UNIVERSITY OF SOUTHAMPTON

DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Dynamic Memory Allocation within a Behavioural Synthesis System

Daniel J. D. Milton

January, 2002

A thesis submitted for the title of
Doctor of Philosophy

UNIVERSITY OF SOUTHAMPTON

Dynamic Memory Allocation within a Behavioural Synthesis System

by

Daniel James David Milton

A thesis submitted for the degree of
Doctor of Philosophy.

Department of Electronics and Computer Science,
University of Southampton

January, 2002

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

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

Doctor of Philosophy

Dynamic Memory Allocation within a Behavioural Synthesis System

by Daniel James David Milton

MOODS (Multiple Objective Optimisation in Data and control path Synthesis) is a behavioural synthesis system tool that converts behavioural descriptions of users' digital designs into synchronous structural representations. This thesis describes an enhancement to the original MOODS system that allows direct conversion of dynamic memory constructs within the source language into a fully structural design with run-time memory representation.

VHDL is used as the source language for design descriptions, and is capable of directly describing dynamic memory structures in the form of explicitly created structures as well as the more implicit dynamic memory requirements of procedural recursion. The VHDL compiler required extensive modification to handle the increased subset of the language at the behavioural level.

The conversion of explicit structure allocation requires a run-time system that is capable of storing the data represented by the dynamically allocated structures. This system is realised by a behavioural description of a heap management algorithm that is both space and speed efficient and interfaces with the users' designs directly via an automatically generated interface.

Procedural recursion is now synthesisable by MOODS from the inclusion of a dynamically modified call stack created again, automatically within the users' designs, which contains the storage for local variables and passed parameters declared within the subprograms.

Finally, two demonstration systems have been designed and synthesised with the enhanced system, with both designs displaying the use of dynamic memory allocation and the second design showing the use of procedural recursion within a synthesised hardware system.

Contents

Figures

Tables

Acknowledgements

I would like to thank a number of people, with whose help and support this project was completed.

Firstly, I would like to thank my supervisor, Professor Andrew Brown. His persistent encouragement was of great importance for the completion of this thesis, along with his guidance and advice at all stages of the process.

I would also like to thank Dr. Alan Williams for his invaluable help in understanding the internals of MOODS as well as the frequent modifications made to 'modules' along the way.

Thanks also go to Andy Rushton, for the numerous discussions made over many a coffee break on the subject of compilers and the vagaries and inabilities of VHDL.

Finally, thanks to all others who helped, within the Electronic Systems Design Group at the University of Southampton and at LME Design Automation.

Chapter 1

Introduction

Dynamic memory allocation is the term given to the allocation of storage space for objects created and destroyed at run-time. The term, 'object' is used here to encompass anything that is dynamically allocated, not as a reference to any object orientated features of a language. As the number of objects is unknown at compile-time, a run-time system is required that will provide the storage space for any required objects. This system (the memory controller) will generally have a fixed memory space from which to allocate. The method for determining the position of allocated objects within the available memory space is determined by the *allocation algorithm*. The memory space is known as a *heap*. The physical realisation of the allocation algorithm is the heap management system, which is responsible for mapping all allocated objects into the available heap memory space in an efficient and fast manner. The compiled translation of the user's design communicates directly with the heap manager via a direct interface generated by compilation.

Another form of dynamic memory allocation is implied by *procedural recursion*, where a procedure can call itself from within its own body. This is useful for recursive subdivision of problems and for parsing data structures with recursive links. Procedural recursion requires that local variables and the procedural interface have instance-local storage because the procedures are dynamically re-entrant. A memory stack is generally used to store this information due to its close mapping with the type of information being stored, where only the memory space at the head of the stack is used at any one time: this represents the current instance of the executing procedure.

Behavioural synthesis of a digital design takes behavioural description of the design and translates this into an optimised structural description of the same design. The design is described behaviourally, which determines what a design *does*, not how it is *implemented*.

The behavioural synthesis tool is concerned with producing the implementation details for the behaviourally described design.

MOODS (Multiple Objective Optimisation in Data and control path Synthesis) [1,2,3] is such a behavioural synthesis tool. The tool takes as input behavioural descriptions of users' digital designs using the standard language, VHDL (Very High Speed Integrated Circuit Hardware Description Language) [4,5]. The optimisation process of the structural representation is performed in an iterative manner from an initial naïve direct translation. The structural design is created as a data path network that is controlled by a single synchronously clocked state machine. The optimisation process is concerned with the mapping of operations into control states of the state machine (where delay can be reduced) and with the sharing of data path units (where the design area can be reduced).

This thesis describes an enhancement to the original MOODS system that allows direct conversion of dynamic memory constructs [6] defined in the VHDL language, which raises the level of language abstraction that is described as behavioural to include these constructs. In particular, the modifications made to the VHDL compiler are described, along with the creation of the run-time systems of the heap manager and the recursion stack within the structural output of the tool.

The addition of support for explicit dynamic memory allocation and procedural recursion increases the number of operations available from behavioural synthesis and raises the abstraction level further into a software-like design description. This pushing of the borders between the abstraction levels for a hardware description and a software program (that could use the underlying generated hardware) gives some overlapping of available operations in both paradigms. For instance, in a co-design system, with automated partitioning between hardware and software, the partitioning tool would have more trade-offs available for dynamic memory use, due to the overlapping abstraction levels.

This thesis is divided into seven chapters. Chapter 2 provides a general overview of dynamic memory allocation as used within software environments, along with an introduction into behavioural synthesis methods and tools. A description of other implementation methods used for dynamic memory control in two third party synthesis systems is also given.

Chapter 3 describes the MOODS synthesis system in the state before any modifications for dynamic memory allocation were made, and finishes with an overview of the changes required for dynamic memory.

The creation of the heap management system and the modifications to the VHDL compiler are described in Chapter 4. This allows the explicit allocation of objects by a user's design.

The creation of the subprogram stack, which enables procedural recursion is described in Chapter 5. These modifications allow the implicit allocation of objects made by each subprogram call.

Chapter 6 describes the development of a general purpose FPGA prototyping board with bias towards the underlying storage requirements of the heap. Two demonstration designs are also introduced, which show the use and power of the new techniques. Some measured comparison results are also gained from an implementation of a small language parser design.

Finally, Chapter 7 concludes the thesis with a number of suggested enhancements and modifications, giving scope for further work.

A number of appendices are also provided, where Appendix A describes a number of collateral projects used within the modified system. Appendix B contains a paper given at the Forum on Design Languages Conference, 2000. Appendix C contains detailed descriptions of the demonstration designs produced within the scope of the project. Finally, Appendix D details the modified file format descriptions used internally by the MOODS synthesis process.

Chapter 2

Behavioural synthesis and dynamic memory

This chapter describes the background material used in the research project. Section 2.1 gives a general overview of behavioural synthesis. Section 2.2 then discusses the use of different languages that can be used as input for behavioural synthesis. Then an overview of dynamic memory allocation is given in Section 2.3, with most information gained from the software domain. Finally, Section 2.4 describes the current state of dynamic memory integration within synthesis, with two examples that have some mechanism for dynamic memory allocation built into the synthesis stream.

2.1 Behavioural synthesis

A digital design can be described with any number of levels of detail, sometimes called *abstraction levels* [7]. The process of synthesis is concerned with the conversion of a high-level abstract description into a lower-level description. Table 2.1 describes the various abstraction levels used in the design and evolution of a digital design, with the emphasis on the structural representation domain.

Behavioural synthesis [8,9] is the process of converting a design given in the behavioural, algorithmic representation into an RTL and/or structural representation of the same design, where the generated output feeds further lower-level synthesis systems in order to generate a physical hardware system. The benefit of behavioural description is that the high abstraction level enables the user to describe a system in terms of '*what it does*', rather than '*how it does it*'. Behavioural synthesis is the process of generating an optimised architecture that describes how a system works, where the synthesis tool rather than the system designer makes a large number of architectural trade-offs given a number of input constraints, such as maximum area, delay and power [10] values. The use of behavioural

synthesis is discussed in [11], along with a discussion of the benefits and drawbacks of using a higher-level abstraction level, including the possible use of memory allocation in system-level descriptions.

Abstraction Level	**Description**
System	A system is described as a number of high-level components, such as processors, memory, buses and other subsystems, partitioned and linked together to form the global design.
Behavioural	A behavioural level description is used to describe the functionality of a subsystem without giving any implementation details. The subsystem is described in terms of algorithms and operation sequences, contained within any number of concurrent blocks. A full system could be described by a single behavioural subsystem or by a number of subsystems.
RTL	A register transfer level description is used to describe the same subsystem in terms of abstract registers and combinational transfer logic in the form of Boolean equations and mathematical operators. An RTL design description faces more language constraints than a behavioural description, particularly with restrictions placed upon process timing.
Structural	A structural representation of the same subsystem describes the system in the form of a linked structure of concurrent units, where each unit describes a low-level cell device, such as a register, functional operator unit (adder, multiplier, comparator etc.) or combinational interconnection unit (multiplexor). This is a lower level subset of an RTL description, effectively forming a netlist.
Device	Each cell used by a structural description can be mapped onto a physical device, described by the linkage of transistors, capacitors and resistors. This is the lowest abstraction level, with some final implementations seen by the user only requiring a description of the black-box behaviour of the device.

Table 2.1 Abstraction level in the structural domain

2.1.1 Design flow

A typical behavioural synthesis system consists of a number of phases within the design flow [12], each performing a different construction task. This design flow can be seen in Figure 2.1. Though the flow is shown as a number of separate phases, different synthesis systems may perform a number of these phases concurrently.

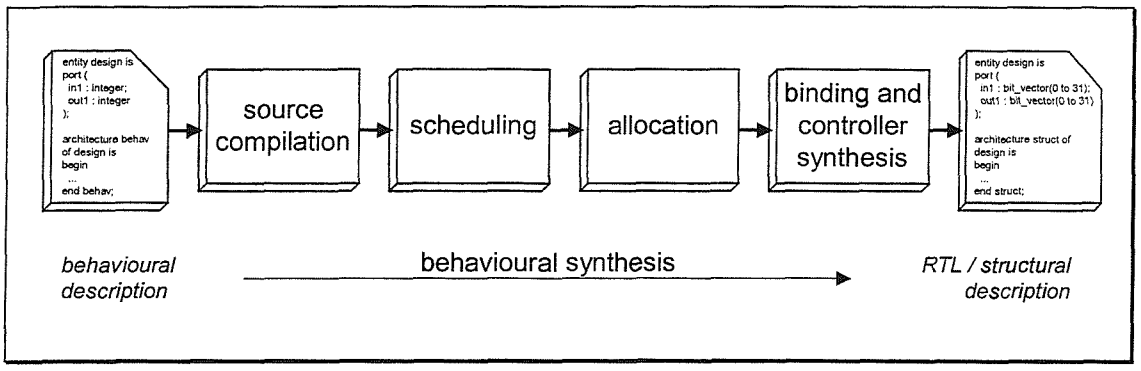


Figure 2.1 Design flow of a generic behavioural synthesis system

The first phase is concerned with *compiling* the behavioural description from the source language format into an internal instruction-based representation, and a number of compile-time optimisations (loop unrolling, procedural inlining) may be performed during this process. The result of compilation is a design specified in terms of a number of simple instructions, similar to a software assembly language representation, often contained in some form of *instruction flow graph*, with both the operations on design data and the flow of the sequential instructions being represented within the graph. This graph still contains only behavioural information, and no structural.

The next three phases are concerned with translating the behaviour into structure, and form the core of a behavioural synthesis system. The synthesis optimisation process is either performed during the construction of the data structures or during an iterative refinement process after the initial data structures are created, or perhaps both methods are utilised. A number of different data structure styles can be used, including the *Extended Timed Petri-Net* (ETPN) representation [13], which separates the control flow from the data flow into two data structures with cross-links, or the *Control Data Flow Graph* (CDFG) representation [14,15], which contains the structure in a composite graph, representing blocks of data dependent instructions within a subgraph; the *Data Flow Graph* (DFG) [14], contained by conditional control bounds such as loops and conditional expressions within the parent graph.

The *scheduling* phase determines the time-step at which every compiled instruction is executed by the final sequencing controller, usually implemented by a single-clocked Finite State Machine (FSM). There is scope for multiple instructions being scheduled in the same time-step (control state of the FSM). One general goal is to reduce the number of different control states to a minimum, which speeds execution. The *allocation* phase

assigns the available data path resources used for the execution of instructions that act upon the data flow. For instance, more than one add-instruction may be executed by a single adder resource, which shows a second general goal to reduce the number of data path units, which reduces the area of the synthesised design. The scheduling and allocation phases determine the balance between reducing the number of clock cycles (or control states) that a design requires for execution versus the resource sharing that can occur if operations are performed by different control states. If scheduling is performed before allocation, the scope for operator sharing can be considerably impaired. The trade-off made between the two goals of area and delay minimisation can produce a number of implementations that form the achievable design space, as seen in Figure 2.2.

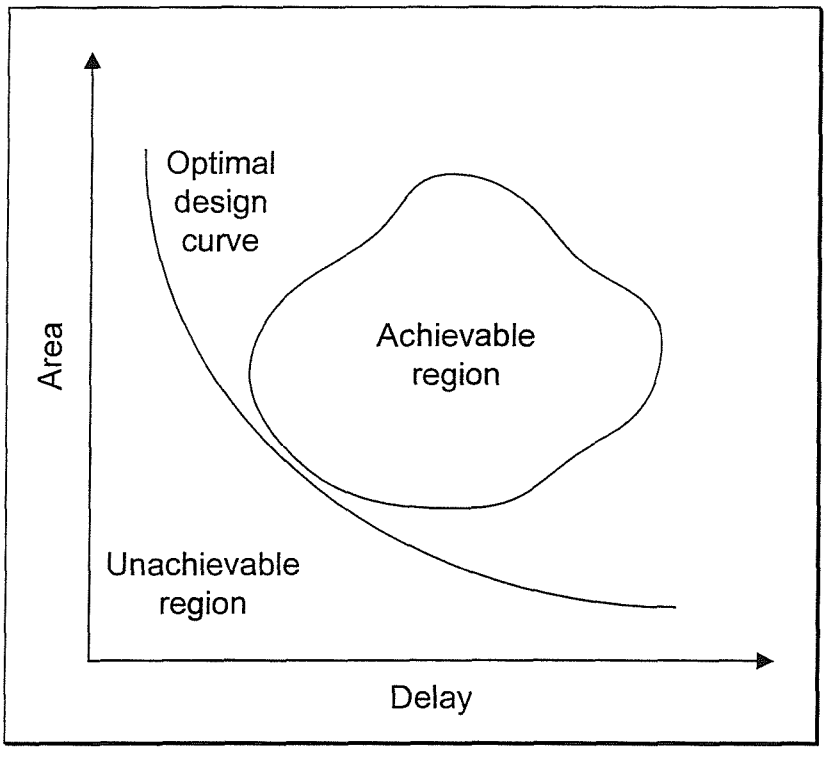


Figure 2.2 Two dimensional design space

Scheduling determines how many states are allocated within the state machine. Scheduling is affected by both resource constraints, specified by a given area or maximum number of types of functional units, or time constraints, where operations must be scheduled within a certain number of clock cycles. Various constructive scheduling techniques exist, including an *As Soon As Possible* (ASAP) [7], *As Late As Possible* (ALAP) [7], *force directed* [16] or *list scheduling* [7]. These constructive techniques do not allow backtracking.

Allocation determines the assignment of data variables and instructions into a number of storage units that are used to store the data over multiple clock cycles (registers, counters, RAMs) and functional units that perform the operations required by the instructions (adders, multipliers). If the instructions executed by a shared functional unit perform different operations, then multi-function units can be allocated in these cases. Various algorithms for data path allocation exist, including *clique partitioning* [14] and the *left-edge algorithm* [17].

Binding is the process of selecting a particular *instance* of a type of data path unit from a list of alternatives, dependent on the physical attributes of the unit to be bound and the given user constraints. It is in the final binding phase that the controlling FSM is built from a number of physical components also, or described in a more abstract manner for use by a further RTL synthesis stage. The top-down approach of behavioural synthesis may now join the bottom-up approach of module generation, which can be used to create the actual data path units from a set of parameterisable descriptions.

2.2 Languages

The reason that language is an issue is due to the variable ability of different languages to represent dynamic data structures and program flow, with some languages more suited to this than others. Each language is designed with a particular abstraction level or range of levels in mind, where some languages are not capable of handling the higher description levels. Another reason for discussion is the new trend for trying to create a unified language that can be used for both hardware description and software description. Such languages can be described as *system description languages* (SDLs).

There is a need for a single unified language that is easy to parse and understand, copes with both concurrency and components with a good library control mechanism. However, the porting of existing designs can be seen as a drawback to the introduction of a new language. In these cases, the language will be seen as an additional unwanted burden. This means that only languages that can cope with the multiple abstraction levels of hardware and system description with a non-verbose syntax that is easy to understand and port from existing HDLs will gain acceptance.

The traditional hardware description language of VHDL is discussed first, giving both its merits and drawbacks for system description using dynamic memory. Then, the C/C++ language is briefly discussed in terms of its hardware description abilities.

2.2.1 VHDL

VHDL [4,5] is the traditional Hardware Description Language used within system design for simulation and synthesis [18]. It allows for highly structured design with the language containing library management constructs as part of the syntax and semantics. The language is designed firstly for the simulation environment, with synthesis use being introduced later, with the introduction of RTL synthesis [19] tools first, then with migration into behavioural synthesis tools.

The language is designed to describe a system at all abstraction levels from the device level up to the system level. The reason for its segregation in the hardware description environment is due to the language requiring simulation in order to execute in a computing environment. This requires a simulation tool [20], which usually has a relatively large financial cost when compared to software development environments. VHDL is not designed for the description of software in an efficient manner. The language is also unnecessarily verbose, being based upon the equally verbose ADA language [21].

However, VHDL is still the best single language for the description of hardware in the synthesis environment, even with its limitations, and is why VHDL is still used by the MOODS synthesis system described in the body of this thesis. VHDL is capable of describing concurrent blocks of sequential code, where the sequential element describes the behaviour of the concurrent block at any abstraction level. Each design can be encapsulated by a library definition of its interface, which highlights the ability of VHDL to describe a system in terms of a set of modular concurrent components. Sequential blocks such as procedures and functions can also be placed into a VHDL library, enabling library storage of algorithmic descriptions also.

The sequential code also allows dynamic memory operations of explicit object creation, along with the more implicit procedural recursion, which also requires dynamic memory storage. This dynamic memory element is built into the language. The type restrictions of the language however, do not facilitate the easy creation of generic dynamic data

structures, which are frequently used within an algorithmic software environment. The reason for this is the lack of templates, type casting or void pointers in VHDL, any of which would allow generic data structures to be built. Instead, localised data structures require definition at the point of use, which reflects on the verbosity of the language.

2.2.2 Extended C or C++

The C/C++ language has up to this point been used in the generation of software. It has also been used for the limited testing of hardware design algorithms, where designs are first described and then refined in the C/C++ language. This enables a fast turnaround for the evaluation of potential algorithms due to the well-established software debugging and verification tools found in most compilation software environments. This use of the language is pure, without modification or extension for HDL descriptions, as once a design is verified, it is ported into a traditional HDL such as VHDL for further synthesis and timing evaluation.

The use of the language in the initial stages of design has prompted a number of methods for direct synthesis from the language. However, to describe hardware fully, a number of modifications to the standard language are required, namely in order to describe concurrency, more varied extendable hardware types, process communication, timing constructs and interface definitions [22]. These constructs are all part of the language specification of a traditional HDL, but are found lacking in most software languages.

One method that can be applied is the use of a directly modified C standard to include new keywords within the syntax and changes to the semantics for these new structures. This is the approach taken in [23], which describes a behavioural synthesis system that uses a directly modified C language called BDL (Behavioural Description Language). The language is optimised for behavioural or RTL descriptions. The drawback of a modified standard language is that the standard language compilers do not compile the new standard, which negates a lot of the benefits of description in this manner. It also ties the user into a particular synthesis/compilation environment, with yet another language to understand. The synthesis tool in this case does not support any pointer use, dynamic memory management or recursion.

An alternative approach is to use the extensibility of the C++ language in the generation of a number of class objects that can describe the extra components required by a concurrent, timing critical system description, such as concurrent processes and signal definitions. This allows all standard language compilers and verification environments to compile the HDL directly, which enables fast system simulation (without the need for a simulator), integrated debugging and statistical verification. This is the approach taken in [24], which describes a set of C++ classes, globally called Scenic. This specification has migrated into the public domain, now being known as SystemC [25,26]. The class libraries are capable of describing an RTL system level up to a full system specification, including behavioural descriptions. The classes form a wrapper around the standard language constructs, providing a runtime environment with concurrency and data communication.

Another benefit of using the C++ language is the ability to describe polymorphic data types, where the types used can be interchanged without modification to the underlying code that uses the type. Better specification of generic abstract data types is also possible in C/C++, which enables better modularisation. However, current implementations of synthesis systems that use this form of the C/C++ language do not support pointers, dynamic memory management or recursion within the synthesisable subset of the language. This limitation is due to the synthesis tool, not the software verification environment.

Two synthesis environments that use C/C++ descriptions with limited dynamic memory support are described in Sections 2.4.1 and 2.4.2.

2.3 Memory allocation overview

Memory allocation describes how a system assigns storage to the translation of source language input, both in terms of implicit system structure and of user data. User data memory allocation comes in many forms, both implied by the language and explicitly referenced by the design. Storage requirements will either be statically created during compilation or dynamically grow and shrink with the execution of the design. The storage requirements are also dependent on the methods used to translate the given language.

For instance, in a software environment, every aspect of the translated design will eventually be stored by some form of memory, from the storage of the translated program

instructions requiring a fixed amount of static space, through the static allocation of global program variables, then with a dynamic stack block used for storage of local variables and parameters used by subprograms and finally with an explicit dynamic heap block used for storage of explicitly created dynamic objects. This is due to software languages generally being translated for use in conventional von Neumann architectures.

In the behavioural synthesis environment, the 'program' itself does not require memory storage, as the design is translated into a static structure of low-level hardware components (this is not strictly true now, due to the introduction of FPGA programmable devices that require the hardware configuration to be stored in a large ROM, and once configured, the 'hardware' is actually built from a number of configurable SRAM-based logic blocks - however, this structure can still conceptually be considered as static hardware). The data within this structure does however require some form of storage. This kind of system separates the storage requirements of the design from the storage requirements of the data within the design, unlike the von Neumann processor targeted software.

Typical static creation of memory in a hardware design relates to the allocation of static registers to store variables or signals [17]. An extension of register allocation is with the generation of counter variables, where the storage element itself is used to perform operations on the data contained within it. Static multi-dimensional arrays also have a direct translation into fast indexed SRAM memories. Behavioural synthesis could also assign groups of single static variables into memory blocks for storage efficiency reasons [27]. The creation of pipelined functional units also requires implied register storage at the end of each pipeline stage [28]. Each of these memory requirements is statically created by the compilation or synthesis stages of a behavioural synthesis tool.

The dynamic memory constructs in a source language require a completely different method of storage. Dynamic memory is used extensively in software development due to the data abstraction that is possible with its use. The underlying storage mechanism is still based upon the same types of index-addressed memory, but the interface into that memory requires runtime systems in order to manage the allocation of dynamic objects within the available data space. Dynamic memory has had little use in the synthesis environment due to the inherently static nature of hardware description, but with the raising of the abstraction level of design description away from a 'direct' translation of the source description comes the increased desire for the use of abstract dynamically allocated

runtime data types. For this reason, most literature focuses on the software applications of dynamic memory, although most have equal applicability in the hardware domain.

Two mechanisms are generally used for the allocation of dynamic memory in a software environment. A stack is used by the procedure call mechanism and a heap is used for explicitly created objects. Both are formed from controlled data structures.

2.3.1 Stack allocation

Software subprograms in the form of procedures or functions are implemented by a separate instruction list within an area of program memory. Most languages support re-entrant subprograms, where the local data controlled by the subprogram is replicated on every instance of the subprogram being called, so that any one instance does not overwrite the data contained by any other instance. The need for re-entrant subprograms is two-fold: The first is that a concurrent runtime environment could be in use, which allows the same procedure to be called from different threads at the same time (interweaved by context switches), and the second reason derives from procedural recursion, where one instance of a procedure can call another instance of the same procedure either directly or indirectly via other recursive procedures. Each instance of the procedure requires a new set of local variables - this is where the stack is used.

A stack is formed from a very simple data structure of a large contiguous block of memory accessed from a current index position, indicating the current data set being accessed by a subprogram. The stack is of fixed length and can occupy a shared underlying memory system. The stack can only be accessed from its head, with procedure calls allocating enough local memory space from the stack by incrementing the stack pointer by a number of words. A return from a procedure decrements the stack pointer by the same number of memory words as the original increment, leaving the memory above the head of the stack containing free data. The stack control is formed as part of the compiled code.

The stack mechanism is so widely used in software descriptions that most implementing microprocessors have special instructions for stack supporting operations, for example the '*push*', '*pop*', '*call*' and '*ret*' instructions (shown in assembly code mnemonics), which are used for insertion and removal of stack data and by the procedure calling mechanism.

Advanced circular register windows are also used by certain microprocessors to speed the memory access time of the stack [29].

The data stored in a typical stack mechanism relates to three data sections of a subprogram. The local variables are the first, with passed I/O parameters being the second (really a subset of the first). The procedure calling mechanism also requires stack storage for the return address of the position to jump back into once the subprogram completes. The stack is the natural place to store this information, due to the instance-local requirements. The storage of the return address on the stack provides the final mechanism required for procedural recursion.

2.3.2 Heap allocation

As well as dynamic data that follows the program procedure calling mechanism, dynamic data can be allocated explicitly from any position within the program, used after that point and deallocated at any position after allocation. In many languages, the method used to reference the allocated data is by a single base address that references a contiguous block of memory space that the object can use. The number of words that the allocation space for an object requires may be determined during compilation, or even may be determined at runtime.

Allocation of dynamic objects is performed by an *allocator*, which is directly accessible by the source language, and potentially hidden from the user. A *deallocator* performs deallocation of dynamic objects similarly. The data that represents the created object is accessed in a way that is dependent on the type of object created. The values in an array object are accessed by an offset index value; whereas a record element is accessed by a constant element offset value defined by the compiler.

The allocator requires a given memory object size, provided as a count of bytes or words and returns a reference to the memory allocated for the object. The deallocator requires only the reference to the object to be given, with the underlying mechanism able to determine the object size from the containing heap data structure and the objects position in memory. The reason for this is that a compile-time object size may not be able to be calculated by the deallocation call due to the ability to define the size of an allocated

object at runtime, at the point of allocation. For this reason, the size of the returned objects must be stored by the heap mechanism.

Due to the very abstract nature of supplying storage for an object from an available data space, a number of heap allocation mechanisms exist, with various trade-offs made with respect to allocation strategy and policy [30]. Each mechanism effectively uses a different underlying data structure and method for selecting blocks of memory to return via the allocator. The goal is to reduce wasted memory space and the time for each allocation.

The overall strategy should be able to exploit the regularities in the memory allocation request stream, with the policy determining the implementable decision procedure for placing blocks in memory. The mechanism forms the set of algorithms and data structures that implement the policy. The memory allocation 'algorithm' usually refers to the underlying mechanism for memory allocation, partly because the only point at which memory management occurs is during the allocator and deallocator interface execution.

2.3.2.1 Methods

The data structures used by a memory management scheme are usually built from a number of *header fields* (Figure 2.3a), stored in the same memory space as the allocated data. The information stored within the headers relate to the sizes of objects being created and links to other header structures, which form the containing data structures. Tree-type data structures can be formed with the use of header and footer structures, which support memory block splitting and coalescing [31]. Another useful structure used by various allocation mechanisms is the *free list* (Figure 2.3b). These utilise the same data space as all allocated data, where the list does not effectively consume any memory space, due to the free list structure being formed within the free memory blocks themselves.

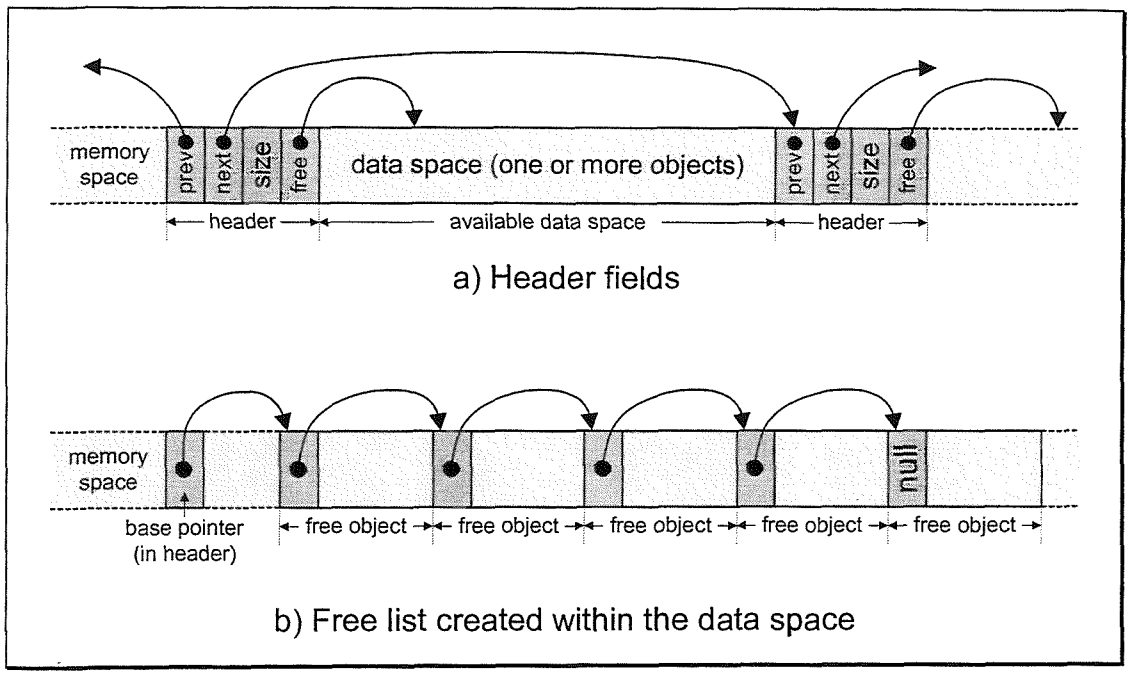


Figure 2.3 Heap management data structures

The heap management algorithm could effectively be asked for any size of object, and expected to return a block of contiguous memory large enough for the contained object. However, the probability of the different object sizes being used is dependent on the application that uses the heap management system. If an allocator can optimise itself dependent upon the memory requirements of a particular application [32,33,34], then this could speed the allocation requests and minimise memory wastage. A more general case than this suggests that the treatment of smaller objects should be different to the treatment of larger block objects, as the bulk of all allocation requests are for small objects. A mechanism that exploits this fact may perform almost, or just as well as an optimised allocator.

There are five main types of documented allocation mechanisms: sequential fits, segregated free lists, buddy systems, indexed fits and bitmapped fits. Each describes a basic mechanism, with various real-life allocators using parts of each mechanism type, with conglomerate allocators using a mix of mechanisms (example measurements in [35]). A brief mention of all basic mechanisms is given below.

1. The 'sequential fits' mechanisms are based upon a single linear list of free blocks. The first-fit mechanism [36] allocates an object by searching the free list from the start point, returning the first block large enough to store the required data. If the

block is larger than the returned object, then it is split into the returned object block and the rest of the free space, which is re-inserted into the list of free blocks. The next-fit mechanism is a derivation on this concept, except that the starting point for the search is the last checked free block. A best-fit mechanism searches the entire free list to find the smallest free block large enough to satisfy the request, so this does not scale well in larger memory systems.

2. The 'segregated free lists' mechanisms are based upon an array of free lists that contain objects of particular sizes. Objects are returned from the first free list that is capable of holding the required sized object. If an exact match is not available, then a larger object contained by alternative free lists is returned, with some wasted space. A derivation of this mechanism is used within this thesis, based upon [37,38], where each free list is defined within a separate memory page, allocated at runtime to store objects of particular sizes.

3. The 'buddy systems' mechanisms use splitting and coalescing of memory blocks into pairs, where these pairs can be of equal size: binary buddies, or of different ratios: fibonacci buddies, weighted buddies. Blocks are split until the correct memory size can be returned via the allocator. This system uses a binary tree storage data structure. Another variant of the buddy system is the double buddy system, where two binary buddies are used, with different sized base objects. This enables closer matching of the required object size to the returned memory block.

4. The 'indexed fits' mechanisms use structured indexes to implement a desired fit policy. This is really a container for multiple fit strategies, which use a number of different data structures to speed allocation searches.

5. The 'bitmapped fits' mechanism is a derivation on the 'indexed fits' mechanism. This uses a bitmap block into the entire memory space, indicating which blocks in memory are allocated and free. Fast bitmap searches are formed from the densely packed information, allowing fast allocation.

2.3.2.2 Fragmentation

There are two types of fragmentation that can occur in the allocated data space of the heap. The first is external fragmentation, where an allocation fails even when free memory is available. This can occur if the requested block size is too big for any contiguous block of free memory, or if the object size is too small to split a large free memory block in naïve mechanisms. The second type is internal fragmentation, which occurs when a returned block is larger than the required object size, resulting in wasted space within the returned allocated block. This is deemed as internal fragmentation as the waste is part of an allocated block. This situation can occur in some mechanisms due to the need to round up object sizes to the nearest power of two or closest match available.

The splitting and coalescing of free memory blocks generally combats fragmentation. These operations operate upon the free memory area only, with any space allocated for objects being immovable while allocated. Allocation can fail if there are no free blocks next to each other that can be coalesced. As the mechanism affects where the objects are created, this can affect the ability to coalesce free blocks. This could mean that some analysis of the behaviour of real programs could help with the selection of an allocation mechanism.

General analysis of various designs has highlighted three types of allocation behaviour. These behaviours are classified as follows:

1. *Ramps*: Data structures are accumulated over time, with the program solution found quickly once complete, allowing the quick destruction of the data structures.

2. *Peaks*: Bursts of allocation and then deallocation. This behaviour is seen within phased programs.

3. *Plateaus*: Data structures are built quickly and kept for a long duration until the solution is found. The data structures are then removed quickly.

Also, extra information discovered while profiling real-life memory accesses has indicated that objects that are allocated at about the same time are likely to be deallocated at about the same time. Another general observation indicates that objects of a different type (hence size) are likely to be deallocated at different times in the program flow. The

conclusion of these observations is that an allocation policy should sequentially allocate objects in adjacent positions, with segregation dependent on type (size). This exploitation of the non-random behaviour of most programs should reduce the effects of external fragmentation, with the increased chance of block coalescing.

Size segregation with enough different sizes for efficient block fitting also reduces internal fragmentation for most (small) objects.

2.3.2.3 Garbage collection

Garbage collection is an alternative method over explicit deallocation of objects in the heap. It forms an alternative to a structured design methodology that removes the need to explicitly free any dynamically created object. The memory is taken care of by the garbage collector when the user cannot reference the object any more.

The removal of reference can be due an explicit overwriting of the reference value or from the reference simply going out of scope [39]. The scope of an object is determined from local variables in the stack or from other dynamic objects that could contain references to the object. Static references of global variables or processor registers can also contain base pointers of data structures, which determine the scope of objects.

The 'mark and sweep' algorithm performs typical garbage collection [40]. This algorithm requires an entire heap object search from the set of base pointers, which could be derived from the stack variables, static data or processor registers. The algorithm receives no cooperation from the compiler [41], hence pointer ambiguity requires resolving, when determining the path of all reachable objects. The algorithm firstly clears the 'mark-array', and then works through the heap objects from the set of base pointers, marking each object that is reachable within the mark-array. After this phase, a sweep of the entire heap is performed, which removes any unmarked object from the heap data structures.

While garbage collection is active, no memory operations can be serviced, which halts all processes that use the heap. This is unacceptable in a real-time environment. Incremental collection [42] can reduce the effects of process halting, but still results in unknown memory timing behaviour. Structured programming techniques are more acceptable in a real-time environment than garbage collection.

2.4 Synthesis systems and dynamic memory

The field of behavioural synthesis has been around for many years, along with the field of dynamic memory allocation in software. It is only recently that the integration of the two fields is being attempted.

A number of behavioural synthesis systems exist, both academic and commercial. Some academic systems are: CADDY [43], Cathedral-2 [44], CAMAD [45,13], Chippe [15] and Balsa [46,47]. The major commercial systems are: Synopsys Behavioural compiler [48], Cadence Visual architect [49] and Mentor Graphics Monet [50].

None of the mentioned systems has support for procedural recursion within the synthesised designs created by the synthesis tools and no system has support for direct synthesis of explicit dynamically allocated objects. However, two systems have been created that form a layer on top of behavioural synthesis, allowing design exploration before behavioural synthesis is applied. Both these systems use Synopsys Behavioural Compiler as the behavioural synthesis tool at the back-end. The front-end system exploration and pre-optimisation of both systems support the concept of explicit dynamic memory allocation. These two systems are briefly described in the next two sections.

Other issues that have become more important in recent times have mainly been due to the mobile electronics market, where power consumption is a large factor that determines battery life. Links have been made to the use of memory [51], where power minimisation can be achieved by using a number of smaller single-port memory blocks over one large block. Memory accesses cost power, so the removal of transfer redundancy can also help. Embedded memories [52] can also help with power use, with the removal of power-hungry external ports and the widening of internal busses, reducing the number of data transfers. Memory bandwidth is also a factor of system design, where compiler optimisations and automated system synthesis may help [53].

2.4.1 SpC

SpC [54] is a tool from Stanford University that supports synthesis of standard C behavioural models, including support for pointers and data structures. The phases of the tool include memory binding into location sets, pointer analysis, dynamic memory

allocation resolution, pointer resolution, memory partitioning and conversion into a traditional HDL to complete the behavioural synthesis flow.

The first stage in the tool flow is memory partitioning. This stage is required to separate the various variables that are notionally stored by a single address space (C being a software language) into a set of independent mutually exclusive locations that can be accessed in parallel. These location sets can contain single variables, arrays, structures, arrays containing structures, structures containing arrays and dynamic memory data structures. Each location set holds a single item, where a practical implementation for hardware synthesis is sought, each location set will eventually be mapped onto a separate memory unit. In the case of arrays of structures, this could be separated into a number of arrays of each element type within the structure, allowing for better memory utilisation, where each array is contained by a different location set. Location sets with single variables can get mapped onto registers, or may not even require storage, being mapped onto wires. Location sets containing arrays can be mapped onto register banks or RAMs.

After the location sets are defined, the accessing of the data in the sets can be via any number of pointers. This is the reason for the static pointer analysis stage [55], where each pointer is resolved at compile time. The analysis determines the set of locations that a pointer could reference. The results of pointer analysis must be both safe and accurate, where a safe analysis finds all alternative pointer locations and an accurate analysis minimises the amount of logic that is generated to access the memory locations. Pointer analysis is used by the pointer resolution stage to build the accessing logic for the referenced location sets of each pointer.

Even though the SpC designers are trying to make the entire ANSI-C language synthesisable, there are limitations that are introduced during pointer analysis. The first limitation deals with a set of parallel processes, where no shared variables are allowed between processes, as static pointer analysis cannot cope with concurrent access to the same variables, unless some kind of interface is synthesised for communication between the processes. The second limitation is due to the lack of full support for subprograms in the underlying behavioural synthesis tool and the differences between the C-based subprogram and an HDL-based subprogram. The limitations imposed are that procedural recursion is not supported due to the lack of dynamic stack data. Subprograms are also usually inlined by a behavioural synthesis tool.

All dynamically allocated data is represented by a specific location set, with a heuristic used to separate different data structures within the heap allocated data into different sets. Any general pointer index is separated into two fields. A tag field is used to determine which location set is referenced by the pointer and the index field stores the index as a number of strides within a location set. A stride determines how many memory locations are required per index. Offsets from the index are used when referencing sub-structures. Limitations on the number of bits used for location set tags and indexes place upper limits on the sizes of data structures.

The support of runtime memory allocation requires an allocator [56]. This is provided in hardware-controlled form, due to the synthesis nature of the tool. Memory is managed by a number of user defined memory segments, where a segment is an array of finite size with data allocated within it by a unique hardware allocator. The memory segment may be later mapped onto one or more physical memories during synthesis. The user of the system determines how many memory segments are created and which allocations occur in which segment. The user also sets the physical size of the memory segment. Allocations are made with the use of the standard C-runtime '*malloc*' and '*free*' functions, which are translated into calls to the allocator defined for the relevant segment. The tool generates every allocator used in the different segments and communication is formed using handshakes with the main user's design.

A number of optimisations can be made with selection of the allocator used by the memory segments. There are currently three supported allocators, which allow a certain degree of tailoring. The first allocator is a general-purpose allocator that can allocate objects of any size. This uses a first-fit mechanism with direct coalescing on deallocation. The second allocator is an optimised form of the first, which performs better deallocation performance through better data structure linkage. The final allocator has a specific purpose. It is capable of allocating objects of only one size, similar to the segregated free lists mechanism. This allocator can only be used when all objects allocated within the segment are of the same size. It also borrows from the bitmapped fits mechanism to determine which objects are available in the data space.

A further optimisation may also be applied, which reduces the number of allocations and deallocations in very limited circumstances. This optimisation is introduced to cope with legacy code that may be used. The optimisation is essentially to convert a sequence of

'*malloc*' then '*free*' into a static location set. This can only be applied in a purely sequential block, such as within loop bodies or conditional bodies, with no branching between allocations. The allocated object must also be of known size during compilation, at the point of allocation.

The translations made by SpC output the design using Verilog HDL. A standard behavioural synthesis tool, Synopsys Behavioural Compiler, then performs the final synthesis stage of SpC.

The approach taken by SpC of synthesising a design directly from the standard C language overly restricts the use of the base language, originally designed for software descriptions. In this respect, too much effort is placed upon translating the software-like description methodology of C over a direct translation possible from an HDL description, such as VHDL.

2.4.2 Matisse

Matisse [57,58,59] is a system design environment that has the capability of describing systems with intensive data storage, transfer and real-time requirements. Designs are specified in a modified C++ language, which is capable of describing both software and hardware. Dynamic memory management is supported with the use of a number of abstract data types, which are mapped onto an optimised memory architecture. Traditional behavioural synthesis is performed after the Matisse system exploration. The target of system exploration is an embedded single chip solution with both hardware and software implementation sections.

The dynamic memory management phases of the design flow determine both the containing data structures for user data, the methods used to allocate the data and the custom physical mapping of the data structures to a number of distributed memories. The Abstract Data Types (ADTs) are used to contain all dynamically allocated objects. All dynamic memory management behaviour is synthesised in hardware due to the power savings made over a software implementation.

The language used as input to the system is a syntactically and semantically modified C++, which contains extra structures for the definition of concurrent tasks and

synchronisation between these tasks. The model used for system design is based upon these new constructs, where a system is defined as a set of processes that communicate with control of communication handled by synchronisation. The processes are statically created, each with its own virtual memory space. Communication is realised by global pointers. Synchronisation occurs with the use of a set of atomic functions, where these synchronisation functions may only be executed by one process at any one time.

The system design flow consists of six phases, of which Abstract Machine (AM) generation is the first. Abstract Machine generation is used to build an executable specification that can be used in simulation and profiling from the modified C++ language. This phase converts the modified C++ into standard C++ with runtime additions added for process concurrency and communication. The Dynamic Memory Management (DMM) additions are inserted after this, where the DMM phase consists of refinement of the ADTs and with the selection of the Virtual Memory Management (VMM) scheme. Process concurrency management follows; where this phase is used for concurrency extraction, thread scheduling, processor allocation and Inter-Process Communication insertion. The underlying memory subsystem is created in the Physical Memory Management (PMM) phase, where an area and power efficient distributed memory architecture is generated. The final stage is synthesis, where system software is created along with the interface to the hardware, which is synthesised from the generated behavioural description using Synopsys Behavioural Compiler.

The phases pertaining to memory management are the ADT refinement stage, Virtual Memory Management and Physical Memory Management. Each stage has some effect upon the power and area of the final design, each optimised to sustain a certain data throughput bandwidth.

All dynamic data is contained within the Abstract Data Types. The underlying data structures that implement the ADTs are built from four primitive dynamic data structures, the linked list, tree, dynamic array and dynamic pointer array. Each of these types are combined to form the more complex structures, using access keys at each layer. Refinement of which underlying data structures to use is performed by a heuristic, that sets an ordering of the refinement decisions, which generates the best combination of underlying data types. The heuristic is defined to give a power optimal structure [60].

Hashing is an underlying data structure that can be used as an extra layer when a non-uniform key distribution is expected as input to any of the basic structure types.

Virtual Memory Management is applied after the data structures are refined. This phase reserves storage space for each data type obtained by ADT refinement within memory segments, where each segment has a custom memory manager designed. Similar underlying data structures can be set to share the same memory segment at this stage, but only if the allocated data of both data structures are allocated in different phases of design execution.

Each segments' custom memory manager can be built using a number of low-level mechanisms. The mechanism type is selected from a search space of available mechanisms. There are currently three supported mechanisms [61], the state-variable mechanism, a free-list mechanism and a FIFO mechanism. All controllers are built to allocate fixed block sizes [62], which simplifies the mechanism. The state-variable mechanism keeps a state bit per object, which provides a fast bitmap lookup of allocated objects. The free-list mechanism simply pushes and pops from the head of the free lists when required. The FIFO mechanism has head and tail pointers into the segment. This is only used in FIFO communication schemes.

Physical Memory Management is used to share the virtual memory segments between a number of physical memories. A single memory is not automatically mapped, as the available cycle budget may not allow for sequential memory accesses, which frequently occurs in data intensive applications. The generated distributed memory architecture exploits parallelism in the data accesses in order to reduce the number of cycles to perform particular memory operations. The method for determining the number of used memories and sharing configurations is automated given the area and power constraints. More memories allow for a reduction in power, where power is dependent more on data transfer than on the core system power.

The automated method for memory sharing begins with the introduction of basic groups, where a segment is split into a number of separate groups. These groups are later assigned to physical memory. Scheduling of memory operations on these basic groups is then set, which determines which groups are made simultaneously accessible. Then, the physical memory assignment phase assigns the basic groups in clusters [63] to physical memories,

taking conflicting accesses into account with the assignment of multiport memories or separate memories.

Various examples of the use of Matisse are given in the literature [62,63,64,65], with all examples being partial systems used by an ATM communications network. Examples of a Segment Protocol Processor (SPP) are given in [62,64] and an Operation And Maintenance component (OAM) in [65]. Comparisons between different implementations of different designs are given in [63].

Matisse is a system design environment that performs trade-offs between a limited set of abstract data structures before behavioural synthesis. In this respect, the tool is not as general purpose as a synthesis tool supporting dynamic objects directly. The use of a software-derived non-standard input language also affects the general use of the tool.

Chapter 3

The MOODS synthesis system

The behavioural synthesis system used and modified for the dynamic memory synthesis research is called MOODS [1,2,3,66,67,68] (Multiple Objective Optimisation of Data and control path Synthesis). The system has been developed to compile a behavioural description of a digital design using behavioural level VHDL into a structural description of the same design using structural level VHDL as output [19]. The structural description then feeds a variety of third party tools for the physical design implementation.

This chapter describes the synthesis system before any additions were made for dynamic memory. Section 3.1 describes the VHDL compiler used as the front end to the system, while Section 3.2 describes the operation of the core synthesis process. Finally, Section 3.3 gives an overview of the modifications made to MOODS for the implementation of dynamic memory, giving the modified system data flow, with descriptions of the additions and modifications made to the system. Sections 3.1 and 3.2 are essentially a précis of previous development of MOODS. The material is included as necessary background.

The term MOODS refers to the entire behavioural synthesis system. However, the system is built from a two main tasks, where the second core synthesis task is also referred to as MOODS. The initial system data flow before the dynamic memory additions were made is shown in Figure 3.1. The tasks communicate via a number of intermediate files. The actions performed by these tasks are listed below.

1. The behavioural VHDL description can be provided by a number of source files. Each file is passed into the VHDL compiler, 'VHDL2IC'. The compiler builds an internal representation of the VHDL parse tree and translates this into a simpler intermediate description using simple two-input instructions. This description is created as ICODE (Intermediate CODE), which is a proprietary language-neutral design description file.

2. The single ICODE description file is then fed into the main behavioural synthesis task, along with a set of user objectives and technology libraries. An internal data structure is built that links the ICODE description into the control and data path graphs. The initial data structure contains one ICODE instruction in each control state, with the functionality of each instruction being bound to a separate data path node. The synthesis process is formed from iteratively modifying the data structures until the user objectives are met. The structural description of the design is created from a direct translation of the internal data structures. This translation is performed in the final stages of the MOODS core synthesis task.

3. The final stages of a system implementation utilise a number of third party tools, such as Synopsys Design Compiler [69], Cadence Synergy [70], Leonardo Spectrum [71] or Xilinx Foundation [72]. These take the structural VHDL description generated by MOODS as input. Each tool performs low-level logic synthesis and technology mapping, which translates the design into a physical circuit to be implemented in an ASIC or FPGA [73].

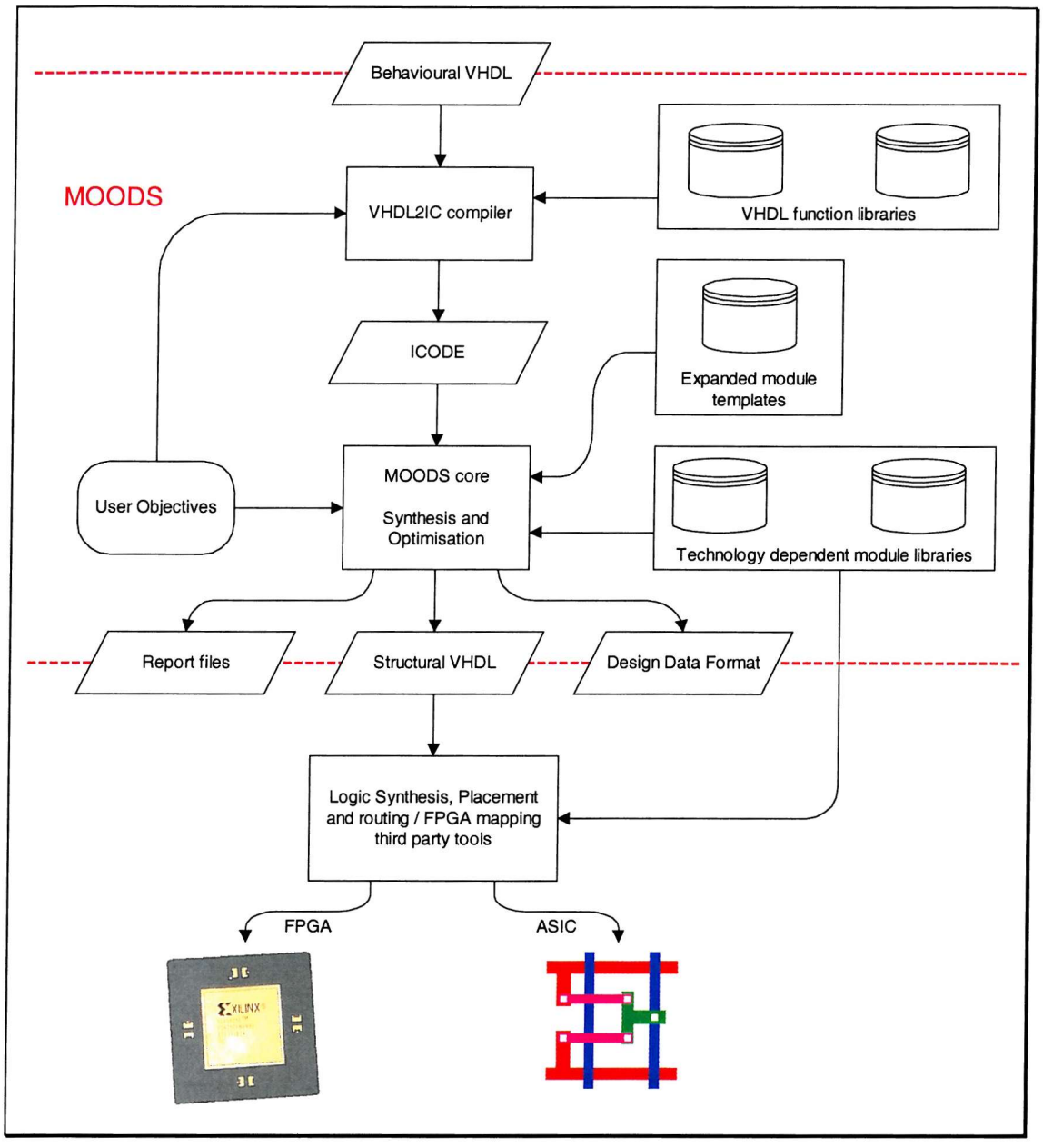


Figure 3.1 Original MOODS system data flow

The ICODE description file is used as input to the core synthesis process, as this provides a language neutral input method. This allows other languages to be incorporated into the MOODS synthesis system by creating only the compiler for them. Each language compiler would translate the source description into the proprietary ICODE format.

The VHDL function libraries are formed from a number of VHDL packages that are linked into every input description. These packages contain a number of conversion functions, type declarations and operators upon these types. The compiler uses these packages internally, forming translation optimisations when the items within the packages are used.

The expanded module templates [3,74,75] that are input to the core MOODS synthesis task are used for inline expansion of multiple instruction tasks by the synthesis process. Expanded modules are used to form alternative descriptions for complex operations that can be broken down into more area efficient multiple-iteration versions of the operation. Expanded modules are formed from a number of ICODE operations initially generated from the compiler. The ICODE operations form a submodule template [76]. Expanded modules form an extension to the standard module paradigm used by MOODS, where a module is formed from direct translation of a subprogram.

The technology dependent module cell libraries hold all information about the structural unit components that are bound to all control and data path nodes. The control path is created from a set of bound control node components that implement the controlling state machine from a one-hot token-passing architecture. The variables operated upon by the ICODE instructions can be bound to various types of memory components in the data path, including registers, counters and RAMs. The data path also contains functional units, which perform the operations described by the ICODE instructions. These are bound to combinational cells such as adders, multipliers and comparison operators. Finally, the data path contains interconnect-units, which provide the controlled data routing through a binding to multiplexor cells. The cell libraries contain physical values that describe such items as the speed and size of the library unit and the synthesis process for binding and sharing decisions uses these values. The libraries also contain RTL VHDL descriptions for every cell that is used by the third party tools. The data structures used by the MOODS core synthesis task are shown in Figure 3.2, which shows the linkage between the structures.

The design data format file is another output of the synthesis system. This file contains a readable description of the data structures used by MOODS. The file can be parsed in order to regenerate the same data structures within the synthesis core.

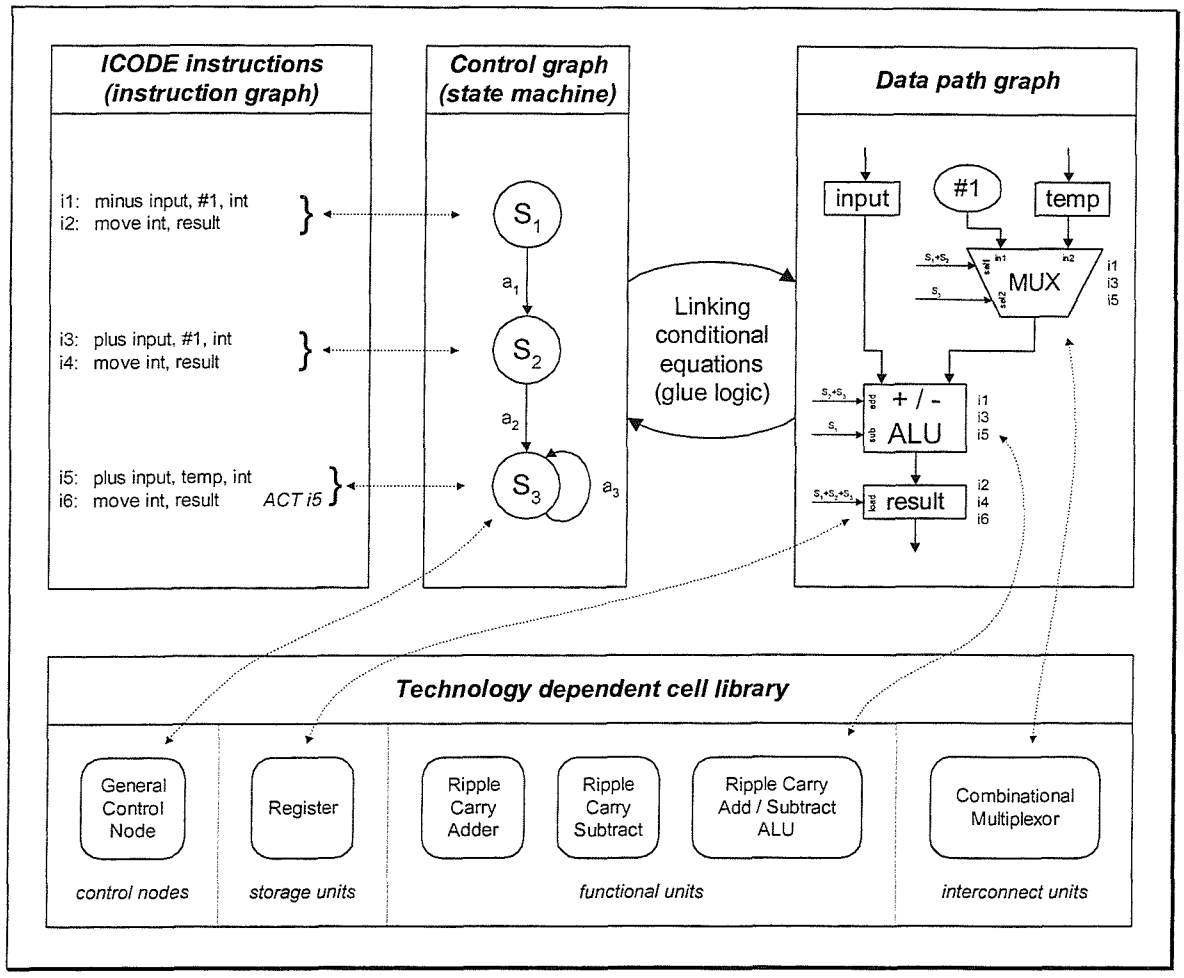


Figure 3.2 Data structures used by the MOODS synthesis task

3.1 VHDL Compiler

The VHDL compiler that forms the front-end to MOODS is designed with a number of phases, which translates the original VHDL description into another description at a lower language level. The conversion process translates a number of inputted VHDL files into a single ICODE file that is representative of the original VHDL, albeit in a form that is similar to an assembly representation of a software language [77]. Compilation, assembly and ICODE generation are all performed by a single program, which means that every source file that is input to a synthesised design requires re-compilation each time any of the input files are edited. The program flow is seen in Figure 3.3, which shows the consecutive phases that form the compilation flow.

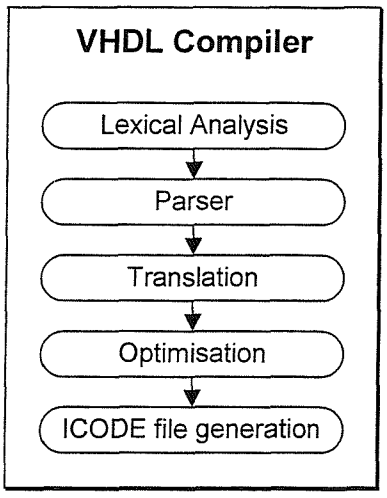


Figure 3.3 VHDL Compiler program flow

3.1.1 Synthesisable VHDL subset

Behavioural VHDL tends to use more from the sequential part of the language contained by the process construct, rather than from a number of concurrent constructs. RTL descriptions are formed more from a linkage of concurrent components, with process descriptions being limited to only one clock cycle being described by the entire process loop, with one wait statement per process iteration or the process being controlled by a sensitivity list. A behavioural description for MOODS, on the other hand, allows multiple consecutive clock cycles in a single process. The synthesis process directly controls the number of cycles used, with the sequential VHDL operations being converted into ICODE instructions that are scheduled within these cycles, under control of the generated finite state machine. Timing constraints may be placed in the VHDL source with the use of multiple wait statements. This guarantees a level of output timing adherence when communicating with external components.

The general behavioural VHDL constraints [78] placed upon the language by the synthesis tool, MOODS [79] are listed below. The limitations are placed upon a single synthesis run, with the generated VHDL output of the synthesis process able to be referenced by a structural VHDL container description for use with the third party low-level logic synthesis and technology mapping tools. The limitations are formed from both the relaxed timing model utilised for behavioural synthesis and from the difficulty in implementation certain features of the VHDL language.

1. A design is described by a single **Entity** / **Architecture** pair, where the **Entity** describes the I/O port signals passed into the design and the **Architecture** describes the actions performed by the design. Any number of VHDL **package**s may be referenced and used by the design.

2. Packages are limited to containing only constants, type declarations and subprogram declarations and definitions, with concurrent component declarations disallowed. This effectively removes the ability to build up a number of concurrent library components, while allowing sequential subprograms to be reused. The component limitation is due to the limited concurrency features allowed in the architecture body.

3. The architecture body may contain any number of concurrent processes, with component instantiation, generate statements and concurrent signal assignment operations disallowed. The contents of the process may reference any constants, types and subprograms defined within the used packages or architecture declaration.

4. Built-in support for the '*bit*' type and '*bit_vector*' **array** derivative, along with a set of operations on these types and a number of conversion functions are provided by a package that is linked into every design passed through the MOODS compiler. These types must be used if efficient (compile-time) conversion to and from integer types is required. Integer types are used as for-loop iterators and array index values. These base types may have sub-type derivatives declared and used in the same manner.

5. Composite type declarations are limited to the use of fixed-length **array** types that form 2-dimensional variables. An array of '*bit_vector*'s (itself an array of '*bit*'s) can be stored by a bank of multi-bit registers or by a RAM cell. The composite **record** type is not supported, as it is virtually useless without dynamic memory support.

6. No explicit dynamic memory support is provided, with the lack of composite **record** types, **access** types (dynamic object reference mechanism) and unconstrained **array** types. Incomplete types, used for cyclic data structure creation are not supported. Explicit object allocation and deallocation is disallowed. These restrictions are removed with the implementation described in Chapter 4.

7. Subprograms can call other subprograms. However, support for recursive subprogram calls is disallowed. This restriction is removed with the implementation of procedural recursion described in Chapter 5.

8. The I/O passed as parameters through subprogram calls is limited to values that can be held by a single multi-bit register (1-dimensional types).

9. There is support within VHDL for abstract **file** types, which are generally used to drive long sequences of test vector values in test-benches. No support for files or the underlying file system is provided during synthesis.

10. Floating-point number support was in the process of being integrated with MOODS during the initial stages of dynamic memory support. The compiler did not support floating-point numbers at the beginning of this research.

11. The synthesis process ignores **assert** statements. These statements are used by simulation to provide feedback on abnormal situations or to provide messages about the state of the simulation. There is no meaningful translation for synthesis.

12. The sequential operations contained by a process are simulated within zero simulation time (delta-time), with **wait**-statements defining timing breaks. The synthesised design can take a number of clock cycles to perform the same operations. Reliance on relative timing for communication between processes is therefore not guaranteed to work. Hence, it is recommended that all communication be controlled by explicit communication protocols.

3.1.2 Lexical analysis

The lexical analyser takes the source VHDL file as input. This phase feeds the parser directly with a tokenised representation of the VHDL language. The tokens that it generates are representative of every type of item in the VHDL language. Keywords, operators, delimiters and values form the various classes of token returned, along with a translation of meaning for value class items such as integer constants and identifier strings. Any white space is ignored by the lexical analyser, so is not fed into the parser as a token. An example lexical analysis stream is shown in Figure 3.4.

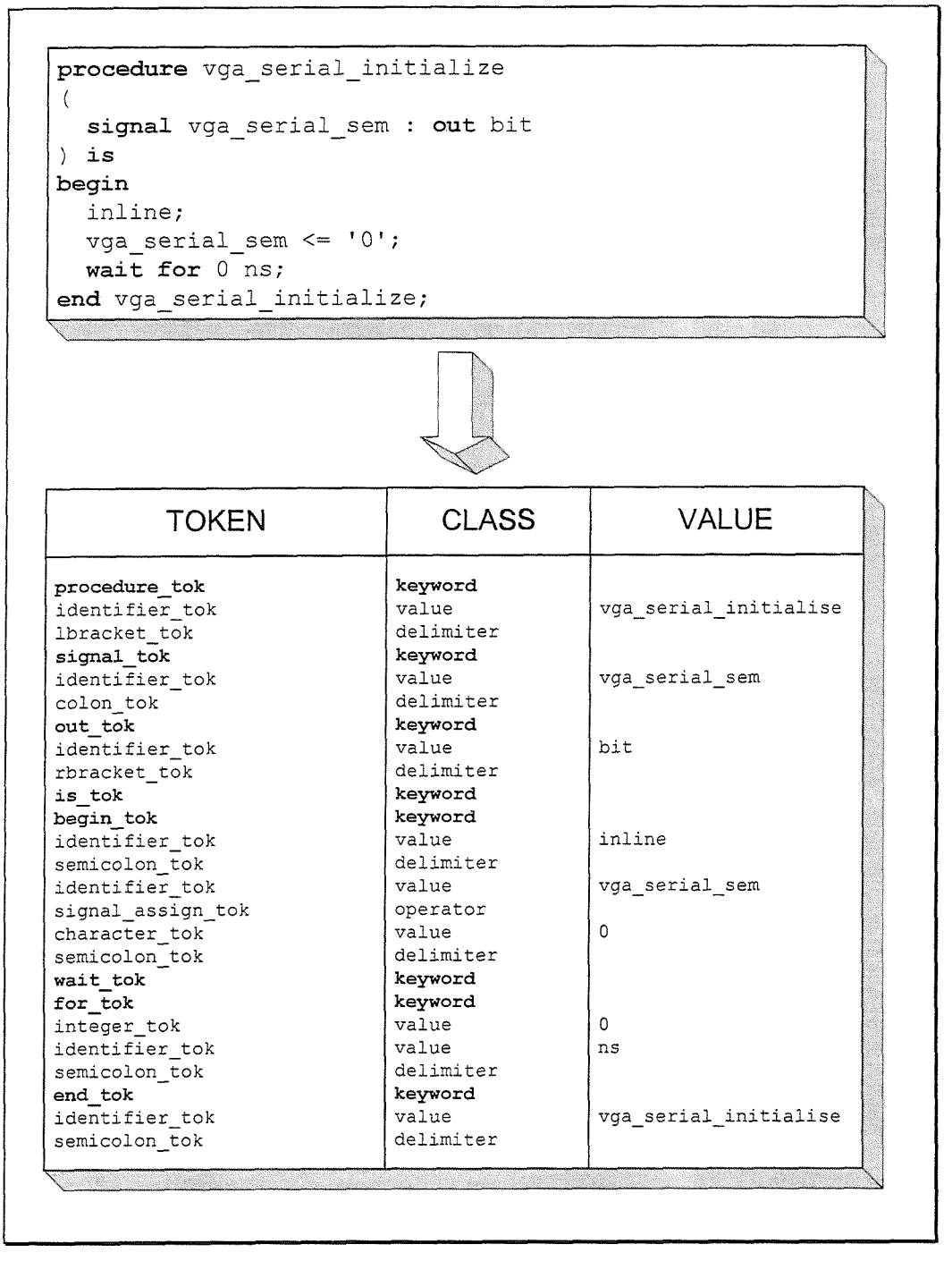


Figure 3.4 VHDL lexical analysis

3.1.3 Parser

The parser stage is used to build up an internal representation of the VHDL source code by accepting a limited sequence of lexical analysis tokens and value translations. A keyword token is usually used to begin a sequence of the language syntax. For instance, the initial token in Figure 3.4 is a '**procedure**' keyword. The only acceptable token to follow this keyword is the procedure name identifier. After this token is the possibility for a choice

between a left bracket delimiter, which defines that a port list exists for the procedure; a semi-colon delimiter token, which would finish the procedure as its declaration; or the '**is**' keyword, which would start the definition of the contents of the procedure. In the example case, a port list does exist and is terminated by the right bracket delimiter after the list of port items is parsed. The same choice then exists between the '**is**' keyword and the semi-colon delimiter tokens. It is in this way that the entire VHDL language is parsed. The compiler builds up an internal representation of the parse tree with cross-references made to the relevant data structures whenever identifier tokens are parsed.

3.1.4 Translation

The next stage in the compilation process is the generation of a simpler internal data structure that is a direct representation of the ICODE file to be generated. This is the translation step. Some constructs in the VHDL parse structure have a simple one to one mapping. For instance, VHDL **procedure**s and **function**s map directly onto an ICODE module and the **entity/architecture** definitions map onto the single ICODE program module that forms the root of the systems control flow.

VHDL **variable**s and **signal**s [80] are translated into ICODE '*register*'s, '*ram*'s, '*rom*'s, '*counter*'s, '*inport*'s or '*outport*'s depending on their use. An ICODE '*ram*' and '*rom*' are specified directly by the user, while a '*counter*' is inferred from variables defined by a loop construct. An ICODE '*inport*' or '*outport*' is defined for every input or output item in the I/O list of the module, with VHDL **inout** ports translated into separate ICODE '*inport*'s and '*outport*'s.

The processes in an architecture definition are merged into the root ICODE program module during translation, with the concurrent operation being defined by an initial multiple instruction activation list whose control flow never re-converges. The **process** is the only concurrent construct that is converted into ICODE. All other concurrent constructs are disallowed.

The **process**, **function** and **procedure** bodies all contain a sequence of VHDL operations. These operations may be formed from complex expressions. These expressions are translated into a list of ICODE instructions by recursively following the complex expression VHDL parse tree and building up a sequence of simple ICODE operations that

are based upon the VHDL operators involved in the complex expression. These operations act upon the registers, RAMs and counters that are translated from the original VHDL variables and signals. To break the complex expressions into simpler ICODE instructions, various temporary variables are created for the transmission of data from one simple instruction to the next. The number of bits required for the temporary variable storage is inferred from the operations taking place, where for example, an addition operation between two 8-bit variables is translated into an ICODE '*plus*' instruction with the result placed into a temporary variable of 9 bits. The width of the resultant temporary is enough to contain all possible resultant values of the operation.

Loop constructs in the parse tree are translated into actions on the loop variable that is directly translated into an ICODE variable. Assigning the starting value to the loop variable initialises the loop. This is performed by a simple ICODE '*move*' instruction. Any expressions contained by the loop follow the initial assignment. The loop iteration test is inserted after the translated contents of the loop. A conditional activation choice is made between the first generated ICODE instruction in the loop and the first translated ICODE instruction following the loop construct. This test determines whether to exit the loop or to continue for another iteration. The conditional activation is implemented using an ICODE '*if*' instruction, passing the result of a comparison of the loop variable with the loop end-condition as the single parameter. The loop iterator is incremented or decremented at this point.

A VHDL '**if**' expression is directly translated into an ICODE '*if*' instruction that is fed with the Boolean result of the translated VHDL expression. The conditional activations that follow activate the first translated ICODE instruction in either branch of the condition. The two branches of the condition are translated into two sequences of ICODE instructions that follow each other in the ICODE file. To stop the first branch activating the second branch after it is complete, the first branch performs an activation of the first ICODE instruction that follows the translated VHDL '**if**' statement in the ICODE file.

A VHDL '**case**' statement is translated in a similar way, with a direct translation into an ICODE '*switchon*' instruction. The multiple alternatives to the case test are made via multiple conditional activations of a number of ICODE sequences. Each sub-sequence in the ICODE file activates the first ICODE instruction that follows the case statement by translation.

A **procedure** or **function** call has a direct translation into the ICODE '*moduleap*' instruction. The call is made to a translated procedure or function, which in ICODE is translated into a '*module*'. The '*moduleap*' instruction references the module by name. The mechanism for parameter passing in both ICODE semantics and the structures generated by MOODS is *pass-by-reference* for both input and output parameters. This means that the parameters passed into the module are acted upon directly by the operations contained by the module. These semantics of translation allow VHDL signals to be passed through subprogram I/O lists directly, as the VHDL semantics for signal parameter passing is also *pass-by-reference*. VHDL variables however, use *pass-by-value* semantics, where the passed parameters are copied. A direct translation for variable parameter passing into ICODE *pass-by-reference* semantics is possible at this stage without any side effects due to the non re-entrant module structures generated by synthesis. Note that the modifications made for procedural recursion (described by Chapter 5) effectively change the parameter passing semantics into *pass-by-value* so that the VHDL behaviour for variable parameter passing is not broken for recursive subprograms.

3.1.5 Optimisation

The optimisation phase of the compiler is extremely naïve, and not to be confused with the optimisation capability of the MOODS core. It is simply used to reduce the number of ICODE operations that represent the design. The translation process uses dummy instructions as placeholders around block operations. This simplifies the translation process, but the dummy operations require removal. This is the first job of the optimiser. The second job is to remove redundant '*move*' instructions. These can occur between translated operation blocks and results in a sequence of '*move*' instructions that pass a single value through the sequence. These are optimised into single '*move*' instructions.

3.1.6 ICODE file

The ICODE file is generated directly from the internal representation of the ICODE data structures. There is a one to one mapping between the internal translated data structure and the generated file. This forms the last phase of the compiler program flow.

A fuller description of the ICODE format is contained in Appendix D.1. This description includes the modifications made for procedural recursion, explained in Chapter 5. The

most salient points about the format are listed below, followed by an example, Figure 3.5, which shows a fragment of a generated ICODE description, along with the VHDL source used to generate it.

- An ICODE file can contain a number of '*module*'s, which are translations of serial subprograms. The main '*program*' module forms the translation of the architecture body, which can contain the translation of any number of concurrent processes. Forming multiple unconstrained activations of the first translated ICODE instruction of each process supports process concurrency.

- An ICODE instruction forms a single sequential operation. It has the general form:
  `label: OPERATION <inputs>, <outputs> <activation list>`

- Each ICODE instruction can be activated by any number of other ICODE instructions. The instruction is then notionally executed. The instructions contained in the executed instruction's activation list are then activated. The activations can be conditional on the result of the operation performed by the executed instruction, allowing the values of resultant data variables to influence the direction of the control flow. Within the ICODE file, if no activations are listed for an instruction, then the single instruction following the executed instruction executes next. For example, in Figure 3.5, instruction '*i9*' activates '*i10*', which then activates '*i11*', followed by '*i12*'. In contrast, the instruction labelled '*i3*' only activates instruction '*i5*' given by the single actual activation and not the following instruction '*i4*'. Note the conditional activations are formed by the ICODE '*if*' instructions.

- Temporary variables are represented as integer constants in ICODE, and translations of actual VHDL integer constants are prefixed with '#'.

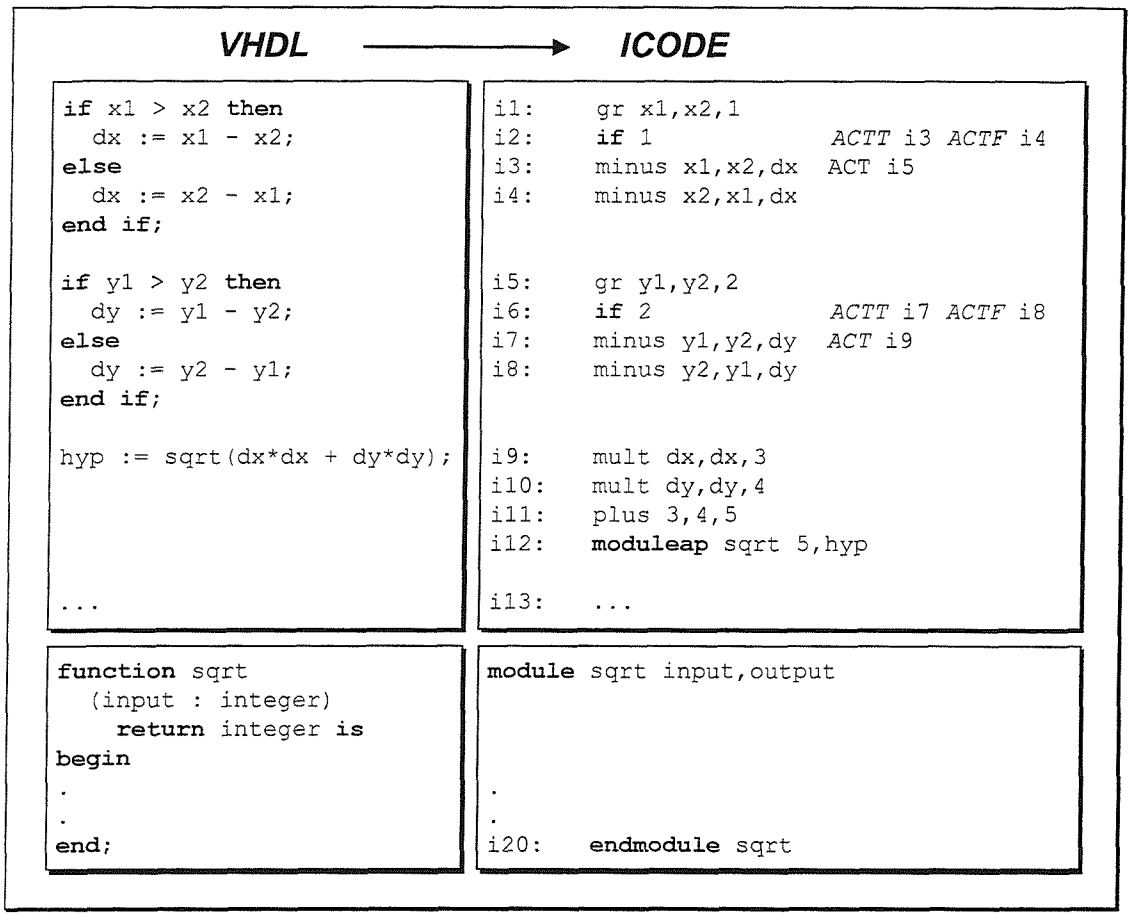


Figure 3.5 Unsigned hypotenuse calculation ICODE fragment

- The translation of complex expressions is split into a number of simpler ICODE instructions, with temporary variables being used to pass data through each operation.

- The activity of a subprogram '*module*' is initiated via the calling ICODE instruction '*moduleap*', which halts execution of the calling flow until the called module completes execution. A module completes when the '*endmodule*' instruction is activated. The instruction activated by the '*moduleap*' instruction (instruction '*i13*' activated by '*i12*' in Figure 3.5) is the first instruction executed after the call to the module completes.

3.2 MOODS synthesis core

The internal MOODS core data structures hold both the behavioural representation of the ICODE along with a fully bound structural implementation of the behavioural data path and control path. It is possible to output a structural representation of the system at any

point during the synthesis process after the ICODE file has been loaded into the internal data structures. The MOODS synthesis process is effectively the act of optimisation of these data structures by using multiple simple control and data path transformations, controlled by a transformation selection algorithm.

There are two main core data structures, the control path and data path graphs (see Figure 3.2). The control path holds a graph representation of every state in the controlling state machine, where each state executes one or more ICODE instructions. The data path holds a number of data path nodes that implement the operations performed by the ICODE instructions. The behavioural ICODE representation is not directly used by the structural output, which instead relies upon just the control graph and data path graph structures.

The structural representation is built in the initial stages of the MOODS core from the input ICODE. The initial generated structure is formed from a naïve implementation of the behaviour, where the structural construction algorithm places each ICODE instruction in a separate control state node and creates a separate data path node for each functional ICODE operation and ICODE variable. This means that the initial structure is both maximally serial, with no shared operations and variable storage elements in the data path.

The controlling state machine controls the data path from the state enable signals, where each state has a single enable signal that is high during an active state. These active state signals indirectly drive the data path nodes via a number of conditional signals. Particular data path units have controlling inputs that are driven from the control path. For instance, register-type data path nodes have the load-enable signals driven and multiplexor-type data path nodes have the selection inputs controlled. Feedback from the data path to the controlling state machine is formed via the same conditional signals, which can be used to determine the next state from conditional branches of the state machine. The outputs of comparison-type data path node operators are used for these conditional choices.

3.2.1 Control path

The control path data structure is formed internally from a graph structure, where each graph node represents a single control state in the controlling state machine. Each state is used to execute one or more ICODE instructions. Control arcs between the graph nodes form the links to the next and previous control states in the state machine.

The internal control graph data structure is sufficiently abstract that any number of physical implementations of the state machine could be used, each describing the same number of control states and transitions between states. At present, only one implementation method is used; a one-hot encoded token-passing structure, where each control state node is built from a control cell that contains a single register bit that is activated for one clock cycle by one or more token inputs to the cell. The activating token signals are representative of the arcs between the control states of the abstract control graph and the registered state bit forms the state enable signal, used to control the data path. This style of implementation suits the register-rich Field Programmable Gate Array (FPGA) architecture that is used for the demonstrators. Alternative state machine implementations could be formed from a binary or grey-coded state representation in limited register environments or even by a micro-coded controller.

Figure 3.6 shows the initial control graph and a data flow representation of the data path graph of the partial design described in Figure 3.5. The figure shows that each instruction is contained in a separate control state (S_1 to S_{13}), with the data flow between control states being stored in temporary registers (labelled 1 to 5) and translated intermediate registers ('*dx*' and '*dy*'). The data path operators that implement the ICODE instructions are shown on the right hand side.

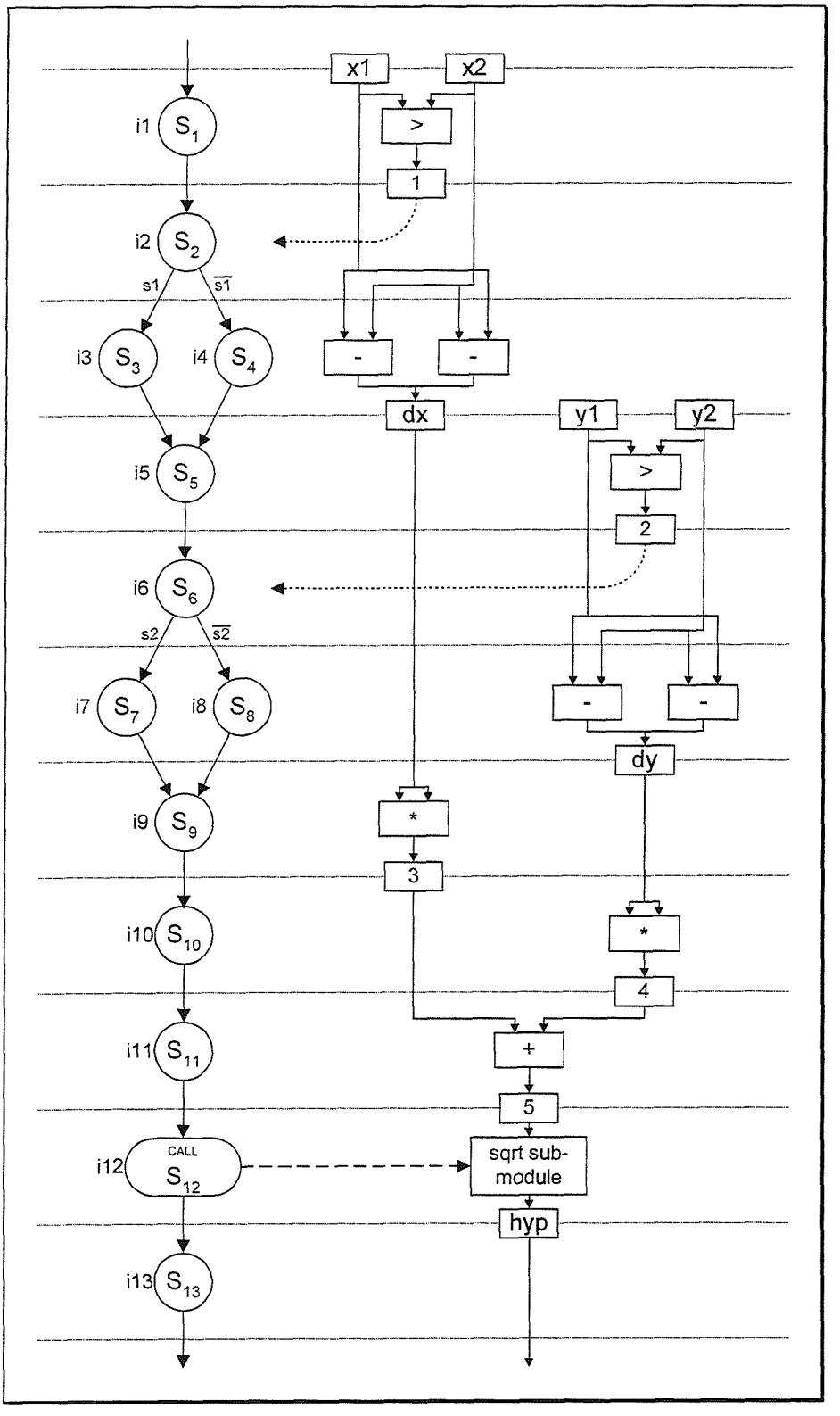


Figure 3.6 Initial control and data flow graphs for the unsigned hypotenuse calculation

Each control state data structure holds a list of ICODE instructions that are executed in that state. The instructions within a state are also partitioned into a number of *groups*, where each group contains an acyclic subgraph of instructions, where the graph

determines the data dependency between instructions in the same control state given by the flow of data between them. Each group can execute concurrently with any other group in the control state, due to the data independence of the instructions held by different groups. The data dependency information is only useful in a single control state, as it allows state-local concurrency to be utilised. Any instructions contained in any other control state execute at a different time, with no concurrent execution issues.

A single group of two data dependent operations in a single control state is shown in Figure 3.7a. The operations are data dependent as the result of the first addition operation forms one input operand of the second addition operation. The consequence of chaining the two additions in a single control state is that two separate adder data path units are required and the propagation delay for both operations must be summed together in order to calculate the register-to-register delay. This value is used to determine the minimum possible clock period for a design. Figure 3.7b shows the available time for each instruction, along with the present clock period idle time.

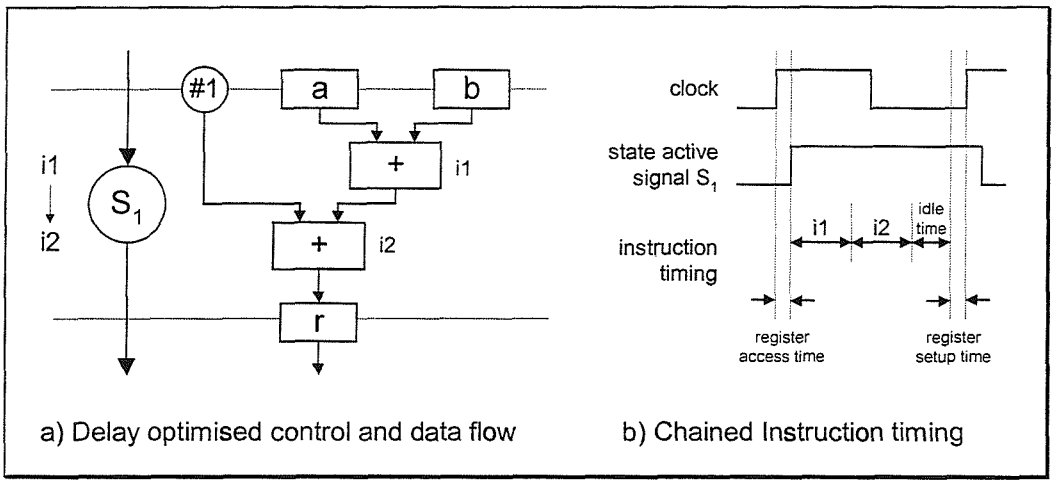


Figure 3.7 Execution of chained instructions in a single control state

The synthesis optimisation process requires knowledge of the execution time for each ICODE instruction in order to fully optimise the control path with respect to the required clock period. Characterisation data is fed from links to the implementing data path nodes of the relevant ICODE instructions. All data path nodes are fully bound to a physical technology-specific library cell during synthesis, from which the characterisation data is gained.

Six different types of control nodes are used in the control graph data structure. These are listed in Table 3.1 below. This is only a naming convention used to highlight the actions of different sections of the controlling state machine. When the state machine is optimised using scheduling transformations, the distinctions between the different types of control nodes become less apparent. The exceptions to this are the *collect* and *call* nodes, which cannot be merged with any other type of node. The collect node can be completely removed by the parallel merge transformation however.

Control node type	**Description**
General	This has a single input and a single output arc and can contain any ICODE instructions other than '*collect*', '*moduleap*' or conditional instructions.
Fork	This node can contain the same ICODE instructions as the general node and has a single input arc and two or more unconditional output arcs. This node is used to initiate a set of parallel execution threads.
Collect	This node contains a single ICODE '*collect*' instruction only and is used to synchronise a set of parallel execution threads into a single thread. The node has two or more input arcs and a single output arc. The node does not activate the next state node until a fixed number of input activations are received. This node complements the concurrent branching fork node. Note that the threads formed from the translation of VHDL process concurrency are not actually collected in this manner. Also, the VHDL compiler, rendering the collect node obsolete, no longer supports a concurrent translation of sequential threads. The mechanism is still supported by the MOODS core however, and is listed here for completeness.
Conditional	This node can contain any ICODE instruction supported by the general node as well as requiring a conditional ICODE instruction such as '*if*' or '*switchon*' to form the conditional branching choice. The node has a single input arc and two or more conditional output arcs. This node is used to initiate only one branch of a set of mutually exclusive execution threads.
Dot	This node is the complement to the conditional node. It has two or more input arcs and a single output arc. Any of the input arcs can activate the node. It forms the convergence of any number of mutually exclusive execution threads. It supports the same set of ICODE instructions as the general node.
Call	This node only contains a single '*moduleap*' ICODE call instruction. It has a single input and output arc. The control node forms the basis of the sub-module calling mechanism within the control graph. The call node stays active throughout the duration of the sub-module call, only activating the next state when the sub-module completes execution.

Table 3.1 Descriptions of the different control node types

Once the control graph is optimised, the only distinct types of node are the call, collect and general node types. These are physically realised by the '*call control cell*' described in Section 5.3.4.2 and the '*collect control cell*' (obsolete), with all other nodes realised by a '*general control cell*' described in Section 5.3.4.1.

3.2.2 Data path

The internal representation of the data path is formed from a disjoint graph of data path node units, connected indirectly via data path nets, which themselves have a level of indirection used to determine the bit-range connectivity of the multi-bit nets. The core graph node, the data path unit, describes the data path operations, storage and selective connectivity that implements the data processing side of the source ICODE file. There are three main types of data path node unit.

1. A *functional unit* implements ICODE operations such as additions, multiplications and comparisons. These operations are purely combinational, executing without the need for a controlling clock. These types of nodes are not controlled directly; they instead rely on the linked system to drive the inputs of the functional unit and to read the results of the unit at controlled time points. The job of the combinational functional unit is to produce the result of the operation in a specified amount of physical time beginning from the time that the inputs to the unit are modified. An exception to this rule is formed from the use of ALU type functional units, which can perform more than one type of operation. The type of operation is selected via a set of controlling input signals, driven from the controlling state machine. An example of an ALU unit is an add/subtract unit, where the unit is used in place of a single add and a single subtract unit under area considerations. Note that only one type of operation may be used in any single control state.

2. A *storage unit* implements the translation of ICODE variables (both user defined and temporary). A variable requires physical storage when its value is written to and read from, from within different clocked time periods. Each state of the controlling state machine executes in a different clock period, which means that any variable that is operated upon within two or more states with data flow between these states requires physical data storage. A number of different types of storage unit exist, optimised for different purposes. The general register type storage unit is used for

the storage of temporary variables and general data variables. A variable that is only ever reset and incremented (or decremented) is translated into a counter type storage unit. A third type of storage unit is formed from a multi-level array variable, where a '*ram*' type storage unit is created for this purpose. The controlling state machine directly controls storage units, where each unit has a set of input control signals. The type of control is dependent on the type of storage unit, with the most common operation being a register load operation. A register can be read at any time from a simple link to the register output. Note that the register is updated at the very end of the execution period of the control state in which a write operation is performed, leaving the rest of the execution period to calculate the value written into the register. This situation is seen in Figure 3.7b, where both addition ICODE instructions i1 and i2 are performed in the same control state, S_1, as the register load operation of variable '*r*'.

3. The final type of data path node unit is the *interconnect unit*. These units are used to select the inputs of any shared data path unit that has multiple input nets. The interconnect units are only physically generated as a post-processing step after the core optimisation process has completed. The library cell that implements the interconnect node type is a multiplexor. As these multiplexor cells are not physically created until the post-processing step and the cells have both area and delay factors, the optimisation process must take into account these factors from the implied position of the multiplexor cells. Multiplexors are not physically created during synthesis for both time and code size efficiency of data path modification reasons. A multiplexor is implied when a data path unit has more than one input net connection, where selection between the different inputs is required at different times in the control flow, controlled by the state machine. The multiplexor has a number of selection inputs that are driven from the state machine enable signals via the conditional equations, in a similar manner to the ALU select signals.

Every data path node is treated in the same manner and with the same priority as every other data path node. The data path node graph element is stored in a generic data structure block. Links into the cell library give the bound functionality of each node, along with the area and delay information used in the optimisation process. The generic nature of the data path nodes gives technology independence to the synthesis core, while allowing

technology specific cell information in the form of the area and delay estimates stored within the selected cell library to feed the synthesis process.

The signals that link the data path nodes to the control path nodes are represented by Boolean logic equations. This abstraction of the control signal generation allows for further logic optimisation of these linking signals. One reason that direct-linkage between the control and data paths cannot be used is due to the scheduling optimisations merging control states together, including conditional branches. If, for example, a variable is updated by one conditional branch of a control flow and not in the other, and both branches are optimised into a single control state, the register load enable signal requires a conditional drive dependent on the branch selection comparison result now calculated in that control state. Another, perhaps simpler reason for the need of linkage equations is that a register may need updating from a number of control states. The register load enable signal in this case is formed from the logical-OR of all of the state enable signals in which the variable being stored by the register is updated.

