------------------------
335     variable rend_com    : bit_vector(3 downto 0);  -- current GFX command
336     variable cursor_offs : bit_vector(2 downto 0);  -- last cursor offset
337     variable cursor_addr : bit_vector(13 downto 0); -- last cursor address
338     variable loop_count  : bit_vector(7 downto 0);  -- vertical loop counter
339     variable render_sem_ack_val : bit := '1';
340
341   begin
342     render_loop: loop
343       ack <= '0';                                  -- signal ready for command
344       res_addr <= "00000000000";                   -- clear resource base address
345       while execute /= '0' loop                    -- go if or when execute low
346         wait for 100 ns;                           -- can use "not execute"
347       end loop;
348
349       rend_com := command;                         -- copy command type
350
351       ack <= '1';                                  -- signal loading data
352       wait for 100 ns;                             -- wait 1 cycle
353
354       if rend_com(3 downto 1) = "000" then         -- move and plot set cursor
355         cursor_offs := x(2 downto 0);
356         cursor_addr := convert_int2bv(0,14);
357         cursor_addr(5 downto 0) := x(8 downto 3);
358         cursor_addr(11 downto 3) := cursor_addr(10 downto 3) + y;
359         cursor_addr(13 downto 5) := cursor_addr(12 downto 5) + y;
360       elsif rend_com(3 downto 2) = "11" then       -- set render mode
361         render_mode <= rend_com(1 downto 0);
362       end if;
363
364       -----------------------------------------------------------------------
365       -- setup loop_count and res_addr according to command type
366       -----------------------------------------------------------------------
367       case rend_com is
368         when "0111" =>                             -- horizontal line
```

```
369           loop_count := y;
370           res_addr(3) <= '1';                    -- fill byte resource addr
371         when "0110" =>                           -- vertical line
372           loop_count := y;                       -- x is delta x or y
373           res_addr(2 downto 0) <= cursor_offs;   -- get mask bits
374         when "0101" | "0100" =>                  -- plot character
375           loop_count := convert_int2bv(7,8);     -- no. of lines per char
376           res_addr(10 downto 3) <= y;            -- char resource base addr
377         when "0011" =>                           -- fill screen
378           cursor_addr := convert_int2bv(0,14);
379           render_data <= y;
380         when "0010" =>                           -- fill byte with data
381           loop_count := convert_int2bv(0,8);
382           render_data <= y;
383         when "0001" =>                           -- plot point
384           loop_count := convert_int2bv(0,8);
385           res_addr(2 downto 0) <= cursor_offs;   -- get mask bits
386         when "0000" =>                           -- move & default(decode)
387           next;
388         when others =>
389           next;
390       end case;
391
392       wait for 100 ns;                           -- update and wait a cycle
393       protect;                                   -- insert ICODE protect inst
394
395       ------------------------------------------------------------------------
396       -- loop starting from cursor_addr doing stuff according to command type
397       ------------------------------------------------------------------------
398       render_addr <= cursor_addr;                -- output address is cursor
399       ack <= '0';                                -- signal data loaded
400       wait for 100 ns;                           -- wait a cycle
401
402       count_loop: loop                           -- output loop_count bytes
403         if rend_com(3 downto 1) /= "001" then    -- not fill byte/fill screen
404           render_data <= res_data;               -- get byte to output
405         end if;
406
407         render_sem <= not render_sem;            -- output via memory process
408         while render_sem_ack /= render_sem_ack_val loop
409           wait for 100 ns;
410         end loop;
411         render_sem_ack_val := not render_sem_ack_val;
412
413         if rend_com /= "0011" then               -- if not fill screen command
414           exit when loop_count = convert_int2bv(0,8);      -- loop loop_count
415           loop_count := loop_count - convert_int2bv(1,8);  --   down to 0
416         end if;
417
418         exit when render_addr = convert_int2bv(10239,14);  -- end of vid. ram ?
419
420         ------------------------------------------------------------------------
421         -- update output address for next output: +1 horizontal or +40 vertical
422         ------------------------------------------------------------------------
423         if  rend_com(1 downto 0) = "11" then     -- horiz line/fill screen
424           render_addr <= render_addr + convert_int2bv(1,14);
425         else
426           render_addr(13 downto 3) <= render_addr(13 downto 3) + "00000000101";
427         end if;
428
429         if rend_com(3 downto 1) = "010" then     -- char: get next data cell
430           res_addr <= res_addr + convert_int2bv(1,10);
431         end if;
432
433         wait for 100 ns;                         -- update signals, read ROM
434       end loop count_loop;
435
436       ------------------------------------------------------------------------
437       -- move cursor after plot according to command type
438       ------------------------------------------------------------------------
439       case rend_com is
440         when "0111" =>                           -- plot horizontal line
441           cursor_addr := render_addr;            -- last cursor=current cursor
442           cursor_offs := "111";                  -- offset is end of block(7)
```

```
443         when "0101" =>                              -- plot character vert
444           cursor_addr := render_addr + convert_int2bv(40,14);
445         when "0100" =>                              -- plot character horiz
446           cursor_addr := cursor_addr + convert_int2bv(1,14);
447         when others =>                              -- rest: cursor is end posn
448           cursor_addr := render_addr;
449       end case;
450
451     end loop render_loop;
452   end process render;
453
454 end behaviour;
455
456 ------------------------------------------------------------------------------
```

Listing C.4 FFTCont.vhd - Main system and FFT execution controller

```
1  ------------------------------------------------------------------------------
2  -- FFTCont.vhd - Main system and FFT control        Alan Williams, October 1996
3  --
4  -- Part of the audio band spectrum analyser PhD. project.
5  ------------------------------------------------------------------------------
6  -- Main system control device: loads sample data into memory, calculates the
7  -- a 512 point FFT using the FFT processor (FFTProc.vhd) and outputs the
8  -- results to the video screen using the video processor chip (VidProc.vhd).
9  -- The display is also controlled by four push button switches connected to
10 -- the controls input which allow the display to be frozen, zoomed and panned.
11 --
12 -- More details can be found in PhD thesis chapter 7 and appendices.
13 ------------------------------------------------------------------------------
14 -- Modifications to optimised structural VHDL (see file "bits.vhd"):
15 --    . Convert vid_command, vid_x and vid_y ports into tri-states controlled
16 --      by fp_vid_noe.
17 ------------------------------------------------------------------------------
18
19 --use work.icode_ops.all;                                  -- for simulation only
20 --use work.macro_ops.all;
21
22 entity FFTControl is
23   port( sys_reset   : out bit;                             -- reset the whole system
24         controls    : in bit_vector(3 downto 0);
25
26         ram_addr    : buffer bit_vector(9 downto 0);  -- fft RAM address
27         ram_rnw     : out bit;                           -- fft RAM rnw
28         ram_noe     : out bit;                           -- fft RAM noe
29         rom_addr    : buffer bit_vector(9 downto 0);  -- fft w ROM address
30         rom_noe     : out bit;                           -- fft w ROM op enable
31
32         adc_ready   : in bit;                            -- adc sample control
33         adc_noe     : out bit;                           -- adc output enable
34
35         fft_ready   : in bit;                            -- fft processor idle
36         fft_load    : out bit;                           -- fft load j/k control
37         fft_j_noe   : out bit;                           -- fft output bus = j_val
38         fft_k_noe   : out bit;                           -- fft output bus = k_val
39
40         mod_data    : in bit_vector(7 downto 0);         -- modulus value
41
42         vid_command : buffer bit_vector(3 downto 0);  -- video command number
43         vid_x       : buffer bit_vector(8 downto 0);  -- video x position
44         vid_y       : buffer bit_vector(7 downto 0);  -- video y and data
45         vid_execute : out bit;                           -- video execute signal
46         vid_ack     : in bit;                            -- video acknowledge
47         fp_vid_noe  : out bit                            -- FPGA video bus enable
48       );
49 end FFTControl;
50
51 architecture behaviour of FFTControl is
52   ----------------------------------------------------------------------------
53   -- signals used in both processes - set by controls and used but output
54   ----------------------------------------------------------------------------
55   signal disp_base : bit_vector(5 downto 0) := convert_int2bv(0,6);
56   signal zoom : bit_vector(3 downto 0) := "1111";
57   signal hold_on : bit := '0';
58
59   ----------------------------------------------------------------------------
60   -- rom_addr is for both FFT and modulus - aliases for readability
61   ----------------------------------------------------------------------------
62   alias fft_w_addr : bit_vector(8 downto 0) is rom_addr(8 downto 0);
63 begin
64
65   ----------------------------------------------------------------------------
66   -- Control monitor process  -
67   --   Continually monitors the controls input for button presses and then
68   --   sets the current display state signals accordingly. disp_base is the
69   --   sample to display on the far left /8, zoom is the width of a sample bar
70   --   and hold_on is set for display pausing. Note: it is very important that
71   --   the controls be de-bounced to avoid multiple key presses.
72   ----------------------------------------------------------------------------
```

```
 73
 74   controls: process
 75     ------------------------------------------------------------------------
 76     -- prev_cont is previous button states so simulataneous edges are detected
 77     ------------------------------------------------------------------------
 78     variable prev_cont : bit_vector(3 downto 0) := "000";
 79   begin
 80     while controls = prev_cont loop                    -- wait for a key press
 81       wait for 100 ns;
 82     end loop;
 83     prev_cont := controls AND (NOT prev_cont);         -- detect changes
 84
 85     ------------------------------------------------------------------------
 86     -- right button pressed: zoom out or pan right depending on zoom key state
 87     ------------------------------------------------------------------------
 88     if prev_cont(0) = '1' then
 89       if controls(3) = '0' then
 90         if disp_base /= convert_int2bv(63,6) then      -- pan right
 91           disp_base <= disp_base + convert_int2bv(1,6);
 92         end if;
 93       else
 94         if zoom /= "1111" then                         -- zoom out
 95           zoom <= zoom + convert_int2bv(1,4);
 96         end if;
 97       end if;
 98     end if;
 99
100     ------------------------------------------------------------------------
101     -- left button pressed: zoom in or pan left depending on zoom key state
102     ------------------------------------------------------------------------
103     if prev_cont(1) = '1' then
104       if controls(3) = '0' then                        -- pan left
105         if disp_base /= convert_int2bv(0,6) then
106           disp_base <= disp_base - convert_int2bv(1,6);
107         end if;
108       else
109         if zoom /= "0001" then                         -- zoom in
110           zoom <= zoom - convert_int2bv(1,4);
111         end if;
112       end if;
113     end if;
114
115     ------------------------------------------------------------------------
116     -- hold button pressed so toggle current hold_on state
117     ------------------------------------------------------------------------
118     if prev_cont(2) = '1' then
119       hold_on <= NOT hold_on;                          -- toggle hold_on
120     end if;
121
122     prev_cont := controls;                             -- previous is current
123     wait for 100 ns;
124   end process;
125
126   ------------------------------------------------------------------------
127   -- Main FFT sample, calculate and display process -
128   --   This splits into five sequential blocks. It would be nicer to have them
129   --   all parallel but we're limited by the chip size and more importantly the
130   --   number of memory busses.
131   --   1. Initialise all the various ports and signals and clear the screen.
132   --   2. Draw static bits of the screen - axes, titles etc. These are
133   --   stored in the top 512 words of the 2 lookup rom with the last 32 entries
134   --   only plotted when hold is on.
135   --   3. Get 1024 samples into the FFT ram. This enables the adc control chip
136   --   to access the ram data bus and controls the ram r/w lines as required.
137   --   Samples are stored in bit reversed order in preparation for the FFT.
138   --   4. Calculate 512 point FFT using samples in the FFT ram. All the main
139   --   FFT looping bits are in here which set up the samples to process and the
140   --   w rom lookup address, with the FFT processor carrying out the update
141   --   calculations using a 12 bit fixed point multiplier.
142   --   5. Plot the results of the FFT on the screen. This cycles through the
143   --   FFT entries to display retrieving the current value via the modulus ROM
144   --   (whose address lines are connected to the FFT ram data lines) and
145   --   plotting the result as a number of vertical lines.
146   --
```