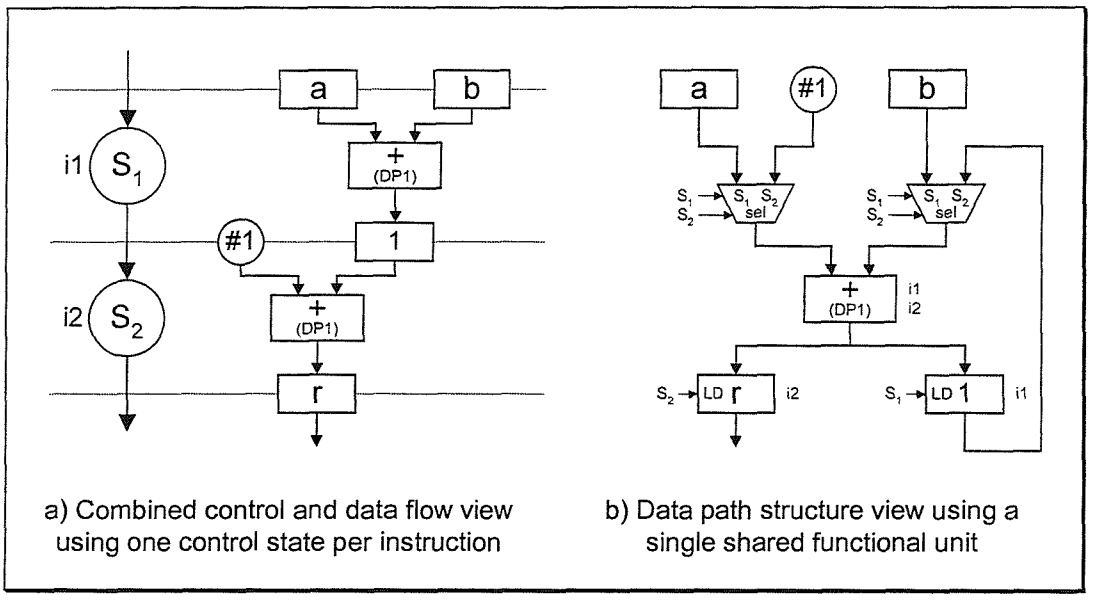


Figure 3.8 Data flow and data path views of a shared adder functional unit

The data flow diagrams shown in Figure 3.6 and Figure 3.7a show the activity of operations unrolled into the time-steps in which they operate. This is not fully representative of the data path graph described by this section. Figure 3.8 illustrates the differences between the data flow representation and the actual generated data path for a shared adder functional unit, used unconditionally in two consecutive control states. The temporary result of the first add-operation requires register storage as it is written in state

S_1 and read in state S_2. Note the inclusion of the interconnection multiplexors, which are generated by the post-processing step. This type of structure is generated if a design is optimised for area, or if the add-operations cannot be chained together when delay optimising due to a specified clock period being less than the combined delays of two adder functional units.

3.2.3 Transformations

The optimisation is an iterative process, where the task is split into many small local optimisation transformations on selected parts of the design. This allows the traditional synthesis sub-tasks of scheduling, allocation and binding to be performed simultaneously within the optimisation loop [81,82]. The transformation selection and design section to transform are selected by the optimisation algorithm.

Each local transformation is semantic preserving, resulting in a complete design after every execution of a transformation on the design. The synthesis data structures hold a complete and fully bound design throughout the entire synthesis process. There are fourteen different transformations, each performing slight changes in the design to adjust the scheduling of the controlling state machine and the allocation and binding of the data path. The fourteen transformations include six inverse transformations that allow backwards steps to be taken within the simulated annealing optimisation algorithm.

There are four steps that relate to the application of a single design transformation. This forms a single iteration of the optimisation process. The steps are listed below.

1. *Selection*. The initial stage of a transformation is to select the transformation to apply from the fourteen available, and the portion of the design to which the transform is to be applied. The optimisation algorithm controls this stage.

2. *Testing*. The second stage is used to test the validity of the given transformation on the portion of the design that has been selected. It is possible for some transformations to alter the design behaviour if applied incorrectly. This stage is used to filter out these misapplications by aborting the transformation.

3. *Estimation*. The third stage calculates an estimate of the effects that the transformation would have upon the design performance. This stage does not affect the core data structures, leaving the design intact. The optimisation algorithm uses the result of the estimation to determine whether it is beneficial to apply the transformation. The transformation can be aborted at this stage.

4. *Execution*. The final stage performs the physical design transformation.

3.2.3.1 Scheduling

Scheduling transformations are used to modify the control graph with a change in the assignment of ICODE instructions to control nodes and a change in the number of control nodes used to perform a number of ICODE instructions. There are four state merging transformations, two inverse state-splitting transformations and a clock period adjustment transformation supported. These transformations and their effects are listed in Table 3.2 below.

Transformation	Effect
Sequential merge	The most basic control node merging transform takes two consecutive control nodes and moves all the instructions contained in the second into the first. The second node is then removed from the state machine, as it implements no ICODE instructions. Any data dependencies between ICODE operations in the merged control node result in these operations being chained together within an acyclic instruction group graph, with all intermediate data values having their registers bypassed.
Parallel merge	This merging transform is applied to a concurrent branching fork node, where the first nodes in each branch are merged into a single successor node. This replaces the unconditional arcs to the multiple concurrently executed nodes with a single unconditional arc to a single control node that performs all of the operations previously each controlled by a separate control node.
Merge fork and successor	This transformation combines elements of the first two, taking a branching node (fork or conditional) and merging the successor instructions contained in one branch into the branching node. This also results in operator chaining and register bypassing. Another feature of this transformation occurs when two conditional branches are merged into the branching node, forming an unconditional activation of the successor to both branches.

Transformation	Effect
Group instructions on register	This transformation is geared to removing temporary variable register storage by trying to remove registers with one input net and one output net. These variables are accessed by one reading instruction and one writing instruction. The transformation attempts to merge the group containing the writing instruction into the control state containing the reading instruction. This results in the register being bypassed (removed) if successful.
Ungroup into groups	This inverse transformation moves groups of instructions in a control node into two separate control nodes with the first node containing a single selected group and the second node containing all other instruction groups originally contained in the single control node. As groups of instructions are data independent, the execution order of the separated groups cannot break the behaviour.
Ungroup into time slices	The second inverse scheduling transformation splits all the instructions in a control node into a sequence of control nodes such that the time taken by any instruction group in any generated node does not exceed a specified time value. This transformation can reinstate previously bypassed registers used to store temporary values between control states. Any instructions that exceed the specified time period on their own require multi-cycling.
Clock set / multi-cycling	This transformation is really a global optimisation step that specifies the clock period to optimise to. The ungroup into time slices transformation is applied after the clock period is adjusted, so that no control node violates the supplied clock period.

Table 3.2 Scheduling transformations

3.2.3.2 Allocation and binding

As with the scheduling transformations operating upon the control path, the allocation and binding transformations act upon the data path, where the transformations are concerned with the sharing and unsharing of data path units. The four unsharing transformations are provided as inverse transformations to the two sharing transformations. A further binding transformation is also provided that can select different functional units to perform the same operation. This does not require an inverse transformation, as it can reverse the actions of previous binding transformations itself. These seven data path transformations are detailed in Table 3.3 below.

Transformation	**Effect**
Combine functional units	This transformation tries to merge two functional units into a single functional unit. This is only allowed when none of the linked instructions performed by each source unit are performed at the same time. This has the effect of further time-sharing a unit between multiple operations, where a functional unit can perform only one operation at a time. A merged functional unit results in a number of different inputs to the unit, which are selected by the multiplexor interconnect nodes, which are themselves controlled by the state enable signals. Note that the availability of multi-function ALU units in the cell libraries enhances the actions of this transformation by allowing different types of operation to be merged.
Share registers	This transformation attempts to share storage units, in particular register units. This can only be performed if the variables stored in the two registers being shared have non-overlapping lifetimes. Lifetime analysis for each variable takes into account mutually exclusive conditional branches and variable persistence through loop constructs.
Uncombine instruction from unit	This transformation forms the inverse of the combine functional units transformation. It takes a functional unit that implements two or more ICODE instructions and removes one of these instructions from the unit, creating a separate functional unit specifically for the single removed instruction. This relies on the cell library to determine the type of unit to use for the implementation of the extracted instruction. The unit from which the instruction is extracted may also have the unit type re-evaluated, as an ALU could now be replaced by a single-function unit, dependent on the types of operation left being performed by the original unit.
Uncombine unit fully	This transformation utilises the uncombine instruction from unit transformation described above to completely uncombine all ICODE instructions from a functional unit into a number of functional units, each performing only one instruction from the original shared unit.
Unshare variable from register	In a similar manner to the first uncombine transformation, the unshare variable from register transformation takes a single shared register type storage node and splits one of the implemented variables into a separate storage node.
Unshare register fully	This transformation utilises the unshare variable from register transformation described above to completely unshare all ICODE variables being implemented by a single register storage unit. This results in a number of separate registers, each being used to store only one ICODE variable.

Transformation	Effect
Alternative implementation	This is the only binding transformation. For this transformation to have any effect, two or more different implementations of a type of unit must exist in the cell library. The transformation attempts to replace a unit of any type with an alternative implementation that has different area and delay characteristics, changing the cost of the unit. The cost function used by the optimisation algorithm is used to determine whether to accept the new unit binding.

Table 3.3 Allocation and binding transformations

3.2.4 Cost function

The cost function is used during the estimation phase of transformation application. It provides a measure of the change in design characteristics over the application of a single transformation. The function is determined by the target objectives specified by the user, where the multiple, possibly conflicting objectives are used by the cost function in a weighted-sum calculation to generate a single value representation of the change of "energy" of the system with the application of a transformation.

The user objectives can be the design area, delay, power consumption or any other measurable factor that is stored by the cell library. The user assigns priorities to each measurable item, which is used to weight the level of influence of each objective used by the cost calculation. The objectives and the state of the system are used to generate an actual cost value.

The change in energy of the system is given by:

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

Where $C_{estimate}$ is the estimated cost of the system after the transformation is applied, $C_{previous}$ is the cost of the system before the transformation and $C_{initial}$ is the cost of the design after its initial construction. A negative result indicates an improvement in the design structure with respect to the user objectives.

3.2.5 Optimisation algorithms

There are two optimisation algorithms that MOODS currently has implemented. Both use the transformation selection, testing, estimation and execution method of applying the single transformations in the main synthesis loop. The algorithms are in charge of the transformation and design portion selections as well as the number of transformation iterations to execute.

3.2.5.1 Simulated annealing

The first algorithm is based on physical annealing [83,84,85,86], which is performed by slowly cooling a material from a high-energy liquid state into a minimal low energy solid state. If the cooling is controlled properly, the final energy state will stabilise at a globally minimum level for the whole material, reaching thermodynamic equilibrium as the material freezes.

It is surmised that a structural design could have many local minima on the configuration path to achieving a global minimum cost value. It is specifically for this reason that the simulated annealing algorithm is used. Figure 3.9 illustrates this, where a physical cost value of a number of closely related one-dimensional design configurations are shown.

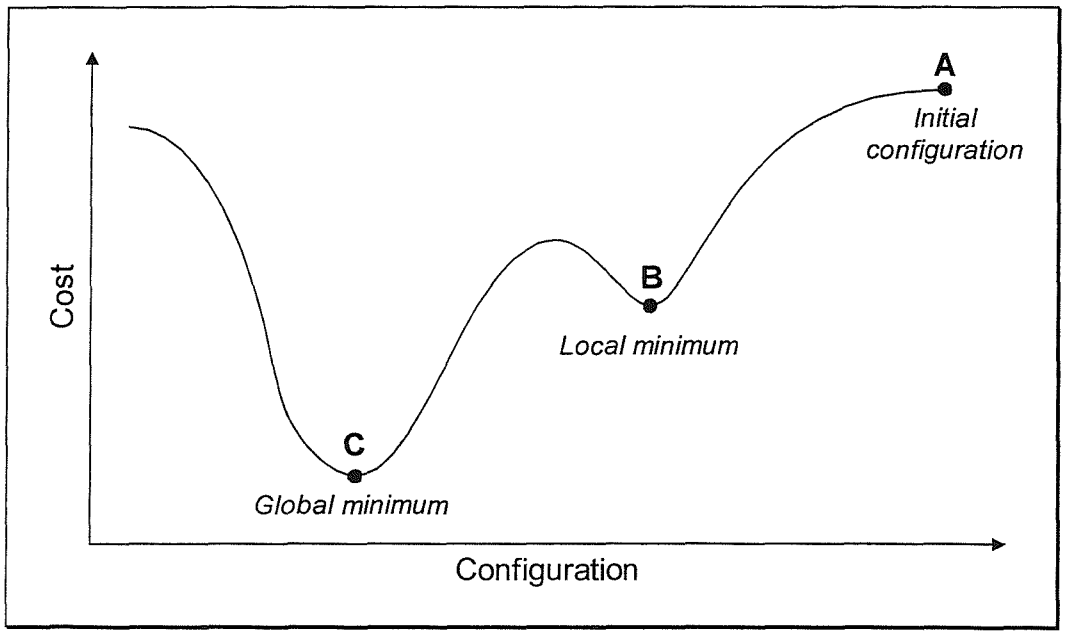


Figure 3.9 Design cost plotted against a single dimensioned configuration space

The annealing algorithm works by selecting a random transformation and design section to operate upon and performing the estimation of the cost value after the transformation. From this, the cost function is used to determine the change in energy of the system, whether it results in degradation or an improvement in the design structure.

Any cost *improvement* is automatically accepted, and a cost *degradation* is accepted dependent on the probability:

$$P = \exp\left(\frac{-\Delta E}{T}\right) \quad : \quad \Delta E > 0$$

Where P is the resulting probability given between 0 and 1 of acceptance of degradation, ΔE is the estimated positive change in energy given by the transformation and T is the temperature set by the annealing algorithm. This shows that as the temperature decreases, the probability of acceptance of degradation also decreases, as well as a large degradation having less chance of acceptance than a smaller degradation at a given temperature. The actual choice of whether to accept is made from the comparison of a normalised random number with the acceptance probability.

The annealing algorithm is implemented by a nested pair of loops, with the outer loop generating slowly decreasing temperature values and the inner loop counting for a fixed number of transformation iterations performed at each temperature value. The initial temperature, final temperature, rate of change of temperature and the number of iterations performed at each temperature level determine the optimisation speed, the ability to find the global minima and the point at which optimisation ceases. These values require manual selection by the user, with trial and error used to determine the best annealing schedule for each design. Generally, optimisation speed is traded off against the quality of the resultant design structure.

3.2.5.2 Tailored heuristic

As degradation of design cost is allowed in the simulated annealing algorithm, with a slow reduction in probability of degradation acceptance over the course of the algorithm, simulated annealing is found to take a large amount of time. This led to the creation of a faster heuristic algorithm that utilises a fixed optimisation schedule, guided by an analysis

of the design, which produces a predictable final structure for every optimisation run of any fixed design. The heuristic is designed to perform trade-offs between area and delay only, with knowledge of the trade-offs involved gained through analysis of a number of test designs.

The heuristic [87] uses the same set of transformations that are used by the simulated annealing algorithm apart from the inverse transformations. This means that the algorithm applies only improvement steps without any backtracking. Two base routines are provided that optimise for area and delay. These are:

1. *Compact control path.* This routine utilises the scheduling (control graph merging) transformations to reduce the number of control states used to execute sequences of ICODE instructions. This reduces the delay by performing more within a single control state and slightly reduces the area with the removal of control state nodes and with temporary register bypasses.

2. *Compact data path.* This routine utilises the allocation and binding (data path node merging) transformations to optimise the data path for area. Trade-offs are made here for the area saved by merging operations with the area created with the creation of input driving multiplexors.

Both routines make use of a number of design metrics, such as the control path's *critical path*, which determines the control nodes, which affect the circuit delay the most. Each data path unit is given a *shareability factor*, which determines the best units to share for the best area savings to be made. Each control node is also given a *share factor*, which gives an indication of the effect that control node merging has on the ability for future data path unit sharing. An equivalent of the share factor for the data path is the *critical path factor* that is assigned to each data path node. This gives an indication of how close the functional unit is to executing ICODE instructions on the critical path.

Throughout the optimisation process, trade-offs are made between the results of merging control states into one, creating operator chaining, against the results of merging the functional units, forcing separate control states for operations performed by a shared data path unit. The two routines are designed to optimise to a particular design objective, while reducing the effect on the secondary objective.

The two objectives are area and delay minimisation, which can be assigned different or equal priorities by the user. If the delay objective is given priority, the *compact control path* routine is called with an increasing share factor threshold until the delay target is met or no more compaction can be performed. The *compact data path* routine is then called by a similar loop with an increasing critical path factor threshold until the area target is met or no more sharing is possible. If the area objective is given priority, then the compaction routines are called in the reverse order. If both the area and delay objectives are given the same priority, then the compaction routines are called by a single loop, with both an increasing share factor threshold and critical path factor threshold. After all of these optimisation runs, register-sharing transformations are applied to any registers capable of being shared and alternative cell selections are made.

3.2.6 Subprogram conversion

Subprograms get converted into ICODE modules during compilation. These modules have a contained control flow, with a single entry point and single exit point in the form of the '*module*' and '*endmodule*' instructions. During optimisation, the single exit point may be split into a number exit-points dependent on any conditional control flow branches and control node merging (see Figure 3.10c). The '*endmodule*' instruction is removed from the instruction flow, as it has no physical meaning, being replaced by an end-signal driven from all control nodes that contain an exit point.

Module activity is initiated by a '*moduleap*' (module-leap) call instruction contained in a separate module. The special '*call control node*' state machine cell implements the call instruction. The extra '*activate*' signal generated by the call node is used to drive one of the token inputs of the start-node in the called module, which initiates the called module at the same time as the call node itself (see Figure 3.10d for an example timing diagram).

The end-signal generated by the called module is used to feed the extra '*end*' input of the call control node. The call control node activates its successor node (containing the instructions that follow the subprogram call) only when the call control node is active and the '*end*' signal is driven. The call control node then enters an inactive state. The call control node is left in an active state for the duration of the call. All this is shown by the example given in Figure 3.10(a to d).

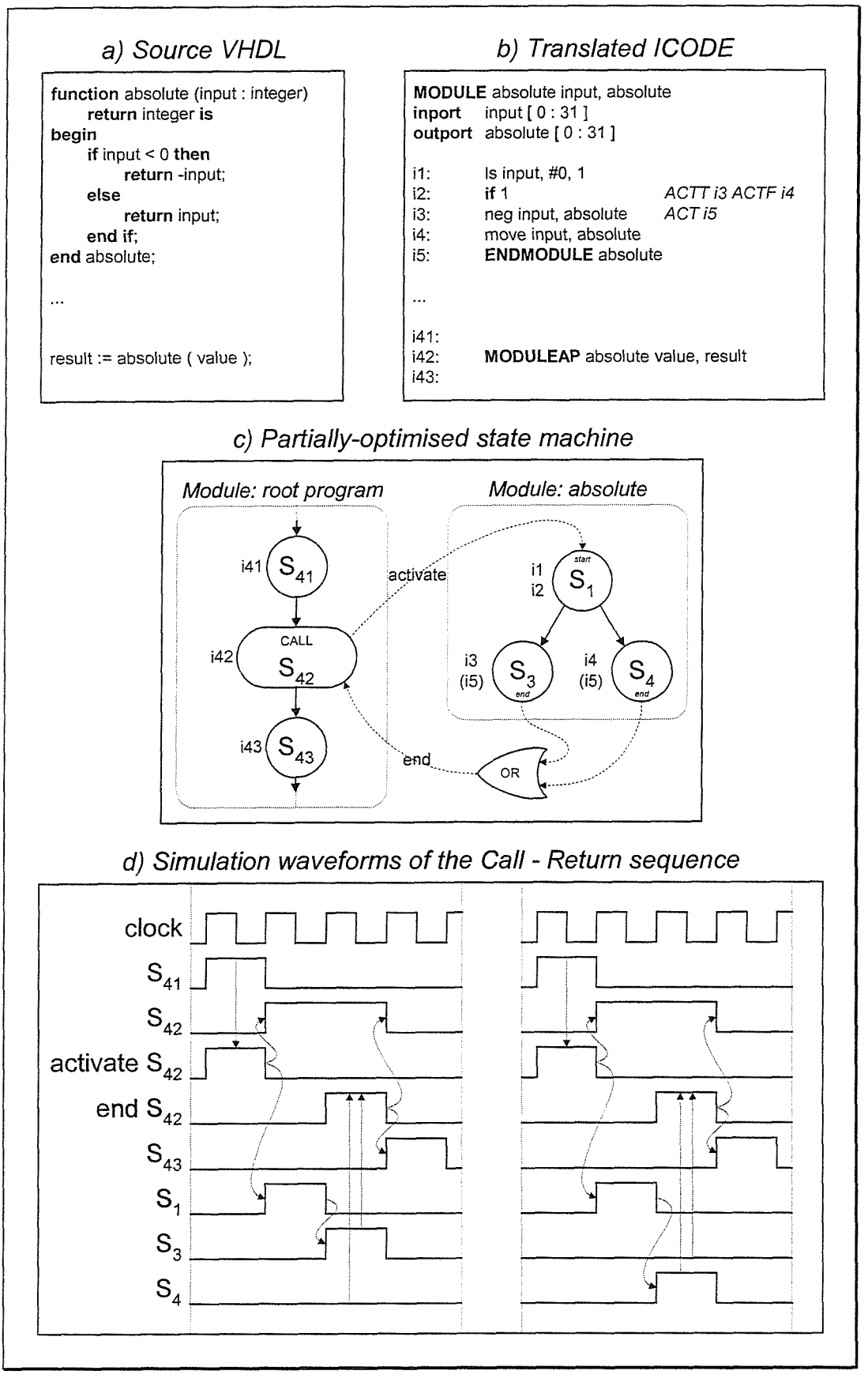


Figure 3.10 Module call-mechanism example

Any I/O variables are passed by reference in the generated output structure; the registers used to hold the passed output values are driven directly by the submodule controller and the inputs are read directly from the passed values, which can be constant or variable. The controlling signals for the register load-enable inputs and multiplexor select inputs are generated from the multiple levels of '*moduleap*' ICODE instructions for nested calls, or more specifically, from the call control node activity signals, combined with the local data path unit driving signals.

All of the control signals used by the calling mechanism are generated during the post-processing stage of MOODS from a translation of the ICODE instructions that implement the call.

3.2.7 Post-processing

The post-processing stage is used to complete the structural description of a design. It has been said that MOODS contains a full structural description during the entire synthesis optimisation process. This is not untrue, as optimisation may be stopped at any point. The post-processing stage is only used to complete the structures that are implied, during optimisation, by the allocation of ICODE instructions to control nodes and from the multiple input nets into data path units. The post processing stage can also be used to insert a number of run-time data path tests [88,89].

The first step of the post-processing stage is to generate any multiplexors that are required, as these interconnect data path nodes are completely implied during optimisation for efficiency reasons. A multiplexor is required when a data path node (other than a multiplexor) has multiple input nets that are driven when specified ICODE instructions are active. The multiplexor is created and linked into the data path structure and also given a copy of the link to the ICODE instructions that are used to drive each input. These instructions are used by the second post-processing step.

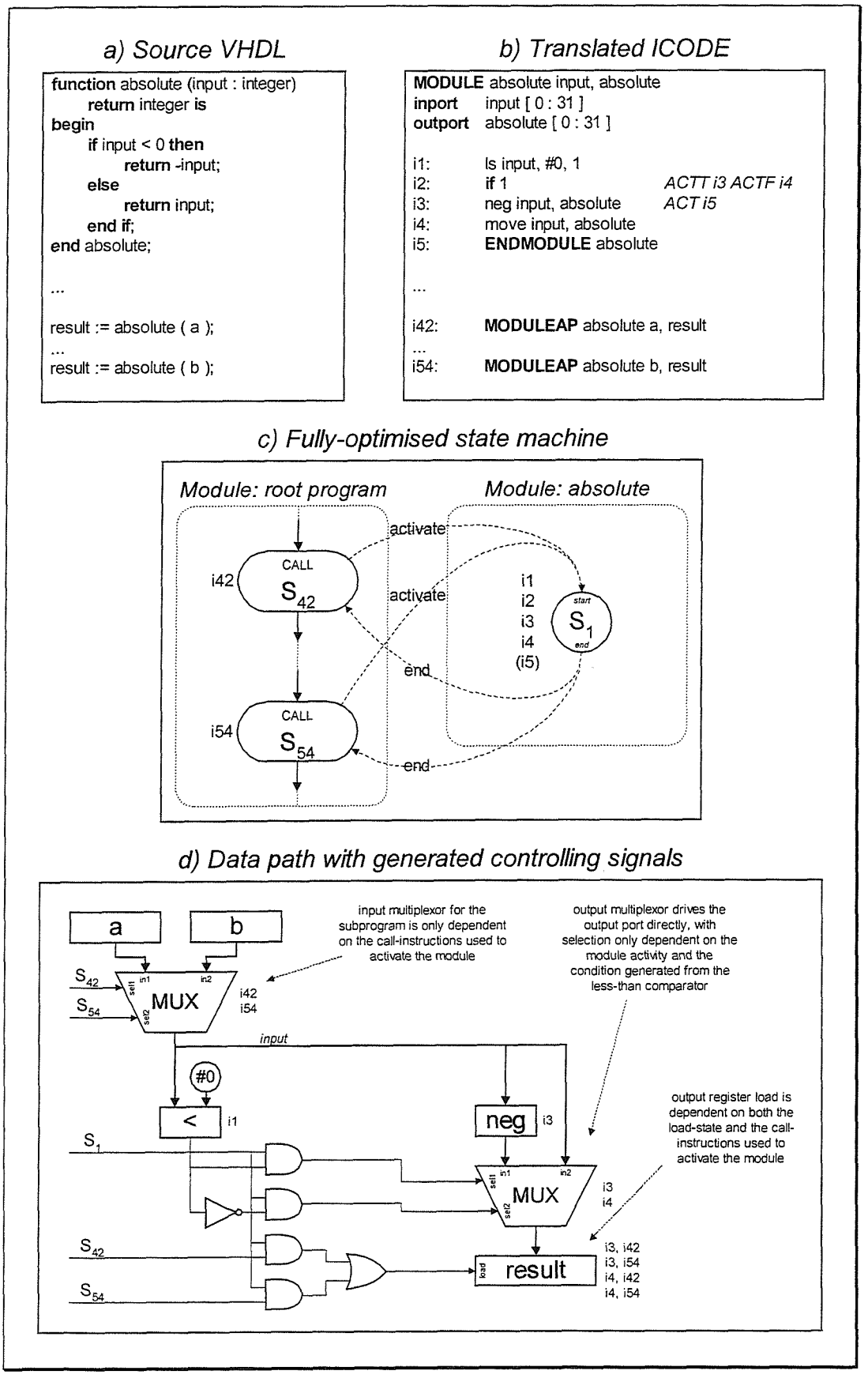


Figure 3.11 Control signal generation example

The second step is to generate a number of control signals within the conditional signal list. These are formed from Boolean equations and are used to drive the control inputs of every data path node. These control inputs are typically created for multiplexor and ALU selection signals, register load-enable signals, counter signals and memory read/write signals. This action effectively fills in the controlling links between the control graph and the data path graph. These signals are generated from a translation of the ICODE instructions that are implemented by the data path units. The control nodes in which the linked ICODE instructions are contained form the sources of the generated signal. Any relevant conditional branching equations active within the control node in combination with any call stack control nodes are included in the generation of the control signals. This is shown by Figure 3.11d, which uses the same VHDL '*absolute*' function used by Figure 3.10. In the second case however, the control graph is fully optimised (Figure 3.11c), with instructions '*i1*' to '*i5*' merged into a single state. The function is called twice, with different parameters passed (Figure 3.11a and Figure 3.11b).

The final post-processing step is used to tidy up the data path graph, removing any unused registers, which have been bypassed during the optimisation process. After this final stage, the output files are generated, including the DDF data structure dump, explained in Appendix D.2 and the structural VHDL file generation, explained in Appendix A.5 as part of a new process addition to MOODS introduced in the next section.

3.3 New features

The addition of support for dynamic memory raises the abstraction level in terms of data structure creation and supported language features above the present level. The version of MOODS described in this chapter creates structural designs with all data and the controlling state machine created with a static memory paradigm. The point of implementing a system that uses abstract data types is that the simple hardware one-to-one description is migrating towards a similar abstraction level to software. The increased support for composite data structures and procedural recursion leads to the increased use of subprogram procedures to handle common actions upon these data structures. The additions for procedural recursion and dynamic object creation, each increasing the benefits of the other, require modifications to the system shown in Figure 3.1. The modified system is shown in Figure 3.12.

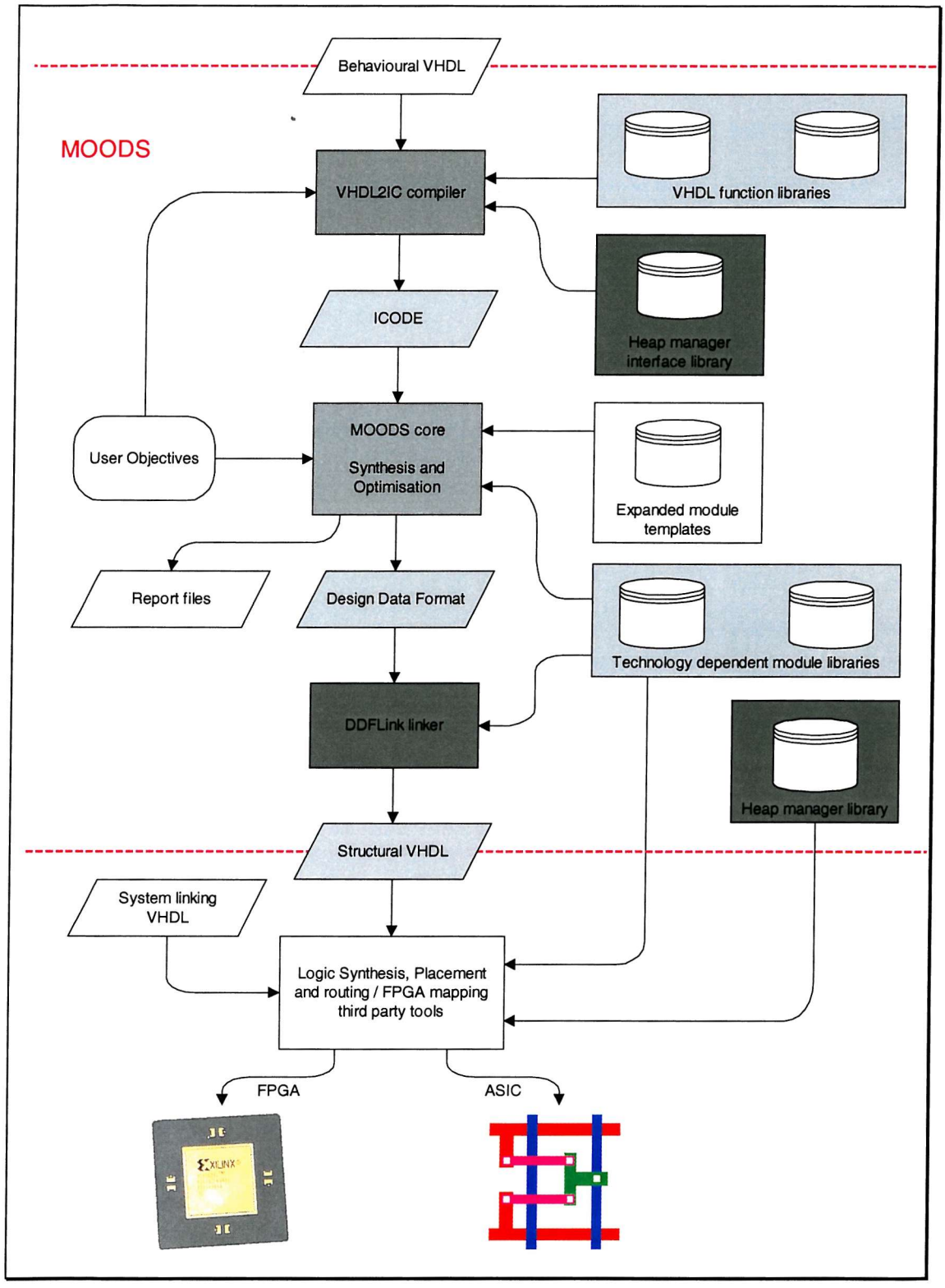


Figure 3.12 Modified MOODS system data flow

The diagram above shows the system data flow structure after the inclusion of the modifications that implement dynamic memory allocation. The darker shaded regions indicate a larger amount of modification than the lighter shaded regions. The darkest blocks indicate a new part of the structure. The diagram shows modifications made to the

VHDL compiler, the inclusion of a heap manager interface library, the ICODE file format, the MOODS core and to the DDF file format. It also shows the inclusion of a new back-end linker program used to generate a slightly modified structural VHDL file format.

Not contained in Figure 3.12 is the creation of a Graphical User Interface, GUI program that controls the synthesis process from internally generated scripts, displays sections of the internal data structures, namely the control graph (see Appendix A.5.5) and allows a project structure to contain the various input files used by a user's design. This program was initially designed only to display sections of the MOODS data structures and is now being continuously developed as a full controlling GUI.

Explanations of the modified and created structures in Figure 3.12 are listed below, highlighting the changes made for dynamic memory:

1. *Compiler and libraries*. The compiler is modified by increasing the synthesisable subset of the language by enhancing the parse tree and the translation of the parse tree into ICODE format. Explicit dynamic memory actions interface to a heap manager with the use of an interface library. Procedural recursion is supported via the generation of dynamic return address and stack manipulation instructions. Procedural inlining is also implemented for efficiency reasons.

2. *ICODE file format*. The ICODE file format is modified to support procedural recursion. No modifications are required for heap memory support. Refer to Appendix D.1 for the full ICODE description.

3. *MOODS core and cell libraries*. Both the MOODS core and cell libraries are modified to support procedural recursion. The cell library has a different type of call control node inserted, which is created for the recursive calling mechanism. The MOODS core heuristic optimisation algorithm uses a modified critical path calculation. The post-processing stage of MOODS is also enhanced with the inclusion of compiler-generated return addresses used in the control flow decision-making process via automatically generated decoders and conditional linking signals.

4. *DDF file format.* The DDF file format is modified to support procedural recursion. Changes to fully support the post-processed structural design data structures are also made. Refer to Appendix D.2 for the full DDF description.

5. *DDFLink linker and modified structural VHDL.* This program is completely new. It was anticipated that it would act as a linker for various structural designs including an automatic link to the heap manager system for any design using dynamic memory. For a full description of 'DDFLink', refer to Appendix A.5. The program is now used for the generation of the structural VHDL file output, previously outputted by MOODS directly.

6. *Heap manager library.* This library is used to link with any design that uses dynamic memory. The library contains the heap manager controller and underlying memory controller, explained in Chapter 4.3. Using MOODS to synthesise the behavioural description of the controller creates the library. The heap manager forms a concurrent system that interfaces with any users' designs that require dynamic memory storage. The library is linked during logic synthesis with the use of the top-level VHDL file.

Chapter 4

Dynamic allocation

This chapter describes the integration of dynamic memory allocation into the MOODS synthesis tool. The language used for input of behavioural designs is VHDL, an IEEE standard. VHDL is capable of describing dynamic data structures as part of the standard language [6], so the language constructs are used to describe any dynamic memory behaviour directly.

The chapter begins with a description in Section 4.1 of the use of dynamic memory in the VHDL context and introduces the modified system structure and enhancements to the supported VHDL subset. Section 4.2 describes the modifications made to the VHDL compiler in order to fully support dynamic structures. The dynamic data that is created for the user requires support at runtime in the form of memory management. A memory management scheme optimised for a particular design style is described in Section 4.3. The effects of incorporating dynamic memory in the described manner with respect to behavioural optimisation are noted in Section 4.4 and the effects and handling of potential errors are shown in Section 4.5. Finally, any limitations and alternative implementations and methods are explored in Section 4.6.

4.1 General overview

Dynamic memory is defined as storage space that is created, used and destroyed at runtime. The concept of '*runtime*' in a synthesis environment is with respect to the synthesised design, where the time in which the design is active either during simulation or as a powered physical design defines runtime.

Many languages have support for the use of dynamic memory in its various forms. The most commonly used form of dynamic memory comes from the implicit use of a stack to

hold the local contents of the memory within a procedure. This kind of dynamic memory is described in Chapter 5 as part of the implementation of procedural recursion.

The explicit creation and deletion of storage space for object types is the other form of dynamic memory that goes hand in hand with the method for referencing the dynamically created memory. This memory is not directly linked to the call tree structure of a user's design, as is the stack frame; it is created at any point within the flow of the design and deleted at any arbitrary point further into the execution flow from a memory space known as the heap.

Most languages provide access to dynamically created objects via a direct memory address pointer, which uniquely identifies the object within an address space and allows direct access to the address value. VHDL uses the concept of **access** types that contain the reference to a particular object type without allowing access to the actual value stored by the **access** type. This means that a translation of VHDL can use a direct memory address to store an object reference, but no access of the actual underlying address may be gained from the language. A VHDL variable is used to store the value held by an **access** type just as a variable would store an integer or bit vector, but the supported operations on the variable are limited to dereferencing operations, with no direct modification allowed.

For every object type that is stored dynamically, VHDL requires that a type must be defined that points to objects of the particular type requiring allocation. This is because the strong typing of VHDL disallows a generic pointer type. This also means that only **access** type references to a particular object type can be used to reference that object, with no casting between **access** types or with any other types allowed.

The main use for dynamic memory is found with the creation of complex data structures that can be manipulated in a structure by simple reference re-assignment. This reference modification is built into the language along with methods for creation and deletion of dynamic objects using these references and methods for gaining the value referenced by the **access** type variable (dereferencing). This makes source code neater and smaller than would be found if the user were required to explicitly create and manage a structure to hold and manipulate the required dynamic data structures.

The use of complex data structures has been well proven in software programs to help with the solving of complex problems. As the level of programming style of behavioural synthesis is becoming more abstract and approaching the level of abstraction found in many software languages, the addition of dynamic structures to designs created by a synthesis system will bring the behavioural synthesis abstraction level even higher, which enables problems previously only easily solvable within a software environment to migrate into the hardware domain of behavioural synthesis, with its associated benefits.

The rest of this chapter describes how the concept of dynamic memory access is incorporated into the MOODS behavioural synthesis system. It details the many modifications to the VHDL compiler that enable the language defined methods to be used and describes a memory management environment from which the dynamic memory is allocated via an interface that is automatically incorporated by the VHDL compiler into the user's design.

The integration of the dynamic memory access is accomplished with no changes to the core synthesis optimiser. The major changes are made to the VHDL compiler, which generates designs that interface to the heap management system - the completely new runtime system.

4.1.1 Generated system structure

The additions made to the MOODS synthesis system result in a structural output design that interfaces to a heap management system. The output structure is not created by the user, but by the MOODS synthesis tool itself. The generated structure is shown in Figure 4.1 below.

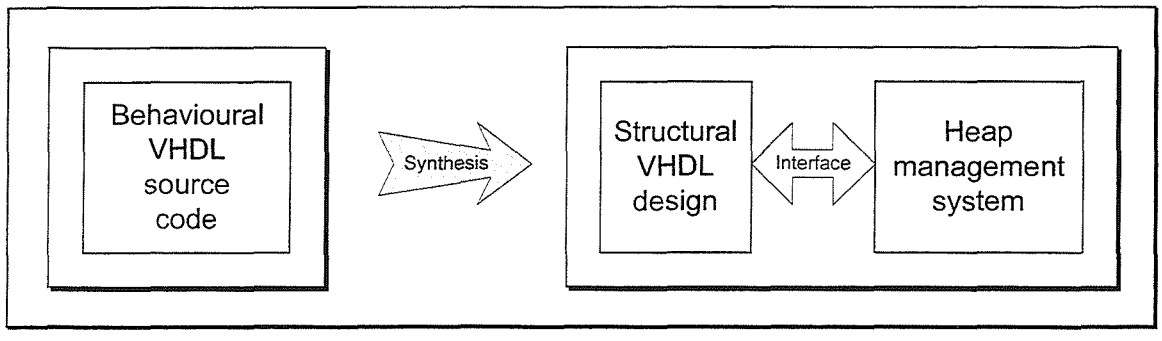


Figure 4.1 Generated system structure

The user provides source code that uses the dynamic memory constructs of the VHDL language. These constructs are then translated into a form that directly accesses a heap management system by a number of interface procedures. The interface ports that link to the structural implementation of the heap management system are added automatically to the entity port list declaration of the generated structural VHDL design. This design then links to the heap management system using the generated port signals, which have direct opposite equivalents in the heap manager.

MOODS generates the structure shown in Figure 4.1 after the modifications described in this chapter were added. The changes required for this included expanding the language capabilities of the original VHDL compiler, and generating ICODE that utilises a new interface into a heap management system that is created as a concurrent runtime system.

The initial concept was to perform most of the system enhancement through modification of the VHDL compiler front-end and provide a back-end linker to automatically link the user's generated structural output with the heap management system component. However, the use of third party RTL synthesis tools allows the linker system to be bypassed, where the netlisting capabilities of the RTL synthesis tool and a top level VHDL linker file are used instead. This means that the synthesis of dynamic memory objects is performed solely by modification of the VHDL compiler and the provision of an interface system and the heap management system into which to interface. No modification to the MOODS synthesis core is necessary. The top level RTL VHDL file contains the structural output of MOODS for any user designs linked as components, with clocking systems and ancillary buffers added as required. The implementation of the linker is left as further work. Details of the initial implementation of the linker are given in Appendix A.5.

4.1.2 Synthesisable VHDL subset enhancement

The VHDL compiler consists of three main phases. The first is to generate an internal representation of the parse tree from the source VHDL. The second phase is to convert this internal representation into a simpler representation in the form of ICODE. The ICODE file is then output by the third stage as a direct representation of the converted internal structures.

The changes to the compiler consist of an increase in the language capabilities with the addition of a more complete parse tree, as the original compiler would not parse the entire VHDL language, only the synthesisable subset required by MOODS. The modifications only increase this subset, not complete it.

The first stage of modification to the compiler is to increase the scope of the parse tree so that the VHDL type constructs of **access** types and **record** types can be generated. Along with these capabilities is the ability to define incomplete types, so that recursive data structures can be built. The **array** type definition also requires enhancing so that unconstrained **array** types can be defined for the creation of arrays with runtime-defined array index lengths. The size of the **array** is defined at the point of allocation in these cases, not at the point of base type declaration.

The ability to parse **access** types requires that the object creation and deletion constructs be parsed also. The ability to gain access to the type referenced by an **access** type is also added by enhancing the name lookup abilities to include the **access** type dereferencing method.

The ability to parse **record** types requires that the extra name lookup for sub-object elements contained by the **record** types be added. A partial lookup is implemented which allows each element to be accessed individually. The **record** element accessing and **access** type dereferencing are performed in a similar way.

4.1.3 Dynamic memory interface

The language parser enhancements are only useful with the capability for translation into the relevant ICODE constructs. All modifications to the translation process enable the ICODE structure and file format to stay the same. This is accomplished by the use of interface procedures (which were already capable of being parsed and translated) that communicate with the heap manager via a number of automatically generated external port signals.

These heap interface procedures are contained in a VHDL package that is automatically parsed from a specified file as input when a design requiring dynamic memory is supplied. Another package is parsed in the same way that defines the size constants to be used

internally within the compiler. These constants define the address path range and data path width of the underlying heap manager. The constants are discussed further in Section 4.2.1.2, with the interface procedures in Section 4.2.1.3 and the automatically generated port signals in Section 4.2.1.4.

4.1.4 Translation into ICODE

The translation of an **access** type static variable is simply an ICODE register defined with the same width as the address path constant defined in the inputted heap constants package. The allocation of an object occurs via a generated call to the allocation procedure defined by the interface package. The number of dynamic data words that an object requires when allocating is calculated from the type of object being created. The size is just one parameter passed into the heap allocation interface procedure.

An **access** type dereference action is either translated into a call to the heap read or write interface procedures, depending on whether the dereferenced object is the source (read from) or destination (written to) of an expression. The leaf left hand side of an assignment operation is the destination, with the items on the right hand side of the assignment forming the source. The basic translation for an assignment operation and a procedure call are shown in Figure 4.2, where '*a*' and '*b*' are both variables that store an **access** type reference of an integer. Note that the translation uses the **access** type variables as addresses, all memory accesses are via interface procedure calls and all address offsets are a constant zero (an integer requires only one memory word for storage). Note also that the '*test*' procedure parameters are of mode *in* and *out* respectively.

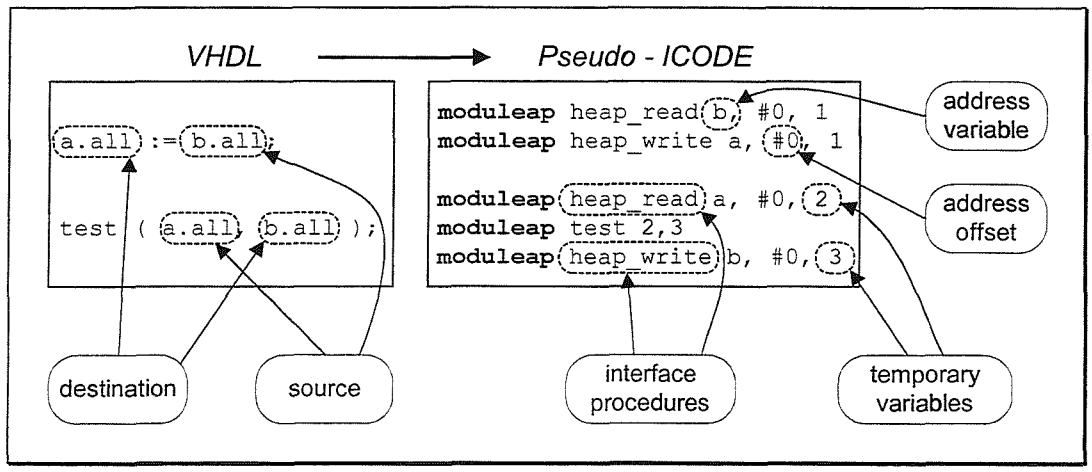


Figure 4.2 Translation of access type dereferencing

The actual interface procedures take the I/O generated by the compiler (shown in Figure 4.2) and also a link to the relevant interface port signals (not shown in Figure 4.2) that are automatically created within the design's external port list when dynamic memory is used. A number of port signals are created, each with a direct equivalent in the heap manager component itself. The '*moduleap*' calling mechanism is described in Section 3.1.6.

Using procedures to interface to the external port signals creates inefficiencies: it is not possible to merge ICODE instructions into the control states used by the generated ICODE module (procedure) with the calling module. This wastes clock cycles. Allowing particular procedures to be inlined solves this problem. By inlining the interface modules into the calling modules, more control states tend to be created with repetition wherever the called modules are inlined, but the savings made in terms of critical path delay reduction and register sharing more than compensate for the increased number of control states generated. In fact, inlining proves to be an optimisation step that has benefits when applied to other procedures outside the heap interface. Inlining is discussed further in Section 4.2.2.

It is possible, even likely, that more than one concurrent process will be created within any one design. If dynamic memory is accessed from two or more concurrent processes, then access to the heap management system needs multiplexing between the multiple processes that use the heap. If no dynamic data is shared between the processes then an alternative to multiplexing the accesses to a single heap is to create a separate heap manager for each concurrent process. The first method was chosen to allow support of shared **access** type variables that can be used to pass dynamic data structures between processes.

4.1.5 Heap management

The heap manager forms the runtime memory control system used by the structural designs created by MOODS. Linking the relevant I/O port list signals from the user's design with the heap manager component forms the interface with the manager. The manager consists of the underlying DRAM controller and the heap management algorithm that utilises the underlying memory.

Having a completely separated heap management system allows for changes in the management algorithm with no change to the original user's source code. This means that

different bolt-on controllers can be used dependent on the underlying hardware available and the speed, size and memory requirements.

The interface is formed at the structural level using only standard bit and bit vector types, with no reference to the original source **access** types. The interface has a 32-bit wide data path and a user adjustable address path width. The address range used in the demonstration designs utilises 1MWord of DRAM, which equates to a 20-bit address path. This is the size of all registers that are formed from the static translation of the original **access** types within the user's design.

The management component is formed from a pre-compiled and synthesised behavioural description. The component is synthesised using the MOODS tool itself. The particular management algorithm chosen for the initial implementation used by a design with dynamic memory is described in detail in Section 4.3.

4.1.6 Summary

The rest of this chapter details the methods used to implement explicit dynamic memory allocation as part of any behavioural design. The major points to remember during this description are that:

- The heap manager component is created as a concurrent run-time system that integrates with the user's design.

- The heap manager is linked to the user's designs via an automatically generated external interface port through which the memory traffic is communicated.

- The VHDL language constructs for explicit memory allocation are used directly, with enhancements to the compiler parse tree and VHDL file parsing for generation of the VHDL types and name dereferencing methods used for dynamic memory. The explicit parse structures for object creation and deletion are also created.

- The translation of the parse tree into ICODE generates calls to the heap manager interface procedures, which are inlined by the final stages of the compiler.

- All translatable types are limited to a maximum of two aggregation dimensions at the ICODE level.

- Concurrent access to the heap is provided for any process within a single design via the automatic generation of a controlled multiplexor into the single external access port, with each process that requires concurrent access having equal priority set by a round-robin sequential interrogation approach.

- One implementation of the bolt-on heap manager is provided that is optimised for limited object sizes, using a 1MWord address space in DRAM.

4.2 Compiler modifications

The modified compiler program flow is seen in Figure 4.3, which shows the consecutive phases that form the compilation flow. The phases are shaded with relation to the degree of modification that was required for the addition of dynamic memory, with the darker phases being modified the most.

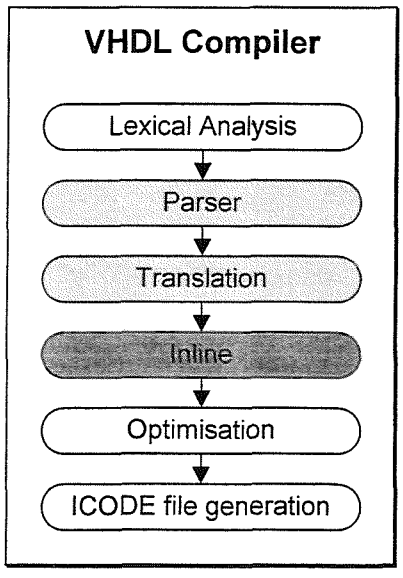


Figure 4.3 VHDL Compiler program flow with inlining

The next sections detail the modifications to the VHDL parser and translator that are required to support dynamic memory. This includes the integration of the heap manager interface procedures where required, concurrent access issues and the inlining of the interface procedures to increase global design efficiency.

4.2.1 Heap manager interface

The heap manager interface is formed from the generation of a number of calls to the various low-level interface procedures defined within the heap manager interface package that is loaded when required. A number of external ports are added to the user's design that have direct equivalent linking signals within the heap manager component that is linked after synthesis. These are discussed in detail in Section 4.2.1.4. These extra port signals are driven by the interface procedures by passing the signals into the interface procedures for modification. The interface procedures also take a number of inputs and drive a number of outputs dependent on the procedure being called.

Another package that is loaded along with the heap interface defines the various sizes of signals to be used by the user's design and the sizes of the external signals that interface with the heap manager component. This allows all generated designs to be translated with compile-time configurable data path widths, relating to the size of the underlying heap management data space. The constants are used during the generation of the heap manager also, allowing configurability in the algorithm.

The heap access procedures and constants are read into the compiler from a source VHDL package that is parsed by the compiler into a number of internal data structures. The compiler automatically loads this package even with no reference in the user's code when dynamic structures are in use. The dynamic interface section of the compiler then uses the given packages to create links to the constants and procedures defined by hard-coded names.

The port signals that are automatically added into the port list of the user's design are manually linked to the heap manager after synthesis. This is performed with a VHDL file that specifies the linkage between the various synthesised components of the top-level system.

4.2.1.1 Communication

The communication protocol used between the user's design and the heap manager is defined both by the heap manager itself and the interface procedures called from the user's design. The interface procedures are the masters of all communication to the heap manager and form an internal abstraction layer within the compiler.

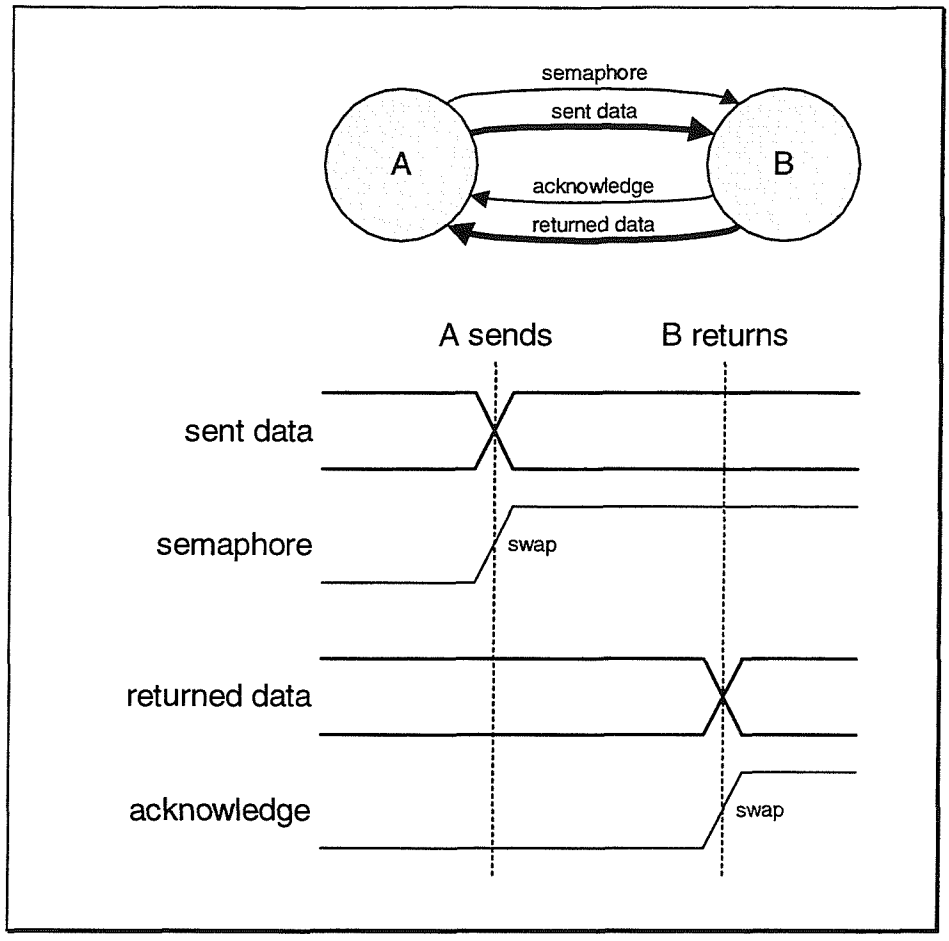


Figure 4.4 Communication between concurrent systems

The underlying communication is based upon the use of a single '*semaphore*' and the return '*acknowledge*' signal, which determine the direction of communication. Both signals are initialised to zero. The user's design initiates all communication with the heap manager by inverting the '*semaphore*' signal. The heap manager responds by inverting the '*acknowledge*' signal. Data flows between the two designs under control of these two signals. The user's design is the master of all communication and the heap manager the slave. All communication is blocking, but designed not to block until a secondary communication is initiated.

Data is transmitted to the heap manager by first checking that all previous communication is finished. A previous communication is complete when the '*semaphore*' and '*acknowledge*' signals are equal in value. The data to be transmitted is set next, followed by the inversion of the semaphore, which initiates the communication. The heap manager, acting as the slave, reacts to the communication once it has finished any previous clean up operations from a previous communication.

The heap manager takes a copy of the incoming data and acts upon it. The data transmitted by the user's design includes the type of action required of the communication (binary encoded), any data being written into the heap and possibly the address and offset of the memory location to operate upon. If no return data is specified for the type of communication, then the '*acknowledge*' signal is inverted straight away, which allows the user's design to initiate a further communication. If, however, a result is required, for instance from a memory read or object allocation, the heap manager performs the actions specified and writes the result into the returned data output. The '*acknowledge*' signal is only then inverted after the returned data is written.

The returned data from the heap manager is copied by the interface procedure into a variable local to the user's design only after waiting again for the two communication signals to be equal. The user's design, dependent on the information returned then uses this local variable in further operations.

4.2.1.2 Heap constants

There are five constant values that are taken from the parsed heap constants package. These define the widths of internal address and data paths and also define the communication port sizes. The heap manager uses the same constants. The example values used here relate to the demonstrators discussed in Chapter 6.

Constant	Example Value	Description
heap_dpwidth	32	Data bus width
heap_adwidth	20 (1M)	Address bus width
object_size_bits	12 (4K)	Maximum allocatable object size
heap_proc_bits	2	Communication control data width
heap_stat_bits	3	Status register width

Table 4.1 Heap size constant widths

4.2.1.3 Interface procedures

The interface procedure communication abstraction layer has four main communication procedures and a single setup procedure that is used to initialise the communication semaphores. Each procedure takes a list of the interface port signals to modify and read in order to form the communication. The other I/O parameters passed into these procedures

form the links into the user's dynamic data and address references. The procedures are listed in Table 4.2.

Procedure	Action
heap_setup	Reset the communication semaphore at startup.
heap_allocate	This procedure is used to allocate an object. An allocation requires that an object size be provided. The size given is a count of the number of memory words required to hold the object being allocated. The allocation returns the base pointer address within the memory space that has been allocated for the object. An allocation is a direct translation of the original VHDL *'**new**'* allocator.
heap_deallocate	This procedure is used to deallocate an object from the data space within the heap manager. The procedure only requires the base pointer address of the object being deallocated. The heap manager knows of the number of words that the object uses, so will deallocate just that single object. The procedure returns nothing. A deallocation is a direct translation of the original VHDL *'DeAllocate'* procedure.
heap_read	The data held by an object is read using this procedure. Dynamic data is read when an **access** type is dereferenced as the source of an expression. The procedure takes the base pointer address that has been previously supplied by an object allocation and an offset into the object, which is calculated from an **array** index or **record** element number. The offset is calculated by the compiler and may be provided as a constant or as a dynamic index. The procedure returns the full data path data found at the given address.
heap_write	Object data is written using this procedure. Dynamic data is written when an **access** type is dereferenced as the target of an expression. The procedure takes the base pointer address and offset in the same manner as the read procedure. The data to be written into the heap data space is also provided upon the data bus. The procedure returns nothing.

Table 4.2 Interface procedures

The user's design will call each procedure as required and initiates every communication as the master system. The first procedure is called only once before any other operations occur within the user's design.

The '*heap_deallocate*' and '*heap_write*' procedures return no data, so the communication interface is designed to return straight away, leaving the heap manager performing the specified operation. In these situations, control will flow back to the user's design, leaving both systems active at the same time. This is an implementation of a level of pipelining in the communication path and enables better memory utilisation speeds rate limited by the

underlying memory implementation, not communication latency. Any further memory operations will be blocked until the heap manager completes the previous operation.

4.2.1.4 Additional ports and variables

There are eleven distinct signals that are automatically added into the port list of the synthesised user's design. These can interface directly into the heap manager port signal list. All communication is controlled by the '*semaphore*' and '*acknowledge*' signals along with the '*control*' signal that tells the heap manager what type of access is being made; allocation, deallocation, read or write. The control data values are binary encoded as '*00*', '*01*', '*10*' and '*11*' (zero to three) respectively for each type of access operation.

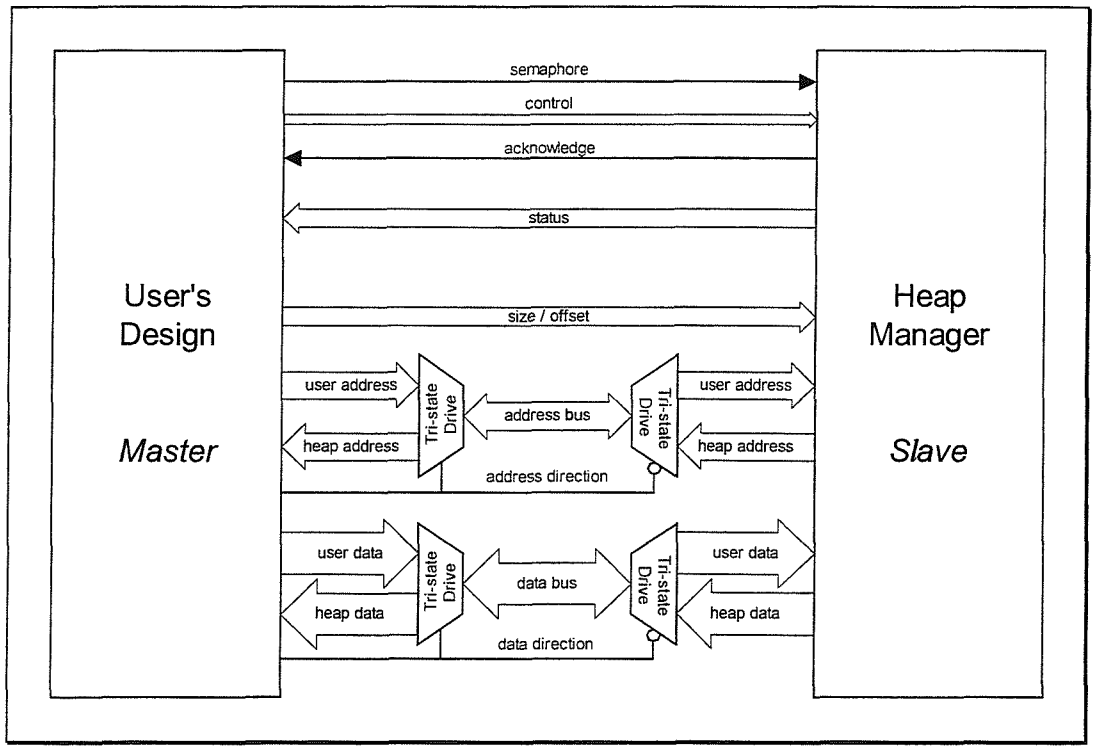


Figure 4.5 Communication port linkages

A status register output is formed within the heap manager and can tell the user's design extra information about the status of the heap, whether it is full or whether an invalid access was attempted. This status output is not currently used by any designs, but is accessible.

A dedicated object size or offset port is created for use with allocation, where it provides the size of object to be created; or for use with read and write accesses, where it provides the offset from the base address into the object being dereferenced.

The address and data paths consist of two separate port signals each. This enables bi-directional data flow, and can be connected directly. However, the restrictions on the number of pins used between separate systems that run on different FPGAs require that both the address and data paths have time-direction multiplexing. This is achieved with the use of two extra controlling signals that specify the direction of data travel along the address and data busses. Each access procedure requires a single direction of data flow for each bus, so no extra communication semaphores are required to control the swapping of direction. The directions for both busses are listed in Table 4.3, along with the use of the size/offset bus. The direction signals are modified at the start of each communication.

Interface procedure	Address direction	Data direction	Size / Offset
heap_setup	n/a	n/a	n/a
heap_allocate	Into user's design	n/a	Size
heap_deallocate	Into heap	n/a	n/a
heap_read	Into heap	Into user's design	Offset
heap_write	Into heap	Into heap	Offset

Table 4.3 Bus use for each interface procedure

To make the underlying use of the interface procedures more efficient, three registers are created within the user's design. These registers are passed into the interface procedures and act as the holding point for the sent and returned data values and address values. The registers are then used by the user's design for further actions upon the data or addresses contained within them.

4.2.1.5 Generating calls

The interface procedures originate from the source VHDL packages that are loaded when dynamic memory is used. This means that the compiler treats them in the same manner as any other procedure defined explicitly within any other source code. The parse structure of the procedure is converted into the ICODE equivalent '*module*' during the translation stage only if marked as used. Procedures are marked as used within the translation process itself, which begins by translating the root VHDL architecture. Procedures are only used if a translated call to them exists. The compiler automatically generates calls to the heap interface procedures whenever dynamic memory is accessed by the source VHDL. Calls to the translated interface '*module*'s are implemented by the ICODE '*moduleap*' instruction just as any normal calls to a procedure are implemented. However, the compiler automatically inserts the values passed into the call. These form links into the additional

signal port lists of the heap interface, internal register variables created to hold a copy of the dynamic data and controlling offset constants and variables.

The mappings of which VHDL parse structures generate calls to the heap manager interface procedures are described in Section 4.2.3.

4.2.2 Inlining

The act of procedural inlining is to replicate the code for a procedure in place of every call to that procedure. Inlining can be implemented at various stages within a compilation and synthesis environment. The implementation of inlining produced for MOODS performs the operation in the compiler, after the translation stage and before the generation of ICODE. The algorithm acts upon and modifies the generated ICODE modules and module call structures. Previously, inlining was tested with MOODS, along with other various source-level optimisations [92,93,94], but the source-level optimisation method is incompatible with the increased VHDL subset used for dynamic memory by the compiler.

The reason for performing the inline operation within a synthesised design is that it allows better sharing of the data path nodes created specifically to perform the instructions within one module. Nodes created for instructions in one module cannot be shared with nodes created for instructions in any other module. Inlining effectively moves the instructions of a called module into the parent calling module, causing one less module to be built.

The drawback of module inlining is that a number of extra control states get created in place of every call to the inlined module. If a module requires a large number of control states, these states are replicated wherever the module is inlined. This can produce large control graphs. However, the benefits of inlining modules that optimise to a small number of control states and perform a relatively large number of operations can be significant. This can include situations where an inlined module is merged into the control nodes preceding and following the call, resulting in a zero time overhead for the call.

Figure 4.6 shows an example of one module '*B*' being inlined into another module '*A*' in two places as replacement for two calls.

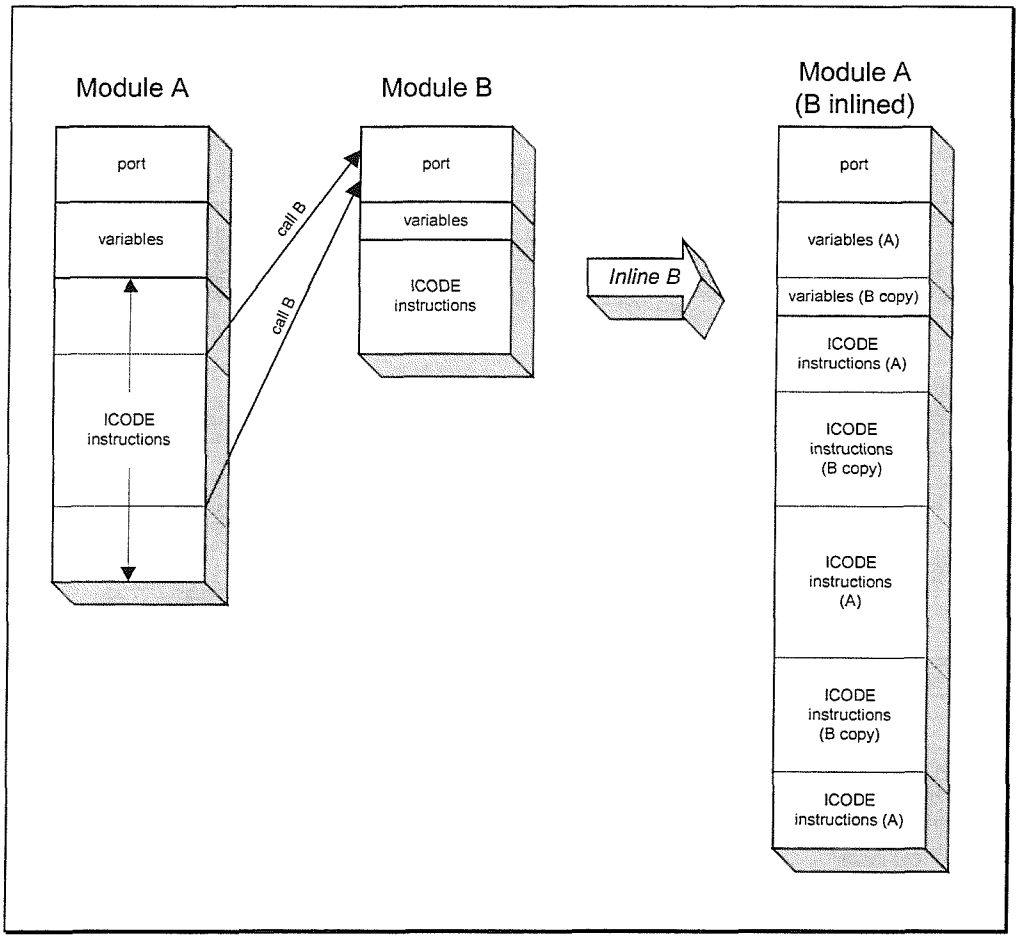


Figure 4.6 Module inlining example

The inlining operation has another benefit of removing the module call instruction, which means that the dedicated control state used for the call is not required. This means that the first instructions in the inlined module may be able to run concurrently with or chained from the last instructions before the call, and the last instructions in the inlined module may be able to run concurrently with or chained with the instructions following the module call. This benefit forms the initial reasoning behind the implementation of inlining, as the procedure calls that perform the interfacing with the heap manager operate more efficiently when they are inlined, as the control state latency between interface operations is reduced.

Any user-defined procedure or function may be inlined. This allows the use of the inlining facility beyond the initial use of inlining the heap interface procedures. The best places to perform inlining are with small procedures that are called frequently and conversely with larger procedures that are called very infrequently.

4.2.2.1 Determining which modules to inline

A procedure is either not inlined or inlined on every call. The method used does not allow selective procedural inlining - a consequence of the selection method used to identify the procedures to be inlined. A call to the '*inline*' dummy VHDL procedure from anywhere within the body of a VHDL procedure or function is detected during parsing phase of the compiler and flags the subprogram for future inlining. The dummy '*inline*' procedure is defined by the MOODS macro operations package that is parsed along with all users' designs. The simulation equivalent of the '*inline*' procedure performs no operations and the call to the '*inline*' procedure produces no ICODE equivalent.

4.2.2.2 Method

A module is inlined after the ICODE generation for the module has occurred. There are four steps to the operation.

The first step is to physically copy the ICODE instructions that form the body of the subprogram. Activations between the instructions are also copied. The module has one starting activation instruction, the '*module*' definition instruction and one finishing activation from the '*endmodule*' instruction. The local variables used by the module are also copied for each parent module having the inlined module inserted. These copied local variables are name-mangled and inserted into the variable list of the parent module.

The second step is to work through the copied ICODE instructions and replace references to all local variables with links to the newly copied local variables. At the same time, all references to any I/O signals defined by the port list of the module being inlined are replaced with links to the variables and constants passed through the parameter list of the call to the module being inlined.

The third step is to physically insert the copied ICODE instructions with all the internal activations after the module call that is being replaced with the inlined version of the module instructions. Breaking the activation from the call instruction so that it now activates the first instruction of the inlined module does half of this. The instruction that the call instruction previously activated is then set as activated by the last instruction in the inlined module. This completes the third step.

The final step for inlining a module at a particular call position is to change the call instruction, the inlined module definition instruction and the inlined module end instruction into dummy instructions that will be optimised away, with all activations linked correctly after the final optimisation stage of the compiler.

If a hierarchy of inlined procedures exists in the source code, it does not matter in which order the procedures are inlined, as any ordering produces the same ICODE structure. The actual order in which the generated modules are inlined is defined by an outer-loop that iterates through every module in the module list, testing for the inlining flag. If found, an inner-loop then iterates through the same module list, checking all call instructions in the body of the inner module. If a call is found to the outer module being inlined then the inlining method described above is used. If a module attempts to inline into itself due to a single-level recursion then this is flagged as an error. Note that recursive procedures can also be inlined, which can serve to reduce the number of procedure levels in recursive loops. The implementation of procedural recursion is described in Chapter 5.

4.2.3 Parsing and translation enhancement

As dynamic memory allocation is a new addition to MOODS, the VHDL language constructs that are used specifically for dynamic memory, and those constructs that are of little use without dynamic memory were not originally supported. These constructs include the **access** type, which is used to reference dynamic memory objects and the **record** type, which is used to aggregate together several unrelated types together into a single parent holding type. Recursive data structures can be built by using these two VHDL type constructs. Recursive data structures are built using circular definitions, which requires that incomplete type definitions have support also. The **array** type definition has been extended so that run-time array lengths may be used with the inclusion of unconstrained **array** types.

VHDL types do not define the data values themselves, but define the style of the contents of the data values held by variables, signals or dynamic references. The VHDL types have no direct ICODE equivalent until they are used by the variables, signals or dynamic references, where they affect the style of the ICODE generated. Enhancements to the parse tree structure for types are given in the following Sections 4.2.3.1 to 4.2.3.4.

Dynamic VHDL objects are created and destroyed explicitly. The parser and translation enhancements for this are given in Sections 4.2.3.5 and 4.2.3.6, along with example translations using the heap manager interface.

The name lookup for **record** types and **access** types is also implemented. The name lookup is termed as *dereferencing*. Both the name lookup and unconstrained **array** types require slight additions to the lexical analyser token list as well as modification to the parser. Section 4.2.3.7 discusses both the parsing and ICODE translation of dereferencing, with an example of both dynamic and static name dereferencing.

ICODE is generated from a depth-first traversal of the parse structure creating sequences of ICODE equivalent instructions from the structure and from resolved links to other parts of the parse tree. Any translation of the use of dynamic objects will result in the need for the heap manager interface described in Section 4.2.1 to be inserted into the design.

4.2.3.1 Access types

The **access** type is the method that VHDL uses to reference objects created dynamically. It does not have any particular representation standard, as the value of any variable that stores an **access** type cannot be read explicitly. This means that any representation method can be used by a system that uses VHDL as the source language. In this sense, any variable declared as an **access** type can be translated into a pointer to a memory location that contains the object data, as this method allows complete referencing of the object.

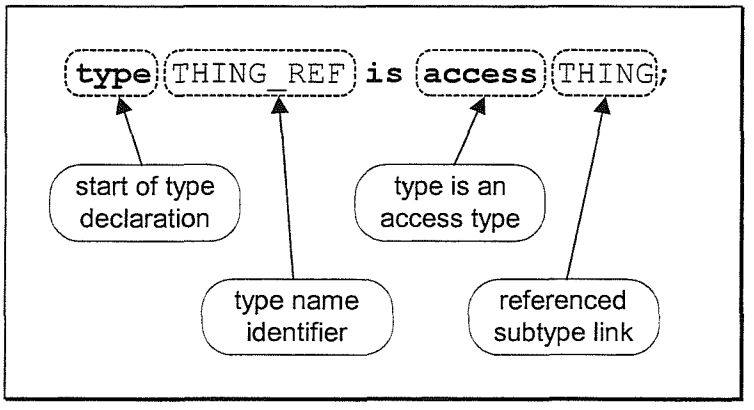


Figure 4.7 VHDL structure for an access type declaration

Every **access** type declaration can only point to one type of subtype object. For every type of object created dynamically, an **access** type declaration is required. As VHDL is a very

strongly typed language, no casting is allowed between types, which means that even similar **access** type variables cannot be cast to point at similar objects.

All **access** types must be stored by variables, signals are not allowed. This means that references to dynamic object types cannot be passed through port declarations of entity-architecture pairs, but may be concurrently shared within a design with the use of shared variables. This means that a design is completely self-contained in terms of dynamic memory.

The internal parse-tree in the compiler is extended to store the subtype information used by the **access** type. Any variable defined as a particular **access** type is dereferenced both by the VHDL '**all**' keyword and in a way defined by the referenced subtype.

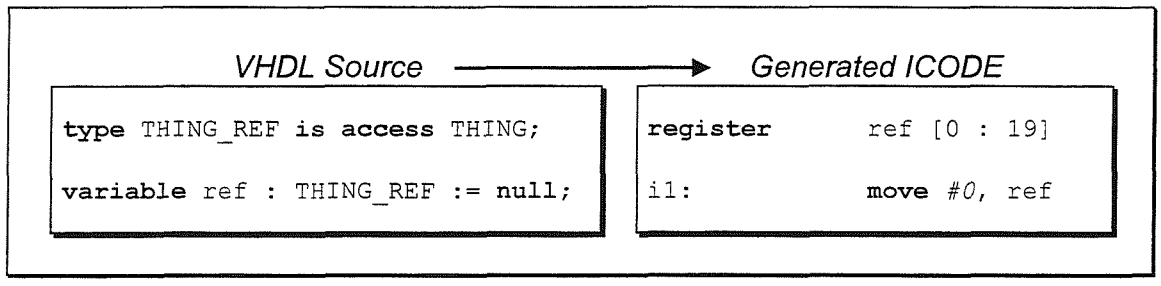


Figure 4.8 ICODE generated for a statically declared access type variable

The example translation in Figure 4.8 shows the static creation of an ICODE register to represent the contents of a VHDL **access** type variable. Note the bit-range of the generated register has 20-bits. This value is derived from the address path width constant, '*heap_adwidth*' described in Section 4.2.1.2. Note also the translation of the VHDL '*null*' keyword into the constant zero.

A dynamic representation of the contents of an **access** type requires the same 20-bits, except that the storage for this data is held in the lower 20-bits of a single 32-bit memory word that is accessed through communication with the heap manager. **Access** types generally require dynamic storage when contained as **record** elements, forming recursive data structures. The dereferencing of objects stored dynamically is explained in Section 4.2.3.7.

4.2.3.2 Record types

The **record** type is the method used by VHDL to group together a number of sub-elements of unrelated type into a single aggregate type. The compiler did not previously parse the **record** type, even though it does not require dynamic memory to exist. Before dynamic memory was introduced, the **record** type would only have been capable of making the source VHDL neater by grouping related items into objects. This was not enough reason for an implementation of the **record** type parse structure. An example of the VHDL code used to describe the declaration of a **record** type is shown in Figure 4.9.

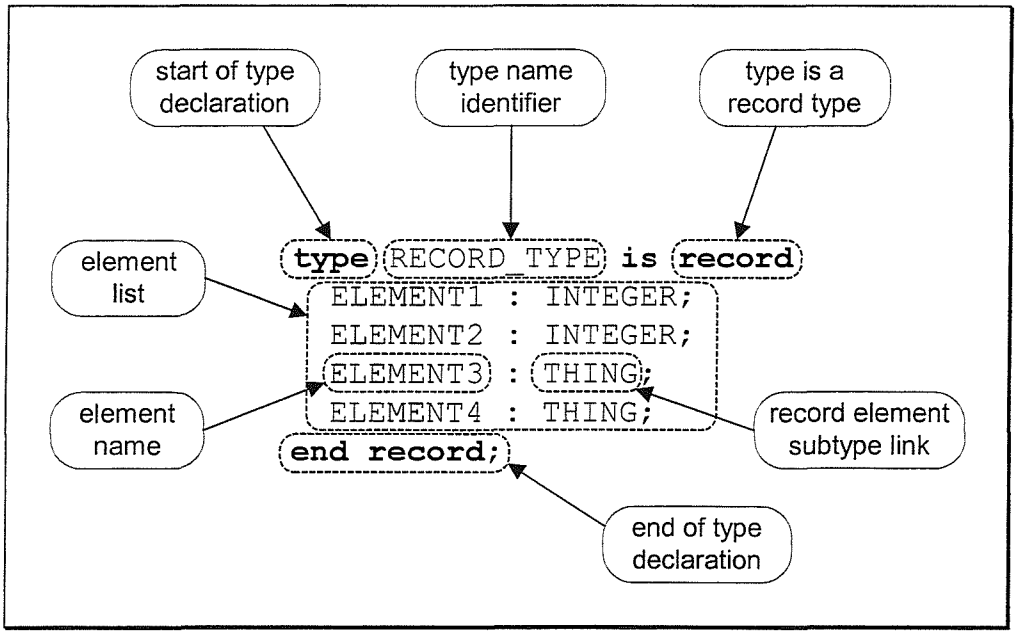


Figure 4.9 VHDL structure for a record type declaration

A **record** type can contain any number of elements of any number of different sub-types. One constraint of the **record** type generation is that the referenced element subtypes are completely defined before the record. It is possible to incompletely define a type, described in the next section, in order that recursive data structures may be built. Each elements subtype is resolved during parsing and an error thrown if the type does not exist.

The internal parse-tree in the compiler is extended to store the element list information used by the **record** type. Each record element has the element name and the subtype link information stored. Any variable defined as a particular **record** type has each element dereferenced by name. This method is shown in Figure 4.10.

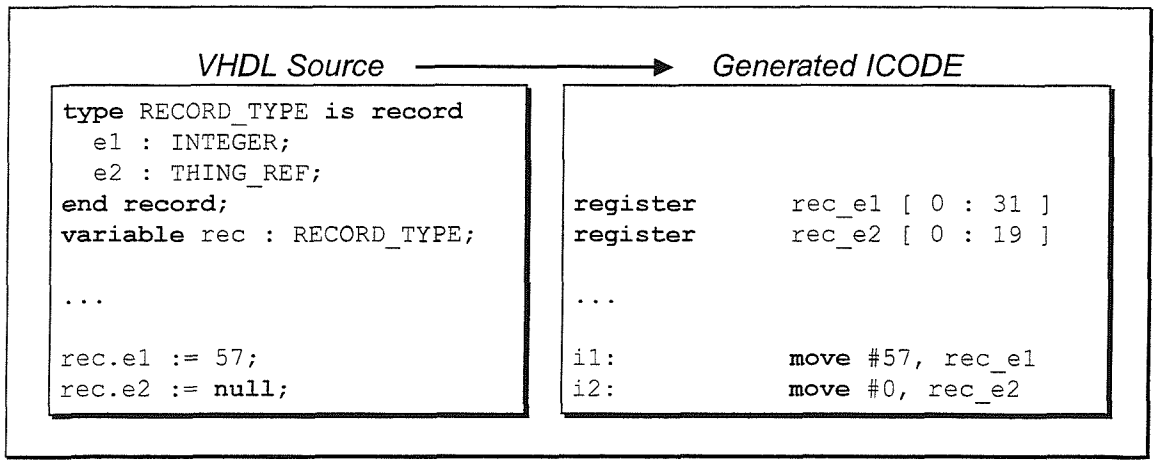


Figure 4.10 ICODE generated for a statically declared record type variable

The example translation in Figure 4.10 shows the static creation of two separated ICODE registers to represent the contents of both elements within an example VHDL **record** type variable. Note the bit-range of the generated registers is dependent on the elements subtype requirements.

A dynamic representation of the contents of a **record** type requires the same number of 32-bit memory words, as there are elements to the record. Each element is accessed one memory word at a time through communication with the heap manager. Each element within the record is assigned a constant offset by the compiler. This offset is with respect to the base pointer reference returned from allocation by the heap manager.

4.2.3.3 Incomplete types

The power of the **record** type is only apparent once a record element contains an **access** type that references another **record** of the same type. Once this occurs, recursive dynamic data structures can be described and generated, which includes complex data structures such as linked lists, trees and graphs.

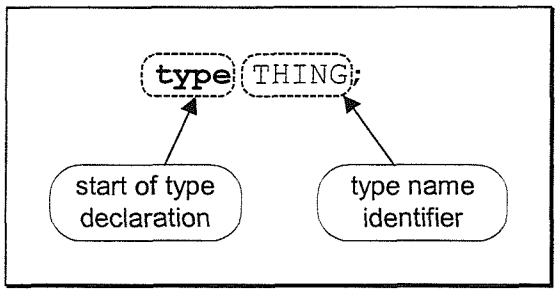


Figure 4.11 VHDL structure for an incomplete type declaration

An incomplete type is simply created from the definition of the name of the type, as shown by Figure 4.11. **Access** types can then be set to reference the incomplete type, even though the type is not fully defined. The type declaration can be completed at a later stage, usually being formed as a **record** type. This situation is shown in Figure 4.12, with the creation of a linked list structure. The recursive nature of the definition is that the '*LIST_REF*' **access** type references the '*LIST*' and the '*LIST*' **record** type contains elements of the '*LIST_REF*' type, forming references to other '*LIST*' objects.

```
type LIST;                              -- incomplete list type declaration
type LIST_REF is access LIST;           -- list pointer type declaration
type LIST is record                     -- list type declaration completed
  nxt : LIST_REF;                       -- next list item pointer
  prv : LIST_REF;                       -- previous list item pointer
  data : THING;                         -- data contained by list
end record;

variable list_base : LIST_REF;          -- static base of list pointer
```

Figure 4.12 Incomplete type declaration used for linked list creation

The compiler implements incomplete type declarations by using a flag on the parse tree node that describes a type declaration. The initial incomplete declaration simply creates the type structure and flags it as incomplete. Whenever a new type is parsed, a check is made for a repetition of the type name identifier. If a type declaration already exists with the given name, then it must be flagged as incomplete as a redefinition is invalid VHDL and an error is thrown. If the type declaration is flagged as incomplete, then the secondary definition fills in the type information.

4.2.3.4 Unconstrained array types

A slight modification to the **array** type definition is required in order to dynamically create different sized arrays from a single type definition. This situation is allowed in VHDL with the definition of the unconstrained **array** type declaration and the ability to pass a sub-range into the allocator when creating a new **array** object.

The compiler limits the use of unconstrained arrays to dynamic allocation and static creation of the array with a constraining range definition at that point. The type can also be used by subtype declarations. The unconstrained base type cannot be used to directly pass information between port declarations or to define internal signals or variables.

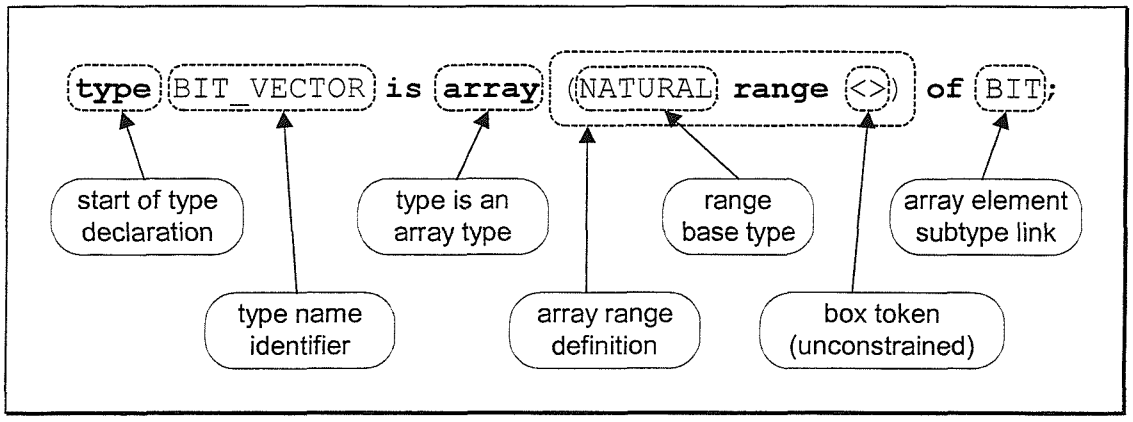


Figure 4.13 VHDL structure for an unconstrained array type definition

The VHDL method for defining an unconstrained **array** is shown in Figure 4.13. The example shows the definition of the standard bit vector type that is an array of the bit enumeration type with an unconstrained array length. The bit vector type is never used directly, but is used by subtype declarations that define the array length.

A flag is set in the parse tree node whenever the 'box token' is given as the defining range of the array. This tells the translator that the type is unconstrained and may not be used directly.

4.2.3.5 Allocation

The VHDL explicit dynamic object allocator uses the keyword '**new**' for the dynamic allocation of all objects. The construct returns an **access** type reference to the type of object passed into the allocator. This is shown in Figure 4.14.

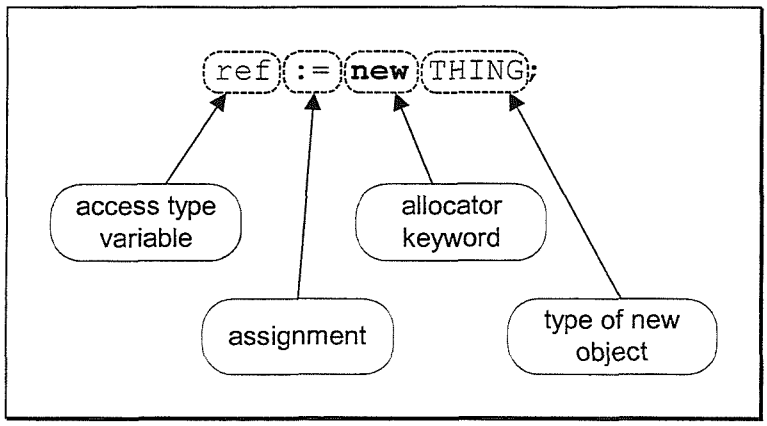


Figure 4.14 VHDL structure for object allocation

The allocator acts at the same precedence level in the parse tree as name lookup (for variable and signal referencing), constant literal definition and sub-expression creation. This means that the allocator is at the leaf of the parse tree in the definition of complex expressions. As such, the allocator returns a result that directly feeds an assignment operation, as seen in Figure 4.14. When allocating unconstrained arrays, the allocator must have a range defined at the point of allocation. This is shown by the second allocation in Figure 4.15, where the range is translated into the number of memory words to allocate.

Figure 4.15 shows an example of the translation of three object allocations. The first is simply the allocation of a single integer, the second is the allocation of an **array** of integers with the array size defined at the point of allocation and the third is the allocation of a **record** type object containing two elements.

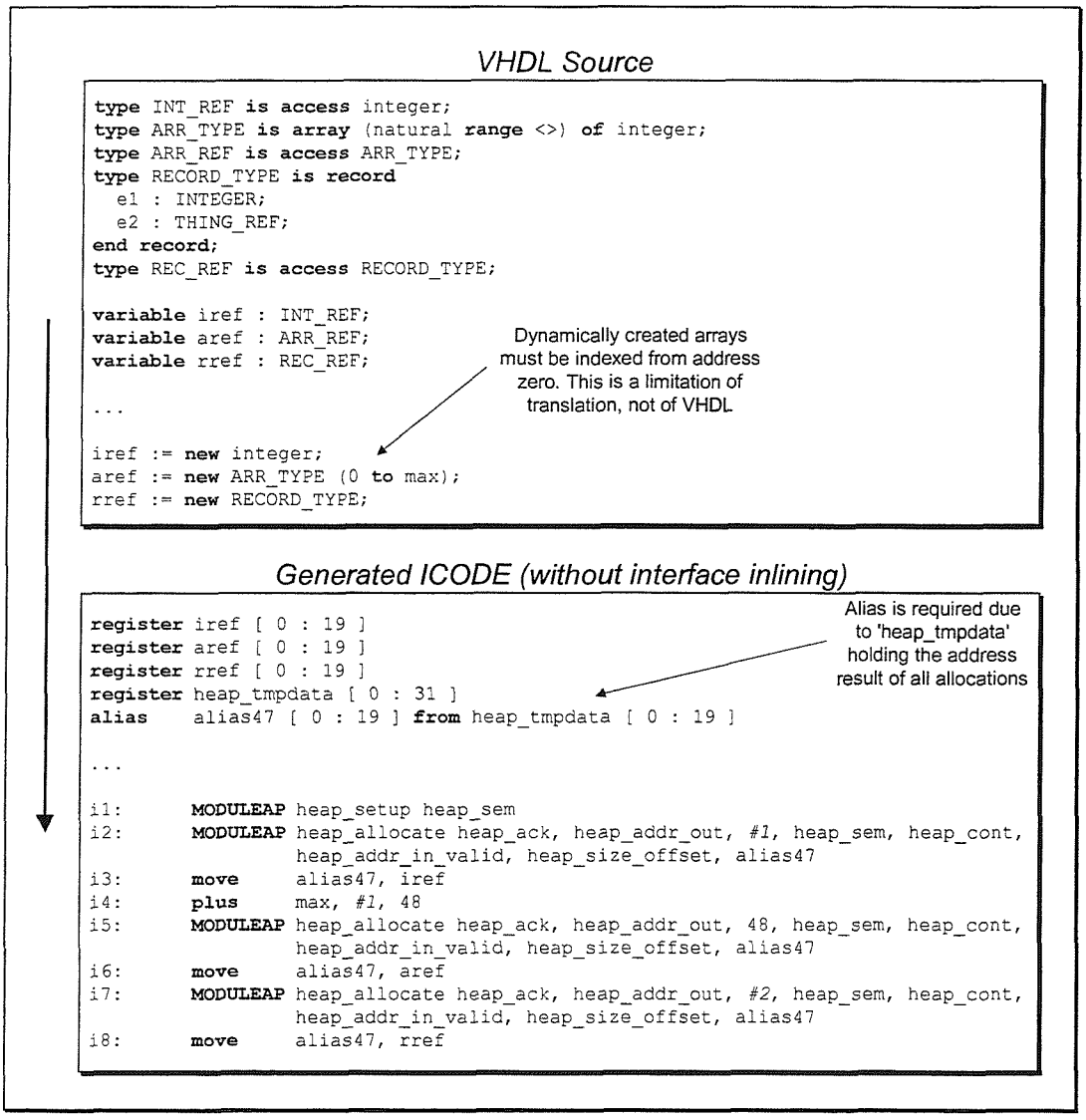


Figure 4.15 ICODE generated for the dynamic allocation of three different types

The three allocations each return references to the allocated objects, which are translated as physical memory addresses by the compiler. These addresses are written into the statically declared ICODE registers in the example, but can also be written into the contents of other dynamically created objects through assignment to a dereferenced **access** type object. Note the generation of the '*heap_tmpdata*' register and the *alias* into the lower 20-bits of the register. This statically declared register is used to temporarily store the returned address from every allocation. Each ICODE '*move*' instruction translation of the VHDL assignment operations move the data referenced by the alias into the actual target of the assignments, '*iref*', '*aref*' and '*rref*'. Also note that the heap interface module calls have not been inlined in the shown example. This is only due to brevity and readability, as each allocation actually performs nine ICODE instructions to every inlined module call.

A translation of the '**new**' operator results in a generated '*moduleap*' call instruction to the heap interface procedure '*heap_allocate*'. The type of object that the allocation operation creates defines the size parameters passed into the interface procedure. The size is given as a number of data words capable of storing the entire object. Various limitations on the types of objects that are creatable are given in Section 4.2.4. Generally, one word is used to store non-aggregate types (enumerations, integers and **access** types), **array**s are stored with the same number of words as there are valid indexes to the array and **record**s are stored with the same number of words as there are elements contained by the record.

VHDL defines that the initial values for objects created dynamically can be set up during allocation with an aggregate assignment. However, this is not implemented, along with general aggregate assignment. It is possible to manually set up the contents of the dynamic data after it has been allocated, one element at a time. VHDL also defines that if no initial values are given to a newly created object, the contents are set to zero or null depending on the types involved. This feature is not implemented due to the increased number of unnecessary instructions that are generated. This means that manual initial values must be assigned after dynamic variables are generated, for the VHDL semantics to be preserved through synthesis.

4.2.3.6 Deallocation

The explicit deallocation of dynamic objects in VHDL is performed by the '*DeAllocate*' procedure that is implicitly defined for every **access** type declaration. The procedure

accepts a single input object reference to be deleted from the memory contents. This is shown in Figure 4.16.

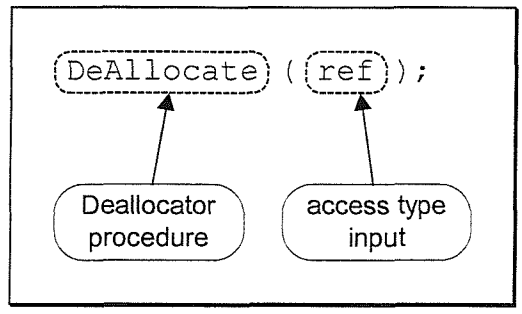


Figure 4.16 VHDL structure for object deallocation

As the parser already handles procedure calls, no modification to the generated parse tree structure is required. However, as VHDL defines that the '*DeAllocate*' procedure is *implicit*, this procedure requires physical insertion into the procedure list used in the compiler. This is achieved by placing a general '*DeAllocate*' procedure in the heap manager interface. Any calls to '*DeAllocate*' within the user's source code then link into this procedure. The translation stage of the compiler uses the knowledge that the implicit nature of this procedure is used to perform direct mapping into the heap manager interface. An example of the ICODE translation of an object deallocation is shown in Figure 4.17.

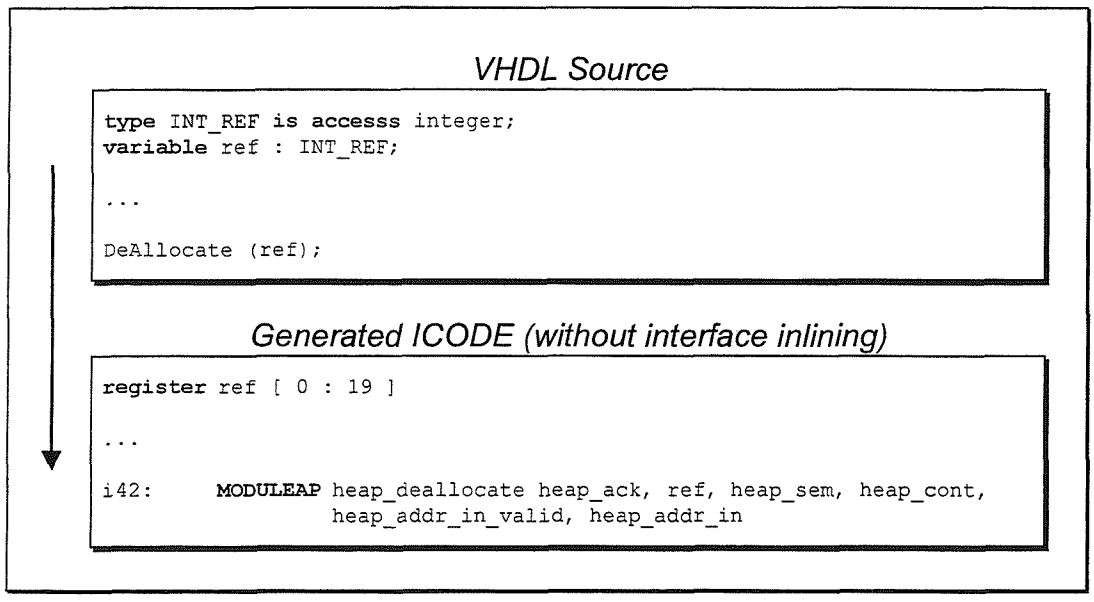


Figure 4.17 ICODE generated for an object deallocation

The example in Figure 4.17 shows the VHDL '*DeAllocate*' procedure being translated into an ICODE '*moduleap*' call to the '*heap_deallocate*' procedure defined in the heap

interface package. Only the base address stored by '*ref*' of the object being deallocated is supplied. This link to this variable is gained from the parse structure that holds the original VHDL procedure call information. The input variable itself could be gained from a statically defined register, or from a temporary result of a previous dynamic memory read access of an object that contains a reference of the object to be deallocated.

```
register ref [ 0 : 19 ]

...

        // ********** inline module heap_deallocate **********
i42:      eq     heap_sem, heap_ack, 487
i43:      if     487                          ACTT i44 ACTF i42
i44:      move   #1, heap_cont
i45:      move   ref, heap_addr_in
i46:      move   #%1, heap_addr_in_valid
i47:      not    heap_sem, heap_sem
        // ********** inline end module heap_deallocate **********
```

Figure 4.18 Inlined ICODE generated for an object deallocation

The single deallocation shown in Figure 4.17 is translated into a single call to the '*heap_deallocate*' procedure defined in the heap interface package. The actual translation of this interface is inlined into the user's source code. Figure 4.18 shows the actual ICODE generated in replacement of the '*moduleap*' call instruction. The deallocation operation requires six ICODE instructions to form the interface. All calls to the heap interface procedures are inlined in a similar way. Note the communication being formed from the checking and assignment of the '*heap_sem*' and '*heap_ack*' signals and the assignment of the '*ref*' input onto the address bus.

4.2.3.7 Dereferencing

The act of dereferencing is to perform a selected name element lookup from within a **record** type item or to access the value at a particular **array** index position or to access the item referenced by an **access** type. Multiple levels of dereferencing may also be parsed, where it is possible to dereference an array element from a dynamic reference to an array, or dereference a particular record element from a dynamic reference to a record item. A fully recursive dereferencing mechanism is supported for all supported types. Examples of the types of dereferencing mentioned are shown in Figure 4.19. The translation of an object dereference forms a read or write operation dependent on the position of the dereference within an expression.

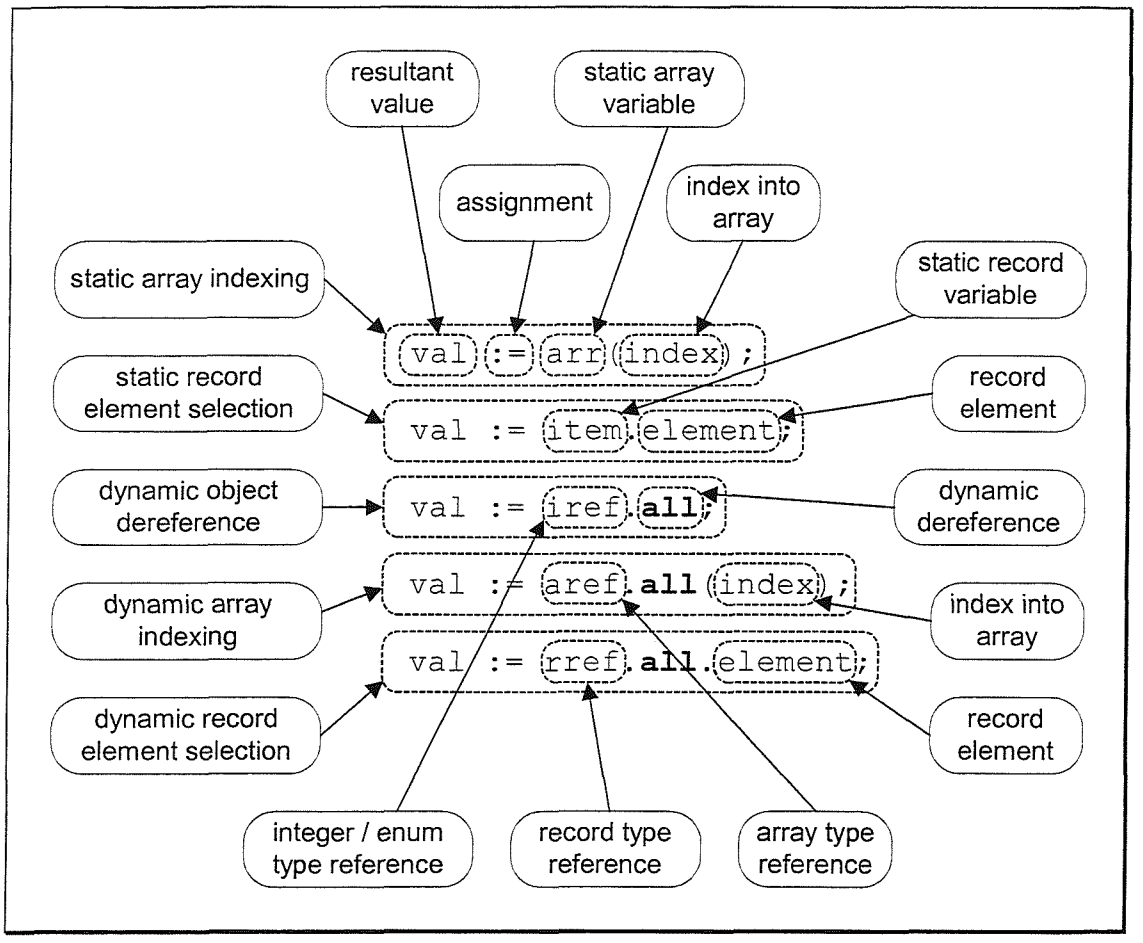


Figure 4.19 VHDL structure for object dereferencing

A name lookup forms a recursive structure within the compiler, with each type of access of a base name formed from either the base name itself or via indexing, slicing or selection of sub-elements within the composite base type. Hence, name lookup is only ever formed from composite types with more than one element (**array**s and **record**s) and from **access** types. Enumeration and integer types do not have sub-elements from which to access. However, these types can form the leaf types of the recursive name lookup.

An **access** type variable is dereferenced using '*<name>.all*', which follows the base **access** type variable name. An element is selected from a **record** type in a similar manner, except that the '**all**' keyword is replaced by the element name within the base **record** type: '*<name>.<element>*'. If a reference to a **record** type is given, then an element within the dynamically created record variable is accessed by first dereferencing the **access** type that points to the **record** object, then selecting the record element by use of its identifier: '*<name>.all.<element>*'. If the element itself is another **access** type, then this can be dereferenced again by simply appending '*.all*' in a fully recursive manner. An example of

this is shown in Figure 4.20, by the destination of the example assignment (translated blocks 4 and 5).

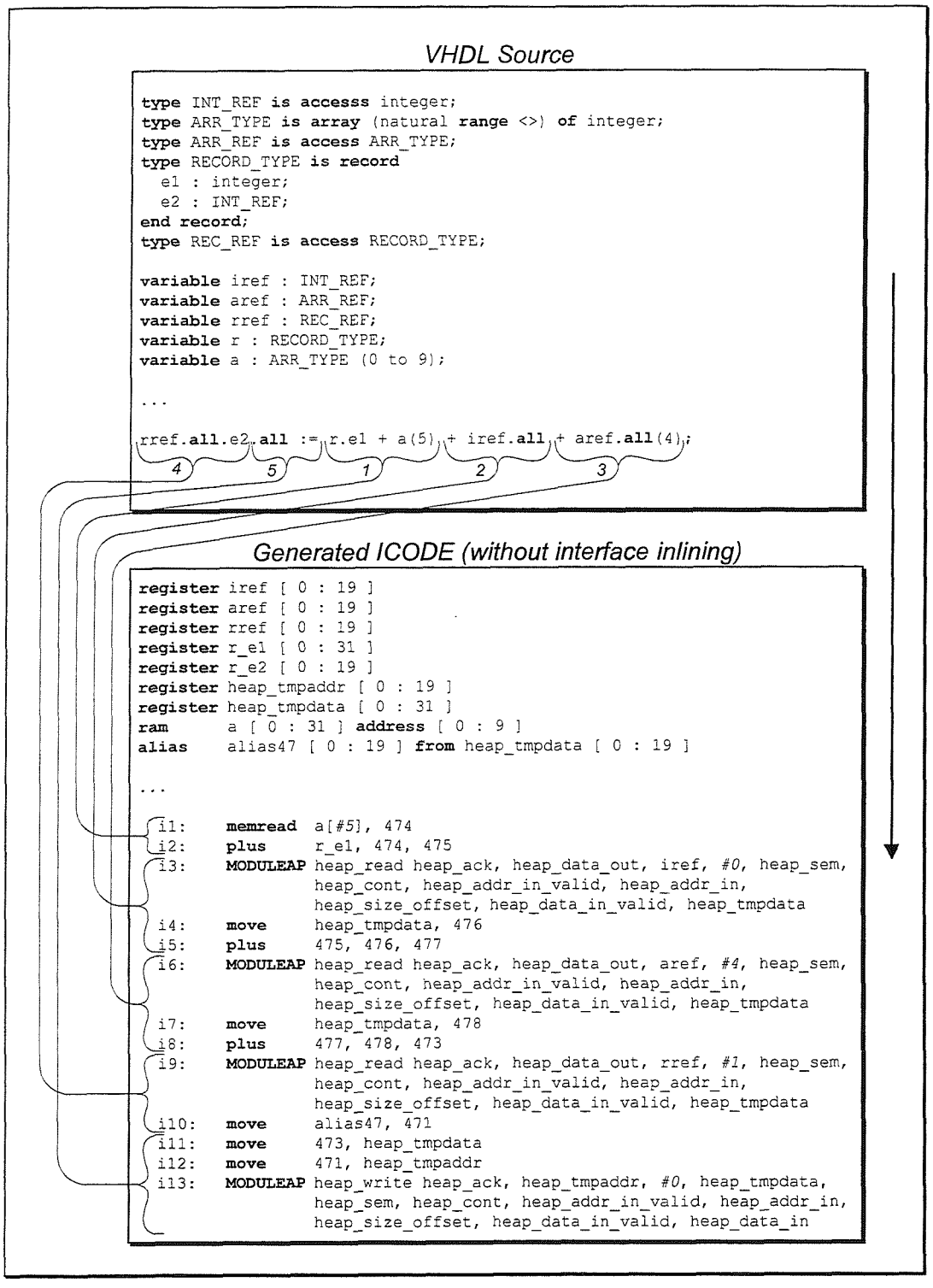


Figure 4.20 ICODE generated for dynamic and static dereferencing

The direction of data flow (whether the data is read or written) is dependent on whether the dereferenced object is the source or target of an operation. Only the leaf left hand side

dereferenced object of an assignment is the target of this operation, which translates into a memory write at the address held by the dynamic variable being dereferenced (translated block 5 in Figure 4.20). Another target of a dynamic memory dereference is as the output port of a procedure call, where a temporary variable is used to store the result of the procedure call, and a memory write performed after the procedure call has finished, which writes the temporary result of the procedure into the dynamic memory space.

The right hand side sources of an assignment expression (translated blocks 1, 2 and 3) or any non-leaf left hand side dereference (translated block 4) or any input ports to a procedure call are translated into dynamic memory read operations of the same form as the write operation for any dynamic object name lookup (translated block 2, 3 and 4). Again, the dereferenced reference-variable holds the base address of the referenced object.

Note that a multiple dereference results in more than one memory operation, where the first operation returns a result to be used as the address within the second memory operation. This situation is shown by translated blocks 4 and 5 in Figure 4.20, where the address held by record element '*e2*' is read before this address is used as the base address of the write operation used to store the result of the assignment expression. Note the use of another generated temporary variable '*heap_tmpaddr*' in this chaining of operations.

As aggregate types are formed from multiple words in the dynamic data space, whenever an item in the aggregate is dereferenced, the offset into the data space that contains the object is calculated from the position of the sub-element in the aggregate. This means that the constant element number in a **record** type is used as the offset, starting from an offset of zero for the first element, incrementing by one for each further element (translated block 4). It also means that an item in an **array** type variable is accessed from the base pointer of the array variable and the offset defined by the given index into the array (translated block 3).

Note the translation of the static variables in Figure 4.20, translated block 1. The 2D static **array**, '*a*' is implemented as a RAM cell and the static **record** type variable, '*r*' has each element translated into separate registers. The ICODE '*ram*' variable is accessed using the dedicated '*memread*' and '*memwrite*' instructions. These instructions take the possibly dynamic index address as an extra input.

4.2.4 Variable dimensions

The use of fully recursive composite type declarations allows variables to be created with any number of dimensions. A dimension equates to one level of aggregate type definition, so any type that contains a subtype in an inclusive way adds another dimension into the definition. The reason that variable dimensions are mentioned is that there are limitations placed upon the number of allowable dimensions within an object for synthesis, which limits the type of objects that can be created. This limitation is not a language constraint. The reason for the limitation is the increased complexity of indexing into multi-dimensional variables. The maximum allowable variable has two dimensions, and by definition, the minimum is without aggregate dimension. The types allowed within the dimensional limits are shown in Table 4.4.

0D types	1D types	2D types
Enumeration with 2 states	Enumeration with 3 or more states	Record containing 1D or 0D element types
	Array of 0D types	Array of 1D types
	Integer type	
	Access type	

Table 4.4 Allowable variable type dimensions

An example of an enumeration with two states is the simple '*bit*', which can represent '*0*' or '*1*'. If there are more than two states that need encoding, the representation requires more than one underlying memory element to store the value. The dimension relates to the translation of the ICODE representation of a variable with given type. This is why the enumeration with three or more states is a one-dimensional type.

The integer and **access** types are both implemented using more than one underlying data bit to store the entire value. In fact the integer uses up to 32-bits and the **access** type uses the minimum number of bits that can represent the entire dynamic data space, which is 20-bits in the demonstrators described in Chapter 6, allowing a 1MWord of data space. An **array** of zero-dimensional types (bit) is also a one-dimensional type.

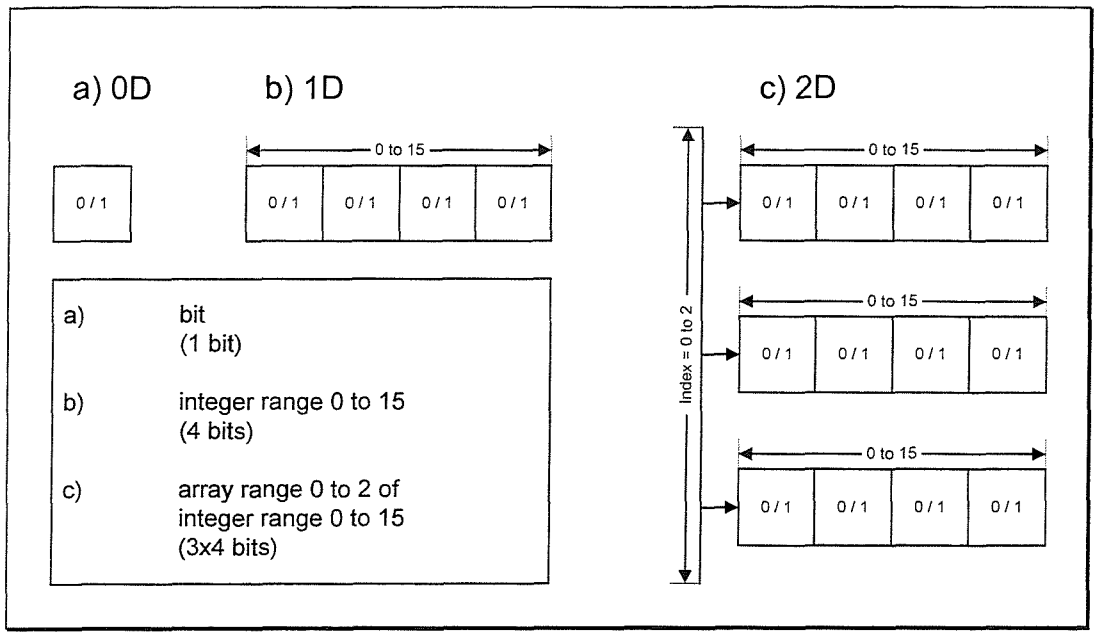


Figure 4.21 Example underlying data structures for allowable dimensions

Two-dimensional types are formed from an **array** of one-dimensional types, for instance, an **array** of integers, **access** types or bit vectors. The **record** type container is always created as a two-dimensional type, which means that any other type cannot directly contain the **record** type. The element types allowable within a **record** type are based upon zero or one-dimensional types only.

The limitations shown are the same for both static and dynamic generation of the variables associated with the types. Whether an object is created statically or dynamically affects only how the object is accessed and manipulated.

4.2.4.1 Dynamic variable storage

All zero and one-dimensional variables will be stored by a single memory word. This places a further limitation upon these types in that they must be able to fit within the data path width specified for the underlying heap management implementation. The implementation of the demonstration management scheme uses a 32-bit data path, which enables full range integers, all **access** types, bit vectors of up to 32-bits and enumerations with up to 2^{32} states to be held. Any space in the available 32-bits that the type does not require will be wasted.

Two-dimensional variables are stored dynamically using more than one memory word. The space required for the dynamic variable is allocated by a single allocation operation.

A two-dimensional **array** uses the same number of memory words as there are elements in the array. The underlying one-dimensional subtype of the 2D **array** follows the same restrictions in size specified for the 0D or 1D variables shown above. All accesses of the array elements by their index will feed the dynamic memory offset port with the index value directly. It could actually be fed from a subtracted version of the index, dependent on the original VHDL definition of the minimum range value of the **array**.

A dynamic implementation of a **record** type variable will use one memory word per record element. The underlying zero or one-dimensional subtype in the 2D **record** follows the same restrictions in size specified for the single variables shown above. Each element is accessed directly by the compiler providing a constant offset from the base pointer that references the dynamic **record** type variable. The offset value is defined by the position of the element within the definition order of the record.

4.2.5 Limitations

Futher limitations other than the general dimensional limit exist for the use of all variables. These limitations are for the use of the 2D type variables.

The maximum number of elements that can be stored by a 2D **array** or **record** item is limited by the maximum object size that can be allocated from the heap in one allocation. This is defined by the '*object_size_bits*' constant in the heap constants package described in Section 4.2.1.2, and is set as 4KWords for the demonstrations.

Only 1D type variables may be passed into procedures, ports and assignment operations. This means that any accesses to a 2D type variable must be performed one element at a time, which reduces the dimensional order of the resultant lookup into a 1D type variable. This means that a static **record** or static 2D **array** may not be passed through procedure ports. However, an **access** type variable that references a dynamic **record** or dynamic 2D **array** may be passed through a procedure's parameter list, as the **access** type is defined as a 1D type.

The VHDL language supports the use of aggregate assignment, which enables multiple items within an **array** or **record** to be assigned by one assignment operation. The compiler does not support this, with manual assignment of each element used instead.

4.2.6 Multi-process access

The VHDL language allows only variables, not signals, to contain **access** type objects. This means that all dynamic data local to a design is contained by the architecture, as the entity port can only contain signals. The dynamic data stored in a design cannot be passed directly through its interface.

Even in the architecture, all dynamic data is limited to creation and use by the various concurrent processes, which form the sequential program flow. Each process is capable of using dynamic memory. As using signals supports communication between processes and the **access** type objects cannot be passed in this manner, it would suggest that even communication of dynamic data between processes is impossible.

However, an amendment to the language that is included in the VHDL'93 standard [5] is the ability to declare shared variables. A shared variable allows variable containers for objects to be defined within concurrent regions of a design. This allows variables that contain **access** types to be declared at the same position as the signals that are used for communication between processes. This allows more than one process access to the same dynamic data structure at the same time through a shared address space [90].

One feature of shared variables is that the simulation behaviour is not completely defined, which could produce different results between different simulators [20]. This is an anomaly in a language designed for exact simulation, designed to give reproducible results. The reason for the possible differences in results between simulators gained from a design using shared variables is the fact that a variable is updated straight away when it is assigned to. If a shared variable is assigned to within more than one process at the same time, the value held in the variable after the two assignments is the last value assigned, as there are no deferred assignments as used for signals. As different simulators may handle the various concurrent processes in different orders, and the process handling order has an effect on the results, the two simulations can produce differing results. This situation is to be partially dealt with in the next VHDL standard [91], by providing a standard wrapper mechanism for shared variables.

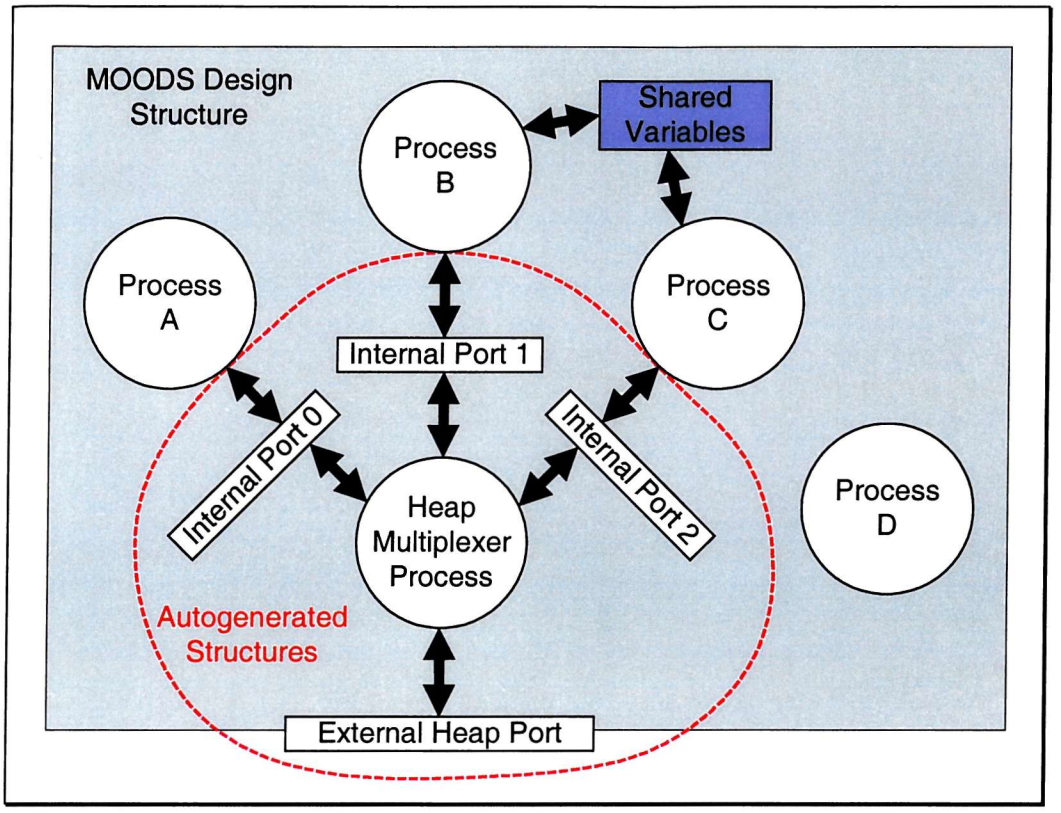


Figure 4.22 Example MOODS design structure with concurrent heap access

A method to cope with the undefined behaviour problem is to control the assignments to the shared variable with the use of other communication signals between the concurrent processes that use the shared variable. This can be accomplished with the use of a semaphore-acknowledge system.

With the ability to share dynamic data structures across process boundaries comes the need for each process to be able to access the same data space that contains the dynamic data. An alternative would be to have a separate heap manager for each concurrent process, which would speed up the accesses of the underlying data but remove the ability for data communication between processes using shared **access** type variables.

The example of the structure created by the MOODS synthesis system is shown in Figure 4.22, which shows three processes (A, B, C) that use dynamic data internally. A fourth process (D) does not use dynamic data at all. Two processes (B, C) communicate with each other using shared **access** type variables. The three processes each access the same heap manager system via a generated heap multiplexer process that controls the sequencing of every concurrent access by making each access follow each other sequentially. Each process has equal priority to access the heap and does this by

communicating with the multiplexer process via a set of internal signals that form a notional port interface. The compiler using standard ICODE instructions just as the original singular interface is generated generates this extra structure.

4.2.6.1 Determining concurrent access

There are three different situations found in a design when checking for the use of dynamic data structures. The first is that no dynamic structures are used, so a heap interface is not required. The second is that one process in the design uses dynamic structures, so a heap interface is required. The heap is accessed directly using the generated external port in this situation. The third situation is that more than one process uses dynamic data structures, which means that a design structure similar to that shown in Figure 4.22 is required.

Determining which situation is found in a design starts in the parsing process of the compiler. Whenever any dynamic data is accessed from a procedure, function or process, the parse tree structure is marked as requiring some form of heap access.

After the initial parsing, the call tree is pre-translated, in that the parse tree is followed for every call made from each concurrent process. Any access found in the tree starting from one process requires a single port into the heap manager. If more than one process is found that contains any reference to dynamic data in the entire call tree that can be called from the process, then concurrent access of the heap is required and a number of internal heap access ports are generated for this purpose.

4.2.6.2 Heap access ports

A heap access port is created for each concurrent process requiring access to the heap. If only one process requires access, no internal heap access ports are generated and the external heap port is used directly by the single process.

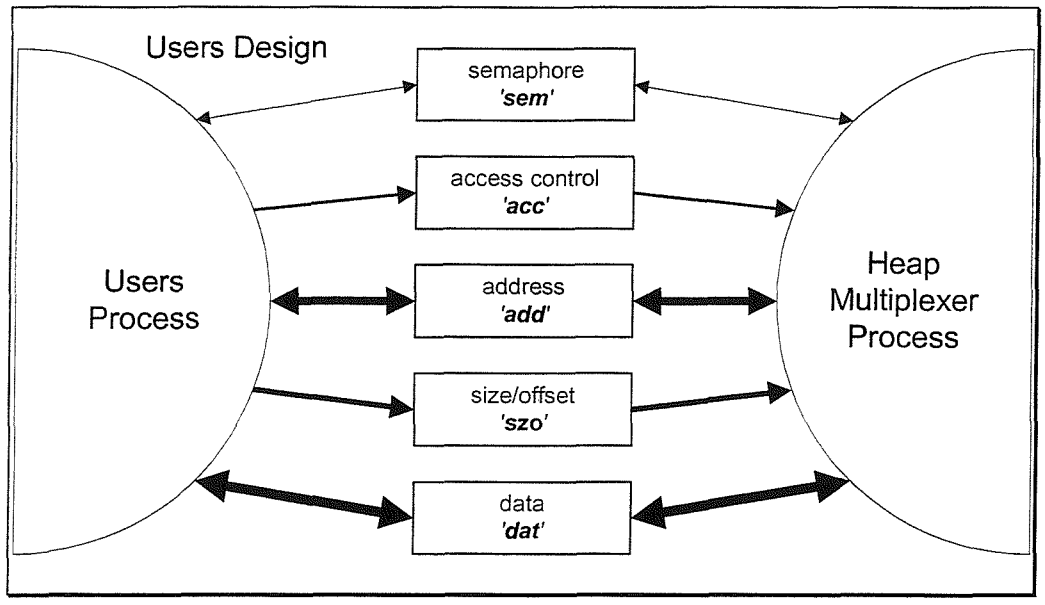


Figure 4.23 Concurrent heap access port

Access ports contain a set of internal communication signals including a copy of the data, address, size/offset and control signals. These have no direct connection to the design's single external heap access port that interfaces directly with the heap manager. Instead, the heap multiplexer process services them. Concurrent copies of the communication signals are implemented so that internal cycle-based speed benefits occur with the ability for each process to start a communication even when the heap is already busy.

The access procedures used by the generated structures of the user's processes are slightly different from the external interface access procedures. When multiple processes using dynamic data are found, the interface procedures for each process are replaced with the procedures that interface with the heap multiplexer process. The interface procedures have the relevant internal heap access port signals passed as parameters. The procedures are loaded from the same heap interface package and are listed in Table 4.5, with the external heap interface equivalent procedures.

Action	External Interface Procedure	Internal Interface Procedure
Initialise	heap_setup	heap_setup
Allocate	heap_allocate	heap_mux_alloc
Deallocate	heap_deallocate	heap_mux_dealloc
Read	heap_read	heap_mux_read
Write	heap_write	heap_mux_write
Service	n/a	heap_mux_service

Table 4.5 Concurrent equivalent heap access port procedures

Note that the '*heap_setup*' procedure is used in both the initialisation of the external interface and the initialisation of all internal interfaces. Concurrent calls to this procedure are possible without replication of the procedure due to the procedure being inlined. Inlining is explained in Section 4.2.2.

Also note the addition of the '*heap_mux_service*' service procedure within the internal port procedure list. This procedure is used by the generated heap multiplexer process and has no direct external port equivalent.

4.2.6.3 Servicing the heap access ports

The heap multiplexer process is completely auto-generated by the translation stage of compilation. The process contains an infinite loop that consists of multiple calls to the heap service procedure. The heap service procedure is called the same number of times as there are concurrent processes accessing the heap. Each call to the procedure has a different set of internal port signals passed into it. As each concurrent heap access port is serviced in turn, this gives all concurrent processes equal priority at access to the heap.

The service procedure effectively connects the internal port signals onto the external port for a limited amount of time. Only one access of the heap occurs in this time. The service procedures are inlined into the process that calls them for efficiency reasons.

4.3 Heap management

The heap manager subsystem that is linked into the designs produced by MOODS has a defined interface as the slave to all communication from the user's design. This system exists to control the underlying memory space and to return address positions within this memory that contain the allocated dynamic objects that the user requires. The heap management algorithm performs the control of the position of the allocated objects.

The implementation of the heap manager has a fixed address space from which to work. As a consequence, there is an upper limit on the number of objects that can be allocated by any allocation algorithm. The initial implementation of the allocation algorithm described in Section 4.3.1 stores the controlling data structures used by the management algorithm in the same data space as the user's data.

In general it is observed [30] that a behavioural design will allocate many small objects of the same size, so a management algorithm suited to this allocation style is implemented. The generated system is written using behavioural VHDL and synthesised using MOODS. The algorithm described is more specialised than the standard allocation methods used in software design, and is shown to improve performance in both the VHDL compiler and the MOODS core software systems when a software version of the algorithm is used as a layer on top of the standard allocation methods.

4.3.1 Algorithm

The algorithm used is both space and speed efficient [37,38]. It is optimised to allocate multiple objects of the same size, which use the same number of data words. The data space is split into a number of *pages* that all start out as initially free. Each page can be used to store objects of one size only, where the size is determined from the first allocation of an object from within the page. The size of object that a page holds is determined at run-time and is dependent on what objects the user's design allocates.

If an object allocation finds that no pages that contain the required object size exist, then a page is taken from the list of free pages and set up to store objects of the required size. The object is then allocated from the data space contained by the newly set up page. Alternatively, if any page is found to contain objects of the required size and the page is not full, then the object is simply allocated from the existing page.

The objects within the page are controlled in a similar manner to which the pages are controlled. If any object is deallocated from a page, the space for the deallocated object is stored in an internal free list. If the page is used again to allocate an object of the same size, the first object in the free list is reused and returned as the allocated object. In this manner, the allocation method is very fast.

There is also very little memory space overhead for the algorithm, as all free lists are implemented within the data space for objects that are free. There is also relatively little header information required for objects, with only a single header structure required for each page, which contains many objects packed tightly together. A single page is also used by the algorithm to store base pointers of all lists of pages with particular object sizes. The first page in the memory space is used for this.

4.3.1.1 Data structures

The data structure style shown in Figure 4.24 shows an example of the entire data structure with a user system actively allocating objects.

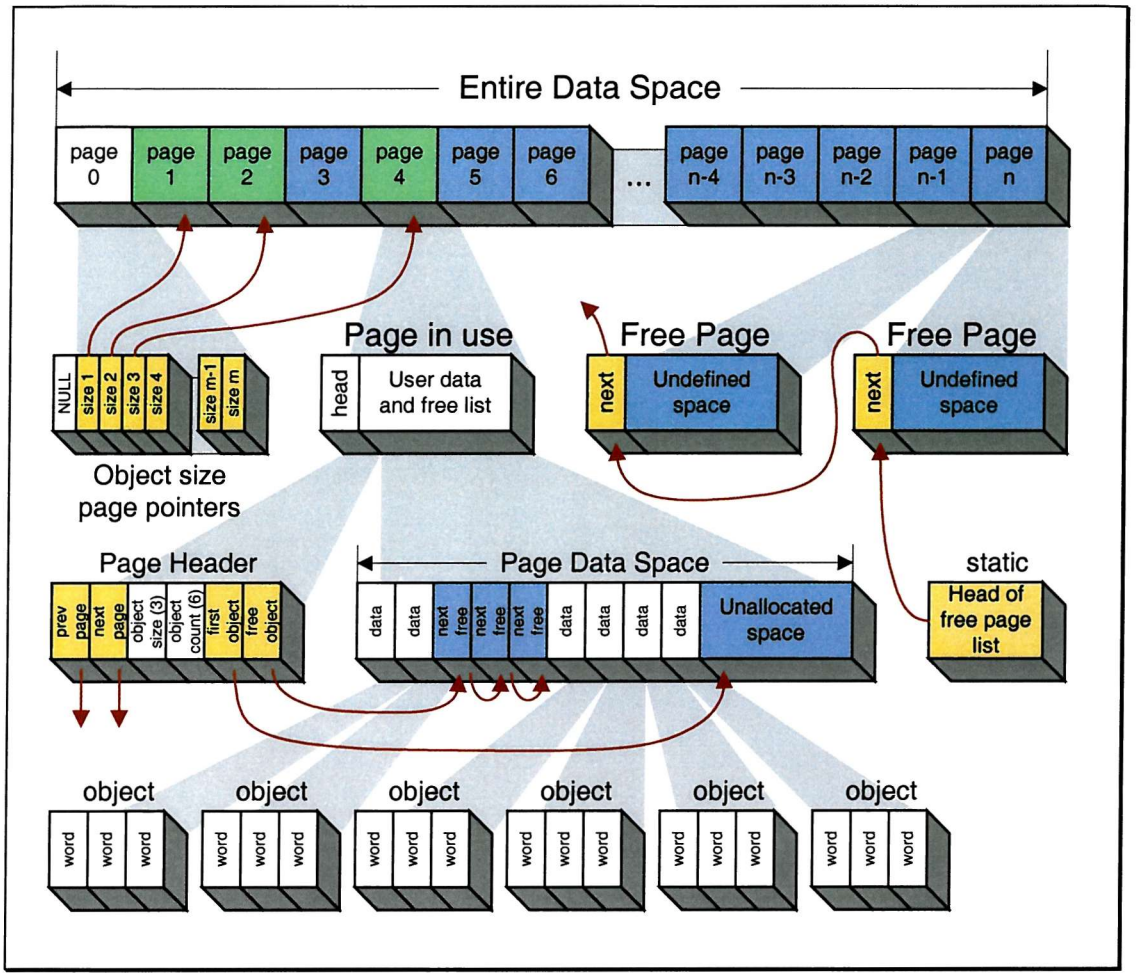


Figure 4.24 Heap management data structures

The diagram shows the formation of the list of free pages starting from a static list head pointer held by the heap manager. All further pointers that define the next free page in the list are contained as the first word of each free page. The last free pages next-pointer is set to null (zero), which signals the end of the list.

Also shown is the first page that is used to store pointers to a list of pages in use with particular object sizes contained. The index of the word in the first page is used as a reference of object size and initially all pointers within the first page are set to null. The list of pages that contain objects of the same size is formed from the active page header, with a pointer to the next and previous pages within the list. The list is actually circularly

linked for ease of page insertion and removal. The example shows a single active page that contains objects of three words size.

The data space in an active page contains a mixture of allocated objects, all of the same size (three words in the example); a free object list, that is created from deleted objects in the page and an unallocated space, from which new objects are allocated. The page header contains information about the size of objects to allocate from the page and the current number of objects allocated within the page. A pointer to the base of the unallocated space and another to the first free object in the data space are also contained in the header. The free object list is formed from the base pointer within the header, with the first word in each free object used as the next pointer of the free list. There are three free objects shown in the example. A null pointer again terminates the free object list. There are six allocated objects shown in the example.

4.3.1.2 Initial setup

The allocation algorithm requires that portions of the control and data space are set up to hold initial values that describe a data space that has a number of free pages, with no pages holding any particular object size existing. This setup procedure is performed once at the beginning of operation of the heap manager.

Setting every word within the first page to null initialises part of the data space. This tells the allocator that no active pages exist for any objects of all allocatable sizes. The free page list also requires initialisation, with each page inserted onto the free list by a loop that works through each page, from page n down to page 1. The first word within each page space is set as the next free page pointer and the free list base pointer is set to point to page 1. The n^{th} pages next page pointer is set to null, which terminates the list.

4.3.1.3 Allocation

Object allocation, as required by the user's design, starts with communication with the user's design. The information provided by the master system is the size of object that is required. The expected result from the slave heap manager is a pointer to the base address of a contiguous block of memory that contains the space for the data object.

The first job of the allocator is to find whether any pages that contain objects of the same size are currently in use. This is done by checking the word within the initial page that points to a list of pages of the particular given object size. The object size page pointer at an offset of the required object size from address zero provides this access to any page that is currently used to store objects of the given size. If a non-null pointer exists at the given size location, then a page has been found. However, if the pointer is null, then no pages that contain objects of the required size exist or any that do are completely full. In this situation, a new page is required.

A new page is taken from the head of the free list of pages. The head pointer of the free list is reassigned to the next free page after the head page by using the next free page pointer from the returned new page. The new page is then set up, with the header information set so the page contains objects of the required size. The page is also inserted onto the active page list by setting the page size pointer within the initial page and setting the next and previous page pointers within the page header to point to itself, forming a circular list. The object count value stored in the page header is set to zero and the deallocated object free list pointer is set to null. The page data space is totally unallocated, with the object base pointer set to the word following the active page header. The page is now completely set up, ready to allocate objects from the data space.

The next stage is to allocate an object from the given page, which could be newly set up or taken directly from the active page list, in which case, objects of the same size are taking up a portion of the data space of the page. An object is allocated from the page by first checking the free list of objects from the page header. If any free objects exist, determined by a non-null pointer to the head object within the free list, then this is the object returned to the user. The object is unlinked from the free list by assigning the free list head pointer with the next free object. This is pointed to by the first word in the object being returned. If the free list contains no objects, then there must still be unallocated space defined at the end of the page data space. In this situation, the returned object is taken from the base pointer of the unallocated space. The base pointer is incremented by the number of words used to store the object. If the increment takes the pointer out of range of the page data space, then the pointer is set to null, meaning that there is no more space for objects in the page.

The allocation of a single object from a page causes the object count value to be incremented. If, after allocation, both the free object list and the unallocated space base pointer are null, then the page is completely full and cannot contain any more objects. In this case, the page is unlinked from the circular list that defines active pages of a particular size, as the list points only to pages that still have space for object allocation.

The base address used to reference the newly allocated object is returned directly to the user's design. Further operations allowed from the address returned are numerous read and write operations or the deallocation operation.

4.3.1.4 Deallocation

The deallocation of an object starts with the provision of the address from which to deallocate from. The operation does not produce a visible result, in that no value is returned. The first operation is to find the page from which the object is allocated. This is achieved by simple address masking, as all pages are created the same size with power-of-2 boundaries.

An object is removed from a page by inserting the given object address onto the free list of objects within the page. This is done by assigning the present free list base pointer value into the first word of the object, and then reassigning the free list base pointer to the address of the object being freed. The deletion process does not touch the unallocated data space pointer. The object count held by the page header is decremented.

With an object removed from the page, the page is now able to hold more objects. The page is empty when the object count-value reaches zero. In this situation, the page is returned to the list of free pages, ready to be used again to store objects of a potentially different size. Before this, however, the page is removed from the active page list for the particular object size. This is done by relinking the pages pointed to from the previous and next page pointers within the active page header of the page being removed, so that the two pages pointed to will now point at each other. If the next and previous pages point to the page being removed, then this operation is not required, as the page being removed is the only one contained in the circular list. If the base pointer of the circular list points at the page being removed, then it is reassigned to another page in the circular list. If no other pages exist in the list, then the base pointer of the list is set to null.

The insertion onto the free page list of a page being removed is achieved by assigning the free page head pointer value into the first word of the page being removed. The free page list head pointer is then reassigned to point to the page being removed.

If there are still objects contained by the page after the object removal operation and the page was previously full of objects, then the page is reinserted onto the active page circular list for the particular object size. A check is made of the base pointer of the list, which could be null. If the base pointer is null, then it is set to point to the page with deleted object. If not, then the page is inserted onto the list by relinking the previous and next pointers of the inserted page with the base page and next page pointed to from the base page. The base page and the next page from the base page then are linked onto the page being inserted by adjustment of their next and previous pointers respectively.

4.3.1.5 Reading and Writing

The addresses that the allocator returns reference the base pointer of the data space allocated for a user defined object. The data in the allocated space is manipulated by direct memory accesses from the user's design. The two memory operations provided are a read and write of a single data word. Both operations are the formed from the translation of a VHDL source code dereference of an **access** type variable. A target dereference performs a memory write and a source dereference performs a memory read.

Objects that are contained in more than one memory word are referenced from the base pointer that is returned from the allocator and a memory offset value that the compiler provides. The offset value, in the case of the described implementation, is simply a direct address offset, so the actual address read from or written to is calculated from the sum of the base address and the provided offset.

A memory read returns a value, so this blocks the communication with the user's design until the result is read from the underlying memory, whereas a memory write does not return a value, so it resets the communication semaphore before the underlying memory write occurs but after the address and offset have been registered. This allows the user's design to continue processing any other operations that follow the write.

4.3.1.6 Limitations

There are two major limitations with the implemented algorithm, due mainly to the allocation method itself. The first limitation is that only a limited set of distinct object sizes can be allocated at any one time. This is due to the limited number of pages from which to allocate objects. As each page may only contain objects of one particular size, the maximum number of object sizes that can be allocated is the same as the number of underlying pages. If more than one page is used to store objects of the same size, then the number of available sizes is reduced further. This limitation is only a problem if completely dynamic objects such as arrays with run-time length definitions are used.

The converse of the limited object size numbers is the limited maximum size of the allocatable objects. As all objects are allocated from within a page, the page size forms the upper limit on the maximum object size that can be allocated. Again, this only really affects dynamic **array**s, with the object size determined by the number of elements within the array. A dynamic **record** cannot realistically reach this upper limit, as most records are formed from a composition of relatively few elements compared to array objects.

A trade-off is made between these two limitations at compile time. As all objects are allocated from a fixed address space, the trade-off is made between the number of pages within the address space and the size of each page, where a doubling of page size reduces the number of pages by half. The underlying address space can also be varied in the same manner, but is more dependent on the underlying storage mechanism, which in some cases is fixed before the user's design is built.

4.3.1.7 Advantages

A major benefit of using the described heap management system is that the controlling structures allow for implicit reallocation of objects. This means that if space is available within a page for an object with particular size, the object will be allocated from that page.

The traditional problem with dynamic memory control is that the memory becomes fragmented, with objects being deleted from random locations producing a memory map with spaces difficult to reuse. This problem does not generally occur with the described system due to the limited object sizes in a memory region giving close proximity between

similar objects. The memory space will become fragmented, but the problems normally associated with fragmentation are reduced due to the object space reuse.

The algorithm is both speed and space efficient: the allocation of objects takes a maximum of twenty-four memory accesses (twelve reads and twelve writes). This forms all of the setting up and list manipulation for each allocated object. Deallocation takes a maximum of sixteen memory accesses (nine reads and seven writes), but returns control to the user before the first memory access. The objects are densely packed into the memory space because of the limited number of header structures required for the algorithm and the embedded free lists effectively take no memory space, as the free objects themselves hold the structure required of the free list.

4.3.2 Implementation

The system for the addition of dynamic memory support for designs produced by behavioural synthesis described in this chapter is supported by a physical implementation of various demonstration designs. These designs are described in Chapter 6 with further details contained in Appendix C. The heap management algorithm described in Section 4.3.1 has a physical implementation also. This is described in Chapter 6 and Appendix C. All of these designs are written using behavioural VHDL and synthesised with MOODS.

The underlying memory space that is controlled by the heap management algorithm is realised by a fast-page-mode DRAM. This type of memory [95] requires constant refreshing and is accessed via a multiplexed address path. The sequencing of the controlling signals that drive the DRAM is performed by one process in the heap management system.

The signals defined in the port list of the behavioural description of the heap management system that are passed between the systems are described as simple vectors of bits. As all ports, signals and variables are converted into the standard '*std_logic_vector*' representation [96] within the final structural design produced by MOODS, no conflicts of type are found from using **access** type variables in the user's source and the '*bit_vector*' representation used by the implementation of the heap manager.

Behavioural simulation does not require the behavioural description of the heap manager in order to simulate, as the user's description uses language constructs that can be simulated directly. A structural simulation of the design produced by MOODS however, will require the structural representation of the heap management system in order to simulate fully.

4.4 Impact on optimisation

As there is no modification to the optimisation core of MOODS with the implementation of dynamic memory allocation, no fundamental changes to the optimisation methods are carried out. However, the different code style that is generated by the modified compiler when dynamic memory is in use affects the optimisation process indirectly.

4.4.1 Inlined interface procedures

The heap management system and the user's design communicate via interface procedures that are inlined into the translation of the user's processes and subprograms. Each communication uses at least one clock cycle as a looping control state that checks and waits for the communication semaphore signals to become equal. Another looping state is required by communications that have a value returned (allocation and the memory-read accesses). This forces a control state between the operations performed before the loop and the operations that follow the loop. As the called interface procedures are inlined, the sequential nature of the control flow will actually allow some operator chaining and control state sharing for independent instructions. This produces tighter control flow with greater utilisation of control states. As the interface is relatively simple, with only read and write operations of the external interface ports and wait constructs forming the interface procedures, the time taken for the synthesis of the interface is relatively small, but is dependent on the number of dynamic operations found within a design.

4.4.2 Heap manager component

The heap manager is formed from a separate external component with its own separate optimisation run. This means that the optimisation process for the user's design is not slowed down by the synthesis of the heap management system for every synthesis run

performed on it. The only extra synthesis time for the user's design is taken with the synthesis of the interface to the heap management component.

4.5 Error handling

There are various stages at which errors in a design can be caught. The errors may stem from synthesis limitations, source design errors or from a new set of errors that manifest themselves from the runtime environment of the heap manager.

The constraint of the heap management algorithm that limits the maximum size of object that can be allocated is generally caught during compilation. The only variables capable of doing this are formed from multidimensional composite types. In fact, the only type really capable of exceeding the size limitation is a 2D **array**, where the number of elements in the array could exceed the given limit. In the case of the composite **record** type, the number of memory words used is directly related to the number of elements in the record. When the size of object is known during compilation, the check is made there. Compilation will fail in this case, giving a relevant error message. If the size of the object being allocated is defined by some runtime variable, then a check of the range of the variable that is used to determine the range of the dynamic variable being created is made. A warning is given during compilation in this case.

Another source of possible erroneous execution is with the pathological user design errors, where incorrect user code is the cause of an error that leads to a memory over-write of the heap management data or user data stored by the management system. Possible sources of errors are generally found from accessing objects that have been deleted, which is possible by having multiple references to a single object. The object free-list may be corrupted by this action. Another source of error is when an object with a null reference is accessed. This could over-write the base page object size table, leaving invalid page pointers in the map. The error leads to undefined behaviour and will be caught by post-synthesis simulation [20], which should always be undertaken before the physical system is implemented.

The memory system that is controlled by the heap management algorithm is of limited size. This means that a user's design could attempt to allocate a number of objects that cannot fit within the memory space available. This is a runtime error and is handled by

simply returning a null reference from the allocator, which is the correct VHDL response to an allocation that fails. No explicit handling of the error occurs. It is left to the user to handle the return of a null reference. Any failure of the allocation will return a null reference. Even if there is space available in the heap, it is possible to return a null reference due to the limited set of pages from which to work from, and the defined size of objects in each page.

4.6 Alternative implementations

After initial implementation, it became apparent that some modifications to the interface methodology could produce designs that are optimised by MOODS in a faster manner, producing faster and smaller implementations.

When the interface with the heap manager is created using dedicated ICODE instructions for each heap access, there is more scope for ICODE optimisation. The actual physical interface is then generated as a post-processing stage of the MOODS core instead, from a direct compiler translation. The implementation of the physical communication contained within the interface procedures is then created using the expanded module methodology [3], and all extra communication ports and concurrent process multiplexing is added after the main synthesis process.

With the heap interface now formed from a set of dedicated ICODE instructions, rather than from the indirect '*moduleap*' calling mechanism, ICODE optimisation for common sub-expression removal can lead to a reduction in the number of memory accesses generated by a direct translation of the user's source code. This leads to faster runtime execution with removal of some memory accesses in a trade off that generates extra registers to hold temporary results from the heap.

Due to the fixed data path used by the heap management system, it is more than likely that some data objects are stored by only parts of the data word. This inefficiency leads to a lot of wasted memory space. A reduction of the wasted space can be achieved by careful design of the objects stored by the heap, where physical data is manually shared within a single 32-bit word, which is the current size of the dynamic data word. A more automated method is to have some form of automatic data packing for increased utilisation of the underlying storage. This optimisation provides a trade-off between the multiple memory

accesses required for a write becoming a read-modify-write (to keep the other data bits in the memory word valid) and the amount of wasted space in the memory. A read-modify-write access is then required in the heap interface; with the underlying DRAM storage capable of performing this action faster than a separate read then write.

The heap manager itself can have various modifications made, which optimise the system with respect to the data allocated by the user's design. Different underlying memory sizes and types can be chosen and different data path widths form the various parameters that can be optimised for the purposes of a user's design. SRAM based caching [97,98,99] or more dedicated use of SRAM [100] by the algorithm can generate trade-offs between the extra memory overhead and the allocation and deallocation speed.

Chapter 5

Recursion

Once explicit dynamic memory allocation is integrated into MOODS, the ability to create recursive data structures is gained. To complement this, procedural recursion is integrated, as this addition allows a greater level of behavioural abstraction for the manipulation of the dynamic recursive data structures.

The rest of this chapter deals with the specifics of how procedural recursion is integrated into MOODS. Section 5.1 gives an overview of procedural recursion and on the general methodology for integration. Section 5.2 details the specific VHDL compiler modifications and changes to the ICODE file format to pass recursion-specific information into the MOODS core. Section 5.3 shows the modifications to the MOODS core and to the synthesised structures generated by the system. Section 5.4 shows detailed timing for recursive procedure calls, while Section 5.5 details the impact on the optimisation process of the MOODS core. Finally, problems associated with the method of integration are shown in Section 5.6, with possible solutions given.

5.1 General overview

Procedural recursion occurs when a procedure is reachable from itself by following all possible calls and indirect sub-calls from that procedure. The result of recursion in a sequential language/system is a loop in execution flow for the design through the call graph, where every iteration of the loop has a new set of iteration-local variables (the local variables of the procedure). This is where the power of recursion as a high level technique is gained, as no explicit dynamic stack of information is described by the source code; instead, the language infers it from the call structure. This can result in much smaller and easier to understand source code than the explicit iterative technique.

Figure 5.1 shows six procedures being called from a single process. The arrows denote a procedure call, with the called procedure pointed to. Where any call-loop between procedures is found, the procedure being called must be capable of recursion. These recursive procedures are shaded. Note that procedure A calls itself directly, forming the tightest recursion loop, whereas procedures E and F call each other, creating one level of indirection. Also note that although procedure B is called from the recursive procedure A, it is not recursive, as it can never be entered again from its position in the call stack.

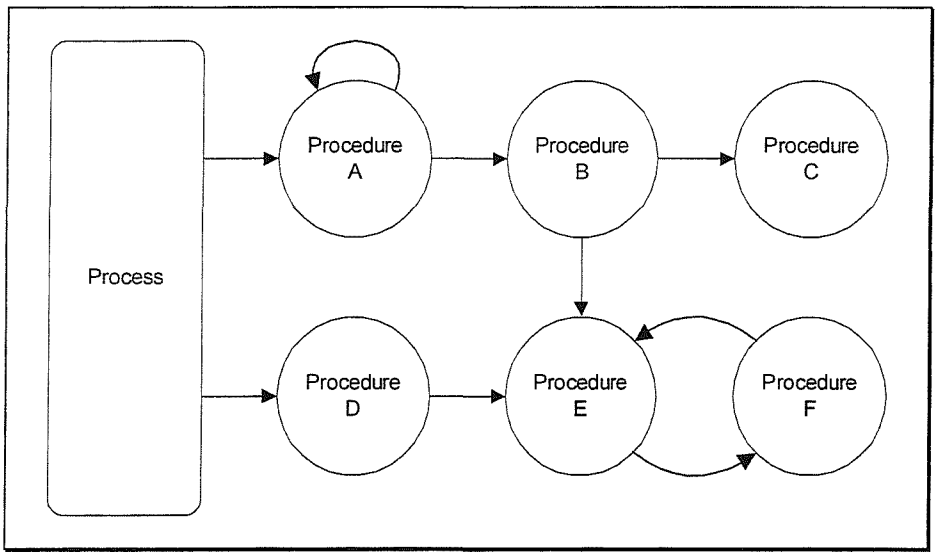


Figure 5.1 Recursive procedure loops

Unfortunately, recursion does not lend itself to behavioural synthesis in a straightforward manner, as all procedures are effectively statically created using a fixed control and data path [101], with local variables mapped onto static registers, which makes each procedure non-re-entrant.

The task of integrating recursion is essentially the conversion of each procedure capable of recursion at run-time into a re-entrant procedure, by providing automatically created call-stack dependent local data and all necessary controlling structures for this data.

5.1.1 Language implied storage requirements

Any language that allows recursion automatically implies a dynamic storage method for all the local variables within its procedures. In the case of software languages, the method chosen to implement this data storage is the stack [77], which is formed from a single contiguous block of memory for each concurrent call tree. This method is chosen as it

suits the calling mechanism of sequential languages, that of the call-return pair, where local storage is created on the top of the stack for each procedure call and then thrown away when returning from the procedure. In software, the stack is typically also used as a method for passing input and receiving output parameters from the procedure calls and for returning control to the correct calling procedure (a procedure could be called from multiple places, only one of which is valid at run-time).

The stack is a very quick dynamic data structure to use, as it only requires a stack pointer to give the present frame into the stack for the currently active procedure.

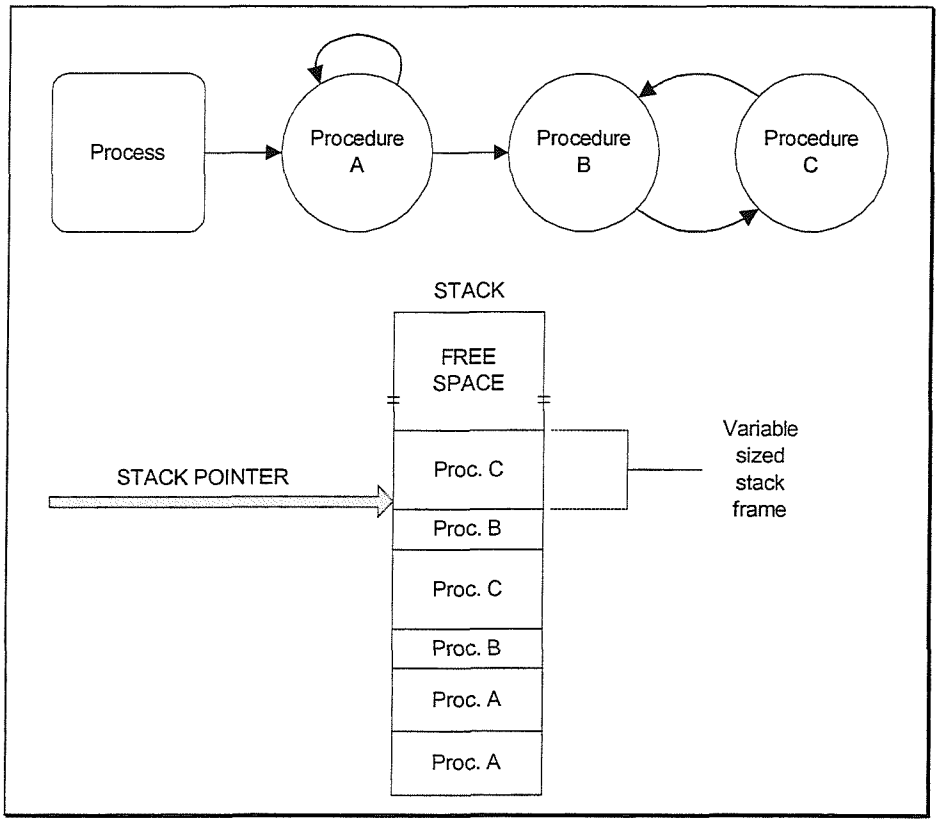


Figure 5.2 Procedure stack

Figure 5.2 shows a procedure call structure and an example stack image working from the bottom of the stack upwards. The stack is a contiguous block of memory with a single pointer acting as the stack frame reference. Note that each procedure can have different sized stack frames dependent on the number of variables in the procedure that require dynamic storage. Each variable is stored at a different offset from the base stack frame pointer.

5.1.2 Original procedure call methods

An explanation of the methods used for procedure calling before the additions made for recursion is necessary, as parts of the underlying system are used by the recursive procedure call methodology and the old method is still used fully when no recursion is detected in the user's source code. The old calling method is still used where possible, as the structures generated for this are simpler and smaller, with recursion requiring extra resources (both execution time and design area) for the dynamic memory storage.

5.1.2.1 ICODE modules and calling method

The user's design is converted into ICODE by the VHDL compiler. A procedure is represented in ICODE by the instruction '*module*'. Any converted procedures or functions in VHDL are translated into ICODE modules (unless they are explicitly inlined). The module contains information about its I/O ports in its header. Local variables are also specified locally in the module, along with all temporary variables used by the module.

After the I/O and variable declarations come the ICODE instructions themselves that form a completely contained flow of control for the module. The only exception to this is the '*moduleap*' instruction, which is the ICODE method of calling other procedures. This instruction contains a list of I/O parameters that are mapped onto the called procedure's I/O ports. This map could be different for different calls to the same procedure, so some method is required at run-time to determine which parameters to use. An example of generated ICODE is shown in Figure 5.3.

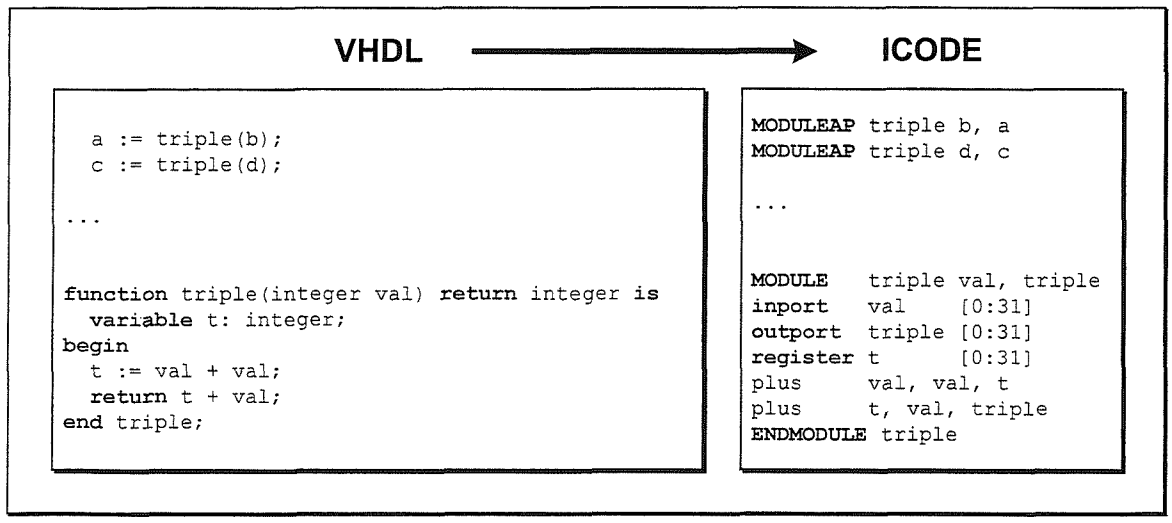


Figure 5.3 VHDL function translated into ICODE module

5.1.2.2 Passing parameters by reference

MOODS takes the meaning of the parameter map of the '*moduleap*' instruction to be pass-by-reference for the various I/O parameters. VHDL signals are also passed by reference, with variables and constant inputs passed by any method the implementation chooses. A design is erroneous if any side effects of the implementation differences produce differing results. This means that a pass-by-value implementation can also be used with equal validity for variable and constant parameters. Since a difference in the implementation only affects the concurrent access behaviour of the passed parameters, it is possible for all parameters to be passed by reference or value without any breakage of the sequential behaviour (constant inputs are driven from expressions that create temporary results, that effectively copy the value). The addition of recursion to the system requires a pass-by-value implementation. Signal passing through recursive procedures is therefore disallowed.

The meaning of pass-by-reference as used by a synthesised design is explained more in the next section, which deals with the final structure generated as output from MOODS.

5.1.2.3 Structural output

The main features of an implementation of a procedure call is the ability to handle the I/O parameter passing to and from the procedure with its parent and the knowledge of which parent call to return to, as the procedure could be called from many places. Both these fundamentals are handled in part by the '*call control node*', which forms one state in the generated finite state machine for every call made by the user's design.

The method for returning to the correct point in the control flow is handled by the call node itself, as it leaves itself active throughout the duration of the call to the procedure (see Section 3.2.6 and Figure 3.10). It only activates the following node when the called procedure reaches an end node with a valid exit condition. A call node only contains the '*moduleap*' ICODE instruction. All other instructions before and after the '*moduleap*' instruction are scheduled in the preceding and subsequent control nodes respectively.

This method of control node activation is not suitable for recursion due to the necessity of keeping the call node active as a placeholder for returning to the correct point in the

control flow. In a recursive implementation, a different method is used for returning from a call correctly.

The method for passing the input and output parameters (actual arguments) of a procedure call are different in the MOODS structural output. In the case of inputs, a port signal is defined for each input, which maps onto the actual arguments passed into every invocation of the given procedure for that input. If the procedure is invoked more than once with different actual arguments, then a multiplexor is created with each different argument as a selected input. The selection of which input parameter to use is gained from the active call node, which is active throughout the entire duration of the call. Figure 5.4 shows an example structure generated to pass input and output parameters directly, as references.

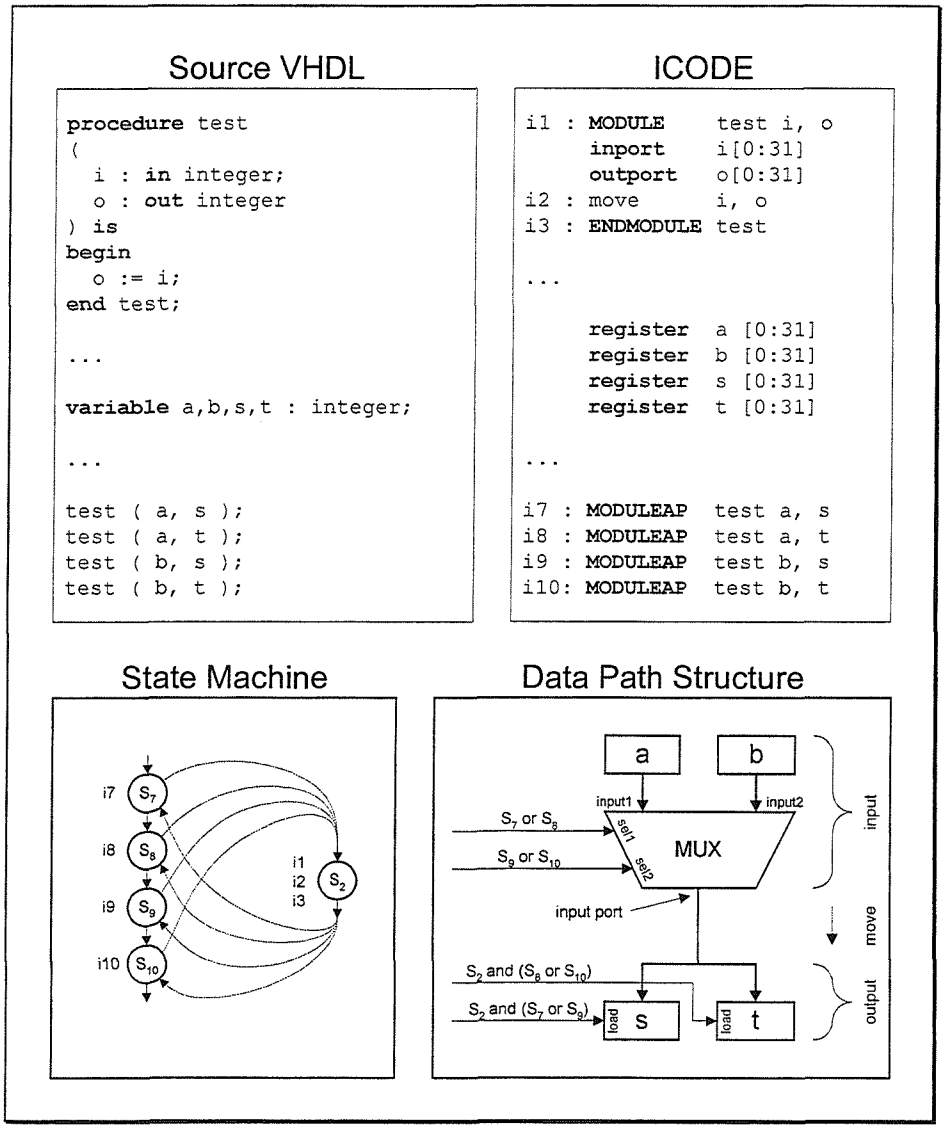


Figure 5.4 Input and output parameter passing

In the case of output parameters, all outputs are mapped onto registers. When an output is written by the called procedure, a load signal is generated from the control node of the called procedure's local finite state machine. This is linked to the load-enable input of the register containing the referenced variable. If more than one map for the output parameter is found from the multiple calls to the procedure, then the load-enable signal drives all the registers that have been mapped. However, only the relevant output must be updated, with all others left alone. This is achieved in a similar way to the input multiplexor select signals, where instead of using the load-enable signal directly for each register, it is first logically-ANDed with the logical-OR of all the call control node active signals, whose call instruction passes the controlled register variable as output. This means that the referenced register is written directly when an instruction inside the called procedure performs a write to its output port. This demonstrates the meaning of pass-by-reference, as the mapped register variable is written from the output port reference.

5.1.3 Additions required for recursion

An outline of the additions made to MOODS for the support of procedural recursion is given here. The general concepts shown here are then expanded in the rest of the chapter.

5.1.3.1 Control nodes and return addresses

As explained in Section 5.1.2.3, the '*call control node*' acts as the return control method from a procedure call. This method is not suitable for a recursive call, as the act of calling a procedure leaves the call node active (for returning correctly and referencing the correct parameters of the procedure's I/O). A recursive call requires a different type of state machine control node that is not active throughout the call duration.

Due to the loss of information from the calling mechanism of the state machine, it is necessary that a 'return address' variable is created for each procedure capable of recursion. Each recursive call instruction then has a unique return address value associated with it. The return address is then used to determine the correct calling control node to reactivate after the procedure has completed execution. The return address is also used to select the correct I/O parameters to reference.

The split into separate state machine calling mechanism and return address allows the dynamic modification of the return address value dependent on the recursion depth, allowing return control to be determined by the generated stack, introduced by the next section.

5.1.3.2 A dynamic stack and stack pointer

On their own, the return addresses do not enable recursion. However, the compact form of a return address enables easy insertion into a stack memory. This enables recursive calls to the same procedure to be stacked one on top of another by 'pushing' the return address onto the top of the stack when performing a call and 'popping' the return address when returning from a call.

The necessity of a stack is determined at compile time, as it is at this stage that any possible recursion is detected in the user's source code. The stack is formed from a RAM cell with the current stack position held by a stack pointer register, which contains an address into the stack RAM cell.

All recursive procedure calls are assigned a unique constant return address value that is held by the called procedure's return address variable when the recursive call is made. Stack manipulation of the return address is performed by explicit auto-generated ICODE instructions. The compiler in this way generates all associated control of the stack and return addresses in the sequential control flow.

5.1.3.3 ICODE modifications

The ICODE operations generated by the compiler for control of the stack and return addresses for each procedure use standard ICODE instructions. However, the extra information generated by the compiler of the return address value associations for recursive calls and return address register associations for recursive modules are also required by MOODS. This information is passed into the synthesis core by slight modifications to the ICODE file format.

These modifications allow MOODS to generate all the associated control logic around the return address variable instead of the '*call control node*', which was its previous method for flow control and parameter passing.

5.1.3.4 Pass by value parameter I/O

The method used to implement recursion involves a change in the parameter passing mechanism from pass-by-reference to pass-by-value. This removes the ability to pass signals through recursive procedures, as all parameter values are copied, not referenced. An alternative implementation could have kept the pass-by-reference semantics, but would have required extensive modification to the structural mechanism used to reference the passed parameters, with the multiple levels of procedure call no longer being used directly to reference the passed parameters. The method would instead rely on compile-time parameter analysis and run-time reference selection logic to determine the correct parameter to reference for each module call.

The method chosen was to keep the underlying pass by reference method for one level of procedure call, but to perform variable copying within the calling procedure, so that the resulting reference only references variables local to the calling procedure. The control of this variable copying is performed by auto-generated ICODE from the compiler. As all I/O parameters are stack-frame local, the place to store these copies is on the stack.

The same applies to local variables in recursive procedures, as these variables are created statically using ICODE register variables. These are also stored on the stack before each recursive call in case they are re-used when the recursive procedure is re-entered. Their values are restored from the stack when the called recursive procedure exits. These modifications effectively add a dynamic storage element to the underlying static hardware storage.

5.1.4 Summary

The rest of this chapter details the methods used to implement procedural recursion as part of any behavioural design. The major points to remember during this description are that:

- The static translation of the VHDL procedures and functions into ICODE '*modules*' is further enhanced in order to provide dynamic storage for the local data and passed parameters, so that multiple versions of the same data set can exist at different recursion depths.

- The dynamic storage mechanism for this dynamic data is held by a stack, which is added by the compiler when possible recursion is detected. An internal stand-alone SRAM-based contiguous memory block data path unit implements the stack.

- The stack pointer is used to reference a single element on the top of the stack, with all stack operations working from this memory address, one element at a time. The compiler generates the stack pointer register variable and all stack operations by using standard ICODE instructions.

- The controlling state machine has a new type of control node that is used for recursive calls. The call node deactivates completely once it activates the starting node of the called procedure.

- A single return address variable is created for every module capable of recursion. The address value represents any of the calling instructions that can activate the recursive module. Each recursive call has an associated unique return address value. The return address holds a single reference to one recursion depth (all other recursive return address values are held in the stack). Structural I/O parameter selection is made dependent on the return address value.

- All stack modification is performed in the calling module, on either side of all recursive call instructions. The return address, local variables and I/O parameters of the calling module are stored before the call and retrieved after the call.

- The pass-by-reference paradigm is used at a single recursion depth only, allowing variables that cannot be concurrently accessed to be passed through recursive calls without a breakage of the pass-by-reference rules. However, the pass-by-value paradigm is used for further recursion depths, which disallows the passing of signals (which can be concurrently accessed) through the procedural interface.

- Both the non-recursive call mechanism and the recursive call mechanism can be used in any one design, with full integration of both mechanisms, allowing any non-recursive modules and module calls to retain their original speed and area overheads, which are less than a recursive implementation (with the inclusion of the stack and stack modification operations).

5.1.5 Example

An example of a recursive design is presented here. Both the source VHDL is given in Figure 5.5, along with the translation in the form of the intermediate ICODE file in Figure 5.6. The example is referenced throughout the chapter for illustrative purposes. The translation is presented before a full explanation of the modifications made, as an understanding of the modified synthesis system appears to be recursive in itself, where an understanding of one section of the implementation relies upon the understanding of another section, which is reliant on understanding the first. The example is a recursive implementation of the Fibonacci series calculation.

```
1   entity test_fibonacci is
2     port (
3       result : out integer
4     );
5   end;
6
7   architecture behave of test_fibonacci is
8   begin
9     main_process : process
10
11      function fibonacci (x : integer)
12        return integer is
13      begin
14        if x = 1 or x = 2 then
15          return 1;
16        else
17          return fibonacci(x-1)+fibonacci(x-2);
18        end if;
19      end fibonacci;
20
21      variable val : integer := 1;
22    begin
23      loop
24        result <= fibonacci(val);
25        exit when val = 50;
26        val := val + 1;
27        wait for 100 ns;
28      end loop;
29      val := 1;
30      wait for 100 ns;
31    end process main_process;
32  end behave;
```

Figure 5.5 Fibonacci test design source code

Note that the shaded lines of Figure 5.6 highlight the extra ICODE variables and instructions added for the stack and its explicit management.

```
1   PROGRAM             test_fibonacci    result
2   outport             result            [0:31]
3   register            val               [0:31]
4   register            fibonacci_ra      [0:1]
5   register            fibonacci_x_in    [0:31]
6   register            fibonacci_fibonacci_out    [0:31]
7   register            stack_pointer_1   [0:7]
8   ram                 stack_1           [0:31] address [0:255]
9                       move              #0, stack_pointer_1
10                      move              #%00, fibonacci_ra
11  .main_process_PR1   move              #1, val
12  .label5             MODULEAP          fibonacci val, 100
13                      move              100, result
14                      eq                val, #50, 101
15                      if                101               ACTT if8_true_11 ACTF if8_false_9
16  .if8_false_9        plus              val, #1, val
17                      protect                             ACT label5
18  .if8_true_11        move              #1, val
19                      protect                             ACT label5
20                      ENDMODULE         test_fibonacci
21
22  RECMODULE           fibonacci         fibonacci_ra x, fibonacci
23  inport              x                 [0:31]
24  outport             fibonacci         [0:31]
25                      eq                x, #1, 108
26                      eq                x, #2, 109
27                      or                108, 109, 110
28                      if                110               ACTT if18_true_19 ACTF if18_false_20
29  .if18_true_19       move              #1, fibonacci     ACT label50
30  .if18_false_20      minus             x, #1, 103
31                      memwrite          fibonacci_x_in, stack_1[stack_pointer_1]
32                      plus              stack_pointer_1, #1, stack_pointer_1
33                      move              103, fibonacci_x_in
34                      memwrite          fibonacci_ra, stack_1[stack_pointer_1]
35                      plus              stack_pointer_1, #1, stack_pointer_1
36                      move              #%01, fibonacci_ra
37                      RECURSE           fibonacci #%01 fibonacci_x_in, fibonacci_fibonacci_out
38                      minus             stack_pointer_1, #1, stack_pointer_1
39                      memread           stack_1[stack_pointer_1], fibonacci_ra
40                      protect
41                      minus             stack_pointer_1, #1, stack_pointer_1
42                      memread           stack_1[stack_pointer_1], fibonacci_x_in
43                      protect
44                      move              fibonacci_fibonacci_out, 104
45                      minus             x, #2, fibonacci_x_in
46                      memwrite          104, stack_1[stack_pointer_1]
47                      plus              stack_pointer_1, #1, stack_pointer_1
48                      memwrite          fibonacci_ra, stack_1[stack_pointer_1]
49                      plus              stack_pointer_1, #1, stack_pointer_1
50                      move              #%10, fibonacci_ra
51                      RECURSE           fibonacci #%10 fibonacci_x_in, fibonacci_fibonacci_out
52                      minus             stack_pointer_1, #1, stack_pointer_1
53                      memread           stack_1[stack_pointer_1], fibonacci_ra
54                      protect
55                      minus             stack_pointer_1, #1, stack_pointer_1
56                      memread           stack_1[stack_pointer_1], 104
57                      protect
58                      move              fibonacci_fibonacci_out, 106
59                      plus              104, 106, fibonacci
60  .label50            ENDMODULE         fibonacci
```

Figure 5.6 Fibonacci test design ICODE translation

5.2 Compiler modifications

The VHDL compiler that is the front-end to MOODS required some modification in its parsing abilities and ICODE generation functionality to fully handle procedural recursion. This section details those modifications.

5.2.1 Forward declarations

As VHDL requires that all procedures be declared before their use, an indirect recursive call structure forces the use of forward declarations of procedures. There are four declarative regions in VHDL in which this is possible. The first is the package header; the second is the package body; the third is the architecture declarative region and the fourth is the process declarative region.

It is possible to defer the definition of the body of the procedure out of the local scope of the declaration, but in the general case, most procedures are defined in full in the same scope as the procedure declaration. It was necessary to add the ability for declaration of procedures without their body in the architecture and process declarative regions for the purpose of forward declaration. This ability already existed in the package header and body due to the package body being able to hold the definitions of procedures declared by the package header. This was previously the only place in which one could define recursive procedures, which was rather limiting.

Declarative Regions

```
package pck is
  -- package header declarative region
end pck;

package body pck is
  -- package body definition region
end pck;

architecture behaviour of design is
  -- architecture declarative region
begin
  pro: process
    -- process declarative region
  begin
    null;
  end process pro;
end behaviour;
```

Forward Declaration Example

```
-- within a declarative region

-- declaration of A
function A(inA: integer) return integer;

-- declaration and definition of B
function B(inB: integer) return integer is
begin
  return A(inB);
end B;

-- definition of deferred function A
function A(inA: integer) return integer is
begin
  if inA = 1 then return 1;
  return B(inA-1);
end A;
```

Figure 5.7 Example VHDL: Declarative regions and forward declarations

The support for recursive forward declarations is added by allowing two definitions of the same procedure in the compilers internal data structures. One holds just the forward declaration information and the other contains the same information as the forward declaration and in addition, contains the body of the procedure as well. Any call to a procedure whose body is not defined links to the declaration version of the internal procedure data structure. When the ICODE is generated in the translation stage of the

compiler, any link to a declaration-only procedure is resolved onto the full-body procedure data structure. This is achieved by searching in each region in turn, starting from the process' declarative regions, then the architecture's declarative region, then any referenced package bodies followed by the package headers.

5.2.2 Detecting recursion

The necessity of detecting when recursion is possible is so that designs with no need for recursive features are created without the extra overhead that is involved with recursion. It is not even enough to detect that recursion is possible in a design, as locating exactly where in the design recursion can occur allows the extra overhead to affect only the procedures that can be recursed.

There are two pieces of information that need to be extracted from the user's design in order for recursion to be detected. These are determining which procedures are contained by recursive loops and which calls to these recursive procedures are recursive calls by checking that the call forms an arc in one of these recursive loops. Both of these sets of information are required for an efficient implementation of recursion, as the information is used to determine which ICODE modules require a return address and which call instructions need to modify the stack contents. If one or more module is found to be recursive, then the compiler generates the stack and all the associated control for that stack.

The reason for needing to know which procedure calls are contained by a recursive loop is that the call can be implemented by the original call method if it is not contained by such a loop. Both the old static calling method and the new dynamic calling method of the state machine are supported in the same design.

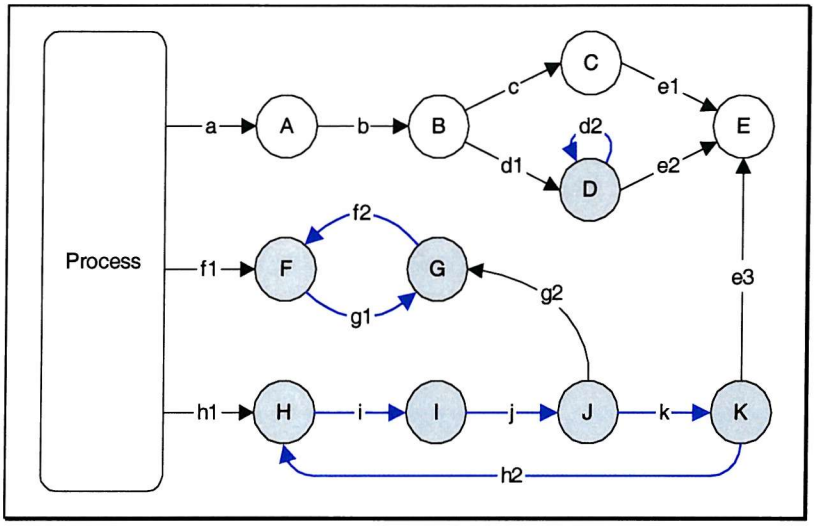


Figure 5.8 Determining recursive procedures and procedure calls

Figure 5.8 gives an example of a call graph. It shows a number of procedures (A-K) represented by the nodes of the graph and a number of calls between these procedures (a-k) represented by the arcs of the graph. Recursion exists in the design if there are any loops in the graph. A procedure can be recursed if it is contained by any graph loop. Similarly, a procedure call is a recursive call when its representative arc is part of a graph loop.

Note that all initial calls (a, f1, h1) from the root process are non-recursive, as it is impossible to invoke the process from a procedure in VHDL.

The algorithm that calculates which procedures and procedure calls are recursive is implemented simply as a depth-first traversal of the entire graph starting from the root process. For each iteration a marker is left in the ICODE module (procedure) being tested when a call to that module is followed. When all the calls from a module have been tested, the marker is removed and the algorithm jumps back to the calling module. In this way, if any module jumped into already has a marker set then the module is marked permanently as recursive. Two loops around any recursive loop in the call graph are required to fully determine every recursive module. The recursive calls are marked in the same way by the same algorithm. Thus the shaded modules in Figure 5.8 are identified as recursive.

5.2.3 Auto-generated ICODE

The compiler generates all the necessary control of the stack and stack pointer when recursion is found in a design. These additions occur after the VHDL parse tree has been translated into an internal representation of ICODE and after procedural inlining. Figure 5.9 shows the multiple phases used in the compiler.

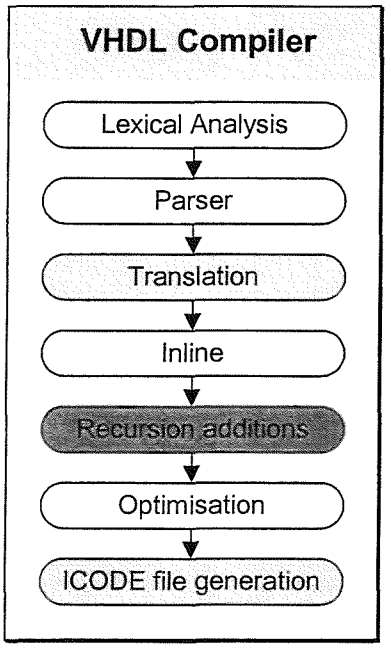


Figure 5.9 VHDL Compiler program flow with recursion modifications

Note that the additions required to implement recursion are made after module inlining. An effect of inlining selective modules in a recursive loop is that the loop becomes tighter. It is impossible however, to inline a module that calls itself directly. This is flagged as an error if attempted.

All the additions made for recursion are made to the internal representation of ICODE in the compiler. This data structure is far simpler than the VHDL parse tree from which the ICODE is based (by translation). The ICODE file, which forms the output of the compiler, is a direct representation of the internal ICODE data structures in the compiler.

5.2.4 Return address generation

The return addresses serve two purposes in the final structural design. They tell the currently active procedure which call-instruction activated the procedure in the first place.

This means that when the called procedure exits, the relevant return address indicates which control node of the calling procedure or process to reactivate. The second purpose of the return addresses is for determining the correct references for any I/O passed as parameters to the procedure.

As each return address contains information relevant to one procedure only, each recursive procedure is assigned its own return address ICODE register type variable. The values held by the return address can reference every call instruction to the procedure. Every recursive call to the recursive procedure will be assigned a unique constant reference number (starting from 1) that can be stored by the return address variable at run-time. Any non-recursive calls to the procedure (those not part of recursive loops) are assigned the constant zero for its address by default. There is no need to make any distinction between non-recursive calls to the procedure, as the old procedure call method is used in these cases. The '*call control node*' that is left active (explained in Section 5.1.2.3) holds the return information. The return address is stored in a register with a number of bits capable of storing the highest constant reference address generated for each recursive call to the relevant procedure. This information is in plain binary format.

The return address ICODE variables are generated in the main ICODE program (Figure 5.6, line 4). This is because of the scoping rules of the ICODE, where a variable declared in a module can only be accessed from inside that module. As the return address requires external modification in the same location as a recursive call to the relevant module (lines 36 and 50), it is placed in the ICODE program, which has global scope.

5.2.5 ICODE instruction modification

After recursion has been detected, each module found to be recursive has its module header instruction changed from the '*module*' instruction into a new instruction defined for recursion, '*recmodule*' (Figure 5.6, line 22). This change allows MOODS to determine which modules are recursive. The '*recmodule*' instruction is then followed by the name of the module and then the return address variable associated with the module, before the normal definition of the I/O list. The structural output in the MOODS core requires the return address variable association.

Similarly, every call that is found to be recursive has the ICODE instruction '*moduleap*' changed into a new type of call instruction also defined specifically for recursion, '*recurse*' (Figure 5.6, lines 37 and 51). This change allows MOODS to distinguish between the two types of calling methods. A '*recurse*' instruction is followed by the name of the module being called and then the associated return address constant value, before the normal definition of the I/O map for the particular call, as the structural output in the MOODS core also requires the associated return address constant.

For a full definition of the modified ICODE file format, see Appendix D.

5.2.6 Parameter passing

The underlying structure of the final generated hardware still uses pass-by-reference for the procedure's I/O parameters. It is necessary to change this into a form where the reference only references a variable local to the calling procedure, effectively reducing the method to a pass-by-value with one level of reference indirection. The reason this is necessary is due to the new method for selecting which I/O arguments to reference, namely the return address, which holds only a single reference to the parent calling procedure at run-time, not the entire stack of return addresses (which are required in order to deduce the referenced root variable).

A register variable is created for every input and output port of every recursive procedure (Figure 5.6, lines 5 and 6). These registers are used to mirror the arguments passed as I/O parameters from each recursive calling-module into every recursive called-module. The registers are added to the top level ICODE program, not the module from which the mirror registers derive, as they must also have global scope in the same way as the return address.

For every recursive call to a recursive procedure, the '*recurse*' instruction is modified to use these new mirror registers as the values passed into its I/O map. In addition to this, for the case of input parameters, the values that originally would have been passed into the '*recurse*' instructions I/O map are copied to the mirror registers before the call (Figure 5.6, lines 33 and 45). In the case of output parameters, the values are copied back from the mirror registers into the originally passed output parameters after the recursive call returns (lines 44 and 58). This requires the addition of an ICODE '*move*' instruction per

parameter, added before (in the case of inputs) or after (in the case of outputs) the '*recurse*' instruction.

5.2.7 Stack manipulation

There are two main implementations for the stack that were considered. The simplest implementation, a single contiguous block of memory used for all stack frame data is used. This permits flexibility in the final hardware implementation, where any number of memory types may be used for the actual data storage. The initial implementation uses an internal RAM cell, with user-defined address space. This can easily migrate onto an external memory description, with a change in the interfacing methods, allowing larger stack depths to be used.

The second alternative implementation considered is to have multiple stacks that keep track of one design variable each. This enables a more memory efficient allocation strategy for every register, with the stack data width tuned to the width of the variable that it references. This strategy enables concurrent stack modification for every variable, which allows an increase in the speed of the final design by reducing the number of clock cycles required. The drawback of this method is due to the variable sized stack frame requirement of each variable, which is dependent on the position of the recursive call within the calling module as to whether the variable requires stack storage or not (see Section 5.2.7.5). Therefore, it is possible for some variables to require more stack frames than others. The act of balancing the number of stack elements for each variable, so that one stack block does not fill up before another is not trivial. In fact, it is not calculable at compile-time as the analysis problem is not static. The amount of space required is determined at run-time, as there is the possibility and likelihood of performing recursive calls conditionally, dependent on run-time decision data.

5.2.7.1 Stack and stack pointer creation

The present system creates a single stack of 32-bits data path and a user-defined address space. Concurrent processes invoking recursive procedure calls have been disallowed (see Section 5.2.8). The 32-bit data path specifies the largest variable width that is capable of being stored on the stack. This particular width is chosen to mirror the space required to store a VHDL integer and to handle the full dynamic object data path width specified in

Chapter 4. The stack itself is created in the top-level ICODE program as a RAM variable (Figure 5.6, line 8). This means that one 32-bit word of memory is accessible within one clock period for read or write-access. All stack addressing occurs via the stack pointer variable, also defined in the top-level ICODE program (line 7), which is a register capable of holding the full address to any object held on the stack. The stack pointer width is dependent on the address range of the stack, which is user defined.

5.2.7.2 Push and pop operations

All the dynamic operations added to the recursive call instructions access the stack. The two main operations performed on the stack are *push* and *pop*. A *push* operation writes a given value into the stack and increments the stack pointer ready for the next *push*, while a *pop* operation performs the reverse of a *push* by decrementing the stack pointer and reading back the value at the decremented position. The equivalent ICODE instructions for the *push* and *pop* operations are shown in Figure 5.10.

ICODE variable definitions and setup

```
register stack_pointer_1 [0:7]
ram      stack_1         [0:31] address [0:255]

move     #0, stack_pointer_1
```

PUSH (val)

```
memwrite val, stack_1[stack_pointer_1]
plus     stack_pointer_1, #1, stack_pointer_1
```

POP (val)

```
minus    stack_pointer_1, #1, stack_pointer_1
memread  stack_1[stack_pointer_1], val
protect
```

Figure 5.10 ICODE equivalent instructions for stack modifiers

In the example, the stack is created with 256 addresses, so the stack pointer requires 8-bits to store the full address. The '*move*' of zero into the stack pointer occurs only once at initialisation. All other modifications are performed by the '*plus*' and '*minus*' ICODE

instructions that form part of the *push* and *pop* operations. Note that the *pop* operation has a '*protect*' instruction after the stack modification. The '*protect*' instruction forces a clock cycle break between the instructions above and below the '*protect*'. This is required in the case of popping the return address, as there is no dependency checking for I/O references across module borders in MOODS. This is explained in greater detail in the next subsection. The extra '*protect*' has no adverse effects, even if included for every *pop* operation, as every *pop* includes one memory read instruction of the stack, each of which already requires a separate control state.

5.2.7.3 Return address manipulation

All stack modification is performed in the calling module, rather than the called module, with the controlling ICODE instructions added before and after every '*recurse*' instruction.

When a recursive call is made, the return address of the called module is set to the address associated with the recursive call. This operation tells the called procedure which I/O to reference and where to return control after the called procedure exits. However, the return address could be holding a valid address already, from a previous call to the procedure. This then, is the point at which this previous address is pushed onto the stack (Figure 5.6, lines 34-35 and 48-49), before the modification to the new address (lines 36 and 50). The call then executes with the knowledge that the return address is valid, and that the previous return address for the called procedure is stored on the stack for future reinstating.

The complement of the *push* of the return address for a called procedure is to *pop* the return address straight after the procedure has returned from the recursive call (Figure 5.6, lines 38-40 and 52-54). Note that the return address must be reinstated before any other *pop* operations from the stack, as the return address (which may change at this point) determines the correct I/O parameters to map onto, which may be referenced by the following *pop* operations. It is for this reason that the '*protect*' instruction is made part of the *pop* operation, so that the return address register updates before the following instructions are executed. Normally, the data dependency information held by MOODS is used to determine whether the registered version or non-registered version of a variable is used during a control state. However, the dependency of the following *pop* operations with

the return address is not directly specified, it is only implied by the internal structural use of the return address.

Another effect of the single-cycle update time for the return address is found with the rare case of no further operations taking place after the *pop* of the return address. The *pop* operation is scheduled in one of the end-states of the module in this case. However, as the registered return address is used to reactivate the calling control node, the return address is not valid until the end-state completes. This is too late to reactivate the correct calling control node (with activation made incorrectly using the current return address). Detecting this rare case in the ICODE and adding an extra '*protect*' after the *pop* operation for the return address solves this problem. This has the effect of creating an extra control state after the *pop* operation for the return address, in which the registered version of the return address is valid. This extra state becomes an end-state of the control graph (replacing the *pop* operation state as an end-state) and is used to reactivate the correct calling control node. Section 5.4.3 gives an example of timing for the return address setup cycle.

5.2.7.4 Output parameters

The output parameters of any call do not require updating, as any writes to them during the procedure reference the passed parameters directly. The parameters do not require stack storage, as by definition, they form the result of the procedure. However, the mirror registers, which are passed into the procedure as the referenced variables contain the results of the procedure. Section 5.2.5 explains the purpose of mirror registers. The mirror registers are used to update the original passed parameters defined in the calling procedure. The output variables are updated after all of the stack operations.

5.2.7.5 Input parameters and local variables

The input parameters of a recursive call, on the other hand, do require insertion onto the stack. It is at this point that the mirror registers for these values are modified to the values being passed into the called procedure. However, these mirror registers could be holding a valid value from a previous call to the procedure being called, so before the mirror register values are modified to the values being passed, the old values are pushed onto the stack for future reinstating (Figure 5.6, lines 31-32). Note that the *push* operations occur before the *push* of the return address, so that all references to I/O of the current procedure are still valid.

Local variables (including temporaries) are treated in exactly the same way as input parameters, in that they are pushed onto the stack after the inputs (lines 46-47). The only difference is that the values held by the variables are not modified to any particular new value; they are modified by the next iteration of the procedure currently being jumped out of. These values are pushed in the knowledge that further calls to the present procedure are possible by recursion.

Both input parameters and local variables that were pushed before a recursive call are popped after the call in the opposite order, which brings these values back into their original state as found before the call (lines 41-43 and 55-57).