```
147   --   The process continually loops stages 2 to 5 (1 is power on reset only),
148   --   but misses out 3 and 4 when in hold mode.
149   --
150   --   Note: to send a command to the video processor the exec_video_command
151   --   procedure should be called to do the handshaking stuff.
152   ------------------------------------------------------------------------
153
154   FFTControl: process
155
156     ------------------------------------------------------------------------
157     -- variables used generally - shared
158     ------------------------------------------------------------------------
159     variable addr : bit_vector(9 downto 0);
160
161     ------------------------------------------------------------------------
162     -- variables used for FFT control
163     ------------------------------------------------------------------------
164     variable delta : bit_vector(9 downto 0);     -- 1 to nn/2
165     variable w_delta : bit_vector(9 downto 0);   -- 0 to nn/2
166     variable j : bit_vector(9 downto 0);
167     variable k : bit_vector(10 downto 0);
168     alias k_base : bit_vector(9 downto 0) is addr(9 downto 0);    -- 0 to nn/2
169
170     ------------------------------------------------------------------------
171     -- variables used for rendering - manual sharing of registers using aliases
172     ------------------------------------------------------------------------
173     variable zoom_count : bit_vector(3 downto 0);
174     alias zoom_start : bit_vector(3 downto 0) is delta(3 downto 0);
175     alias modulus : bit_vector(7 downto 0) is j(7 downto 0);
176
177     ------------------------------------------------------------------------
178     -- procedure to control access to video processor
179     ------------------------------------------------------------------------
180     procedure exec_video_command is
181     begin
182       vid_execute <= '0';                              -- signal command ready
183       while vid_ack = '0' loop                         -- wait for acknowledge
184         wait for 100 ns;
185       end loop;
186
187       vid_execute <= '1';
188       while vid_ack /= '0' loop                        -- wait till data read
189         wait for 100 ns;
190       end loop;
191
192       -- force an extra state at the end of the procedure
193       protect;
194       protect;
195     end;
196
197   begin
198
199     ------------------------------------------------------------------------
200     -- initialise/reset processor/memory/bus controls etc.
201     ------------------------------------------------------------------------
202     sys_reset <= '1';                                  -- reset the system
203
204     ram_rnw <= '1';
205     ram_noe <= '1';
206     fft_j_noe <= '1';
207     fft_k_noe <= '1';
208     fft_load <= '0';
209
210     adc_noe <= '1';
211     vid_execute <= '1';                                -- don't execute video
212     rom_noe <= '1';                                    -- w ROM bus HiZ
213     fp_vid_noe <= '1';                                 -- vid command bus free
214     wait for 100 ns;                                   -- wait a cycle
215     protect;
216
217     sys_reset <= '0';                                  -- let the system go ...
218     while vid_ack /= '0' loop                          -- wait till video ready
219       wait for 100 ns;
220     end loop;
```

```
221
222     fp_vid_noe <= '0';                                    -- FPGA drives video bus
223     vid_command <= "0011";                                -- fill screen
224     vid_y <= "00000000";                                  -- default vid mode=fill
225     exec_video_command;
226
227     ------------------------------------------------------------------------
228     -- main infinite loop - allows one time initialisation and easy "exit/next"
229     ------------------------------------------------------------------------
230     inf_loop: loop                                        -- loop forever from now
231
232     ram_noe <= '1';
233     fp_vid_noe <= '1';                                    -- FPGA release vid bus
234     wait for 100 ns;                                      -- wait a cycle
235     protect;
236
237     ------------------------------------------------------------------------
238     -- initialise screen using data from the top 512 words of the w lookup ROM
239     ------------------------------------------------------------------------
240     rom_addr <= convert_int2bv(512,10);                   -- start from addr 1024
241     rom_noe <= '0';                                       -- w ROM drives the bus
242     wait for 100 ns;
243
244     setup_loop: loop
245       exec_video_command;
246       if hold_on = '0' then
247         exit when rom_addr = convert_int2bv(991,10);  -- no hold: miss last 32
248       else
249         exit when rom_addr = convert_int2bv(1023,10); -- hold: do all
250       end if;
251       rom_addr <= rom_addr + convert_int2bv(1,10);
252       wait for 100 ns;                                    -- data setup cycle
253     end loop setup_loop;
254
255     ------------------------------------------------------------------------
256     -- load samples and calculate FFT ONLY if hold_on is 0
257     ------------------------------------------------------------------------
258     if hold_on = '0' then
259
260     ------------------------------------------------------------------------
261     -- load samples in bit-reversed order into memory from ADC
262     ------------------------------------------------------------------------
263     adc_noe <= '0';                                       -- adc drives ram bus
264     wait for 0 ns;
265
266     addr := convert_int2bv(0,10);
267     sample_loop: loop
268       ram_addr <= bitflip(addr);                          -- bit reverse address
269       wait on adc_ready until adc_ready = '1';            -- wait for sample ready
270
271       ram_rnw <= '0';
272       wait for 100 ns;                                    -- wait a cycle
273       protect;
274
275       ram_rnw <= '1';                                     -- write data to ram
276       wait for 100 ns;
277
278       exit when addr = convert_int2bv(1023,10);           -- take 1024 samples
279       addr := addr + convert_int2bv(1,10);
280     end loop sample_loop;
281
282     adc_noe <= '1';                                       -- adc release ram bus
283     wait for 0 ns;
284
285     ------------------------------------------------------------------------
286     -- calculate FFT using Cooley-Tukey algorithm with FFT processor
287     -- note: w rom still enabled from setup part
288     ------------------------------------------------------------------------
289     fft_w_addr <= convert_int2bv(0,10);           -- initialise w address
290     delta:=convert_int2bv(1,10);
291     w_delta:=convert_int2bv(512,10);              -- w_delta: nn>>1
292
293     l_loop: loop                                  -- loop: 0 to n-1
294       k_base := convert_int2bv(0,10);
```

```
295       k_base_loop: loop                          -- k_base: 0 to delta-1
296         k := convert_int2bv(0,11);
297         k(9 downto 0) := k_base;
298
299         k_loop: while k(10) = '0' loop           -- k_base to nn-1 step 2*delta
300           j := k(9 downto 0) + delta;
301
302           while fft_ready /= '1' loop            -- wait until FFT is idle
303             wait for 100 ns;
304           end loop;
305
306           ------------------------------------------------------------------
307           -- control memory bus to send j,k memory values to the processor
308           ------------------------------------------------------------------
309           ram_addr <= j;                         -- make ram output j val
310           ram_noe <= '0';
311           fft_load <= '1';                       -- tell fft to read it
312           wait for 200 ns;                       -- wait 2 cycles
313           protect;
314           protect;
315
316           ram_addr <= k(9 downto 0);             -- ram outputs k value
317           fft_load <= '0';                       -- tell fft to read it
318           wait for 200 ns;                       -- wait 2 cycles
319           protect;
320           protect;
321
322           ram_noe <= '1';                        -- disable RAM bus
323           while fft_ready /= '0' loop            -- wait till FFT starts
324             wait for 100 ns;
325           end loop;
326
327           ------------------------------------------------------------------
328           -- twiddle thumbs until calculation complete
329           ------------------------------------------------------------------
330           wait until fft_ready = '1';
331
332           ------------------------------------------------------------------
333           -- control FFTProc bus outputs and write results back to memory
334           ------------------------------------------------------------------
335           fft_k_noe <= '0';                      -- fft can now drive the bus
336           ram_rnw <= '0';
337           wait for 100 ns;                       -- 1 cycle - ram access
338           protect;
339
340           ram_rnw <= '1';
341           wait for 100 ns;                       -- 1 cycle - data stable
342           protect;
343
344           fft_k_noe <= '1';
345           wait for 100 ns;                       -- 1 cycle - Hi Z
346           protect;
347
348           ram_addr <= j;                         -- set fft_address to j now
349           wait for 100 ns;                       -- 1 cycle - address setup
350           protect;
351
352           fft_j_noe <= '0';
353           ram_rnw <= '0';
354           wait for 100 ns;                       -- 1 cycle - ram access
355           protect;
356
357           ram_rnw <= '1';
358           wait for 100 ns;                       -- 1 cycle - data stable
359           protect;
360
361           fft_j_noe <= '1';                      -- all done
362           wait for 100 ns;
363
364           k(10 downto 1) := k(10 downto 1) + delta;
365         end loop k_loop;
366
367         fft_w_addr <= fft_w_addr + w_delta(8 downto 0); -- update w addr, auto
368                                                         --   zeros on overflow
```