An extra optimisation in the number of *push/pop* operations performed around a recursive call is possible. The optimisation involves calculating which input parameters and which local variables are written before a recursive call and read after the call. If the values are never written before and read after a call, then these values do not require insertion onto the stack, because the data they hold before the call is never accessed after the call. The check is further reduced to just checking for a value being read after a call, as if a variable was not written before, then the value will be invalid anyway, which is caught by behavioural simulation. Figure 5.6 shows that the input mirror register is only stored around the first recursive call, as the input is not referenced after the second call. Also, of all the local variables in the Fibonacci module, temporaries 103, 104, 106, 108, 109 and 110, only temporary 104 is stored around the second call, as this holds the result of the first recursion, used after the second recursion.

Following all control-flow paths from the '*recurse*' instruction (including loops and all paths from conditionals), finding whether each local variable and input variable of the '*recurse*' instruction is ever read after the call, makes this check. If the variable being checked is found to be written to, then the check need not carry on any further along that path, as the write overwrites any previous value, which also means that the variable does not require stack storage.

5.2.8 Limitations

There are a few limitations when using recursive procedures, but most restrictions are not too great or can be worked around.

The first limitation is that signals [80] cannot be passed into a recursive procedure. The reason for this is explained in Section 5.1.3.4, with the introduction of pass-by-value parameter passing. The reason that signals cannot be passed is that they require a full pass-by-reference method, so that the root signals are updated directly at the wait statements in a procedure. The pass-by-value method will not work correctly because the output is updated only on the return of a call. A workaround for this limitation is to use global signals or entity ports directly in the procedure if the recursive procedure is defined in the architecture or processes declarative regions of a design.

Another limitation is that only one process in a design is allowed to call recursive procedures. This is not too great a restriction: multiple processes that are required to call recursive procedures can be split into separate design units. This restriction could be removed in the future if multiple stacks are created for each concurrent process in a design. This requires the replication of recursive procedures if they are called from more than one process, as each procedure implementation statically accesses one stack.

The 32-bit data path of the current stack configuration is a limitation if data greater than 32-bits requires dynamic storage. This limitation is caught during compilation. A method to cope with this problem is to create a stack with a compile-time configurable data path width that is optimised to the greatest data path width that is stored on the stack. This problem would also disappear if the alternative stack structure, described in Section 5.2.7 were used. The generated structure is of a separate stack memory for each dynamic variable, where the data path width of each stack is the same as the data path width of each variable requiring storage.

The final limitation for recursive procedures is that these procedures are not allowed to contain RAM arrays as local variables, as making dynamic copies of the entire array for every stack frame takes a time proportional to the address size of the array, and fills the stack memory extremely quickly. If access to a RAM array is required, then the array could be placed in the same declarative region as the procedure definition, which moves the array from the procedure's scope (requiring stack-frame local data) into the parent scope of the procedure.

5.3 Hardware generation

The modifications made to MOODS and the structural VHDL generator, described by this section, complement the front-end modifications necessary for recursion made to the compiler, described in Section 5.2. These changes are mainly with the introduction of the '*recurse control node*', the alternative to the '*call control node*' and the utilisation of the return addresses by the controlling state machine and module I/O selection.

5.3.1 Modules in MOODS

The MOODS data structures are built from the inputted ICODE file generated from the compiler. The ICODE '*module*' has a direct equivalent structure in MOODS, with the addition of more information about its present optimisation state. This includes the control graph that represents the controlling state machine. The ICODE instructions are contained in this control graph and the act of optimisation moves the ICODE instructions across control states, allowing the number of states to be changed.

The addition of procedural recursion affects the MOODS structure containing the ICODE '*module*', as the module is derived from the original VHDL source procedure or function that could be recursive. The extra information required by MOODS is generated by the compiler and contained in the '*recmodule*' and '*recurse*' ICODE instructions. This gives MOODS the information that a module can be recursively entered, the return address associated with the module and the return address constant allocated to every recursive call. The ICODE file contains this information.

5.3.2 Post-optimisation step

While MOODS performs the core optimisations to a design, the internal data structures hold the complete design that represents the structural output at any time. However, these data structures do not contain explicit one to one mappings with the actual hardware generated. Instead of the conditional signals that form the link between the controlling state machine and the data path being created explicitly on the fly as part of the optimisation process, they are simply implied by the ICODE instructions that perform the various operations. This is far more efficient in terms of optimisation speed.

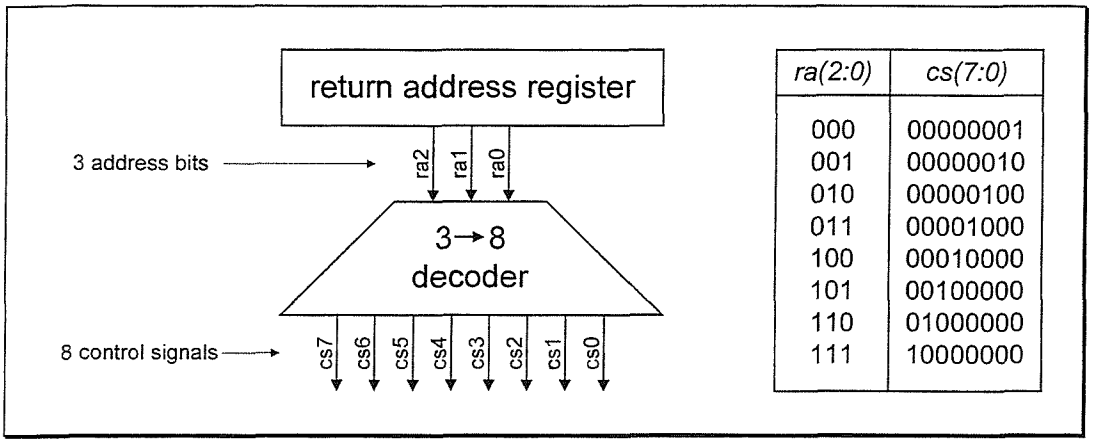
Equally, when MOODS shares data path nodes, they then have multiple inputs that are active at different times in the control flow. Instead of explicitly creating a multiplexor for these data path nodes during optimisation, the multiplexor is implied by the existence of multiple drivers for the shared data path node. The effects of the multiplexors are also implied during optimisation, which means that the delay and area costs of the multiplexor are taken into account without the existence of a physical multiplexor.

These implied components are instantiated during a post-optimisation step of MOODS. At this point all multiplexors are physically created where required and the controlling conditional signals are generated to drive the multiplexor select signals, register load-enable signals and all other control inputs for every data path node in the design where required. These signals also form the path back from comparison data path nodes, such as an equality comparator, into the generated state machine, so that data-dependent control flow can occur.

It is during this post-optimisation step that the many structural additions required for recursion are realised.

5.3.3 Return address decoder and control signals

The dynamic control of the final structural design occurs via the conditional control signals that are derived from the controlling state machine. This system is now augmented with the return address registers that are used in conjunction with the modified state machine. The values stored by the return addresses form a binary representation of a particular control node to reactivate. This binary representation is converted into a one-hot output, where only one signal is active for each possible return address value. This is accomplished using a decoder data path node for every return address register. The decoders are fed directly from the return address registers and generate a number of control signals, each of which is singularly active dependent on the return address values. These signals are used to feed the various conditions in the condition list, which in turn are used to control the flow of the state machine and the I/O selection for each module. The condition list stores every conditional equation, which forms the glue-logic between the control and data paths, as shown by Figure 3.2.



ra(2:0)	cs(7:0)
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

Figure 5.11 Example return address decoder

The example return address decoder shown by Figure 5.11 converts the 3-bits of the example return address into the eight control signals that drive the conditional equations.

5.3.4 State machine

The state machine, used to control the sequence of operations performed by the design, is derived from a direct conversion of the control graph in the MOODS data structures. Each control state has a corresponding control cell that implements the state. There are three basic control cell types now in use. The first is the '*general control cell*' used for all control nodes except call nodes, the second is the '*call control cell*' used for non-recursive module calls and the third is the new '*recurse control cell*' used in conjunction with the return address for recursive module calls. In most designs, the control graph is formed mostly from the '*general control cell*'. The other cells are used exclusively for the two calling mechanisms now supported.

Each module has its own separate control graph, which is activated by one of the two methods of module calling. Each module has a single start-node and can have multiple end-nodes. When generating control signals in the post-optimisation step, a single end condition is created from the logical-OR of all end node tokens in the module. The tokens themselves may be the product of a logical-AND with any data-dependent conditions active in the end-node. This end-signal defines when a module finishes its execution flow. The use of this signal is explained in more detail in the following sections.

5.3.4.1 General nodes

The general node forms the basic control cell that implements the token-passing, one-hot-encoded state machine. The control cell is designed using structural VHDL, with the number of input tokens defined by a generic parameter of the cell. The input tokens form the activating signals that are used internally to activate the control cell for one clock period. Figure 5.12 shows a representation of the general control node. This example shows a node with three activating tokens, which means that the node can be activated from the tokens of three other control nodes (which can also include itself).

If a token input is linked directly to the token output of another control cell, then the control cell will always be activated one clock cycle after the directly linked control cell was active. Conditional branches are implemented by forming Boolean expressions with the control node token inputs, using the original source comparison operator result as an input to the Boolean expression.

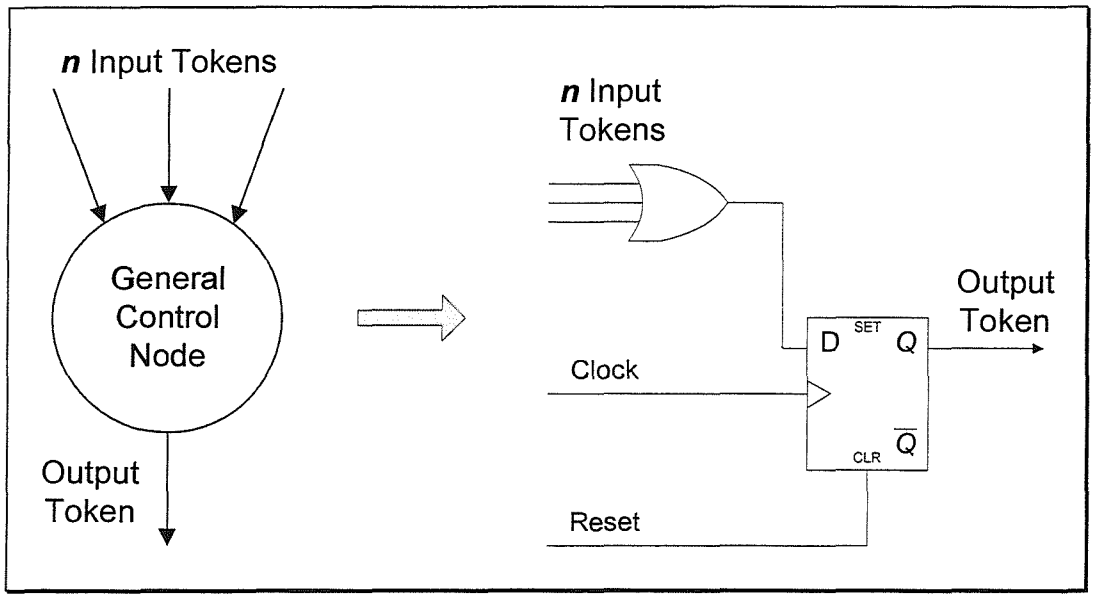


Figure 5.12 The general control node

Each general node is realised by a single register and an n-input OR-gate, which gives a very dense and efficient representation in the register-rich environment of the FPGA.

The link back to the data path of the design is formed via the token signals themselves (see Section 3.2.2). These tokens are used for the various control inputs of the data path nodes, such as register load-enable signals and multiplexor-select signals. Just as with conditional branch execution flow, these signals can either be used directly, or via extra Boolean

expressions that conditionally determine whether a data path node is used in a single control state.

5.3.4.2 Call nodes

The call node is the implementation of the ICODE '*moduleap*' instruction. No other instructions are scheduled in the same state as the call. The call node has similarity with the general node in that it takes a number of input tokens that activate the node, and produces a single token that is used to activate the node that implements the state containing the instructions that follow the call instruction. Details of the calling mechanism are given in Section 3.2.6, with Figure 3.10 providing an example of timing waveforms.

Figure 5.13 shows the implementation of the call control node. Notice that there are three extra signals defined for this node. The first, 'Activate', serves as the activation signal for the sub-control graph that forms the controller of the module being called. This signal drives one of the input token lines of the start-node of the called module.

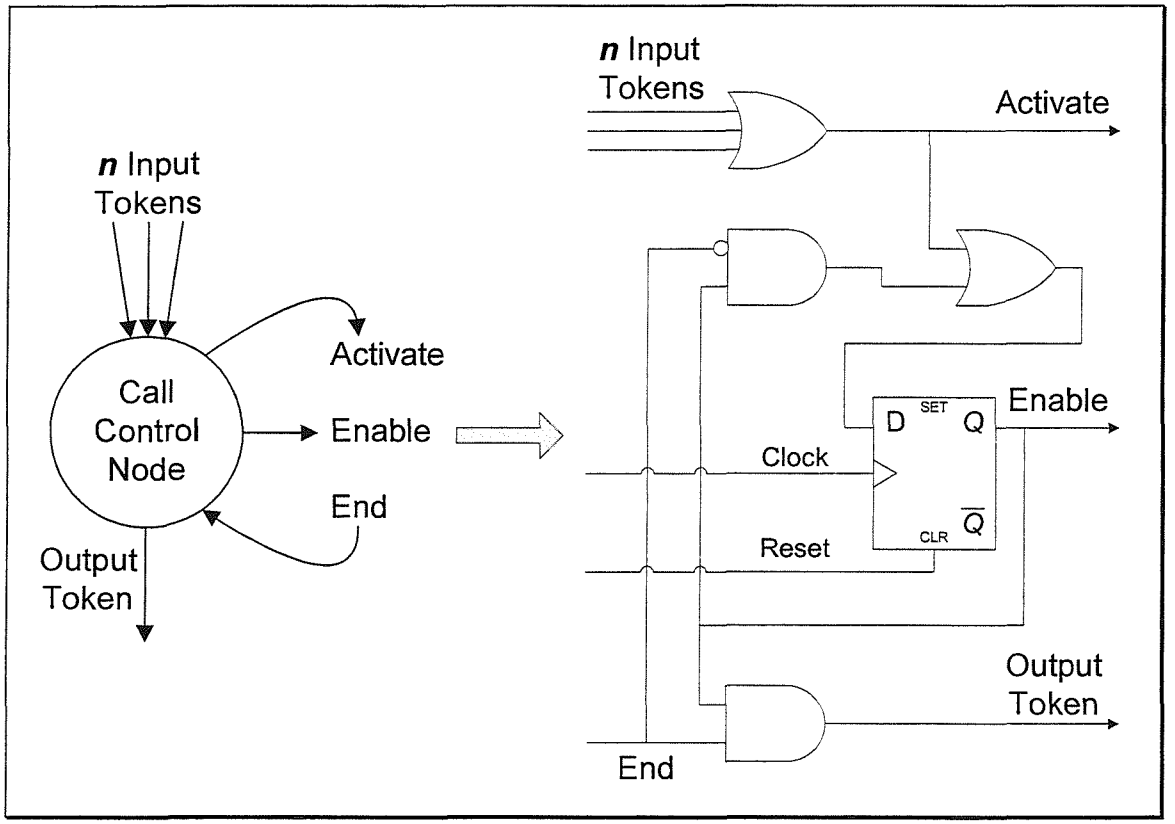


Figure 5.13 The call control node

The second signal, 'Enable', is the output of the register that stays active throughout the entire duration of the call. This is the signal that is used to reference the called modules I/O that is mapped during the call to the module, for this particular call.

The third signal, 'End', forms the link back from the module being called. Each module has an end signal associated with it that is generated from the output tokens of any of the end-nodes of the module. This end-signal is fed back to every call control cell that activates the module.

Note that the 'Activate' signal is driven directly from the logical-OR of every token input, which means that the start node of the module being called is activated at the same time as the call control node register is activated.

The register in the call node is set when the call node is activated by any of the input tokens. It stays in this state by way of feedback from itself unless the 'End' signal becomes active; in which case, the registered value is reset on the next rising edge of the clock. The complement of this action is to set the output token from the call node when the 'End' signal is active along with the call node being active. The token output activates the node that follows the call. As no other instructions can be contained by a call state, no conditional activations can be formed from the call control node. This means that a single node is activated after the call node.

5.3.4.3 Recurse nodes

The recurse control node is part of the mechanism used for the implementation of a recursive call instruction. The '*recurse*' ICODE instruction is the only instruction scheduled in a recurse node. The recurse node only forms half of the controlling actions of a recursive call, with the other half implemented by the return address associated with the module being called.

The '*recurse control node*', shown by Figure 5.14, is formed from a reduced version of the '*call control node*'. As no register can be left active for the call duration, this node does not contain a register. Instead, it just forms the link that activates the start-node of the called module and the link back from the end-nodes of the called module that activates the node following the recursive call.

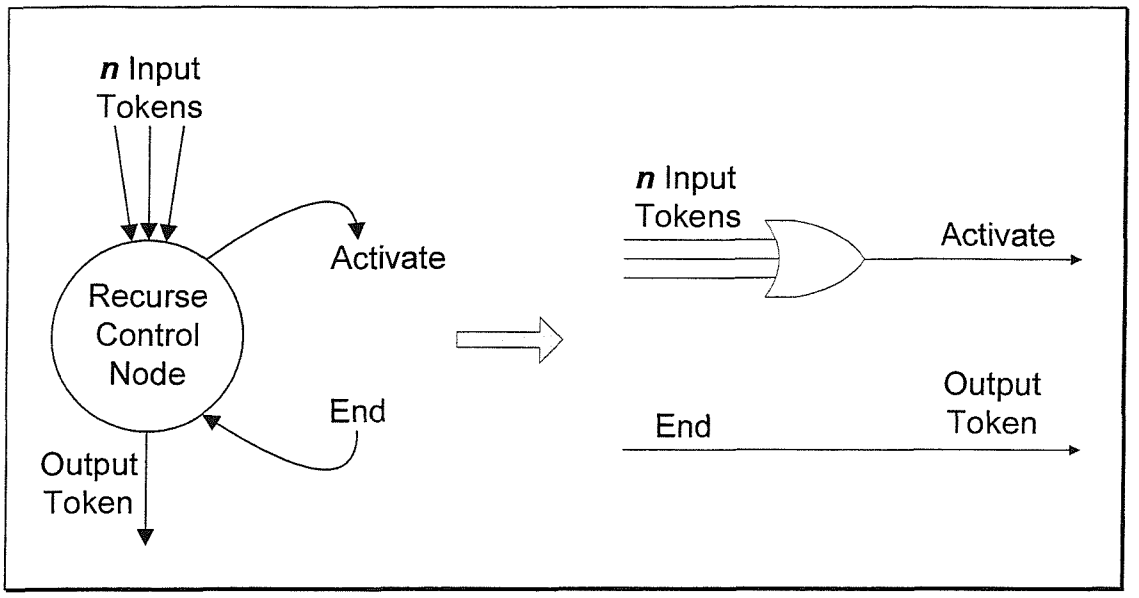


Figure 5.14 The recurse control node

The 'Activate' signal does exactly the same job as the equivalent signal of the call control node, in that it is linked to one input token of the start control node of the called module. The 'End' signal simply drives the output token that activates the following control node and is driven from a Boolean equation derived from the return address, described in the next subsection. This serves as the calling mechanism for the state machine (see Section 5.4 for an example in recurse node timing).

5.3.4.4 Linking the return address

The returning mechanism for the '*recurse control node*' is achieved with the use of the return address value that is decoded into a number of separate signals, as explained in Section 5.3.3. If the 'End' signal of the '*recurse control node*' were driven directly from the end-signal of the recursive module, then this would mean that all nodes following any recursive call to the called module would be activated after the call finishes, producing incorrect behaviour.

The decoded return address signal whose value represents every '*recurse*' instruction by a unique constant identifier is used along with the called modules end-signal by a logical-AND of these two signals, to correctly determine which node to reactivate. Every '*recurse control node*' uses the relevant decoded return address signal specified for the particular recursive call. It is assumed that the registered return address is valid at the time of the last

state of the recursive module. This is a valid assumption, as it is checked and fixed during compilation, as explained in Section 5.2.7.3.

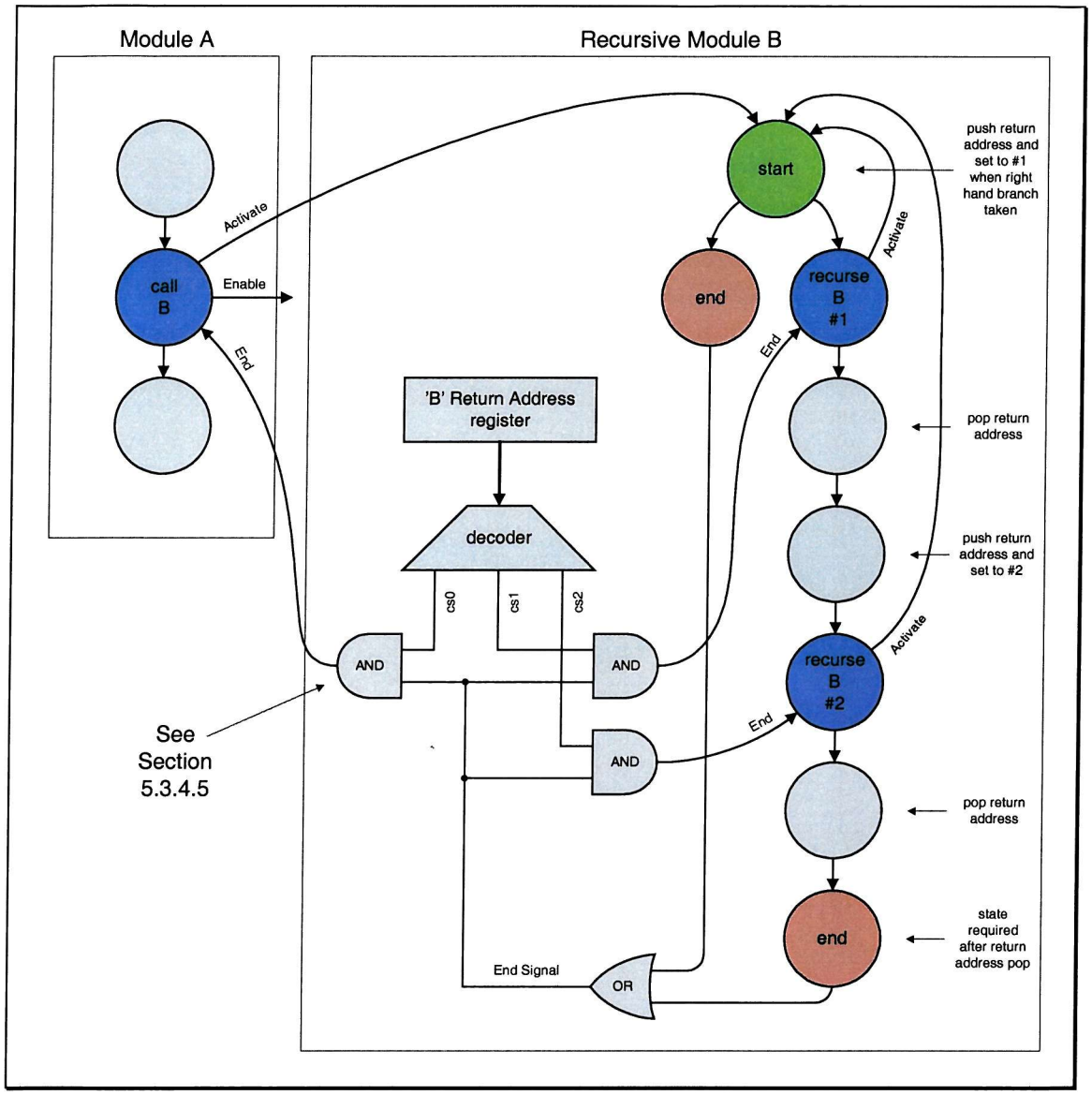


Figure 5.15 State machines use of the return address

The example in Figure 5.15 shows a single recursive module 'B', which can recursively call itself twice. Hence, the return address for module 'B' requires three values (including zero for all non-recursive calls). Note that the start-node of module 'B' is activated unconditionally from every call to the module. The call and recurse nodes are reactivated dependent on the condition of the decoded return address register. This register is controlled explicitly by the ICODE instructions generated by the compiler, scheduled in the preceding and successor control nodes of the recursive call nodes.

5.3.4.5 Mixing call mechanisms

There are rules for which calling mechanism is used depending on the recursive status of the module being called, the module being called from and whether the call is part of a recursive loop. There are five valid combinations of these three criteria shown in Figure 5.16. Note that the type of call mechanism is really only dependent on whether the call is part of a recursive loop. However, all combinations are listed, as each requires a different level of integration with the return address mechanism used by recursive modules.

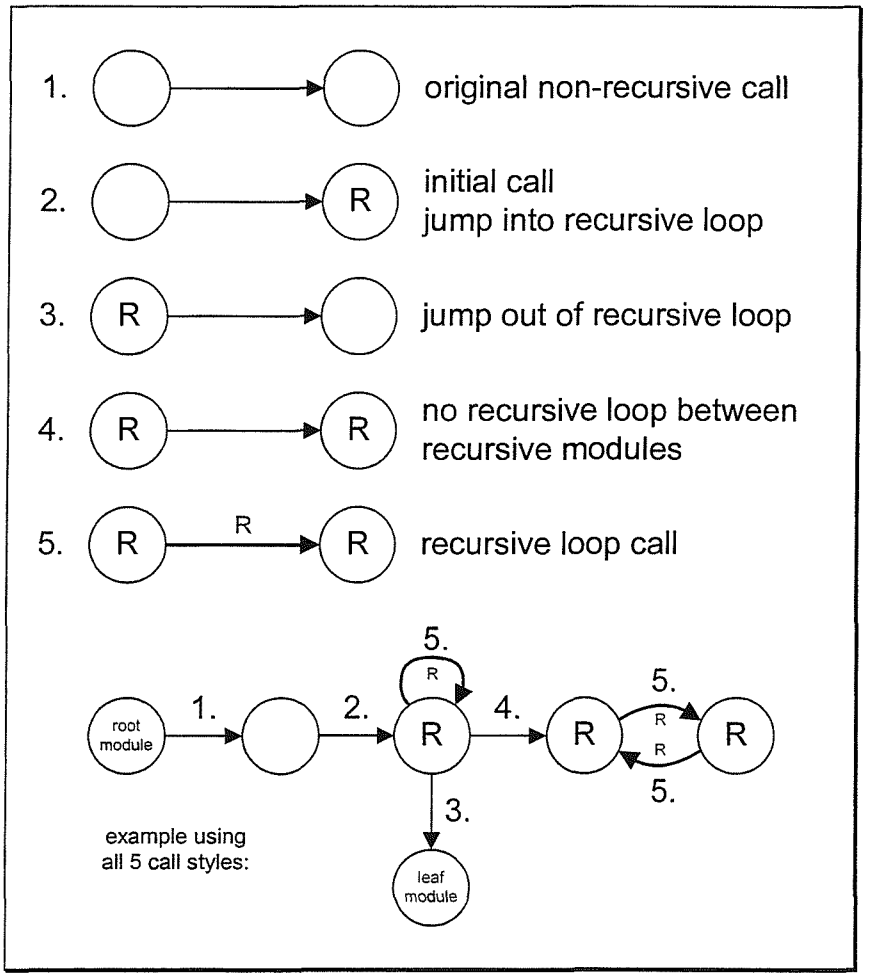


Figure 5.16 Module call styles

The ability to form a recursive call between two recursive modules (style 5) is described by the body of this chapter.

If a call is made from and to a non-recursive module (style 1), then this call uses the existing call method of the '*call control node*'. No extra signalling is required for this. Similarly, if a call to a non-recursive module from a recursive module occurs (style 3),

then no extra signalling is required for this situation, as it is safe to leave the '*call control node*' active within the recursive calling module, as there is no possibility of recursively reaching that call point again.

In the case of calling a recursive module using a non-recursive call (styles 2 and 4), some extra signalling is required that was not present before. This situation occurs when calling from both recursive and non-recursive modules and actually forms all initial calls into recursive module loops. The non-recursive calling mechanism is used in these cases, which leaves the '*call control node*' active throughout the duration of the call. However, the end signal that is fed into the call control node cannot be derived directly from the called modules end signal. This is because the end signal could be activating a previous invocation of the recursive module and not the initial call.

The method used to distinguish between reactivating any recursive call node and reactivating the initial call node is to use the return address again. The non-recursive initial call is reactivated only when the return address holds the value zero. The return address decoder, along with all the other recursive return addresses, generates the signal that describes this situation. A logical-AND using the decoded signal that describes address zero and the modules end signal is used to feed the 'End' signal of all non-recursive calls to the module. No distinction is required between any multiple non-recursive calls to the module due to only one '*call control node*' being enabled at a time, which means that control returns to the correct position.

As each recursive modules return address is used for non-recursive calls to recursive modules, all return addresses must be reset to zero before each call. This is best achieved at the beginning of the execution flow for the whole design, as it then becomes unnecessary to reset the return address before each call, as they are reset to back to zero by the last stack frame of a recursive procedure.

5.3.5 I/O referencing

All module I/O is passed as references. This means that the storage space used to hold the parameters passed into a module are written to and read from directly by the module when it accesses them from its interface. As different parameters can be passed into the same module, a mechanism to select which values to reference is required for any operations

that use the module I/O parameters. The structural mechanism for both inputs and outputs is explained in Section 5.1.2.3.

This mechanism, used before the additions required for recursion were added is expanded upon to fully support the correct referencing of I/O across recursive call boundaries. The method for referencing I/O has not changed; only the way in which the selection of which mapped I/O variable is achieved. The new method does this by incorporating the use of the decoded return address signals in the source selection hardware.

5.3.5.1 Input multiplexors

All inputs to a procedure are referenced in the final structural VHDL by a signal defined for the input. This signal is used wherever the input is referenced. If only one input source is passed into the module, then the generated input signal is driven directly from the passed input parameter. Passing different parameters into the module requires a multiplexor to drive the input signal. This multiplexor is driven from the various sources of input, which could include constants or registered variables. The selection of which input to use is determined by the multiplexor select signals, which are generated from the controlling state machine. In the case of non-recursive calls to the module, the input selection signals are driven by the call control nodes 'Enable' signals, which are active throughout the duration of the call, so selects the correct input throughout the entire call.

Recursion does not change this situation; only the input selection signal was changed, as the 'Enable' signal of a call control node does not have a direct equivalent in the '*recurse control node*'. Instead, the return address associated with the particular recursive call of the module is used to determine the source of the input. Note that due to the addition of the mirror registers, the only values recursively passed are the mirror registers themselves, which are selected by the logical-OR of all recursive return addresses of the module. Also note that the non-recursive call parameters selection signal is driven from the logical-AND of the original selection signal and the zero return address decoded signal.

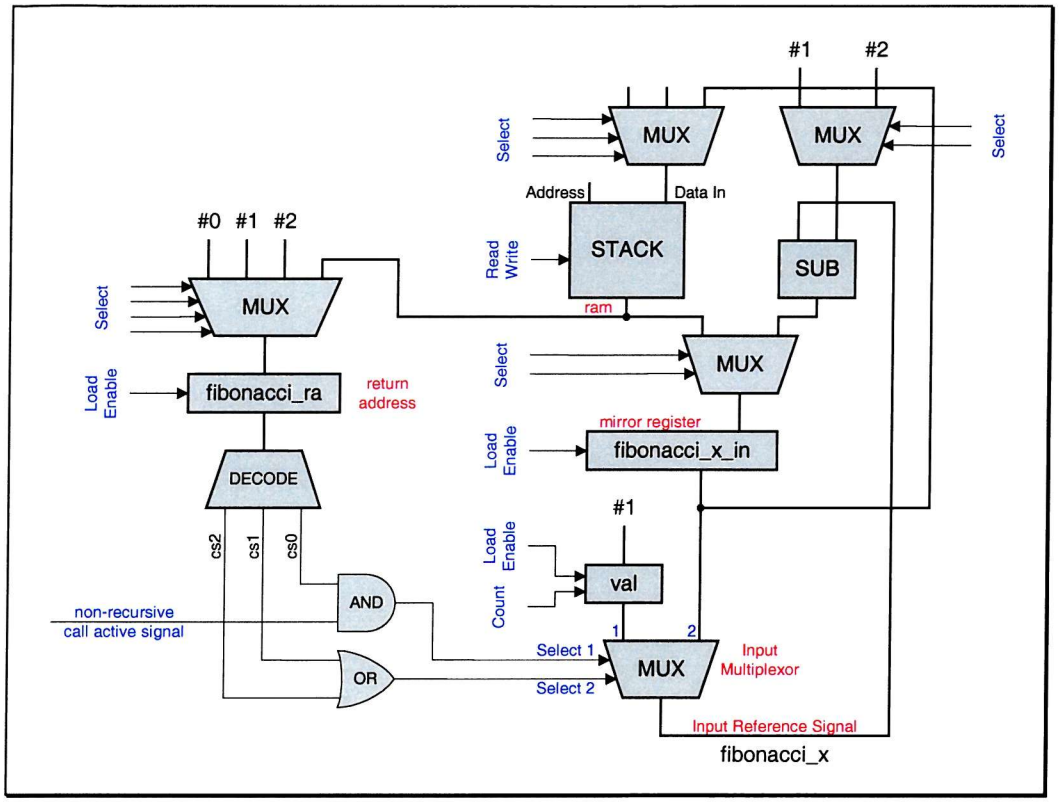


Figure 5.17 Example generated structure for module inputs

Figure 5.17 shows a portion of the generated data path structure for the example given in Section 5.1.5. It highlights the generation of the mirror register '*fibonacci_x_in*' and the use of the input reference signal to feed the mirror register with a subtraction of one or two from itself. Notice that the mirror register is also fed from the stack, as the input is read after the first recursive call to the function. This means that the input is pushed and popped around that call, so that it will hold the correct value originally passed into the procedure. The multiplexor-select signals and register load-enable signals are all generated from the control cell tokens and the return address associated with the 'fibonacci' function.

5.3.5.2 Output registers and multiplexors

The situation is similar, but not entirely the same for the output parameters of a call into a recursive procedure. Because registers hold all output parameters, the separate reference signal as used for the module inputs is not required. The source VHDL guarantees that outputs are variable, as constants cannot be passed into output parameters. The limitations of behavioural synthesis also stipulate that RAM variables cannot be passed as parameters into procedures.

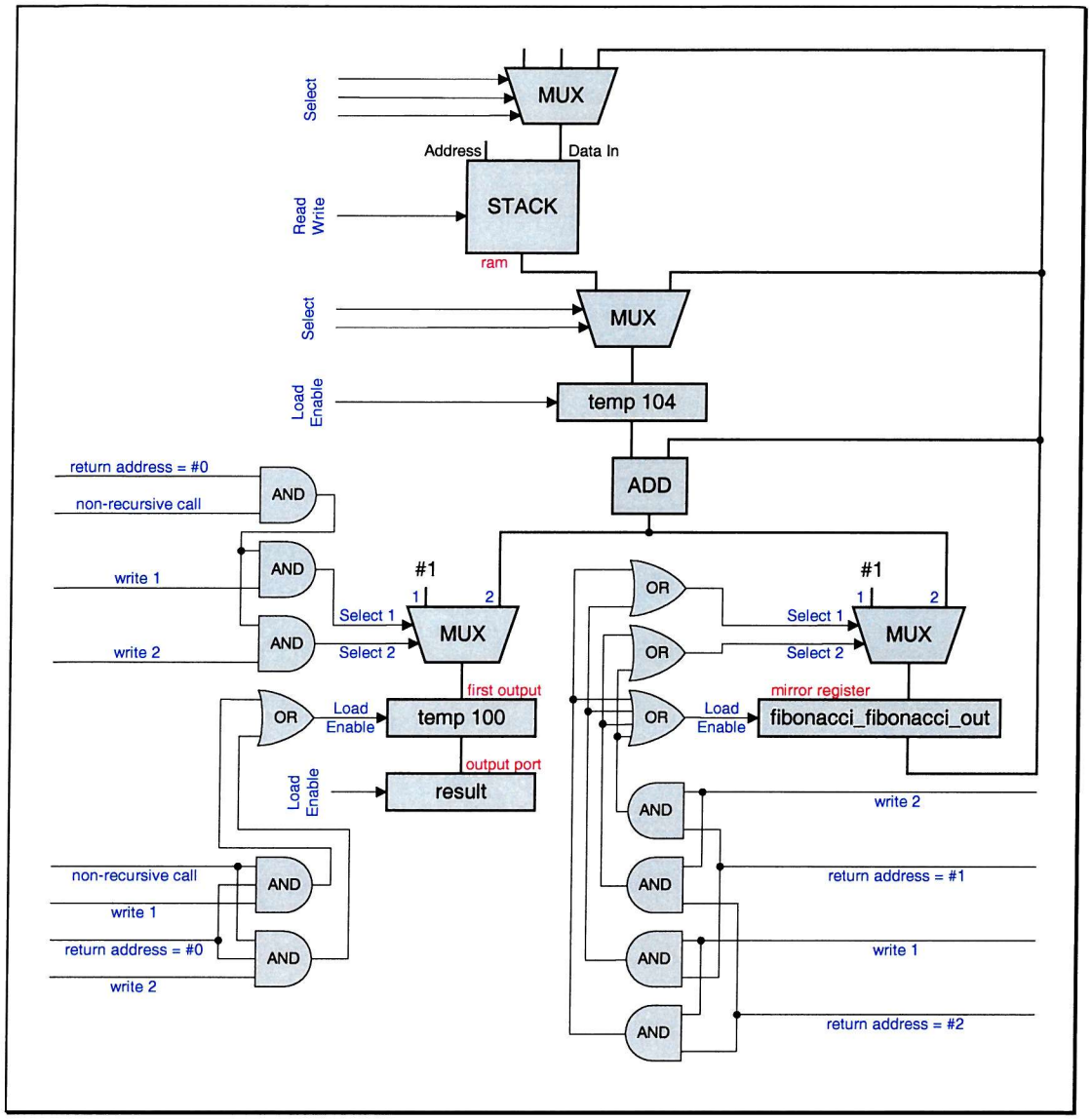


Figure 5.18 Example generated structure for module outputs

The addition of mirror registers limits the number of parameters passed recursively, but the initial non-recursive calls to the module allows referencing of other variables other than the mirror registers. The first stack frame of the module references the initial parameters passed into the module.

As all outputs are guaranteed to be held in a register type data path node, output write-referencing is implemented by using multiplexors to feed the register from the relevant value being written and by driving the register load-enable control inputs at the same time as selecting the correct input to assign to the output register.

The selection of which output to write to is made using the decoded return address signals again. Any recursive call passes the mirror registers as the references, so the mirror

registers are written whenever a second iteration of the recursive module is entered. This situation occurs when the return address is not zero. When the return address is zero, the parameters passed in the initial call are used. A separate multiplexor is used per output as the register may be written outside of the procedure. Figure 5.18 shows a limited section of the data path for the same Fibonacci example introduced in Section 5.1.5. The portion of the data path shown by Figure 5.18 shows the connections of the output-referenced registers with respect to the operations performed on them in the 'fibonacci' function.

5.3.6 DDF file format change

The MOODS internal data structures can be dumped at any stage during optimisation and after the post-optimisation hardware generation step has occurred. The file format (see Appendix D.2) is entirely proprietary and mirrors the essentials of the entire data structure. The reason that it is mentioned here is that a back-end translator exists (see Appendix A.5 DDFLink), which generates the final structural VHDL directly from the internal data structures stored by the DDF file. Due to the changes made for recursion in the MOODS data structures, specifically to the module and ICODE instruction structures, this information is incorporated into the DDF file format and the relevant changes made to the back-end translator.

5.4 Recursion timing

The overhead of recursion is in two parts. The first is the area overhead, which can be attributed to the space required by the stack, stack pointer, return addresses and their decoders, mirror registers and all the extra controlling signals used to integrate the control path with the data path. The second overhead is the timing required to implement a recursive call. These cycle-based timing requirements form the extra cycles required for stack modification, mirror register usage and the recursive call itself. As the underlying control flow is generated using ICODE instructions, the final implementation timing is dependent on the optimisation of these instructions. However, the scope for sharing clock cycles for the auto-generated ICODE is limited by the stack storage mechanism of the single RAM used to hold the dynamic data frames. As a result, MOODS always produces the minimum timing flow without impacting on the total area of the design.

5.4.1 Recurse control node

The recursive call control node is designed to mirror the timing characteristics of a normal call node. The only ICODE instructions implemented by a call node of any type are call instructions. This means that no other instructions are scheduled while the call node is active. This is because a call node executes at the same time as the control nodes that implement the state machine of the called module, where the overlapping time slots do not allow any other instructions to be implemented in the call node, as the call node effectively has no time in which to schedule instructions. Both call nodes are designed with these timing characteristics to minimise the time taken for a call.

If ICODE instructions were allowed to execute in a call control node, then the time taken to execute these instructions would impinge upon the time remaining in the start node of the module being called. Data dependency checks would need to be made across the module call boundary for every call made to the module. This is an impossible situation to optimise.

5.4.2 Stack modification

All stack modification is implemented by the *push* and *pop* operations described by Figure 5.10. This figure shows that a *push* is made from a write into a RAM variable and then an increment of the address used for the write. A *pop* is made from a decrement of the address, followed by a memory read at the new address location followed by a '*protect*' instruction.

Due to the ordering of the two instructions of the *push* operation, no data dependency exists between them, so they can both be executed in the same control cycle. Note that several successive pushes onto the stack create data dependencies between the increment of the stack pointer and the stack pointer being used as the address for the next *push* operation. This, and the fact that a RAM can only be accessed one address at a time forces every *push* operation to execute consecutively.

The *pop* operation is different in that the memory read is dependent on the result of the increment of the stack pointer used as the address. The extra '*protect*' instruction that follows preserves the validity of the return address across module call boundaries. This

extra instruction also has a side affect of forcing the stack pointer decrement operation and the memory read operation to be chained in the same control state. This means that a *pop* operation executes in one clock cycle also.

The sequential nature of these operations that use the same storage variable forces every *push* and *pop* operation into its own separate state. This is where most of the timing overhead of recursion occurs. Note that the input mirror register assignments occur in the sequence of *push* operations. The output mirror register assignments occur after the final *pop* operation.

5.4.3 Return address setup cycle

Return address manipulation is discussed in Section 5.2.7.3. Part of the discussion mentioned that the return address is required to be valid for every end-state of the controlling state machine. This is achieved with the insertion of an extra ICODE '*protect*' instruction where required. This has the effect of inserting an extra control state after the return address modification *pop* cycle. The cycle after the *pop* of the return address is then used as an end-state of the controlling state machine.

This situation is shown in Figure 5.19 below. The left side of the figure shows an example state machine. There are two separate state diagrams shown, where the left diagram represents the main controller module 'A' and the right diagram represents a recursive module 'B'. The call control node labelled as 'c3' makes the link between the two control flows. A single recursive call to module 'B' is made by the '*recurse control node*' labelled 'c8'. States 'c6' and 'c10' form the two end states of module 'B'.

The timing diagram to the right of the state diagram shows an example state machine flow for every token output of the control nodes shown. The extra signals that the call and recurse control nodes generate are also shown. The flow shows the initial call to the recursive module, followed by the recursive call back into itself. The alternative conditional route being taken in the second iteration of the recursive module breaks the recursive loop. Control is then seen to return to the node after the recursive call node by reaching the end node 'c6' before finally returning to the node after the initial call node from the second end-state 'c10'.

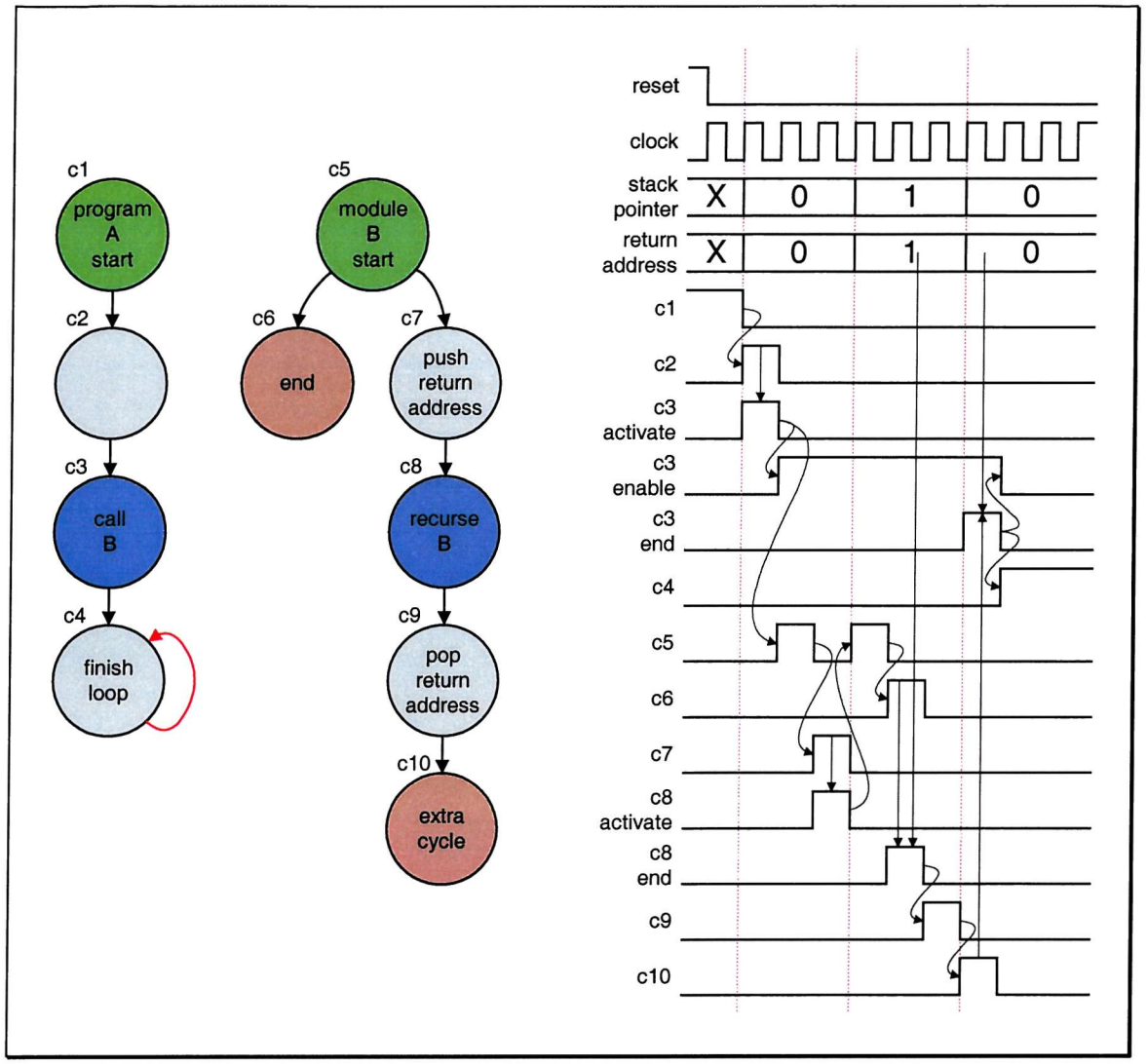


Figure 5.19 Example state machine timing flow

The need for the extra state after a *pop* of the return address is seen from the 'c3 end' signal. This signal is generated from the return address and is only set when the return address equals zero and when one of the end states is active. It can be seen that the return address is assigned to the correct value at the end of the state before the 'c3 end' signal is generated. If 'c10' were merged into 'c9', then the 'c3 end' signal would evaluate to false in this merged state, which would break the control flow.

The timing diagram also shows that the return address and stack pointer values are assigned their initial values in the first state 'c1', which means that their values update at the end of this state.

5.5 Impact on optimisation

The additional operations added to a design that control the data flow through recursive calls to the same procedures affects the final optimisations that MOODS carries out. The main limitation is that the additional ICODE operations added into the control flow force the design to include a number of sequential states in which the stack, mirror registers and return-addresses are modified. This affects the speed and size of the design produced using recursive procedure call methods.

5.5.1 Module ordering

When MOODS optimises a design using the simulated annealing algorithm, the order in which the modules are optimised is irrelevant, as each module is treated entirely independently. In contrast, the quasi-exhaustive heuristic algorithm requires knowledge of the call hierarchy, as it operates on modules in order, from leaf to root.

The reason for this is that to calculate the critical path length of each module, knowledge is required of the critical paths of all the modules it invokes. When a module is optimised, its internal timing may change and this change must be reflected back up the calling tree. The addition of recursive capabilities means that the algorithm to calculate the linear order of modules from the call graph is modified. The ability to introduce recursive loops in the module call graph prevents forming a linear leaf to root ordering of the modules; instead, a best approximation is sought.

5.5.2 Critical path calculations

Another impact of allowing recursive module calls is that the critical path calculation requires some modification. The modifications do not have any direct effect upon the final structural design produced by MOODS. The critical path is a requirement for optimisation, but begins to lose meaning when recursion is taken into account. This is because the path taken through a design no longer has a resolvable maximum length.

However, some value is required as the result of the critical path calculation. A critical path can be calculated for any module. If recursion were not allowed, then the critical path would be a count of the greatest amount of control states that it takes to reach any end

node from the start node of the module. Any calls to other modules add the critical path of the called module, not just the state required for the call control node. As recursion is now allowed, this calculation requires modification. It was decided to place a limit on the recursive depth of critical path calculations so that if a call is found to a module that has been calculated before, then the cost for the call is taken as one cycle.

This solution produces module-local critical path calculations dependent upon the initial module to have the critical path calculated. As the critical path is used to determine the shareability factors of the data path units and control states, and the modified critical path calculation is conservative in its estimation, the only noticeable side effect is a slight slowing of the optimisation process [102].

5.6 Problems and Improvements

The main problem with the present implementation of procedural recursion is the possibility of stack overflow. As the additions effectively make static variables into dynamic variables, the dynamic values need infinite storage space in theory. Obviously, storage space of this capacity is impossible.

5.6.1 Stack overflow

The consequence of stack overflow is to over-write dynamic data stored in another stack frame back down the call hierarchy. The effects are only seen when control returns back down the call stack, to find corrupted data. As the return address is included in the stored data on the stack, it is possible for the corruption of data to break the returning control flow, which could effectively halt the design in an incorrect infinite loop. The alternative to this is simply to produce the wrong result, but the effects cannot be predicted due to the dependence on dynamic data produced at runtime.

There are four solutions to this: The first is to try to cope with stack overflow when it occurs by some exception handling system built into the final structural design. This does not generate the correct result, but tells the user that an error has occurred before the effects due to the error are seen.

The second solution is to try to allocate enough space so that it never does. The second solution is not really a solution, in that it just moves the problem further away, but the first solution always results in the breaking of behaviour, which is not very desirable.

The problem can be alleviated by careful design of the use of recursion in the first place and by selecting a stack size that is capable of holding the full stack depth for the biggest problem being solved by the design.

A third solution that does not break behaviour is to have some secondary storage solution that is used in exceptional circumstances to page in and page out large chunks of the stack frame. An example would be to use the heap system described by Chapter 4 as the space for secondary storage. This system does not break behaviour, but it impacts on speed, as large blocks of data are transferred.

A fourth solution that is slightly neater than the third is to use stack frame windowing, as used by the SPARC RISC processor [29]. This enables better time utility of secondary storage, as it is only used when stack overflow or stack underflow occurs. If the heap management system is used for secondary storage, then small amounts of stack frame data can be allocated when required.

5.6.2 Multiple stacks

Section 5.2.7 describes an alternative to a single stack memory block. Its solution is to have multiple stack blocks that contain one dynamic stack copy of a single variable. The use of this method enables faster stack modification due to the inherent concurrency of multiple blocks of data. It also allows more efficient data path widths to be used. A trade-off in this solution may be to share these dynamic blocks of data on individual terms, say between variables stored for different recursive modules. However, this method negates having a single external SRAM based stack.

Chapter 6

Practical synthesis

This chapter describes demonstrators built using the capabilities described in the previous two chapters. Section 6.1 introduces the physical system structure that is used for the demonstrators. The first demonstrator described in Section 6.2 shows the use of dynamic memory. The second demonstrator described in Section 6.3 shows recursion in use, along with further use of the heap-allocated data. Finally, Section 6.4 contains a comparison between different implementations of a small language parser, built upon different platforms with differing implementation language restrictions. Timing results are obtained from simulation time measurement and computer runtime results.

6.1 Demonstrator system

The system created to demonstrate the capabilities of MOODS is realised with the use of multiple printed circuit boards (PCBs) that are designed for this specific purpose and have been built by a third party PCB manufacturing company.

The boards are designed to be completely self-contained with expansion ports provided to allow multiple boards to be linked together to form a larger overall system, or to accept other types of daughter board.

Two types of board were built, each with a different set of ancillary components that can be used. Each board has, at its heart, an FPGA that will contain any designs produced by MOODS.

6.1.1 First PCB

The board designed to demonstrate the capabilities of the dynamic memory system [103] has a set of onboard devices and a set of external interfaces. The core of the system is a

XILINX FPGA [104]. The package used for the FPGA is a pin-grid array with 475 pins. This allows only a single type of FPGA to be connected to, the XILINX XC4062XL chip. At the time of designing the PCB, the FPGA was of medium sized capacity. The size information is given in Table 6.1 below. A CLB or Configurable Logic Block is the basic building block from which all gates are created within the FPGA, and are usually held in a square grid pattern, with signal routing between the CLBs.

Device	CLB count	CLB matrix	Flip-flop count	Typical gate range
XC4062XL	2,304	48 x 48	5,376	40,000 - 130,000

Table 6.1 Available XILINX devices using the PG475 package

The set of external interfaces includes a VGA adapter, which is driven from an onboard video DAC and sync signal buffers. The DAC and buffers are directly connected to the system FPGA, which is used to generate all of the signals that drive a standard VGA monitor.

The VGA adapter is included on this board with the design of the VGA controller system, explained in Appendix A.1. This system requires a bitmap ROM for 256 characters, each contained by a square of 8 by 8 pixels. A single bit, being either set or reset, stores the state of each pixel in the bitmap. A whole row of a single character is accessed at the same time, requiring that the ROM have an 8-bit data path. Hence, the number of addresses used to store the entire bitmap is 2K. This 2K ROM has space designed into the PCB for it.

A standard buffered serial port external interface is provided in order to communicate with any other system via this standard method. This can be used to transfer data between systems (at a relatively slow rate). An onboard chip that converts between the 5V system voltage levels and the ±9V levels expected by the serial port interface and vise versa provides the buffering. The serial port controller is part of the system created inside the FPGA. Appendix A.3 explains this.

Buffering is also provided for two PS2 external ports, which can be used to interface with many standard devices, including keyboards and mice. Both of these standard devices have controllers designed for them, where the controller is again part of the FPGA system. The keyboard controller is explained in Appendix A.2 and the mouse controller in [105].

Two 96-way connectors provide two general-purpose expansion ports. The pins of these connectors are directly connected to a number of the I/O pins of the FPGA. One connector also provides power and ground supplies, so that the interfaced boards may be powered directly by the PCB system. The other expansion port includes every FPGA programming pin, so that the FPGA may be programmed externally.

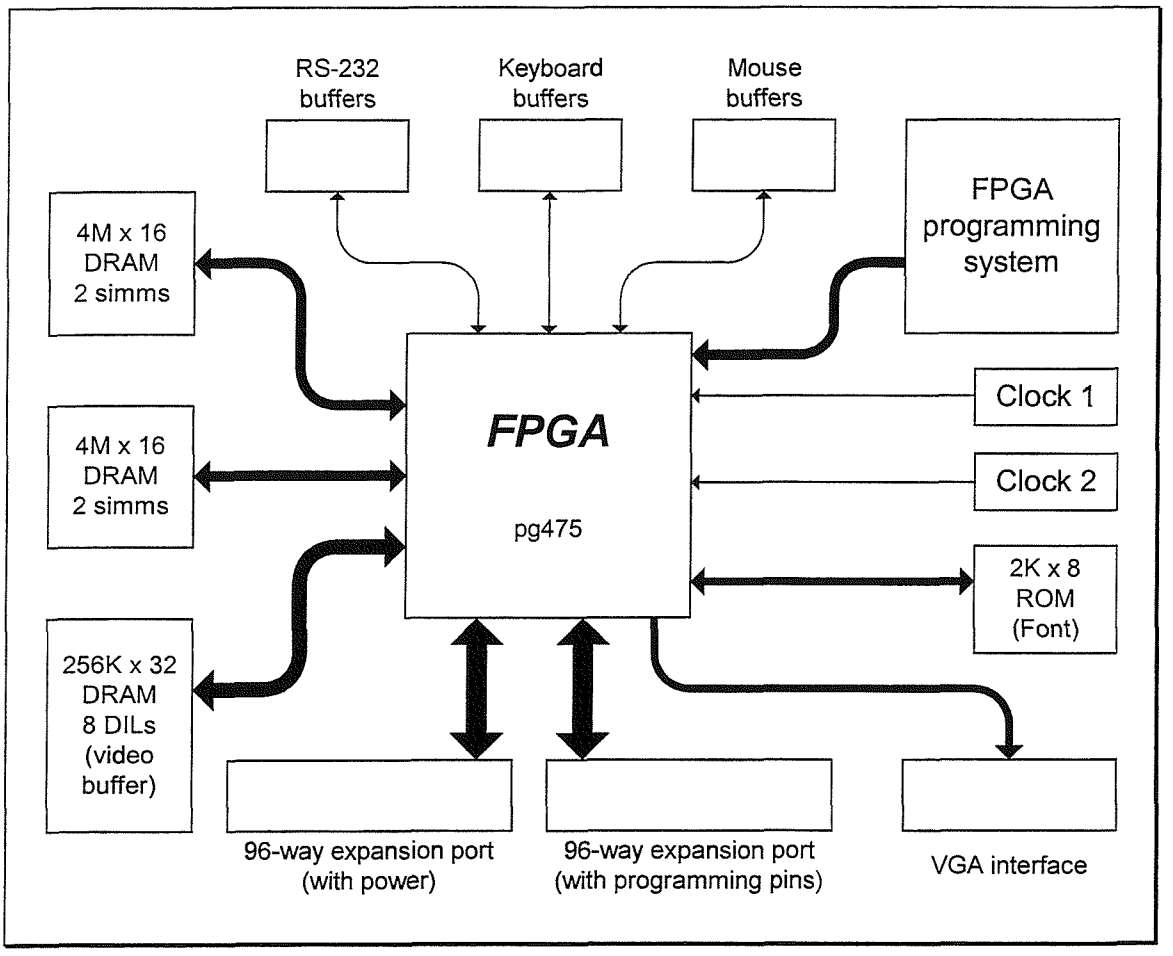


Figure 6.1 First PCB System connection

The alternative to external programming of the FPGA is to perform this action using the onboard system, created specifically for this purpose. The FPGA is capable of being configured in various ways. The two methods supported by the PCB are slave serial mode (default) and master parallel-up mode. The configuration mode is set with a number of DIP-switches. Slave serial mode is used when downloading the configuration directly from an external PC during the development process and master parallel mode is used when a design has been settled upon and the system is allowed to program itself from an onboard EPROM, which has space designed into the PCB for it.

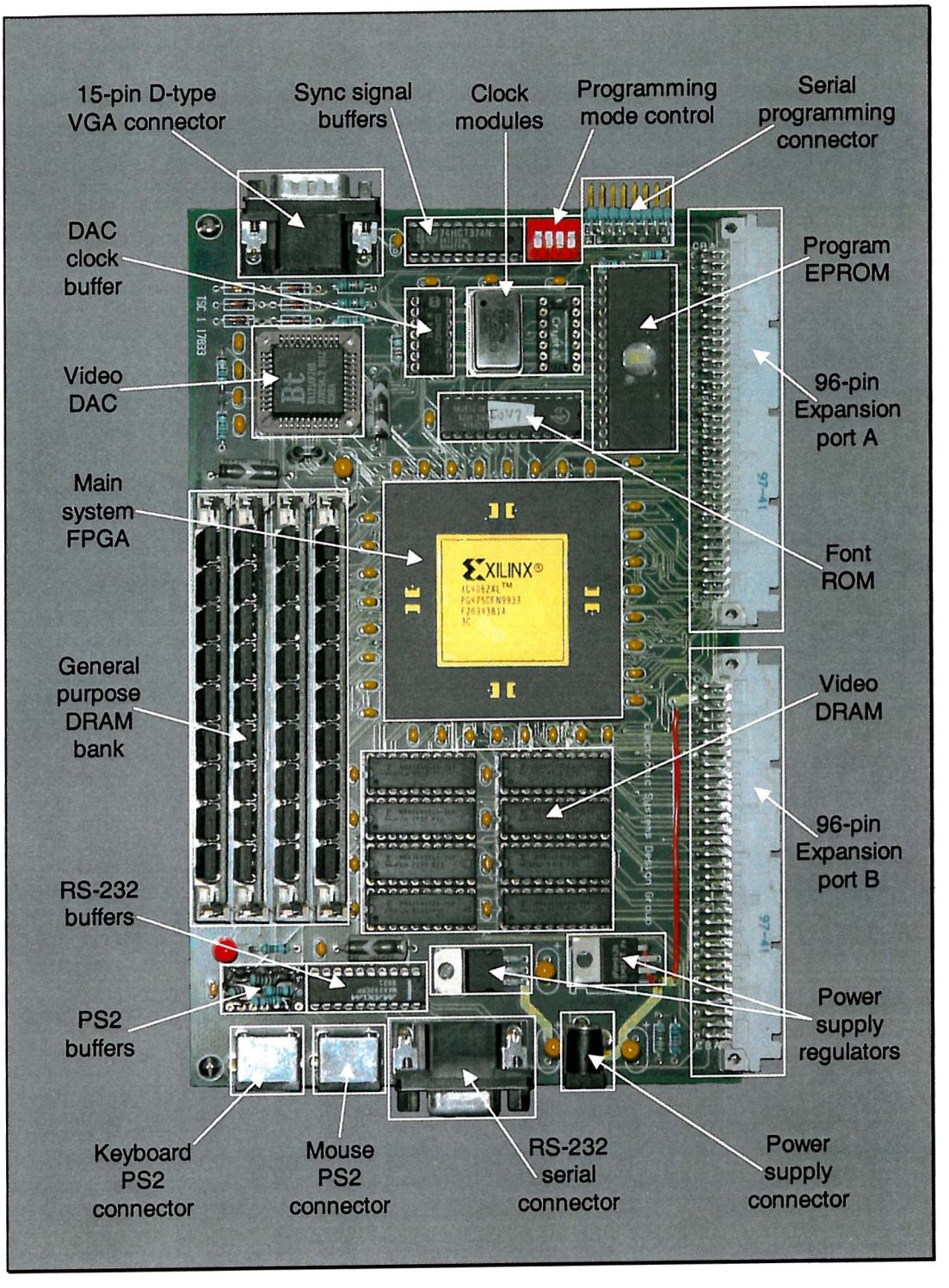


Figure 6.2 First PCB System layout picture

Figure 6.1 shows the connection between the various devices and external interfaces on the first board, while Figure 6.2 shows a picture of the physical board, with the various devices and interfaces highlighted.

All designs produced by MOODS are synchronous systems driven from a fixed clock signal. Two clock module inputs are provided for this purpose. Two clocks are provided so that multiple internal systems may be run asynchronously within the same FPGA.

The VGA system is designed to use a DRAM bank as a frame buffer from which the monitor signals are rasterised. All displayed pictures are then formed from writing to the frame buffer memory directly. This frame buffer memory is stored within a number of DRAM chips, which are accessed in parallel to form a 32-bit data path.

Two other banks of DRAM in the form of 4 32-pin SIMMs are included as onboard devices. Each SIMM is capable of storing up to 4MBytes, with 1MByte usually used. This memory is the foundation storage for the dynamic memory system described in Chapter 4, but may be used for any other purpose. Each SIMM has a completely separate 8-bit data path, while the address and control paths are shared between a pair of SIMMs. This enables the memory space to be used in three configurations: As two separate spaces of 16-bit data paths, each with a maximum of 4MWords, where 1 word is 16 bits; as a combined address space of 8MWords, still with a 16 bit word or as a combined data space of 4MWords, where one word is now 32 bits.

The final configuration is used by the heap management system, where the two SIMMs are combined to form a 32-bit data path, with a maximum of 4MWords of address space. The actual address space used by the demonstrators is 1MWord.

6.1.2 Second PCB

The board designed to demonstrate the floating-point capabilities of MOODS [106,107] is designed for this purpose over all others. It is not designed for the demonstrators described in this chapter. For this reason, a different set of ancillary components interfaces to a different, larger core FPGA. The general structure of the system follows the same style as the first PCB, in that a core FPGA is the central system unit, with a number of satellite components and external interfaces, both buffered and directly connected. A fuller description of this board is found in [106].

The core FPGA uses the same pin grid array style, but this time within a 559-pin package. This enables a choice between FPGAs to be made, with the XILINX XC40250XV being

the chip used by the final demonstrations. This chip is capable of holding a configuration that has about four times the number of CLBs of the XILINX XC4062XL FPGA used on the first board, which is the best indication of relative capacity. A number of devices may be used in place of each other, with their relative sizes shown in Table 6.2.

Device	CLB count	CLB matrix	Flip-flop count	Typical gate range
XC4085XL	3,136	56 x 56	7,168	55,000 - 180,000
XC40125XV	4,624	68 x 68	10,336	80,000 - 265,000
XC40150XV	5,184	72 x 72	11,520	100,000 - 300,000
XC40200XV	7,056	84 x 84	15,456	130,000 - 400,000
XC40250XV	8,464	92 x 92	18,400	160,000 - 500,000

Table 6.2 Available XILINX devices using the PG559 package

A large proportion of the external interfaces are mirrored from the first board, with the inclusion of an RS232 serial port interface, two PS2 interfaces (keyboard and mouse) and the twin 96-way expansion ports, with the same pin connections where required for compatibility.

However, the VGA interface is not provided on this board, which means that the frame buffer DRAM memory and the text bitmap ROM are also not required. The one concession to DRAM random access memory storage is with the provision of a single 32-pin SIMM socket.

Two clocks are provided in exactly the same manner as in the first board, which allows multiple internal asynchronous clocks to be used.

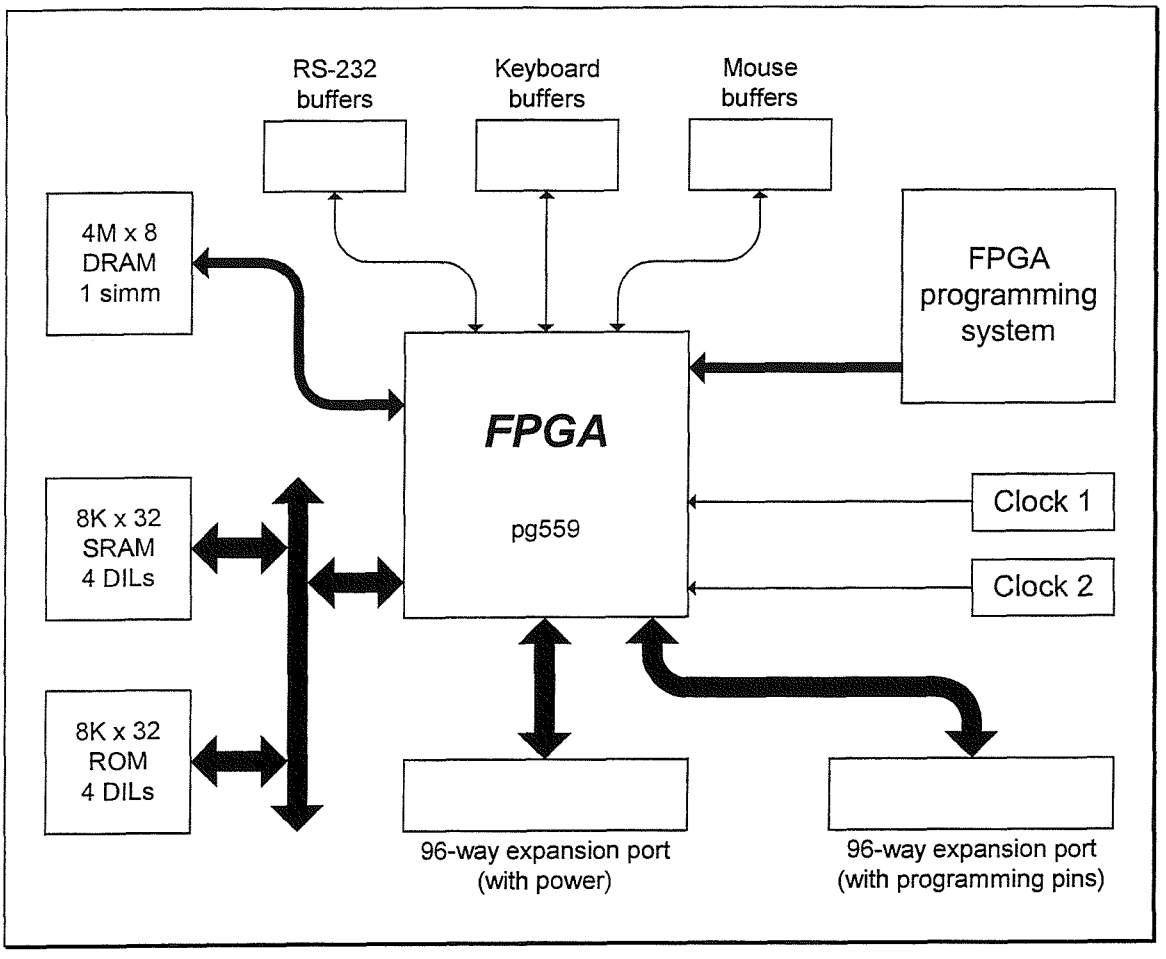


Figure 6.3 Second PCB system connection

The FPGA configuration program is downloaded into the FPGA in exactly the same manner as with the first PCB. The same two modes of configuration are supported, with the provision of an onboard EPROM, which is capable of storing a single program.

As the floating-point systems generated by MOODS require a number of lookup ROMs and a fast scratchpad memory, an onboard ROM and SRAM bank [100] are provided on the PCB. These share the same address and data path busses, with separate control busses for distinction between the sources of data between the two.

Figure 6.3 shows the connection between the various devices and external interfaces on the second board, while Figure 6.4 shows a picture of the physical board, with the various devices and interfaces highlighted.

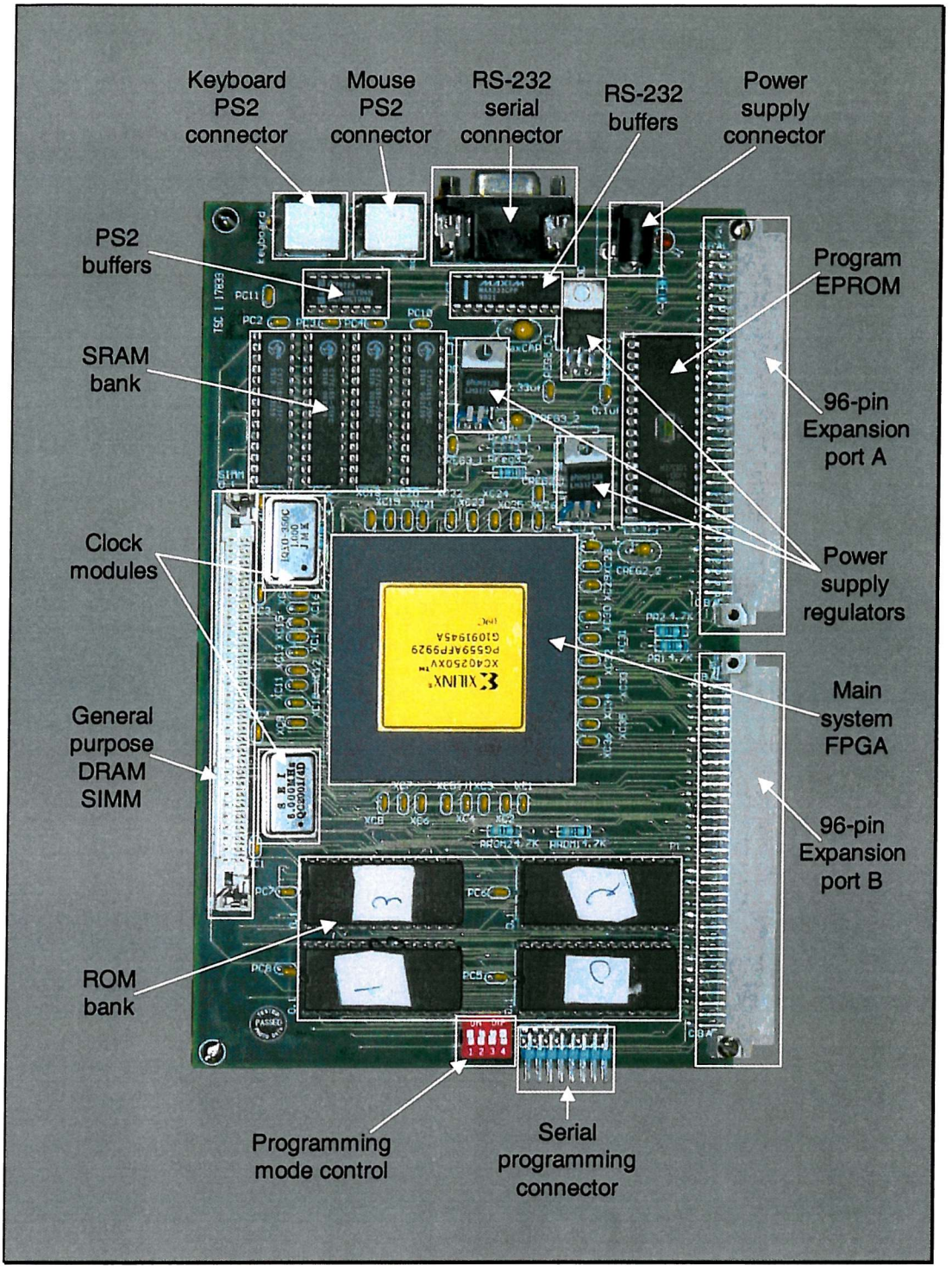


Figure 6.4 Second PCB system layout picture

6.1.3 System structure and partitioning

During the development process of the demonstrations, it was realised that two graphical displays were required, one to display the output user interface of the demonstration itself and the other to display a real time representation of activity in the heap management system.

For this reason, at least two boards with the VGA interface buffering were required, along with a method for connection between the two. It was decided to implement the core design on the board that controls the user interface VGA signals and to implement the heap management system and heap monitor VGA output on the other board. However, with the growing size of the demonstrations, a single board containing the XC4062XL chip was found to have insufficient capacity for both the VGA driver design and the core system design.

Efficient partitioning between both the core board and the heap management board was considered, as the heap management system does not require the full capacity of the other XC4062XL chip, even with the real-time heap monitor extensions to the heap management system. However, it was decided to opt for the more expandable option of using a third board to contain the full core system, where the main system board uses the second PCB described in Section 6.1.2. This configuration was chosen with the additional knowledge that an audio buffering system was required by one of the demonstrators, and no audio interfaces existed on the designed PCBs, leaving a further subsystem to be built.

6.1.3.1 Motherboard

A detailed diagram of the four system boards is shown in Figure 6.5. This shows an interface partition between each board, with the core system being at the centre of all the interfaces.

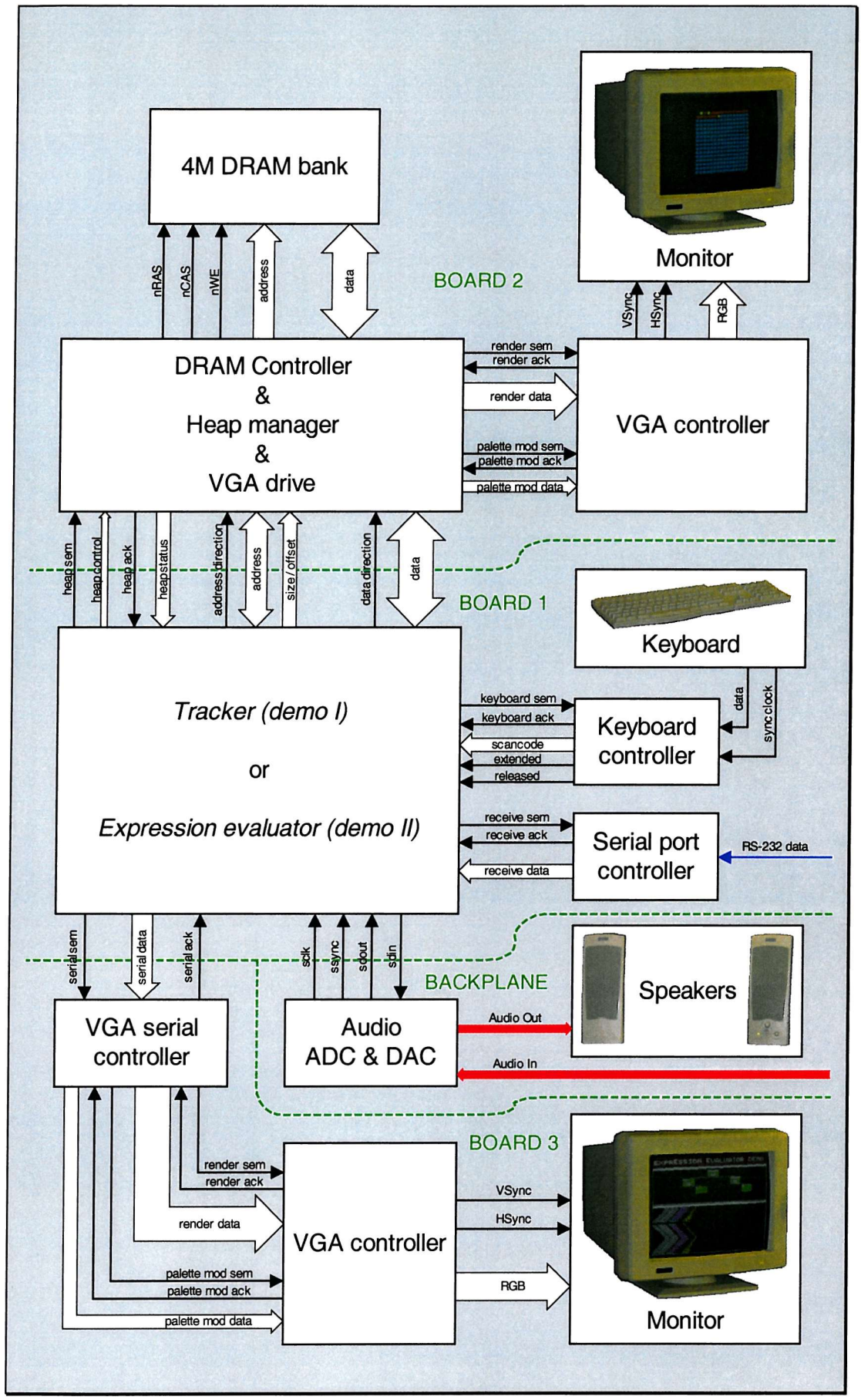


Figure 6.5 Demonstrator system partitioning and connectivity

The physical system is implemented with the use of a back-plane board, which each subsystem board plugs into via their 96-way expansion connectors as daughter-boards. The back-plane board was built manually and contains direct linkage between the four subsystems and an expansion of the configuration programming system for the core system, which allows more than one core design to be configured. The board structure is shown in Figure 6.6 below.

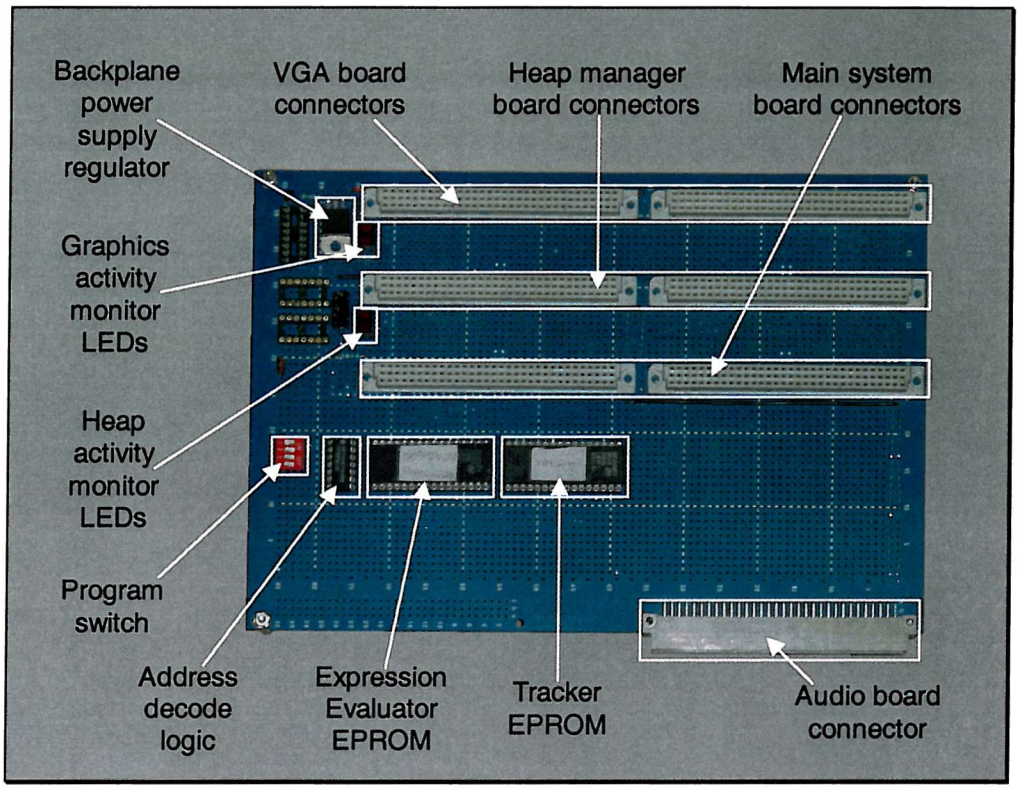


Figure 6.6 Handmade backplane board

The three main daughter boards plug into the double expansion ports at right angles to the motherboard, while the audio interface board plugs sideways onto the motherboard. The reason for a separate audio interface board is that this board had already been produced for another project that used the single FPGA system described in Section 6.1.1. The audio board is further described in Section 6.1.3.6.

6.1.3.2 Communication

All of the physical connections between the four subsystem boards are created from manually soldered direct wire connections. While the demonstrators progressed in time, it was realised that synchronous communication, with each system using the same base clock was not feasible due to the skew and amount of interference produced by having a

single central clock module driving every subsystem. With the amount of noise produced by the clock modules and the lack of a ground plane within the motherboard, it was found that each subsystem was prone to resetting also, with a centralised resetting mechanism.

The resetting problem due to the centralised clock system, reset system and all other communication is solved by two methods, with the clock skew problem being solved also. The main solution is to have each board provide its own clock signal using the onboard space provided for them on each PCB and to have each system communicate asynchronously.

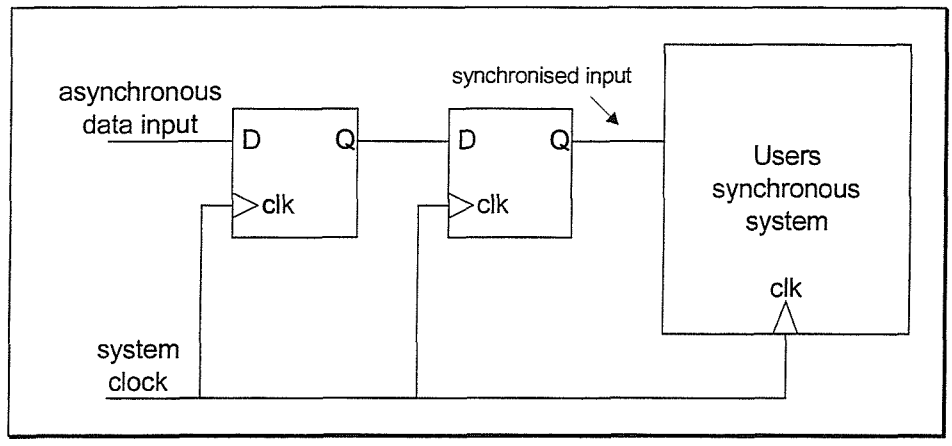


Figure 6.7 Asynchronous double buffering

One drawback of asynchronous communication is that extra buffering is required between the asynchronous systems for all inputs to a design. This double buffering shown in Figure 6.7 is used to remove meta-stability [108] that can occur when the input signal changes on the clocking edge of the system clock. The double buffer serves to greatly decrease the probability of an erroneous transmission of data. One drawback is the added communication delay that the double buffer introduces. The clock skew problem is completely removed with the removal of synchronous clocked systems.

The replacement of the centralised resetting mechanism with a distributed resetting mechanism with a start-up synchronisation system solved the resetting problems. The problem was found to be noise on the single shared reset signal. By allowing each board to reset itself, the reset signal has no opportunity to gain enough noise to falsely reset each system. Each system then synchronises itself to the others at start-up by data transmission between the systems, which has no impact upon the now distributed resetting mechanism.

6.1.3.3 Main System board

The main system board, shown in Figure 6.5 contains a larger FPGA than the boards designed for the VGA interface. It is for this reason that the core demonstrator designs are implemented using this board. It can be noted that both demonstrators could have been implemented fully within a single FPGA found on the second board if only the boards were designed for this purpose, with two VGA outputs, banks of DRAM and the audio interface required.

Both demonstrator designs share the keyboard interface, but the serial port interface is only used in one. Both designs use the heap management system board, which communicates via 73 expansion port pins and the user interface VGA system board that communicates via 15 expansion port pins.

The heap manager has a reduced pin count through the use of bi-directional address and data busses described in Section 4.2.2.4. Additional internal double buffering is provided for asynchronous communication.

The external VGA interface has a reduced pin count from the internal VGA interface with the addition of a time multiplexed serial communication method, whose controller is found within the programmed VGA output board. A limited set of equivalent VGA interface procedures are provided for the core designs.

A different clock frequency is used by each core design. Demonstrator I uses a 12MHz clock, mainly due to the audio system interface requiring synchronous communication at this speed, where the audio ADC and DAC derive the sampling speed from the clock rate. Demonstrator II does not require any particular speed for subsystem communication, so this allows the design to be optimised for area, having a reduced clock rate of 10MHz provided by a separate clock.

6.1.3.4 Heap manager board

The heap management algorithm described in Section 4.3 is implemented on the heap manager board. The initial implementation of the heap manager was modified to produce a real-time memory map monitor also. The provision of a VGA system on which to view the memory map requires the use of a 25MHz clock speed as input to the subsystem. This

clock is used directly by the VGA controller system, while the heap management system uses an internally divided clock of 12.5MHz. This frequency allows greater scope for operation chaining in the core DRAM controller and the heap management algorithm itself.

The implementation of the heap manager used by both demonstrators has the 1MWord address space divided into 256 pages. Each page has 4KWords from which to allocate objects. Each word is 32 bits. This implementation allows for 255 differently sized objects to be allocated at any one time, with an object size of up to 4KWords minus the page header size of 6 words.

An example of the displayed memory map is shown in Figure 6.8. The information is displayed in real time, which reduces the level of information that is capable of being displayed to general information about each page.

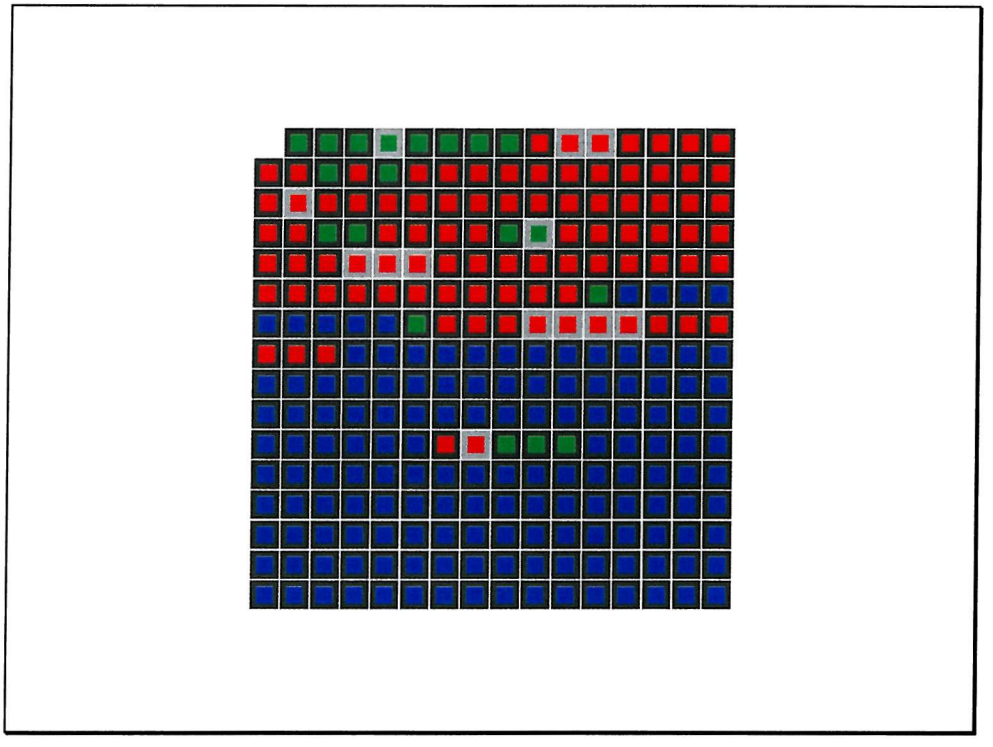


Figure 6.8 Example real time memory map picture

The information displayed is a 16 by 16 square representation of the 256 pages of the memory space. Each page has two types of information displayed about it. The colour of the internal square represents the allocation status of the page, where blue indicates a page free of objects, green indicates that a page has objects allocated within it and has space for

more and red indicates that a page is totally full of objects. The shading around the central colour determines whether the page is being accessed, with a lighter shade showing an access has occurred within one raster period ($1/60^{th}$ of a second). A dark border to the central page colour indicates that the information in the page has not been accessed within the same period.

The grid of pages is ordered from page 0 in the top left hand corner, counting upwards across the grid first, meaning that page 15 is in the top right hand corner. Page 16 is on the left hand side of the next row down, with each row containing increments of 16 pages, leaving page 255 displayed at the bottom right hand corner. Note that page zero is left blank, as it does not contain user data. This page is used as the active page lookup table for the different object sizes being created.

```
Design Summary:
   Number of errors:        0
   Number of warnings:     35
   Number of CLBs:           1103 out of  2304   47%
      CLB Flip Flops:      837
      CLB Latches:           0
      4 input LUTs:       1907 (4 used as route-throughs)
      3 input LUTs:        374 (99 used as route-throughs)
      32X1 RAMs:            32
      16X1 RAMs:            44
   Number of bonded IOBs:     217 out of   384   56%
      IOB Flops:           210
      IOB Latches:           0
   Number of clock IOB pads:    1 out of    12    8%
   Number of TBUFs:            32 out of  4800    1%
   Number of BUFGLSs:           2 out of     8   25%
Total equivalent gate count for design: 26841
Additional JTAG gate count for IOBs:    10416
```

Figure 6.9 Heap manager size statistics

The heap management system is further explained in Appendix C.5. The design summary log for the final implementation of the heap manager, including the VGA system and the drawing process within the heap manager is shown in Figure 6.9. The total design takes 47% of the capacity of the smaller FPGA, with 1103 CLBs used.

6.1.3.5 Graphical display board

The interface connecting the graphical display board to the main system design is via a reduced pin count serial interface. The serial data controller forms the front-end that drives the VGA controller in the graphical display design. The serial controller is detailed in Appendix C.4. The controller converts a number of serial words into a form that directly

drives the VGA interface. The core designs that use the graphical display board via the serial interface perform the graphics commands via a limited set of interface procedures, which have near direct equivalent VGA interface procedures.

Both the serial VGA instruction controller and the VGA controller take a small percentage of the area contained by the graphical display board's FPGA. The placement and routing log shown in Figure 6.10 shows that only 23% of the FPGA capacity is used. The clock used to drive all systems in the FPGA is another 25MHz clock, which forms the dot-clock rate for the graphical output.

```
Design Summary:
   Number of errors:        0
   Number of warnings:     17
   Number of CLBs:            539 out of  2304   23%
      CLB Flip Flops:      504
      CLB Latches:           0
      4 input LUTs:        944 (5 used as route-throughs)
      3 input LUTs:        155 (51 used as route-throughs)
      16X1 RAMs:            44
   Number of bonded IOBs:     111 out of   384   28%
      IOB Flops:            91
      IOB Latches:           0
   Number of clock IOB pads:    1 out of    12    8%
   Number of BUFGLSs:           1 out of     8   12%
Total equivalent gate count for design: 12943
Additional JTAG gate count for IOBs:    5328
```

Figure 6.10 VGA display driver size statistics

6.1.3.6 Audio interface board

The audio interface board is designed to plug straight into either PCBs interface port that contains the power pins. It is built as a general purpose audio I/O system, initially used by an audio filter design [109]. It is utilised as the audio I/O system in demonstrator I and is driven from the motherboard connection. The board, shown in Figure 6.11 is powered directly from the connecting system and contains a single ADC/DAC chip with audio connections via a low pass filter system.

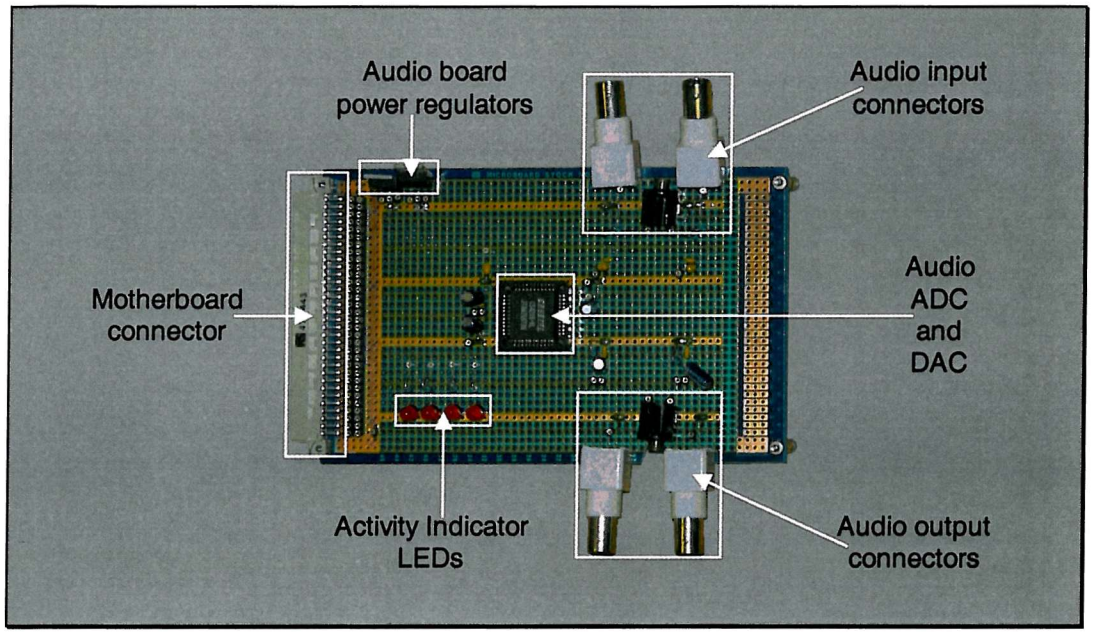


Figure 6.11 Audio board

6.2 Demonstrator I: The tracker

The definition of a 'Tracker' is an audio system that combines multiple audio sampling and playback with an audio sequencer, which allows music to be produced, that uses the stored samples as instruments. Many commercial tools exist to perform these actions both separately and combined and both in hardware and software.

The reason that a tracker is chosen as the demonstration system is due to the design being relatively complex, so when built in a small amount of time (1 month for the core), would prove the validity of the synthesis tool and of increasing the synthesisable subset of the source language to include dynamically created data structures. As the tracker is a real time audio system, it also shows the validity of using the dynamic data structures within a strict timing environment, with little concessions made for this.

The tracker demonstrator uses 35% of the main FPGA's capacity, which is shown in Figure 6.12, the design summary produced by the placement and routing stage.

```
Design Summary:
   Number of errors:        0
   Number of warnings:     18
   Number of CLBs:           2969 out of  8464   35%
      CLB Flip Flops:     2376
      CLB Latches:           0
      4 input LUTs:       5125 (34 used as route-throughs)
      3 input LUTs:       1436 (408 used as route-throughs)
      16X1 RAMs:           180
   Number of bonded IOBs:     102 out of   448   22%
      IOB Flops:           139
      IOB Latches:           0
   Number of clock IOB pads:     1 out of    12    8%
   Number of BUFGLSs:            1 out of     8   12%
Total equivalent gate count for design: 62965
Additional JTAG gate count for IOBs:    4896
```

Figure 6.12 Tracker design size statistics

A more complete explanation of the implementation details of the core tracker design is contained in Appendix C.6.

6.2.1 General overview

The user input interface is a standard computer keyboard, which could easily be modified to use a musical type keyboard in the future. Visual feedback is provided by a VGA graphics system that drives a standard monitor. The information shown on screen relates to all the internal dynamic data structures. A musical tune is built up from direct user input and interaction with the displayed information and outputted audio.

The system is capable of storing an arbitrary number of 16-bit stereo samples of arbitrary length. Each sample is recorded at the standard 44.1kHz CD sampling rate. There are 8 stereo mixing channels that are combined to form the single stereo output, along with the real time audio input. This means that up to 8 instruments can be played at once. Any sample can be played on any channel at any playback rate, which determines the pitch of the played note. A sequence of notes to be played at particular time points on particular channels can be built up dynamically and played back at any point. A musical tune is made from an arbitrary number of sequences, which themselves can be sequenced using a playlist of these sequences, where the playlist stores a list of sequences in an arbitrary, possibly repetitive order.

6.2.1.1 Data structures

As most of the data structures stored by the tracker system are best based upon a list structure, it was decided to create a general doubly linked list data structure for every different type of data that required list storage. This means that list operations such as creation and deletion of the lists, insertion and deletion of elements within the list and element iteration can all use the same basic procedures.

One disadvantage of VHDL within the context of dynamic memory structures is the very strict type adherence. **Access** types cannot be cast into referencing other types of elements. Because of this, a completely general linked list structure cannot be built. Instead, each list element must contain all relevant data types to be stored by the list. This is shown in Figure 6.13 below. An equivalent structure in the C language could store the various data pointers in a union structure, as each item is mutually exclusive and dependent on the type that the list contains.

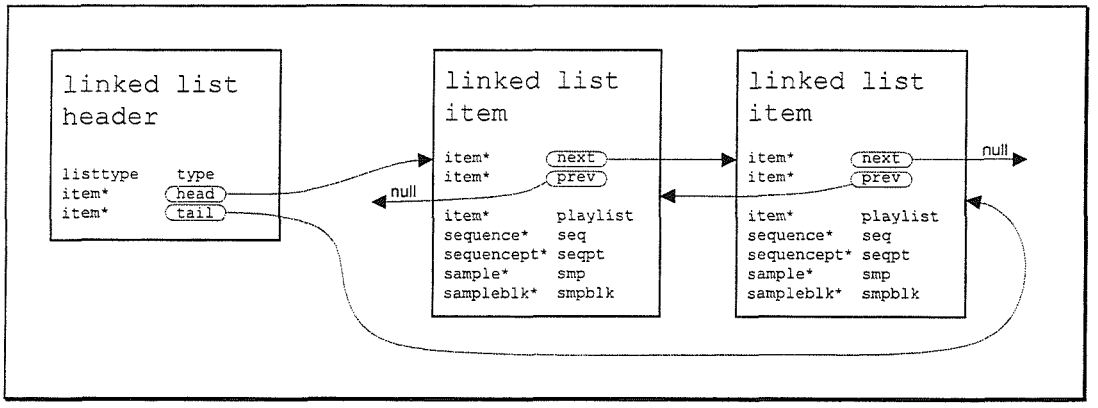


Figure 6.13 Linked list container with two elements

The general linked list structure has a header **record** that contains the type of elements to be stored by the list and a reference to the head and tail elements of the list. Each element has a next and previous element reference, where the head's previous element is null and the tail's next element is null. Only one type of element data is valid, dependent on the list type enumeration value stored in the header structure.

Five types of data are stored in the list structure. The playlist data is a reference to another linked list element. It should point to an element held within the sequence list. Each sequence in the sequence list has a list of points that determine when the notes are played. The sequence points have time and channel information along with a reference to a sample

that is held within the list of samples. Each sample in the sample list has unknown length when recording, so the sample is stored as a number of sample blocks, which each contain a fixed number of stereo sample values. This general structure is shown in Figure 6.14.

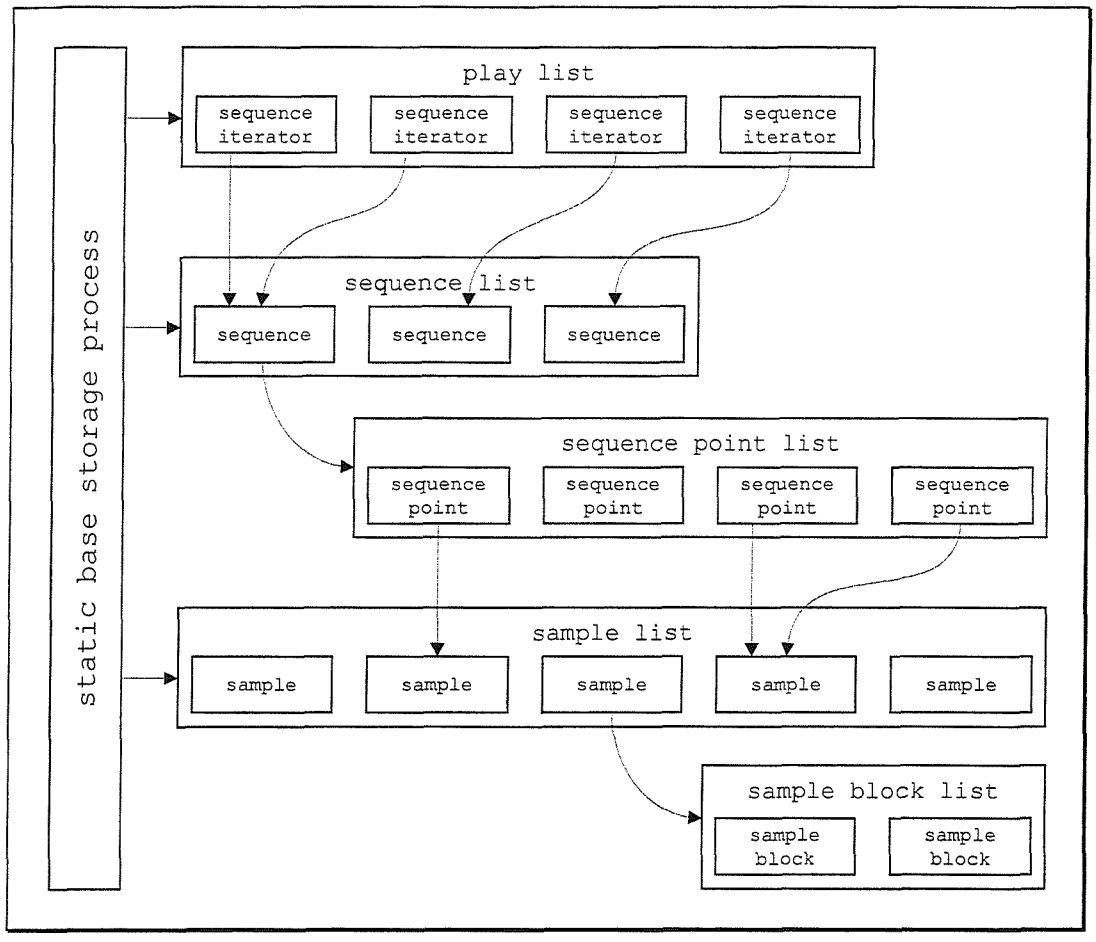


Figure 6.14 General tracker data structure linkage example

Of the five types of linked list structure, the play list, sequence list and sample list have a static base. The sequence point list is dynamically created whenever a new sequence element is added into the sequence list and the sample block list is created whenever a new sample is added into the sample list.

6.2.1.2 Processes

As the tracker design processes real-time audio streams, the design is split into two main processes. The main process handles the user input, creates and deletes the dynamic data structures and processes the real time audio data. As the latency involved with using the dynamic allocation methods has a known maximum value, it is possible to allocate new objects from the audio processing process so long as the audio streams are buffered using

a number of FIFO buffers. These buffers effectively remove the dependence on the allocation latency and place a dependence on the general memory access bandwidth for all memory operations in the core audio process. The FIFO buffers are created as concurrent processes.

The other core process drives the output user interface, the VGA controller system, through the interface provided by the serial version of the VGA interface. This process accesses the same core data structures as the main audio process with the use of shared variables that store the three static **access** types of the three base linked lists. The drawing is performed concurrently with the audio process, as the drawing time is unpredictable. The core audio process initiates all drawing whenever the user input changes or playback of the sequences and samples occurs.

6.2.2 User guide

There are four modes of operation of the tracker design, with playback of the sequences and samples possible in all but one of the modes. These modes relate to the part of the data structures that are being modified, with three modes dedicated to the modification of the three main linked list structures of the samples, sequences and playlist. The fourth mode allows the serial port download of a number of samples and sequences. The currently selected mode is determined by the selection colour of the viewed representation of the data structures shown by the VGA output picture (see Figure 6.15).

6.2.2.1 Sample mode

The sample mode is selected by pressing the 'F9' key. This mode allows for the modification of the sample list with new samples being created by recording them directly from the input audio stream. A sample is recorded by pressing the 'R' key for the record duration. This action creates and inserts a new sample after the currently selected sample within the sample list and then fills the sample block list with the audio stream data.

When not recording a sample, it is possible to move through the list of samples using the up and down arrow keys. This selects the current sample, which is used when inserting new notes into the sequence and for manual playback of the sample.

A portion of the currently selected sample is displayed on the output screen, and using the left and right arrow keys can scroll through the displayed sample values.

A sample can be set to play back as either looped or not looped. Pressing the 'L' key while in the sample mode toggles the looped status of the currently selected sample.

6.2.2.2 Sequence mode

In this mode, selected by the 'F11' key, new sequences can be inserted into the sequence list by pressing the 'Insert' key. Movement through the sequence list occurs by pressing the '+' and '-' keys on the keypad. Movement through the list redraws the representation of the currently selected sequence.

The currently selected sequence is drawn with the number of audio channels represented by 8 columns and the time positions as a number of rows. The arrow keys are used to select a time point and audio channel.

Insertion of notes at the current position occurs by pressing the 'Space-bar' key, which toggles the sequence point insertion mode. The recording mode is represented by a red position cursor, while the normal playback mode is represented by a green cursor.

The sequence point can have a note placed at the cursor position by pressing a number of keys, which are further explained in Section 6.2.2.5. The currently selected sample is used as the note instrument. A note can be stopped at the current position by inserting a stop bar by pressing the 'Tab' key. Highlighting the sequence point and pressing the 'Delete' key can remove a sequence point. All insertion and deletion operations increment the time position by one, facilitating faster sequence editing.

6.2.2.3 Playlist mode

This mode, selected by pressing the 'F10' key, allows complete control over the playlist. Movement through the playlist occurs by using the up and down arrow keys. Insertion and deletion of playlist elements occurs by pressing the 'Insert' and 'Delete' keys respectively. The currently selected sequence is the one inserted into the playlist after the currently selected playlist position. The sequence reference can be changed at the current playlist position by using the '+' and '-' keypad keys to iterate through the sequence list.

6.2.2.4 Download mode

The serial port download mode is selected by pressing the 'F12' key. All download operations after that point are controlled by the system that downloads the information into the tracker design. A computer software program has been developed to download a number of samples from standard 'wav' files and to download a number of sequences from proprietary format files. The global sequence playback speed can also be set by the serial download. The download program releases the tracker design back into its previous mode once the download is complete, where the tracker cannot play anything until the download completes.

6.2.2.5 Playback

General playback of the currently selected sequence is initiated by pressing the 'P' key. Pressing the 'O' key starts playback of the entire playlist. Pressing the 'Escape' key stops any sequence playback.

Playback of the currently selected sample at the various pitches that make up the standard musical scale is shown in Table 6.3. This forms one octave of the entire scale. The samples can be played in one of five octaves, selected from keys 'F1' to 'F5'. The 'F3' key selects the middle octave, with middle-C being played by key-press 'C'. As samples are recorded in middle-C, the playback of a middle-C note streams the audio data at the same rate as the sample is recorded.

Note	A	$A^{\#}$	B	C	$C^{\#}$	D	$D^{\#}$	E	F	$F^{\#}$	G	$G^{\#}$
Key	Z	S	X	C	F	V	G	B	N	J	M	K

Table 6.3 Played note to key pressed

The manual playback of notes allows polyphony by using the 8 audio channels. The first free channel not playing a sample is used to start the playback of the note. Releasing the relevant key stops the note playback. The same keys are used when inserting sequence point notes. Both the selected octave and the note key are stored as part of the sequence point.

6.2.2.6 User interface

The user interface is shown in Figure 6.15 (this figure is generated by direct simulation of the synthesised tracker design, utilising the simulator's PLI-interface within the simulation of the VGA display system).

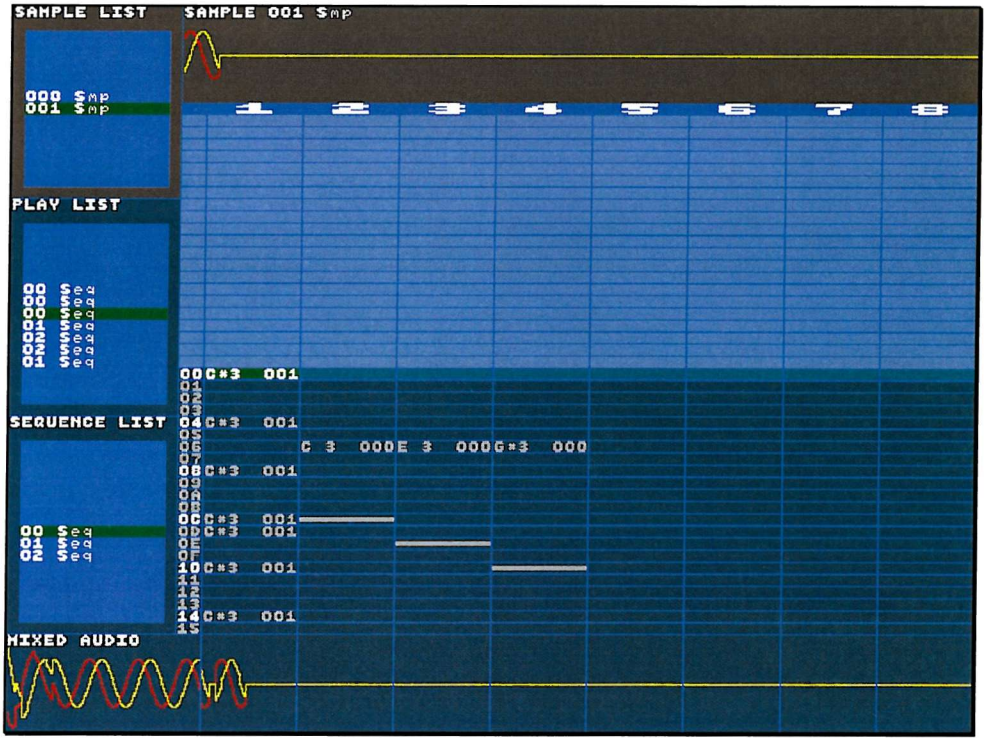


Figure 6.15 Simulated tracker screenshot

In this simulated example, two samples are held by the sample list, with sample '001' selected. The sample list is shown in the top left hand corner. The selected sample waveform is shown along the top of the screen. The tracker is in the sample mode, shown by the red background colour of the sample list and sample. All other windows have the blue background colour.

The playlist is displayed to the middle left of the screen, with 7 links into the sequence list. The third playlist element is selected, which is sequence '00'. The same sequence is selected in the sequence list, which is shown below the playlist. The selected sequence takes the most part of the displayed screen, with the 8 audio channels across the screen and the time points being shown down from the currently selected position, which is at time point '00' with channel '1' selected. The cursor is green, which means that the sequence is not in recording mode. In this example, most inserted notes are in channel '1', with C-sharp ($C^{\#}$) in the middle octave (3) using sample '001' being highlighted by the cursor.

The three lines shown in channels 2 to 4 denote a note-stop item, which halts all playback of the sample in those audio channels when the stop item is reached.

The bottom windows display a real-time representation of the audio streams being played on each channel. The left hand side larger window shows the mixed output audio stream. A simulated input sine wave is being mixed into the output stream.

6.3 Demonstrator II: The expression evaluator

This design is written to demonstrate recursion in behavioural synthesis, which is explained in Chapter 5. The demonstrator serves little other purpose than this. The point of this demonstrator is to recursively evaluate a binary tree expression. Most mathematical equations can be built from a number of operations stored within a binary tree, as most mathematical operations have one or two operands and return a single result. The operations supported are integer operations, where all integers are represented by 32-bit storage.

The design also uses the dynamic memory capabilities in order to build the recursive data structure of the binary tree. The demonstrator is designed to give a visualisation into the binary tree structure and to perform operations on the tree in a recursive manner.

The expression evaluator demonstrator core system uses 37% of the main FPGA's capacity, shown in Figure 6.16, the design summary of placement and routing.

```
Design Summary:
   Number of errors:        0
   Number of warnings:     80
   Number of CLBs:           3187 out of  8464   37%
      CLB Flip Flops:    1466
      CLB Latches:          0
      4 input LUTs:      5390 (50 used as route-throughs)
      3 input LUTs:       843 (145 used as route-throughs)
      32X1 RAMs:          256
      16X1 RAMs:            4
   Number of bonded IOBs:      95 out of   448   21%
      IOB Flops:          135
      IOB Latches:          0
   Number of clock IOB pads:    1 out of    12    8%
   Number of TBUFs:           256 out of 17296    1%
   Number of BUFGLSs:           1 out of     8   12%
Total equivalent gate count for design: 86433
Additional JTAG gate count for IOBs:    4560
```

Figure 6.16 Expression evaluator design size statistics

Note the use of 256 '32x1' internal RAM cells. This forms the implementation of the space required by the recursion stack.

A more complete explanation of the implementation details of the core expression evaluator design is contained in Appendix C.7.

6.3.1 General overview

The binary expression is built up directly from user input via the keyboard and viewed using the VGA output screen. There are two views into the operations performed on the dynamic data structures, the first being a view of the binary tree from a particular node in the tree and the second being a log of all results produced by the evaluation of the expression held by the tree.

The design is realised by a single process, as there are no timing critical sections to the system. This simplifies all accesses to the dynamic expression, as no memory conflicts are possible, which can occur with a multiple process implementation.

6.3.1.1 Data structures

The expression to be evaluated is stored by a binary tree structure, with an example of this shown in Figure 6.17. The tree is built from a single **record** type with left and right child references to the same type of **record**. Each tree node can either hold a fixed value, which is used for the leaf nodes of the tree, or an operation upon the left and right child operands.

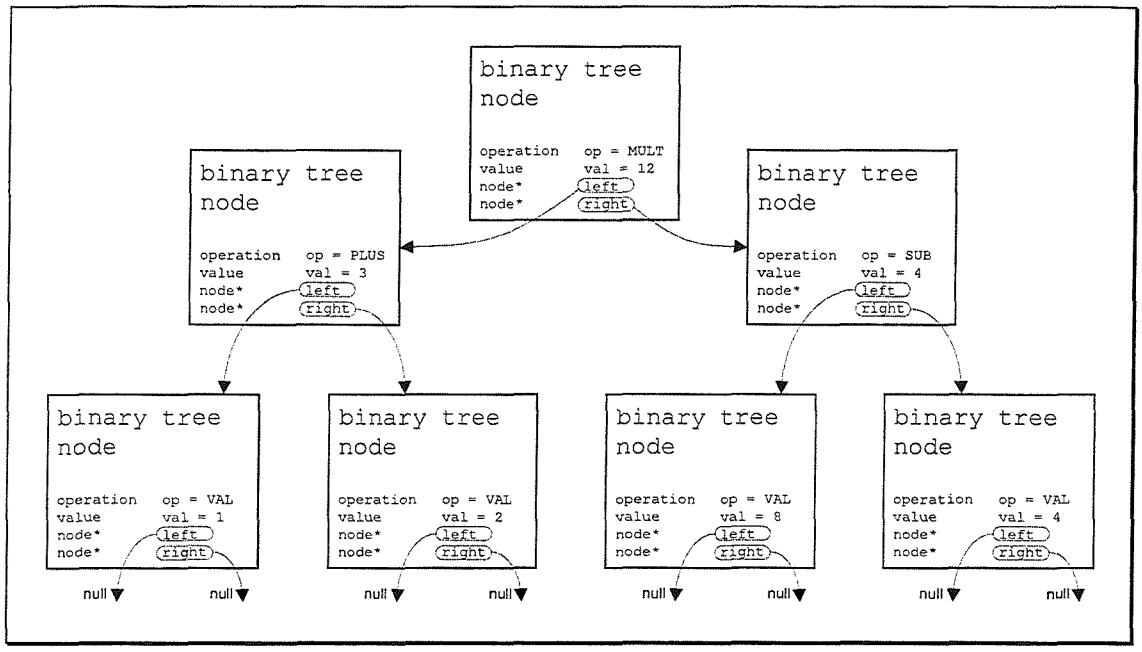


Figure 6.17 Binary tree container with 8 elements

Both unary and binary operations are supported, with the unary operations acting upon the right hand child only. The example shown evaluates the expression:

$$(1 + 2) * (8 - 4) = 12$$

The log of all results produced from the evaluator is created dynamically. The log is formed from a doubly linked list that contains a full line of text for each element of the list. The information stored for each character is a combination of the ASCII character code and colour. The linked list structure is similar to the data structures formed as part of the tracker demonstrator, except that only the one type of element is stored.

6.3.1.2 Recursive operations

As the point of the demonstrator is to highlight the use of recursion within a system, a number of procedures are written in a recursive manner. The first procedure produced is a recursive implementation of the factorial operation [110]. This operation is best computed iteratively, but it served to test the implementation of recursion throughout the design and integration process, so is included in this demonstrator.

The two core tree creation and deletion operations are performed by recursive procedures. The creation of the tree is handled by the evaluation function, and the operations in the evaluation function are directly controlled by the user input from the keyboard. The tree

node deletion is formed from a simple recursive procedure that calls itself, passing the left and right node references before deleting the node data itself.

The tree drawing algorithm is also implemented recursively, with each level of the binary tree being drawn by a recursive jump into the child nodes. This means that the tree is drawn in a depth first manner.

6.3.2 User guide

There are two modes of operation within the expression evaluator, relating to the implementation of the direct test of recursion from the factorial procedure against the evaluation of the expression.

6.3.2.1 Factorial mode

This mode is included to show a direct test of a recursive procedure in action. The factorial procedure is a unary operation that produces valid results from input integers in the range 1 to 15. Any input larger than 15 will produce a result that cannot be held by the 32-bit representation of the integer.

This direct test can only be made when not evaluating an expression. Pressing the 'F' key evaluates the factorial expression with a repeated loop of 1 to 15 as the input values to the procedure. The results of the factorial procedure can then be seen in the event log, with the log being indented as a representation of recursion depth (see Figure 6.18).

6.3.2.2 Expression evaluation mode

Once the evaluation procedure has been called, the design will halt inside this procedure awaiting user input. A tree node is available in the evaluation procedure, which is modified by the user to generate the expression.

Pressing the 'E' key makes the initial entry into the expression evaluation mode. If no root tree node exists, then one is created within the evaluation procedure. The generation of the binary tree is simply a case of evaluating the left and right nodes by pressing the 'L' and 'R' keys respectively. This recursively calls the same evaluation procedure, so that if the left or right nodes do not exist, they will be created in the same manner as the root node.

Pressing the 'Q' key makes a recursive return from the present level of the evaluation procedure. If a return is made from the root tree node, then control returns to the base process, allowing the factorial procedure to be evaluated directly again.

While in the user interface loop within the recursive evaluation procedure, the binary tree node at the present evaluation position in the tree can be manipulated. The list of valid node operations is shown in Table 6.4, along with the keyboard key to press for them.

Operation	Leaf value	Not	Factorial	Add	Sub	Mult	Xor	And	Or	Shift left	Shift right
Valid Operands	-	R	R	L, R	L, R	L, R	L, R	L, R	L, R	L, R	L, R
Key	V	N	F	+	-	*	X	A	O	left arrow	right arrow

Table 6.4 Expression operations

Other tree manipulation operations that are provided include the ability to swap left and right tree branch operands by using the 'S' key; the manual creation of left and right operand nodes by pressing 'C' followed by the 'L' or 'R' keys; and the recursive deletion of the left or right operands by pressing 'D' followed by the 'L' or 'R' keys.

Tree nodes are initially created using the leaf value operation, whose value may be incremented and decremented using the up and down arrow keys respectively. A leaf value does not have any valid operands by definition of not being an operation. If any left or right operands exist for operations that do not require them, the entire sub-branch is highlighted as invalid by the displayed tree view.

6.3.2.3 User interface

An example of the expression evaluator screen view can be seen in Figure 6.18, which is generated from direct simulation of the synthesised expression evaluator.

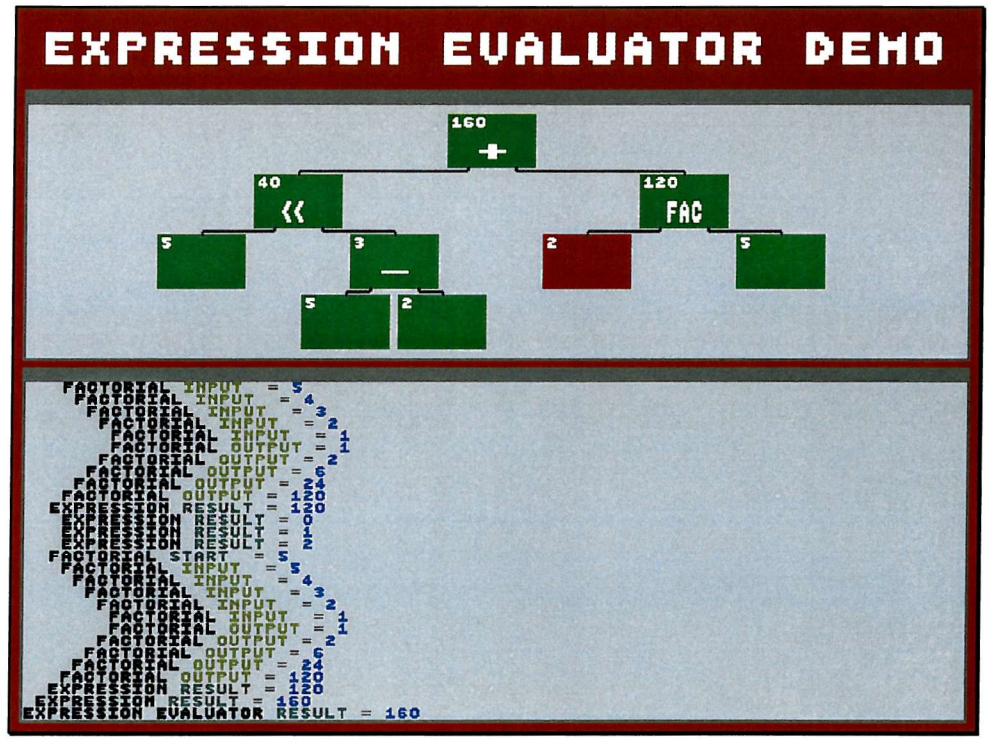


Figure 6.18 Simulated expression evaluator screenshot

The top window shows a snapshot of the entire tree view, with the viewed root node being the presently edited node. The expression shown in the example is:

$$(5 << (5 - 2)) + Factorial(5) = 160$$

Notice that the factorials left operand is shaded in a red colour. This is because the factorial is a unary operation, with only the right hand operand being used as input. The values in the top left corner of the representation of the tree nodes are evaluations of the value at the particular node. The type of operation is shown in the centre of the node.

The bottom window shows the bottom section of the full log that has been generated during all the tree manipulation and evaluation operations. Notice the indenting, which is a graphical representation of the recursion depth reached. The log may be scrolled up and down by using the up and down arrow keys and the page up and down keys when not evaluating an expression. Pressing the 'Escape' key removes the log contents.

6.4 Simulation experiment

The simulation experiment is written to perform relative timing comparisons between different implementations of the same design. The experiment tests both recursion and dynamic memory in both software and hardware domains.

6.4.1 Small language parser

The design that is used to test the system is an implementation of a small language parser. In fact, the language can be fully described using the BNF notation shown in Figure 6.19 and can be stored by a single binary tree data structure.

```
expression ::= '(' expression operator expression ')' | number
number ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
operator ::= '+' | '-' | '*'
```

Figure 6.19 Language description in BNF

The language is parsed into a data structure that is identical in form to the expression evaluator's expression tree data structure.

6.4.2 Comparable implementations

There are various items that are comparable between the different implementations. The experiment is set up primarily to measure the time taken, both of the internal phases of the design and of the total duration.

There are six different implementations that are measured, which are formed from a combination of hardware vs. software (on two platforms) and of recursive vs. non-recursive implementations of the same design. There are three phases to each design that are measured. The first is the data structure creation by language parsing, the second is a depth first traversal that calculates the result of the expression parsed and the third is the removal of the entire data structure tree.

It is possible to write any notionally recursive system in an iterative manner. However, the source code is invariably larger and more difficult to understand for any reasonable problem. The method used for the non-recursive implementation of the language parser is

to store the current tree position and returned data as part of the tree data structure itself. Both software versions are written in the C language, with direct equivalent hardware versions written using behavioural VHDL.

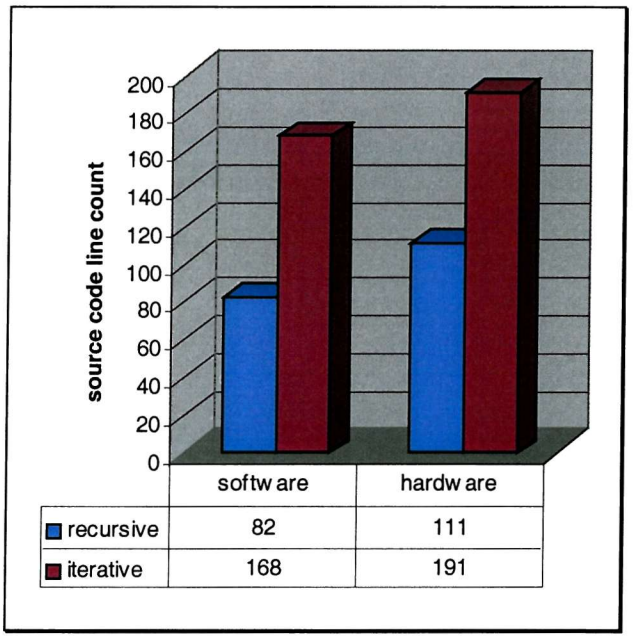


Figure 6.20 Source code line count for each implementation

The size results shown by Figure 6.20 show that for both the hardware and software implementations, the recursive version is almost half the size of the equivalent iterative version. Notice that the software version written in C is slightly less verbose than the equivalent behavioural VHDL description. The source code for each implementation is available in machine-readable form, which shows the more readable style of the recursive versions.

The hardware designs are simulated with a 25MHz clock, with the same 4Mbyte heap manager as used by the demonstrators. The memory space is simulated as standard fast page mode DRAM and the procedural recursion stack held in onboard simulated SRAM.

The software designs are directly measured on two platforms in order to gain a better understanding of the interaction between the many differences between them and the hardware being tested. The first platform is an Intel Pentium 75MHz PC, with 16 Mbytes of DRAM clocked at 33 MHz and 256Kbytes of SRAM processor cache. This machine uses the Windows95 operating system. The second platform is a laptop computer, running an Intel Pentium III 850MHz SpeedStep processor with 256Mbyte SDRAM clocked at

100 MHz and with a 256Kbyte level-2 integrated SRAM cache running at the processor frequency. This machine uses the Windows2000 operating system.

6.4.3 Comparison

A set of different sized data files is created, where each file is parsed to generate the tree data structure of the test design. The files are created to generate balanced tree depths of 5, 10, 15 and 20 when parsed.

The generated expression consists of a balanced tree of addition operators with the number 3 at every leaf of the tree. Each tree node is used to store either the leaf number or an operator and two references to child nodes.

Tree depth	5	10	15	20
Character count	125	4,093	131,069	4,194,301
Object count	63	2,047	65,535	2,097,151
Iterative memory (words)	441	14,329	458,745	14,680,057
Recursive memory (words)	252	8,188	262,140	8,388,604
...(3+3)... result	96	3,072	98,304	3,145,728

Table 6.5 Data set statistics

The measurement of time in the hardware version is made by direct simulation. This produces exact results each time. Reading a fine-grained timer before and after each design phase performs the software time measurement. Repeating the experiment many times and calculating the average time increases the accuracy for the software implementations.

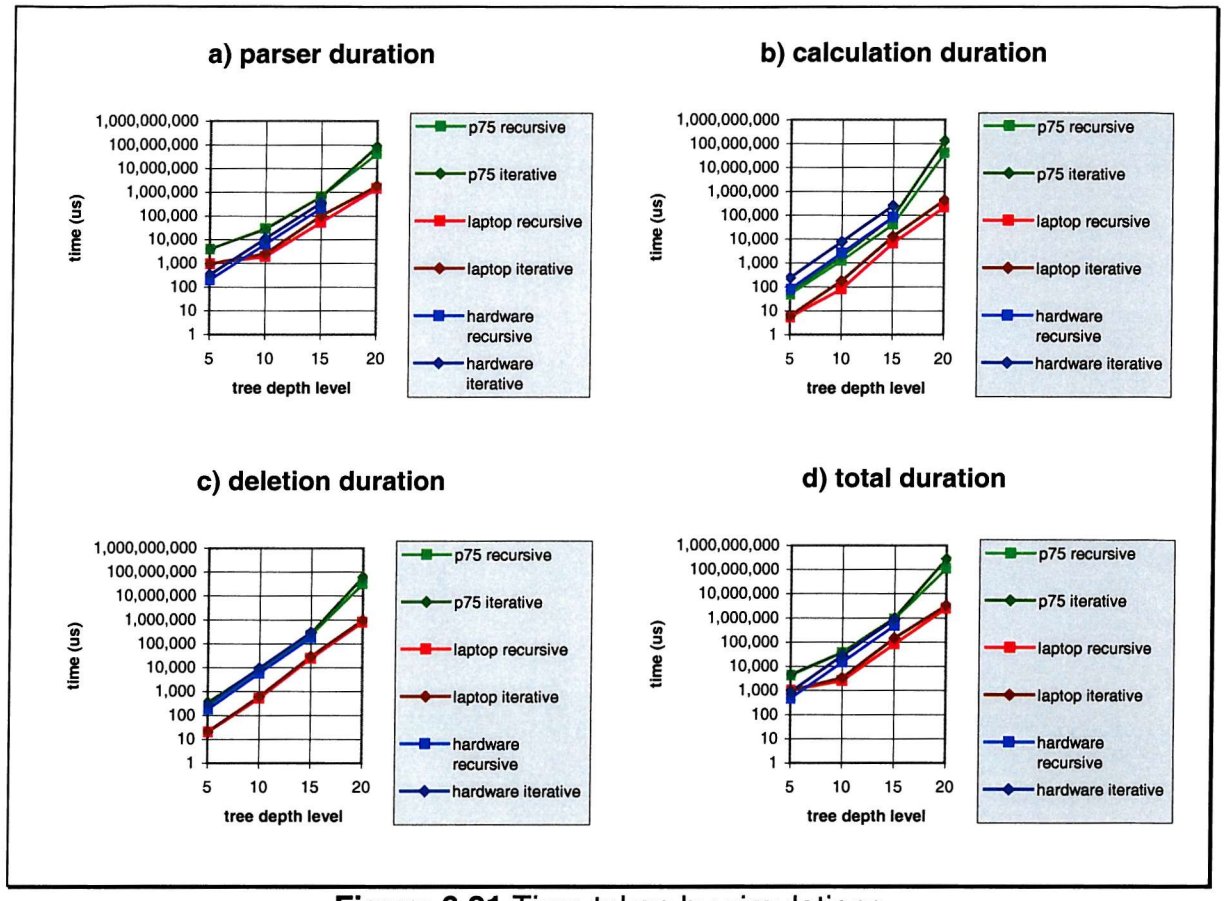


Figure 6.21 Time taken by simulations

The results shown in Figure 6.21 are the measured times for each design compared against each other for the three different phases (Figure 6.21a to Figure 6.21c) with the total duration shown in Figure 6.21d. The time is shown on a logarithmic scale due to the logarithmic complexity of the problem for the different tree depths created. Each extra level added to the tree doubles the problem size, memory requirements and the time taken for tree traversal. The hardware implementations are only simulated up to a tree depth level of 15 due to memory size restrictions, while the software is also measured for a tree depth level of 20.

The results show a straight-line trend in every implementation except for the software parsing of smaller designs and of the calculation phase of the largest tree depth level on the P75 platform. The deviation for the small software designs can be attributed to the increased proportional overhead of the file handling routines that are included as part of the simulation times. The P75 platforms performance suffers with the largest design, as the memory requirements of the problem exceed the available memory space available from the system DRAM, with the system resorting to memory page swapping with the hard disk. The changes in proportions between the different phases of the design are shown

graphically in Figure 6.22, which shows the normalised phase time for the different tree depth levels for each implementation of the system (Figure 6.22a to Figure 6.22f).

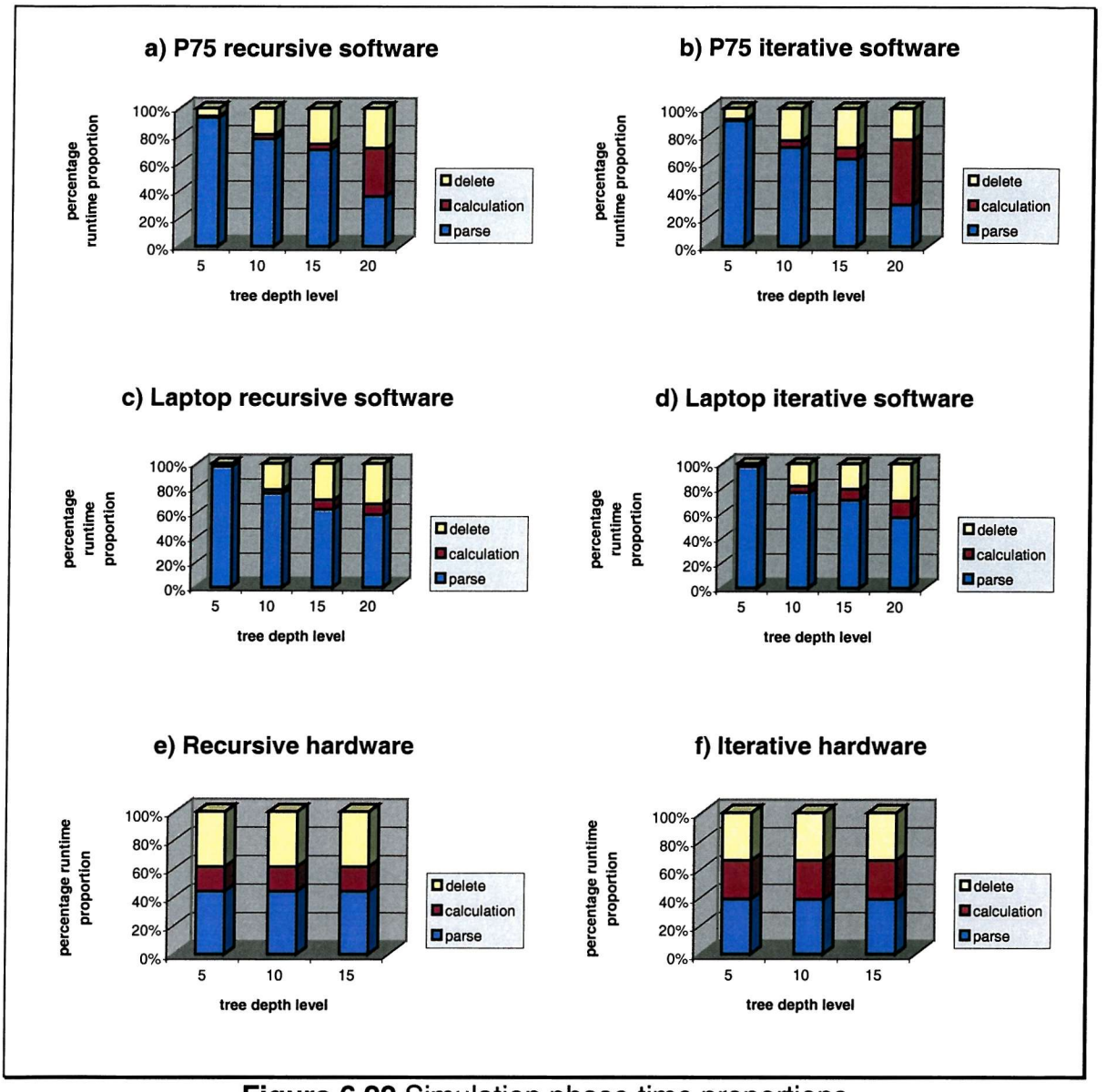


Figure 6.22 Simulation phase time proportions

Figure 6.22 shows that the hardware implementation is unaffected by the depth level of the tree. This means that the allocation and deallocation methods take a proportional amount of time for the three phases for each input file size. As the number of memory operations in each phase procedure is fixed, this shows that the hardware allocator takes a linear amount of time to allocate an object. The proportional results shown for the software implementations both show that smaller designs spend more time parsing the input file. This is due to the file handling routines being measured as part of the parser timing statistics, whereas the hardware design receives its data through a communications port, which has a linear simulated communication time.

The measured times are also affected by the simulated clock speed of the hardware and the CPU and memory clock speeds of the software platforms. However, the major bottleneck in all systems is still the available memory bandwidth, as the test is extremely memory intensive. In this respect, the choice of memory type and access structure (caching system) is extremely critical for all platforms.

	Tree depth level	5	10	15	20
P75 recursive software	Parse	3,994	29,291	647,104	41,138,408
	Calculate	48	1,241	40,468	40,172,254
	Delete	253	7,203	240,763	33,367,057
	Total	4,295	37,735	928,335	114,677,719
P75 iterative software	Parse	4,144	27,360	635,735	85,288,260
	Calculate	64	1,950	82,193	134,946,119
	Delete	362	9,007	287,874	64,087,000
	Total	4,570	38,317	1,005,802	284,321,379
Laptop recursive software	Parse	981	1,913	52,767	1,434,524
	Calculate	5	80	6,699	213,303
	Delete	20	530	24,688	800,837
	Total	1,006	2,523	84,154	2,448,664
Laptop iterative software	Parse	981	2,591	103,131	1,827,548
	Calculate	6	177	12,961	429,444
	Delete	21	610	30,124	969,628
	Total	1,008	3,378	146,216	3,226,620
Recursive hardware	Parse	205	6,622	211,963	
	Calculate	81	2,621	83,972	
	Delete	181	5,858	187,484	
	Total	467	15,101	483,419	
Iterative hardware	Parse	338	10,968	351,180	
	Calculate	240	7,857	251,649	
	Delete	296	9,605	307,737	
	Total	874	28,430	910,566	

Table 6.6 Complete measured time results in μs

Table 6.6 shows the measured times for every simulation in tabular form. Looking at a tree depth level of 15 and the two recursively implemented designs, the total measured time taken for the software version on the laptop is 84,154μs while the simulated hardware version takes 483,419μs. This means that the laptop software runs 5.74 times faster than the hardware. This is to be expected though, as the two systems are not realistically comparable due to the completely different system specification. A better comparison is made with the P75 software platform, which takes 928,335μs. This result shows the

software to be 1.92 times slower than the hardware. Closer inspection of the P75 platform however, shows that most time is spent in the parsing phase, where a combination of memory allocation and file handling is measured. The actual calculation phase times of 40,468μs for software (P75) and 83,972μs for hardware show the software running 2.08 times faster than the hardware, which can be partially attributed to the differences in memory systems between the two platforms, with the SRAM cache system on the software platform and the slow random accesses made to DRAM on the hardware platform. Conversely, notice that the deallocation phase in the same test takes 240,763μs for the P75 and 187,484μs for the hardware, making the hardware 1.28 times faster.

The 25MHz simulated hardware clock speed is set due to the assumed implementation of an FPGA. The equivalent design, optimised for implementation in an ASIC would reach clock frequencies that match and exceed the memory frequency used in the laptop platform. Also, the underlying hardware memory implementation can be tailored to the application, with the use of different underlying memory types, speeds, allocation methods and caching systems. Embedded DRAM [52] could be used for better power performance and wider data interface or an SRAM version of the same heap controller could be designed. This is estimated to increase the available memory bandwidth by a factor of six. All of these memory structure optimisation choices are only possible in the hardware synthesis environment.

The argument for the use of recursion by a system derives from the increased complexity of the behavioural source code and the increased measured time taken for an iterative implementation of both the hardware and software designs. However, the reason for the increased time taken for the iterative versions shown here is that they require more heap memory accesses (random access) due to their implementation method. The hardware version shows this especially, with differences in the types of memory used for the heap and stack dynamic memory, with a recursive version more dependent upon the SRAM based stack memory capable of outperforming a design more dependent upon the slower DRAM based heap memory. If an SRAM based heap manager were implemented, then the 1.9 times speedup gained by using recursion would be reduced to almost the same level. The same argument holds for the software-based implementations, where the procedure stack is partially implemented using an SRAM based circular buffer within the processor.

Chapter 7

Conclusions and further work

The work described in this thesis extended the scope of the MOODS synthesis system to include dynamic memory support, both explicit allocation of user objects with the use of the heap and implicit allocation of local subprogram variables with the use of the stack, allowing procedural recursion.

The source language used for all designs synthesised by MOODS at present is VHDL. This language is compiled into a language neutral ICODE format that is directly processed by MOODS. Most modifications made to the system consisted of additions made to the VHDL compiler, with only procedural recursion requiring changes to the synthesis core. VHDL is a language designed for the description of hardware, but allows for many abstraction levels of design description, including the software-like behavioural level, which now, due to the work carried out, includes the ability to directly describe dynamic objects within the synthesis environment.

The methods used in the modified behavioural synthesis, allowing dynamic memory, borrow heavily from the software domain, with the implementation of both a heap management system and a stack controlling mechanism. The algorithm used in the heap manager is both space and speed efficient, giving a fixed maximum time for allocation and deallocation of objects. It is very simple to interface to a different memory management scheme that is optimised for different area constraints (both physical design area and memory efficiency) and memory allocation performance, as the allocation scheme is not built into the synthesis process.

The enhancements made to MOODS allow the generation of two demonstration systems that both demonstrate the usefulness of dynamic object creation, especially when the source language supports the allocation constructs. The tracker demonstrator shows that it is possible to use the dynamically created objects in a real-time environment, with the use

of concurrent processes and shared data structures. The expression evaluator demonstrator shows the ability and use of procedural recursion, especially when used to control the recursive data structures created dynamically on the heap.

The research carried out within this project enables the MOODS synthesis system to synthesise designs with dynamic memory constructs. There is scope for improvement in the currently implemented system, both with the synthesis process in general and with the methods used in the dynamic memory subsystems. A number of suggested enhancements are described in the rest of this chapter, which could form the basis for future research topics.

7.1 ICODE optimisation

The use of source-level optimisation has been shown to produce better synthesis results [94], with modifications made directly to the source VHDL. However, this technique could not be used at the same time as the modified VHDL compiler, as the source level optimiser does not support the memory constructs used in the VHDL language.

A solution to this problem would be to move the stage at which the source optimisation occurs into the ICODE domain. The benefits of this are twofold. The first benefit would be the ability for source code optimisation to be used in conjunction with dynamic memory constructs, even allowing optimisation of the number of memory accesses required [111] due to common sub-expression sharing. The second benefit would be complete language independence for the optimisation process, with optimisation occurring on the language neutral ICODE.

The benefits of just one optimisation at the ICODE level have been shown in this thesis, that of procedural inlining. A full set of optimisations [112] could drastically reduce the area and delay of some designs that are written for clarity, not efficiency.

7.2 Heap modifications

The modifications discussed here relate to the explicit object creation part of the dynamic memory allocation structure to which each user design is linked.

The first modification relates to the efficiency of behavioural synthesis when dynamic memory is used. At the present time, each translation of a dynamic memory access is a number of ICODE instructions that interface with the underlying heap manager. This leads to a lot of replication of sequences of ICODE instructions, each performing exactly the same operation. Every instance of the access functions is optimised separately during synthesis, and this slows the optimisation.

Expanded modules [3] are designed with optimisation efficiency in mind, where a sequence of instructions is represented by a single instruction. The use of expanded modules to describe the memory access procedures would allow a speedup in optimisation time, with the pre-optimised interface sequence being expanded from the single instruction reference in the last stages of optimisation.

The creation of a number of expanded module interface operations would also allow the migration of the concurrent heap interface multiplexor process into the MOODS core, which frees all future compilers from needing to generate this structure.

The current implementation of the heap manager subsystem uses a single allocation method, with direct access to the underlying DRAM, using completely random access (negating Fast-Page-Mode use). The second modification could be to implement a number of allocation algorithms within a set of heap management subsystems, where the choice of which allocation method to use could be explicitly selected by the user, or left to an automated choice, dependent on user constraints and/or source analysis. The automatic linkage of this subsystem would also be preferable.

The underlying storage mechanism used by the heap manager could be implemented within a number of technologies, including SRAM, faster DRAM of various types (FPM, EDO, SDRAM), or a mix of technologies with the faster memories being used for speed sensitive areas of allocation. This allows even more choice in the number of heap management systems that can be used.

If the hardware destination of a design is to be an FPGA, then the use of more specialised FPGA architectures that contain a number of SRAM memory blocks could be utilised. These would allow the stack and heap to be implemented internally either partially or

fully, dependent on memory requirements. The XILINX 'Virtex' series of FPGA, containing 'SelectRAM' is an example of such a specialised architecture.

The completely random access of the storage space of the underlying dynamic objects negates the use of advanced memory data streaming, such as fast-page-mode access or burst-mode access of particular types of DRAM. This slows the available memory access speed to the random access speed. There are two possible solutions to this problem.

The implementation of a cache controller between the heap management algorithm and the underlying memory controller forms another memory interface level. Such a system could be designed to use the faster memory accesses of the available large-scale memory, while providing single-cycle access to data that is referenced by the cache, from the use of SRAM based storage. The use of a caching system could be another parameter used in the selection of a heap management system.

The second solution could be to provide an enhanced interface to include memory-type specific accesses in the generated ICODE. This could require knowledge of which heap management subsystem is to be used before compilation, as the enhanced interface may not be available in all management systems. Such accesses could be formed from analysis of the source code, where a streamed memory access contained by a loop could map onto a fast-page-mode DRAM access or burst-mode access [113]. This requires investigation to determine the workability of the solution over the cache solution.

7.3 Stack modifications

The modifications discussed here relate to the implicit object creation part of the dynamic memory allocation structure that is created for each design utilising procedural recursion.

At present, a design using procedural recursion is limited to a single process that uses recursive procedures. A design may contain more than one process, with only one able to use recursion. This is due to the current implementation of the call stack, as a single contiguous memory block per design.

The lack of analysis for a list of all possible concurrent calls to all the statically generated procedures also has effects upon the generation of recursive procedures. If any procedure

is called from more than one process, then this procedure either has to be replicated (if the procedure does not communicate directly with any internal signals or design ports), or blocked for concurrent access (with the automatic generation of mutex wrapper function blocks around the procedure). At present, all concurrent access to a generated subprogram is reliant upon explicit user-defined access control within the design source code, not upon the synthesis system. The generated structures for concurrent access of general procedures require specification before the additional structures required for concurrent access of recursive procedures can be specified.

Dependent on whether a recursive procedure is replicated or access-controlled during generation allows for different methods for the allocation of procedure-local frame data. Replication can enable access to completely concurrent stacks using different access ports, whereas a single access-controlled procedure can use only a single port into a stack system. In this case, the underlying stack mechanism of the contiguous block of memory with a single stack head pointer cannot be used. A more complex system possibly based upon the heap allocation methods and with an automatic generation of a linked list structure between stack frames could be used.

There is only one implementation of stack-frame handling in the current system, where the data is stored in a single memory array with a single access port. This forces every access of the stack to happen in separate control states, inducing a sequential delay to designs. An alternative implementation could be used for designs optimised for delay, where each variable requiring stack storage could have an associated local stack. This would allow concurrent access of each stack variable. Sharing of stack arrays for local variables between mutually exclusive recursive procedures could also be allowed. Some static analysis of the likely storage space requirements would be required, in order to balance the amount of data stored in each stack array.

For larger stack requirements, it may be useful to create an external stack interface, like the heap manager interface. This would allow interfacing to a number of memory technologies, including SRAM and DRAM again.

An alternative to the separate external stack memory and heap memory could be to combine the two in some manner, where a limited built-in stack is provided in the same manner as before, with the heap memory accessed whenever the stack overflows or

underflows. The heap could be used to allocate enough space for the stack frame used in a recursive procedure when the stack overflows. In this way, the stack is used as a circular window into a potentially larger memory base, with the slower heap only being accessed during excessive recursion. This also removes the stack overflow exception.

7.4 Exception handling

There are currently two hardware exceptions that are not directly accessible from the VHDL language, which can break a design under certain circumstances. These exceptions are effectively memory allocation problems due to the limited underlying memory space available to the user's design. The exceptions occur when the heap manager is incapable of allocating the required user object or when the stack overflows due to excessive procedural recursion. The stack overflow problem could be reduced to the heap allocation problem, with the last given stack modification of the combined windowing stack.

The main disadvantage encountered is the lack of accessibility of the exception as part of the VHDL language. The only way that the exception can currently be handled is if an allocation returns a null reference. This means that the user's design must test every allocation to see if it failed, and act accordingly in some controlled manner. This can lead to large, ugly and unreadable source code. A better mechanism, which removes the need for user testing, may be to build the exception handling into the synthesised design, where some form of registered status output could give an indication of system health. This indicator could allow for the switching of backup systems in place of the currently active system.

Appendix A

Collateral projects

During this research, a number of smaller projects were undertaken to gain familiarity with the MOODS synthesis system. These projects are designed to enhance the system in general and supplement the demonstrators described in Appendix C. Other users of MOODS for further development and demonstrations have subsequently used some of these projects.

The first VHDL based project was to implement a VGA controller library. This subsystem has been the most widely used bolt-on component and has been implemented in three major versions using two different technology bases. The system is used as a viewable output method for designs requiring a visual user interface, where a standard VGA monitor screen is used to display the signals directly generated from the VGA controller. The interface to the concurrent controller is procedure based with direct calls made from the user's design used to draw objects such as rectangles, text and straight lines from arbitrary points.

The second VHDL based project was to implement an interface to a standard PS2-based 101-key ASCII keyboard. This subsystem will form the input to various designs requiring a number of keyboard switches. This system is the second most widely used with most demonstrators comprising of a monitor, keyboard and the created driving hardware.

The final VHDL based project was to implement an interface to a standard serial port. This project was designed specifically for the tracker demonstrator described in Chapter 6 and Appendix C. However, the general-purpose nature of the subsystem lent itself to being created as a completely self-contained project. Only the input half of the serial port interface is implemented, with the output half being an easy addition if ever required. This interface is implemented, as the first two, with a concurrent controller driven by calls to a set of sequential interface procedures.

The next project, the wave viewer is software based. It was designed to gain some more familiarity with the MFC class libraries used to generate windows programs. A wave viewer generally forms one of the visualisation methods of a digital simulator. A proprietary input file format was designed for use with this program and the viewer has been used with a neuron simulator and is currently forming the output visualisation method for a digital simulator based on the internal MOODS data structures.

Older versions of MOODS generate the final RTL VHDL output directly from the core. With the introduction of multiple concurrent components, in the form of the various library components and more importantly, the heap management component, a method to link these automatically into the final structural output becomes necessary. The program; 'DDFLink'; is used as a final stage data structure translation tool, and is currently being expanded in order to generate EDIF output directly from the data structures represented by the 'DDF' file format described in appendix D.

The first six months of the PhD were used to try to find a niche within the 3D graphics research area that could be the base for the entire research project. However, this time proved the beginning of the period of major commercial growth in this area, which effectively halted the research efforts in the direction that was beginning to be taken. Within this period, a small 3D graphics engine was written in software that was loosely based upon the PHIGS hierarchical data structures used to generate 3-dimentional objects. This software used a 3D accelerator card created by the company '3DFX'. This program is described in the final section of this appendix.

A.1 VGA controller library

The VGA controller is designed as a component to which the user's design interfaces via the VGA Interface package that contains various set-up and drawing procedures. This section explains the various aspects of all the versions of the VGA controllers and their respective interfaces. The VGA controller library forms the cornerstone of many of the demonstrators built by the research group. Various versions of this library exist, with optimisations made with respect to the target hardware and to the type of design that utilised the library.

A.1.1 Overview

The VGA system is a graphical interface that can be used by calling VHDL procedures. It outputs all the signals necessary to drive a VGA monitor at the standard 640x480 resolution [114]. The dot-clock for this resolution is 25.125MHz, which determines the clock frequency supplied to every version of the VGA controller.

It is targetted at the PCB described in Appendix C. This board uses a XILINX XC4062XL FPGA, has 1M of DRAM frame buffer memory and has a triple 8-bit DAC for driving a monitor. It has since been updated to use a third party board provided by XESS, which uses a XILINX Virtex-800 FPGA, with a different frame buffer stored within SRAM and a RAMDAC instead of a DAC. The use of SRAM allows single cycle read-access, which enables an 8-bit per pixel (256 colour) version to be produced (even with a reduced SRAM data path width), with a direct port to the frame buffer being provided for this instead of the object rendering capabilities found in the 4-bit per pixel (16 colour) version initially designed.

A.1.1.1 Controller

This component forms the concurrently active low-level driving system that generates the SYNC signals and colour output that drives the VGA monitor screen directly. It makes use of a frame buffer memory into which stored images are rendered. These images are serialised into a raster-scan, passed through a palette-lookup system and drive the monitor with the generated signals. The memory effectively introduces a large time buffer in which to work asynchronously with the output signal generation.

The images are produced by various memory modification routines that, in the case of the 16-colour version, are capable of drawing bitmapped text characters, rectangles and straight lines anywhere within the memory space. The 256-colour version of the controller has two direct memory access ports to the underlying memory and no direct rendering capabilities itself. This design is optimised for producing more colour intensive outputs such as realistic picture visualisation.

A.1.1.2 Interface

This package is used by the user design and handles the communication between the user design and the VGA controller system. It is wise to use only the procedures defined within the package and not to drive the signals directly. Some of these interface procedures are inlined and others are not, depending on tests for what gives the best results.

A.1.1.3 Simulation

A Modelsim simulation library exists for the 4-bit (16 colour) version of the VGA controller. The simulation uses Modelsim's C-interface into the simulation structures that enable various hooks to be utilised. The point of the simulation library is that the user can link to the virtual VGA controller given by the library and it will create a window on the simulation computer that displays exactly what will be displayed after the synthesis and place and route process.

File name	Description
vga_sim_pck.vhd	VHDL component interface - virtual VGA controller interface
vga_sim_bdy.vhd	VHDL component body - empty body required by Modelsim
text2col2.bmp	The text ROM stored as a bitmap
mti_vga1.dll	The executable that generates the window

Table A.7 Files required for VGA controller simulation

Simulation can be either at the behavioural level (source code before synthesis – use '*vga_controller_source*') or at the structural level (VHDL code after synthesis – use '*vga_controller*'). The VGA Controller component used during RTL synthesis is replaced by the simulation component given in '*vga_sim_pck.vhd*'. Note that a simulation of the text ROM is included within the DLL, so the address of the text ROM is supplied to the controller, not the resulting data. The simulation name, position of the text ROM bitmap and the default palette are set-up using VHDL generics passed into the simulation component. The position of the DLL executable is set up within the architecture itself and should remain fixed.

At present, no simulation library exists for the 256-colour version of the controller. The only indirect method for visualising what will be produced is to use the actual controller source itself. Accepting the semaphore accesses may be enough for visual inspection of whether the user design is working or not.

A.1.1.4 Source VHDL structure

The three versions of controller are each contained in a single file each. The controllers interface to one of two constants packages and one of two palette setup packages. The interface package used by the user also has two versions. The two versions are necessary, as each provides a completely different interface to the underlying controller.

The VGA Controller design is created from the VHDL files shown in Table A.8. Note that only the 4-bit version of the controller is available for the university board.

	4-bit (16 colours)	8-bit (256 colours)
University Board	vga_const.vhd vga_palette.vhd vga_controller.vhd	N/A
XESS Board	vga_const.vhd vga_palette.vhd vga_controller_xess.vhd	vga_const_8bit.vhd vga_palette_8bit.vhd vga_controller_xess_8bit.vhd

Table A.8 Files required for the VGA controller

The user must interface to the VGA controller using the files shown in Table A.9. Note that the interface is not dependent on which PCB is used, only the number of bits used to store a single pixel.

	4-bit (16 colours)	8-bit (256 colours)
University Board	vga_const.vhd vga_interface.vhd	N/A
XESS Board	vga_const.vhd vga_interface.vhd	vga_const_8bit.vhd vga_palette_8bit.vhd

Table A.9 Files required for the user to interface to the VGA controller

A.1.1.5 Design structure and style

The VGA controller system is designed as a completely independent component that is linked with any user design after MOODS optimisation. At present, the linking is performed manually (by the user) with an RTL VHDL wrapper file. To use the VGA system, the user must include interface signals in the design port list and call the interface procedures as defined within the interface package, passing the relevant ports into the procedures. All one-way communication is controlled by a number of semaphore-acknowledge signal pairs, which are toggled when data is transferred. The master of

communication drives the semaphore (indicating that data is available for transfer or that data is required from the slave). The slave to the communication reacts to the toggled master semaphore by toggling the acknowledge signal, either reading the data from the master or returning data to the master (or both). The semaphores are controlled by the interface procedures.

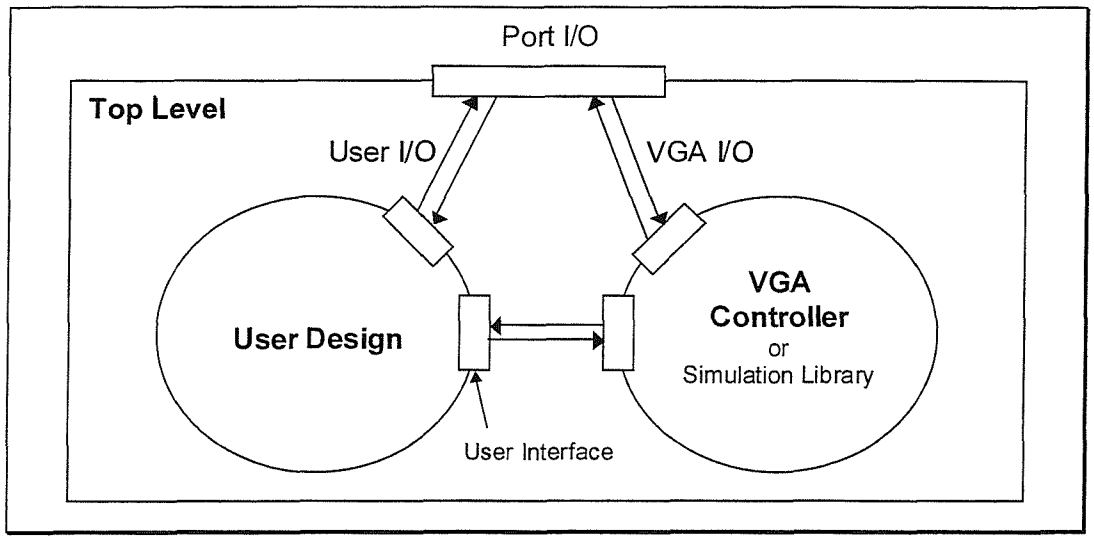


Figure A.1 VHDL Wrapper file structure

The wrapper file is required to link together any user designs with any pre-compiled library components such as the VGA controller system. The user must supply it. Each version of the VGA controller has a different interface and needs different buffering and tri-stating actions performed to some of its ports.

The clock supplied to the VGA controller system must be 25MHz. Any other component can be run at their designated speed, including the component that interfaces to the VGA controller. If two different clocks are used, then extra buffers will be required between the subsystems. If asynchronous clocks are supplied, then all signals will require double buffering in both directions. If divided clocks are generated from the 25MHz base clock, then single buffers are required for all outputs from the fast system into the inputs of the slow system.

Pin constraints are used to define which pins of the FPGA to use for the I/O signals. They are contained in a UCF file. Two versions are available, one for each board. The constraints file maps onto the names given in the main entity port list, so all VGA specific ports keep the same name.

The VGA system on the XESS board conflicts with some of the Ethernet interface chip connections. Directly drive the Ethernet output enable line with 1' to disable the Ethernet chip.

A.1.2 Original 16-colour interface

The low-level interface that is provided by the controller gives the ability to draw very fast horizontal lines in the current drawing colour. The only alternative to this is to draw the same horizontal line masked with a pattern derived from a ROM. This second alternative allows bitmapped text characters to be produced.

This simple interface is not directly accessible to the user. Instead, the user performs drawing actions that utilise multiple communications with the underlying low-level interface to draw more complex objects. These high-level drawing algorithms are contained in the interface procedures provided.

Most interfacing procedures are simple value setting procedures that check whether it is safe to set the value by the use of the drawing semaphore and acknowledge that are part of the interface required in the user's design. The procedures will block until the VGA controller allows them to continue. All interfacing uses the '*render_sem / render_ack*' semaphore-acknowledge pair to handle the transfer of data, except any palette modification that uses a dedicated port using a similar pair of signals '*palette_modify_sem / palette_modify_ack*'. This means that any palette modification can occur in parallel to any drawing procedures.

All drawing procedures should be called from only one user process. Multiple process use requires the user to serialize the calls to the interface procedures using some form of user-generated semaphores. Single process use of the interface is recommended for the various interface calls for simplicity and safety.

The system is set up as a registered state machine and the set-up parameters are passed separately from the drawing procedures. For example, to draw a rectangle in XOR mode using colour 15, first call the **vga_setmode** procedure, then set the drawing colour by calling **vga_setforecolour** and finally draw the rectangle by calling the **vga_drawrect**

procedure. The set-up information is persistent, so to draw another rectangle in the same colour in the same mode only requires another call to **vga_drawrect**.

Most interface procedures are inlined into the user code using the method of placing a call to the '*inline*' procedure in the body of the procedure being inlined. This means that most calls to the VGA interface will be hidden in the final implementation. An exception to this is the arbitrary line drawing algorithm, due to its complexity. If it were inlined, the compilation time for multiple calls to the inlined line drawing procedure would be prohibitive and generate more hardware than necessary. Definitions of the interface procedures are provided in the following sections, along with descriptions of their actions. These procedures are defined in the interface package.

A.1.2.1 Interface types

A number of VHDL types are defined for interfacing with the VGA controller. These types are defined within the constants package and are shown below:

```
subtype vga_colour_type is bit_vector(COLOUR_BITS-1 downto 0);
subtype vga_red_type is bit_vector(RED_BITS-1 downto 0);
subtype vga_green_type is bit_vector(GREEN_BITS-1 downto 0);
subtype vga_blue_type is bit_vector(BLUE_BITS-1 downto 0);
subtype vga_rgb_type is bit_vector(RGB_BITS-1 downto 0);
subtype vga_mode_type is bit_vector(MODE_BITS-1 downto 0);
subtype vga_page_type is bit_vector(PAGE_BITS-1 downto 0);
subtype vga_xpos_type is bit_vector(XBITS-1 downto 0);
subtype vga_ypos_type is bit_vector(YBITS-1 downto 0);
subtype ascii_type is bit_vector(ASCII_BITS-1 downto 0);
subtype vga_textsize_type is bit_vector(TEXTSIZE_BITS-1 downto 0);
subtype vga_text_xpos_type is bit_vector(TEXTPOS_XBITS-1 downto 0);
subtype vga_text_ypos_type is bit_vector(TEXTPOS_YBITS-1 downto 0);
subtype vga_text_inc_type is bit_vector(TEXTPOS_INCBITS-1 downto 0);
```

A.1.2.2 System setup

The user's design is the master of communication with the slave VGA design. The communication semaphores used within the interface procedures require initialisation at startup. An initialisation procedure is provided, which is used to set up both interface semaphores '*render_sem*' and '*palette_modify_sem*' (called once per semaphore signal):

```
procedure vga_initialise(signal semaphore : out bit);
```

A.1.2.3 Drawing attributes

The VGA controller system has registered settings for the currently rastered page, the page to render into, drawing mode, colour and background colour. The interface procedure declarations for modification of these settings are listed below:

```
procedure vga_setdefaults (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal render_page : out vga_page_type;
  signal raster_page : out vga_page_type;
  signal render_mode : out vga_mode_type;
  signal render_colour : out vga_colour_type;
  signal render_backcolour : out vga_colour_type
);
```

Description: Set pages to **PAGE0**, mode to **MODE_DD_BOTH** and colours to **COL_0**.

```
procedure vga_setrasterpage (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal raster_page : out vga_page_type;

  -- user input
  page : in vga_page_type
);
```

Description: Set the viewed page to the '*page*' input. Page values can be **PAGE0**, **PAGE1**, **PAGE2** or **PAGE3**.

```
procedure vga_setrenderpage (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal render_page : out vga_page_type;

  -- user input
```

```
  page : in vga_page_type
);
```

Description: Set the page to draw to from the '*page*' input. Page values can be **PAGE0**, **PAGE1**, **PAGE2** or **PAGE3**.

```
procedure vga_setmode (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal render_mode : out vga_mode_type;

  -- user input
  mode : in vga_mode_type
);
```

Description: Set the drawing mode (XOR/direct, foreground only/both foreground and background). Mode values can be **MODE_DD_FORE**, **MODE_DD_BOTH**, **MODE_XOR_FORE** or **MODE_XOR_BOTH**.

```
procedure vga_setforecolour (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal render_colour : out vga_colour_type;

  -- user inputs
  colour : in vga_colour_type -- which colour to adjust to
);
```

Description: Set the foreground colour (0 to 15). The colour is a 4-bit value and 16 constants have been defined for each value, ranging from **COL_0** to **COL_15**.

```
procedure vga_setbackcolour (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal render_backcolour : out vga_colour_type;
```

```
  -- user inputs
  colour : in vga_colour_type -- which colour to adjust to
);
```

Description: Set the background colour (0 to 15). This colour is only used within the character drawing procedure. The colour is a 4-bit value and 16 constants have been defined for each value, ranging from **COL_0** to **COL_15**.

A.1.2.4 Palette modification

Dynamic palette modification capabilities are provided, which can enhance the static palette values used within the VGA controller. Two interface procedures are provided:

```
procedure vga_setpalette_rgb (
  -- ports
  signal palette_modify_sem : inout bit;
  signal palette_modify_ack : in bit;
  signal palette_modify_addr : out vga_colour_type;
  signal palette_modify_val : out vga_rgb_type;

  -- user input
  colour : in vga_colour_type; -- which colour to adjust
  rgb : in vga_rgb_type        -- the new rgb value
);
```

Description: Set the palette colour (0 to 15) with RGB (0 to 4096). The '*colour*' input says which colour index to adjust the palette of. The '*rgb*' input gives the 12-bit concatenated RED & GREEN & BLUE value.

```
procedure vga_setpalette (
  -- ports
  signal palette_modify_sem : inout bit;
  signal palette_modify_ack : in bit;
  signal palette_modify_addr : out vga_colour_type;
  signal palette_modify_val : out vga_rgb_type;

  -- user input
  colour : in vga_colour_type; -- which colour to adjust
  red : in vga_red_type;       -- the red component of the palette
  green : in vga_green_type;   -- the green component of the palette
```

```
  blue : in vga_blue_type      -- the blue component of the palette
);
```

Description: Set the palette colour (0 to 15) with separate Red, Green and Blue values. The '*colour*' input says which colour index to adjust the palette of. The '*red*', '*green*' and '*blue*' values give the palette shade in the 4-bit triple.

A.1.2.5 Drawing horizontal lines

The VGA controller has the horizontal line as the primitive from which all other drawing procedures generate their screen objects. The character drawing procedure uses a slightly different version of the same low-level horizontal line drawing routine in the controller, by allowing masking of each horizontal line via the text ROM. The low-level horizontal line algorithm is the most efficient and simplest method of drawing; hence the direct interface to it as a procedure call within the interface. The low level renderer does not care which way round the two X-values for the left and right position of the horizontal line are given. It will draw between and including the two X positions at the given Y position. An interface procedure is provided:

```
procedure vga_drawhorzline (
  -- ports
  signal render_sem : inout bit;
  signal render_ack : in bit;
  signal render_type : out bit;
  signal render_xone : out vga_xpos_type;
  signal render_xtwo : out vga_xpos_type;
  signal render_ypos : out vga_ypos_type;

  -- user input
  x1 : in vga_xpos_type;
  x2 : in vga_xpos_type;
  y : in vga_ypos_type
);
```

Description: Draw a horizontal line (fast) between inputs '*x1*' and '*x2*' at y-position '*y*'.

A.1.2.6 Drawing filled rectangles

The rectangle-drawing algorithm is simply a loop between the two Y positions given. A test is made before the loop to determine which Y position is larger, and swapped if necessary. For each Y position, a horizontal line is drawn between the two given X positions. This functionality is provided within the interface procedure:

```
procedure vga_drawrect (
  -- ports
  signal render_sem : inout bit;
  signal render_ack : in bit;
  signal render_type : out bit;
  signal render_xone : out vga_xpos_type;
  signal render_xtwo : out vga_xpos_type;
  signal render_ypos : out vga_ypos_type;

  -- user input
  x1 : in vga_xpos_type;
  y1 : in vga_ypos_type;
  x2 : in vga_xpos_type;
  y2 : in vga_ypos_type
);
```

Description: Draw a rectangle between (and including) two corner points given by the inputs ('*x1*', '*y1*') and ('*x2*', '*y2*').

A.1.2.7 Drawing arbitrary lines

Due to speed and size constraints, an integer incremental line drawing algorithm is used to draw a single-pixel thick line from two specified end points. Any two points can be given with no restrictions on the orientation of the line or relative positions of the end points.

Bresenham first demonstrated the underlying algorithm [116], which is limited to lines from 0 to 45 degrees above the x-axis. Modifications were required to allow the algorithm to work with arbitrary end points and orientation. These modifications include optimisations to use the low-level draw horizontal line algorithm efficiently.

Bresenham's algorithm works by keeping a cumulative integer error value by adding and subtracting one of two terms dependent on whether the present value of the cumulative value is larger than zero. The algorithm and example line is shown in Figure A.2.

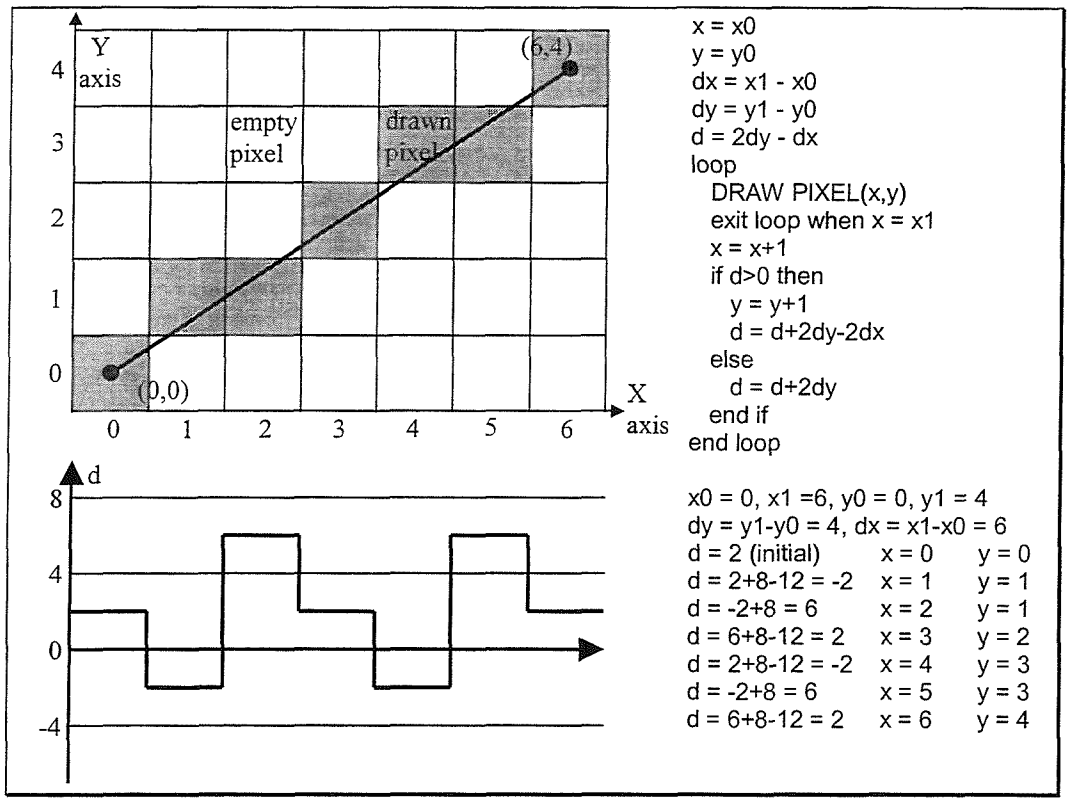


Figure A.2 Bresenhams line drawing algorithm

The algorithm shows that the x-position is incremented on every iteration of the loop and the y-position is only incremented when the decision value ***d*** is larger than zero. The algorithm also shows that when the decision variable ***d*** is smaller than or equal to zero, the next value of ***d*** is an increment of ***2*dy***, which is always positive or zero with slope restrictions. The alternative increment of ***2*dy - 2*dx*** is always negative or zero as ***dy*** is always smaller than or equal to ***dx*** for slopes limited from 0 to 45 degrees.

To allow lines of any orientation to be drawn requires two modifications. The first modification takes the two end-points and calculates the positive versions of ***dy*** and ***dx***, and makes a note of which (if any) were negative. The starting point is set as the point with the smaller x-value and the ending point is set as the point with the larger x-value. The algorithm then increments the x-value as normal until it reaches the ending x-value. The decision-making is exactly the same. When an increment of the y-value is required, the polarity of both ***dx*** and ***dy*** are tested to see whether they are the same. If both are

positive or both are negative, then the y-value is incremented as usual, but if only one is negative, then the y-value is decremented. This allows the shaded range shown in Figure A.3 to be rendered correctly.

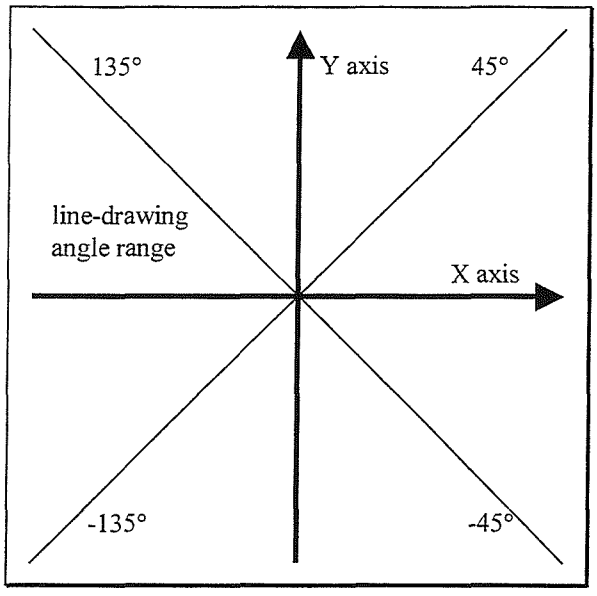


Figure A.3 Rendering angles for partial line drawing implementations

This is half of the total line-drawing angle range that is possible. The ranges from 45 to 135 degrees and -45 to -135 degrees also need supporting. This is done with the second modification. Once the positive values of ***dx*** and ***dy*** have been calculated, a simple test of which is larger will determine whether the range is from the shaded range (***dx*** > ***dy***) or the non-shaded range (***dx*** ≤ ***dy***). If ***dx*** > ***dy*** then the present algorithm will work. However, the alternative range requires the second modification. The modification is to have a second copy of the line drawing algorithm that works in the alternative range. This is simply a case of rotating the algorithm by 90 degrees by exchanging x-values for y-values, so that the y-value is always incremented and the x-value is incremented (or decremented) dependent on the decision variable.

The final modification only applies to the original algorithm that increments the x-value. This is for time-efficiency reasons and involves replacing the draw-pixel call with a draw-horizontal-line call that is only called when the y-value changes.

All of the functionality described above is provided in the interface procedure. The procedure is not inlined into the user's code due to the size of the replicated structures:

```
procedure vga_drawline (
  -- ports
  signal render_sem : inout bit;
  signal render_ack : in bit;
  signal render_type : out bit;
  signal render_xone : out vga_xpos_type;
  signal render_xtwo : out vga_xpos_type;
  signal render_ypos : out vga_ypos_type;

  -- user input
  x1 : in vga_xpos_type;
  y1 : in vga_ypos_type;
  x2 : in vga_xpos_type;
  y2 : in vga_ypos_type
);
```

Description: Draw a line between (and including) two arbitrary points given by the inputs ('*x1*', '*y1*') and ('*x2*', '*y2*').

A.1.2.8 Drawing characters

As the text character x-position is forced to be in alignment with the 8-pixel word boundaries, due to the limited pixel masking available for the text mask, the y-position is also forced into 8-pixel alignment from the top of the screen. This means that the character will snap to the 8-pixel aligned top-left position. In fact, a different co-ordinate system is used that holds positions 80 x 60 text positions instead of 640 x 480 pixel positions. Only one position is supplied to the character drawing algorithm.

The character x-position and x-size are given to the low-level text-line algorithm (within the VGA controller) first, as these do not change for the rest of the character algorithm. Then the code enters a loop that counts up to eight, for each line of the 8 by 8 character. The first action is to output the address of the present line of the particular character being drawn to the text ROM. Then the code enters another loop that is nested within the first, which loops depending on the given y-size. The y-size can be 1 of 4 values that allow a character to be drawn with a height of 8, 16, 24 or 32 pixels. For each iteration of the inner loop, one horizontal text-line is drawn at the present y-position and the y-position iterator is incremented. The character is drawn once the final rendered line of the eighth character line is reached.

Each character line uses the modified horizontal line drawing algorithm within the VGA controller. The value given from the text ROM is used to mask the horizontal line in one of four text X-sizes, with widths possible in the same range as heights (8, 16, 24 or 32 pixels).

The text mask is generated from a bitmap shown in Figure A.4. A small program has been produced that will convert such a bitmap used in the Modelsim simulation into the required format for an internal VHDL lookup table, or as an EPROM bitmap for use in an external 2K ROM.

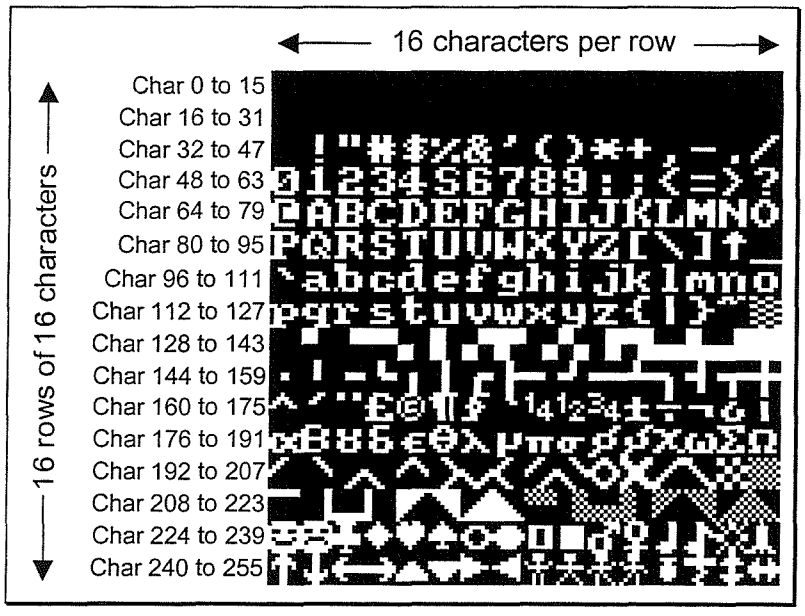


Figure A.4 ASCII character map image in a 2K ROM

The declaration of the interface procedure to draw characters is shown below:

```
procedure vga_drawchar (
  -- ports
  signal render_sem : inout bit;
  signal render_ack : in bit;
  signal render_type : out bit;
  signal render_xone : out vga_xpos_type;
  signal render_xtwo : out vga_xpos_type;
  signal render_ypos : out vga_ypos_type;
  signal render_text_size : out vga_textsize_type;
  signal render_textrom_addr : out vga_textrom_address_type;
```

```
  -- user input
  x : in vga_text_xpos_type;
  y : in vga_text_ypos_type;
  xsize : in vga_textsize_type;  -- x size
  ysize : in vga_textsize_type;  -- y size
  ascii : in ascii_type      -- ASCII code for the character
);
```

Description: Draw an ASCII text character at an 8-pixel aligned position given by the inputs '*x*' and '*y*'. The character is drawn with X and Y sizes given by inputs '*xsize*' and '*ysize*'. Four constants are provided to describe the size as **TEXTSIZE8**, **TEXTSIZE16**, **TEXTSIZE24** or **TEXTSIZE32**. The four enumerated values can be stored in two bits. The '*ascii*' input requires eight bits in order to reference the 256 available characters, with various constants defined using the ASCII standard character map. The text ROM holds the character masks used within this procedure.

A.1.2.9 Vertical blanking

The VGA controller generates a signal that defines when the raster-scan is within the vertical blanking period. It forms a 60Hz signal with 45/480 mark/space ratio (45 blanked lines to 480 drawn lines). This signal is passed into the two vertical blanking procedures to determine when to synchronise the drawing of items to the screen:

```
procedure vga_wait_for_vertical_blanking (
  signal vert_blank : in bit );
```

Description: Wait for the vertical blanking period to begin.

```
function vga_vertical_blanking (
  signal vert_blank : in bit) return boolean;
```

Description: Return whether the raster-scan is currently in the vertical blanking period (true) or rastering the memory contents (false).

A.1.2.10 Using the interface

Any user design can drive a VGA screen. The requirements for doing this are as follows:

- include references to the constants and interface packages.

- Add a list of port signals to the user design as given in comments at the top of the interface package.
- Call the initialise procedure (as defined in the interface package) passing both the rendering semaphore '*render_sem*' and the palette modification semaphore '*palette_modify_sem*' as parameters.
- Call any other drawing procedure as required, passing references to the relevant ports defined at the top of the user code and any other parameters that the drawing procedures require.

This is illustrated in the following example:

```
1   use work.icode_ops.all;        -- icode operations
2   use work.vga_const.all;        -- VGA constants
3   use work.vga_interface.all;    -- VGA interface
4   entity vga_test is
5     port (
6       -- VGA controller interface ports - must be included by user
7       render_sem : inout bit;
8       render_ack : in bit;
9       render_type : out bit;
10      render_xone : out vga_xpos_type;
11      render_xtwo : out vga_xpos_type;
12      render_ypos : out vga_ypos_type;
13      render_page : out vga_page_type;
14      raster_page : out vga_page_type;
15      render_mode : out vga_mode_type;
16      render_colour : out vga_colour_type;
17      render_backcolour : out vga_colour_type;
18      render_text_size : out vga_textsize_type;
19      render_textrom_addr : out vga_textrom_address_type;
20      palette_modify_sem : inout bit;
21      palette_modify_ack : in bit;
22      palette_modify_addr : out vga_colour_type;
23      palette_modify_val : out vga_rgb_type;
24      vert_blank : in bit
25    );
26  end;
27
28  architecture behave of vga_test is
29  begin
30    control_process : process
31    begin
32      -- initialize the interface to the vga system
33      vga_initialize(render_sem);
34      vga_initialize(palette_modify_sem);
35
36      vga_setrasterpage(render_sem, render_ack, raster_page, PAGE0);
37      vga_setrenderpage(render_sem, render_ack, render_page, PAGE0);
38      vga_setmode(render_sem, render_ack, render_mode, MODE_DD_BOTH);
39      vga_setforecolour(render_sem, render_ack, render_colour, COL_15);
40      vga_setbackcolour(render_sem, render_ack, render_backcolour, COL_0);
41
42      -- draw the background using the set colour value
43      vga_drawrect(render_sem, render_ack, render_type, render_xone, render_xtwo,
44                   render_ypos, convert_int2bv(0,10), convert_int2bv(0,9),
45                   convert_int2bv(639,10), convert_int2bv(479,9));
46
47      -- draw all characters
48      for ch in 0 to 255 loop
```

```
49          vga_drawchar(render_sem, render_ack, render_type, render_xone,
50                       render_xtwo, render_ypos, render_text_size,
51                       render_textrom_addr, convert_int2bv(10,7),
52                       convert_int2bv(6,6), TEXTSIZE8, TEXTSIZE8,
53                       convert_int2bv(ch, ASCII_BITS));
54        end loop;
55
56        -- runtime palette modification
57        for pal in 0 to 15 loop
58          vga_setpalette(palette_modify_sem, palette_modify_ack,
59                         palette_modify_addr, palette_modify_val,
60                         pal, pal, pal, pal);
61        end loop;
62
63        -- wait forever
64        loop
65          wait for 100 ns;
66        end loop;
67      end process control_process;
68    end behave;
```

The example shows how to set up the drawing area by drawing a rectangle covering the entire visible screen area (line 43). This is required as the underlying memory that contains the raster image powers up with random values contained in each pixel. The example also shows how to draw all the characters, with each character drawn in the same position (line 49). As the drawing mode is set to directly draw both the foreground and background colours (line 38), each character will overwrite any previous character image completely. Finally, the colour palette is modified into a grey-scale (line 58) by looping through all sixteen colours, changing the values for red, green and blue into the same value as the colour index.

A.1.2.11 General tips for use

The initialise procedures must be called for each semaphore port. The set defaults procedure is not required, but can prove useful for setting global start-up parameters.

There are four viewable pages within the frame buffer, the raster page is the one that is being viewed (raster-scanned onto the VGA screen) and the render page is the page to which all drawing procedures write. The pages can be the same if only one page is required. Drawing actions can be hidden if drawn to a non-viewed page, and then viewed by swapping the raster page onto the previously hidden page.

There are sixteen viewable colours. The foreground colour is the one used in every drawing procedure. The background colour is only used within the text drawing procedure, and forms the text character mask zero bits. The zero bits of the text character

mask are drawn in the background colour only when the mode is set to draw foreground and background colours.

There are four modes made from a combination of two switches, Direct-draw the foreground and background colours (MODE_DD_BOTH), Direct-draw the foreground colour only (MODE_DD_FORE), XOR the foreground and background colours with the presently held colours within the frame buffer (MODE_XOR_BOTH) and XOR the foreground colour only with the frame buffer (MODE_XOR_FORE). XOR mode is useful for drawing mouse cursors.

The 16-colour palette holds 12-bit RGB representations (4-bits red, 4-bits green, 4-bits blue), so there are a possible 4096 shades of colour that can be drawn, with only 16 viewable at any one time. The default palette is set up within the VGA controller by including packages with differing constant values. The set-palette procedures can be used at run-time to dynamically update the palette values.

The vertical blanking position is a 60Hz waveform that is true when the raster-scan is not one of the viewed 480 lines of the actual 525 raster-scan lines. The blanking period is used for the monitor to make the raster-scan fly back to the top of the screen ready for the next scan. By waiting for the blanking period to begin, it is possible to perform all the drawing procedures after this event (within the blanking period) so that the drawing is hidden. Conversely, the time could be used to swap raster pages, which would mean that tearing (a feature of changing the raster page midway through rastering) does not occur.

The four drawing procedures provided perform all the write access to the frame buffer. All except the character drawing take X-positions using the VHDL type '***vga_xpos_type***', and Y-positions using the VHDL type '***vga_ypos_type***' defined in the constants package. These are defined as constrained bit_vectors of lengths 10 and 9 respectively. They form the co-ordinate system of 640 by 480 resolution (can actually hold 1024 by 512). The origin is in the top left corner of the screen, with X increasing to the right, and Y increasing downwards.

The horizontal line drawing procedure is the most efficient and should be used if any horizontal lines are to be drawn. The rectangle-drawing algorithm should be used for any vertical lines or upright boxes, giving any two corners to draw the rectangle between. The

line drawing algorithm should only be used to draw lines of unknown orientation. Characters can be drawn on 8-pixel boundaries, and consequently their co-ordinates are stored as 7 and 6 bit wide bit_vectors using the VHDL types '***vga_text_xpos_type***' and '***vga_text_ypos_type***'. The characters can be drawn with different sizes in both X and Y co-ordinates. Each character is an 8 by 8 pixel mask that can be stretched to be drawn in 8 (**TEXTSIZE8**), 16 (**TEXTSIZE16**), 24 (**TEXTSIZE24**) or 32 (**TEXTSIZE32**) pixels. The size information is passed as '***vga_textsize_type***'. The character drawn is given as an ASCII number (8 bits) defined as '***ascii_type***'.

A.1.3 Original 16-colour controller

This was the first controller produced and has been revised since that time. This section explains the overall layout of the source code, explaining what each concurrent process does. It will not go into great detail of the source code. The controller is written to control the underlying DRAM memory [95]. The data path width is 32-bits, which allows 8 pixels to be accessed within one memory read/write.