```
369         k_base:=k_base + convert_int2bv(1,10);
370         exit when k_base = delta;
371       end loop k_base_loop;
372
373       exit when w_delta(0) = '1';
374
375       delta(9 downto 1):=delta(8 downto 0);       -- delta: delta<<1
376       w_delta(8 downto 0):=w_delta(9 downto 1);   -- w_delta: w_delta>>1
377       delta(0) := '0';
378       w_delta(9) := '0';
379
380     end loop l_loop;
381
382     ------------------------------------------------------------------------
383     -- End of hold_on conditional section
384     ------------------------------------------------------------------------
385     end if;                                       -- if not hold_on
386
387     ------------------------------------------------------------------------
388     -- send data to video processor for a single displayed frame
389     -- display in rectangle (24,207) to (287,64)
390     ------------------------------------------------------------------------
391     ram_addr <= convert_int2bv(0,10);             -- clear all address bits
392     ram_noe <= '0';
393     rom_noe <= '1';                               -- w ROM release bus
394     wait for 100 ns;                              -- wait a cycle
395     protect;
396
397     fp_vid_noe <= '0';                            -- FPGA drives vid bus
398     ram_addr(8 downto 3) <= disp_base;            -- first freq to display
399     zoom_start := zoom;
400     zoom_count := zoom;
401     vid_x <= convert_int2bv(48,9);                -- corresponding x posn
402     wait for 0 ns;
403
404     ------------------------------------------------------------------------
405     -- loop across screen from vid_x to 287, getting and plotting results as
406     --   vertical bars with thickness determined by the zoom signal
407     ------------------------------------------------------------------------
408     x_loop: loop                                  -- loop across screen
409
410       ----------------------------------------------------------------------
411       -- video command: move x,64: move to the top of the current bar
412       ----------------------------------------------------------------------
413       vid_command <= "0000";
414       vid_y <= convert_int2bv(64,8);
415       exec_video_command;
416
417       ----------------------------------------------------------------------
418       -- video command: mode clear: set plot mode to clear bits
419       ----------------------------------------------------------------------
420       vid_command <= "1110";
421       exec_video_command;
422
423       ----------------------------------------------------------------------
424       -- get FFT result from modulus rom port. If address exceeds the number of
425       -- points (512) then force result to 0
426       ----------------------------------------------------------------------
427       if ram_addr(9) = '0' then                   -- >512 => no result
428         modulus := mod_data;
429       else
430         modulus := convert_int2bv(0,8);           -- force to 0
431       end if;
432
433       ----------------------------------------------------------------------
434       -- output the result as a vertical bar separated by one blank line
435       -- output vertical section for zoom_count > 0, otherwise separator line
436       ----------------------------------------------------------------------
437       if zoom_count /= "0000" then
438         ------------------------------------------------------------------
439         -- draw vertical line by first clearing the pixels above it to delete
440         -- any old bits, then draw the bottom bit is white.
441         ------------------------------------------------------------------
442
```

```
443         ------------------------------------------------------------------------
444         -- video command: vert dy-mod_data-1: clear from cursor to top of bar
445         ------------------------------------------------------------------------
446         vid_command <= "0110";
447         vid_y <= convert_int2bv(143,8) - modulus;
448         exec_video_command;
449
450         ------------------------------------------------------------------------
451         -- video command: mode set bits: set plot mode to set bits
452         ------------------------------------------------------------------------
453         vid_command <= "1101";
454         exec_video_command;
455
456         ------------------------------------------------------------------------
457         -- video command: vert mod_data-1: draw bar from cursor to baseline
458         ------------------------------------------------------------------------
459         vid_command <= "0110";
460         vid_y <= modulus;
461         exec_video_command;
462
463         ------------------------------------------------------------------------
464         -- zoom_count is set from zoom to determine bar thickness - zoom_cout
465         -- is incremented for each sample plots a line when non-zero. ie. if
466         -- zoom is 15, bar is 1 pixel wide, if 14, bar is 2 wide etc..
467         ------------------------------------------------------------------------
468         zoom_count := zoom_count + convert_int2bv(1,4);
469       else
470         ------------------------------------------------------------------------
471         -- zoom_count=0 so draw vertical line in black to form a separator
472         -- video command: vert dy-mod_data-1
473         ------------------------------------------------------------------------
474         vid_command <= "0110";
475         vid_y <= convert_int2bv(143,8);
476         exec_video_command;
477         zoom_count := zoom_start;
478         ram_addr <= ram_addr + convert_int2bv(1,10);  -- go to next frequency
479       end if;
480
481       ------------------------------------------------------------------------
482       -- stop incremening display at the end of graph block (x: 64->287)
483       ------------------------------------------------------------------------
484       exit when vid_x = convert_int2bv(287,9);
485
486       vid_x <= vid_x + convert_int2bv(1,9);           -- move right to next bar
487       wait for 100 ns;
488
489     end loop x_loop;
490
491     ------------------------------------------------------------------------
492     -- end of main infinite loop
493     ------------------------------------------------------------------------
494     end loop inf_loop;
495   end process FFTControl;
496
497 end behaviour;
498
499 ------------------------------------------------------------------------------
```

Listing C.5 FFTProc.vhd - Fast fourier transform processing engine

```
1  ------------------------------------------------------------------------------
2  -- FFTProc.vhd - FFT processor engine                Alan Williams, October 1996
3  --
4  -- Part of the audio band spectrum analyser PhD. project.
5  ------------------------------------------------------------------------------
6  -- FFT processor engine which does the hard sums in the FFT inner loop,
7  -- controlled by the main FFT controller.
8  -- When ready is high two complex numbers are loaded on load +ve and -ve edges
9  -- after which the Cooley-Tukey algorithm arithmetic part is computed.
10 -- Throughout the calculation w_in is assumed to hold the stable value for the
11 -- roots of -1 constant stored in the w rom.
12 -- Uses the fpmult macro operator to do fixed point multiplication so don't
13 -- forget to expand it using AOM during (not after) optimisation.
14 --
15 -- More details can be found in PhD thesis chapter 7 and appendices.
16 ------------------------------------------------------------------------------
17 -- Modifications to optimised structural VHDL (see file "bits.vhd"):
18 --    . Combine j_val and k_val into the output half of data_bus with selection
19 --      controlled by j_noe and k_noe.
20 --
21 -- DONT FORGET TO USE THE SIGNED MODULES FOR LOW-LEVEL SYNTHESIS
22 ------------------------------------------------------------------------------
23
24 --use work.icode_ops.all;                                  -- for simulation only
25 --use work.macro_ops.all;
26
27 ------------------------------------------------------------------------------
28 -- Package defining subtypes - complex numbers are stored as 23 bit fixed point
29 -- real/imaginary pairs concatenated into a 23 bit word, bits 11 downto 0 are
30 -- the real component, bits 23 downto 12 are the imaginary. Binary point is
31 -- between bits 9 and 10.
32 ------------------------------------------------------------------------------
33 package VidFFTTypes is
34     subtype ComplexSingle is bit_vector(11 downto 0);
35     subtype ComplexPair is bit_vector(23 downto 0);
36 end VidFFTTypes;
37
38 use work.VidFFTTypes.all;
39
40 entity FFTProcessor is
41   port( data_bus  : in ComplexPair;          -- input half of bus
42
43         j_val     : buffer ComplexPair;      -- output half of j bus
44         k_val     : buffer ComplexPair;      -- output half of k bus
45
46         w_in      : in ComplexPair;
47
48         j_noe     : in bit;                  -- bus control - j driving
49         k_noe     : in bit;                  -- bus control - k driving
50         load      : in bit;                  -- load in j/k values
51         ready     : out bit                  -- ready to start, output etc.
52       );
53 end FFTProcessor;
54
55 architecture behaviour of FFTProcessor is
56 begin
57
58   ------------------------------------------------------------------------------
59   -- FFT calculation process -
60   ------------------------------------------------------------------------------
61
62   fft: process
63     ------------------------------------------------------------------------
64     -- temporary variables to hold partial results
65     ------------------------------------------------------------------------
66     variable temp_Re,temp_Im : ComplexSingle;
67
68     ------------------------------------------------------------------------
69     -- aliases for separating out real and imaginary components from inputs
70     ------------------------------------------------------------------------
71     alias w_Re : ComplexSingle is w_in(11 downto 0);
72     alias w_Im : ComplexSingle is w_in(23 downto 12);
```

```
73      alias j_Re : ComplexSingle is j_val(11 downto 0);
74      alias j_Im : ComplexSingle is j_val(23 downto 12);
75      alias k_Re : ComplexSingle is k_val(11 downto 0);
76      alias k_Im : ComplexSingle is k_val(23 downto 12);
77    begin
78      ------------------------------------------------------------------------
79      -- signal we're ready to go then load in j & k on load +ve & -ve egdes
80      ------------------------------------------------------------------------
81      ready <= '1';                                   -- processor ready to go
82      wait for 100 ns;
83
84      while load /= '1' loop                          -- wait for j value ready
85        wait for 100 ns;
86      end loop;
87      j_val <= data_bus;                              -- read into j_val register
88
89      while load /= '0' loop                          -- wait for k value ready
90        wait for 100 ns;
91      end loop;
92      k_val <= data_bus;                              -- read into k_val register
93
94      ------------------------------------------------------------------------
95      -- signal busy and then get on with it. note j/k not needed after this
96      ------------------------------------------------------------------------
97      ready <= '0';                                   -- signal busy calculating
98      wait for 100 ns;                                -- wait a cycle
99      protect;                                        -- keep the above separate
100
101     ------------------------------------------------------------------------
102     -- complex multiplication Re(w*jval) - uses fpmult macro operator so do AOM
103     ------------------------------------------------------------------------
104     temp_Re:=fpmult(w_Re,j_Re);
105     temp_Re:=temp_Re - fpmult(w_Im,j_Im);
106
107     ------------------------------------------------------------------------
108     -- complex multiplication Im(w*jval)
109     ------------------------------------------------------------------------
110     temp_Im:=fpmult(w_Im,j_Re);
111     temp_Im:=temp_Im + fpmult(w_Re,j_Im);
112
113     ------------------------------------------------------------------------
114     -- combine FFT pairs j and k.
115     ------------------------------------------------------------------------
116     j_Re <= k_Re - temp_Re;
117     k_Re <= k_Re + temp_Re;
118     j_Im <= k_Im - temp_Im;
119     k_Im <= k_Im + temp_Im;
120     wait for 0 ns;                                  -- update values immediately
121
122     ------------------------------------------------------------------------
123     -- normalise results so they always stay in range -1.0->1.0 so we can keep
124     -- within the 12 bit/10 bit point range - divide by 2 using a sign extended
125     -- shift right
126     ------------------------------------------------------------------------
127     if j_Re = "111111111111" then                   -- is it -1 ? if so set to 0
128       j_Re <= "000000000000";
129     else
130       j_Re(10 downto 0) <= j_Re(11 downto 1);       -- shift right keeping top bit
131     end if;
132
133     if j_Im = "111111111111" then
134       j_Im <= "000000000000";
135     else
136       j_Im(10 downto 0) <= j_Im(11 downto 1);
137     end if;
138
139     if k_Re = "111111111111" then
140       k_Re <= "000000000000";
141     else
142       k_Re(10 downto 0) <= k_Re(11 downto 1);
143     end if;
144
145     if k_Im = "111111111111" then
146       k_Im <= "000000000000";
```

```
147      else
148        k_Im(10 downto 0) <= k_Im(11 downto 1);
149      end if;
150
151      wait for 100 ns;                              -- update and wait a cycle
152
153    end process fft;
154
155  end behaviour;
156
157  ------------------------------------------------------------------------------
```

Listing C.6 ADCCont.vhd - Analogue to digital converter control and level meter