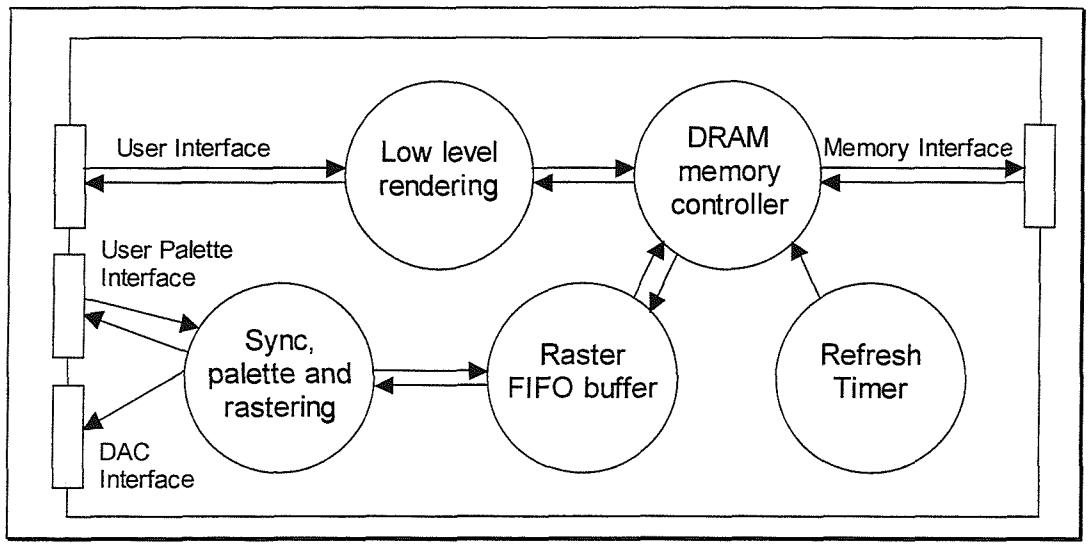


Figure A.5 DRAM-based VGA controller process communication

The user interface links to the ports provided by the user and driven by the interface package procedures. It controls the low level rendering system. This is the process that draws horizontal lines and text masked horizontal lines. The process itself is written in such a way to re-use the fundamental algorithm, but with slight changes for the two different drawing styles. Read-modify-write access is given, as the XOR mode, text drawing and line ends require that the present memory value is masked or used in some way in the new value to place into the same memory location.

The rendering process interfaces directly to the memory controller, which provides a port into the external single-port DRAM. The port provided is a read-modify-write port, with page mode access provided. The memory controller also provides another port for the raster FIFO buffer. This port has fast page mode read access. The process itself generates all the timing signals necessary to drive the external DRAM. Alternatives to the single-port DRAM, optimised for video applications exist [115], but are not used here.

As DRAM requires constant refreshing, a refresh timer is provided within a separate process that tells the memory controller to perform a refresh action. The refresh actions are buffered so a constant refreshing rate is guaranteed. Refreshing has the highest priority of any action on the memory but will not stop any other type of access once started.

The raster FIFO buffer is provided to increase the maximum data throughput bandwidth, as the memory controller has an associated latency. It also allows the use of fast page mode accesses within the memory controller. It stores 16 data words, each of which holds 8 pixels, so the FIFO holds 128 pixels at a time. The buffer is told to fill up with information when it goes below a certain threshold and is told to stop filling, when it becomes full. In this way, raster access takes only 1/3 of the available memory access time in large memory access bursts.

The sync and raster process is written in such a way that it can be optimised down to one control cycle. This is achieved with no read-after-write accesses of any local variables (those that are not optimised out), instead generating a pipelined data flow stream with the use of write-after-read accesses. It generates the VSYNC, HSYNC, blanking, vertical blanking and RGB output signals to feed the monitor and DAC. The clock frequency defines the dot-clock (25 MHz = 1 pixel output every 40 ns). This means that eight pixels (4 bits each) in one internal data-word (32 bits) are read every eight clock-cycles.

A.1.4 XESS 16-colour controller

The second controller took the first controller as a template and was re-designed to use a RAMDAC for the output stage (no internal palette lookup), and give an SRAM interface to the same amount of data space, but with a reduced data path width of 16-bits. The controller layout is shown in Figure A.6.

The low level rendering system and raster FIFO buffer are exactly the same as the DRAM version. The refresh timer process has been removed due to SRAM not requiring refreshes. An extra palette modification and RAMDAC setup process has been added to drive the data bus of the RAMDAC used on the XESS board.

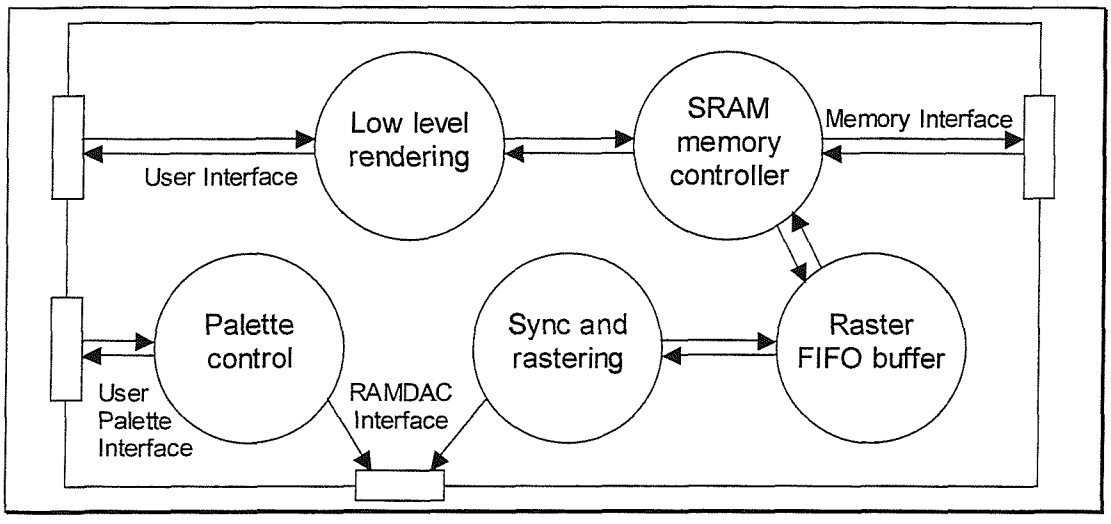


Figure A.6 SRAM-based VGA controller process communication

The sync and rastering process has changed in terms of not requiring internal palette lookup due to the external palette as part of the RAMDAC, but remains largely the same, outputting the palette index instead of the resolved RGB colour value.

The major change is to the memory controller, due to the use of SRAM instead of DRAM, but also due to the reduced data path width and increased address count. Internally, everything in the VGA controller still uses a 32-bit data path width, so the memory controller now has to perform 2 reads / writes for every data word. This is the source of inefficiency in the memory access method.

A.1.5 XESS 256-colour interface

This section describes how to use the 256-colour VGA controller, or more importantly, how to interface to the controller. This version has no rendering capabilities. Instead, two frame buffer direct memory interfaces are provided. There are two interface ports for the initial use of the 256-colour version.

The version of the controller that this interface uses can only use the SRAM frame buffer due to speed considerations.

There is also an extra output from the controller in this design, indicating when the palette modification is active. This is due to the shared RAMDAC data bus with the Ethernet IO signals with the FPGA. When palette modification is required, this extra signal goes high, which forces the Ethernet drivers to tri-state and enables the data lines to the RAMDAC.

All rendering procedures have been removed for the 8-bit version, leaving only selected procedures that have any meaning, such as '**vga_setrasterpage**' or '**vga_setpalette**'.

Only the raster page is defined. As no rendering occurs, there is no render page. There are now only two available pages, due to the increased memory cost of 8-bits per pixel instead of 4-bits per pixel.

Palette modifications occur in the usual way, by checking the palette modification semaphore and acknowledge signals, and only performing a modification when allowed. The procedures are blocking.

Vertical blanking exists for this version and is used in exactly the same way.

Two new procedures are added, which define the only method of reading and writing to the SRAM frame buffer. They are '**vga_mem_read**' and '**vga_mem_write**'. The port passed into these two procedures is one of the two defined for the VGA controller. Each port has its own semaphore and acknowledge. These two procedures form a direct memory access port into the framebuffer memory.

A.1.5.1 Using the interface

The requirements for interfacing to the 256-colour VGA screen are as follows:

- Include references to the constants and interface packages.
- Add a list of port signals to the user design as given in comments at the top of the interface package. Only include as many memory ports as are required. Port 1 has access priority over port 2.
- Call the initialise procedure (as defined in the interface package) passing the memory access semaphore '*mp1_sem*' and/or '*mp2_sem*' as parameters.

- Call any other memory access procedure as required, passing references to the relevant ports defined at the top of the user code and any other parameters that the procedures require.

The system is set up as a frame buffer that gets rastered to the VGA screen. Call **vga_setrasterpage** to setup which page is rastered. The user interface is now a direct memory interface, so knowledge of the addressing scheme is required for correct access. The memory data path is 16 bits wide, which contains 2 pixels, horizontally next to each other. The left-most pixel is contained in bits (15 downto 8) and the right-most pixel is contained in bits (7 downto 0) of the data path used in the direct memory interface.

The addressing scheme of the 2-pixel word is given in Figure A.7.

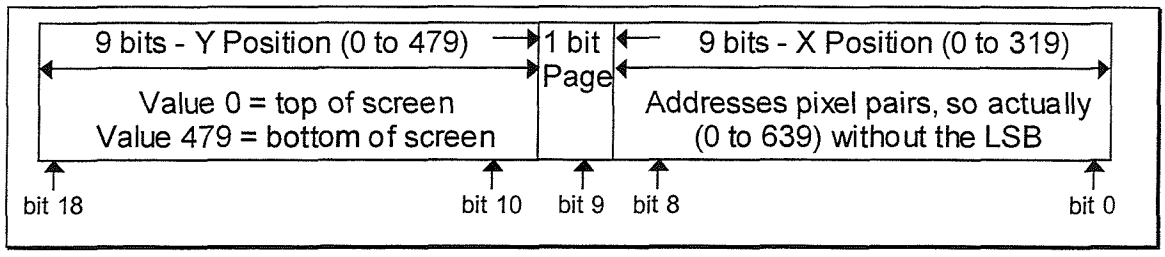


Figure A.7 Pixel addressing scheme for the 256-colour controller

A.1.5.2 Interface types

A number of VHDL types are defined for interfacing with the 256-colour VGA controller. These types are defined within the constants package and are shown below:

```
subtype vga_colour_type is bit_vector(COLOUR_BITS-1 downto 0);
subtype vga_red_type is bit_vector(RED_BITS-1 downto 0);
subtype vga_green_type is bit_vector(GREEN_BITS-1 downto 0);
subtype vga_blue_type is bit_vector(BLUE_BITS-1 downto 0);
subtype vga_rgb_type is bit_vector(RGB_BITS-1 downto 0);
subtype vga_page_type is bit;
subtype vga_xpos_type is bit_vector(XBITS-1 downto 0);
subtype vga_ypos_type is bit_vector(YBITS-1 downto 0);
subtype vga_framebuffer_address_type is bit_vector(FRAMEBUFF_ADBITS-1 downto 0);
subtype vga_framebuffer_data_type is bit_vector(FRAMEBUFF_DPBITS-1 downto 0);
```

A.1.5.3 Interface procedures

This section describes each interface procedure, the port list and use. The procedures are defined in the interface package.

```
procedure vga_initialise ( signal semaphore : out bit );
```

Description: Called once at startup within the process that is driving the VGA system to setup the interface semaphores '*mp1_sem*' and '*mp2_sem*'. Call once per semaphore.

```
procedure vga_setrasterpage (
  -- ports
  signal render_sem : in bit;
  signal render_ack : in bit;
  signal raster_page : out vga_page_type;

  -- user input
  page : in vga_page_type
);
```

Description: Set the viewed page to the '*page*' input. Page values can be **PAGE0** or **PAGE1**.

```
procedure vga_setpalette_rgb (
  -- ports
  signal palette_modify_sem : inout bit;
  signal palette_modify_ack : in bit;
  signal palette_modify_addr : out vga_colour_type;
  signal palette_modify_val : out vga_rgb_type;

  -- user input
  colour : in vga_colour_type; -- which colour to adjust
  rgb : in vga_rgb_type        -- the new rgb value
);
```

Description: Set the palette colour (0 to 255) with RGB (0 to 16777215). The '*colour*' input says which colour index to adjust the palette of. The '*rgb*' input gives the 24-bit concatenated RED & GREEN & BLUE value.

```
procedure vga_setpalette (
  -- ports
  signal palette_modify_sem : inout bit;
  signal palette_modify_ack : in bit;
```

```
  signal palette_modify_addr : out vga_colour_type;
  signal palette_modify_val : out vga_rgb_type;

  -- user input
  colour : in vga_colour_type; -- which colour to adjust
  red : in vga_red_type;       -- the red component of the palette
  green : in vga_green_type;   -- the green component of the palette
  blue : in vga_blue_type      -- the blue component of the palette
);
```

Description: Set the palette colour (0 to 255) with separate Red, Green and Blue values. The '*colour*' input says which colour index to adjust the palette of. The '*red*', '*green*' and '*blue*' values give the palette shade in the 8-bit triple.

```
procedure vga_wait_for_vertical_blanking (
  signal vert_blank : in bit );
```

Description: Wait for the vertical blanking period to begin.

```
function vga_vertical_blanking (
  signal vert_blank : in bit ) return boolean;
```

Description: Return whether the raster-scan is currently in the vertical blanking period (true) or rastering the memory contents (false).

```
procedure vga_mem_read (
  -- ports
  signal mp_sem : inout bit;
  signal mp_ack : in bit;
  signal mp_rd : out bit;
  signal mp_addr : out vga_framebuffer_address_type;
  signal mp_data_out : in vga_framebuffer_data_type;

  -- user input
  address : in vga_framebuffer_address_type;
  data : out vga_framebuffer_data_type
);
```

Description: The memory read access procedure. Use this with either of the ports into the frame buffer. The '*address*' input is the 19-bit address and the '*data*' output is the 16-bit result from the read (this contains two 8-bit pixels).

```
procedure vga_mem_write (
  -- ports
  signal mp_sem : inout bit;
  signal mp_ack : in bit;
  signal mp_rd : out bit;
  signal mp_addr : out vga_framebuffer_address_type;
  signal mp_data : out vga_framebuffer_data_type;

  -- user input
  address : in vga_framebuffer_address_type;
  data : in vga_framebuffer_data_type
);
```

Description: The memory write access procedure. Use this with either of the ports into the frame buffer. The '*address*' input is the 19-bit address and the '*data*' input is the 16-bit value to be written into the frame buffer (this contains two 8-bit pixels).

A.1.6 XESS 256-colour controller

The VGA controller with 8-bits per pixel is again derived from the original controller. The final design is able to raster to the screen using only half of the memory access time for the rastering process.

This level of data bandwidth was achieved with a different method for buffering the raster data-stream.

Figure A.8 shows the system layout.

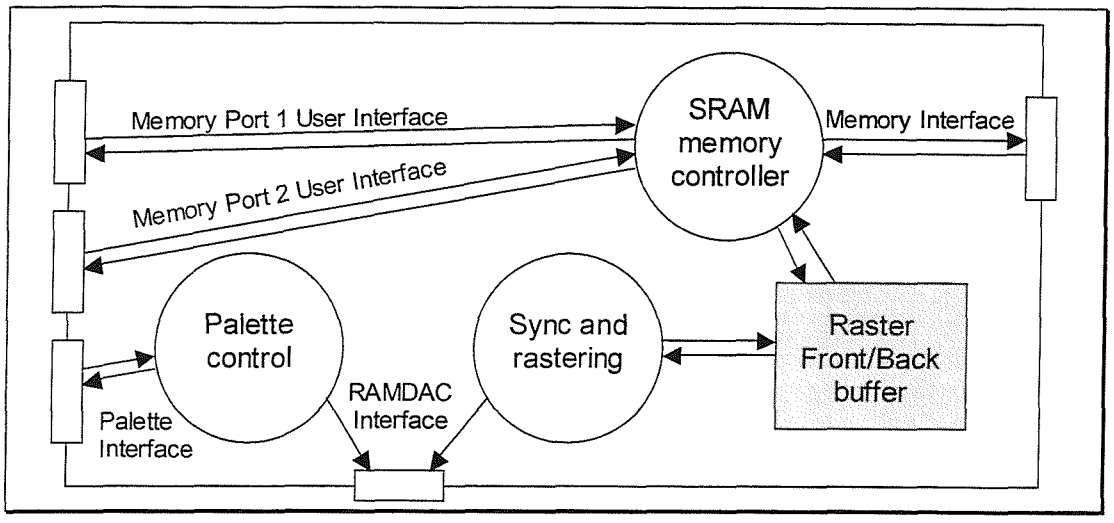


Figure A.8 SRAM-based 8-bit per pixel VGA controller process communication

The raster buffer is no longer interfaced to via a process. It is now written to by the SRAM memory controller directly and read from the rastering process directly. It is set-up not as a dual-port RAM, but as two RAM-arrays - a front/back buffer arrangement. While the system is rastering from the front buffer, the SRAM controller is filling the back buffer. This means that no addressing conflicts exist between the two buffers. When the rasterer reads the last address of the front buffer, the two buffers are swapped over and rastering now continues from what used to be the back buffer. At this point, the SRAM controller is triggered to fill up what used to be the front buffer with the next set of raster data.

The SRAM controller is able to read in one word per clock-cycle, which means that it can read two pixels per clock-cycle. The rasterer outputs one pixel per clock-cycle, so consequently this allows half of the memory access time to be used for the external user ports, which perform single accesses on the memory (two pixels at a time). Two user ports are provided at the request of the initial users of this particular controller.

This version of the palette no longer contains constant values for each colour, due to the shear number of constants required (256). Instead, a procedure is called to setup the default palette from within the RAMDAC setup process within the controller. All other RAMDAC access procedures are now stored within the palette package. The default palette is grey-scale, as this produces the simplest setup algorithm that gives a unique colour for each palette index value.

A.2 Keyboard controller library

The purpose of the keyboard controller is to form an interface with a standard PC keyboard from any design using MOODS. This forms a powerful direct user input method for many designs.

The library forms a concurrent component that is inherently timing critical due to the serial nature of the transmitted data from the keyboard. However, this criticality is removed if the design is written with the assumption that the keyboard controller is run at a greater speed than the serial data that forms its input. This means that behavioural synthesis becomes more suited to the problem. This is found to be the case in all demonstrators so far, as the serial data clock is in the order of kHz, and most designs use at least 1MHz as the system clock.

A.2.1 VHDL files

The interfacing method is the same as used in the VGA controller system, with a concurrent component accessed by interface procedures that drive an external port attached to the user's design. The three required VHDL files listed below.

Filename	**Description**
Keyboard_const.vhd	Constants package
Keyboard_controller.vhd	Concurrent controller component
Keyboard_interface.vhd	Interface package to the controller

Table A.10 Files required for the keyboard controller

The constants package is referenced within the controller component for the scan-code data type definition and the serial data width constant. The constants package also requires referencing within the user's design for the same data types and a number of constant scancode conversions that make the source VHDL easier to read. The user's design also requires a link to the interface package, which contains a number of interface procedures that can be called to determine whether any key has been pressed.

The controller file contains an entity/architecture pair that completely defines the actions of the controller. The port list of the controller takes the serial data lines as input and

drives the interface ports that the interface procedures link into. Note that a standard keyboard uses a common-collector drive, which enables bi-directional serial data flow used for keyboard setup, but the controller only implements the receiving of data from the keyboard. This means that the clock and data lines are taken as inputs only.

As the keyboard inputs are completely asynchronous to any design that uses it, with no access to the internal clock used by the keyboard itself, the serial data and clock require extra double buffering to stop the possibility of metastability occurring on the inputs from feeding into the controller.

The same theory is used between the user's design and controller design if different asynchronous clocks are utilised within each subsystem.

A.2.2 Controller

The controller component performs two actions upon the serial data to fully provide information about the keypresses occurring to the keyboard.

The first action is to translate the incoming serial data into a parallel representation, and the second is to perform partial translation upon this data to gain information about the keypresses such as whether the key pressed is an extended character and whether the key has been pressed or released. The basic control flow can be seen in Figure A.9.

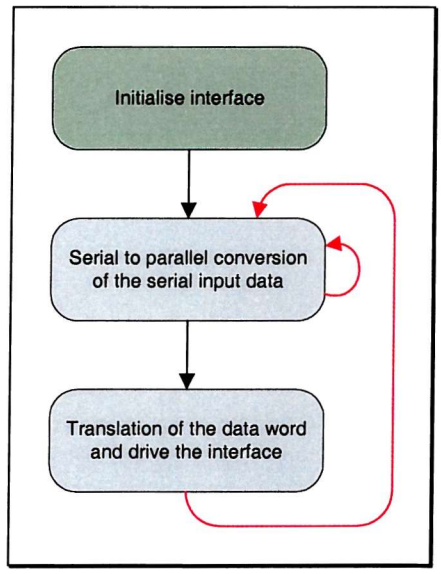


Figure A.9 Keyboard controller design flow

A.2.2.1 Serial to parallel

The two inputs generated by the keyboard are single signals. The first is a clock signal that is used to synchronise the other signal, the serial data line. The keyboard itself also requires a power supply.

The serial data is assigned on the rising edge of the clock, which means that the data can be read on the falling edge of the clock. An initial start bit is added to the serial data stream, which then has a single byte transmitted from the least significant bit to the most significant bit of the byte. A final stop bit finishes the stream for a single byte. The stream can be seen in Figure A.10.

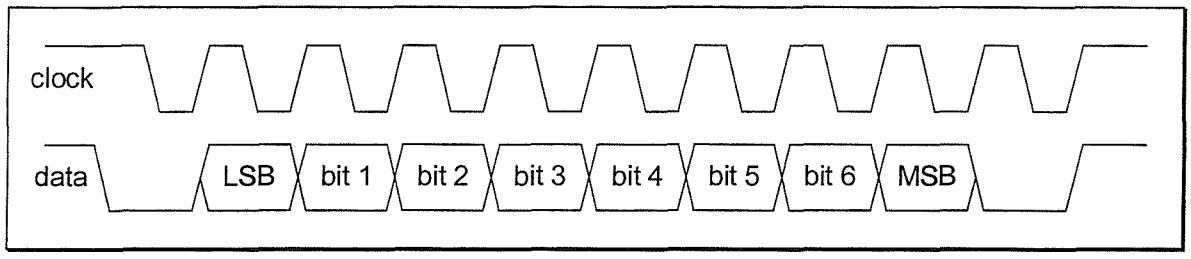


Figure A.10 Keyboard serial data stream

The method used to read this data into internal parallel storage is to shift the data into the most significant bit of an 8-bit shift register. After the initial start bit, the data is shifted for 8 cycles and then the final stop bit is acknowledged without shifting in the stop data. A counter is used to count the fixed number of input clock cycles.

A.2.2.2 Translation of meaning

Once the data is stored in an internal byte, the meaning of the data is partially translated. The keyboard sends a single byte for a key-press or auto-repeated key-press. This single byte is the scan-code for the particular key being pressed. If a valid scan-code is received, then the controller drives the interface semaphores to say that a valid key-press has occurred.

If the data byte is a special character hex '*F0*', this determines that the next byte to be sent will be a scan-code value. The meaning of this is that the scan-code sent in the next byte will indicate that the key has been released, not pressed.

Another possible special character is hex '*E0*', which determines that the following scan-code value will form an extended character. The use of these characters is for double-purpose keys such as the two sets of arrow keys, so that the same scan-codes can be used for each key, with finer-grained distinction being available through whether the scan-code given is from an extended key. If an extended key is released, then both '*E0*' and '*F0*' will be sent before the scan-code value itself.

The final special character is hex '*E1*', which means that the break key has been pressed. It is followed directly with seven other bytes that effectively presses and releases a single extended key. This situation is coped with by counting the bytes after the initial special character.

A scan-code is only released to the interface once all the precursor bytes have been read and translated into single bits that are transmitted through the interface so that the interface procedures can make use of them.

The returned translation of the keyboard data byte stream is illustrated in Table A.11. The '*SC*' item within the table represents a valid scan-code that is not any of the special characters: '*F0*', '*E0*' or '*E1*'. The '*xx*' item is a don't-care byte. Each stream becomes a single communication with the user's design, returning the scan-code and whether extended and/or released.

Input stream	Scancode	Extended	Released
SC	SC	False	False
F0-SC	SC	False	True
E0-SC	SC	True	False
E0-F0-SC	SC	True	True
E1-xx-xx-xx-xx-xx-xx-xx	E1	False	False

Table A.11 Keyboard data stream translation

A.2.3 Interface

The interface is defined in terms of interface procedures that the user passes the port parameters that link directly into the keyboard controller. As well as the relevant port parameters, the other parameters of the interface procedures will take other controlling parameters and return the scan-code with any other extra information.

The interface procedures act upon the interface semaphores and transmitted data to control the flow of information between designs. The acknowledge to the communication semaphore requires setting up at the beginning of the user's design using the dedicated setup procedure, passing the semaphore signal as a signal to modify. Note that the keyboard controller is the master of communication and the user's design is a slave.

The rest of the interface procedures will return the scan-code and whether the key is extended. Four versions of the procedure call exist that allow the user to call blocking or non-blocking versions and to either check or specify whether the key has been pressed or released.

A.2.3.1 Interface types

A single VHDL subtype is created for use when interfacing to the keyboard controller. It defines the storage required for the scancode and is defined within the constants package.

```
subtype scancode_type is bit_vector(KEY_SCAN_BITS-1 downto 0);
```

A.2.3.2 Interface procedures

This section describes each interface procedure, the port list and use. The procedures are defined in the interface package.

```
procedure keyboard_setup (
  signal kint_ack : out bit );
```

Description: Called once at startup within the process that is accepting keyboard input. The procedure sets up the interface acknowledge signal '*kint_ack*'.

```
procedure keyboard_getkey_wait (
  -- interface
  signal kint_sem : in bit;
  signal kint_ack : inout bit;
  signal kint_released : in bit;
  signal kint_extended : in bit;
  signal kint_scancode : in scancode_type;
```

```
  -- user data
  released : out bit;
  extended : out bit;
  scancode : out scancode_type
);
```

Description: The procedure will return only when the keyboard data stream is fully decoded. Information about the key pressed is passed by the outputs '*released*', '*extended*' and '*scancode*'.

```
procedure keyboard_getkey_nowait (
  -- interface
  signal kint_sem : in bit;
  signal kint_ack : inout bit;
  signal kint_released : in bit;
  signal kint_extended : in bit;
  signal kint_scancode : in scancode_type;

  -- user data
  valid : out bit;
  released : out bit;
  extended : out bit;
  scancode : out scancode_type
);
```

Description: The procedure will return straight away, even if no key-press data is available. The '*valid*' output tells the user whether a key has been pressed.

```
procedure keyboard_getkey_updown_wait (
  -- interface
  signal kint_sem : in bit;
  signal kint_ack : inout bit;
  signal kint_released : in bit;
  signal kint_extended : in bit;
  signal kint_scancode : in scancode_type;

  -- user data
  released : in bit;
  extended : out bit;
  scancode : out scancode_type
```

```
);
```

Description: The procedure will return only when a key action has been decoded that is the same as the given key-press direction of the '*released*' input.

```
procedure keyboard_getkey_updown_nowait (
  -- interface
  signal kint_sem : in bit;
  signal kint_ack : inout bit;
  signal kint_released : in bit;
  signal kint_extended : in bit;
  signal kint_scancode : in scancode_type;

  -- user data
  released : in bit;
  valid : out bit;
  extended : out bit;
  scancode : out scancode_type
);
```

Description: The procedure will return straight away, even if no key press data is available ('*valid*' returned as false). The procedure also filters out keyboard actions that do not comply with the direction of the '*released*' input by returning the '*valid*' output as false.

A.3 Serial port library

The standard serial port is similar in concept to the data transmitted through the keyboard controller. The main difference is that a timing clock is not provided. Instead of this, the serial data stream is generated with a known data rate: the baud rate.

A serial port interface is bi-directional, with two separate data channels for each directional flow. In fact there are four channels, with the two other channels used for flow control. The typical configuration though is in null-modem form that just utilises the two data channels. This is the form of controller that is implemented.

There are now a wide diversity of baud rates that form the standard settings for a serial interface transmission, ranging from 120 data bits per second up to 256k data bits per second. These differences are due to the standard serial port protocol surviving through a large number of technology speed-ups. Any baud rate is supportable within the implemented controller with simple external constant changes.

The standard data is sent as packets of up to eight bits. One start bit is always required for data synchronisation; with the possibility for a parity bit for error checking and the stop bit length can be adjusted. The implemented controller accepts only one protocol, that of one start bit, eight data bits, no parity bits and one stop bit, which allows full bytes to be transmitted at any time.

The system is split into two separate halves, one for the transmission of serial data and the other for receiving of serial data. Only the data receiver is implemented at the present time with scope built in to the interface to transmit a data stream. This is because the only system requiring this capability to date is the tracker demonstrator (see Appendix C). The data transmitter is relatively simple in comparison to the receiver.

A.3.1 VHDL files

The interfacing method is the same as the VGA controller and keyboard controller systems, with a concurrent component accessed by interface procedures that drive an external port created within the user's design. There are four VHDL files listed in Table

A.12 that implement the whole library. Note that the transmitter controller component is not currently implemented.

Filename	**Description**
Serialport_const.vhd	Constants package
Serialport_interface.vhd	Interface package to concurrent controllers
Serialport_receive_controller.vhd	Receiver controller concurrent component
Serialport_transmit_controller.vhd	Transmitter controller concurrent component

Table A.12 Files required for the serial port interface

The constants package is referenced by the controller components for the data type definition and the serial data width constant. The constants package also requires referencing by the user's design for the same data types and a number of constant values that define the baud rate for various controller clock speeds. The user's design also requires a link to the interface package, which contains a number of interface procedures that can be called to either transmit or receive serial data.

A.3.2 Receiver controller

The receiver controller is implemented using two concurrent processes that communicate with each other using internal signals. The first process controls the flow of each bit into the system, while the other process with which the first communicates acts as a serial bit-rate timer for synchronisation timing with the incoming data stream.

The timer process takes a constant value that defines the rate of data flow (baud rate) after the controller clock is taken into account. Exact timing can be produced with careful VHDL code that is passed through MOODS. This is achieved by the implementation of the timing process, which has one control state that activates itself continuously.

The timer process can act in one of two modes, either count for a full serial data period, or count for half that data period. The reason for this is due to the position of reading the incoming data flow. Synchronisation is achieved by sensing the start bit of the data flow. Straight after this, the timer is set going to count for half a data period. The timer communicates with the controlling process to say when this period is over. This means that control is given back to the controlling process half way into the start data bit. It is at this point that the data is most likely to be stable.

The rest of the data bits are then read in one after the other with the intermediate times counted by the data rate timer that is now set to count for whole data periods. This means that every bit after the start bit is read in the middle of the data bit. This will only work when you receive data that has been transmitted at the same baud rate.

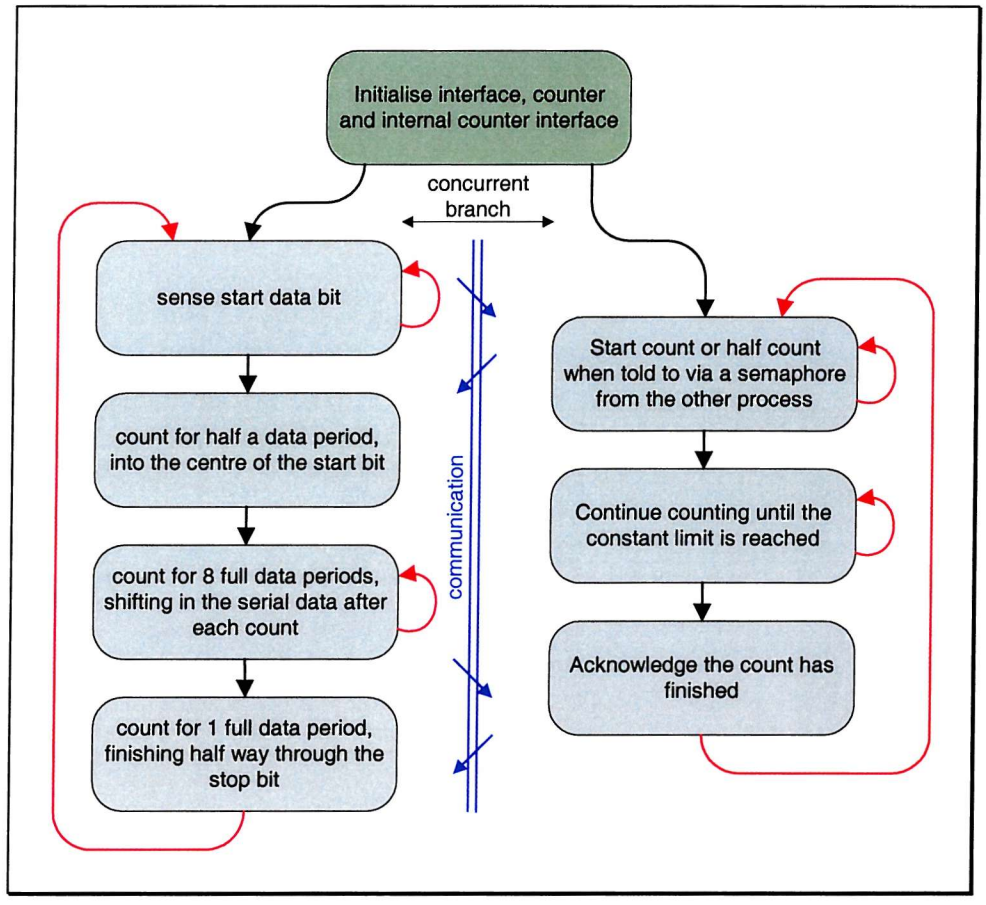


Figure A.11 Serial port receiver controller design flow

The conceptual control flow is shown in Figure A.11 above. The right hand flow represents the data rate timer that is started by the main controlling process represented by the left hand flow. The controlling process uses the timer in the last three conceptual states. The first state takes the control flow into the middle of the start bit, which is always '1'. The next state loops eight times for each data bit, reading in the data stream into a shift register. The last state reads for another whole data period, which leaves the controller reading the middle of the stop bit, which is always '0'. As the stop bit is always '0', it is safe to leave the controller reading the middle of that bit. This enables re-synchronisation with the next data stream that is sensed on the next rising edge of the incoming data. The serial to parallel data stream reading is shown in Figure A.12.

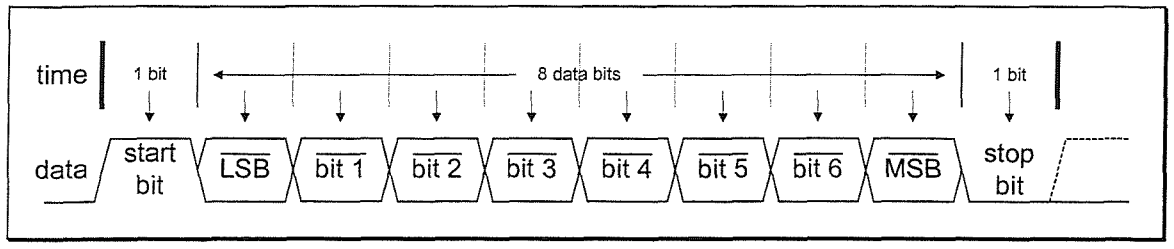


Figure A.12 Serial link data stream

Data is read at the mid-point of each data bit so that the data stream is at its most stable. Note that the actual data is inverted, so the serial data is fed through an inverter before being shifted into the shift register that will contain the data byte at the end of the serial data stream. A single sample is taken with no oversampling.

The timing constant is calculated from a formula that takes the required baud rate (bits per second) and the system clock rate (Hz) as parameters. The constant value generated is the value that the counter process needs to count that will represent the period of one data bit. An exact representation is not necessary due to the re-synchronisation characteristics of the data stream before every transmitted byte. The formula is shown below.

$$count = \min\left(\frac{system\_clock}{baudrate}\right) - 1$$

A.3.3 Interface

The interface is formed from procedures that access a port into the two controllers. Only the receiver controller is implemented, but the interface to the transmitter controller exists within the interface library. The interface is set up using the initialisation procedure provided. After that, it is simply a case of calling the set of four procedures to transmit and receive data. Two versions of transmit and receive procedures exist. The first pair are blocking procedures in that they will only return once the data has been transmitted or received. The second pair are non-blocking and will return straight away even if no data has been transferred. A returned bit tells the user whether any data was transmitted or received.

All communication with the controllers is formed using semaphore-acknowledge signal pairs, with data I/O sequenced using the semaphores.

All communication is un-buffered, which means that a transmission will block over a number of bytes sent and the receiver has to accept data at the byte-rate or above for the data not to be corrupted by missing received bytes. The act of buffering the serial port is left to the user if required. This is indeed what happens within the tracker demonstrator.

A.3.3.1 Interface types

Two VHDL types are defined for interfacing with the serial port controllers. These types are defined within the constants package and are shown below:

```
subtype serialport_data_type is bit_vector(SERIALPORT_DATA_BITS-1 downto 0);
subtype serialport_baudrate_type is bit_vector(SERIALPORT_BAUDRATE_BITS-1 downto 0);
```

A.3.3.2 Interface procedures

This section describes each interface procedure, the port list and use. The procedures are defined in the interface package.

```
procedure serialport_initialize (
  signal serialport_sem_ack : out bit );
```

Description: Called once at startup within each process that receives or transmits serial data. The procedure sets up the transmission semaphore '*serialport_trans_sem*' or the receiver acknowledge signal '*serialport_recv_ack*'.

```
procedure serialport_transmit_data (
  -- interface
  signal serialport_trans_sem : inout bit;
  signal serialport_trans_ack : in bit;
  signal serialport_trans_data : out serialport_data_type;

  -- user data
  dataword : in serialport_data_type
);
```

Description: Transmits a single data word '*dataword*' through the serial port, blocking the user's design until transmission is possible.

```
procedure serialport_transmit_data_nonblocking (
  -- interface
  signal serialport_trans_sem : inout bit;
  signal serialport_trans_ack : in bit;
  signal serialport_trans_data : out serialport_data_type;

  -- user data
  transmitted : out boolean;
  dataword : in serialport_data_type
);
```

Description: Transmits a single data word '*dataword*' through the serial port, without blocking the user's design. The output '*transmitted*' tells the user whether the data was sent.

```
procedure serialport_receive_data (
  -- interface
  signal serialport_recv_sem : in bit;
  signal serialport_recv_ack : inout bit;
  signal serialport_recv_data : in serialport_data_type;

  -- user data
  dataword : out serialport_data_type
);
```

Description: Receive a single data word '*dataword*' through the serial port, blocking the user's design until data is received.

```
procedure serialport_receive_data_nonblocking (
  -- interface
  signal serialport_recv_sem : in bit;
  signal serialport_recv_ack : inout bit;
  signal serialport_recv_data : in serialport_data_type;

  -- user data
  received : out boolean;
  dataword : out serialport_data_type
);
```

Description: Receive a single data word '*dataword*' through the serial port, without blocking the user's design. The output '*received*' tells the user whether any data is received.

A.3.4 Serial port pin specification

The serial port standard also includes a number of pin specifications that are shown here. There are two standard pin specifications for the two different connectors that are used. These connectors are a 9-pin D-type and a 25-pin D-type connector. The pin-out specification is shown in Table A.13.

9-Pin	25-Pin	Acronym	Full-Name	Dir	Meaning
3	2	TxD	Transmit Data	→	Transmit data from port
2	3	RxD	Receive Data	←	Receives data into port
7	4	RTS	Request to send	→	RTS/CTS flow control
8	5	CTS	Clear to send	←	RTS/CTS flow control
6	6	DSR	Data set ready	←	Incoming data ready
4	20	DTR	Data terminal ready	→	Outgoing data ready
1	8	DCD (CD)	Data carrier detect	←	Modem connected to another
9	22	RI	Ring indicator	←	Telephone line ringing
5	7	GND	Signal Ground	-	Earth

Table A.13 Serial port pin specification

A.4 Wave viewer

This software project was written with the need to display a set of digital and analogue waves. Simulation is an essential part of any design flow. Whilst commercial simulators all obviously have their own display subsystems, the synthesis flow in MOODS goes through two intermediate forms, ICODE and DDF. Simulation of these is addressed by other projects, but a common viewer is an extremely useful tool. The wave history file used in the viewer is a simple text file explained in the next section. The program is written for a windows-based environment using MFC.

A.4.1 Wave file

The file used to display the events on a number of signals from a digital simulator is defined here. It contains signal drawing set-up information and the actual history of each signal. Comments can be added on each line by the delimiter #. Signal types can be defined by the keyword ***TYPE*** and can be based on an enumeration, integer or analogue value by the keywords ***ENUM***, ***INT*** and ***ANALOGUE*** respectively.

The definition of an enumeration is used for discrete signals such as the VHDL 'bit' that can have values '0' and '1'. The set-up of an enumeration includes the method of display for each enumeration value and the initial value. The definition of an integer allows a wave to have any discrete values within the specified range and also specifies the initial value. This wave is displayed by value. The definition of an analogue signal allows a wave to have any floating-point value within the specified range and also requires an initial value. This signal is displayed as an actual analogue waveform. No interpolation between values is performed at present.

The starting time is given using the keyword ***TIME***, which also defines the time scale in terms of a unit ***s***, ***ms***, ***us***, ***ns***, ***ps*** or ***fs***.

Signals are defined using the keyword ***SIGNAL*** after which the name of the signal is supplied along with the type of signal that must be one of the previously defined types.

Vectors of signals can also be defined using the keyword ***VECTOR*** after which the name, type and vector length is supplied. It is planned in future to allow vectors of vectors of any recursive depth, but this is currently not implemented.

The rest of the file is made from wave history values for each signal. These lines start with a time value. The next item is the name of the wave that is changing at the specified time point. Listed after the name is the new value (or values for vector waves). An example wave file is shown below:

```
1   # this is the first wave view file
2   # created by Dan on 6/12/98
3
4   # TYPE is a data type that you use
5   # for instance bits or std_logic are an Enumerated Type
6   # or a byte is a restricted integer
7   # or a voltage is an analogue signal
8   # implemented types at the moment :
9   # TYPE ENUM     name numVals InitialVal
10  #               list of ENUM values with drawing style
11  # TYPE INT      name min     max        InitialVal
12  # TYPE ANALOGUE name min     max        InitialVal
13
14  # defining type 'bit' with 2 enumeration types
15  # with '0' being the default starting value
16  TYPE ENUM bit 2 0
17  0 LOW  BLACK
18  1 HIGH BLACK
19
20  # defining type 'std_logic' with 9 enumeration types
21  # with 'U' being the default starting value
22  TYPE ENUM std_logic 9 U
23  0 LOW  BLACK
24  1 HIGH BLACK
25  U BOTH GREEN
26  X BOTH RED
27  Z MID  BLACK
28  W BOTH BLUE
29  L LOW  BLUE
30  H HIGH BLUE
31  - BOTH PURPLE
32
33  # defining type 'byte' as an integer range 0 to 255
34  # with 0 being the starting value
35  TYPE INT byte 0 255 0
36
37  # defining type 'voltage' as an analogue floating point
38  # value with range -1.8 to 7.54 with 0 being the starting value
39  TYPE ANALOGUE voltage -1.8 7.54 0.0
40
41  # Time definition - define start time (float)
42  # and time scale (s,ms,us,ns,ps,fs)
43  TIME 0.0 ns
44
45  # signal definitions (using types)
46  # two types of signal : (1) Single, (2) Vector
47  SIGNAL test1 bit
48  SIGNAL test2 std_logic
49  SIGNAL test3 byte
50  SIGNAL test4 voltage
51  VECTOR test5 bit       8
```

```
52   VECTOR test6 std_logic 4
53   VECTOR test7 byte      2
54   VECTOR test8 voltage   2
55
56   # wave history lists
57   1.0 test1 1
58   5.0 test1 0
59   6.0 test1 1
60   7.0 test1 0
61
62   0.5 test2 U
63   1.5 test2 0
64   2.5 test2 1
65   3.5 test2 X
66   4.5 test2 Z
67   5.5 test2 W
68   6.5 test2 L
69   7.5 test2 H
70   8.5 test2 -
71
72   3.2 test3 128
73   3.8 test3 255
74   4.5 test3 1
75
76   2.1 test4 0.01
77   2.2 test4 0.5
78   2.3 test4 1.5
79   2.4 test4 2.5
80   2.5 test4 3.5
81   2.6 test4 4.0
82   2.7 test4 4.3
83   2.8 test4 4.6
84   2.9 test4 4.8
85   3.0 test4 4.9
86   3.1 test4 4.95
87   3.2 test4 5.0
88
89   10   test5 00001111
90   12.5 test5 01011110
91
92   10   test6 XXXX
93   11   test6 ZZ00
94   12   test6 11ZZ
95   13   test6 LLHH
96
97   1.0  test7 0   255
98   2.8  test7 128 254
99   5.6  test7 129 253
100  12.3 test7 129 252
101
102  3.6  test8 0.36 0.0
103  17.3 test8 5.0  0.0
104  34.0 test8 4.3  0.5
105  45.1 test8 5.0  2.5
106  73.0 test8 4.9  4.8
107  80   test8 1.0  5.0
108  # end of file
```

A.4.2 User interface

The user interface is very simple to use. Open the file created from the simulator and the initial view will be of the whole wave history. Four buttons at the bottom left of the window allow the user to zoom in and out in time. The **LAST** button will zoom to the last

position shown before a button press. The **RANGE** button will zoom out to the full history range and the **IN** and **OUT** buttons will zoom in and out. The horizontal scroll bar allows the user to display a selected time period when zoomed in and the vertical scroll bar determines which waves are displayed if not enough room exists to display them all. A screenshot of the program displaying a portion of the file shown above is shown in Figure A.13 below.

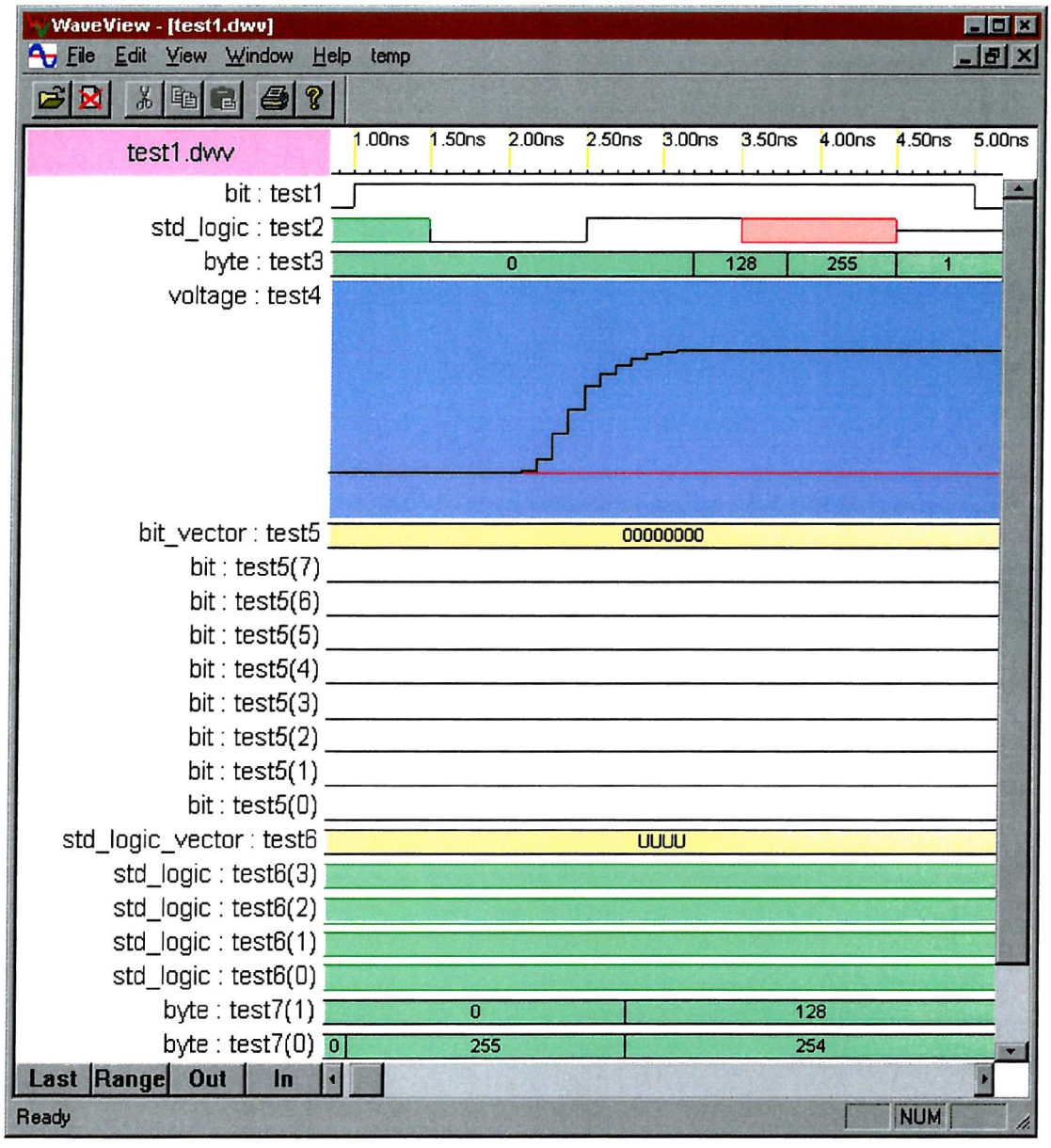


Figure A.13 Wave viewer screen shot

A.5 DDFLink

The DDFLink system was originally written to be a linker for various sub-designs produced by MOODS. The plan was to build an object capable of holding the entire data structure that MOODS contains. This object would then be made into a persistent object with the addition of reading and writing file I/O functionality using the DDF file format that was previously used within the expanded modules section of MOODS and is generated as one of the last processes in MOODS.

The reason a linker was necessary was due to the methodology of having a separate heap memory controller that formed the underlying heap management structure for the dynamic memory allocation. There was also scope for making the linker more general purpose in that many designs are built from a multitude of concurrent components that require joining together at the structural level to form a complete design.

This can be seen from the first three sections of this Appendix, where each VHDL project produces a concurrent component that requires linking into the user's main design.

The required functionality of the linker is to load in the multiple designs (in internal MOODS data structure form), perform the linking actions upon the data structures, which creates a single DDF object from the multiple input DDF objects. Then, the structural output VHDL is generated directly from the linked DDF object.

However, the full project had to be significantly reduced in scope due to increasing time pressures. It was felt that a full automatic linker, though desirable, was not an essential part of the dynamic memory system that is discussed in Chapter 4.

The reduced project now forms the basis for a data structure converter for the MOODS internal data structures. The complete MOODS data structures can be held within the DDF object as planned, the object is fully persistent and structural VHDL can be generated directly from this persistent object. DDFLink forms a separate console process that is executed after each synthesis run of a design.

The DDFLink project was not written from scratch. It utilised a large section of code that was written specifically for expanded modules within MOODS. The expanded module code creates a number of objects that represents a large proportion of the entire MOODS data structures. It is from this code that the DDF object is based. This code had the definitions of the internal data structure objects and a nearly complete parser for inputting those objects from a DDF file. From this basis, a completed object was produced with a full input parser. The object was then made persistent by writing the output routines from the object that recreates the DDF file used for input. In this way, if any modifications to the DDF object occur, the results can be stored back into another (or the same) DDF file.

The object persistence routines that formed the output DDF file were generated from a rough translation of the actual MOODS internal data structures that are implemented in a different style of C.

The VHDL output file that forms the translation from the internal DDF object into a VHDL form that can be parsed by a third party RTL synthesis tool was generated in much the same way as the object persistence routines, in that a rough translation from the old output stage of MOODS was made to use the newer DDF object being created for DDFLink. The VHDL output was then modified to produce a more readable and traceable version of the output file. This was accomplished by adding more comments and more importantly, removing a lot of indirection between data path nodes and control nodes.

A.5.1 DDF object

Sections A.5.1.1 to A.5.1.16 contain brief notes on the internal data structures used in DDFLink. They are intended to be used in conjunction with a source browser and the MOODS internals documentation [66] and do not stand alone. The data structures mirror the core structures used within MOODS.

The DDF object is created as a single class with no base class. Within the class, a number of structures and classes are defined. These define the subcomponents of the DDF object. These substructures are used internally to hold the entire data structure. A single list template class is utilised that allows generic doubly linked lists to be created as wrappers around the various data structures.

The DDF object is the root container for a full MOODS design. It contains the root lists of modules, control nodes, control arcs, ICODE instructions, data path nodes, data path nets, ICODE variables and conditions. It also points to the module that forms the main program.

The sub-objects that are directly and indirectly contained within the DDF object are explained in the next subsections.

The main links to sub-objects that the DDF object contains are the list of modules and the program module pointer, the list of data path nodes and the list of conditional signals.

A.5.1.1 Module

A module represents a complete flow of control. This could be a conversion of a single procedure, or the conversion of the root design, with the various concurrent branches representing the concurrent processes. The module is identified by a unique ID and by a string name that represents the original name of the translated procedure or entity name.

Each module contains a list of control nodes that implement the entire control flow for the module. This is achieved with a single pointer to the starting control node and a list of end nodes. A single ICODE instruction is used within the module to represent the I/O parameter list. This is the header instruction used within the ICODE file. A link to a single conditional signal defines the end signal for the module.

The actual I/O parameters of the global design are held in a number of 'ModPin' structures.

A.5.1.2 ControlNode

Each control node has a unique ID in the form of an integer. There are also various node types, which are represented by an enumeration. The probability of the node be active is also given, which allows power calculations to be made.

The control node structure contains a link back into the module that contains the control node. There is another link to a module that is only used within a call control node. This link defines the module to activate on the call.

The control graph is implemented using the control node and control arc structures. These, when linked together, form a completely contained graph of control flow. The control node has a list of input arcs that activate the node, and a list of output arcs that activate the next node. These links only form the structure of the graph, with control of the flow performed by other structures.

Each control node has a conditional signal defined that is linked into the implementing cell. This determines when the node is active. Another conditional signal is defined here for call nodes. This signal defines the end signal linked into the call control cell.

Each control node will implement a number of ICODE instructions. These are linked to via a list of 'Instruction' structures.

A link to the underlying cell that implements the control node is made via an integer ID representation of a cell number within the technology dependent library.

A.5.1.3 ControlArc

The control arc structure forms the arc between the control node structures. A single link to the previous control node and another single link to the next control node are contained within this structure.

The conditional signal used to determine whether this branch of control flow is taken or not is contained in the control arc. Each arc also has a unique ID and a probability factor. The arc also knows whether it is a feedback arc.

A.5.1.4 Instruction

The Instruction structure represents an ICODE instruction. As such, it has a unique instruction ID and a reference to a group of instructions that define a data dependent flow of control. The instruction type is contained as a member of the structure, along with the input and output parameters of the instruction, held in two lists pointing to the 'InstIO' structure. Extra parameters dependent on the instruction type are contained in a member union of data types. A link to a list of data path nodes is contained. The data path nodes listed performs the actions of the ICODE instruction.

A link back to the module that contains the instruction is given. This is replicated by the link within the control node that contains the instruction.

As the instruction has a close resemblance to the original source code, a link back to this code is given with two links into the 'file_info' structure, that reference the source VHDL file position and the generated ICODE files instruction position.

Once the final stage of MOODS has been performed, a number of extra conditions exist. Two of these conditions are held in the instruction structure. The first is for general instructions and determines the exact point at which the instruction is executed. The second was an addition made for recursion. This instruction is used within recurse call nodes, and determines the end condition for a particular recurse control cell.

A.5.1.5 InstIO

The 'InstIO' structure contains the I/O parameters for every ICODE instruction represented by the 'Instruction' structure. This structure points to either an ICODE variable (could be a temporary variable) or a constant. If a constant is represented, then the structure must feed only an input of an ICODE instruction. A variable link is formed from a direct link to the 'Variable' structure, and a constant is represented by an integer with a defined base.

A.5.1.6 DataPathNode

The data path node structure is a representation of the linkage to the actual cell that is the implementation. A link to the underlying cell that implements the data path node is made via an integer ID representation of a cell number within the technology dependent library.

The instructions that the data path node implements are held within a member list of the structure. An adder, for instance, could be shared among various add ICODE instructions.

The node has a unique number and type. The width (number of bits) of the data that flows through the node is also given here.

The data path is formed from the linking of various data path nodes together via an intermediate data structure, the 'DPNet' structure. These nets form the inputs into the data

path node and the outputs from it. Multiple nets could be formed each way, so a list of nets is used to represent the I/O. Data path nodes represent cells such as registers, adders and multiplexors. These cells require some form of control. This is formed from a link into the 'DPControl' structure with another list of controls per bit of the data path node.

A.5.1.7 DPNet

The 'DPNet' structure, or the data path net to give it its full representative name creates every link between the data path nodes. It does this indirectly through the 'DPNetPin' structure. It has a link to the source data path node and the destination data path node through two links to the 'DPNetPin' structure. It also contains a link to a conditional signal that is used to control multiplexor select control signals. This select signal is indirectly created from the list of instructions that that are also linked to within the 'DPNet' structure.

A.5.1.8 DPNetPin

The reason that the 'DPNet' structure does not link directly to its source and destination data path nodes is that in certain cases, the link is not to a data path node at all. This is the reason for the 'DPNetPin' structure. It is possible for this structure to link to a data path node, a conditional signal or a constant. Obviously, the constant can only be the source part of the 'DPNet', as a constant cannot be the destination of a calculation.

The 'DPNetPin' structure has a member determining the type of linkage. It also has a union member that contains the links to the relevant data type. The structure also contains the active bit-range of the connection, a link to an ICODE variable if the link is found to drive a data path node with a variable representation and an activation instruction and condition that tells when the net will be active.

A.5.1.9 DPControl

This structure forms the controlling input to data path nodes. It links to a particular pin of the data path node and a specified range of the bits controlled within the node. It has a reference to the ALU pin to access in the case of the data path node cell representing a multi-function node type. It also has a reference to the activating ICODE instruction and controlling conditional signal. The conditional signal may be null, which means that the

control input is fed with a constant zero. This situation occurs when particular data path node control inputs are unused, such as for the '*Clear*' and '*Set*' inputs of most generated register nodes.

A.5.1.10 Variable

The 'Variable' structure references an ICODE variable that has been maintained through the optimisation process. The reason for this structure is so that the generated VHDL files that are created from the data structures have some form of correlation with the inputted design.

Each variable has a unique ID and is of known type. The variable has a name that is representative of the original variable name that flows through from the compiler. The variable also has known width and knows from which module it is created. A link back to original source code and the representative ICODE file positions are gained from two links to the 'file_info' structure.

A data path node in the generated structures represents a variable, so a direct link to the relevant node is contained within the structure. A number of extra parameters are also stored, dependent on the variable type. If the variable is an alias then a link to the parent variable is given, along with the bit-range of the parent variable that it aliases. If the variable is representative of a ROM, then an extra link is made into the ROM constant data that is used directly within the outputted VHDL file. This link is to the 'const_node' structure. The variable could be representative of an I/O port, in which case a reference of the relevant 'ModPin' structure is formed.

A.5.1.11 Condition

A conditional signal represents a single bit equation. The 'Condition' structure is used throughout the DDF object to represent the linking signals between the control and data paths and visa versa. The signal has a unique integer identifier and a reference to a containing net. The actual Boolean equation is formed from the 'BoolEqn' class object, which is linked to within the 'Condition' structure.

A.5.1.12 BoolEqn

The Boolean equation structure is created from an internal binary tree representation. Each binary node of the tree can represent an equation operator. The operators allowed are 'AND', 'OR', 'XOR' and 'NOT'. The structure of the binary tree represents the hierarchy within the equation. The source of the signals that the equation operates upon can be from many sources. The sources allowed are an ICODE variable, a control node active signal, another conditional equation or a constant.

A.5.1.13 Const_node

This small structure is used to contain a single ROM value. Generating a number of these structures, and filling the contents with the ICODE ROM values create the full ROM.

A.5.1.14 ModPin

This small structure contains information about the module I/O lists. Both the width of the port and the link to the representative variable is contained, along with a pin number of the data path node that represents the modules I/O parameters.

A.5.1.15 CaseSelect

The 'CaseSelect' structure is used within the ICODE instruction when the multiple alternative control flow is created with the use of the 'SWITCHON' or 'DECODE' instructions. The ICODE instruction contains a list of these structures with the number of items in the list dependent on the number of alternatives to the ICODE instruction.

The structure contains a constant number that represents each alternative. A constant '-1' represents the default alternative. The structure also contains a link to the conditional signal that gets activated on the given switch alternative. This effectively means that the created signal will be formed from a particular output of a decode data path node.

A.5.1.16 File_info

This structure was retrofitted into the variable and instruction structures. It contains a single reference to a source file position; both line number and column position. The file

reference is made via an integer index into a map of these indexes with the string representations of the full path of the filename.

A.5.2 DDF parser

Reading an ASCII representation of the object from a file creates the DDF object. The structure of the file means that it requires parsing. As with any language, the parser uses a lexical analyser to read the input data in the form of lexical tokens.

The parser to feed the data structures with the relevant information then uses these tokens. Two passes of the file are required due to the forward declaration of various objects within the file. The first pass builds up most of the data structures, but missing most of the links between the structures. The second pass is used to fill in these links.

The reason a parser is required in the first place is that the ASCII representation is human readable. It is not wise to manually edit the DDF file unless the user knows exactly how it represents the underlying object.

The definition of this internal proprietary language is found in Appendix D.

A.5.3 DDF output

The generation of the DDF file is simpler than the reading of the same file. The DDF file is generated from the internal data structures in exactly the same form as it was read into the objects. This means that the objects have file-based persistence.

The definition of the internal proprietary language 'DDF' or 'Design Data Format' is found within Appendix D. The file is a direct representation of the data structures. The output dump was written from scratch using the MOODS DDF dump as a template for the style of file to be produced.

A.5.4 VHDL output

The DDF class is used to generate the final structural VHDL. It performs this operation in a near one to one relationship with the underlying data structures. This is because the

underlying data structures are designed to represent a structural representation of the behavioural design inputted by the user.

The VHDL output consists of various areas of the file that correspond to the various data structures within the DDF object. These main areas are the conditional signal equations, the control graph representation and the data path representation. Between these three sections, links are made directly between the items created with the declared signals that are produced within each section. These signals are declared within the single architectural representation of the entire design. The body of the architecture contains the instantiation of the control path, data path and conditional equations. The file also contains an entity declaration that defines the interface to the outside world. This interface is similar to the initial entity port declarations of the source code, except that clock and reset signals are added.

The direct translation of the original methods for outputting of the structural VHDL was only the first stage of creating the new output style of VHDL. The underlying methods for output stayed the same, yet the code produced is more readable and traceable with the removal of all indirection between data path nodes in the form of the data path nets. Whenever a reference to a data path net is found, the link, in the form of the data path net and data path net pins is traced back to the connecting item, and the signal that forms the output of that item is used instead.

A number of information comments are now also passed through the system, which tells the reader of the VHDL file where all the inputs to the data path nodes are derived from, what the connectivity of the control path signals are and what instructions are executed by each data path node. Variable names are used where applicable, with shared variable definitions being defined also.

The underlying conditional equation store was found to be inefficient, so a simple indirection removal produces links into the condition store that can be derived directly from the inputs to the condition store. This allows direct linkages through the conditional equations to be bypassed completely, resulting in more traceable linkage between the control and data paths.

The data path nodes are implemented by a component instantiation of a pre-defined generic component. The width is passed into the component, which defines the size of the item to be produced. The library that contains these underlying components is optimised for RTL synthesis. Control nodes are implemented within the same library and in the same manner. Control nodes correspond to control states in a one to one representation. The linkage between control nodes are formed from the signals defined for each node and the conditional signals used to direct the flow of control dependent on the actions of the data path.

A.5.5 VDF output

While developing the DDF system, a graphical user interface (GUI) was also being produced. This GUI enables multiple views into the synthesis process to be created. One of these views is a direct window onto the DDF data structures, displayed in graphical form. At the time of writing, one main view existed into the internal data structures of MOODS; that of the module list with contained control graph (nodes and arcs) with instruction linkages.

The method for input into the GUI is via two methods, the first is by direct transmission between the concurrent programs via a pipe and the second is via loading a representation of the DDF object from a file. The VDF file was produced as an alternative to the DDF representation due to the loading speed of a full parse of the whole DDF file.

The VDF representation contains only the information required about the control graph and instructions contained within the graph. The file is stored in exactly the same method as the pipe transmission data, which means that only one pass is required to build up the full representation within the GUI. The file is ASCII based, though a binary version would be more efficient.

The VDF file is created within the final internal object conversion stage at the same time as the resultant structural VHDL file is generated. The GUI displays it by loading the DDF file (which links to the VDF file internally). An example of the displayed graph produced by the GUI is shown in Figure A.14.

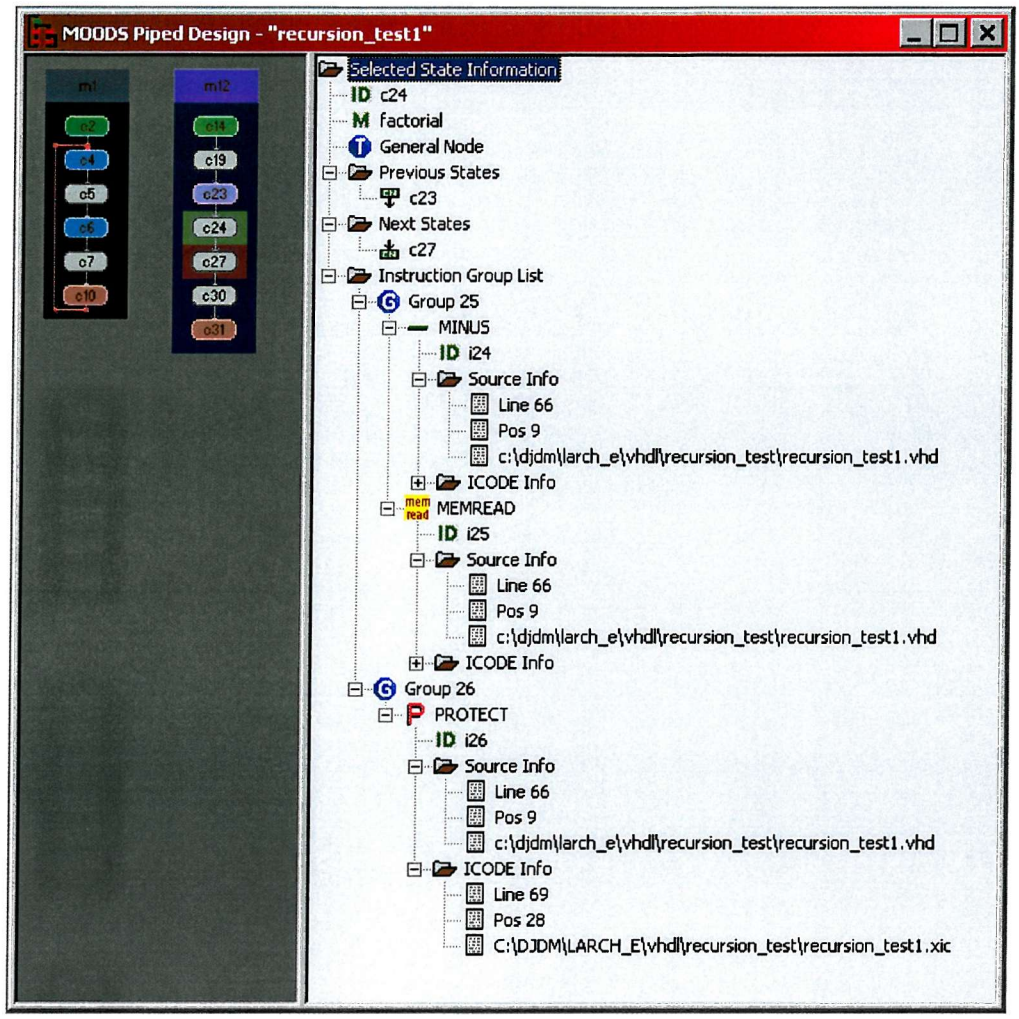


Figure A.14 Control graph and highlighted control node instructions

The figure shows two modules within the design, the left hand module 'm1' being the main program and the right hand module 'm12' a recursive procedure. Nodes 'c4' and 'c6' are calls to the recursive module 'm12' and node 'c23' is the only recursive call to the same module. Node 'c24' is highlighted with 'c27' being shown as the only node activated by 'c24'.

The right hand view pane displays information about the highlighted node 'c24'. It lists the next and previous nodes activated through the control arcs and lists the instructions that are active within the control node. These are 'i24' and 'i25' within group 25 and 'i26' within group 26. Note the link back into the original source file and the ICODE file for every instruction.

A.5.6 Linking DDF objects

The original purpose of the DDFLink program was to link together multiple representations of designs produced by MOODS stored in DDF format into a single representation and outputted in a structural VHDL form. The base for this procedure has been formed with all the file handling and VHDL generation sections completed. The only task now is to link multiple input files in some way and output the new results.

A number of items are required in order to fully link multiple designs together. These items are noted from the experiences of linking MOODS designs together manually using a structural VHDL top-level file representation that contains links to the underlying MOODS designs as components within the top-level file.

Simply creating a netlist representation of the pin linkages between MOODS designs is not quite enough; although this would form the input method of describing what subsystems are linked to other subsystems. Because each subsystem can be driven from a different clock, buffers are required between each system that synchronise the transmission signals into the input clock rate of each design. This means that asynchronous subsystems require double-buffered inputs and systems using clocks derived from each other can use a single input buffer to remove the unknown timing element on inputs.

The user makes the decision about clock speeds for the subsystems, but as the initial source has no explicit reference to the clock inputs of a design, another method of input is required. This may be performed by synthesis directives or by external clock selection. The linker may then generate clock dividers between systems automatically.

Another useful extra component used for interfacing to the outside world is the tri-state buffer. This is heavily used within memory controllers and any design that requires access to a shared bus. The control of the tri-state direction is made via another output from the user's design.

A.6 3D graphics

The beginning of the PhD was spent researching into the general area of three-dimensional graphics [116]. The general research topic was within the area of 3D graphics primitive rendering. More specifically, the rendering of 3D primitives is best achieved using some form of hardware acceleration. In particular, research into the possibility of parallelisation techniques using existing 3D rendering accelerators, such as the Voodoo Graphics chipset produced by 3DFX (a company based in California) was undertaken. This was partly due to the vastly decreased cost of commercial rendering chipsets, which had occurred because of the gaming industry [117], and the availability of high performance PCs.

Historically, the initial use of 3D graphics was mainly with flight simulation. The 3D graphics subject area was and still is booming with interest, due to the attraction of submersive 3D games. This means that many people worldwide are researching into the whole subject area.

A.6.1 Hierarchical rendering engine

While researching into the 3D graphics area, a demonstration rendering-engine was produced. This software program uses a few ideas from the 3D graphics area to create a rendering system that takes a hierarchical description of three-dimensional objects within a world and displays them from any position and direction. The program utilises an underlying hardware accelerated triangle renderer from 3DFX.

The following will outline some of the software techniques used. The test program renders a virtual world, which is defined in a hierarchical manner, using an acyclic graph of object nodes and node transformations. The language used to store the full hierarchy is also discussed.

A.6.1.1 Composition

The basic rendering unit used within the hardware accelerator is the triangle with texture mapping and linear colouration changes. Objects can be composed from the base triangle

primitive. The reason for the use of triangles as the basic drawing primitive is that it allows a fast rendering algorithm with the benefit that the triangle primitive is planar.

For instance, a rectangle can be composed of two triangles, and a cube can be built from six squares, and hence twelve triangles. These are very simple objects, but serve as demonstration.

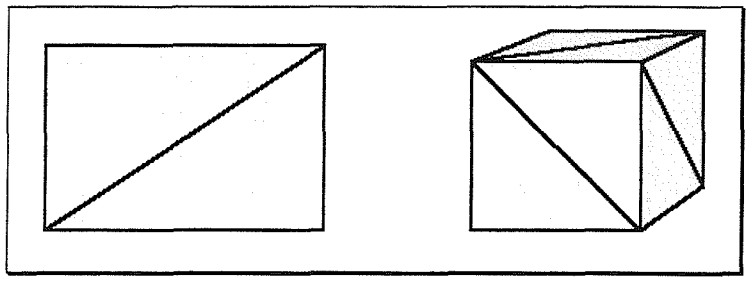


Figure A.15 Basic primitive composition

For some objects, such as curved surfaces, the composition using triangles will result in an approximation to the real object.

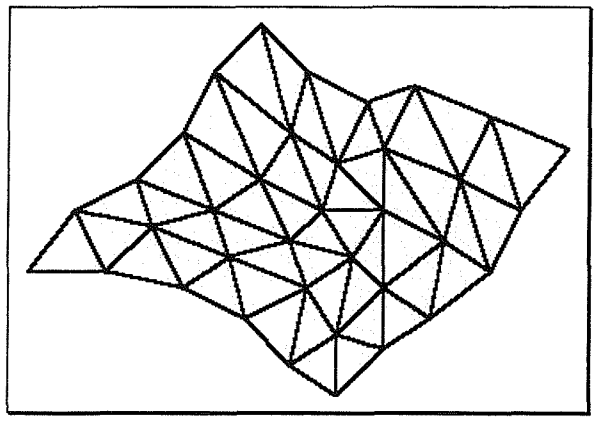


Figure A.16 Curved primitive composition from approximation

The level of detail used to approximate the real object is in a trade-off between rendering speed, as it takes longer to render a greater number of triangles due to the extra calculations involved with dimensional transformation [118].

A.6.1.2 Frustrum

The viewing frustrum defines the volume of a virtual world that is visible within the 2D representation of the world. In the case of a perspective view on the world, it is composed of six intersecting planes, where any object contained within the conical box shaped

volume is displayed. The diagram below shows the frustrum within world space and can have an arbitrary orientation.

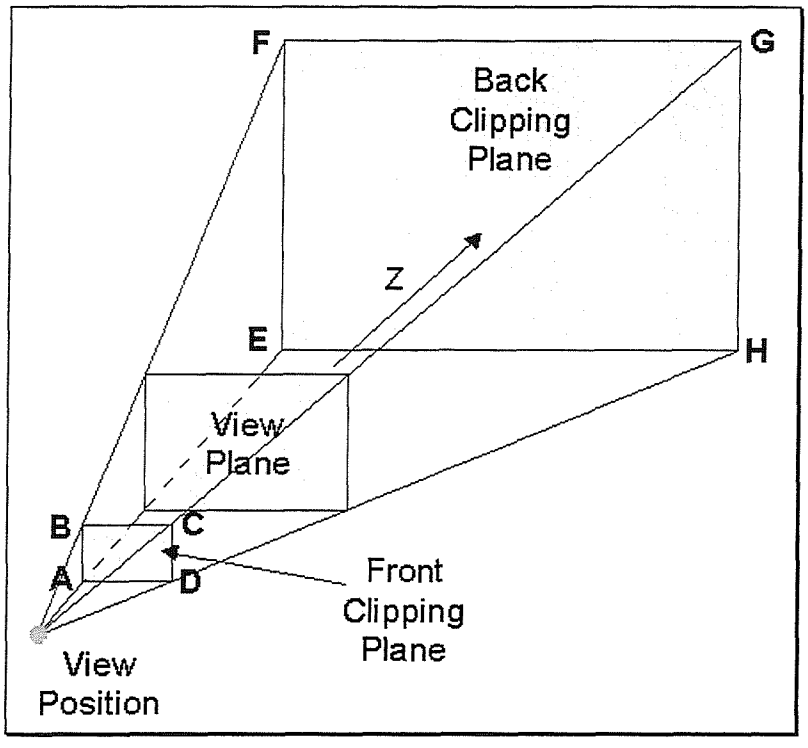


Figure A.17 Frustrum for perspective views

The six planes intersect at the eight points, A to H within Figure A.17. These planes are called the clipping planes as any object which intersects them has both a part within the viewing volume and a part outside of the viewing volume, and hence the object requires clipping before it is drawn. The six planes are defined within Table A.14.

Plane	Defining points	*Plane*	Defining points
Front	ABCD	***Right***	DCGH
Back	EFGH	***Top***	BFGC
Left	EFBA	***Bottom***	AEHD

Table A.14 Frustrum view plane definitions

The viewplane as shown in Figure A.17 is a representation of the 2D screen used to view the 3D volume. It defines the plane relative to the viewing position where any object at the same distance as the viewing plane from the viewing position will be rendered with no change in scale due to the perspective transformation, which is introduced in Section A.6.1.6.

A.6.1.3 Hierarchical objects

In the representation of a virtual world, the world is made from a graph of object nodes, with the transition from each node to another representing a relative node transformation.

An object node holds a list of primitive objects (triangles) to render, a list of transformations of other nodes and a bounding sphere. The bounding sphere is explained more in the next section, but it is simply a minimal sphere that encompasses the 3D space taken up by the list of triangle primitives and the list of transformed child nodes.

A node transformation consists of a change in relative position, scaling and orientation of child nodes from a parent node. A 4 by 4 multiplication matrix can represent this transformation.

The node-transform structure of the graph allows recursive loops to be formed between the nodes. If any loops exist, then the rendering pipeline would recurse forever. The program has set a limit to the depth of the graph arbitrarily to be 100 nodes deep. Any child nodes after this depth will not be drawn. This means that a loop within the graph could exist, but it is recommended that the designer of the world does not use loops, as the highly recursive nature of the program could result in exponential decrease in rendering performance.

The graph structure allows multiple instances of a node object to exist. This allows great possibilities for object re-use. For example, in Figure A.18 the top node of a 'Car' may have four pointers of transformations to a 'Wheel' node. The transformations of the four pointers each give a relative position, scale and orientation of the 'Wheel' node object from the parent car object node. Another transformation pointer within the 'Car' node may give information about the rest of the car body.

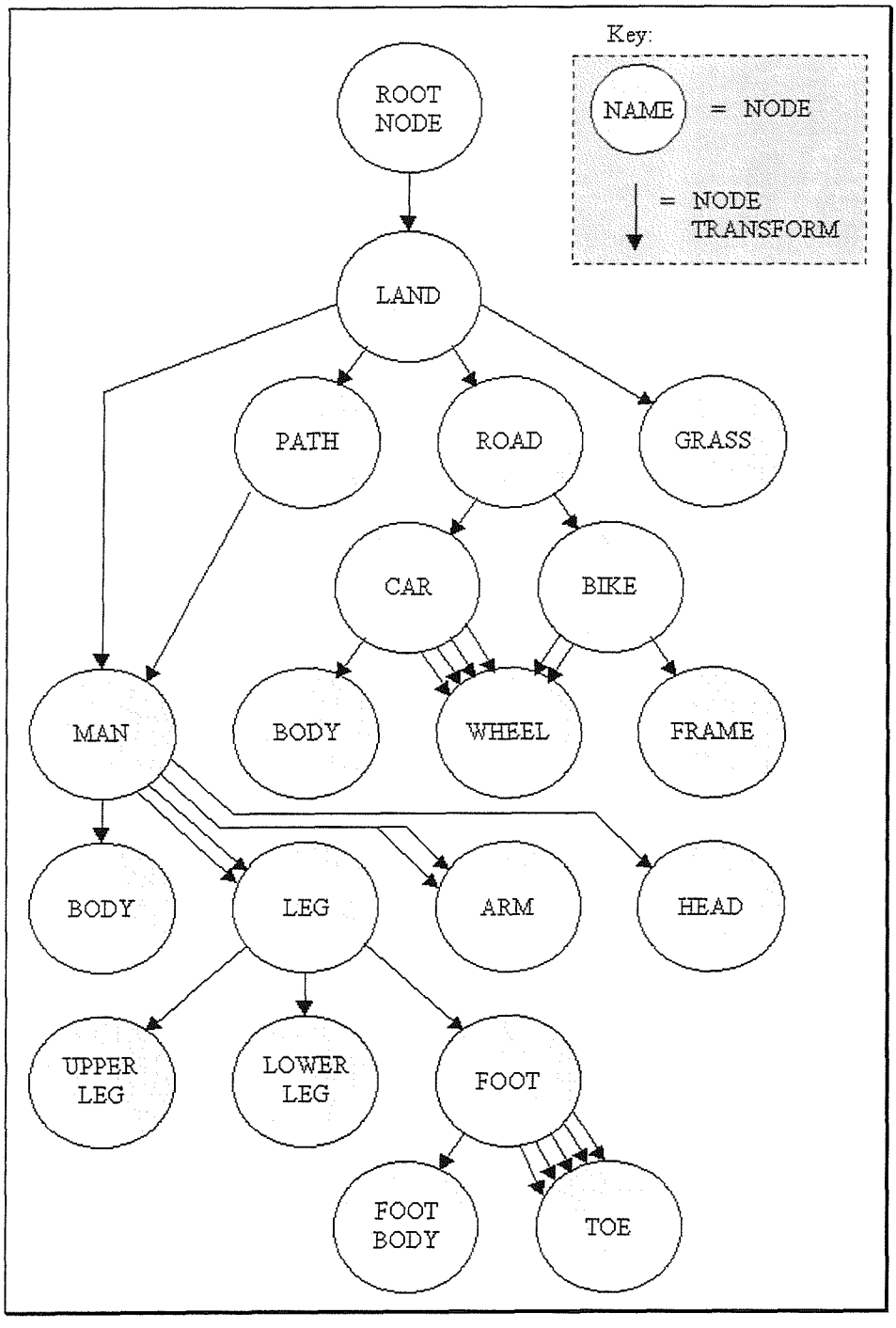


Figure A.18 Hierarchical graph world construction

Note that a node does not have to contain any triangle primitives or any child node transformations. It is usual for the initial parent nodes of the graph to hold nothing but node transformations. The 'leaves' of the graph hold no transformations, but may contain triangle primitives to render relative to the transformation path starting from the root node.

A.6.1.4 Bounding spheres

The application of bounding spheres is in speeding up the rendering of a scene by performing global culls of large sections of the world graph. Any volume can be used for this selective cull, but a sphere is an object that requires only one transformation per level of hierarchy within the object node graph.

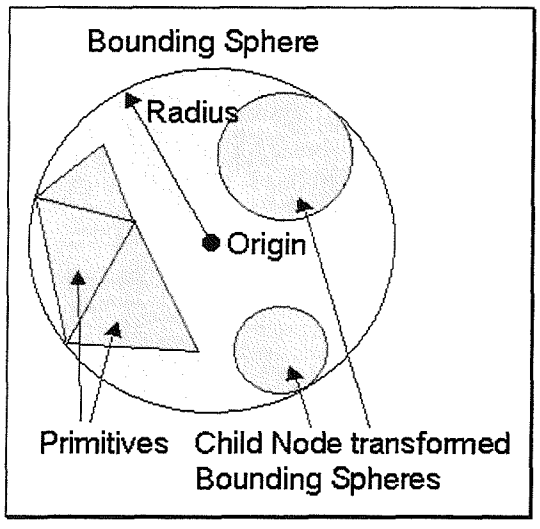


Figure A.19 Bounding sphere definition

The sphere is linked to a single node of the object graph. Its volume contains the entire list of objects and sub-objects of those objects and the drawing primitives themselves.

A calculation is made when the graph is first implemented that determines the minimum volume of the sphere that holds the entire sub-branch of all child objects and primitives.

The sphere is used when recursively following the world object graph when rendering a scene. A check is made upon the sphere against the viewing frustrum. If the spheres volume is found to be completely outside of the viewing frustrum, then none of the objects within the sphere need be drawn, so the entire branch of the object graph can be omitted from the rendering system.

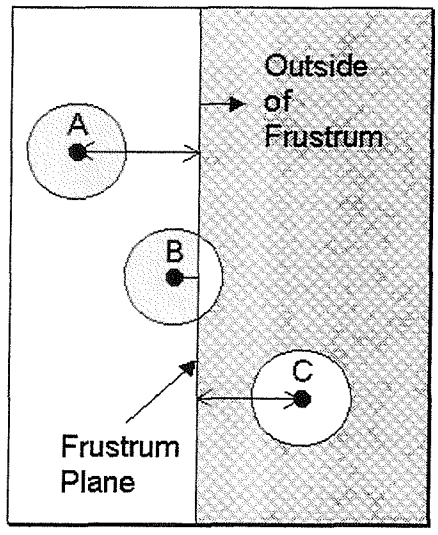


Figure A.20 Bounding sphere check against frustrum

Another alternative occurs when the sphere is completely contained within the viewing frustrum. In this case, no further clipping calculations of child objects need be applied, as they are guaranteed to be completely contained within the frustrum, as the sphere that contains them is completely contained.

If the sphere is intersected by one of the planes of the viewing frustrum then all child objects require further tests for frustrum intersection individually. The reason these tests are required is due to the necessity of clipping 3D objects.

A.6.1.5 Clipping

When an object node contains drawing primitives (triangles) that could intersect with the viewing frustrum, then tests need to be made upon each primitive in order for individual clipping to occur.

Clipping is required so that primitives are not drawn out of the visible screen area that the world is being viewed from. There are four types of intersections of a triangle with a view frustrum illustrated in Figure A.21.

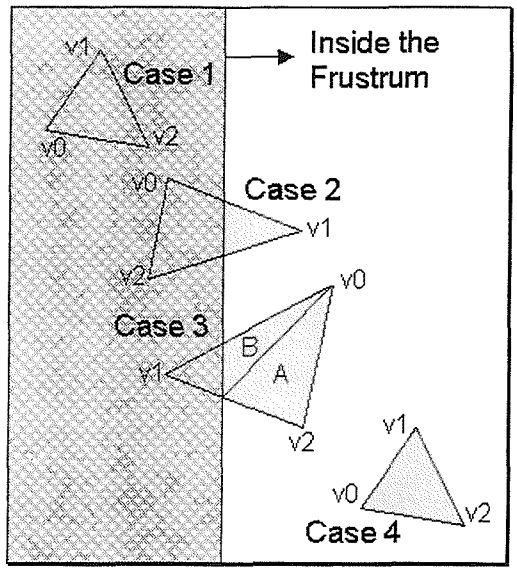


Figure A.21 Primitive intersections with frustrum resulting in clipping

The first case occurs when all three vertices are outside of the frustrum. This means that the primitive triangle does not require drawing. The second case occurs when one vertex is contained within the frustrum, but the other two are not. This means that the two intersection points need calculating and a single triangle drawn using the two new points and the single point contained within the frustrum. The third case occurs when two vertices are contained within the frustrum and the third is not. Similarly to the second case, the two new intersection points are calculated. The resulting object has four vertices however, so requires splitting into two triangle fragments for the underlying hardware renderer. The fourth case is when all three vertices are completely contained within the frustrum. No clipping is required in this case and the triangle is drawn directly.

Note that the test for intersection with every plane of the frustrum is required, which could result in a number of fragments in extreme cases of triangles intersecting more than one plane.

A.6.1.6 Depth transformation

After the objects have been rotated, translated, scaled and clipped into the view frustrum volume, the next stage is to perform a transformation that gives some depth queuing information into the resultant image. This transformation is the act of transforming the view frustrum conical volume into a volume with parallel planes.

This transformation effectively scales the x and y position of a point using the inverse z position of the same point. This means that the closer an object is to the view position, the further the transformation will move the x and y positions of the point. This gives a perspective feel to the resultant image.

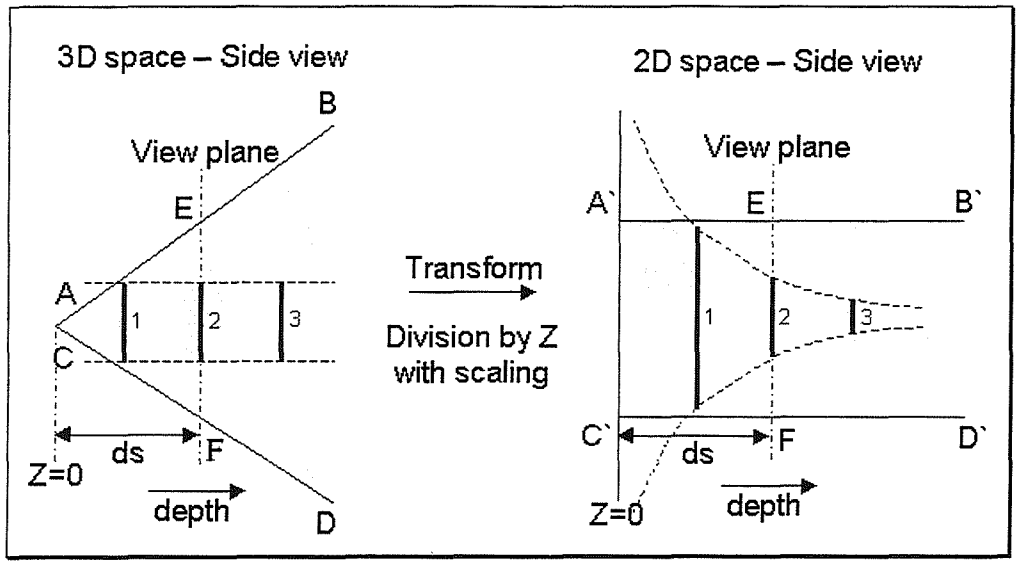


Figure A.22 Depth transformation

After the transformation, the z position is no longer used as part of the positional information of the point, so the transformation performs a 3D to 2D screen transformation with respect to the frustrum. This can be seen in Figure A.22. The three lines in the diagram show their transformed sizes in the 2D space. Note that infinite scaling occurs at the viewing position. That is why the front clipping plane is needed, so not to produce any division by zero calculations within this transformation.

A.6.1.7 Rendering pipeline

All of the features discussed so far form part of the rendering pipeline. This is the path that is taken to draw the entire scene derived from the graph of hierarchical objects. The pipeline starts from the root node of the world graph and works through the entire graph making decisions dependent on the current viewing position, direction and angle (forming the frustrum). The program flow is seen in Figure A.23.

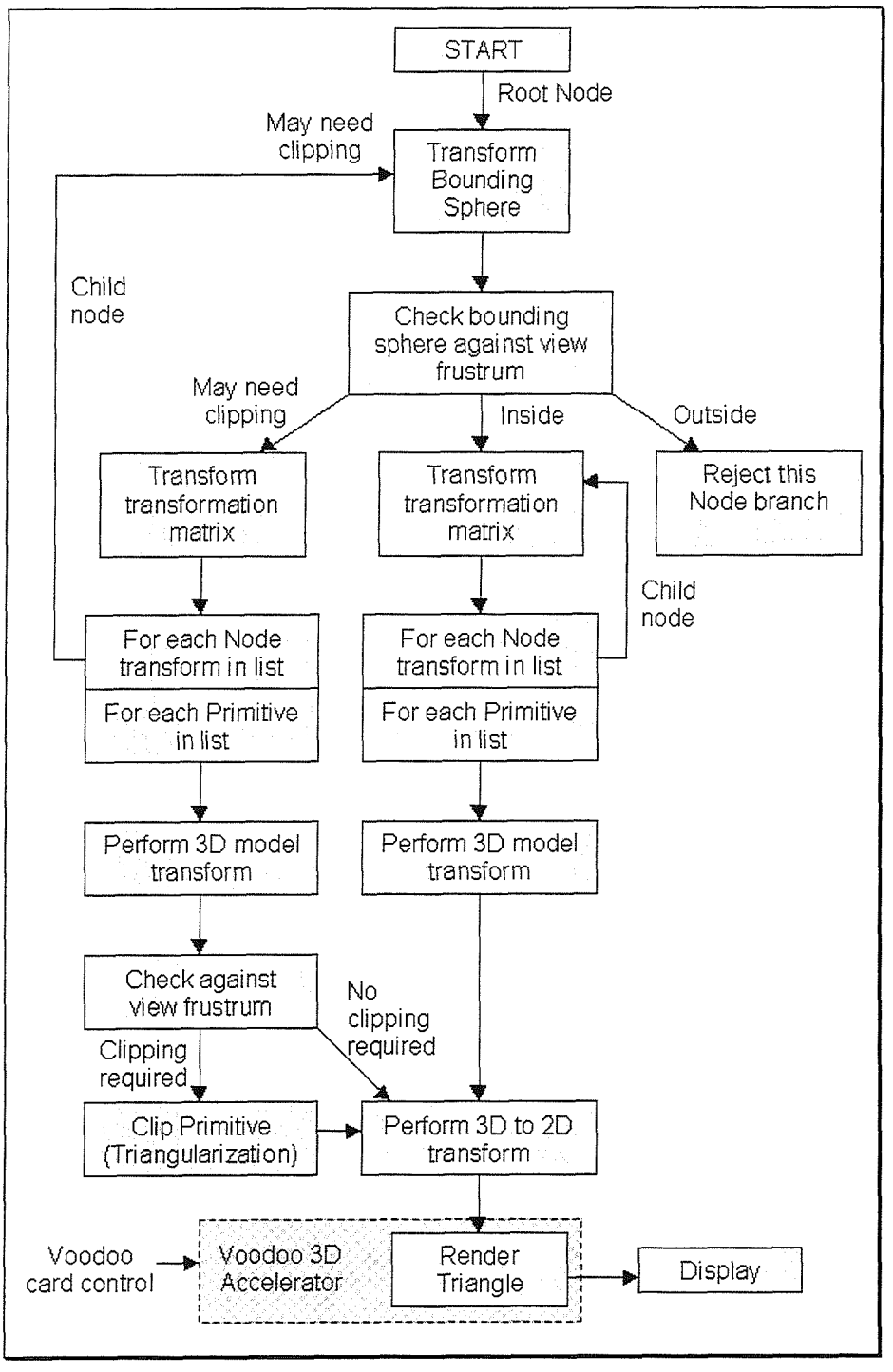


Figure A.23 Rendering pipeline

A.6.1.8 Hierarchical language

The language used to describe the hierarchical world is written in plain ASCII text. It uses integer IDs to distinguish between graph node objects. A graph node also contains a name for information only. The root node is made distinct by having a 'ROOT' item within the node definition. Each node can also contain any number of triangles and transformations of child nodes, referenced by their IDs.

The user walks around the world using a vehicle, which can have an associated object graph of its own, so that multiple users could recognise each other within the generated environment.

Limited animations are allowed within the transformation that are referenced by animation IDs of the 'ANIMATE' object. Animations in scale and rotation are given.

A number of lighting objects can also be created that have global effect upon the world. These have their own descriptors. All of these features can be seen within the example description below.

```
 1  NODE
 2   ID 5
 3   NAME road_bend
 4   TRIANGLE
 5    V0=[-5.000 0.000 5.000]
 6    V1=[5.000 0.000 5.000]
 7    V2=[5.000 0.000 -5.000]
 8    C0=[255.0 255.0 255.0 255.0]
 9    C1=[255.0 255.0 255.0 255.0]
10    C2=[255.0 255.0 255.0 255.0]
11    TEXTURE road3.3df
12    T0=[0.0 256.0]
13    T1=[0.0 0.0]
14    T2=[256.0 0.0]
15   TRIANGLE
16    V0=[-5.000 0.000 5.000]
17    V1=[5.000 0.000 -5.000]
18    V2=[-5.000 0.000 -5.000]
19    C0=[255.0 255.0 255.0 255.0]
20    C1=[255.0 255.0 255.0 255.0]
21    C2=[255.0 255.0 255.0 255.0]
22    TEXTURE road3.3df
23    T0=[0.0 256.0]
24    T1=[256.0 0.0]
25    T2=[256.0 256.0]
26   TRANSFORM
27    NODE 17
28    ANIMATE NONE
29    POSITION=[3.200 0.000 -3.200]
30    RX      =[-0.707 0.000 -0.707]
31    RY      =[0.000 1.000 0.000]
32    RZ      =[0.707 0.000 -0.707]
33    SCALE   =[1.000 1.000 1.000]
34
35  NODE
36   ROOT
37   ID 0
38   NAME RootNode
39   TRANSFORM
40    NODE 5
41    ANIMATE 0
42    POSITION=[0.000 0.000 0.000]
43    RX      =[1.000 0.000 0.000]
44    RY      =[0.000 1.000 0.000]
45    RZ      =[0.000 0.000 1.000]
46    SCALE   =[1.000 1.000 1.000]
47
48  VEHICLE
49   VIEW
```

```
50     MOVE
51     POSITION=[0.000 1.000 0.000]
52     N       =[0.000 0.000 1.000]
53     VUP     =[0.000 1.000 0.000]
54     SPEED 0.000
55     PITCH 0.000
56     ROLL  0.000
57     YAW   0.000
58     NODE 4
59
60    ANIMATE
61     ID 0
62     NAME Scale
63     ACTIVE TRUE
64     PITCH 0.000
65     ROLL  0.000
66     YAW   0.000
67     SCALEPEAK=      [0.400 0.400 0.400]
68     SCALEANGLE=     [0.452 2.713 2.973]
69     SCALEANGLERATE=[0.017 0.020 0.052]
70
71    LIGHT
72     ID 0
73     NAME main
74     DIST   50.0
75     ORIGIN =[0.000 0.000 0.000]
76     RX     =[1.000 0.000 0.000]
77     RY     =[0.000 1.000 0.000]
78     RZ     =[0.000 0.000 1.000]
79     ANIMATE=[0.320 0.000 0.200]
80     COLOUR =[255.0 255.0 220.0 220.0]
```

A.6.1.9 Summary

These pages have given a basic introduction to the fundamental viewing frustrum which is used within most 3D applications, along with the depth transform which gives the one point perspective appearance. Various aspects of the rendering pipeline within the software have been introduced, which gives a relatively efficient method of world database parsing.

Some hardware acceleration considerations have been taken into account in the designing of the rendering pipeline, such as with the primitive definition of the triangle and the clipping of these primitives with a method known as 'Triangularization'.

A.6.2 Results

The software rendering-pipeline has been implemented within the test program. It is written using C++. Some screenshots generated from this program can be seen in Figure A.24 to Figure A.28, which visually show the hierarchical structure of the virtual worlds created, with repetition of child objects forming more complex parent structures.

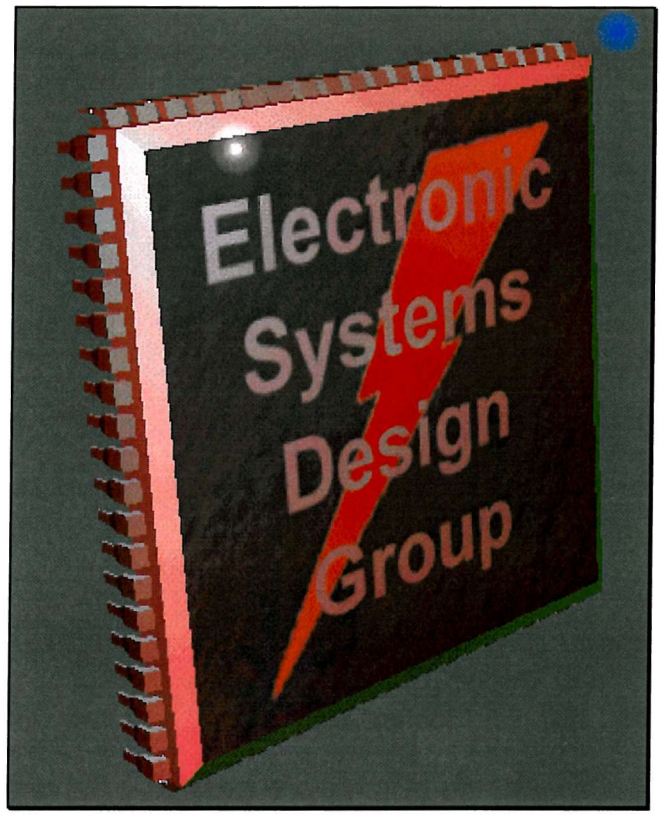


Figure A.24 A potential group logo

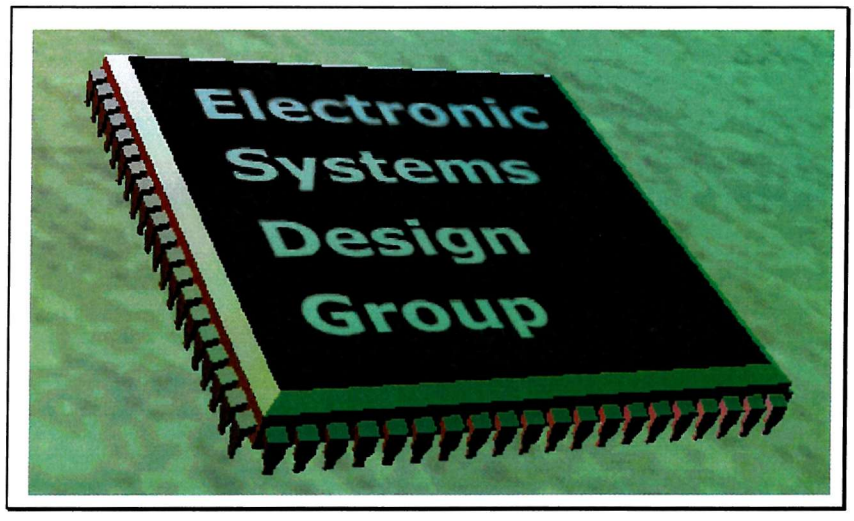


Figure A.25 A second potential group logo

Figure A.26 A street scene

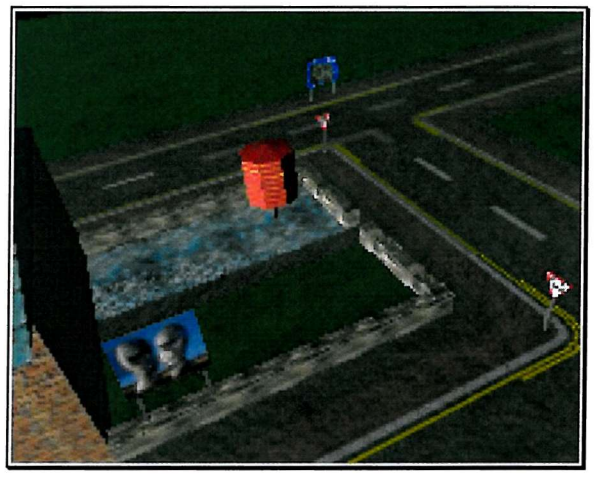


Figure A.27 Another street scene from a different angle

Figure A.28 A wide angled view of the street scene with fog, light and lens flare

Appendix B

Paper

This appendix contains the paper given at the Forum on Design Languages (FDL) conference 2000.

Dynamic memory allocation in a VHDL behavioural synthesis system

Daniel Milton
Southampton University, UK
djdm97r@ecs.soton.ac.uk

Andrew Brown
Southampton University, UK
adb@ecs.soton.ac.uk

Alan Williams
Southampton University, UK
acw@ecs.soton.ac.uk

Abstract

VHDL is capable of describing the dynamic allocation of memory resources at 'run-time'. This paper describes how this concept may be supported in a hardware synthesis environment. This requires a heap management system to be synthesised and implicitly accessed from within any user code, supporting the use of the VHDL access type. A method for controlling the storage of dynamic information (the heap manager) is reviewed. Issues such as timing and fragmentation are also discussed. An example of a design synthesised using the methods shown is reviewed last, which demonstrates the power of the technique.

1. Introduction

Memory allocation has typically been limited to use within the software domain, with no direct equivalence in the field of hardware synthesis. Hardware synthesis is migrating onto higher-level behavioural synthesis, with behavioural descriptions of digital designs in VHDL capable of describing the allocation of memory (variable) resources dynamically at 'run-time'. This paper describes how this capability is supported in the context of a behavioural synthesis suite, MOODS, developed at Southampton University [1].

A *truly* dynamic allocation of storage elements requires that a run-time system exists, with access to a memory resource (the heap) of a size capable of storing the maximum amount of information that the user requires. This subsystem is responsible for managing the memory resource under dynamic access from the user's behavioural design.

The low-level interface to this run-time system defines the four main accesses, namely '*allocate a block of memory*', '*de-allocate a block of memory*', '*read from a memory*

location' and '*write to a memory location*'. The VHDL compiler used during synthesis can then be modified to convert the given abstract behavioural VHDL into low-level calls to the relevant access procedures of the run-time heap management system [3].

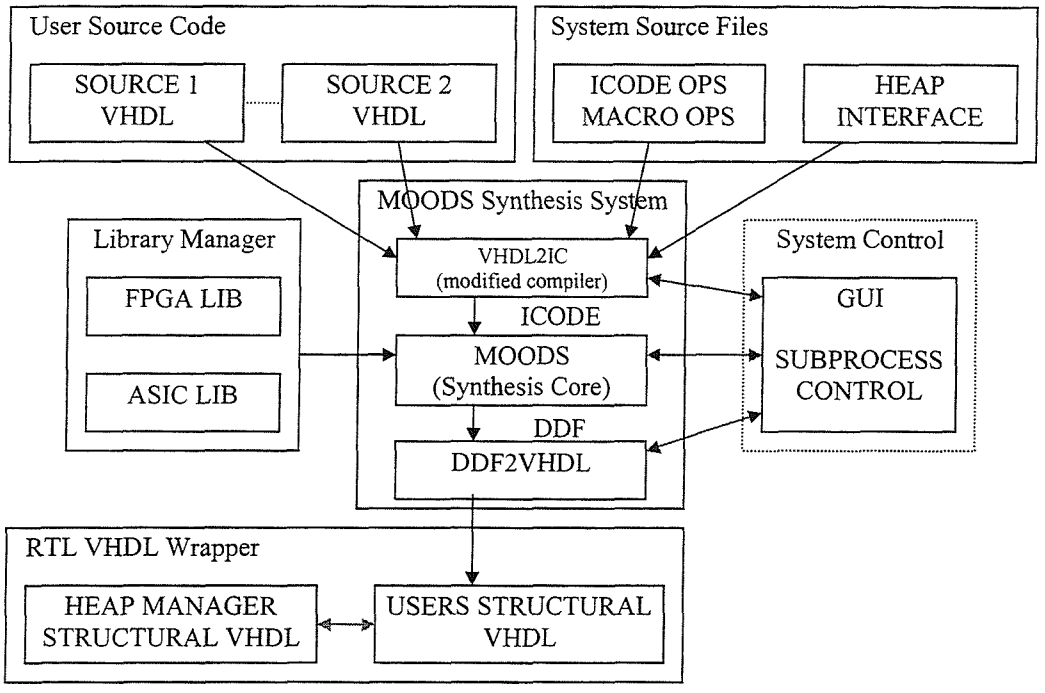


Figure 1 MOODS synthesis system overview

Figure 1 shows a system overview incorporating the subsystem necessary to support dynamic access.

2. Benefits of dynamic allocation

High-level hardware description languages allow the user a rich vocabulary of constructs to describe a system. Almost inevitably, only a subset of this HDL is synthesizable. The goal of this research (alongside all other synthesis research at Southampton) is to increase the size of the synthesizable subset, giving rise to a corresponding increase in the power of the language subset available to the user.

Using access structures allows the user to form relatively complex data structures such as linked lists, tree structures and any other structures more normally associated with software design. This is achieved with the use of an access type (memory pointer)

referencing a record type (collection of elements), with access types (optionally) contained as record elements, which may circularly reference the same record type.

3. Dynamic allocation within VHDL

The use of VHDL as the source language for synthesis puts in place various language-defined constraints when designing with dynamic storage [2,6]. The concept of the access-type (a memory 'pointer') allows a great deal of type checking to be performed, which reduces the probability of errors within the user code. VHDL does not allow type conversions involving access types, or having generic access types such as allowed within C with the use of *void** [5]. Access types must all be defined as variables, which disallows the transmission of the information stored within these types from one concurrent process to another. These limitations are all defined for valid reasons, but in practical use, tend to over-constrain the user's design.

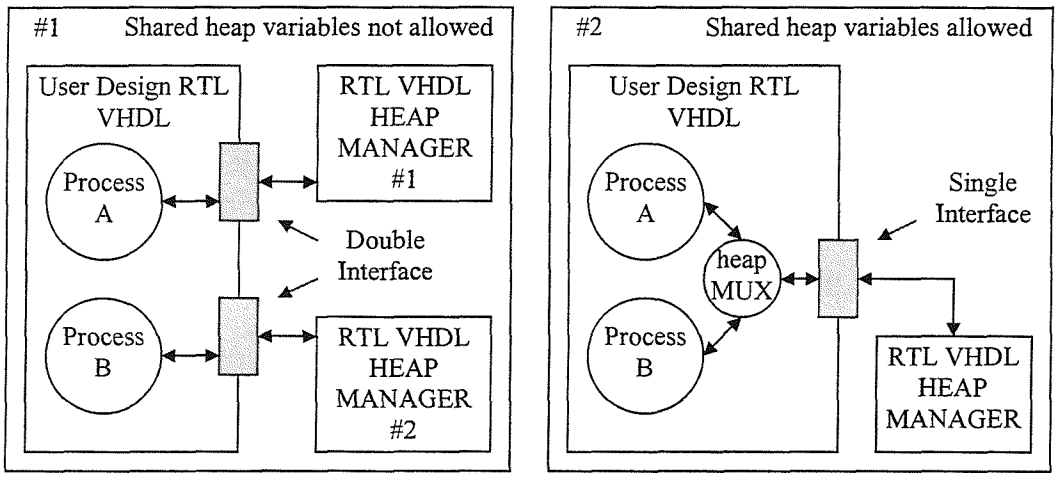


Figure 2 Concurrency support methods

One major aspect of VHDL is its in-built support for concurrency in the form of multiple processes, within which all sequential code is contained. It is perfectly possible to require access to the heap from within more than one process, so that the user can utilize the benefits of concurrency. One method for dynamic memory access in multi-process designs is to create a heap manager for each process that requires it. This would potentially allow different heap management systems to exist for each process. However, this would require extra space (silicon area) for the heap management system and the associated separate memory block. Another method for concurrent dynamic access is to share the heap

manager between the processes that access dynamic information, with access of the heap effectively serialized with the use of mutexes and semaphores and an arbitration process. This second method also allows pointers to be passed between processes (using shared variables in VHDL'93, or by designing in a different language such as System-C). Care must be taken when designing with shared variables, as their use can give differing results within different simulators. Both methods of concurrent access methods are shown in figure 2. The second method of the multiplexed heap is the one adopted in the present system.

4. Heap management

The heap management system is complex, and can form a substantial section of the final synthesized design (1039 CLBs within a Xilinx XC4062XL FPGA, 45% of the chip). However, as the compiler knows only how to interface to this system, with no dependencies on the method of allocation, this allows entirely different heap management systems to be 'plugged-in', with control over this process given to the user via various compile-time constraints. The user need not know of the complexities involved in the sub-component. Moreover, the heap manager is a fixed size overhead.

This section describes the heap management algorithm [4], used to support the method of abstract description taken by the user of the system. The algorithm is relatively simple: it is highly memory space-efficient and extremely fast. However, it has some drawbacks because of this simplicity.

4.1 The algorithm

The algorithm requires a large memory space that can be split up into a number of smaller memory spaces (pages). The size of the page determines the maximum object size that can be allocated. When the heap is active, each page is used to store objects of one size only, with the size of the objects and various list pointers stored at the head of the page. A list of all pages in use is kept within the first page in memory, which is not used for any other purpose. The active-page table and each page header form a small memory overhead. A view of the memory map created by this heap manager is shown in figure 3.

The heap starts out as being initially empty, with a list of free pages being formed by the first word within each page, and the address of the head of the list being kept within the manager. The allocation method first looks at the active-page table, which is stored within the first page of memory and determines whether any page with the required object size is currently in use with space ready for allocation. If a page exists, then the object is returned as the next free object in the active page. If the page is full after the allocation, it is removed from the active page table. If a page with the current object size is not active, then the free-page list is used to get a new page to begin allocation onto. The page header is set up with the required object size and inserted onto the active page list, and the object returned from the first available space within the page. All free-lists are generated within the main memory space, and therefore each list has zero space-overhead. The only wasted space is formed when a page is full of objects, but there exists unused space due to the page object size being too large to fit in the unused space. The active-page table uses one page, and each page has its header, which forms the rest of the space-overhead.

The heap manager is designed with a 32-bit data path. This means that all allocated objects will need to fit within 32 bits, or be split up into smaller chunks so that they do. The present compiler does not support single objects with storage requirements of greater than 32 bits. It does support arrays of objects and records of objects, where each sub-object is still limited to 32 bits. Storage inefficiencies can result from the use of objects that take less than 32 bits. A method to reduce these inefficiencies is data packing, where the data-space can be used to a greater degree by packing multiple objects into the 32-bit data space whenever possible. Various tradeoffs are involved here, involving memory size usage, the speed of access and the extra cost of synthesized hardware to perform data-path multiplexing. The present compiler does not support data packing.

4.2 Implementation

For designs specified in VHDL, both signals and variables are mapped onto physical, hardware registers. (MOODS may optimise some of these out of existence.) For both FPGA and ASIC targets, these registers will be geometrically scattered throughout the design. The dynamic memory system cannot map objects onto these static constructs, so a RAM bank has to be made available to the system. The *size* of this RAM will directly affect the internal synthesized address path widths of the user design and the heap

management system. The heap manager returning a null address, as dictated by the VHDL standard, communicates run time heap exhaustion. It is left to the designer to handle this event.

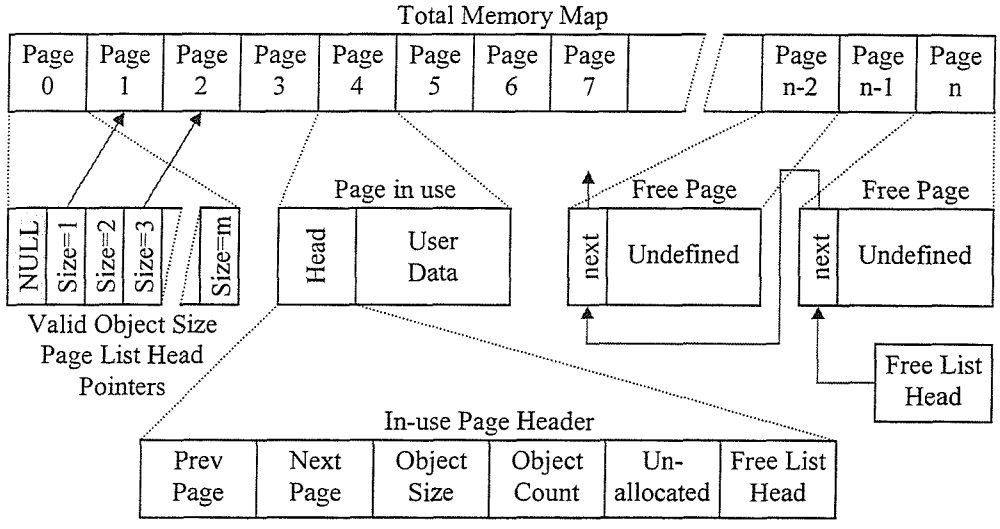


Figure 3 Heap manager memory space

5. Timing

The VHDL timing model for allocation, de-allocation and de-referencing access types (reading from or writing to the value stored on the heap) specifies that each access take zero time. However, it is impossible to meet these timing constraints within physical hardware due to the current lack of zero-latency, infinite-bandwidth memory. The behavioural synthesis paradigm embodied within MOODS allows the relaxation of timing constraints in non-timing critical code sections. This means that the usefulness of dynamic allocation of memory resources is constrained to these non-timing critical sections, at least in terms of access latency. This constraint is not as restrictive as it first may seem, and actually forces the user to use dynamic allocation only within completely behavioural (high-level, non-timing critical) code. This is not to say that the heap access bandwidth cannot be changed with the use of a different heap management sub-component.

The underlying use of DRAM forms a major timing issue, where better bandwidth-performance may be gained from utilizing the fast page mode access and by using faster DRAM or different styles of DRAM such as EDO or SDRAM. Adding a cache between

the DRAM controller and the heap manager may give increased latency-driven performance. A redesign of the heap manager to use multiple blocks of memory for different management sections, such as by using fast SRAM for the dedicated page-lookup tables would increase the performance of allocation and de-allocation. This would be very useful for designs that allocate and de-allocate for a large proportion of their running time. A method for determining the proportion of time spent using the various accesses can be found from a form of profiling, which enables the most frequently used access to be optimized the most over lesser-used accesses.

6. Memory fragmentation

Any system, hardware or software, that supports dynamic allocation and deallocation may experience fragmentation, which can dramatically reduce its efficiency and effective capacity. Careful coding can always be employed to reduce or even eliminate the problem, but the whole point of behavioural synthesis is that the designer can express him/herself in a manner sympathetic to the nature of the design, without having to worry about the implementation details.

Inevitably, some form of defragmentation support must be made available, and equally inevitably, there are tradeoffs:

- Rearranging the memory contents transparently to the parent process at run time implies some kind of memory mapping (v_table) with an associated time and space cost. (The indirection effectively halves the dataflow bandwidth of the heap.)

- The parent process itself may be delayed (locked out) while the defragmentation process is accessing the physical RAM. (The act of copying memory can also take a large proportion of time, where the latency for allocation access when de-fragmentation is required could stop the use of the heap in any code requiring a level of guaranteed timing.)

- The defragmentation controller requires silicon area.

For a given design, a strategic (i.e. human) decision needs to be taken about whether the defragmentation process is invoked manually (i.e. by the high level design) or automatically (i.e. by the heap manager when it decides it is necessary).

7. Procedural recursion

The capability to create recursive data structures is usually accompanied with the capability of using procedural recursion to generate and parse these data structures.

Behavioural synthesis disallows recursion as it creates static instances of the procedures' control mechanism (possibly inlined) and the associated local variables (data path) will generally be held in statically created registers or RAM-banks.

Procedural recursion requires a dynamic structure in the form of a stack, which holds the local variables within the procedure, and the return 'address' of the control flow.

It is planned for MOODS to support recursion, which will complement the dynamic data structure support.

8. Exemplar for the memory allocation techniques

An outline description of a physical FPGA-based exemplar for the memory allocation method described within the paper is given. The example makes use of the dynamic memory capabilities at the behavioural level and demonstrates some of the benefits of increasing the scope of the 'synthesizable subset' of VHDL.

The design consists of five processes, two of which access the same data set within the heap by the use of some shared variables for base pointers.

The object of the design is to act as an audio sequencer with a built-in audio sampler. This setup is commonly known as a 'Tracker'. The overall design uses other synthesised components such as the keyboard controller, which performs basic serial to parallel conversion and the VGA rendering system, which drives a standard 640x480 VGA screen resolution and includes a rudimentary set of hardware implemented rendering functions.

The tracker design includes two FIFO buffers to keep a constant audio stream flowing through the system. The main process receives and sends audio data from and to these buffers. It tries to keep the output buffer as full as possible and the input buffer as empty as possible, so as not to cause any under/overflow. The main control process is designed to keep the global audio bandwidth at a rate of 48Ksamples/second. The FIFO buffers take care of any latency caused by allocation / deallocation from the heap.

The audio I/O controller communicates with an external stereo; 16-bit per channel ADC/DAC chip using this chips serial interface protocol.

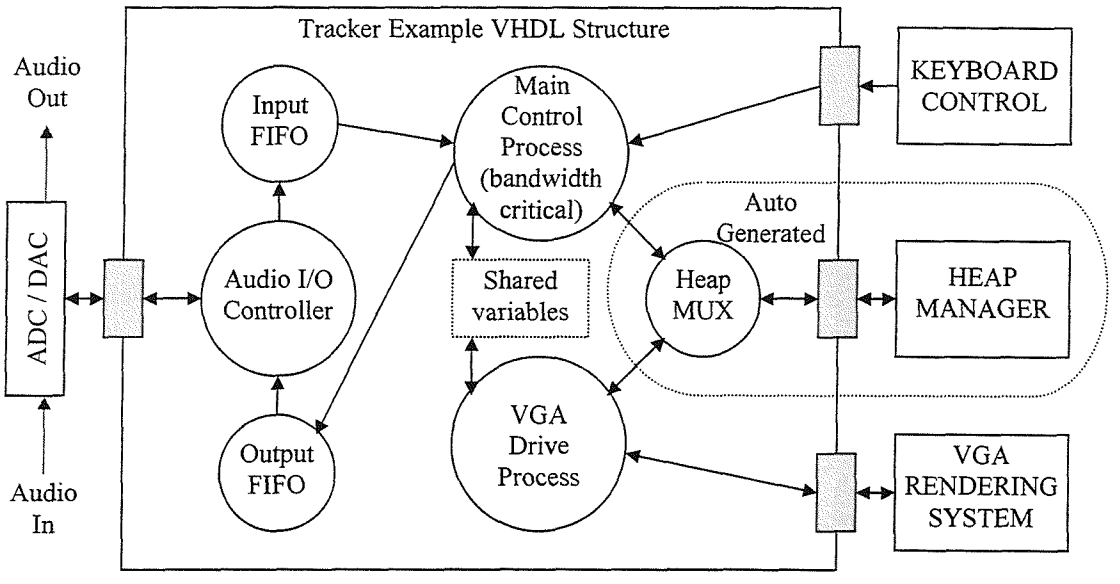


Figure 4 Physical example of an FPGA-based design using the heap manager

The design uses a completely dynamic data structure that is formed from many different types of fixed size structures. A general linked list structure is also defined, along with list object insertion, deletion and iteration procedures. The base pointers of each list which is viewed by the user in some form is created as a shared variable, so that the main process and the VGA drive (drawing) process can access the same data structures concurrently.

The system is completely under the control of the user via the keyboard controller, which directly influences the main control process.

The user is able to record a sample, via the analogue audio input. A sample is created as a sample record that is inserted into the global list of samples. The sample record contains a

pointer to a list of sample data blocks that the audio data is written into. The audio data fills up the sample block at the audio data rate. Once the block is full, a new block of data is allocated and added to the end of the sample block list, and data written into this new block. This continues until the user stops recording the sample. A single contiguous block of data cannot be allocated for the whole sample due to the unknown size of the data array at the time of allocation.

The sequencer section contains two global lists. The first list holds sequence blocks, which have a variable-time length, and themselves contain a list of sequence items, which can be placed at any time point, on any output track. These sequence items determine when particular samples are played and at what frequency they are played back (to form different notes). The user has a choice of eight output tracks on which to play a sample, which means that the system can play up to eight samples simultaneously, digitally mixed together.

The second global list is the playlist of sequence blocks, which forms a list (in playback order) of pointers to the sequence block items within the sequence block list.

The VGA process (or drawing process) has access to the base pointers of the three global lists and draws the selected items under control of the main process. Care is taken not to allow list modification while iteration of the list is occurring within the drawing process.

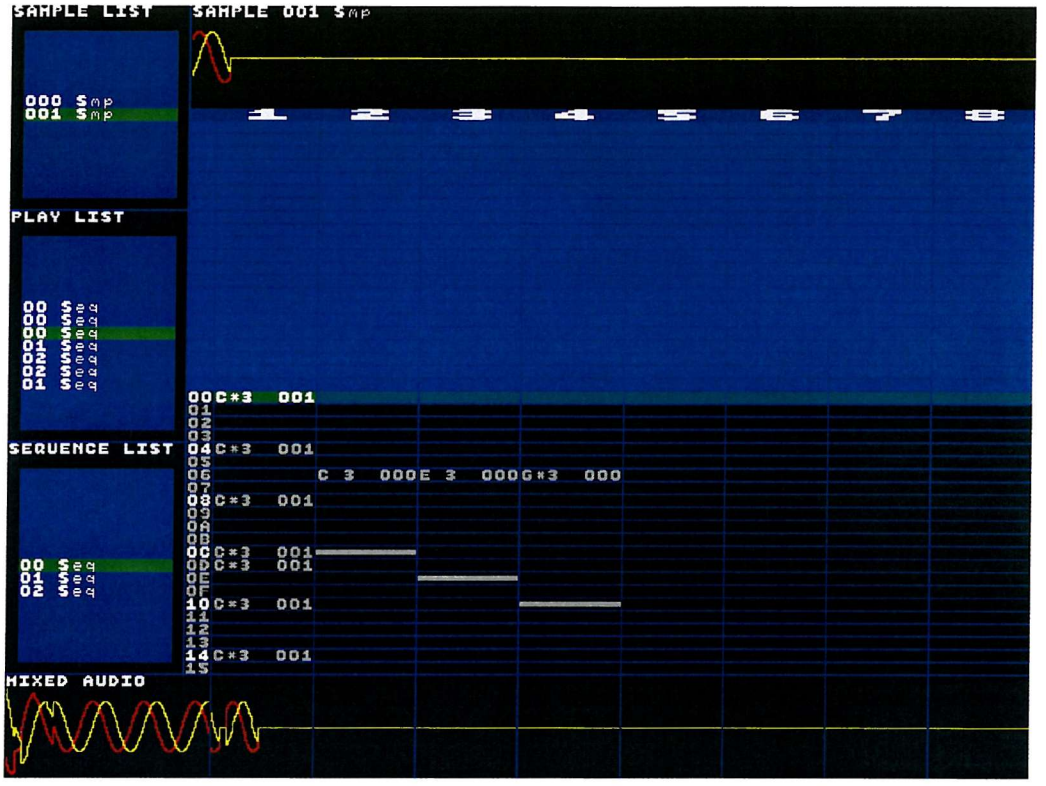


Figure 5 Tracker User Interface

This design took one month to create, simulate and physically build, which demonstrates the power of behavioural synthesis, along with the ease of use of the dynamic data types and abstract structures.

Figure 5 shows a screenshot of the user interface to the tracker in action.

Figure 6 shows a screenshot of the heap manager monitor, that shows which pages are free / partially full / full, and which pages are currently being accessed. The example heap manager uses 4 Mbytes of DRAM and splits this into 255 usable pages that each hold up to 16Kbytes (4 Kwords).

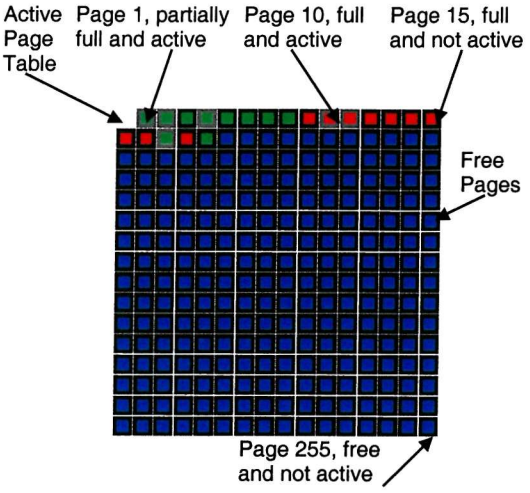


Figure 6 Heap manager monitor

9. References

[1] Williams, Alan C., "A Behavioural VHDL Synthesis System using Data Path Optimisation", PhD Thesis, University of Southampton, October 1997.

[2] Rushton, Andrew, *VHDL for Logic Synthesis*, McGraw-Hill, ISBN 0-07-709092-6.

[3] Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, ISBN 0-201-10194-7.

[4] Gontmakher, Sasha, Horn, Ilan, "Efficient Memory Allocation", *Dr. Dobb's Journal*, January 1999, pp. 116-119.

[5] Séméria, Luc, Sato, Koichi, De Micheli, Giovanni, "Resolution of Dynamic Memory Allocation and Pointers for the Behavioral Synthesis from C", DATE-Conference proceedings, March 2000, pp. 312-319.

[6] "IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS Changes), IEEE Std 1076.1 (proposal)", April 1998.

Appendix C

Demonstrators in detail

This appendix gives implementation details of a number of demonstrators that were produced to illustrate the capabilities of the MOODS synthesis system. Section C.1 describes an audio processor system that uses DRAM memory for sample storage. Section C.2 gives specific information about the general purpose PCB with the VGA output port described within Chapter 6, Section 6.1.1. The connectivity of the motherboard described within Chapter 6, Section 6.1.3.1 is given in Section C.3. The serial interface design that drives the VGA controller system within the two core demonstrators is explained within Section C.4 and the implementation of the heap management system used within the same system is described within Section C.5. The tracker demonstrator core is described within Section C.6, followed by an explanation of the expression evaluator core in Section C.7.

C.1 Echo demo

This design was developed concurrently with the VGA controller system to test the memory controller timing and PCB manufacturing software. The initial version of this design used a general-purpose wire-wrapped FPGA test-board, but migrated to a full PCB stand-alone system as it turned out to be a reasonable demonstrator of the capabilities of MOODS. It is usually referred to as 'the talking widget'.

The original design was written with the aim of producing only one effect, the effect being an echo chamber. It had only 64K of 4-bit DRAM (32Kbyte) that allowed an audio echo with a maximum period of one second. The memory was upgraded to 1Mbyte when the PCB was designed. This memory size is readily available from one 30-pin SIMM and allows a maximum echo period of 26 seconds.

The purpose of the design changed once the relatively simple echo chamber had been proven to work on the test board, as it was found that a larger design could fit within the FPGA that was currently being used. As well as the echo chamber, two further effects were added to the design. The first is a pitch-shift effect and the second is a phasing effect.

The system is very simple to use, with only three push-switches and two variable-gain potentiometers. The first variable resistor controls the analogue audio input level into the ADC. The second controls the analogue audio feedback that is fed from the DAC output and mixed with the audio input that is fed into the ADC.

The first push-button controls the mode of operation. There are four modes of operation, which are indicated by the two LEDs next to the mode button. Mode 0 (both LEDs off) indicates that the audio signal is passed directly through the digital system with minimal delay. Mode 1 (bottom LED on) indicates that the processor is acting as an echo chamber. The level of analogue feedback and the delay period determines the rate at which the echo decays. Mode 2 (top LED on) indicates that the processor is shifting the pitch of the audio input and Mode 3 (both LEDs on) indicates that the processor is producing a continuously variable delay function (phasing using analogue feedback).

The two other buttons are used for the adjustment of the effect attributes, one button for up and the other for down. Each effect has one variable attribute that allows different sounds to be produced for each effect. The variable attribute for the echo effect is the delay. The pitch shift variable is the output rate and the phasing variable is the rate of change of delay.

The design is implemented using a 6MHz clock, from which a 1MHz clock is derived. This is for use by the ADC. The sample rate of 46.9 kHz is also derived from the system clock by a binary division of 128. The ADC (ZN427E-8) and DAC (DAC0800) used are both 8-bit devices. The FPGA that forms the system designed with MOODS is the XILINX Spartan XCS10 in the PC84 package. The memory can be any 30-pin 1Mbyte SIMM.

This section details the implementation of the system, including the analogue and digital data paths, the methods that produce the three effects, the internal FPGA design methods including the memory controller and the design of the PCB for local construction.

C.1.1 Analogue data path

The design is a digital signal processing system. It has an analogue input and an analogue output with some analogue amplification, buffering and feedback. A block-diagram of the overall system is shown within Figure C.1.

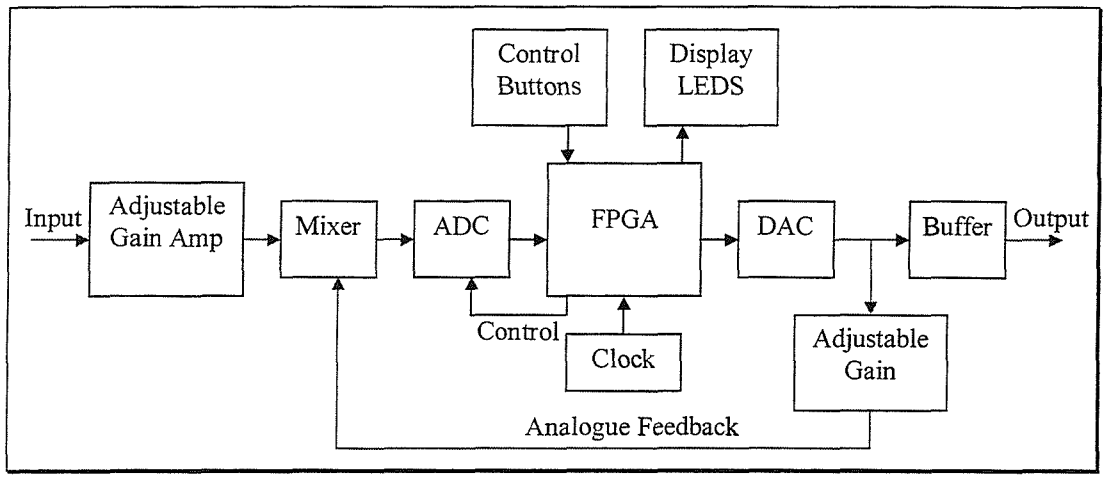


Figure C.1 Effects system dataflow diagram

The system is composed from the core digital system within the FPGA to the ancillary analogue conversion components that convert the audio signal into and from a sampled digital form. The main audio feedback path that is useful for producing the phasing effect is via the analogue path shown on the diagram.

C.1.2 Effect methods

This section describes the methods used to generate each of the effects found within the digital signal processing design.

C.1.2.1 Echo effect

Delaying the audio signal with an amount of time that can be varied produces the echo effect. The echo period can be varied from a few milliseconds to up to 26 seconds. Using the DRAM as a FIFO buffer, where the audio data is inserted at the same rate that it is removed from the memory, produces the delay. Varying the size of the FIFO buffer enables the delay to be changed.

The FIFO-buffer only needs to store one memory address that contains both the start and end points of the FIFO. For every data sample, the memory is read at the present address with this data being pushed onto the DAC, and then written to with the data just received from the ADC. The memory data has effectively been replaced by a newer sample. The address is then incremented ready for the next access. The point at which the address wraps around to zero gives the delay period.

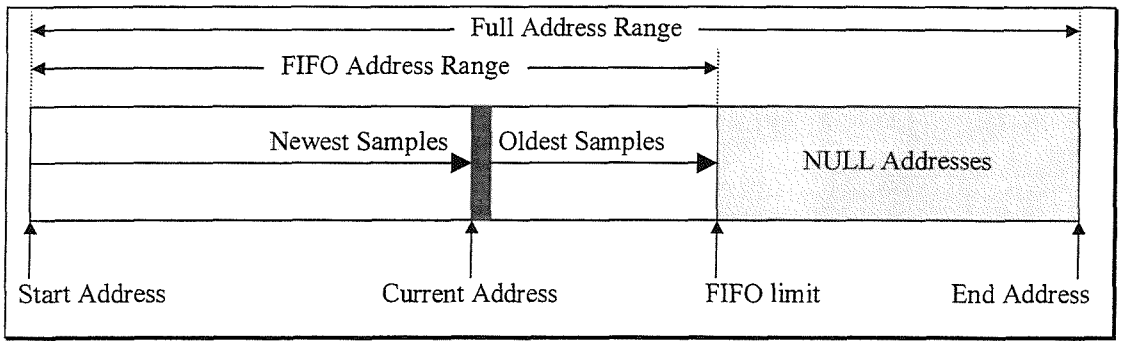


Figure C.2 Echo effect memory mapping

C.1.2.2 Pitch shift effect

The second effect is the pitch shift effect. This effect is generated in a similar manner to the echo effect in that the data is fed into a rotating buffer. However, the difference is that the buffer is of fixed length (1k) and the output is generated from a different address from the input. This effect works by sampling the input data at a fixed rate (46.9kHz) into the rotational buffer and outputting this data at a different rate. This rate is determined by a variable attribute.

The buffer always holds $1/46^{th}$ of a second worth of samples, giving a minimum frequency of 46Hz. By outputting the samples at a different rate, the pitch of the audio is changed. This method produces two unwanted but practically unnoticeable effects. When the output sample rate is faster than the input sample rate, the first anomaly occurs when the output sample position catches up with the input sample. When this happens, the next output sample will effectively be taken from $1/46^{th}$ of a second before the present output sample. This may produce a noticeable jump, or 'click' on the audio output if the start and end samples were noticeably different. The same will occur when the output rate is slower than the input rate, but this time with the jump occurring forward in time by $1/46^{th}$ of a second.

The second anomaly is due to the same procedure for the pitch shift. If the output rate is much faster than the input rate, then the sampled audio signal will be repeated on the output as the output catches up with the input and wraps around backwards in time. On the other hand, when the output rate is much slower than the input rate, some data samples will be lost. This is due to the input data position catching up with the output data position and overwriting the data that was written $1/46^{th}$ of a second ago that has not yet been outputted by the system.

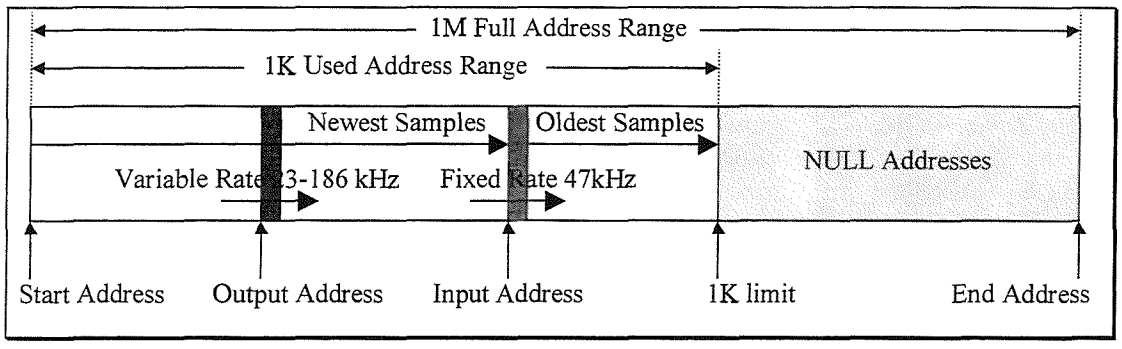


Figure C.3 Pitch shift effect memory mapping

This effect turned out to work well with a proportion of analogue feedback. It produces a phasing-like effect as well as the actual pitch shift. This effect can make speech sound like 'Pinky and Perky' down to 'Darth Vader'.

C.1.2.3 Phasing effect

This effect is generated by continuously shortening then lengthening the FIFO buffer that contains samples with a short delay period. The rate at which the buffer is expanded and contracted is adjustable to create various effects. This effect relies on the analogue feedback to create the interference patterns in the frequency domain that causes the phasing effect.

The method that produces this effect suffers from excessive 'clicking' when the buffer size is being contracted. This is due to the input and output sample rates being the same, with large jumps in time to shorten and expand the delay time. A better method would be to derive the sample rate from a similar method used in the pitch shift effect.

C.1.3 Digital design - multiple processes

As this design is very timing-critical, much use is made of multiple communicating processes. This allows a lot of parallelism and simplifies what would be complex procedures if written with a single thread of code. There are nine processes, each with their own purpose. These are explained in the further sections.

C.1.3.1 Rate process

This process simply generates two pulses every 128 clock cycles (the audio rate). One pulses 64 cycles offset from the other pulse. This is used for the read from memory sample rate generation (output timing) and the write to memory sample rate generation (input timing).

C.1.3.2 Phase shift process

The phase shift process continuously increments and decrements a value that is used for the phasing effect. This value determines the present delay of the effect. The rate at which the value is changed is determined by another value that can be edited by user input.

C.1.3.3 Button process

The button process controls the present state of the system. It forms the interface to the user. It uses the three button inputs to change various system attributes. The first button controls the present effect in operation by cycling through each mode on every button press. The button needs to be pushed and released for every change in mode. The second and third buttons have similar, but opposite effects. Depending on the present mode (which effect is in operation), these two buttons increment or decrement a single value that is used within each effect. For the direct audio pass through mode, these two buttons have no effect. For the echo mode, the two buttons increase and decrease the delay of the echo. For the pitch shift mode, these buttons adjust the relative pitch and for the phasing mode, the rate of change of phase delay is adjusted.

C.1.3.4 Debounce process

The button process to introduce a time delay that is necessary to remove signal bouncing from the button input uses the debounce process. It is also used to set an auto-repeat rate

for the continuous button pressing of the attribute change buttons within the button process.

C.1.3.5 Second rate process

This process is used in a similar way to the first rate process in that it generates a semaphore with a particular time period. However, the frequency for the second semaphore to be set is adjustable from 23 kHz to 186 kHz. This semaphore is used for the pitch bend effect as the rate at which the samples are outputted to the DAC. The value that controls the period is set within the button process.

C.1.3.6 ADC clock process

As the ADC used required a maximum clock period of 1 MHz, a division by six of the system clock was necessary to generate this frequency. This is the sole purpose of this process. Having six control states for which the ADC clock is set for three and reset for the other three performs the clock division.

C.1.3.7 ADC control process

A separate process to control the external ADC was required due to the conversion time of this device approaching the input sample rate. A pipelined approach to the data received from this input was taken, in that the present value received from the ADC is used as the input data, and when a memory write is performed with this data, a new ADC conversion is initiated ready for the next memory write.

C.1.3.8 Control process

The control process performs all the sequencing of the memory accesses, ADC reads and DAC writes dependent on the present mode of operation. It also generates the correct memory address for each access and drives the refreshing of the DRAM. It forms the basic algorithm for each of the effects that are generated.

C.1.3.9 Memory process

The memory process is the main reason for this project to be designed and built. This is the process that was initially used to test the various access modes and timing attributes of

the fast page mode DRAM. However, this incarnation only performs CAS before RAS refreshing and single-byte non-page-mode read and writes. This was all that was required from this system as memory bandwidth and latency was not such a constraint as for the VGA controller.

One major flaw that was found with the interpretation of the given timing from the memory data books was that the address setup time has a minimum of 0ns. However, this does not mean that the address can be set up at the same time as the two address strobes, /RAS and /CAS. The actual timing requires a small period between the two. This is achieved by inserting a clock cycle. This turns out not to impose much timing penalty due to the many other timing constraints that also need to be met. Along with the changes to the memory timing, the MOODS control graph was utilised as the controller state machine. With the initial version of the controller using a style of VHDL that forced MOODS to optimise an entire state machine process into one control state, the produced design was larger than necessary.

C.1.4 PCB design and production

As this design was relatively simple, a simple two-layer PCB could be used for producing the demonstration. This allowed the use of the in-house PCB manufacturing facilities that can produce boards with a minimum track width of 12mil. The same software was used to produce both this design and the VGA controller general-purpose board. Once the board was built, it was populated and tested manually. A composite picture showing the tracks and components from the layout tool is shown within Figure C.4.

One method that was used with this board was to force the routing algorithm to route on the bottom layer whenever it could and to use the top layer only when it was necessary. This meant that there were fewer vias to solder, greater testability of the final board, and only single sided soldering was necessary for every component, which meant that hand soldering was possible. A picture of the final implementation of the effects processor is shown in Figure C.5.

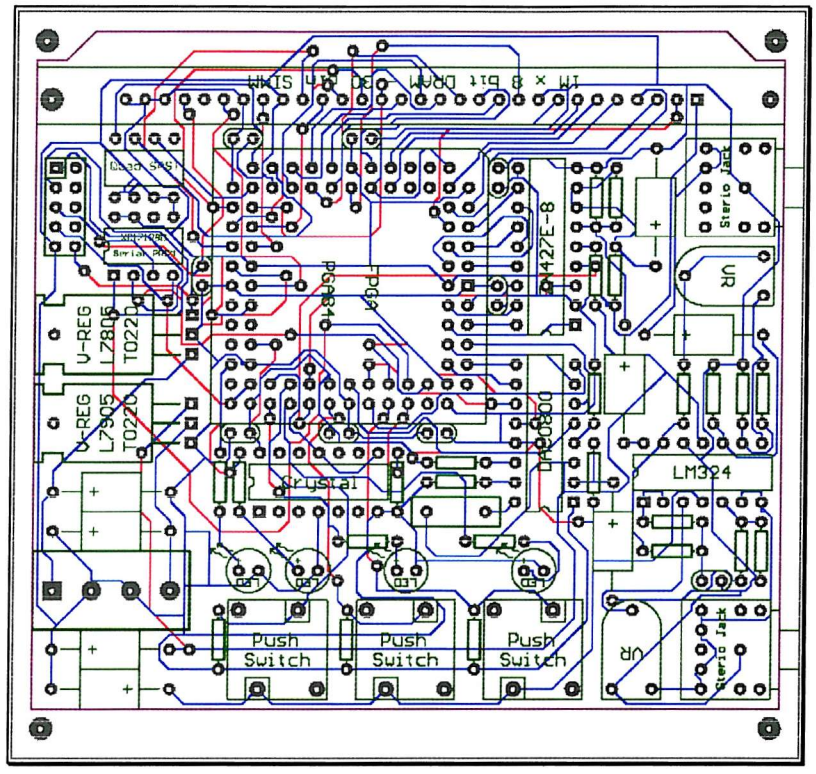


Figure C.4 Effects processor PCB track layout

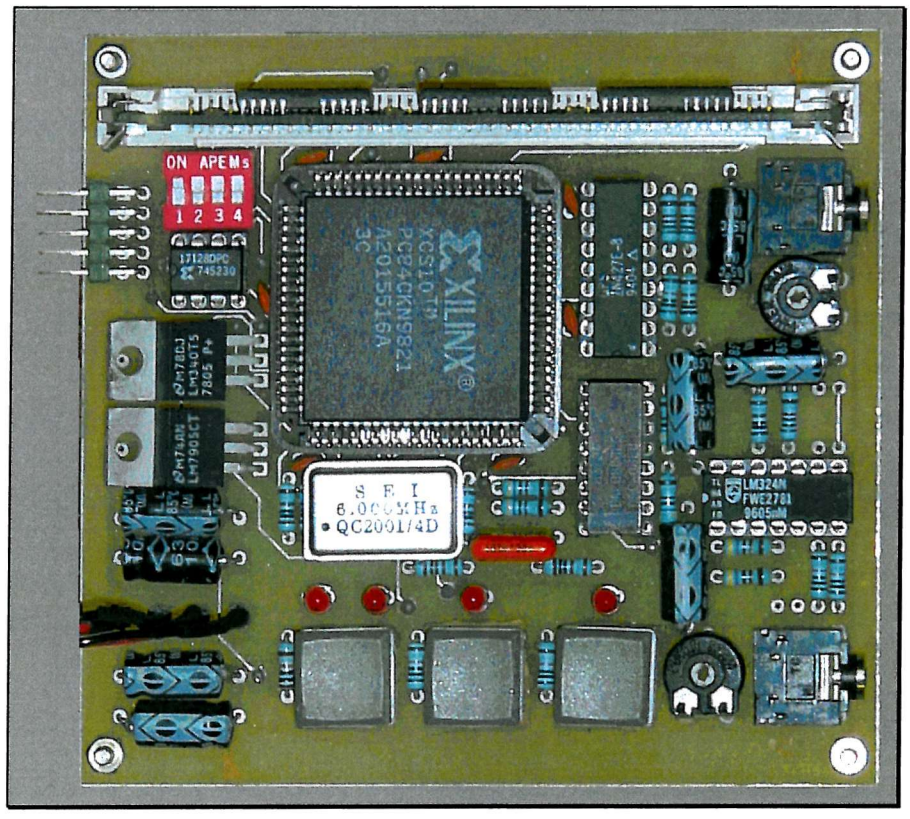


Figure C.5 Effects processor PCB picture

C.2 PCB design

The Printed Circuit Board information shown here relates to the first PCB described within Chapter 6, Section 6.1.1. The second PCB described within Section 6.1.2 was designed separately and is described in more detail within [106].

C.2.1 Programming the FPGA

The FPGA device used can be programmed in one of eight modes. An external interface to the entire programming system is contained within expansion port A, which enables any of these modes to be utilised. However, the board is designed with one of two modes in mind. The FPGA requires programming every time that it is powered up.

The first is the slave serial mode, used to program the device from a computer download cable. This mode is used during development of the digital system and is interfaced by a limited set of programming pins that can be connected via the external programming connector, shown within Figure C.6. The meanings of these pins are further explained within [104].

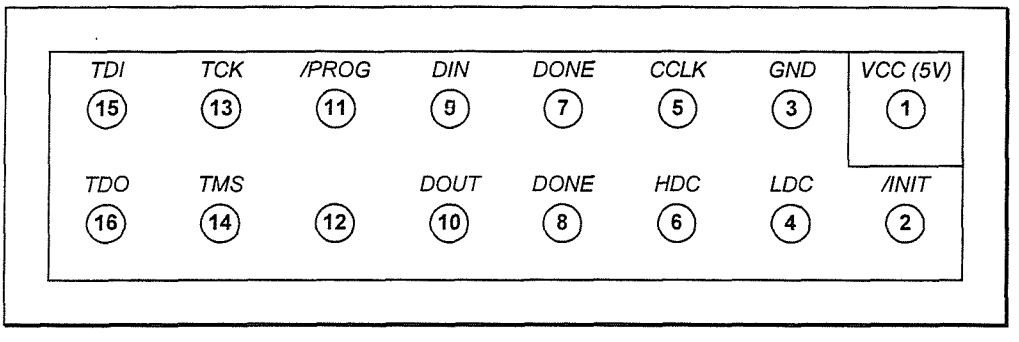


Figure C.6 External programming connector

The second supported mode is the master parallel (up) mode that is used when a design has been settled upon. An onboard ROM that is capable of storing a single configuration supports this mode.

The mode selection is made via a set of DIP-switches that pull the FPGA's mode selection pins low. The mode selection pins are also accessible from expansion port A. The DIP-switch positions for both supported modes are shown within Figure C.7. Note that only three of the four switched are connected.

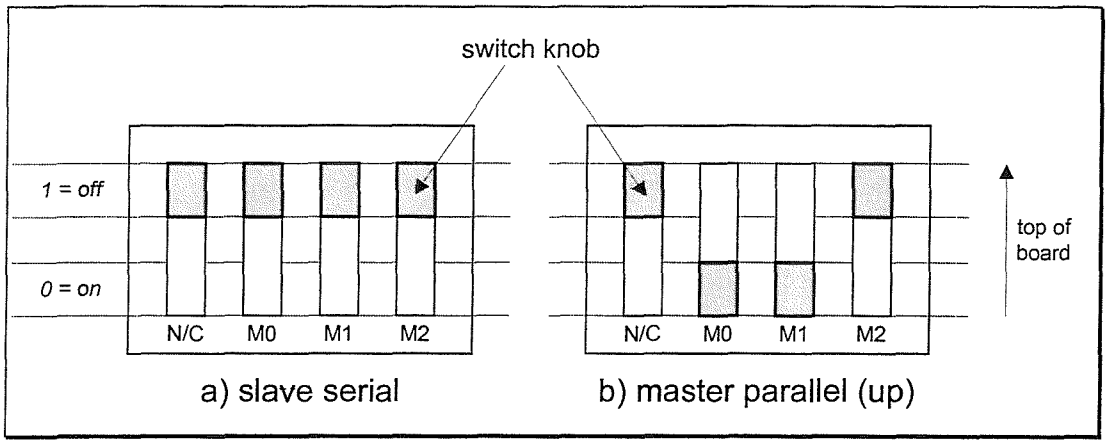


Figure C.7 Programming mode DIP switch settings

C.2.2 FPGA pin-out

The pin constraints listed within Section C.2.3 reference FPGA pad names of the PG475 package whose bottom-view layout is shown within Figure C.8.

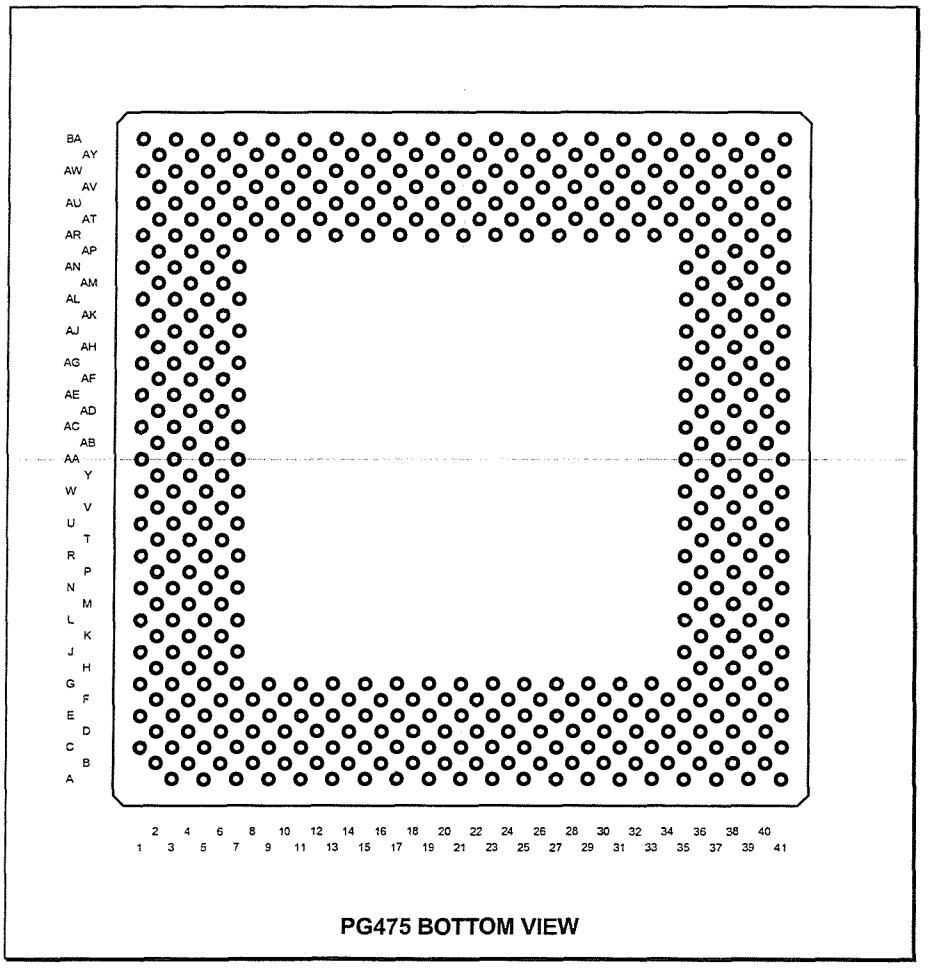


Figure C.8 FPGA package used by the PCB

C.2.3 Pin constraints

All ancillary components designed into the PCB are connected to the core FPGA. The connections between the FPGA pins and component pins are listed in the tables below. A few connections are shared between the onboard components and the external connectors. These do not restrict the use of any component on the board unless connected externally.

Clock pin	FPGA pin	Comment
Clock 1	F38	GCK2, 25 MHz, Shared with expansion port A, B11
Clock 2	J37	GCK3, Shared with expansion port A, B12

Table C.1 Clock pin constraints

Keyboard pin	FPGA pin	Comment
Clock	AN1	Common collector I/O (mainly input)
Data	AN3	Common collector I/O (mainly input)

Table C.2 Keyboard pin constraints

Mouse pin	FPGA pin	Comment
Clock	AM4	Common collector I/O (pull-down only)
Data	AN5	Common collector I/O (pull-down only)

Table C.3 Mouse pin constraints

Serial port pin	FPGA pin	Comment
R1	AP2	Serial Data Receive (RxD)
T1	AW3	Serial Data Transmit (TxD) shared with PA1 program address
R2	AR1	Clear to send (CTS)
T2	AT8	Request to send (RTS)

Table C.4 Serial port pin constraints

Text ROM pin	FPGA pin	Comment
Address 0	F24	Least significant address bit
Address 1	B24	
Address 2	D26	
Address 3	G27	
Address 4	B28	
Address 5	B32	

Text ROM pin	FPGA pin	Comment
Address 6	B34	
Address 7	C37	
Address 8	C33	
Address 9	C31	
Address 10	C29	Most significant address (11 bits = 2K)
Data 0	F22	Least significant data bit
Data 1	C21	
Data 2	D20	
Data 3	E19	
Data 4	F20	
Data 5	D22	
Data 6	G23	
Data 7	G25	Most significant data (8 bits)

Table C.5 Text ROM pin constraints

VGA & DAC pin	FPGA pin	Comment
Blanking	G9	Composite of Vertical and Horizontal blanking
VSync	C15	Vertical Sync signal
HSync	A15	Horizontal Sync signal
Blue 7	D14	Most significant bit for Blue colour (8 bits)
Blue 6	B14	
Blue 5	A13	
Blue 4	A11	
Blue 3	B12	
Blue 2	C11	
Blue 1	D10	
Blue 0	B10	Least significant bit for Blue colour
Green 7	C9	Most significant bit for Green colour (8 bits)
Green 6	G11	
Green 5	E9	
Green 4	F12	
Green 3	E11	
Green 2	E13	
Green 1	D12	
Green 0	F14	Least significant bit for Green colour
Red 7	C5	Most significant bit for Red colour (8 bits)
Red 6	A5	
Red 5	B6	

VGA & DAC pin	FPGA pin	Comment
Red 4	A7	
Red 3	E7	
Red 2	B8	
Red 1	D8	
Red 0	A9	Least significant bit for Red colour

Table C.6 Video signal pin constraints

VGA DRAM pin	FPGA pin	Comment
/RAS	BA11	Inverted Row Address Strobe
/CAS	AY10	Inverted Column Address Strobe
/WE	AW11	Inverted Write Enable signal
Address 0	AV12	Least significant address bit
Address 1	AY12	
Address 2	BA13	
Address 3	AV14	
Address 4	AY14	
Address 5	AT14	
Address 6	AW13	
Address 7	AU13	
Address 8	AT12	Most significant address, multiplexed 9 bits = 256Kword
Data 0	AT16	Least significant data bit
Data 1	AT18	
Data 2	AT24	
Data 3	AU23	
Data 4	AT2	
Data 5	AU3	
Data 6	AU9	
Data 7	AT4	
Data 8	AU17	
Data 9	AV18	
Data 10	AV20	
Data 11	AU19	
Data 12	AV2	
Data 13	AV4	
Data 14	AV10	
Data 15	AU11	
Data 16	AV16	
Data 17	AW17	

VGA DRAM pin	FPGA pin	Comment
Data 18	AW19	
Data 19	AY18	
Data 20	AW1	
Data 21	AW5	
Data 22	AW9	
Data 23	AY8	
Data 24	BA15	
Data 25	BA19	
Data 26	BA23	
Data 27	AY20	
Data 28	AY4	
Data 29	BA5	
Data 30	BA9	
Data 31	BA7	Most significant data (32 bits)

Table C.7 Frame buffer DRAM pin constraints

DRAM 0 pin	FPGA pin	Comment
/RAS	AL3	Inverted Row Address Strobe
/CAS	B4	Inverted Column Address Strobe
/WE	AG3	Inverted Write Enable signal
Address 0	H6	Least significant address bit
Address 1	J5	
Address 2	L7	
Address 3	M6	
Address 4	P4	
Address 5	R7	
Address 6	V6	
Address 7	T4	
Address 8	AB6	
Address 9	AF6	
Address 10	AC3	Most significant address, multiplexed 11 bits = 4Mword
Data 0	G5	Least significant data bit
Data 1	K6	
Data 2	N3	
Data 3	T6	
Data 4	AK6	
Data 5	AG5	
Data 6	AK4	

DRAM 0 pin	FPGA pin	Comment
Data 7	AL5	
Data 8	F4	
Data 9	K4	
Data 10	N1	
Data 11	R5	
Data 12	Y6	
Data 13	AF4	
Data 14	AJ5	
Data 15	AJ3	Most significant data (16 bits)

Table C.8 General purpose DRAM bank 0 pin constraints

DRAM 1 pin	FPGA pin	Comment
/RAS	AM2	Inverted Row Address Strobe
/CAS	C3	Inverted Column Address Strobe
/WE	AG1	Inverted Write Enable signal
Address 0	H4	Least significant address bit
Address 1	J3	
Address 2	L5	
Address 3	L3	
Address 4	N5	
Address 5	P2	
Address 6	U7	
Address 7	U3	
Address 8	AB2	
Address 9	AC5	
Address 10	AC1	Most significant address, multiplexed 11 bits = 4Mword
Data 0	F2	Least significant data bit
Data 1	K2	
Data 2	M4	
Data 3	R3	
Data 4	W5	
Data 5	AE5	
Data 6	AH4	
Data 7	AK2	
Data 8	E1	
Data 9	J1	
Data 10	M2	
Data 11	R1	

DRAM 1 pin	FPGA pin	Comment
Data 12	W3	
Data 13	AE3	
Data 14	AH2	
Data 15	AJ1	Most significant data (16 bits)

Table C.9 General purpose DRAM bank 1 pin constraints

Expansion A pin	FPGA pin	Comment
A1	AR7	PA0, /WS
A2	AW3	PA1, GCK7, shared with Serial port RS232 T1
A3	AU1	PA2, CS1
A4	AM6	PA3
A5	AD6	PA4
A6	AD4	PA5
A7	AB4	PA6
A8	AA3	PA7
A9	Y2	PA8
A10	Y4	PA9
A11	V2	PA10
A12	V4	PA11
A13	H2	PA12
A14	G1	PA13
A15	E3	PA14
A16	E5	PA15, GCK8
A17	G7	PA16, GCK1
A18	D4	PA17
A19	U5	PA18
A20	W1	PA19
A21	AC7	PA20
A22	AD2	PA21
A23	AU5	PD0, DIN
A24	AV8	PD1
A25	AW15	PD2
A26	AW21	PD3
A27	AY22	PD4
A28	BA29	PD5
A29	AV34	PD6
A30	AU35	PD7
A31	AN35	/PROG

Expansion A pin	FPGA pin	Comment
A32	AR35	DONE
B1	E35	M0
B2	A39	M1
B3	G33	M2
B4	AV6	DOUT, GCK6
B5	Y38	/INIT
B6	C41	LDC
B7	G35	HDC
B8	AR5	CCLK
B9	AY6	RDY, /BUSY, /RCLK
B10	AY28	/CS0
B11	F38	GCK2, shared with Clock 1 (25 MHz)
B12	J37	GCK3, shared with Clock 2
B13	AU39	GCK4
B14	AV38	GCK5
B15	AN7	TDO
B16	D6	TDI
B17	F8	TCK
B18	C13	TMS
B19	BA21	/RS
B20	A29	General purpose I/O
B21	D30	General purpose I/O
B22	E31	General purpose I/O
B23	E33	General purpose I/O
B24	D34	General purpose I/O
B25	B36	General purpose I/O
B26	B38	General purpose I/O
B27	C39	General purpose I/O
B28	E41	General purpose I/O
B29	G41	General purpose I/O
B30	H40	General purpose I/O
B31	J41	General purpose I/O
B32	K40	General purpose I/O
C1	D16	General purpose I/O
C2		N/C
C3	C17	General purpose I/O
C4	D18	General purpose I/O
C5	B18	General purpose I/O

Expansion A pin	FPGA pin	Comment
C6	C19	General purpose I/O
C7	A19	General purpose I/O
C8	B20	General purpose I/O
C9	A21	General purpose I/O
C10	B22	General purpose I/O
C11	E23	General purpose I/O
C12	C23	General purpose I/O
C13	A23	General purpose I/O
C14	D24	General purpose I/O
C15		N/C
C16	C25	General purpose I/O
C17	C27	General purpose I/O
C18	A27	General purpose I/O
C19	D28	General purpose I/O
C20	E29	General purpose I/O
C21	B30	General purpose I/O
C22		N/C
C23	A33	General purpose I/O
C24	A35	General purpose I/O
C25	D36	General purpose I/O
C26		N/C
C27	D40	General purpose I/O
C28	F40	General purpose I/O
C29		N/C
C30	H38	General purpose I/O
C31	J39	General purpose I/O
C32	K38	General purpose I/O

Table C.10 Expansion port A pin constraints

Expansion B pin	FPGA pin	Comment
A1	L39	General purpose I/O
A2	M40	General purpose I/O
A3	N41	General purpose I/O
A4	P40	General purpose I/O
A5	R41	General purpose I/O
A6	U39	General purpose I/O
A7	V40	General purpose I/O
A8	W41	General purpose I/O

Expansion B pin	FPGA pin	Comment
A9	Y40	General purpose I/O
A10	AB40	General purpose I/O
A11	AC41	General purpose I/O
A12	AD40	General purpose I/O
A13	AF38	General purpose I/O
A14	AG41	General purpose I/O
A15	AH40	General purpose I/O
A16	AJ41	General purpose I/O
A17	AK40	General purpose I/O
A18	AL39	General purpose I/O
A19	AM40	General purpose I/O
A20	AN41	General purpose I/O
A21	AP40	General purpose I/O
A22	AT40	General purpose I/O
A23	AU41	General purpose I/O
A24	BA39	General purpose I/O
A25	AY38	General purpose I/O
A26	BA37	General purpose I/O
A27	AY36	General purpose I/O
A28	BA35	General purpose I/O
A29	BA33	General purpose I/O
A30	AY32	General purpose I/O
A31	BA31	General purpose I/O
A32	BA27	General purpose I/O
B1	L37	General purpose I/O
B2	M38	General purpose I/O
B3	N39	General purpose I/O
B4	P38	General purpose I/O
B5	R39	General purpose I/O
B6	T38	General purpose I/O
B7	V38	General purpose I/O
B8	W39	General purpose I/O
B9	AA39	General purpose I/O
B10	AB38	General purpose I/O
B11	AC39	General purpose I/O
B12	AD38	General purpose I/O
B13	AF36	General purpose I/O
B14	AG39	General purpose I/O

Expansion B pin	FPGA pin	Comment
B15	AH38	General purpose I/O
B16	AJ39	General purpose I/O
B17	AK38	General purpose I/O
B18	AL37	General purpose I/O
B19	AN39	General purpose I/O
B20	AP38	General purpose I/O
B21	AR41	General purpose I/O
B22	AT38	General purpose I/O
B23	AW39	General purpose I/O
B24	AY34	General purpose I/O
B25	AW33	General purpose I/O
B26	AW31	General purpose I/O
B27	AY30	General purpose I/O
B28	AW29	General purpose I/O
B29	AW27	General purpose I/O
B30	AW25	General purpose I/O
B31	AY24	General purpose I/O
B32	AW23	General purpose I/O
C1		GND
C2		Unregulated power supply
C3	M36	General purpose I/O
C4	N37	General purpose I/O
C5	R37	General purpose I/O
C6	U37	General purpose I/O
C7	V36	General purpose I/O
C8	W37	General purpose I/O
C9	AB36	General purpose I/O
C10	AC37	General purpose I/O
C11	AD36	General purpose I/O
C12	AE37	General purpose I/O
C13	AE39	General purpose I/O
C14	AG37	General purpose I/O
C15	AJ37	General purpose I/O
C16	AK36	General purpose I/O
C17	AM36	General purpose I/O
C18	AM38	General purpose I/O
C19	AN37	General purpose I/O
C20	AP36	General purpose I/O

Expansion B pin	FPGA pin	Comment
C21	AR37	General purpose I/O
C22	AV40	General purpose I/O
C23	AU37	General purpose I/O
C24	AV36	General purpose I/O
C25	AU33	General purpose I/O
C26	AV32	General purpose I/O
C27	AU31	General purpose I/O
C28	AV30	General purpose I/O
C29	AV28	General purpose I/O
C30	AV26	General purpose I/O
C31	AV24	General purpose I/O
C32	AV22	General purpose I/O

Table C.11 Expansion port B pin constraints

C.2.4 Track layout

An automated routing program was used to connect the pins of each component within the PCB. The results of this formed the layer masks used to produce the PCB. The PCB is of a four-layer construction with a ground-plane layer, the power-plane layer (split into the various regulated power supply ranges) and top and bottom signal routing layers. Figure C.9 shows a composite reproduction of the masks used for the PCB production.

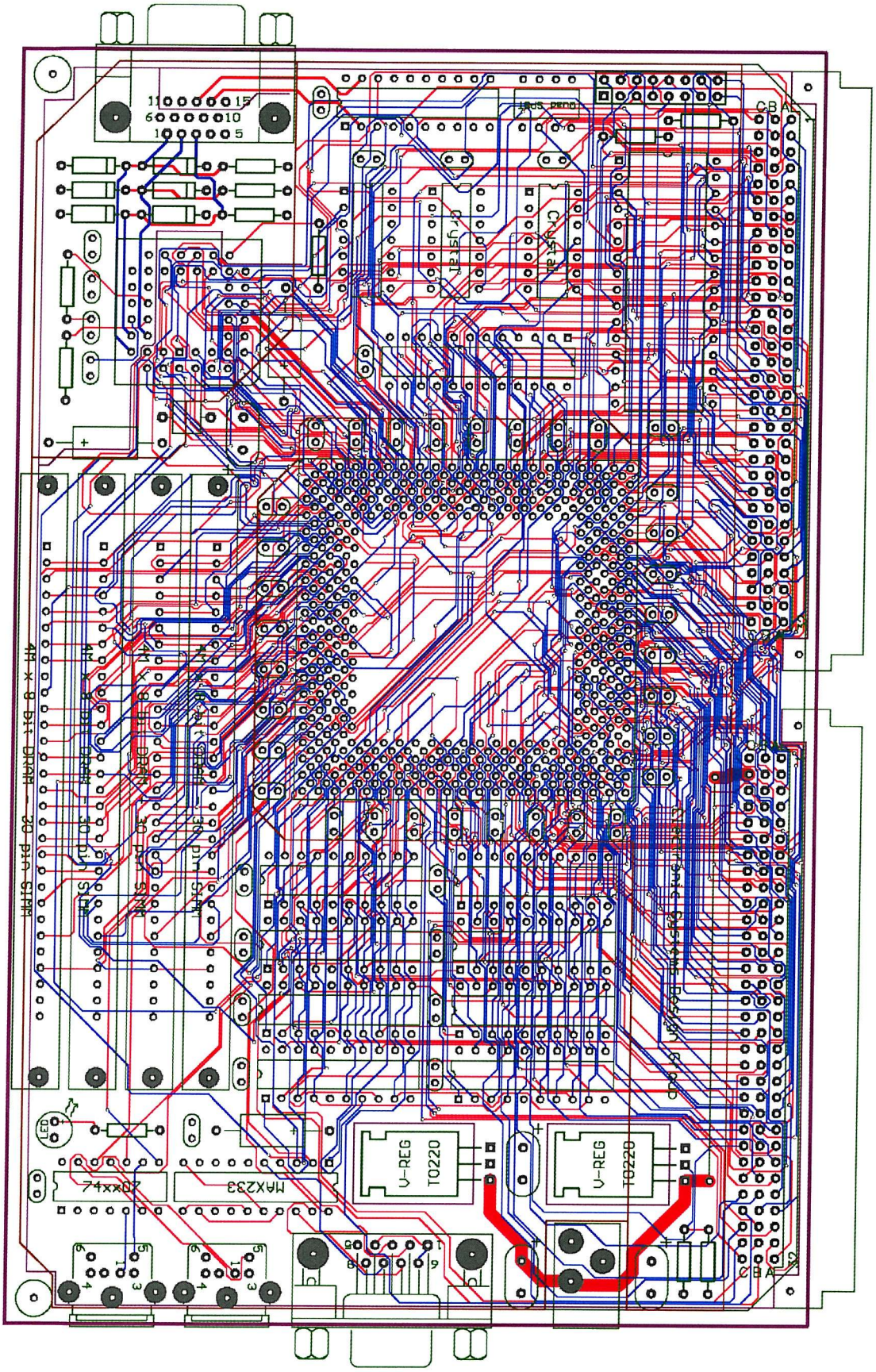


Figure C.9 General purpose PCB track layout

C.3 Demonstrator motherboard

The motherboard used within the demonstration designs was built manually using soldered wire-wrap wire on a grid-type pad board. It has seven 96-way female connectors which plug directly into the three FPGA boards (2 connectors each) and the audio board (single connector). The motherboard is described within Chapter 6, Section 6.1.3.1. The connecting wires that form the linkage between each system are shown within the following tables, which each show the pin linkage for each expansion port of each connecting board.

Table C.12 below contains the wire names for the main system board expansion port B. It links to the heap manager board, VGA-drive board and audio board.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1	BI	heap_addr (19)	I	RESET (GND)		GND
2	BI	heap_addr (18)	I	vga_vert_blank		SUPPLY
3	BI	heap_addr (17)	O	vga_serial_data (11)		
4	BI	heap_addr (16)	O	vga_serial_data (10)		
5	BI	heap_addr (15)	O	vga_serial_data (9)		
6	BI	heap_addr (14)	O	vga_serial_data (8)		
7	BI	heap_addr (13)	O	vga_serial_data (7)		
8	BI	heap_addr (12)	O	vga_serial_data (6)		
9	BI	heap_addr (11)	O	vga_serial_data (5)		
10	BI	heap_addr (10)	O	vga_serial_data (4)		
11	BI	heap_addr (9)	O	vga_serial_data (3)		
12	BI	heap_addr (8)	O	vga_serial_data (2)		
13	BI	heap_addr (7)	O	vga_serial_data (1)		
14	BI	heap_addr (6)	O	vga_serial_data (0)		
15	BI	heap_addr (5)				
16	BI	heap_addr (4)	O	vga_serial_sem		
17	BI	heap_addr (3)	I	vga_serial_ack		
18	BI	heap_addr (2)				
19	BI	heap_addr (1)				
20	BI	heap_addr (0)	O	heap_sem		
21	O	heap_size_offset (11)	O	heap_cont (1)		
22	O	heap_size_offset (10)	O	heap_cont (0)		
23	O	heap_size_offset (9)	I	heap_ack		
24	O	heap_size_offset (8)	O	heap_addr_in_valid	O	audio_clk
25	O	heap_size_offset (7)	O	heap_data_in_valid	O	audio_leds (3)
26	O	heap_size_offset (6)			O	audio_leds (2)
27	O	heap_size_offset (5)			O	audio_leds (1)
28	O	heap_size_offset (4)			O	audio_leds (0)
29	O	heap_size_offset (3)			I	audio_sclk
30	O	heap_size_offset (2)			O	audio_sdin
31	O	heap_size_offset (1)			I	audio_sdout
32	O	heap_size_offset (0)			I	audio_ssync

Table C.12 Main board expansion port B (top)

Table C.13 below contains the wire names for the main system board expansion port B. It links to the heap manager board (data path), contains the synchronisation signals and links to an external configuration ROM system that enables more than one configuration to be loaded into the main system. Two EPROMs contain the core designs for both the tracker and expression evaluator systems. These are held on the motherboard itself, along with the address decoding logic that selects which ROM to use.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1	O	PA0			BI	heap_data (31)
2	O	PA1			BI	heap_data (30)
3	O	PA2			BI	heap_data (29)
4	O	PA3			BI	heap_data (28)
5	O	PA4			BI	heap_data (27)
6	O	PA5			BI	heap_data (26)
7	O	PA6			BI	heap_data (25)
8	O	PA7			BI	heap_data (24)
9	O	PA8			BI	heap_data (23)
10	O	PA9			BI	heap_data (22)
11	O	PA10			BI	heap_data (21)
12	O	PA11			BI	heap_data (20)
13	O	PA12	I	ZERO_BOARD2	BI	heap_data (19)
14	O	PA13	I	ZERO_BOARD3	BI	heap_data (18)
15	O	PA14			BI	heap_data (17)
16	O	PA15			BI	heap_data (16)
17	O	PA16			BI	heap_data (15)
18	O	PA17			BI	heap_data (14)
19	O	PA18			BI	heap_data (13)
20	O	PA19	O	START	BI	heap_data (12)
21	O	PA20			BI	heap_data (11)
22	O	PA21			BI	heap_data (10)
23	I	PD0			BI	heap_data (9)
24	I	PD1			BI	heap_data (8)
25	I	PD2			BI	heap_data (7)
26	I	PD3			BI	heap_data (6)
27	I	PD4			BI	heap_data (5)
28	I	PD5			BI	heap_data (4)
29	I	PD6			BI	heap_data (3)
30	I	PD7			BI	heap_data (2)
31					BI	heap_data (1)
32	O	DONE			BI	heap_data (0)

Table C.13 Main board expansion port A (bottom)

Table C.14 below contains the wire names for the heap manager board expansion port A. It links the heap control, offset and address signals to the main system board, along with the system synchronisation signals.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1					BI	heap_addr (19)
2						
3					BI	heap_addr (18)
4					BI	heap_addr (17)
5					BI	heap_addr (16)
6					BI	heap_addr (15)
7					BI	heap_addr (14)
8					BI	heap_addr (13)
9					BI	heap_addr (12)
10					BI	heap_addr (11)
11					BI	heap_addr (10)
12					BI	heap_addr (9)
13			O	ZERO	BI	heap_addr (8)
14			I	START	BI	heap_addr (7)
15						
16					BI	heap_addr (6)
17					BI	heap_addr (5)
18					BI	heap_addr (4)
19					BI	heap_addr (3)
20			I	heap_sem	BI	heap_addr (2)
21			I	heap_cont (1)	BI	heap_addr (1)
22			I	heap_cont (0)		
23			O	heap_ack	BI	heap_addr (0)
24			I	heap_addr_in_valid		
25			I	heap_data_in_valid		
26			I	heap_size_offset (11)		
27			I	heap_size_offset (10)	I	heap_size_offset (4)
28			I	heap_size_offset (9)	I	heap_size_offset (3)
29			I	heap_size_offset (8)		
30			I	heap_size_offset (7)	I	heap_size_offset (2)
31			I	heap_size_offset (6)	I	heap_size_offset (1)
32			I	heap_size_offset (5)	I	heap_size_offset (0)

Table C.14 Heap manager board expansion port A (top)

Table C.15 below contains the wire names for the heap manager board expansion port B. It links the heap data signals to the main system board. It also drives two LEDs dependent on the activity of the heap transmission data and VGA driver activity for debugging purposes.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1	BI	heap_data (31)		RESET (GND)		GND
2	BI	heap_data (30)				SUPPLY
3	BI	heap_data (29)				
4	BI	heap_data (28)				
5	BI	heap_data (27)				
6	BI	heap_data (26)				
7	BI	heap_data (25)				
8	BI	heap_data (24)				
9	BI	heap_data (23)				
10	BI	heap_data (22)				
11	BI	heap_data (21)				
12	BI	heap_data (20)				
13	BI	heap_data (19)				
14	BI	heap_data (18)				
15	BI	heap_data (17)				
16	BI	heap_data (16)				
17	BI	heap_data (15)				
18	BI	heap_data (14)				
19	BI	heap_data (13)				
20	BI	heap_data (12)				
21	BI	heap_data (11)				
22	BI	heap_data (10)				
23	BI	heap_data (9)				
24	BI	heap_data (8)				
25	BI	heap_data (7)				
26	BI	heap_data (6)				
27	BI	heap_data (5)				
28	BI	heap_data (4)				
29	BI	heap_data (3)				
30	BI	heap_data (2)				
31	BI	heap_data (1)			O	vga_active
32	BI	heap_data (0)			O	heap_active

Table C.15 Heap manager board expansion port B (bottom)

Table C.16 below contains the wire names for the VGA drive board expansion port A. It links all the serially interfaced VGA communication signals and the system synchronisation signals.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1					O	vga_vert_blank
2						
3					I	vga_serial_data (11)
4					I	vga_serial_data (10)
5					I	vga_serial_data (9)
6					I	vga_serial_data (8)
7					I	vga_serial_data (7)
8					I	vga_serial_data (6)
9					I	vga_serial_data (5)
10					I	vga_serial_data (4)
11					I	vga_serial_data (3)
12					I	vga_serial_data (2)
13			O	ZERO	I	vga_serial_data (1)
14			I	START	I	vga_serial_data (0)
15						
16					I	vga_serial_sem
17					O	vga_serial_ack
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						
31						
32						

Table C.16 VGA drive board expansion port A (top)

Table C.17 below contains the wire names for the VGA drive board expansion port B. It exists as a placeholder for the power supply and reset signal (connected to ground). It also drives two LEDs dependent on the activity of the serial transmission data and VGA driver activity for debugging purposes.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1				RESET (GND)		GND
2						SUPPLY
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						
31					O	vga_active
32					O	serial_active

Table C.17 VGA drive board expansion port B (bottom)

Table C.18 below contains the wire names for the audio board connector. The audio board was originally designed to connect to expansion port B of the first PCB directly. The motherboard uses the provided 3.3V regulated power supply to drive all the '1' constants back into the audio board. The audio serial communication signals link directly to the main system board. The audio board also contains four LEDs that are directly driven from the main system board. These are used for debugging purposes.

Pin	Pin row					
	A		B		C	
	dir	wire name	dir	wire name	dir	wire name
1	I	audio_SMODE (3) "0"				GND
2	I	audio_SMODE (2) "1"				SUPPLY
3	I	audio_SMODE (1) "0"			I	audio_clkout
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15	O	+3.3 Volts ("1")				
16						
17	I	audio_MF8_SFS2 "0"				
18	I	audio_MF7_SFS1 "0"				
19	O	<< audio_MF6_DI2 >>				
20	I	audio_MF5_DO2 "0"				
21	I	audio_MF4_MA "1"				
22	I	audio_MF3_F3 "0"				
23	I	audio_MF2_F2 "0"				
24	I	audio_MF1_F1 "0"				
25	O	<< audio_DI1 >>				
26	I	audio_DO1 "0"				
27	I	audio_nPDN "1"			I	audio_leds (3)
28	I	audio_nRESET "1"			I	audio_leds (2)
29	O	audio_SDOUT			I	audio_leds (1)
30	I	audio_SDIN			I	audio_leds (0)
31	O	audio_SSYNC				
32	O	audio_SCLK				

Table C.18 Audio Board

The motherboard also contains a provision for two configuration ROMs that drive the main system board (the other two FPGA boards program themselves directly). The two ROMs contain the tracker design and the expression evaluator design. The particular design is selected via a DIP-switch on the motherboard. The address decoding logic for ROM selection, along with the EPROM connectivity is shown within Figure C.10.

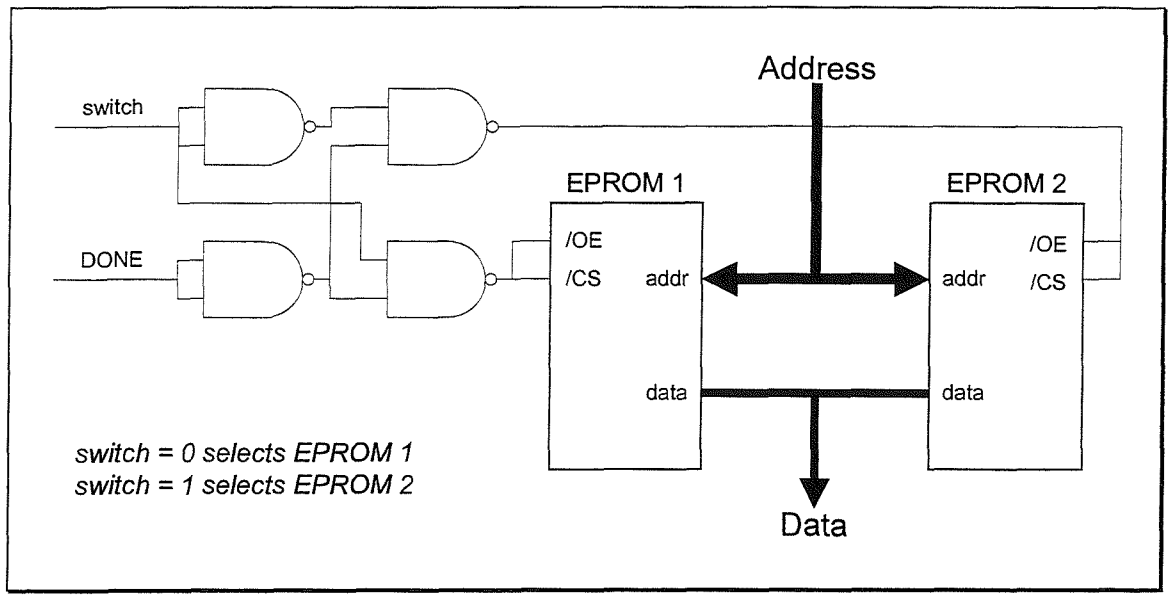


Figure C.10 Address decoder logic

The 'DONE' signal input is driven from the FPGA, and is low when programming. The switch input is from the selector DIP-switch. Two signals are generated that drive the chip-select and output-enable inputs of both EPROMS. This enables the data paths to be joined directly into a single data path, as only one EPROM will drive its outputs at any one time. The FPGA reads all 8 data bits in parallel from the programming data lines when programming. The FPGA also drives the address bus from the programming address lines. It is only in parallel modes that the addresses are driven and all 8 data input bits are used. The address counts from zero, up through the full address range required for a full configuration, controlled by the FPGA, which is master.

C.4 VGA serial interface controller

The serial interface to the VGA controller system was designed to reduce the pin count required between the user's design that draws objects and the VGA controller design that performs the actual drawing actions and displays them on a monitor. It does this by creating communication instructions that contain differing data dependent on the preceding instructions. Different actions require different numbers of instructions dependent on the amount of data that is required. The entire interface requires only 15 bits for all communications, 12 of which are instruction data bits.

C.4.1 Interface

Four types of communication are supported, each relating to a different VGA interface procedure. The procedures supported are a change in foreground drawing colour, a palette change, the drawing of a rectangle and the drawing of a character. Four interface procedures are provided for the user to use within a limited version of the VGA interface procedures. These procedures take the given drawing data from the user's design and split the information into a number of instructions. The first instruction contains information about the type of interface procedure also.

Interface procedure	Set foreground colour	Set palette	Draw rectangle	Draw character
IO bits	4	4 + 12	10 + 9 + 10 + 9	7 + 6 + 2 + 2 + 8
Instruction bits	2	2	2	2
Total bits	6	18	40	27
Instruction 0	type + colour	type + colour	type + x0	type + x0
Instruction 1		RGB	y0	y0 + xs + ys
Instruction 2			x1	char
Instruction 3			y1	

Table C.19 Information contained within serial interface instructions

The number of instructions required for a transmission is dependent upon the number of bits requiring transmission and the data path width of the semi-serial transmission data. The instruction data is transmitted within 12 bits, of which 2 bits are used within the initial instruction to determine the type of transmission. Note that the foreground colour setting is achieved within a single instruction, as it only requires 6 bits of data.

The limited set of interface procedures requires that the VGA controller system be held in constant drawing mode within the same pages. The mode chosen for the tracker and the expression evaluator is a direct-draw of the foreground only (for text), with both foreground and background pages held as page zero.

C.4.2 Controller

The controller is used to directly connect to the VGA controller system. It is contained within the same FPGA, where pin limitations are not an issue. It performs the inverse of the VGA serial interface procedures, by decoding the incoming instruction data.

The control flow within the decoder is shown within Figure C.11. It effectively reads in an initial instruction and works out the type of information held within the instruction and any following instructions.

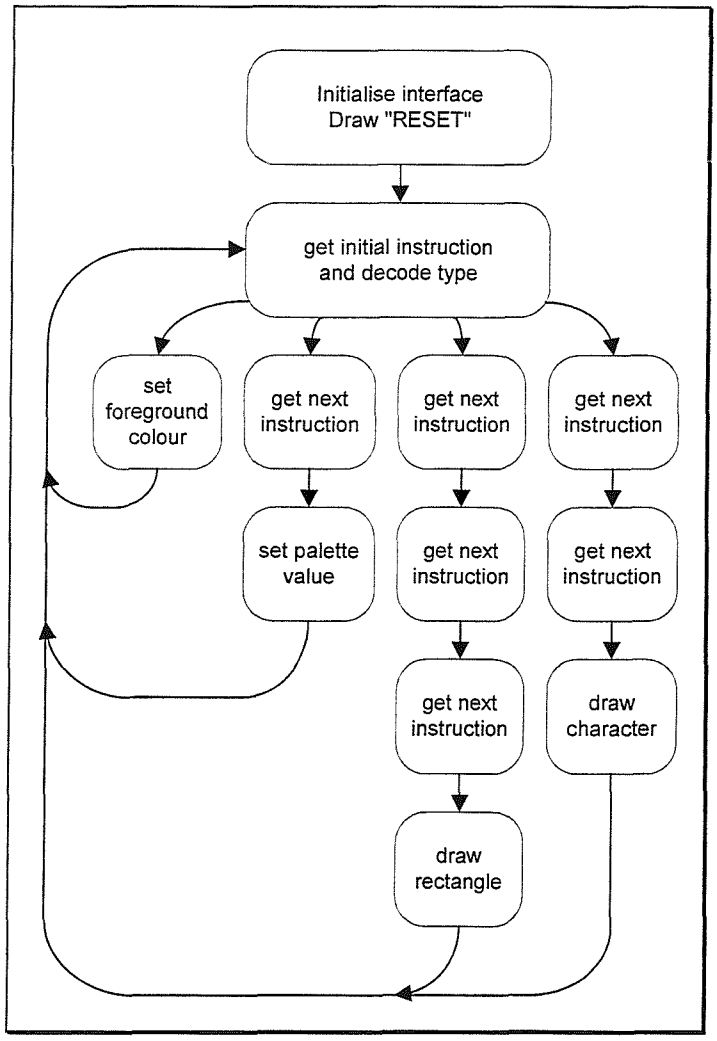


Figure C.11 VGA serial controller control flow

Then a choice of flow is made dependent on the type, which allows a further set of instructions to be read in. After the correct number of instructions is read, the drawing action is performed via the VGA interface procedures to the underlying VGA controller and control returns to the initial point of reading in the next instruction. The controller design is a slave to the user's design, which initiates all transmission.

C.5 Heap manager

The heap management system described within Chapter 4 has been implemented within an FPGA. It is described using behavioural VHDL and synthesised using MOODS itself. The memory space controlled by the system is 1Mword, where each word is 32 bits.

During development, various versions of the manager were created, each based upon the last. The final version used to demonstrate the capabilities of synthesised systems that use dynamically allocated memory has a real-time VGA display driven by the heap system. The information displayed is explained within Chapter 6. The information is displayed in real time with no effect on the speed of allocation or any other communication with the user's design.

C.5.1 Code implementation

The system is designed with the use of various concurrent processes. The main heap management algorithm is implemented within one of these processes. Other processes are used to control the underlying DRAM, for which a refresh counter takes another process. The real time VGA drive monitor resides within another process, using a buffered communication process to form the zero-time-overhead link with the heap management algorithm process. The heap management algorithm is implemented using a number of procedures, some of which are used for communication with the other processes within the design.