```
1  ------------------------------------------------------------------------------
2  -- ADCCont.vhd - ADC controller and level meter     Alan Williams, October 1996
3  --
4  -- Part of the audio band spectrum analyser PhD. project.
5  ------------------------------------------------------------------------------
6  -- Controls 16071 ADC converting the 16 bit serial output into a 12 bit
7  -- parallel taking the top 11 bits and outputting as real bits 10 downto 0,
8  -- sign extend real bit 11 from bit 10, in other words a 12 bit 2's complement
9  -- fixed point number with the binary point between bits 9 and 10. All
10 -- imaginary components = 0.
11 -- Tri-state outputs are used, controlled by noe, so that data can connect
12 -- directly to the FFT ram data bus.
13 -- ADC clock is generated from the system clock divided by 2 or 4 depending on
14 -- the clk_sel input. A 10 segment vu meter is also implemented from the
15 -- sampled data indicating the highest set bit over 1024 samples.
16 --
17 -- More details can be found in PhD thesis chapter 7 and appendices.
18 ------------------------------------------------------------------------------
19 -- Modifications to optimised structural VHDL (see file "bits.vhd"):
20 --    . Make data a tri-state output controlled by the noe input.
21 ------------------------------------------------------------------------------
22
23 --use work.icode_ops.all;                              -- for simulation only
24
25 entity ADCControl is
26   port( ready   : out bit;                             -- high when data is stable
27         data    : out bit_vector(23 downto 0);         -- complex sample tri-state
28         noe     : in bit;                              -- turns data on or hi-Z
29
30         adc_fso : in bit;                              -- 16071 frame shift out
31         adc_sco : in bit;                              -- 16071 shift control out
32         adc_sdo : in bit;                              -- 16071 shift data out
33
34         clk_sel : in bit;                              -- 16071 clock sysclk /2 or /4
35         adc_clk : buffer bit;                          -- 16071 driver clock
36
37         vu_lev  : out bit_vector(3 downto 0)           -- vu level number (0-9)
38       );
39 end ADCControl;
40
41 architecture behaviour of ADCControl is
42 begin
43
44   ------------------------------------------------------------------------------
45   -- ADC clock generation process - divide system clock by 2 or 4
46   ------------------------------------------------------------------------------
47
48   clk_divide: process
49   begin
50     adc_clk <= '0';                                  -- initialise adc_clk
51     wait for 100 ns;
52
53     adc_clock: loop                                  -- loop forever toggling adc_clk
54       adc_clk <= NOT adc_clk;
55       wait for 100 ns;
56       if clk_sel = '0' then                          -- if clk_sel 0 => div by 4
57         wait for 100 ns;                             -- force a clock cycle
58         protect;
59       end if;
60     end loop adc_clock;
61   end process;
62
63   ------------------------------------------------------------------------------
64   -- Convert ADC output from serial into 24-bit parallel and do level meter
65   ------------------------------------------------------------------------------
66
67   serial_to_parallel: process
68     ----------------------------------------------------------------------------
69     -- variables for vu meter - tracks top bit over 1024 samples
70     -- cntr counts 0-1023 and vu is only updated when it overflows
71     ----------------------------------------------------------------------------
72     variable sgn : bit := '0';
```

```
73      variable top_bit : integer range 0 to 15 := 0;
74      variable cntr : bit_vector(9 downto 0) := "0000000000";
75    begin
76      ------------------------------------------------------------------------
77      -- signal data stable and wait until ADC fso signals start of next sample
78      ------------------------------------------------------------------------
79      ready <= '1';                           -- signal output stable
80      wait on adc_fso until adc_fso = '0';    -- wait for next ADC frame
81
82      ready <= '0';                           -- signal data unstable
83      data <= convert_int2bv(0,24);           -- clear the parallel data
84      wait for 0 ns;
85
86      ------------------------------------------------------------------------
87      -- read sample from sdo on rising edge of sco and put into data register
88      ------------------------------------------------------------------------
89      bit_loop: for i in 10 downto 0 loop     -- shift in the MS 11 bits
90        data(23 downto 1) <= data(22 downto 0);
91        wait on adc_sco until adc_sco = '1';  -- catch rising edge of adc_sco
92
93        data(0) <= adc_sdo;                   -- put new bit into data
94        wait for 100 ns;
95
96        if i = 10 then                        -- track sign of sample
97          sgn := adc_sdo;
98        end if;
99
100       ------------------------------------------------------------------------
101       -- vu meter works by tracking the highest bit set over 1024 samples and
102       -- then outputting this bit number to vu_lev. Only the top 10 bits are
103       -- tracked for a 10 segment display and we ignore -ve samples.
104       --
105       -- NOTE: MOODS or WARP really screws this up if its all inline or
106       -- if the /=10 is omitted !!
107       -- set top_bit to the highest +ve sample '1 bit
108       ------------------------------------------------------------------------
109       if i>top_bit AND i/=10 then
110         if adc_sdo='1' AND sgn='0' then
111           top_bit := i;
112         end if;
113       end if;
114     end loop bit_loop;
115
116     ------------------------------------------------------------------------
117     -- finally, sign extend the top data bit, data is now valid
118     ------------------------------------------------------------------------
119     data(11) <= data(10);                   -- sign extend the top bit
120     wait for 0 ns;
121
122     ------------------------------------------------------------------------
123     -- if vu update counter wraps around, update vu level output bits
124     ------------------------------------------------------------------------
125     if cntr = "0000000000" then             -- update when counter wraps
126       vu_lev <= convert_int2bv(top_bit,4);  -- update vu level output
127       top_bit := 0;                         -- reset highest bit counter
128       wait for 0 ns;                        -- update output now
129     end if;
130
131     cntr := cntr + "0000000001";            -- increment vu update counter
132   end process;
133
134 end behaviour;
135
136 ------------------------------------------------------------------------
```

Listing C.7 12-bit fixed-point multiplier macro operator source

```
1  ------------------------------------------------------------------------------
2  -- FpMult12_10.vhd - 12 bit fixed-point multiply    Alan Williams, October 1996
3  --
4  -- Used with the audio band spectrum analyser PhD. project.
5  ------------------------------------------------------------------------------
6  -- Special-purpose fixed point multiplier for numbers in the range -1.0 to 1.0.
7  -- The input number format is a 12 bit 2's complement fixed point number with
8  -- the binary point fixed between bits 9 and 10, giving a range of -2.0 to
9  -- 1.99805. Since inputs are assumed to be -1.0 to 1.0, so must the result thus
10 -- we can ignore the top result bit since it will just be a sign extension of
11 -- its predecessor. Also, output is kept to 12 bits with the same binary point
12 -- position so the top two result bits must be truncated anyway.
13 --
14 -- This code is intended to be used as the basis for a macro operator. Once
15 -- compiled (using /no_sigs), the ICODE should be edited to remove the top-
16 -- level process loop by deleting the activation on the last instruction.
17 -- Once in MOODS it can be saved unoptimised as function f_macro_10 and used
18 -- in VHDL source as macro operator "fpmult". After expansion it should be
19 -- optimised to get all the benefits of sharing etc.
20 --
21 -- More details can be found in PhD thesis chapter 7 and appendices.
22 ------------------------------------------------------------------------------
23
24 --use work.icode_ops.all;                          -- for simulation only
25
26 entity fixed_point_mult_12_10 is
27    port ( in1, in2 : in bit_vector (11 downto 0);
28           output : out bit_vector (11 downto 0)
29         );
30 end fixed_point_mult_12_10;
31
32 architecture behaviour of fixed_point_mult_12_10 is
33 begin
34
35   ------------------------------------------------------------------------------
36   -- 12 bit fixed point booth multipler process
37   ------------------------------------------------------------------------------
38   fpmult: process
39     variable prod : bit_vector (25 downto 0);
40   begin
41     prod := convert_int2bv(0,26);                 -- initialise result with in2
42     prod(12 downto 1) := in2;
43
44     ------------------------------------------------------------------------------
45     -- main loop - rshift product and add/subtract in1 as required
46     ------------------------------------------------------------------------------
47     i_loop: for i in 0 to 11 loop
48
49       case prod(1 downto 0) is
50         when "01" =>
51           prod(25 downto 13) := prod(24 downto 13) + in1;
52         when "10" =>
53           prod(25 downto 13) := prod(24 downto 13) - in1;
54         when others =>
55           null;
56       end case;
57
58       prod(24 downto 0) := prod(25 downto 1);     -- arithmetic rshift
59
60     end loop i_loop;
61
62     ------------------------------------------------------------------------------
63     -- slice result bits from prod. Would be 24->1 for integer but since this
64     -- is fixed point -1.0->1.0 we only need two bits to the left of the binary
65     -- point. Input point was between bits 9 & 10, => result is between 19 & 20
66     -- so to keep the same point position output result bits 21->10, which are
67     -- stored in prod bits 22->11.
68     ------------------------------------------------------------------------------
69     output <= prod(22 downto 11);
70     wait for 100 ns;
71   end process;
72
```

```
73 end behaviour;
74
75 ------------------------------------------------------------------------
```

Listing C.8 Complete system simulation test harness

```
1  ------------------------------------------------------------------------------
2  -- TestSys.vhd - Complete system simulation test    Alan Williams, October 1996
3  --               harness for spectrum analyser
4  --
5  -- Part of the audio band spectrum analyser PhD. project.
6  ------------------------------------------------------------------------------
7  -- Test harness for the simulation of the complete system used for testing
8  -- at all levels from behavioural, through MOODS output, all the way down to
9  -- the gate level after placement onto FPGAs which includes will timing info.
10 --
11 -- The three main components are connected through the various busses to a set
12 -- of memory simulation components and the behavioural implementation of the
13 -- ADC controller. Other bits include a full simulation of the ADC itself based
14 -- on samples stored in a file and the ability to load up and dump the contents
15 -- of the memory cells. This is used to create files of the FFT memory and more
16 -- importantly the frame buffer memory at various stages to test the correct
17 -- operation. When simulating the final FPGA output this can take several days.
18 --
19 -- More details can be found in PhD thesis chapter 7 and appendices.
20 ------------------------------------------------------------------------------
21
22 ------------------------------------------------------------------------------
23 -- Use all the standard file I/O stuff and IEEE std_logic libraries
24 ------------------------------------------------------------------------------
25 LIBRARY std;
26 USE std.textio.all;
27
28 LIBRARY ieee;
29 USE ieee.std_logic_1164.all;
30 USE ieee.std_logic_arith.all;
31 USE ieee.std_logic_textio.all;
32
33 ------------------------------------------------------------------------------
34 -- Use special memory cells and icode operators for behavioural bits
35 ------------------------------------------------------------------------------
36 USE work.memory.all;
37 USE work.icode_ops.all;
38
39 ------------------------------------------------------------------------------
40
41 entity testfftdisp is
42 end testfftdisp;
43
44 architecture structure of testfftdisp is
45
46   ------------------------------------------------------------------------------
47   -- component declarations for the three main system devices
48   ------------------------------------------------------------------------------
49
50   component VideoProcessor
51         port (
52               x: in std_logic_vector(8 downto 0);
53               y: in std_logic_vector(7 downto 0);
54               command: in std_logic_vector(3 downto 0);
55               execute: in std_logic_vector(1 downto 1);
56               vid_data: inout std_logic_vector(7 downto 0);
57               blanking: in std_logic_vector(3 downto 0);
58               res_data: in std_logic_vector(7 downto 0);
59               ack: out std_logic_vector(1 downto 1);
60                vid_addr: out std_logic_vector(13 downto 0);
61               vid_rnw: out std_logic_vector(1 downto 1);
62               vid_noe: out std_logic_vector(1 downto 1);
63               res_addr: out std_logic_vector(10 downto 0);
64               vclk: out std_logic_vector(1 downto 1);
65               vid: out std_logic_vector(1 downto 1);
66               hsync: out std_logic_vector(1 downto 1);
67               vsync: out std_logic_vector(1 downto 1);
68               reset, clock: in std_logic
69             );
70   end component;
71
72   component FFTControl
```

```
73      port(
74           controls: in std_logic_vector(3 downto 0);
75           adc_ready: in std_logic_vector(1 downto 1);
76           fft_ready: in std_logic_vector(1 downto 1);
77           mod_data: in std_logic_vector(7 downto 0);
78           vid_ack: in std_logic_vector(1 downto 1);
79           sys_reset: out std_logic_vector(1 downto 1);
80           ram_addr: out std_logic_vector(9 downto 0);
81           ram_rnw: out std_logic_vector(1 downto 1);
82           ram_noe: out std_logic_vector(1 downto 1);
83           rom_addr: out std_logic_vector(9 downto 0);
84           rom_noe: out std_logic_vector(1 downto 1);
85           adc_noe: out std_logic_vector(1 downto 1);
86           fft_load: out std_logic_vector(1 downto 1);
87           fft_j_noe: out std_logic_vector(1 downto 1);
88           fft_k_noe: out std_logic_vector(1 downto 1);
89           vid_command: out std_logic_vector(3 downto 0);
90           vid_x: out std_logic_vector(8 downto 0);
91           vid_y: out std_logic_vector(7 downto 0);
92           vid_execute: out std_logic_vector(1 downto 1);
93           reset, clock: in std_logic
94         );
95   end component;
96
97   component FFTProcessor
98          port (
99           data_bus: inout std_logic_vector(23 downto 0);
100          w_in: in std_logic_vector(23 downto 0);
101          j_noe: in std_logic_vector(1 downto 1);
102          k_noe: in std_logic_vector(1 downto 1);
103          load: in std_logic_vector(1 downto 1);
104          ready: out std_logic_vector(1 downto 1);
105          reset, clock: in std_logic
106             );
107  end component;
108
109  ------------------------------------------------------------------------
110  -- interconnecting busses, control lines etc.
111  ------------------------------------------------------------------------
112
113  signal sys_execute  : std_logic_vector(1 downto 1);      -- execute command
114  signal sys_ack      : std_logic_vector(1 downto 1);      -- command executing
115
116  signal sys_vid_addr : std_logic_vector(13 downto 0);     -- video address bus
117  signal sys_vid_data : std_logic_vector(7 downto 0) bus;  -- video memory data
118  signal sys_vid_rnw  : std_logic_vector(1 downto 1);      -- video memory rnw
119  signal sys_vid_noe  : std_logic_vector(1 downto 1);      -- video memory noe
120  signal sys_blanking : std_logic_vector(3 downto 0);      -- hor blanking time
121
122  signal sys_res_addr : std_logic_vector(10 downto 0);     -- resource ROM addr
123  signal sys_res_data : std_logic_vector(7 downto 0);      -- resource ROM data
124
125  signal sys_vclk     : std_logic_vector(1 downto 1);      -- screen video data
126  signal sys_vid      : std_logic_vector(1 downto 1);      -- screen video clk
127  signal sys_hsync    : std_logic_vector(1 downto 1);      -- screen hsync
128  signal sys_vsync    : std_logic_vector(1 downto 1);      -- screen vsync
129
130  signal sys_reset    : std_logic_vector(1 downto 1);      -- entire system rst
131  signal sys_knobs    : std_logic_vector(3 downto 0);      -- twiddly buttons
132
133  signal sys_fft_addr : std_logic_vector(9 downto 0);      -- FFT ram addr bus
134  signal sys_fft_data : std_logic_vector(23 downto 0);     -- FFT ram data bus
135  signal sys_fft_rnw  : std_logic_vector(1 downto 1);      -- FFT ram rw cntrl
136  signal sys_fft_noe  : std_logic_vector(1 downto 1);      -- FFT ram tri-state
137
138  signal sys_w_addr   : std_logic_vector(9 downto 0);      -- w rom address bus
139  signal sys_w_data   : std_logic_vector(23 downto 0) bus; -- w rom data bus
140  signal sys_w_noe    : std_logic_vector(1 downto 1);      -- w rom op enable
141
142  signal sys_adc_ready : std_logic_vector(1 downto 1);     -- ADC control ready
143  signal sys_adc_noe   : std_logic_vector(1 downto 1);     -- ADC output enable
144  signal sys_adc_data  : std_logic_vector(23 downto 0);    -- ADC data op enab
145  signal sys_adc_clock : std_logic;                        -- ADC clock
146
```

```
147   signal sys_fft_ready : std_logic_vector(1 downto 1);      -- FFT proc ready
148   signal sys_fft_load  : std_logic_vector(1 downto 1);      -- FFT proc load
149   signal sys_fft_j_noe : std_logic_vector(1 downto 1);      -- FFT select j op
150   signal sys_fft_k_noe : std_logic_vector(1 downto 1);      -- FFT select k op
151
152   signal sys_mod_data  : std_logic_vector(7 downto 0);      -- modulus rom data
153   signal sys_mod_addr  : std_logic_vector(17 downto 0);     -- modulus rom addr
154
155   signal init_mem      : std_logic;              -- init memories from files
156   signal dump_mem      : std_logic;              -- dump memories to files
157
158   signal sys_clock     : std_logic;              -- system clock
159   signal sys_pwr_on_rst : std_logic;             -- system power on reset
160
161   signal adc_fso     : std_logic;                -- ADC frame shift out
162   signal adc_sco     : std_logic;                -- ADC shift control out
163   signal adc_sdo     : std_logic;                -- ADC shift data out
164
165   signal samp_no      : integer := 0;            -- sample counter for tracking
166
167   ------------------------------------------------------------------------
168   -- make the various component entities visible
169   ------------------------------------------------------------------------
170   for all: VideoProcessor use entity work.VideoProcessor;
171   for all: FFTControl use entity work.FFTControl;
172   for all: FFTProcessor use entity work.FFTProcessor;
173   for all: sram use entity work.sram;
174   for all: rom use entity work.rom;
175
176 begin
177
178   ------------------------------------------------------------------------
179   -- Generate 10MHz system clock
180   ------------------------------------------------------------------------
181   clk_drv: process
182   begin
183     sys_clock <= '0';
184     wait for 50 ns;
185     sys_clock <= '1';
186     wait for 50 ns;
187   end process;
188
189   ------------------------------------------------------------------------
190   -- Generate 200 ns power on reset pulse
191   ------------------------------------------------------------------------
192   pwr_on_rst_drv: process
193   begin
194     sys_pwr_on_rst <= '1';
195     wait for 200 ns;
196     sys_pwr_on_rst <= '0';
197     wait;
198   end process;
199
200   ------------------------------------------------------------------------
201   -- Test jig for simulating buttons and loading and dumping memory contents
202   ------------------------------------------------------------------------
203   test_jig: process
204   begin
205     init_mem <= '0';                  -- zero control sigs
206     dump_mem <= '0';
207     sys_knobs <= "0000";
208     wait for 250 ns;                  -- wait until reset done
209
210     init_mem <= '1';
211     wait for 25 ns;
212
213     init_mem <= '0';
214     wait for 25 ns;
215
216     ------------------------------------------------------------------------
217     -- wait for screen to initialise and sampling to start, then dump memories
218     ------------------------------------------------------------------------
219
220     wait until sys_w_noe = "0";       -- screen init
```

```
221     wait until sys_adc_noe = "0";       -- adc sampling
222
223     dump_mem <= '1';
224     wait for 25 ns;
225     dump_mem <= '0';
226     wait for 25 ns;
227
228     ------------------------------------------------------------------------
229     -- dump again after the next screen initialise done
230     ------------------------------------------------------------------------
231
232     wait until sys_w_noe = "0";         -- screen init
233
234     dump_mem <= '1';
235     wait for 25 ns;
236     dump_mem <= '0';
237     wait for 25 ns;
238
239     ------------------------------------------------------------------------
240     -- when sampling then hold, zoom in and pan twice
241     ------------------------------------------------------------------------
242
243     wait until sys_adc_noe = "0";       -- sampling
244
245     wait for 5000 ns;
246     sys_knobs <= "1110";                -- hold and zoom in
247     wait for 1000 ns;
248     sys_knobs <= "0000";
249     wait for 1000 ns;
250
251     sys_knobs <= "0001";                -- shift along
252     wait for 1000 ns;
253     sys_knobs <= "0000";
254     wait for 1000 ns;
255
256     sys_knobs <= "0001";
257     wait for 1000 ns;
258     sys_knobs <= "0000";
259     wait for 1000 ns;
260
261     ------------------------------------------------------------------------
262     -- dump again after the next screen initialise
263     ------------------------------------------------------------------------
264
265     wait until sys_w_noe = "0";         -- screen init
266
267     dump_mem <= '1';
268     wait for 25 ns;
269     dump_mem <= '0';
270     wait for 25 ns;
271
272     ------------------------------------------------------------------------
273     -- wait until outputting results then press hold and zoom in
274     ------------------------------------------------------------------------
275
276     wait until sys_w_noe = "1";         -- fft output
277
278     wait for 10000 ns;
279     sys_knobs <= "1110";                -- hold off and zoom in
280     wait for 1000 ns;
281     sys_knobs <= "0000";
282     wait for 1000 ns;
283
284     ------------------------------------------------------------------------
285     -- wait until screen init done, then dump - do for 2 frames
286     ------------------------------------------------------------------------
287
288     wait until sys_w_noe = "0";         -- screen init
289
290     dump_mem <= '1';
291     wait for 25 ns;
292     dump_mem <= '0';
293     wait for 25 ns;
294
```

```
295     wait until sys_w_noe = "0";         -- screen init
296
297     dump_mem <= '1';
298     wait for 25 ns;
299     dump_mem <= '0';
300     wait for 25 ns;
301
302     wait;                               -- wait forever
303   end process;
304
305   ------------------------------------------------------------------------
306   -- Instantiate the video processor component
307   ------------------------------------------------------------------------
308   video_drv: VideoProcessor
309     port map( x         => sys_w_data(16 downto 8),
310               y         => sys_w_data(7 downto 0),
311               command   => sys_w_data(20 downto 17),
312               execute   => sys_execute,
313               ack       => sys_ack,
314
315               vid_addr => sys_vid_addr,
316               vid_data => sys_vid_data,
317               vid_rnw  => sys_vid_rnw,
318               vid_noe  => sys_vid_noe,
319               blanking => sys_blanking,
320
321               res_addr => sys_res_addr,
322               res_data => sys_res_data,
323
324               vclk     => sys_vclk,
325               vid      => sys_vid,
326               hsync    => sys_hsync,
327               vsync    => sys_vsync,
328
329               reset    => sys_reset(1),
330               clock    => sys_clock
331             );
332
333   ------------------------------------------------------------------------
334   -- hsync timing control - sets the length of the hsync blanking pulse
335   ------------------------------------------------------------------------
336   sys_blanking <= "1001";
337
338   ------------------------------------------------------------------------
339   -- Instantiate the main system controller
340   ------------------------------------------------------------------------
341   fft_cont: FFTControl
342     port map( sys_reset   => sys_reset,
343               controls    => sys_knobs,
344
345               ram_addr    => sys_fft_addr,
346               ram_rnw     => sys_fft_rnw,
347               ram_noe     => sys_fft_noe,
348               rom_addr    => sys_w_addr,
349               rom_noe     => sys_w_noe,
350
351               adc_ready   => sys_adc_ready,
352               adc_noe     => sys_adc_noe,
353
354               fft_ready   => sys_fft_ready,
355               fft_load    => sys_fft_load,
356               fft_j_noe   => sys_fft_j_noe,
357               fft_k_noe   => sys_fft_k_noe,
358
359               mod_data    => sys_mod_data,
360
361               vid_command => sys_w_data(20 downto 17),
362               vid_x       => sys_w_data(16 downto 8),
363               vid_y       => sys_w_data(7 downto 0),
364               vid_execute => sys_execute,
365               vid_ack     => sys_ack,
366
367               reset    => sys_pwr_on_rst,
368               clock    => sys_clock
```

```
369             );
370
371     ------------------------------------------------------------------------
372     -- connection from FFT ram data bus to modulus rom address bus
373     -- modulus address is: top 9 bits imag => a17-a9 & top 9 bits real => a8-a0
374     ------------------------------------------------------------------------
375     sys_mod_addr <= sys_fft_data(20 downto 12) & sys_fft_data(8 downto 0);
376
377     ------------------------------------------------------------------------
378     -- Instantiate the FFT processor
379     ------------------------------------------------------------------------
380     fft_proc: FFTProcessor
381       port map( data_bus => sys_fft_data,
382                 w_in     => sys_w_data,
383                 j_noe    => sys_fft_j_noe,
384                 k_noe    => sys_fft_k_noe,
385                 load     => sys_fft_load,
386                 ready    => sys_fft_ready,
387                 reset    => sys_reset(1),
388                 clock    => sys_clock
389               );
390
391     ------------------------------------------------------------------------
392     -- ADC clock generator - behavioural version from ADCCont.vhd
393     ------------------------------------------------------------------------
394     clk_divide: process
395     begin
396       sys_adc_clock <= '0';
397       wait for 100 ns;
398       sys_adc_clock <= '1';
399       wait for 100 ns;
400     end process;
401
402     ------------------------------------------------------------------------
403     -- ADC controller - behavioural version identical to ADCCont.vhd
404     ------------------------------------------------------------------------
405     serial_to_parallel: process
406     begin
407       sys_adc_ready <= "1";                    -- signal output stable
408       if adc_fso /= '1' then
409         wait until adc_fso = '1';              -- wait for next ADC frame
410       end if;
411       wait for 100 ns;
412       wait until adc_fso = '0';
413
414       sys_adc_ready <= "0";                    -- signal data unstable
415       sys_adc_data <= convert_int2bv(0,24);    -- clear the parallel data
416       wait for 0 ns;
417
418       bit_loop: for i in 0 to 10 loop          -- shift in the MS 11 bits
419         sys_adc_data(23 downto 1) <= sys_adc_data(22 downto 0);
420         if adc_sco /= '0' then
421           wait until adc_sco = '0';            -- catch rising edge of adc_sco
422         end if;
423         wait until adc_sco = '1';
424         sys_adc_data(0) <= adc_sdo;            -- put new bit into data
425         wait for 100 ns;
426       end loop bit_loop;
427
428       sys_adc_data(11) <= sys_adc_data(10);    -- sign extend the top bit
429       wait for 100 ns;
430     end process;
431
432     ------------------------------------------------------------------------
433     -- connect adc controller to FFT ram data bus under sys_adc_noe control
434     ------------------------------------------------------------------------
435     sys_fft_data <= sys_adc_data when (sys_adc_noe = "0")
436                     else (others => 'Z');
437
438     ------------------------------------------------------------------------
439     -- ADC - emulates 16071 ADC returning data from adc_out.dat file
440     ------------------------------------------------------------------------
441     adc_proc: process
442       file     infile  : text is in "adc_out.dat";
```

```
443     variable buff    : line;
444     variable int_val : integer;
445     variable sample  : std_logic_vector(15 downto 0);
446   begin
447     adc_fso <= '0';                              -- initialise control sigs
448     adc_sco <= '0';
449     adc_sdo <= '0';
450     wait for 0 ns;
451
452     readline(infile, buff);                      -- read the first line
453
454     ------------------------------------------------------------------------
455     -- read each line in file, outputting like ADC until end of file found
456     ------------------------------------------------------------------------
457     inf_loop: loop
458       if endline(buff) then                      -- get next line if not eof
459         if NOT endfile(infile) then
460           readline(infile, buff);
461         end if;
462       end if;
463
464       if NOT endfile(infile) then                -- if not eof read value
465         read(buff, int_val);
466       else
467         int_val := 0;
468       end if;
469
470       sample := to_stdlogicvector(int_val,16);   -- convert to bit vector
471       samp_no <= samp_no + 1;
472
473       ------------------------------------------------------------------------
474       -- shift 16 data bits out
475       ------------------------------------------------------------------------
476       for i in 15 downto 0 loop
477         wait on sys_adc_clock until sys_adc_clock = '1';
478
479         adc_sdo <= sample(i);
480         adc_sco <= '0';
481         adc_fso <= '0';
482         wait on sys_adc_clock until sys_adc_clock = '1';
483         wait on sys_adc_clock until sys_adc_clock = '1';
484
485         adc_sco <= '1';
486         wait on sys_adc_clock until sys_adc_clock = '1';
487       end loop;
488
489       ------------------------------------------------------------------------
490       -- shift out 16 zeros
491       ------------------------------------------------------------------------
492       for i in 15 downto 0 loop
493         wait on sys_adc_clock until sys_adc_clock = '1';
494         adc_sdo <= '0';
495         adc_sco <= '0';
496         adc_fso <= '1';
497         wait on sys_adc_clock until sys_adc_clock = '1';
498         wait on sys_adc_clock until sys_adc_clock = '1';
499
500         adc_sco <= '1';
501         wait on sys_adc_clock until sys_adc_clock = '1';
502       end loop;
503     end loop;
504
505     wait;                                        -- to infinity and beyond
506   end process;
507
508   ------------------------------------------------------------------------
509   -- Video memory component 16k x 8bits, dumps to vram_dump.dat on dump_mem
510   ------------------------------------------------------------------------
511   video_ram: sram
512     generic map( s => 14,
513                  w => 8,
514                  bytes => 40,
515                  field => 8,
516                  op_format => binary,
```

```
517                 op_fname => "vram_dump.dat"
518                )
519     port map( addr => sys_vid_addr,
520               data => sys_vid_data,
521               nwe => sys_vid_rnw(1),
522               noe => sys_vid_noe(1),
523               nce => zero,
524               ce => one,
525
526               clear => init_mem,
527               set => zero,
528               load => zero,
529               from_file => zero,
530               to_file => dump_mem
531              );
532
533   ------------------------------------------------------------------------
534   -- Video resource rom component 2k x 8bits, read from resource_rom.dat
535   -- on init_mem edge
536   ------------------------------------------------------------------------
537   resource_rom: rom
538     generic map( s => 11,
539                  w => 8,
540                  ip_format => binary,
541                  ip_fname => "resource_rom.dat"
542                 )
543     port map( addr => sys_res_addr,
544               data => sys_res_data,
545               noe => zero,
546               nce => zero,
547               ce => one,
548
549               from_file => init_mem
550              );
551
552   ------------------------------------------------------------------------
553   -- FFT scratch ram 1k x 24 bits, read from fft_ram_in.dat on init_mem,
554   -- written to fft_ram_out.dat on dump_mem
555   ------------------------------------------------------------------------
556   fft_ram: sram
557     generic map( s => 10,
558                  w => 24,
559                  ip_format => decimal,
560                  ip_fname => "fft_ram_in.dat",
561                  op_format => decimal,
562                  op_fname => "fft_ram_out.dat"
563                 )
564     port map( addr => sys_fft_addr,
565               data => sys_fft_data,
566               nwe => sys_fft_rnw(1),
567               noe => sys_fft_noe(1),
568               nce => zero,
569               ce => one,
570
571               clear => zero,
572               set => zero,
573               load => zero,
574               from_file => zero,
575               to_file => dump_mem
576              );
577
578   ------------------------------------------------------------------------
579   -- FFT "roots of -1" rom 1k x 24 bits, read from w_rom.dat on init_mem
580   ------------------------------------------------------------------------
581   w_rom: rom
582     generic map( s => 10,
583                  w => 24,
584                  ip_format => binary,
585                  ip_fname => "w_rom.dat"
586                 )
587     port map( addr => sys_w_addr,
588               data => sys_w_data,
589               noe => sys_w_noe(1),
590               nce => zero,
```

```
591                ce => one,
592
593                from_file => init_mem
594               );
595
596    ------------------------------------------------------------------------
597    -- Complex number modulus rom 256k x 8 bits,read from mod_rom.dat on init_mem
598    ------------------------------------------------------------------------
599    mod_rom: rom
600      generic map( s => 18,
601                   w => 8,
602                   ip_format => decimal,
603                   ip_fname => "mod_rom.dat"
604                  )
605      port map( addr => sys_mod_addr,
606                data => sys_mod_data,
607                noe => zero,
608                nce => zero,
609                ce => one,
610
611                from_file => init_mem
612               );
613
614  end structure;
615
616  ------------------------------------------------------------------------
```

Glossary

CDFG - Control Dataflow Graph - Combined control and data flow internal representation, used in the Chippe synthesis system.

ELLA - An alternative hardware description language, now largely obsolete in favour of more modern alternatives, such as VHDL.

ETPN - Extended Timed Petri-Net - Internal control and data flow representation used by the CAMAD synthesis system.

ICODE - Intermediate Code - Register transfer level description used as input to the main MOODS synthesis block, compiled from VHDL via the VHDL2IC front end.

MOODS - Multiple Objective Optimisation in control and Data path Synthesis - The synthesis system detailed in this thesis.

SCHOLAR - A hardware description language, used as the original input to MOODS prior to adoption of VHDL.

VHDL - Very High speed IC Hardware Description Language - Standard hardware description language (IEEE standard 1076) used as the main input to the MOODS synthesis system.

FPGA - Field Programmable Gate Array - Programmable device featuring an array of configurable logic blocks and registers.

ASIC - Application Specific Integrated Circuit - Fully custom designed integrated circuit.

References

1. Gajski, Daniel D. [editor], "Silicon Compilation", Addison-Wesley 1988, ISBN 0201099152.

2. Preas, Bryan and Lorenzitti, Michael [editors], "Physical Design Automation of VLSI Systems", The Benjamin/Cummings Publishing Company 1988, ISBN 0805301429.

3. Baker, Keith R., "Multiple Objective Optimisation of Data and Control Paths in a Behavioural Silicon Compiler", PhD Thesis, University of Southampton, September 1992.

4. Baker, K.R. - Currie, A.J. - Nichols, K.G., "Multiple Objective Optimisation in a Behavioural Synthesis System", IEE Proceedings - G, Vol. 140, No.4 August 1993, pp. 253-260.

5. Baker, K.R. - Brown, A.D. - Currie, A.J., "Optimisation Efficiency in Behavioural Synthesis", IEE Proceedings on Circuits, Devices and Systems, Vol. 141, No. 5, October 1994, pp. 399-406.

6. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-1987", IEEE Catalog No. SH11957, 1987.

7. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-1993", IEEE Catalog No. SH16840, 1993.

8. McFarland, M C - Parker, A C - Camposano, R, "Tutorial on High-Level Synthesis", Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 330-336.

9. McFarland, M.C. - Parker, A.C. - Camposano, R., "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 78, February 1990, pp. 301-318.

10. Gajski, Daniel D. - Ramachandran, Loganath, "Introduction to High-Level Synthesis", IEEE Design and Test of Computers, Vol. 11, No. 4, Winter 1994, pp. 45-54.

11. Camposano, R., "From Behavior to Structure: High-Level Synthesis", IEEE Design and Test of Computers, Col. 7, Nol. 5, October 1990, pp. 8-19.

12. Gajski, Daniel D., "Introduction to Silicon Compilation", Silicon Compilation (D. Gajski, Ed.), Addison-Wesley 1988, pp. 1-48.

13. Nijhar, T.P.K. - Brown, A.D., "Source Level Optimisation of VHDL for Behavioural Synthesis", IEE Proceedings on Computers and Digital Techniques, Vol. 144, No.1, January 1997.

14. Nijhar, T.P.K. - Brown, A.D., "HDL-Specific Source Level Behavioural Optimisation", IEE Proceedings on Computers and Digital Techniques, Vol. 144, No.2, March 1997, pp. 138-144.

15. De Micheli, Giovanni, "Synthesis and Optimization of Digital Circuits", McGraw-Hill International Editions 1994, ISBN 0070163332.

16. Lee, Tsing-Fa - Wu, Allen C.H. - Gajski, Daniel D., "A Transformation-Based Method for Loop Folding", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, No. 4, April 1994, pp. 439-450.

17. Brewer, Forrest - Gajski, Daniel, "Chippe: A System for Constraint Driven Behavioral Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 9, No. 7, July 1990. pp. 681-695.

18. Gajski, Daniel D. - Orailoglu, Alex, "Flow Graph Representation", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 503-509.

19. Peng, Zebo, "A Formal Methodology for Automated Synthesis of VLSI Systems", PhD Thesis, Linköping University, 1987, ISBN 9178702259.

20. Eles, Petru - Kuchcinski, Krzysztof - Peng, Zebo - Doboli, Alexa, "Timing Constraint Specification and Synthesis in Behavioral VHDL", Proceedings EuroDAC '95, Session S-04, 1995.

21. Walker, Robert A. - Thomas, Donald E., "Behavioral Level Transformation in the CMU-DA System", Proceedings of the 20th ACM/IEEE Design Automation Conference, 1983, pp. 788-789.

22. Walker, Robert A. - Thomas, Donald E., "Design Representation and Transformation in the System Architect's Workbench", Proceedings of the International Conference Computer Design (ICCD), 1987, pp. 166-9.

23. Thomas, Donald E. - Blackburn, Robert L. - Rajan, J., "Linking the Behavioral and Structural Domains of Representation for Digital Systems Design", IEEE Transactions on Computer-Aided Design, Vol. 6, No. 1, 1987, pp. 103-110.

24. Eles, Petru - Kuchcinski, Krzysztof - Peng, Zebo - Minea, Marius, "Compiling VHDL into a High-Level Synthesis Design Representation", Proceedings EuroDAC '92, 1992, pp. 604-609.

25. Brown, A.D. - Baker, K.R. - Williams, A.C., "On the Nature of the Design Space", In Preparation.

26. McFarland, Michael C., "Reevaluating the Design Space for Register-Transfer Hardware Synthesis", Proceedings of the International Conference on Computer Design (ICCD), 1987, pp. 262-265.

27. De Micheli, Giovanni - Ku, David - Mailhot, Frédéric - Truong, Thomas, "The Olympus Synthesis System", IEEE Design and Test of Computers, October 1990, pp. 37-53.

28. Ku, David - De Micheli, Giovanni, "High-Level Synthesis and Optimization Strategies in Hercules and Hebe", Proceedings IFIP Working Conference on Logic and Architecture Synthesis, May 1990, pp. 111-120.

29. Marwedel, Peter, "A New Synthesis Algorithm for the MIMOLA Software System", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 271-277.

30. Zimmermann, G., "The MIMOLA Design System: A Computer Aided Digital Processor Design Method", Proceedings of the 16th ACM/IEEE Design Automation Conference, June 1979, pp. 53-58.

31. Trickey, Howard, "Flamel: A High-Level Hardware Compiler", IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 2, March 1987, pp. 259-269.

32. Camposano, Raul - Rosenstiel, Wolfgang, "Synthesizing Circuits From Behavioral Descriptions", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 2, February 1989, pp. 171-180.

33. Paulin, P.G. - Knight, J.P., "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 6, June 1989, pp.661-679.

34. Camposano, Raul - Bergamaschi, Reinaldo A., "Redesign Using State Splitting", EDAC '90, Glasgow, March 1990 / IBM Research Report.

35. Peng, Zebo, "Synthesis of VLSI Systems with the CAMAD Design Aid", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 278-284.

36. Tseng, C.J. - Siewiorek, D.P., "Automated Synthesis of Data Paths in Digital Systems", IEEE Transactions on Computer-Aided Design, Vol. 5, July 1986, pp. 379-395.

37. McFarland, Michael C., "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 474-480.

38. Pangrl, Barry M. - Gajski, Daniel D., "State Synthesis and Connectivity Binding for Microarchitecture Compilation", IEEE International Conference on Computer-Aided Design (ICCAD-86), 1986, pp. 210-213.

39. Parker, Alice C. - Mlinar, Mitch - Pizarro, Jorge, "MAHA: A Program for Datapath Synthesis", Proceedings of the 23rd ACM/IEEE Design Automation Conference, July 1986, pp. 461-466.

40. Devadas, Srinivas - Newton, A. Richard, "Algorithms for Hardware Allocation in Data Path Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 7, July 1989, pp. 768-81.

41. Parker, Alice C. - Hafer, Lou, "The Application of a Hardware Descriptive Language for Design Automation", Information Technology, January 1978, pp. 349-355.

42. Hafer, Louis J. - Parker, Alice C., "Automated Synthesis of Digital Hardware", IEEE Transactions on Computers, Vol. 31, No. 2, February 1982, pp. 93-109.

43. Hitchcock, Charles Y - Thomas, Donald E, "A Method of Automatic Data Path Synthesis", Proceedings of the 20th ACM/IEEE Design Automation Conference, 1983 IEEE, pp. 484-489.

44. Thomas, D.E. - Dirkes, E.M. - Walker, R.A. - Rajan, J.V. - Nestor, J.A. Blackburn, R.L., "The System Architect's Workbench", Proceedings of the 25th ACM/IEEE Design Automation Conference, 1988, pp. 337-343.

45. Tseng, Chia-Jeng - Siewiorek, Daniel P, "Facet: A Procedure for the Automated Synthesis of Digital Systems", Proceedings of the 20th ACM/IEEE Design Automation Conference, 1983, pp. 490-496.

46. Septien, J. - Mozos, D. - Tirado, J.F. - Hermida, R. - Fernandez, M. - Mecha, H., "FIDIAS: An Integral Approach to High-Level Synthesis", IEE Proceedings - Circuits, Devices and Systems, Vol. 142, No. 4, August 1995.

47. Burich, Misha R., "Design of Module Generators and Silicon Compilers", Silicon Compilation (D. Gajski, Ed.), Addison-Wesley 1988, pp. 49-94.

48. Nance, S. - Starr C. - Duyn, B. - Kliment, M., "Cell-layout compilers Simplify Custom IC Design", EDN, 15th September 1983.

49. Gajski, Daniel D. - Lin, Y-L. Steve, "Module Generation and Silicon Compilation", Physical Design Automation of VLSI Systems (B. Preas, M. Lorenzetti, Ed.), The Benjamin/Cummings Publishing Company 1988, pp. 283-346.

50. Tuck, Barbera, "Chase for Process Portability Prompts Advances in Cell Library Tools", Computer Design, 1 December 1990, pp. 46-49.

51. De Man, H. - Rabaey, J. - Claesen, L., "Cathedral II - A Silicon Compiler for Digital Signal Processing", IEEE Design and Test, December 1986, pp.13-25.

52. Six, P. - Vanderweerd, I. - De Man, H., "An Intelligent Module Generator Environment", Proceedings of the 23rd ACM/IEEE Design Automation Conference, July 1986, pp. 730-735.

53. Six, P. - Vanderweerd, I. - De Man, H., "An Interactive Environment for Creating Module Generators", Proceedings ESSCIRC, September 1986, pp. 65-70.

54. Wolf, Wayne, "An Object-Oriented, Procedural Database for VLSI Chip Planning", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 744-751.

55. Theeuwen, J.F.M. - Arts, H.M.A.M. - van Eijndhoven, J.T.J. - Sleuters, H.J.H. - Wijdeven, J.H.P., "Module Generation in an Architectural Synthesis Environment", IFIP Transactions A - Computer Science and Technology, Vol. A-22, 1993, pp. 373-384.

56. McFarland, Michael C. - Kowalski, Thaddeus J., "Incorporating Bottom-Up Design into Hardware Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 9, No. 9, September 1990, pp. 938-950.

57. Kowalski, T.J. - Thomas, D.E., "The VLSI Design Automation Assistant: What's in a Knowledge Base", Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985, pp. 252-258.

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

59. Smith, William D. - Jasica, Jeffrey R. - Hartman, Michael J. - d'Abreu, Manuel A., "Flexible Module Generation in the FACE Design Environment", Proceedings of the IEEE International Conference on Computer Aided Design, 7-10th November 1988, pp. 396-399.

60. Müzner, A., "Building block Generation Considering the Inherent Hierarchy of Arithmetic Operations", Proceedings IFIP Working Conference on Logic and Architecture Synthesis, May 1990, pp. 297-306.

61. Müzner, Andreas, "BADGE - A Synthesis Tool for Customized Arithmetic Building Blocks", IFIP Transactions A - Computer Science and Technology, Vol. A-22, 1993, pp. 359-371.

62. Gasteier, M. - Wehn, N. - Glesner, M., "Synthesis of Complex VHDL Operators", Proceedings EuroDAC '93, 1993, pp. 566-571.

63. Ly, Tai - Knapp, David - Miller, Ron - MacMillen, Don, "Scheduling using Behavioral Templates", Proceedings of the 32nd ACM/IEEE Design Automation Conference, Session 7, 1995.

64. Park, Nohbyung - Parker, Alice, "SEHWA: A Program for Synthesis of Pipelines", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp. 454-460.

65. Park, Nohbyung - Parker, Alice C., "SEHWA: A Software Package for Synthesis of Pipelines from Behavioral Specifications", IEEE Transactions on Computer-Aided Design, Vol. 7, No. 3, March 1988, pp. 356-370.

66. Saunders, L., "The IBM VHDL Design System", Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987, pp. 484-490.

67. "TransGate/TransTest User Guide", TransEDA Limited, Version 1.2.8, 1992.

68. Lis, Joseph S. - Gajski, Daniel D., "Synthesis from VHDL", Proceedings of the ICCD, 1988, pp. 378-381.

69. Biesenack, J. - Koster, Michael - Langmaier, Anton - Ledeux, Stephane - März, Sabine - Payer, Michael - Pilsl, Michael - Rumler, Steffen - Soukup, Holger - When, Norbert - Duzy, Peter, "The Siemens High-Level Synthesis System CALLAS", IEEE Transactions on VLSI Systems, Vol. 1, No. 3, September 1993, pp. 244-253.

70. Nagasamy, V. - Berry, N. - Dangelo, C., "Specification, Planning, and Synthesis in a VHDL Design Environment", IEEE Design and Test of Computers, June 1992, pp. 58-68.

71. Wolf, Wayne - Takach, Andrés - Huang, Chun-Yao - Manno, Richard, "The Princeton University Behavioral Synthesis System", Proceedings of the 29th ACM/IEEE Design Automation Conference, 1992, pp. 182-187.

72. Roy, J. - Kumar, N. - Dutta, R. - Vemuri, R., "DSS: A Distributed High-Level Synthesis System", IEEE Design and Test of Computers, June 1992, pp. 18-32.

73. Eles, Petru - Kuchcinski, Krzysztof - Peng, Zebo - Minea, Marius, "Synthesis of VHDL Concurrent Processes", Proceedings EuroDAC '94, Session V-04, 1994.

74. "Leapfrog VHDL Simulator User Guide", Cadence Design Systems, Version 2.2, June 1995.

75. "Getting Started with V-System - VHDL and Verilog Simulation for Workstations and PCs", Model Technology, Version 4.4, April 1996.

76. "VHDLCover User Guide", TransEDA Limited, Version 3.0.3, August 1996.

77. Lis, J.S. - Gajski, D.D., "VHDL Synthesis Using Structured Modeling", Proceedings of the 26th ACM/IEEE Design Automation Conference, 1989, pp. 606-609.

78. Camposano, R. - Saunders, L.F. - Tabet, R.M., "VHDL as Input for High Level Synthesis", IEEE Design and Test of Computers, March 1991, pp. 43-49.

79. Vahid, Frank - Narayan, Sanjiv - Gajski, Daniel D., "A Transformation for Integrating VHDL Behavioral Specification with Synthesis and Software Generation", Proceedings EuroDAC '94, Session V-04, 1994.

80. Rushton, A., "VHDL for Logic Synthesis", McGraw-Hill, ISBN 0-07-709092-6.

81. Ramachandran, Loganath - Narayan, Sanjiv - Vahid, Frank - Gajski, Daniel D., "Synthesis of Functions and Procedures in Behavioral VHDL", Proceedings EuroDAC '93, 1993, pp. 560-565.

82. Stoll, A. - Duzy, P., "High-Level Synthesis from VHDL with Exact Timing Constraints", Proceedings of the 29th ACM/IEEE Design Automation Conference, 1992, pp. 188-193.

83. Ramachandran, Loganath - Vahid, Frank - Narayan, Sanjiv - Gajski, Daniel D., "Semantics and Synthesis of Signals in Behavioral VHDL", Proceedings EuroDAC '92, 1992, pp. 616-621.

84. Genoe, Mark - Vanoostende, Paul - van Wauwe, Geert, "On the use of VHDL-Based Behavioral Synthesis for Telecom ASIC Design", The International Symposium on System Synthesis, Session 3, 1995.

85. Kisson, P. - Closse, E. - Bergher, L. - Jerraya, A.A., "Industrial Experimentation of High-Level Synthesis", Proceedings EuroDAC '93, 1993, pp. 506-511.

86. Fuhrman, T.E., "Industrial Extensions to University High-Level Synthesis Tools: Making it Work in the Real World", Proceedings of the 28th ACM/IEEE Design Automation Conference, 1991, pp. 520-525.

87. Hallberg, Jonas - Peng, Zebo, "Synthesis under Local Timing Constraints in the CAMAD High-Level Synthesis System", Proceedings EUROMICRO '95, September 4-7, 1995.

88. Petru, Eles - Kuchcinski, Krzysztof - Peng, Zebo - Doboli, Alexa, "Post-Synthesis Back-Annotation of Timing Information in Behavioral VHDL", Proceedings EUROMICRO '95, September 4-7, 1995.

89. Gutberlet, P. - Rosenstiel, W., "Interface Specification and Synthesis for VHDL Processes", Proceedings EuroDAC '93, 1993, pp. 152-157.

90. Gutberlet, P. - Rosenstiel, W., "Timing Preserving Interface Transformations for the Synthesis of Behavioural VHDL", Proceedings EuroDAC '94, Session V-08, 1994.

91. Nijhar, T.P.K., "Source Code Optimisation in a High Level Synthesis System", PhD Thesis, University of Southampton, April 1997.

92. "Synergy VHDL Synthesizer and Optimizer Tutorial", Cadence Design Systems, Version 2.2, June 1995.

93. "Warp User's Guide", Cypress Semiconductor, April 1996.

94. Aho, Alfred V. - Sethi, Ravi - Ullman, Jeffrey D., "Compilers - Principles, Techniques and Tools", Addison-Wesley Publishing Company, 1986, ISBN 0-201-10194-7.

95. Allerton, David J. - Currie, Andrew J., "Scholar: Another Approach To Silicon Compilation", IEEE International Conference on Computer-Aided Design, Ch. 97, 1984, pp. 119-121.

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

97. Morison, J.D. - Peeling, N.E. - Whiting, E.V., "Sequential Programming Extensions to ELLA, with Automatic Transformation to Structure", Proceedings of the International Conference on Computer Design (ICCD) 1987, October 1987, pp. 571-576.

98. Allerton, D.J. - Batt, D.A. - Currie, A.J., "A VLSI Design Language Incorporating Self-Timed Concurrent Processes", European Conference on Electronic Design Automation (EDA84), Vol. 232, Ch. 44, No. 232, 1984, pp. 199-203.

99. Baker, K.R., "Final Report: Application Specific Synthesis Enforcing Testability (ASSET)", University of Southampton, October 1995.

100. Rajanala, Arun - Tyagi, Akhilesh, "An Area Estimation Technique for Module Generation", Proceedings of the 1990 IEEE International Conference on Computer Design: VLSI in Computers and Microprocessors, pp. 459-462.

101. Tyagi, Akhilesh, "An Algebraic Model for Design Space with Applications to Function Module Generation", EDAC Proceedings of the European ACM/IEEE Design Automation Conference, 1990, pp. 114-118.

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

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

104. Kirkpatrick, Scott, "Optimization by Simulated Annealing: Quantitive Studies", Journal of Statistical Physics, Vol. 34, Nos. 5/6, 1984, pp. 975-986.

105. Metropolis, N. - Rosenbluth, A. - Teller, A. - Teller, E., "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics, Vol. 21, 1087, 1953.

106. Panda P.R. - Dutt N., "1995 High Level Synthesis Design Repository", University of California, Irvine, February 1995.

107. Vemuri, Ranga - Roy, Jay - Mamtora, Paddy - Kumar, Nand, "Benchmarks for High Level Synthesis", Laboratory for Digital Design Environments, University of Cincinnati, 1991.

108. Brown, A.D. - Baker, K.R. - Williams, A.C., "On-Line Testing of Statically and Dynamically Scheduled Synthesized Systems", IEEE Transactions on Computer-Aided Design, Vol. 16, No. 1, January 1997, pp. 47-57.

109. Williams, A.C. - Brown, A.D. - Zwolinski, M.Z. - Baker, K.R., "On-Line Test of Synthesized Designs Exploiting Latency Analysis", In Preparation.

110. Baidas, Zaher, "Review Report: A High Level Floating-Point Synthesis Library", PhD/MPhil review report, University of Southampton, 4th April 1997.

111. Baker, Keith R., "Writing Behavioural VHDL for MOODS Synthesis - User Manual v1.xx", University of Southampton, July 1993.

112. Cooley, James W. - Tukey, John W., "An Algorithm for the Machine Calculation of Complex Fourier Series", Mathematics of Computation, April 1965, Vol. 19, pp. 297-301.

113. Cochran, William T. - Cooley, James W. - Favin, David L. - Helms, Howard D. - Kaenel, Reginald A. - Lang, William W. - Maling, George C. - Nelson, David E. - Rader, Charles M. - Welch, Peter D., "What is the Fast Fourier Transform ?", IEEE Transactions on Audio and Electroacoustics, Vol. AU-15, No. 2, June 1967, pp. 45-55.

114. "What is LPM ? - Efficient Technology Independent Design Using Library of Parameterized Modules Standard", http://www.edif.org/edif/lpmweb/intro.htm.

115. Press, William H. - Flannery, Brian P. - Teukolsky, Saul A. - Vetterling, William T., "Numerical Recipes in C (First Edition)", Cambridge University Press 1988, ISBN 052135465X.

116. Hennessy, John L. - Patterson, David A., "Computer Architecture - A Quantitative Approach", Morgan Kaufmann 1990, ISBN 1558600698.

117. Swartzlander, Earl E. [editor], "Computer Arithmetic", Dowden, Hutchinson & Ross Inc. 1980, ISBN 0879333502.

118. Weste, Neil H.E. - Eshraghian, Kamran, "Principles of CMOS VLSI Design - A Systems Perspective", Addison-Wesley 1985, ISBN 0201082225.

119. Baker, Keith R., "The MOODS Synthesis System - User Manual v2.xx", University of Southampton, July 1993.