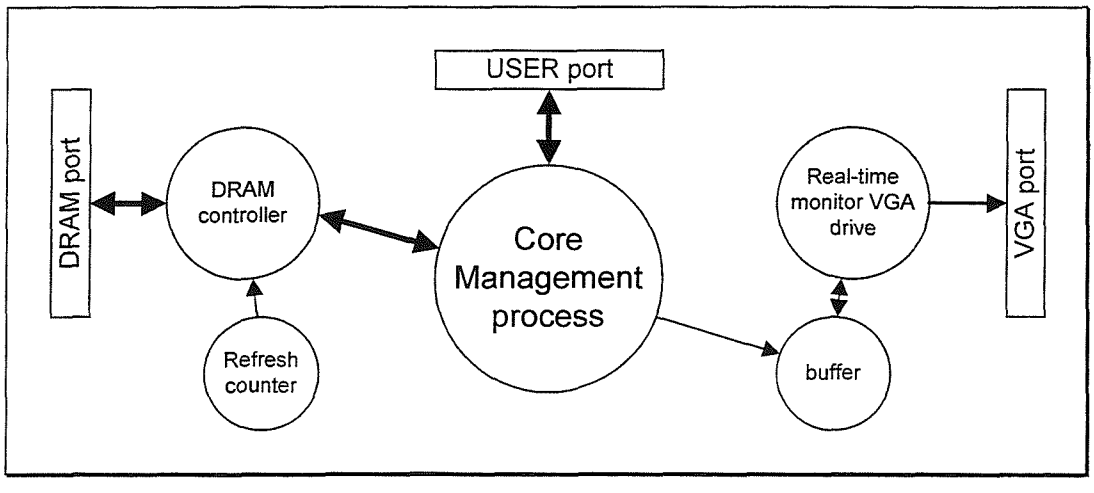


Figure C.12 Heap manager communicating processes

C.5.2 DRAM control process

This process is used to sequence the address and data lines to the DRAM memory bank used to store all heap management and user data. DRAM has a multiplexed address bus that is controlled by row and column strobe signals (/RAS and /CAS). The direction of data flow is controlled by a third 'write-enable' signal (/WE). This type of memory also requires constant refreshing pulses, due to the charged-capacitor method of data storage.

This process controls three interface sequences. The refreshing sequence consists of driving the /CAS then /RAS signals, then resetting both. This operation is required every 128 cycles using a 12.5MHz clock on average. Each refreshing operation refreshes a single row of the memory grid. An internal counter within the DRAM controls the selected row. The two other operations are a single read and write access to any address within the memory. Both accesses begin by setting the row address on the address bus, driving the /RAS signal, then setting the column address on the address bus and then driving the /CAS signal. A memory read is performed by not driving the /WE signal, where the data appears on the data bus after a small delay from driving the /CAS signal. A memory write is performed by setting the data on the data bus and driving the /WE signal before the /CAS signal is driven. All operations return once the /RAS, /CAS and /WE signals are reset.

C.5.3 Refresh timer process

This process simply generates a signal that inverts every 128 clock-cycles. The DRAM memory controller to initiate a refreshing sequence uses this signal and acknowledges it via an internal acknowledge variable. It is impossible to miss a refresh inversion due to the limited time taken by all memory accesses, and the refreshing takes priority over all other accesses.

C.5.4 Core process

This process performs all the controlling actions for the heap algorithm. It forms the interface with the user's design via the heap manager port and interfaces with the DRAM memory controller for all memory reads and writes of the fixed data space that the manager is controlling. It also communicates with the display buffer, telling the display process what to display.

C.5.4.1 Memory access interface procedures

All memory accesses are made via the two interface procedures defined within the core process. These communicate with the DRAM controller via a set of internal signals that hold the address and data busses and the communication semaphores. The internal address bus holds a full width address. The two operations are a completely dynamic access read and write of a single data word (32 bits). The controller does not exploit the fast page mode action of the DRAM.

C.5.4.2 Setup

The initial stages of the algorithm call for the setup of the underlying memory. This consists of the creation of the free-list of all pages and the null page pointer setup.

The free page list is created by looping through all available pages, from page 255 to page 1. The first word within each page is written with the base address of the previous page within the loop and the free list base pointer will hold the base pointer of page 1. Each loop operation requires a single memory write access, totalling 255 writes.

The page pointers held within page zero are all required to point to a null page initially. This is simply achieved by looping for all words within page zero, writing a null address into each data word. This also requires only one memory write access per loop iteration, totalling 4090 writes (the number of valid page sizes).

C.5.4.3 User interface loop

All communication with the user's design is initiated by that design, with the heap management system acting as a slave to the master user's design. Once the heap management algorithm setup has occurred, the heap manager enters the user interface loop, in which any of the four actions upon the heap can be entered. The four actions supported are an allocation of a number of words, the deallocation of a given word block and the read and write of a single word from within the allocated objects. These are all explained within Chapter 4.

C.5.4.4 Heap management procedures

The management algorithm is created from a number of procedures that each perform specific actions upon the heap data space. They are called from the user design communications of each type of access. The call graph for each section within the heap management process is shown with Figure C.13.

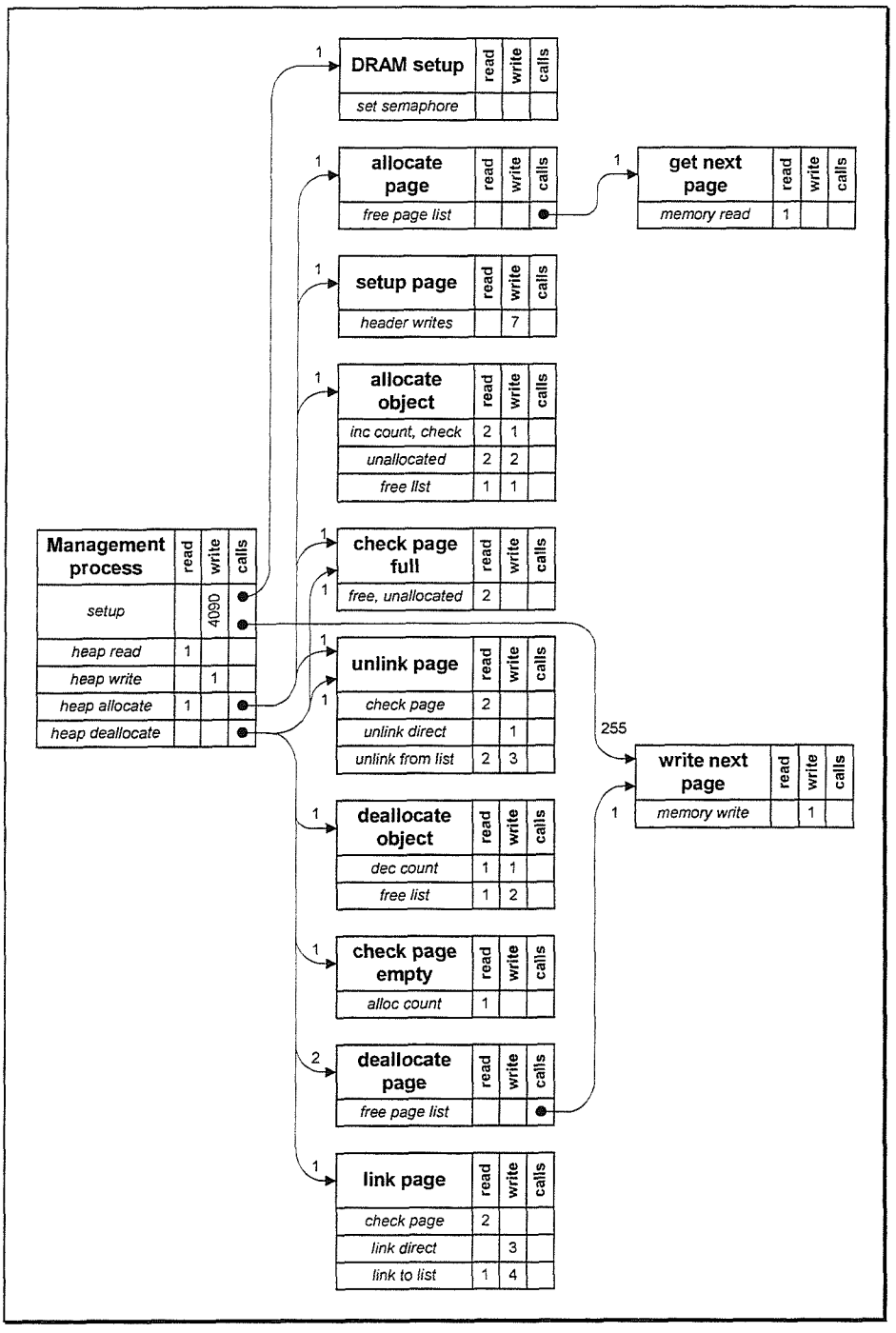


Figure C.13 Heap management algorithm call graph

The DRAM setup procedure simply initiates the communication semaphore used internally by the DRAM read and write procedures.

Note that each procedure contains a count of the number of memory read and write actions performed on the heap data space. This gives an idea of the number of read and write operations required for each action, with the simple heap read and write actions mapping directly onto a single DRAM read and write operation and the more complicated allocation and deallocation operations taking multiple memory read and write operations. All procedures are also inlined into the calling process for speed and area considerations.

The '*write next page*' procedure simply writes a given next page pointer into the base address of a given page. This is used for the insertion of free pages onto the free page list.

The '*get next page*' procedure performs the opposite operation to the '*write next page*' procedure, in that it reads the address of the next page pointer from a given page. This is used for the removal of a page from the free page list.

The '*allocate page*' procedure is used to remove a page from the free page list. It decrements an internal free page counter also.

The '*deallocate page*' procedure is used to re-insert a page onto the free page list. It increments the internal free page counter also.

Once a new page has been taken from the free list of pages, the '*setup page*' procedure is called to fill the header with the valid values for the object size, object unallocated pointer and free object list base pointer. It also inserts the page onto the page size page pointer within page zero and sets up the next and previous pointers of the page to point to itself.

The '*allocate object*' procedure is used to return an object pointer from within a given page. It first checks the free object list from within the page and returns the head of the list if any objects exist within the list. If no objects are in the free list, then the procedure will return the unallocated pointer and increment the same pointer (checking for no more space). The number of allocated objects within the page is also incremented.

The '*deallocate object*' procedure performs the opposite operation to the '*allocate object*' procedure in that it takes an object pointer and inserts the object onto the free list of objects within the given page. It also decrements the object count within the page.

The '*check page full*' procedure is used to determine whether the page is completely full of objects after an object is allocated from the page. It first checks the unallocated pointer then the free list if there is no space left from the unallocated space. If both the free list head pointer and the unallocated space are null, then the page is full of objects.

The '*check page empty*' procedure is used to determine whether the page is completely empty of objects after an object is deallocated from the page. It checks the object count value to determine whether any objects are contained.

The '*unlink page*' procedure is used to remove a page from the doubly linked list of pages currently in use for a particular object size. If the page is the only page within the list, then the page size pointer is set to null. If there are other pages within the list, then the links of the doubly linked list of the next and previous pages are linked together, removing the given page from the list. The page size pointer is reset to the next page within the list if it points to the page being removed.

The '*link page*' procedure performs the opposite operation to the '*unlink page*' procedure. It is used to insert a given page into the doubly linked list. The page size pointer is checked first for the availability of the doubly linked list. If one doesn't exist, then the given page is set up as the only page within the list and the page size pointer is set to point to it. If a list already exists, then the page is inserted before the current head of the list by the adjustment of the four next and previous pointers of the given page and the pages already within the list.

C.5.4.5 Memory status interface procedures

These procedures are called to fill the memory status buffer with the page status information. They communicate with the buffer process via some internal signals and communication semaphores. They are blocking procedures that are called in positions that do not add an increased number of control states required for the communication. They are also inlined into the calling process.

The procedures are called from the heap management process (are not shown within the call graph of Figure C.13). They all take a page as a parameter and write different information into the buffer about the given page. They all set the page as being accessed and used (the page becomes highlighted).

If the user makes a heap read or heap write operation, the '*set page status used*' procedure is called for the page that contains the object that is being accessed. This just sets the page as being accessed and used.

The '*set page status free*' procedure is called from the heap setup phase for each page inserted onto the free list. It is also called from a deallocation, when a page is re-inserted onto the free page list. The buffer is set as the page being empty.

The '*set page status partfull*' procedure is called from both the allocation and deallocation operations. The page is set as being partially full and in use. It is called within allocation when a page is taken from the free page list and becomes active. It is called within deallocation when a page has an object removed, which results in the page being reinserted onto the active page doubly linked list.

The '*set page status full*' procedure is only called from an allocation, where a page becomes totally full from an object allocation. The page is removed from the active page list in this case.

C.5.5 Memory map buffer process

The buffer process is used to control an internal RAM array that stores a representation of the state of each page within the heap management algorithm. Four bits are required per page, with 256 pages (addresses into the RAM) requiring storage. The buffer has an 8-bit address and 4-bit data path.

Two bits are used to describe the allocation state of the page. One bit determines that the page is empty (free) while the other specifies that the page is full. If both are false, then the page is partially full of objects. It is invalid for both to be true, as a page cannot be both empty and full of objects at the same time.

The other two bits determine whether the page has been accessed within the VGA frame raster scan period ($1/60^{th}$ of a second). Two bits are required for persistence of the information through at least one raster scan period. The heap management process for any access of the page sets both bits. The bits are reset one after the other by the VGA drive process, which provides the time-out of the raster scan period. The VGA drive process has an interface to the buffer that is second in priority to the heap manager interface to the buffer. The heap manager interface is write-only, while the VGA drive process has read/write access to the buffer.

C.5.6 VGA drive process

This process is used to draw a graphical representation of the contents of the buffer controlled by the buffer process. The process draws a representation of every page within the heap management algorithm in a 16 by 16 grid in the centre of the screen.

The drawing process begins with the setting up of the interface to the VGA controller, initialisation of the colour palette and the erasure of the background screen. The process then enters an infinite loop. The drawing process is then forced to wait for the vertical blanking period to begin. Once the vertical blanking period is entered, an inner loop counts for each page, with the screen coordinates being calculated to produce a square grid of pages, starting at the top left corner for page zero.

It is at this point within the inner loop that the page buffer is read for the relevant page. An interface procedure is created within the VGA drive process for this purpose. The information for the page is returned and the activity bits contained within the buffer are reset one at a time. The page information is then rendered to the VGA controller via two coloured rectangles that overlap to form a central rectangle with a border at the current page coordinates. The larger rectangle forms the border and is coloured light grey to show page activity and dark grey to show no activity within the page. The inner rectangle is coloured in one of three colours dependent on the allocation status of the page, where blue represents a page on the free page list, green represents a page that has objects contained within it but not full and red represents a page that is completely full of objects.

C.6 Tracker demo

The tracker design is given a basic data structure and user guide overview within Chapter 6, Section 6.2. This appendix section gives a more detailed description of the methodology used within the behavioural VHDL source code to produce a working tracker design.

C.6.1 Code implementation

The two core features to the design methodology used are with the use of the dynamic data structures (what is being demonstrated) and with the use of concurrent processes to handle relatively strict data throughput timing constraints. The audio throughput data requires 44,100 sample values per second, which equates to an allowable time period of 22,675 ns per sample. With a system clock running at 12 MHz, the audio streams have 272 clock cycles of processing time.

This number of clock cycles is enough for the relatively simple operations performed on the audio data. However, the dynamic memory element to the design requires sample block allocation, where each single physical memory allocation takes about 80 cycles to complete due to the underlying sequential DRAM memory operations. General memory accesses take around 10 cycles to complete when asynchronous communication buffering is taken into account.

As all dynamic memory accesses require sequential operation, the number of clock cycles taken within a single audio value period can exceed the 272-cycle limit. However, on average, the number of cycles required for each audio value will be less than the clock cycle limitation, which means that if the audio data streams were buffered with a number of FIFO buffers capable of storing a number of values to average out the differences in time to process each audio data value, the memory latency issue would disappear, leaving only the memory bandwidth as the system limitation.

The system is designed using two core processes to handle the real time audio processing and the output user interface drawing. Each audio stream is buffered using a 16-element FIFO. The audio streams are taken from the ADC/DAC controller process, which communicates with the external interface chip.

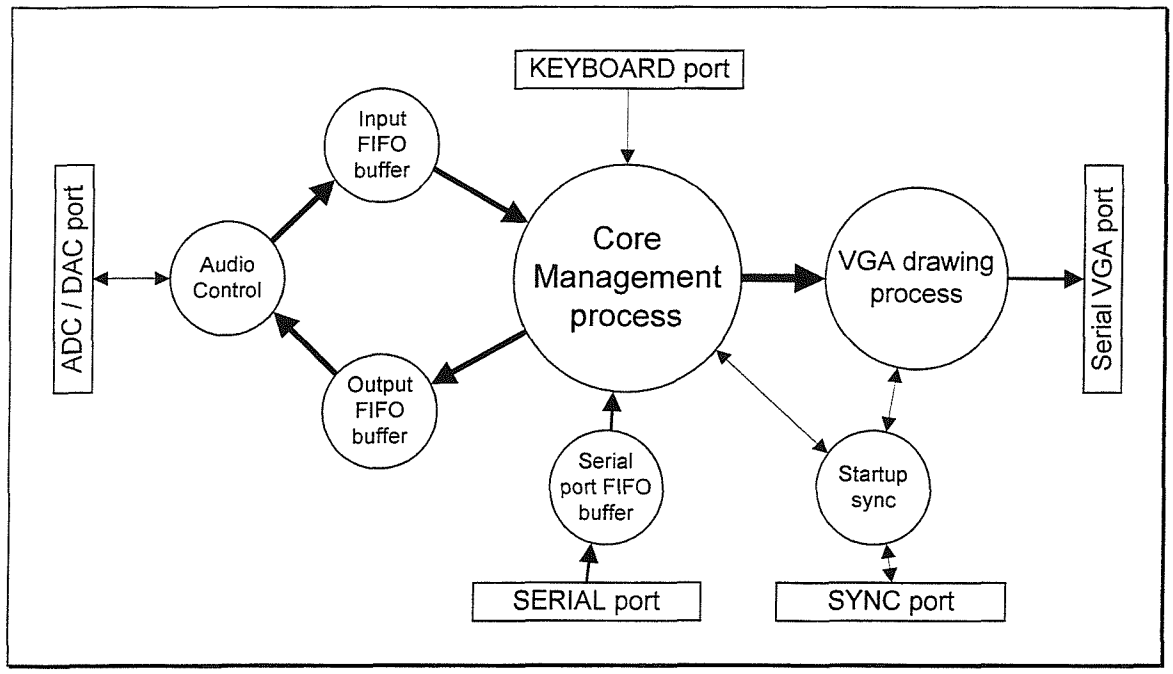


Figure C.14 Tracker processes and data flow

C.6.2 Data structures

The stored samples and sequences are all stored dynamically using VHDL data structure constructs. These are formed from aggregate record types and array types with access type references to the dynamic creation of these data structures.

C.6.2.1 General linked lists

The general linked list structure is explained within Chapter 6.2.1.1. It is formed from a base list record type that stores a list of element type record items, which each point to the actual data structure being listed. This structure allows for common code to implement the list insertion, deletion and traversal operations that are used within each list, leading to smaller source code size. The alternative would be to have each listed data structure form a list within itself.

C.6.2.2 Strings

Strings are used for text descriptions of samples and sequences. They are formed from a limited array (16 elements) of ASCII character types. They are created dynamically with the use of access type variables that reference the array type. In the current

implementation, only the serial download mode is capable of filling the strings with arbitrary description data.

C.6.2.3 Samples

A sample is formed from a record type variable that contains the various items that make up the full description of an audio sample. The record elements include an integer ID for the sample, a string description (referenced by an access type to a string array) and the sample block list, which contains all sample data stream values. The list is formed from a reference to a linked list base data structure.

The sample block list holds a number of fixed length arrays of data stream elements, which when put together, form the dynamic length sample stream. A sample stream is stored in this way due to the unknown length of the sample, even when recording. The sample record type also contains an element that holds the last valid index into the tail sample data block, as the tail may only contain a partial array of valid data stream items.

A pair of sample block iterators are also stored within the sample record, along with index positions within the referenced sample block. These store the sample block looping positions, that are used when a sample is played past its ending position. The sample may be played as looped or single-shot, which is determined by the final sample record element. These sample array references are set to the first and last valid elements within the entire sample by default.

C.6.2.4 Sequences

A sequence is formed from a single record type. This record contains elements that give each sequence a unique integer ID and a text description in the same manner as the sample description. The sequence is capable of holding a reference to a number of note items at any time point using any of the 8 channels. The length of time that a sequence stores is held within an element within the sequence record (the default is 128 time positions, with 256 being the maximum). This stores the last time point to be used within the sequence. The notes stored within the sequence are contained within another linked list structure, with a reference to the base pointer of this linked list being the last element within the sequence record.

The linked list of sequence elements contains a sorted list of items. The sorting occurs by insertion at the correct element position. Sorting occurs in two dimensions, with the time position being the first, followed by the channel number.

Each sequence note item is itself contained within a record, with elements describing the item position, both time and channel, and the description of the note to be played at that time position, using a reference to a sample and a playback speed determined by an octave number (bass-1 to treble-5) and a note number (A to $G^{\#}$). A null sample reference will stop playback on the relevant channel.

The sequence elements are stored in this manner due to the sparse nature of the notes. An array of sequence point items would quickly fill the available memory space. This list is capable of full insertion and deletion capabilities via the input user interface.

C.6.2.5 Playlist

The playlist is formed from a list of references to the iteration list items held within the sequence list. It only references the sequences, forming a secondary link into the sequence list data structure. Full list insertion and deletion capabilities are supported. A link to the iteration element within the sequence list is formed due to the capability of alteration of the referenced sequence via iteration through the sequence list. A reference to the sequence item itself would not allow this, as the sequence has no link to the containing list data structure.

C.6.2.6 Real-time buffer arrays

As the output user interface is drawn within a separate process from the audio processing process and the user interface draws a representation of the various data streams for each channel and the mixed output stream, the information to draw requires transferral from one process to the other. This would usually be achieved with the use of a number of statically created buffers. However, as the amount of data to be stored within these arrays is relatively large, the storage space is allocated from the heap dynamically as the heap is capable of storing a large amount of data. This is only performed once on system initialisation. Shared variables hold the base references to the dynamic arrays and communication is performed by semaphore and acknowledge signals.

C.6.3 Concurrent process communication

The flow of data between processes is controlled by a number of semaphore and acknowledge signal pairs, each controlling the flow of data being transferred by data signals and shared variables.

C.6.3.1 Semaphore signals and shared variables

Shared variables are used for the transferral of access type held information between concurrent processes, while signals are used for the transferral of all other data types. A pair of semaphore and acknowledge signal bits control the data flow with one process as the master. The master process initiates the communication with the inversion of the semaphore signal and the slave process acknowledges the communication with the inversion of the acknowledge signal. Data flow can be in either direction. The core audio processing process is the master of all data communications both to and from it.

C.6.3.2 User interface redraw control

The ability to read the dynamic data structures within one process while concurrently modifying the same data structures within another process has potential pitfalls. For this reason, the user interface process (which draws a representation of the data structures) is controlled by the audio processing process, which is capable of the modification of all sections of the data structures.

All redraws made by the user interface are initiated by the audio processing process from within the internal audio processing loop. At times it is impossible to update sections of the output user interface due to the relevant sections of the data structures being modified. The user interface may also take a long time to draw sections of its display. It is critical that the drawing process does not delay the audio process, so the redrawing of display sections is initiated via internal flags within the audio process that only get serviced when all other drawing has finished.

C.6.4 Core process

The core system process handles all data structure creation and modification via the input user interface. It also drives the output user interface, telling it what to draw. The process

loop is also used to form the mixed audio output signal that drives the output FIFO buffer and records the buffered audio input stream when required.

C.6.4.1 Keyboard interface

A standard QWERTY keyboard is used as the input device for the tracker system. The core process interfaces with it via the non-blocking communication procedure of the keyboard interface. The keys pressed directly influence data structure modification, audio mixing control and sequence recording and playback.

C.6.4.2 Serial port interface

The serial port is interfaced with via an input FIFO buffer. The buffer uses the interface procedures provided by the serial port interface package, while the core process uses internal communications with the FIFO buffer. The serial port is only read within the serial download mode, which stops all audio processing and user interface control.

The serial download loop is used to create all data structures held within the tracker. This facilitates the storage of sequences and samples within a computer hard disk, A communications protocol exists that makes the computer the master system, only releasing the tracker system into the main audio processing loop once all data has been downloaded. All drawing is flagged to redraw after a serial port download.

C.6.4.3 Operation modes

There are three operating modes that select the functionality of various shared keyboard operations. Each mode is contained within the main audio processing loop unlike the serial port download loop. The sample mode allows for the creation, modification and viewing of a number of samples. The sequence mode allows for the creation, modification and viewing of a number of sequences. The playlist mode allows for the modification of the playlist of sequences. All modes, except the recording of sequences or samples, support manual mixed audio playback and the playback of the currently selected sequence or the entire playlist of sequences.

C.6.4.4 Sample recording (sampler)

A new sample is recorded from the input audio stream by being in the sample mode and pressing the 'R' key. This action dynamically creates a new sample and inserts it onto the sample list. The sample record is then initialised and the sample block list is generated. The input audio stream is then taken from the input FIFO buffer and fills the sample block with the data. Whenever the sample block becomes full, a new block is allocated and inserted onto the end of the block list. The incoming audio data then fills the new block. Recording finishes by releasing the 'R' key, which enables the polyphonic playback on each channel again. The sample and sample list are flagged to redraw after a new sample is created.

C.6.4.5 Sequence editing

A new sequence can be created, inserted and initialised within the sequence mode. The cursor position within the sequence denotes the time and channel position where a new note item would be inserted when in sequence recording mode. Instead of the note keypresses driving the polyphonic mixed output audio playback, the note is inserted into the sequence point list in the correct sorted position. A reference to the note point before the highlighted position is always kept as the position to insert after. If a note item already exists at the insertion position, then the note item is deleted before reinsertion of the new note item. The sequence is flagged as requiring a redraw after every modification and the sequence list is flagged to redraw after a new sequence is created.

C.6.4.6 Playlist editing

The playlist can be edited within the playlist mode. The list can be iterated through, which selects a different sequence. Each change of list position flags a redraw of the playlist, sequence list and referenced sequence. A list insertion will insert at the currently selected list position, using the currently selected sequence as the inserted value. Only the playlist is flagged to redraw in this situation. The list modification is performed by the generic list modification procedures.

C.6.4.7 Sample playback

Sample playback is initiated by a manual user keypress or via the sequencer. The same code is used for both. The only difference being that a playback channel is selected for the

manual note playback, while the channel is known for the sequenced playback. Once a channel is set as playing a particular sample using a given note and octave, the data within the sample is incremented through using an over-sampled index. Only the most significant bits of the index are used for the sample index position. This index gives the current playback sample value position within a sample block. Whenever the index wraps around, the next sample block is iterated to.

The rate of audio output playback is fixed at 44.1 kHz. The different frequencies required for the different notes and octaves are created by incrementing the sample index position by a different amount depending on the note being played. This is where the over-sampling of the index position is used, with bass notes being incremented by a number effectively lower than 1 (if the over-sampled index position is taken to be a fixed point fractional number, with the only bits used as the index value being the integer part, with the over-sample bits being the fractional part), and treble notes being incremented by a number effectively greater than 1 index position. Middle-C is the recording rate, which is played back with an increment of exactly 1 index position per stored data value.

The 8 dual-16-bit stereo audio channels and the dual-16-bit audio input stream from the ADC are mixed into the single stereo audio output stream by simply adding the sample values for each channel together, along with the audio input stream values from the ADC. The stereo audio values are treated as two signed numbers, where addition beyond the bit-range limits of the dual-16-bit output will result in the audio signals being clipped to the minimum or maximum limits of twos-complement 16-bit numbers.

C.6.4.8 Sequencer playback

The sequencer can be played from the playlist of sequences or from the selected sequence only. The only difference is that the playlist item is iterated (and looped back) when played from the playlist, which selects a different sequence to play from each time the sequence increments past the last time position.

A sequence is played by incrementing the time position to be played back. Each time this happens, each sequence point item for the current playback time position is read and acted upon. The movement through time of the sequence is performed at a slower rate determined by an internal counter and variable counter limit. This means that the speed of

playback of the sequence can be altered, with the use of the global sequence speed counter. The default rate for playback gives 8 sequence time positions per second.

Whenever the time position changes, the sequence point items for the time row are read and, if they exist, are placed in the playback channel in which they reference. The playback channel holds the current sample being played on that channel, the rate of playback and the current iteration position through the sample. The iteration position is reset to the first sample index within the sample whenever a new item on that channel is found. Storing a null reference to a sample within a sequence time point can halt playback of a note on a given channel.

C.6.5 Drawing process

The drawing process interfaces with the VGA display system via the serial interface to that controller. The serial interface is explained within Appendix C.4. The process draws a representation of the contents of the data structures used within the tracker. The core control process initiates all drawing once the initial setup phase is complete. The drawing mode is fixed as direct draw of the foreground only within page 0, rastering from page 0 also. This means that everything drawn overwrites what was previously at the drawing position, except the text background colour, which is not drawn at all. Only a single frame buffer page is used.

C.6.5.1 Initial setup

The drawing process sets up the interface to the VGA serial controller by calling the initialise procedure defined within the interface package. After this, the palette is set up for the colour scheme used within the tracker. Then the background screen is drawn, with the various window borders and constant description strings. After this setup phase, the main drawing loop is entered, which awaits the core process to tell the drawing process what sections of the data structures to draw.

C.6.5.2 Drawing strings

Two types of string can be drawn, constant and variable type strings. Constant strings hold the various banners used to describe the sections. These are drawn from a constant internal ROM, with base and end indexes into the ROM defined as constants. A procedure that

loops from two indexes into the ROM given a drawing position and size is created to draw constant strings horizontally.

The variable type string is drawn in the same manner, except that it takes an access type variable input that references a string created on the heap. All strings are arrays of 16 characters. The drawing procedure also takes a drawing position and an end index. If a null character is found within the string, then drawing stops at that point.

C.6.5.3 Drawing generic lists

The same generic drawing procedure is used to draw the three base lists of the sample list, playlist and sequence list on the left hand side of the screen. The procedure takes the drawing positions for the upper, middle and lower y-positions and the left and right x-positions. It also takes the current list iteration position and the type of list being drawn. The procedure draws over the background first, removing the previous list contents. This is followed by the iteration through the list, following the previous references drawing the contents up the screen and the next references drawn down the screen. The information drawn is with respect to the given type of list, with samples and sequences having their integer IDs and text descriptions listed. The playlist type draws the referenced sequence information.

C.6.5.4 Drawing real-time audio

The real-time audio signal representation is drawn only once every frame. The blanking period of the VGA controller is used to reset the acknowledge signal that tells the audio process to refill the audio drawing data buffers. Once filled, the audio process sends a semaphore signal to initiate the drawing.

The background box containing the drawn wave is drawn first. Then, simply looping through, reading the array values stored by the shared variable reference of the dynamically created arrays and drawing the wave section at that position draws the waves. The wave is drawn from left to right with the left sample overwriting the right sample. A vertical line is drawn between the old sample value and the new sample value at the current index and x-position.

C.6.5.5 Drawing samples

The currently selected sample is drawn at the top of the screen next to the sample list. Only a small section of the sample is drawn, starting from a sample block iterator not necessarily the first block within the list. The sample drawing uses the same code to draw the wave sections, with iteration through the sample block list and indexes within each block used to provide the stereo drawing data.

C.6.5.6 Drawing sequences

The sequence drawing takes up the proportion of the displayed screen. The drawing algorithm used to draw the selected sequence is formed from a loop, starting from the time position at the top of the screen, finishing at the time position at the bottom of the screen. An inner loop counts through each audio channel. This looping method means that the sorted list of sequence points will be in iteration order.

Firstly, for every iteration of the outer loop, the time value is drawn down the left hand side by drawing over the previous time value and drawing the new time value represented in hexadecimal. Every fourth value is drawn in a lighter shade. Each channel is then drawn across the screen within the inner loop by drawing over the previous item and then redrawing the sequence point at that position if one exists at that point. Every time a sequence point is found, the drawing process iterates onto the next item.

The current cursor position is found in the centre of the screen, where the background is drawn in a different colour. If the looped drawing position is out of range of the sequence time range, then just the background is drawn in a lighter shade, with no time value drawn down the left hand side and no sequence point items drawn.

A valid sequence point is drawn with a representation of the sample ID, note and octave used. A null sample will be drawn as a horizontal line, which represents a note stop position.

C.6.6 Buffer processes

These processes serve to link the core audio processing system to the outside world. They provide the transferral of data between systems eliminating the dependence on the underlying memory latency.

C.6.6.1 ADC / DAC controller

The external audio interface is formed from a combination 16-bit stereo ADC/DAC chip. Communications with the device is via two serial interfaces, both under control from the chip itself. Communication is synchronous due to the shared system clock of 12 MHz. The audio data is read in and sent out to the external chip using two 64-bit shift registers. The 64 bits contain the stereo sample and 32 bits of control data. The shift register process that interfaces with the external chip links with the audio input FIFO buffer and the audio output FIFO buffer.

C.6.6.2 Input audio FIFO

There are 16 memory locations available for stereo samples within this process. The ADC shift register controller process feeds the buffer. The core audio process reads data from the buffer. The audio process only uses the input buffer when recording input samples. When it does this, it attempts to keep the buffer as empty as possible.

C.6.6.3 Output audio FIFO

The buffer size is the same as the input FIFO buffer. Samples are read from the buffer by the DAC shift register controller process. The core audio process writes data samples into the buffer. The audio process always outputs data to the buffer unless in the serial port download mode. The core audio process attempts to keep the buffer as full as possible.

C.6.6.4 Serial port input FIFO

This buffer is only used when downloading data from the serial port. It is capable of storing 16 serial data words (8-bits each). The buffer is filled when new data appears from the serial port receiver controller. The core process only reads data from the buffer when serially downloading data and tries to keep the buffer as empty as possible when downloading.

C.6.6.5 Multi-chip Synchronisation

As the system has been partitioned into three separate boards, each with a separate FPGA containing the local configurations. The timing of the start-up of each system is not guaranteed at any time. Each system has its own resetting mechanism due to various communication problems encountered. Instead of synchronising the resetting mechanisms, a system to guarantee the synchronisation to a particular time point within each design has been created. This allows the communications between the systems to be set up in the correct order. A process exists within the tracker to control this synchronisation, which makes use of the FPGA programming signals.

C.7 Expression evaluator demo

The expression evaluator design is given a basic data structure and user guide overview within Chapter 6, Section 6.3. This appendix section gives a more detailed description of the methodology used within the behavioural VHDL source code to produce the expression evaluator.

C.7.1 Code Implementation

This design was produced as a single process system, so all actions occur in sequence with no concurrent process communication. The partitioning of the underlying system required the same synchronisation as found within the tracker design. This was added with a simple controller process that only allows the main process to continue once the correct synchronisation conditions are satisfied.

The point of the demonstrator is to show the use of recursion within a hardware design. With this in mind, four of the expression modification procedures are created with a recursive implementation.

C.7.2 Data structures

The two main data structures within the system are designed to hold the dynamic coloured text log and to hold a representation of the expression itself. These are dynamic structures allocated using the same heap manager system as used within the tracker design.

C.7.2.1 Dynamic log structure

The dynamic coloured text log is created from a linked list structure that is contained within a record type structure. The record contains next and previous linked list record references that form the links in the list and a reference to a full line of text. A line of text is defined as an array of characters of fixed length (78 characters), where each character is held within a 12-bit representation; the most significant 4 bits containing the character colour and the least significant 8 bits containing a standard ASCII character representation.

The text log will be used to output a continuous stream of text that represents the actions performed on the expression at the various recursion depths. The entire log will be available, even when it cannot be viewed fully on the screen. This is the reason for the text storage of the log. The user will be able to scroll up and down the log, viewing the full event history.

C.7.2.2 Expression binary tree structure

The expression is held within a single record type structure as described within Chapter 6.3.1.1. The record contains left and right child operand references which enable a binary tree to be built and the operation to be performed upon the operands. The tree node also holds an integer value that is used for holding the leaf values and the transitory expression results for each level of the tree. The transitory results are used within the recursive expression drawing and evaluation procedures.

C.7.3 Text log procedures

The log is built from the data structure explained within C.7.2.1, but is requires functionality to create, modify, draw, scroll and delete it. The log is filled from various calls to the log manipulation procedures made from the various tree modification procedures and other user input. The log is created by first creating a line to draw into, then drawing all the relevant text information on that line in character order then finishing the line with an end-of-line (EOL) character. New lines are only created when required, with the log starting as initially empty. Text drawing and insertion occurs on a character-by-character basis with the use of higher level drawing procedures.

C.7.3.1 Full log redraw

The only time that the full log text requires a redraw is during a scroll through the log and when a new line is created that pushes the top line out of viewable range.

The log is drawn within two nested loops, the outer loop working upwards through the log lines, iterating from the current bottom scroll line position and the inner loop used to draw the character on each line. A by-product of redrawing the log is to reset the top scroll line position as the last viewable line. This reference is used within the scrolling procedure. Overwriting the background for every line with the background colour draws each line, followed by the printing of each stored character within the line, starting from the left character and working to the right until the EOL character or the line limit is found. Each character is drawn in its stored colour by resetting the drawing colour before each character is printed.

C.7.3.2 Line creation

A new line must be created before any new text is drawn. The line creation procedure firstly physically allocates the memory required for the linked list iterator record and the lines character array. The reference to the line array is stored within the linked list iterator. The linked list iterator is then inserted at the head of the linked list by reassigning the head reference of the list and linking the new iterators previous reference to the old head. If the list is empty, then the list base reference is also assigned to the new iterator. The bottom scroll position is also set to reference the newly created line iterator.

If the number of lines within the log exceeds the number of viewable lines, then a call to the full log redraw procedure is made. The log is redrawn from the bottom scroll reference, which has been set to point at the new line. The redraw resets the top scroll reference.

Finally, the current cursor position is set as the first character within the new line, ready for the text insertion procedures to use the cursor. The first character also has the EOL character written to it in case the line has no text inserted before the next new line creation.

C.7.3.3 Text insertion

When the log is edited on a single character basis, the drawing always occurs at the bottom of the log at the current cursor position, which is always visible when the log is in the process of being modified. This means that each character is drawn separately without the need for drawing the entire log.

The single character insertion modifies the character at the current cursor position. The cursor position is incremented after the modification. The character is drawn using the current drawing colour, which is set dependent on the information being drawn. The character is both drawn to the screen and inserted into the log line array.

C.7.3.4 Drawing strings

The only strings to be entered into the log are derived from a constant source. These are stored within an internal ROM. The colours used for each string are also stored within a ROM. The string drawing procedure uses the contents of the two ROMs to insert the constant characters of the ROM into the dynamic log at the current cursor position, one character at a time. The log insertion colour is reset to the correct colour before any characters are inserted and the string within the ROM is selected by a starting index into the ROM. Character insertion is performed within a loop that terminates when a null character is found within the ROM contents. The character insertion procedure is used for each character.

C.7.3.5 Drawing integers

The drawing of integers is performed in two stages. The first stage converts the internal 32-bit binary representation into a 40-bit BCD representation. This conversion provides a maximum of 10 characters to draw. The number drawn will be in base-10 format. The second stage is to insert the converted text representation of the decimal numbers contained within the result of the conversion into the log.

The conversion of an integer representation into a BCD representation is formed from a hardware implementation of the algorithm found in [119]. The algorithm is implemented with the BCD representation being stored within a RAM array, which allows a very compact storage space, hence size for the conversion procedure.

The converted number is inserted into the log by looping from the most significant BCD number down to the least significant. Drawing only begins once a number other than zero is found for each BCD value. The conversion of the zero offset number into an ASCII representation is made with the addition of an offset of 48, which is the ASCII number character '0'.

C.7.3.6 Scrolling up and down the log

The contents of the log may only be scrolled once all log modification has ceased and the contents of the log spans more than the viewable number of lines on the screen. The scrolling procedure is passed a number of lines to scroll by and the direction to scroll.

The first stage is to adjust the top and bottom scroll pointers through the linked list of log lines by the number of lines to scroll by. If the limits of the log are found in the direction of scrolling, then the scroll pointers are not modified.

The second stage is to redraw the log contents at the current scrolling position. This is performed by a simple call to the full log redraw procedure.

C.7.3.7 Dynamic log erasure

The user may erase the log contents. The procedure to perform this operation simply iterates through the linked list, deleting each line array and linked list iterator element. After every line has been deleted, the linked list base and head pointers are reset and the full log background is drawn over, which removes all traces of the log from the screen.

C.7.4 Tree modification and recursion

The four recursive procedures within the expression evaluator design are explained within the following sections. The first procedure is a recursive implementation of the factorial calculation. This calculation was used within testing. All of the procedures have had a log output included, which shows the recursive calculations being made with the use of indented lines within the log. Each level of indent represents a single level of recursion depth. The indent value is adjusted as a global variable within each procedure so that it does not require stack storage. The three other procedures relate to the modification, drawing and deletion of the binary tree that holds the expression.

C.7.4.1 Factorial evaluation

The factorial of a number is a unary operation. It takes a single operand and returns a single result. It is just as easy to calculate the factorial iteratively as it is recursively. In this respect, the iterative version is a preferable implementation method. However, the recursive implementation was used during recursion testing and has been included within the expression evaluator because of this.

The basic algorithm has had a number of calls to the log output procedures added. These give a textual representation of the input of the factorial procedure while recursing up the stack and give the result at each level while recursing back down through the call stack.

The algorithm first tests the inputted value for being above 1. If the inputted value is 1 or less, then a result of 1 is returned. If the inputted value is greater than 1, the factorial procedure is called recursively, passing the inputted value subtracted by 1. The value returned by the recursive call is then multiplied by the inputted value and returned from the current factorial call. This results in the factorial result that is formed from the multiplication of a number by the set of numbers less than it, down to the number 1.

C.7.4.2 Recursive expression evaluation

The main expression evaluator procedure passes the single expression tree node reference as an 'inout' parameter, meaning that the given node is both taken as an input, capable of modification and returned as an output. If the passed parameter is null, then the reference does not contain a tree node. In this case, a new node is allocated for it before the main expression evaluation loop is entered. The default settings for a new node are that both child references are set to null and the operation is set as a simple leaf value container, whose value is set to zero.

The core expression evaluation is made within the user interface loop. Firstly, the expression is evaluated with the calculation of the result of the operation held by the given node. This simply uses the results held within the child operands (if they exist) and the operation type within the given node. If any child does not exist, then the operand is taken to equal zero. Secondly, the binary tree is drawn from the current node position by the recursive drawing procedure and the result of the expression printed in the log.

Then, user input via the keyboard is required. The blocking interface procedure is called, which waits for a downward-keystroke. A number of actions to modify the tree node dependent on the key pressed are then undertaken, including the creation, insertion, swapping and deletion of child operands, the recursive evaluation of the child operands (which is formed from a recursive call to this expression evaluation procedure, passing a reference to the relevant child node), the recursive return to the parent expression evaluation procedure (or the root calling process) and the modification of the type of operation at the currently referenced tree node.

Each keyboard action will result in a loop back to the beginning of the user interface loop, with the expression being re-evaluated, drawn and logged.

C.7.4.3 Recursive expression tree drawing

The drawing of the expression tree from the current tree node being evaluated is performed once per user interface loop iteration. The recursion process only recurses to a depth of four node levels due to the limited screen space available. The tree is drawn in a depth first manner, with the local node being drawn before recursing into the child operand nodes. The tree is drawn with the currently edited node being drawn at the root of the tree. Child nodes are drawn below the given node, with left operands drawn to the left. The recursion depth determines the horizontal distance between the nodes.

The contents of the node are drawn first, with the box representing the node having the operation and value drawn within it. Then the link joining the given node to its parent node is drawn using a number of characters. Finally, both the left and right operands are recursively drawn only when drawing space is available. If no space is available and child operands exist, then linking stubs are drawn, indicating further tree nodes exist.

C.7.4.4 Recursive expression deletion

The recursive tree deletion procedure simply checks for a null tree item first, returning straight away when found. The deletion operation is logged and then the deletion procedure is recursively called for both the left and right operands. Finally, the local node is deleted and the deletion procedure recursively returns. This deletes an entire branch of the tree in a depth first traversal.

Appendix D

File formats

This appendix explains the format of the two main data files that contain a representation of the user's design within the MOODS synthesis environment. The first, extended ICODE (Intermediate CODE) is generated from the VHDL compiler and now includes structures that enable recursion. The second, DDF (Design Data Format), is a representation of the entire MOODS internal data structures. This file fully contains all information represented within the initial ICODE file taken as input to MOODS.

D.1 BNF descriptions

The Backus-Naur Format (BNF) is used to describe the full syntax of a parsed language. It does not describe what the language means. The descriptions of the ICODE and DDF file format both use BNF notation (Sections D.2.2 and D.3.2).

The notation is built from a number of base descriptor names starting from the root file descriptor. The names are listed in alphabetical order. Each descriptor has a left and right part, with the left part containing the descriptor name (a single string without space that can contain underscores) and the right part containing an expression that can hold references to other descriptors and physical text:

Descriptor_name ::= expression (descriptor_links, text)

Any text links are given in **bold** font, with single text characters bounded by single quotes. Links to other descriptors are given in normal font with the possibility of extra prepended *italicised* use information, separated by underscore characters:

descriptor_link = *use_info*_descriptor_name

The right hand expression is generated from a number of basic constructs that operate on both text and descriptor link items. An expression can contain a sequence of items and formatting constructs:

- Item1 Item2 Item3 = sequence of items
- Item1 | Item2 = only one item is used (could have more than one | operator)
- { ITEM } = zero or more items, following each other in sequence
- [ITEM] = the item is optional

D.2 ICODE

The ICODE file is a textual representation of the user's design that has passed through the original source compiler. The file is a language independent representation of the original source code, which allows different languages to be used as input to the MOODS synthesis system. At present, the only compiler is the original VHDL compiler.

The style of the ICODE file is of a sequential set of instructions, each with an activation list defining the flow of control through the instructions. If no activations exist for an instruction, then the following instruction is implied as the only activated instruction. Multiple activations can exist from a single instruction, which defines either a concurrent branch or a conditional branch in the control flow.

The instructions are held within a container '*module*', with the root module defining the '*program*'. Each module has a header that is described in terms of a single header ICODE instruction. The header includes all the I/O associated with the module. The list of translated register variables, memory variables, ports, aliases and temporary variables follow the header instruction. The port definitions map directly onto the I/O list of the header instruction and define the direction of the port.

A general variable is contained within an ICODE '*register*'. This defines a single storage space for user data. Arrays of data can be held within '*ram*' definitions, which require an address index into the array whenever used. Constant arrays are held within '*rom*' definitions and have the contents of the ROM follow the ROM variable definition. A ROM can be read from only, never written. The index address is provided in the same way as the RAM address.

The names used for variables and module names can be made from any unreserved combination of alphanumeric characters (including single underscores ('_') but not including ICODE delimiters), reserved names being the ICODE instruction keywords. The names are case insensitive.

Each module defines a completely contained subprogram that can be called from any other module. The addition of procedural recursion allows these modules to call each other within recursive loops. A recursive definition requires extra information in the form of a stack declaration and modification instructions. These are added as auto-generated items from the compiler.

All ICODE instructions act upon the variables, ports and aliases defined within the module. All submodules can access the variables within the root module and the variables contained within the module itself. The root module can only access the variables defined within itself.

Each ICODE instruction line has the general form:

```
<label> INSTR <input list>, <output list> <activations> <info>
```

The label is optional and is used to provide a reference for any activations contained after other instructions. The instruction can either be a built-in instruction or be separately defined within a configuration file. The input list and output list contain references to the variables being acted upon. These lists are comma separated with the number of inputs and outputs being defined by the instruction. The activations reference the initial labels defined before other ICODE instruction. The info field after the instruction defines any related information such as original source line number references and activation probabilities for conditional instructions. Each instruction is contained within one line unless separated into multiple lines using the '\' delimiter.

The control flow of a design is handled by a small set of special ICODE instructions. These inbuilt ICODE instructions are listed in Table D.1.

Special Control Instruction	**Description**
IF and IFNOT	These instructions take a single input variable of a single bit and conditionally take the true branch defined by the ACTT activation if the input bit is '1' for IF or '0' for IFNOT. The false branch defined by the ACTF activation is taken as the alternative.
SWITCHON and DECODE	Both of these instructions test a single input variable of known width for a set of alternative values. The DECODE instruction is used in cases where all alternatives are tested, and the SWITCHON instruction is used for a limited set test. Each case alternative is held on a separate line of the file in increasing sequential order. The SWITCHON instruction will include a default case that forms every alternative to the values tested. A separate activation list defined by ACT is given for each alternative.
COUNT and COUNTDN	These instructions operate on special counter variables that will be mapped onto physical counters. The COUNT instruction increments the counter, while the COUNTDN instruction decrements the counter. The counter value is tested for the loop limit before the counter modification and a true or false conditional branch taken at this point.
COLLECT	COLLECT instructions operate on the return of control flow from multiple concurrent threads. The use of this instruction has been disallowed in the present compiler incarnation due to all branches being conditional apart from the process concurrent branches, which never re-converge. The COLLECT instruction will wait for a fixed number (defined as an input to the instruction) of activations to activate it before it will pass control onto the following instruction.

Special Control Instruction	Description
MODULEAP and RECURSE	These two instructions form the calling method between modules and the root program. The MODULEAP instruction forms the non-recursive call and the RECURSE instruction performs a call within a recursive loop. Both instructions take the module name of the module being called, along with the map of I/O parameters to pass into the parameter list of the defined module. The call instructions activate the first instruction within the called module, only returning control to the instruction after the call instruction when the ENDMODULE instruction of the called module is reached. The RECURSE instruction also passes a constant return address reference value implicitly used within the controlling state machine.

Table D.1 Special control instructions

The ICODE file begins with a list of original source files using a comment declaration. This defines a map of unique integer identifiers to the full path specification of the original file. The identifier is used within the source file information cross-references for each ICODE instruction. A general comment is defined using the C++ delimiter '//', and a file information comment is defined using '//F'.

D.2.1 Example ICODE file with recursion

An example of an ICODE file is given in Listing D.1. The file contains two modules, one of which is recursive. The additions made for recursion can be seen in the variable lists of the root module, the initial setup of the stack pointer and return addresses within the root module and the extra ICODE instructions that modify the stack, return address and I/O ports around each recursive call within the recursive module.

Listing D.1 Example ICODE file

```
1   // ***** Extended ICODE Design File (ACW format 25/4/96)
2   // ***** Generated by Vhdl2IC (v 1.9.5) on Tue Apr 10 13:01:43 2001
3   // ***** from recursion_test1.vhd dated Tue Mar 27 17:27:58 2001
4   // ***** Adding VHDL line number comments (DJDM 17/02/00)
5
6   //F 1 "c:\djdm\larch_e\moods\library\standard.pck"
7   //F 2 "c:\djdm\larch_e\moods\library\packages.vhd"
8   //F 3 "C:\DJDM\LARCH_E\vhdl\recursion_test\recursion_test1.vhd"
9
10  // ***** Main program "recursion_test1" declaration *****
11  PROGRAM          recursion_test1     input, output     {ln:4, pos:1, file:3}
12
13  // ***** I/O port declarations
```

```
14  inport          input              [0:31]    {ln:6, pos:5}
15  outport         output             [0:31]    {ln:7}
16
17  // ***** Variable/register declarations
18  register        ret                [0:31]    {ln:71, pos:14}
19  register        factorial_ra       [0:0]     {file:0}
20  register        factorial_inval_in [0:31]    {ln:58, pos:7, file:3}
21  register        factorial_outval_out [0:31]  {ln:59}
22  register        stack_pointer      [0:7]     {file:0}
23  ram             stack              [0:31]    address   [0:255]
24
25                  move               #0,  stack_pointer
26                  move               #%0,  factorial_ra
27
28
29                  // ********** process PR1 **********
30  .PR1            MODULEAP           factorial #5, ret    {ln:82, pos:5, file:3}
31                  move               ret, output          {ln:83, pos:12}
32                  MODULEAP           factorial input, ret {ln:84, pos:5}
33                  move               ret, output          {ln:85, pos:12}
34                  move               #12:32, ret          {ln:86, pos:9}
35                  protect                                 {ln:87, pos:5}
36                  move               ret, output  ACT PR1 {ln:88, pos:12}
37                  // ********** end process PR1 **********
38                  ENDMODULE          recursion_test1      {ln:92, pos:1}
39  // ***** End of main program "recursion_test1" declaration *****
40
41
42  // ***** Recursive Module "factorial" declaration *****
43  RECMODULE       factorial          factorial_ra inval, outval {ln:56, pos:5}
44
45  // ***** I/O port declarations
46  inport          inval              [0:31]    {ln:58, pos:7}
47  outport         outval             [0:31]    {ln:59}
48
49  // ***** Variable/register declarations
50  register        local              [0:31]    {ln:61, pos:16}
51  // temp         121                [0:32]
52  // temp         123                [0:0]
53
54                  eq                 inval, #1:32, 123            {ln:63}
55                  if                 123 ACTT if14_true_15 ACTF if14_false_16
56                                                {pt:0.500000, pf:0.500000, pos:7}
57  .if14_true_15   move               #1:32, outval ACT label32 {ln:64, pos:16}
58  .if14_false_16  minus              inval, #1:32, 121         {ln:66, pos:25}
59                  memwrite factorial_inval_in, stack[stack_pointer]     {pos:9}
60                  plus               stack_pointer, #1,  stack_pointer
61                  move               121,  factorial_inval_in
62                  memwrite           factorial_ra,  stack[stack_pointer]
63                  plus               stack_pointer, #1,  stack_pointer
64                  move               #%1,  factorial_ra
65                  RECURSE factorial #%1 factorial_inval_in,  factorial_outval_out
66                  minus              stack_pointer, #1,  stack_pointer
67                  memread            stack[stack_pointer],  factorial_ra
68                  protect
69                  minus              stack_pointer, #1,  stack_pointer
70                  memread            stack[stack_pointer],  factorial_inval_in
71                  protect
72                  move               factorial_outval_out,  local
73                  mult               local, inval, outval      {ln:67, pos:25}
74  .label32        ENDMODULE          factorial                 {ln:69, pos:5}
75  // ***** End of recursive module "factorial" declaration *****
```

D.2.2 ICODE grammar in BNF form

Extended ICODE description ::=
{ filemap_comment }
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

binary_integer ::=
'**%**' binary_integer_val { binary_integer_val }

binary_integer_val ::=
'**0**' | '**1**'

call_inst ::=
moduleap_inst
| recurse_inst

collect_inst ::=
COLLECT *collect_count*_integer [info]

conditional_inst ::=
conditional_inst_name *cond*_var actt_list actf_list [info]

conditional_inst_name ::=
IF | **IFNOT**

constant ::=
'#' integer [':' *width*_decimal_integer]

count_inst ::=
count_inst_name *counter*_var ',' *increment*_constant ',' *end*_term actt_list actf_list

count_inst_name ::=
COUNT | COUNTDN

counter_declaration ::=
counter_type *counter*_name *counter*_range

counter_type ::=
COUNTER | COUNTDN

decimal_integer ::=
decimal_integer_val { decimal_integer_val }

decimal_integer_val ::=
'**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**'

declaration ::=
io_port_declaration
| variable_declaration

declaration_part ::=
{ declaration [info] }

decode_inst ::=
DECODE *decode*_var [info]
{ **CASE** constant actt_list [info] }

filemap_comment ::=
//**F** *file_id*_decimal_integer '"' *file_full_path*_string '"'

float ::=
decimal_integer '.' decimal_integer ['**e**' decimal_integer]

general_inst[1] ::=
general_inst_name io_list [actt_list] [info]

---

[1] General instructions are defined in the ICODE instruction database and may be enhanced as required.

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**

hex_integer ::=
'**$**' hex_integer_val { hex_integer_val }

hex_integer_val ::=
'**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**' | '**A**' | '**B**' | '**C**' | '**D**' | '**E**' | '**F**'

info ::=
['{' info_item { ',' info_item } '}']

info_item ::=
info_specifier '**:**' info_value

info_specifier ::=
ln | **pos** | **file** | **pt** | **pf** | **its**

info_value ::=
decimal_integer | float

instruction ::=
general_inst
| memory_inst
| count_inst
| conditional_inst
| switch_inst
| decode_inst
| collect_inst
| call_inst

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]

name ::=
    string

octal_integer ::=
    '**&**' octal_integer_val { octal_integer_val }

octal_integer_val ::=
    '**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**'

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*_integer ':' *high_bit_index*_integer ']'

recurse_inst ::=
RECURSE *recmodule*_name *return_address*_constant io_list [info]

register_declaration ::=
REGISTER *var*_name *var*_range

rom_declaration ::=
ROM *rom_var*_name *data*_range **ADDRESS** *address*_range

string ::=
string char { string_char }

string_char ::=
'a' | **'b'** | **'c'** | **'d'** | **'e'** | **'f'** | **'g'** | **'h'** | **'i'** | **'j'** | **'k'** | **'l'** | **'m'** | **'n'** | **'o'** | **'p'** | **'q'** | **'r'** |
's' | **'t'** | **'u'** | **'v'** | **'w'** | **'x'** | **'y'** | **'z'** | **'A'** | **'B'** | **'C'** | **'D'** | **'E'** | **'F'** | **'G'** | **'H'** | **'I'** |
'J' | **'K'** | **'L'** | **'M'** | **'N'** | **'O'** | **'P'** | **'Q'** | **'R'** | **'S'** | **'T'** | **'U'** | **'V'** | **'W'** | **'X'** | **'Y'** |
'Z' | **'.'** | **'/'** | **'\'** | **'_'** | **'-'**

submodule_declaration ::=
submodule_normal | submodule_recursive

submodule_normal ::=
MODULE *module*_name io_list [actt_list] [info]
declaration_part
process_part
ENDMODULE [*module*_name] [info]

submodule_recursive ::=
RECMODULE *module*_name *return_address*_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

var ::=
    *var*_name | *temporary_variable*_decimal_integer

variable_declaration ::=
    register_declaration
    | alias_declaration
    | ram_declaration
    | rom_declaration
    | counter_declaration

D.3 DDF

The Design Data Format (DDF) file is a direct representation of the full internal data structures used within MOODS. These data structures are explained within Appendix A.5, which describes the program 'DDFLink'. The DDF file can be generated at any time within the synthesis process. Two types of represented data can be contained, with the second type containing all the additional logic generated within the post-processing step. It is the second type that is used within DDFLink. The first type is just a subset of the second type, without the additions made after synthesis.

D.3.1 Example DDF file with recursion

The main sections within the DDF file are the file list, module declarations, data path definition, conditional signal definitions and the module library link.

The file list is used to determine a cross-reference back to the originating source files and ICODE file. A file ID that is used elsewhere within the DDF file references the names.

The list of modules contains the full ICODE description as found within the ICODE file. In addition to this, the ICODE instructions are placed into control states, which operate within one time-step. These control states are held in a subsection of the module. The header ICODE instruction forms the header subsection of each module. The list of variables is held in a separate subsection.

The data path for the full design is held within a single data path section. This section contains all references to the underlying implementing cells and the linkages between these cells in the form of nets.

The condition list is also held within a single section after the data path. This section contains the list of all Boolean equations that form the link between the data path and the various control paths defined within the list of modules.

The final section within the DDF file is the module library section. This section is used for linking technology specific information derived from a particular module library into the DDF file description. A separate module library parser that does not need to know about the DDF files data format reads the data held within this section.

Examples of these sections are shown in Listing D.2 below. The version number at the head of the file determines the present version of the DDF file. This is present so that the DDF parser can load only up to date files. The listing has been cut down for brevity reasons and hence does not contain a complete design.

Listing D.2 Example Design Data Format file

```
ddf_version : 100;

file_list
{
  file f1 : "c:\djdm\larch_e\moods\library\standard.pck";
  file f2 : "c:\djdm\larch_e\moods\library\packages.vhd";
  file f3 : "C:\DJDM\LARCH_E\vhdl\recursion_test\recursion_test2.vhd";
  file f4 : "C:\DJDM\LARCH_E\vhdl\recursion_test\recursion_test2.xic";
}

module m6
{
  header {
    instruction i6 {
      icode : recmodule func;
      input : iv;
      output : ov;
      prob : 0;
      sourcepos : f3,16,5;
      icodepos : f4,36,10;
    }
    end : s63;
  }
  variables
  {
    port v8 : iv[0:3] is u8 sourcepos f3,18,7 icodepos f4,39,7;
    register v9 : ov[0:3] sourcepos f3,19,7 icodepos f4,40,8;
    register v10 : local[0:3] sourcepos f3,21,16 icodepos f4,43,9;
    temp v20 : var_s105[0:0];
  }
  control_path  c7 : c7, c18
  {
    conditional c7 {
```

```
34          cell : 6;
35          signal : s19;
36          prob : 0;
37          group g13 {
38            instruction i7 : eq iv, #1:4, var_s105
39                sourcepos f3,23,13 icodepos f4,46,23;
40            instruction i8 : if var_s105
41                sourcepos f3,23,7 icodepos f4,47,23;
42            instruction i13 : move #1:1, func_ra when s22 prob 0.5
43                sourcepos f3,26,9 icodepos f4,52,25;
44            instruction i12 : plus stack_pointer_1, #1:0, stack_pointer_1 when s22
45                prob 0.5 sourcepos f3,26,9 icodepos f4,51,25;
46            instruction i11 : memwrite func_ra, stack_1[stack_pointer_1] when s22
47                prob 0.5 sourcepos f3,26,9 icodepos f4,50,29;
48            instruction i9 : move #1:4, ov when s20 prob 0.5
49                sourcepos f3,24,12 icodepos f4,48,25;
50            instruction i10 : minus iv, #1:4, func_iv_in when s22 prob 0.5
51                sourcepos f3,26,17 icodepos f4,49,26;
52          }
53          activate a13 : c14 when s22 prob 0.5;
54        }
55        recurse c14 {
56          cell : 13;
57          signal : s28;
58          module : m6;
59          prob : 0;
60          instruction i14 {
61            icode : recurse func;
62            input : func_iv_in;
63            output : func_ov_out;
64            prob : 0.5;
65            end : s64;
66            sourcepos : f3,26,9;
67            icodepos : f4,53,28;
68          }
69          activate a14 : c15;
70        }
71        general c15 {
72          cell : 14;
73          signal : s30;
74          prob : 0;
75          group g16 {
76            instruction i15 : minus stack_pointer_1, #1:0, stack_pointer_1 prob 0.5
77                sourcepos f3,26,9 icodepos f4,54,26;
78            instruction i16 : memread stack_1[stack_pointer_1], func_ra prob 0.5
79                sourcepos f3,26,9 icodepos f4,55,28;
80          }
81          instruction i17 : protect prob 0.5 sourcepos f3,26,9 icodepos f4,56,28;
82          activate a17 : c18;
83        }
84        general c18 {
85          cell : 17;
86          signal : s33;
87          prob : 0;
88          group g19 {
89            instruction i18 : move func_ov_out, local prob 0.5
90                sourcepos f3,26,9 icodepos f4,57,25;
91            instruction i19 : plus local, #1:4, ov prob 0.5
92                sourcepos f3,27,21 icodepos f4,58,25;
93          }
94        }
95      }
96    }
97
98    module m1
99    {
100     header {
101       instruction i1 {
102         icode : program recursion_test2;
```

```
103         input : di;
104         output : do;
105         prob : 0;
106         sourcepos : f3,4,1;
107         icodepos : f4,11,8;
108       }
109     }
110     variables
111     {
112       port v1 : di[0:3] is u1 sourcepos f3,6,5 icodepos f4,14,7;
113       register v2 : do[0:3] is u2 sourcepos f3,7,5 icodepos f4,15,8;
114       register v3 : func_ra[0:0] is u3 icodepos f4,18,9;
115       register v4 : func_iv_in[0:3] is u4 sourcepos f3,18,7 icodepos f4,19,9;
116       register v5 : func_ov_out[0:3] is u5 sourcepos f3,19,7 icodepos f4,20,9;
117       register v6 : stack_pointer_1[0:3] is u6 icodepos f4,21,9;
118       ram v7 : stack_1[0:31] address [0:15] is u7 icodepos f4,22,4;
119       register v36 : func_ra_decode[0:1] is u21;
120     }
121     control_path  c2 : c4
122     {
123       general c2 {
124         cell : 3;
125         signal : s12;
126         prob : 0;
127         instruction i2 : move #0:0, stack_pointer_1 icodepos f4,24,25;
128         instruction i3 : move #0:1, func_ra icodepos f4,25,25;
129         activate a3 : c4;
130       }
131       call c4 {
132         cell : 5;
133         signal : s14;
134         module : m6;
135         prob : 0;
136         instruction i4 {
137           icode : moduleap func;
138           input : di;
139           output : do;
140           sourcepos : f3,33,5;
141           icodepos : f4,29,29;
142         }
143         feedback a4 : c4;
144       }
145     }
146   }
147
148   data_path
149   {
150     port u1 {
151       width : [0:3];
152       cell : 20;
153       net n40 {
154         source : pin output1[0:3] is v1;
155         links : i4;
156         condition : s56;
157         destination : pin input1[0:3] @ u18 is v8 on i4;
158       }
159     }
160     storage u2 {
161       width : [0:3];
162       cell : 21;
163       control : pin load_en[3:3] when s46;
164       control : pin load_en[2:2] when s46;
165       control : pin load_en[1:1] when s46;
166       control : pin load_en[0:0] when s46;
167     }
168     storage u3 {
169       width : [0:0];
170       cell : 22;
171       instruction : i13, i3;
```

```
172       control : pin load_en[0:0] when s50;
173       net n41 {
174         source : pin output1[0:0];
175         destination : pin input1[0:0] @ u21;
176       }
177       net n41 {
178         source : pin output1[0:0] is v3;
179         links : i11;
180         destination : pin input1[0:0] @ u7 is v7;
181       }
182     }
183     memory u7 {
184       width : [0:31];
185       cell : 26;
186       instruction : i16, i11;
187       control : pin write[0:0] when s48;
188       control : pin read[0:0] when s49;
189       net n45 {
190         source : pin output1[0:0] is v7;
191         links : i16;
192         condition : s49;
193         destination : pin input1[0:0] @ u10 is v3 on i16;
194       }
195     }
196     port u8 {
197       width : [0:3];
198       cell : 27;
199       net n46 {
200         source : pin output1[0:3] is v8;
201         links : i10;
202         condition : s48;
203         destination : pin input1[0:3] @ u19 is v8 on i10;
204       }
205       net n46 {
206         source : pin output1[0:3] is v8;
207         links : i7;
208         destination : pin input1[0:3] @ u12 is v8;
209       }
210     }
211     interconnect u9 {
212       width : [0:3];
213       cell : 51;
214       net n47 {
215         source : pin output1[0:3];
216         destination : pin input1[0:3] @ u2;
217       }
218     }
219     functional u13 {
220       width : [0:3];
221       cell : 39;
222       instruction : i10, i15, i19;
223       control : pin select[0:0] select 14 on i19 when s40;
224       control : pin select[0:0] select 15 on i10 when s55;
225       net n50 {
226         source : pin output1[0:3] is v9;
227         links : i4, i19;
228         condition : s57;
229         destination : pin input1[0:3] @ u9 is v2 on i19;
230       }
231       net n50 {
232         source : pin output1[0:3] is v9;
233         links : i14, i19;
234         condition : s52;
235         destination : pin input1[0:3] @ u15 is v5 on i19;
236       }
237       net n50 {
238         source : pin output1[0:3] is v6;
239         links : i15;
240         condition : s49;
```

```
241         destination : pin input1[0:3] @ u16 is v6 on i15;
242       }
243       net n50 {
244         source : pin output1[0:3] is v6;
245         links : i16, i15;
246         condition : s49;
247         destination : pin input1[0:3] @ u17 is v7 on i15;
248       }
249       net n50 {
250         source : pin output1[0:3] is v4;
251         links : i10;
252         destination : pin input1[0:3] @ u4 is v4;
253       }
254     }
255     net n0 {
256       source : #1 on i9;
257       links : i4, i9;
258       condition : s58;
259       destination : pin input1[0:3] @ u9 is v2 on i9;
260     }
261     net n0 {
262       source : #0 on i3;
263       links : i3;
264       condition : s47;
265       destination : pin input1[0:0] @ u10 is v3 on i3;
266     }
267   }
268
269   condition_list
270   {
271     signal s20 : v20 on n39;
272     signal s22 : /s20 on n38;
273     signal s37 : s37 on n37;
274     signal s38 : s38 on n36;
275     signal s39 : s14 on n35;
276     signal s40 : s33 on n34;
277     signal s41 : (s40.s37) on n33;
278     signal s42 : (s41.s39) on n32;
279     signal s43 : (s19.s20) on n31;
280     signal s44 : (s43.s37) on n30;
281     signal s45 : (s44.s39) on n29;
282     signal s46 : (s42 + s45) on n28;
283     signal s47 : s12 on n27;
284     signal s48 : (s19.s22) on n26;
285     signal s49 : s30 on n25;
286     signal s50 : (s49 + s47 + s48) on n24;
287     signal s52 : (s40.s38) on n22;
288     signal s53 : (s43.s38) on n21;
289     signal s54 : (s52 + s53) on n20;
290     signal s55 : (s48 + s49) on n19;
291     signal s56 : (s37.s39) on n18;
292     signal s57 : (s56.s40) on n17;
293     signal s58 : (s56.s43) on n16;
294     signal s59 : (s40 + s48) on n15;
295     signal s60 : s19 on n14;
296     signal s61 : (s60.s20) on n13;
297     signal s62 : (s40 + s61) on n12;
298     signal s63 : (s37.s62) on n11;
299     signal s64 : (s38.s62) on n10;
300   }
```

D.3.2 DDF file format grammar in BNF form

Design Data Description ::=
version
[file_list]
module_list
data_path
condition_list
[module_library]

activation_definition ::=
activate | **feedback** arc_number '**:**' control_number
[**when** *condition*_signal_number] [**prob** float] '**;**'

alias_declaration ::=
alias variable_number '**:**' *alias_variable*_name *alias*_range **from** *variable*_name
*variable_sub*_range

arc_number ::=
['**a**'] decimal_integer

binary_integer ::=
'**%**' binary_integer_val { binary_integer_val }

binary_integer_val ::=
'**0**' | '**1**'

bool_and ::=
bool_term { '**.**' bool_term }

boolean_expression ::=
bool_term | '**(**' bool_and | bool_or '**)**'

bool_or ::=
bool_term { '+' bool_term }

bool_term ::=
'/' bool_term | signal_number | variable_number

condition_list ::=
    condition_list '{'
        { signal_definition }
    '}'

constant ::=
    '#' integer [':' *width*_decimal_integer]

constant_list ::=
    constant { ',' constant }

control_block ::=
    control_path *start*_control_number ':' *end*_control_number_list '{'
        { control_node_definition }
    '}'

control_definition ::=
    control | **erased_control** ':' control_end ';'

control_end ::=
    pin *variable*_name *slice*_range
        [**is** variable_number]
        [**when** signal_number]
        [**on** instruction_number]
        [**select** *alu_item*_decimal_integer]

control_node_definition ::=
    control_type control_number control_specification

control_number ::=
    ['**c**'] decimal_integer

control_number_list ::=
    control_number { ',' control_number }

control_parameter ::=
    loop_its ':' decimal_integer ';'
    | **prob** ':' float ';'
    | **signal** ':' signal_number ';'
    | **cell** ':' *cell_reference*_decimal_integer ';'
    | **module** ':' *called*_module_number ';'
    | **end** ':' *call_end*_signal_number ';'

control_specification ::=
'{'
{ control_parameter }
{ group_definition }
{ instruction_definition }
{ activation_definition }
'}'

control_type ::=
general | **fork** | **collect** | **conditional** | **call** | **recurse** | **dot**

counter_declaration ::=
counter | **countdn** variable_number '**:**' *counter_variable*_name *variable*_range

data_path ::=
data_path '{'
{ unit_definition }
{ net_definition }
'}'

decimal_integer ::=
decimal_integer_val { decimal_integer_val }

decimal_integer_val ::=
'**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**'

file_list ::=
file_list '{'
{ file_list_item }
'}'

file_list_item ::=
file file_number '**:**' '"' *file*_string '"'

file_number ::=
['**f**'] decimal_integer

file_position ::=
file_number '**,**' *line*_decimal_integer '**,**' *column*_decimal_integer

float ::=
    decimal_integer '**.**' decimal_integer ['**e**' decimal_integer]

group_definition ::=
    group group_number '{'
        { instruction_definition }
    '}'

group_number ::=
    ['**g**'] decimal_integer

header_definition ::=
    header '{'
        *header*_instruction_definition
        [**end** ':' *module_end*_signal_number ';']
    '}'

hex_integer ::=
    '**$**' hex_integer_val { hex_integer_val }

hex_integer_val ::=
    '**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**' | '**8**' | '**9**' | '**A**' | '**B**' | '**C**' | '**D**' | '**E**' | '**F**'

icode_instruction_name[2] ::=
    eq | **ls** | **le** | **ne** | **ge** | **gr** | **not** | **and** | **or** | **xor**
    | **neg** | **plus** | **minus** | **mult** | **div** | **lshift** | **rshift** | **rol** | **ror**
    | **move** | **settrue** | **highz** | **memread** | **memwrite** | **count** | **countdn**
    | **if** | **ifnot** | **collect** | **decode** | **switchon** | **moduleap** | **recurse**
    | **program** | **module** | **recmodule**

instruction_definition ::=
    instruction instruction_number instruction_specification

instruction_number ::=
    ['**i**'] decimal_integer

instruction_number_list ::=
    instruction_number { ',' instruction_number }

---

[2] The built-in instructions may be enhanced by extra instructions defined in the ICODE instruction database.

instruction_parameter ::=
icode ':' icode_instruction_name ';'
| **input** ':' *input*_io_list ';'
| **output** ':' *output*_io_list ';'
| **condition** ':' signal_number ';'
| **activate** ':' signal_number ';'
| **prob** ':' float ';'
| **mutual_list** ':' instruction_number_list ';'
| **sourcepos** ':' *source*_file_position ';'
| **icodepos** ':' *icode*_file_position ';'
| **end** ':' *recurse_end*_signal_number ';'

instruction_specification ::=
instruction_specification_block_mode | instruction_specification_single_line

instruction_specification_block_mode ::=
'{' { instruction_parameter } '}'

instruction_specification_single_line ::=
':'
icode_instruction_name
io_list
[**when** signal_number]
[**prob** float]
[**sourcepos** *source*_file_position]
[**icodepos** *icode*_file_position]
';'

integer ::=
binary_integer | octal_integer | decimal_integer | hex_integer

io_list ::=
io_list_item { ',' io_list_item }

io_list_item ::=
*variable*_name | *temporary_variable*_decimal_integer |
*input*_constant | memory_reference

memory_reference ::=
*memory_variable*_name '**[**' *address_variable*_name | *address*_constant '**]**'

memory_declaration ::=
    ram | **rom** variable_number '**:**' *memory_variable*_name *width*_range
        address *address*_range **data** '**[**' constant_list '**]**'

module_definition ::=
    module module_number '{'
        header_definition
        variable_block
        control_block
    '}'

module_library ::=
    module_library '{'
        library_contents[3]
    '}'

module_list ::=
    { module_definition }

module_number ::=
    ['**m**'] decimal_integer

name ::=
    string

net_definition ::=
    net | **erased_net** net_number net_specification

net_end ::=
    pin *variable*_name *slice*_range '**@**' unit_number | constant | signal_number
        [**is** variable_number] [**on** instruction_number]
net_number ::=
    ['**n**'] decimal_integer

net_parameter ::=
    source '**:**' *source*_net_end '**;**'
    | **destination** '**:**' *destination*_net_end '**;**'
    | **links** '**:**' instruction_number_list '**;**'
    | **condition** '**:**' signal_number '**;**'

---

[3] The cell library references are not read with the DDF parser, so are not subjected to this BNF grammar.

net_specification ::=
'{' { net_parameter } '}'

octal_integer ::=
'**&**' octal_integer_val { octal_integer_val }

octal_integer_val ::=
'**0**' | '**1**' | '**2**' | '**3**' | '**4**' | '**5**' | '**6**' | '**7**'

port_declaration ::=
port variable_number '**:**' *port_variable*_name *variable*_range

range ::=
'**[**' *low_bit*_integer '**:**' *high_bit*_integer '**]**'

register_declaration ::=
register variable_number '**:**' *register_variable*_name *variable*_range

signal_defmition ::=
signal signal_number '**:**' signal_specification '**;**'

signal_number ::=
['**s**'] decimal_integer

signal_specification ::=
boolean_expression **on** net_number

string ::=
string char { string_char }

string_char ::=
'**a**' | '**b**' | '**c**' | '**d**' | '**e**' | '**f**' | '**g**' | '**h**' | '**i**' | '**j**' | '**k**' | '**l**' | '**m**' | '**n**' | '**o**' | '**p**' | '**q**' | '**r**' |
'**s**' | '**t**' | '**u**' | '**v**' | '**w**' | '**x**' | '**y**' | '**z**' | '**A**' | '**B**' | '**C**' | '**D**' | '**E**' | '**F**' | '**G**' | '**H**' | '**I**' |
'**J**' | '**K**' | '**L**' | '**M**' | '**N**' | '**O**' | '**P**' | '**Q**' | '**R**' | '**S**' | '**T**' | '**U**' | '**V**' | '**W**' | '**X**' | '**Y**' |
'**Z**' | '**.**' | '**/**' | '****' | '**_**' | '**-**'

temp_declaration ::=
temp variable_number '**:**' *temp_variable*_name *variable*_range

unit_definition ::=
unit_type unit_number unit_specification

unit_number ::=
['**u**'] decimal_integer

unit_parameter ::=
width '**:**' range '**;**'
| **cell** '**:**' decimal_integer '**;**'
| **instruction** '**:**' instruction_number_list '**;**'
| net_definition
| control_definition

unit_specification ::=
'{' { unit_parameter } '}'

unit_type ::=
storage | **functional** | **boolean** | **interconnect** | **port** | **memory**

variable_block ::=
variables '{'
{ variable_declaration }
'}'

variable_declaration ::=
variable_type_declaration
[**is** unit_number]
[**sourcepos** *source*_file_position]
[**icodepos** *icode*_file_position] '**;**'

variable_type_declaration ::=
port_declaration
| register_declaration
| temp_declaration
| alias_declaration
| counter_declaration
| memory_declaration

variable_number ::=
['**v**'] decimal_integer

version ::=
ddf_version '**:**' decimal_integer '**;**'

References

1. Baker, K. R., "Multiple Objective Optimisation of Data and Control Paths in a Behavioural Silicon Compiler", PhD Thesis, University of Southampton, Sept 1992.

2. Baker, K. R. - Currie, A. J. - Nichols, K. G., "Multiple Objective Optimisation in a Behavioural Synthesis System", IEE Proceedings - G, Vol. 140, No. 4, Aug 1993, pp 253-260.

3. Williams, A. C., "A Behavioural VHDL Synthesis System using Data path Optimisation", PhD Thesis, University of Southampton, July 1997.

4. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-1987", IEEE Catalog No. SH11957, 1987.

5. "IEEE Standard VHDL Reference Manual, IEEE Std 1076-1993", IEEE Catalog No. SH16840, 1993.

6. Ecker, W., "Dynamic, Semi-Dynamic and Static Datatypes in VHDL", First International Forum on Design Languages, 1998, Tutorial #3.

7. Camposano, R., "From Behavior to Structure: High-Level Synthesis", IEEE Design and Test of Computers, Vol. 7, No. 5, Oct 1990, pp 8-19.

8. De Micheli, G., "Synthesis and Optimization of Digital Circuits", McGraw-Hill International Editions 1994, ISBN 0-07-016333-2.

9. Gajski, D. D. [editor], "Silicon Compilation", Addison-Wesley 1988, ISBN: 0-201-09915-2.

10. Camposano, R., "Behavioral Synthesis", Design Automation Conference, Ch. 161, Jun 1996, pp. 33-34.

11. Genoe, M. - Vanoostende, P. - Van Wauwe, G., "On the use of VHDL-based behavioural synthesis for telecom ASIC design", International Symposium on System Synthesis, 1995, Session 3.

12. Lin, Y. -L., "Survey Paper: Recent Developments in High-Level Synthesis", ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 1, Jan 1997, pp 2-21.

13. Peng, Z. - Kuchcinski, K., "Automated Transformation of Algorithms into Register-Transfer Level Implementations", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, No. 2, Feb 1994, pp 150-165.

14. Gajski, D. D. - Ramachandran, L., "Introduction to High-Level Synthesis", IEEE Design and Test of Computers, Vol. 11, No. 4, 1994, pp 45-54.

15. Brewer, F. - Gajski, D., "Chippe: A System for Constraint Driven Behavioral Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 9, No. 7, Jul 1990, pp 681-695.

16. 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, Jun 1989, pp 661-679.

17. Kurdahi, F. J. - Parker, A. C., "REAL: A Program for REgister ALlocation", Proceedings of the 24th Design Automation Conference, 1987, pp 210-215.

18. Camposano, R. - Saunders, L. F. - Tabet, R. M., "VHDL as Input for High Level Synthesis", IEEE Design and Test of Computers, Mar 1991, pp 43-49.

19. Rushton, A., "VHDL for Logic Synthesis 2nd Ed.", Wiley, 1998, ISBN: 0-471-98325-X.

20. "Modelsim SE/EE User's manual, Version 5.4c", Model Technology, Jul 2000.

21. "ADA 95 Reference manual", ISO/IEC/ANSI 8652:1995, February 1995.

22. De Micheli, G., "Hardware Synthesis from C/C++ Models", Proceedings of the Design Automation and Test in Europe Conference, 1999, pp 382-383.

23. Wakabayashi, K., "C-based Synthesis Experiences with a Behaviour Synthesiser, "Cyber"", Proceedings of the Design Automation and Test in Europe Conference, 1999, Session 6A, pp 390-393.

24. Ghosh, A. - Kunkel, J. - Liao, S., "Hardware Synthesis from C/C++", Proceedings of the Design Automation and Test in Europe Conference, 1999, Session 6A, pp 387-389.

25. Verkest, D. - Kunkel, J. - Schirrmeister, F., "System Level Design Using C++", Proceedings of the Design Automation and Test in Europe Conference, 2000, Session 2A, pp 74-81.

26. "SystemC Version 1.0 User's Guide", Synopsys Inc., CoWare Inc., Frontier Design Inc., 2000.

27. Balakrishnan, M. - Majumdar, A. K. - Banerji, D. K. - Linders, J. G. - Majithia, J. C., "Allocation of Multiport Memories in Data Path Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 7, No. 4, April 1988, pp 536-540.

28. Park, N. - Parker, A. 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.

29. "The SPARC Architecture Manual, Version 8", SPARC International Inc., 1991.

30. Wilson, P. R. - Johnstone, M. S. - Neely, M. - Boles, D., "Dynamic Storage Allocation: A survey and Critical Review", University of Texas, Department of Computer Science, 1995.

31. Knuth, D. E., "Fundamental Algorithms, Volume 1, on the Art of Computer Programming", Addison-Wesley Publishing Company, 1973.

32. Zorn, B. - Grunwald, D., "Evaluating Models of Memory Allocation", ACM Transactions on Modeling and Computer Simulation, Vol. 4, No. 1, Jan 1994, pp 107-131.

33. Vo, K. -P., "Vmalloc: A General and Efficient Memory Allocator", Software Practice and Experience, Vol. 26, No. 3, Mar 1996, pp 357-374.

34. Grunwald, D. - Zorn, B., "CustoMalloc: Efficient Synthesised Memory Allocators", Software - Practice and Experience, Vol. 23, No. 8, Aug 1993, pp 851-869.

35. Detlefs, D. - Dosser, A., "Memory Allocation Costs in Large C and C++ Programs", Software Practice and Experience, Vol. 24, No. 6, Jun 1994, pp 527-542.

36. Brent, R. P., "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation", ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, pp 388-403.

37. Richter, H. T., "Fast Memory Allocation", Dr. Dobb's Journal, May 1998, pp 78-87.

38. Gontmakher, S. - Horn, I., "Efficient Memory Allocation", Dr. Dobb's Journal, January 1999, pp 116-119.

39. Zorn, B., "The Measured Cost of Conservative Garbage Collection", Software - Practice and Experience, Vol. 23, No. 7, July 1993, pp 733-756.

40. Boehm, H. -J., "Dynamic Memory Allocation and Garbage Collection", Computers in Physics, Vol. 9, No. 3, May/Jun 1995, pp 297-303.

41. Boehm, H. -J. - Weiser, M., "Garbage Collection in an Uncooperative Environment", Software - Practice and Experience, Vol. 18, No. 9, Sept 1988, pp 807-820.

42. Spertus, M., "C++ and Garbage Collection", Dr. Dobb's Journal, Dec 1997, pp 36-41.

43. Camposano, R. - Rosenstiel, W., "Synthesising Circuits from Behavioural Descriptions", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 2, Feb 1989.

44. De Man, H. - Rabaey, J. - Six, P. - Claesen, L., "Cathedral II: A Silicon Compiler for Digital Signal Processing", IEEE Design and Test, Dec 1986, pp 13-25.

45. Peng, Z., "Synthesis of VLSI Systems with the CAMAD Design Aid", Proceedings of the 23rd ACM/IEEE Design Automation Conference, 1986, pp 278-284.

46. Bardsley, A. - Edwards, D. A., "The Balsa Asynchronous Circuit Synthesis System", Proceedings of the 3rd International Forum on Design Languages (FDL), Sept 2000, pp 37-44.

47. Bardsley, A. - Edwards, D. A., "Synthesising an asynchronous DMA controller with Balsa", Journal of Systems Architecture 46, 2000.

48. "Behavioral Compiler User Guide", Version 2000.11, Nov 2000, Synopsys.

49. "Visual Architect", http://www.cadence.com/articles/VisualArc.html, Cadence.

50. "Monet", http://www.mentor.com/monet/, Mentor Graphics.

51. Catthoor, F., "Energy-delay efficient data storage and transfer architectures: circuit technology versus design methodology solutions", Proceedings of the Design Automation and Test in Europe Conference, 1998, Session 9C, pp 709-715.

52. Wehn, N. - Hein, S., "Embedded DRAM Architectural Trade-Offs", Proceedings of the Design Automation and Test in Europe Conference, 1998, Session 9C, pp 704-708.

53. Catthoor, F. - Dutt, N. D. - Kozyrakis, C. E., "Hot topic session: How to solve the current memory access and data transfer bottlenecks: at the processor architecture or at the compiler level?", Proceedings of the Design Automation and Test in Europe Conference, 2000, Session 6B, pp 426-433.

54. Séméria, L. - Sato, K. - De Micheli, G., "Memory Representation and Hardware Synthesis of C Code with Pointers and Complex Data Structures", Computer Systems Lab, Stanford University, 2000.

55. Séméria, L. - De Micheli, G., "SpC: Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioural Synthesis from C", Computer System Laboratory, Stanford University, 1998.

56. Séméria, L. - Sato, K. - De Micheli, G., "Resolution of Dynamic Memory Allocation and Pointers for the Behavioural Synthesis from C", Proceedings of the Design Automation and Test in Europe Conference, 2000, pp 312-319.

57. Verkest, D. - Da Silva, J. L. - Ykman, C. - Croes, K. - Miranda, M. - Wuytack, S. - De Jong, G. - Catthoor, F. - De Man, H., "Matisse: A system-on-chip design methodology emphasizing dynamic memory management", Proceedings of the IEEE Computer Society Workshop on VLSI System Level Design, 1998.

58. "Matisse", http://www.imec.be/matisse/.

59. Kucukcakar, K. - Chen, C. -T. - Gong, J. - Philipsen, W. - Tkacik, T. E., "Matisse: An Architectural Design Tool for Commodity ICs", IEEE Design and Test of Computers, Apr-Jun 1998, pp 22-33.

60. Wuytack, S. - Catthoor, F. V. M. - De Man, H. J., "Transforming Set Data Types to Power Optimal Data Structures", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 6, Jun 1996, pp 619-629.

61. Da Silva, J. L. - Ykman-Couvreur, C. - Miranda, M. - Croes, K. - Wuytack, S. - De Jong, G. - Catthoor, F. - Verkest, D. - Six, P. - De Man, H., "Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer", Proceedings of the 35th Design Automation Conference, June 1998, pp 76-81.

62. Wuytack, S. - Da Silva, J. L. - Catthoor, F. - De Jong, G. - Ykman, C., "Memory Management for Embedded Network Applications", IEEE Transactions on Computer-Aided Design, Vol. 18, No. 5, May 1999, pp 533-544.

63. Ellervee, P. - Miranda, M. - Catthoor, F. - Hemani, A., "System-Level Data Format Exploration for Dynamically Allocated Data Structures", Proceedings of the 37th Design Automation Conference, 2000, Session 32, pp 556-559.

64. Lin, B. - De Jong, G. - Verdonck, C. - Wuytack, S. - Catthoor, F., "Background Memory Management for Dynamic Data Structure Intensive Processing Systems", IMEC, Kapeldreef 75, B-3001 Leuven, Belgium.

65. Ykman-Couvreur, C. - Verkest, D. - Svantesson, B. - Hemani, A. - Wolf, F., "Stepwise Exploration and System Synthesis from SDL of an Operation and Maintenance Component in ATM switches", International Symposium on System Synthesis, Nov 1999, pp 85-91.

66. "MOODS Internals version 1.0", LME Design Automation, July 2001.

67. Baker, K. R., "Final Report: Application Specific Synthesis Enforcing Testability (ASSET)", University of Southampton, October 1995.

68. Brown, A. D. - Williams, A. C., "The MOODS Behavioural Synthesis System", Proceedings of the 3rd International Forum on Design Languages (FDL), Sept 2000, pp 17-21.

69. "Design Compiler User Guide", Version 2000.05, May 2000, Synopsys.

70. "Synergy VHDL Synthesizer and Optimizer Tutorial", Cadence Design Systems, Version 2.2, June 1995.

71. "LeonardoSpectrum User's Guide", Exemplar Logic, Inc. Version 1999.1, 1999.

72. "Quick Start Guide for Xilinx Alliance Series 1.5", Xilinx, Version 1.5, 1998.

73. Rutenbar, R. A. (chair), "Panel: (When) Will FPGAs Kill ASICs?", Proceedings of the 38th Design Automation Conference, 2001, Session 21, pp 321-322.

74. Williams, A. C. - Brown, A. D. - Baidas, Z. A., "Optimisation in behavioural synthesis using hierarchical expansion: Module ripping", IEE Proceedings on Computers and Digital Techniques, Vol. 148, No. 1, Jan 2001, pp 31-43.

75. Williams, A. C. - Brown, A. D. - Baidas, Z. A., "Hierarchical Module Expansion in a VHDL Behavioural Synthesis System", FDL'98, Sep 1998.

76. Ly, T. - Knapp, D. - Miller, R. - MacMillen, D., "Scheduling using Behavioral Templates", Proceedings of the 32nd ACM/IEEE Design Automation Conference, Session 7, 1995.

77. Aho, A. V. - Sethi, R. - Ullman, J. D., "Compilers - Principles, Techniques and Tools", Addison-Wesley Publishing Company, 1986, ISBN 0-201-10194-7.

78. Baker, K. R., "Writing Behavioural VHDL for MOODS Synthesis - User Manual for MOODS v1.xx", University of Southampton, July 1993.

79. Baker, K. R., "The MOODS Synthesis System - User Manual for MOODS v2.xx", University of Southampton, July 1993.

80. Ramachandran, L. - Vahid, F. - Narayan, S. - Gajski, D. D., "Semantics and Synthesis of Signals in Behavioral VHDL", Proceedings EuroDAC '92, 1992, pp. 616-621.

81. Williams, A. C. - Brown, A. D. - Zwolinski, M., "Simultaneous optimisation of dynamic power, area and delay in behavioural synthesis", IEE Proceedings on Computers and Digital Techniques, Vol. 147, No. 6, Nov 2000, pp 383-390.

82. Williams, A. C. - Brown, A. D. - Zwolinski, M., "A VHDL Behavioural Synthesis System Featuring Simultaneous Optimisation of Dynamic Power, Area and Delay", Proceedings of the 3rd International Forum on Design Languages (FDL), Sept 2000, pp 23-30.

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

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

85. Kirkpatrick, S., "Optimization by Simulated Annealing: Quantitive Studies", Journal of Statistical Physics, Vol. 34, Nos. 5/6, 1984, pp. 975-986.

86. Metropolis, N. - Rosenbluth, A. - Teller, A. - Teller, E., "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics, Vol. 21, 1087, 1953.

87. Baker, K. R. - Brown, A. D., "A Goal Directed Heuristic Optimisation Algorithm for the MOODS Synthesis System", University of Southampton, 1995.

88. 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, Jan 1997, pp 47-57.

89. Williams, A. C. - Brown, A. D. - Zwolinski, M., "In-line test of synthesised systems exploiting latency analysis", IEE Proceedings on Computers and Digital Techniques, Vol. 147, No. 1, Jan 2000, pp 33-41.

90. Kumar, V. - Grama, A. - Gupta, A. - Karypis, G., "Introduction to Parallel Computing. Design and Analysis of Algorithms", Benjamin/Cummings Publishing Company, 1994, ISBN: 0-8053-3170-0.

91. "Draft IEEE Standard VHDL Reference Manual, IEEE Std 1076a-2000".

92. 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, Jan 1997.

93. Nijhar, T. P. K. - Brown, A. D., "HDL-Specific Source Level Behavioural Optimisation", IEE Proceedings on Computers and Digital Techniques, Vol. 144, No. 2, Mar 1997, pp 138-144.

94. Nijhar, T. P. K., "Source Code Optimisation in a High Level Synthesis System", PhD Thesis, University of Southampton, Apr 1997.

95. "Fujitsu MB81C4256 DRAM data sheet", Fujitsu.

96. "IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)", IEEE Design Automation Standards Committee (1993b).

97. Chiou, D. - Jain, P - Rudolph, L - Devadas, S., "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches", Proceedings of the 37th Design Automation Conference, 2000, pp 416-419.

98. Kandemir, M. - Ramanujam, J. - Irwin, M. J. - Vijaykrishnan, N. - Kadayif, I. - Parikh, A., "Dynamic Management of Scratch-Pad Memory Space", Proceedings of the 38th Design Automation Conference, 2001, Session 42, pp 690-695.

99. Grunwald, D. - Zorn, B. - Henderson, R., "Improving the Cache Locality of Memory Allocation", Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, 1993, pp 177-186.

100. "Cypress CY7C185 SRAM data sheet", Cypress Semiconductor Corporation, Aug 1998.

101. Ramachandran, L. - Narayan, S. - Vahid, F. - Gajski, D. D., "Synthesis of Functions and Procedures in Behavioral VHDL", Proceedings EuroDAC '93, 1993, pp 560-565.

102. 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, Oct 1994, pp 399-406.

103. Milton, D. J. D. - Brown, A. D. - Williams, A. C., "Dynamic Memory Allocation in a VHDL Behavioural Synthesis System", Proceedings of the 3rd International Forum on Design Languages (FDL), Sept 2000, pp 45-51.

104. "The Programmable Logic Data Book", XILINX, San Jose, CA, PN 0010323, 1998.

105. Harrild, B., "High Level Behavioural Synthesis of Conway's "Life"", B.Eng. Project report, Department of Electronics and Computer Science, University of Southampton, 2001.

106. Baidas, Z. A., "High-level Floating-point Synthesis", PhD Thesis, University of Southampton, July 2000.

107. Baidas, Z. A. - Brown, A. D. - Williams, A. C., "A VHDL Behavioural Synthesis System with Floating Point Support", Proceedings of the 3rd International Forum on Design Languages (FDL), Sept 2000, pp 31-36.

108. Wakerly, J. F., "Digital Design Principles and Practices", Prentice Hall, 1990, ISBN: 0-13-212838-1.

109. Sacker, M. - Williams, A. C. - Brown, A. D., "Case Study: Comparing Behavioural with RTL Synthesis in the Development of a Programmable Digital Filter using VHDL", Proceedings of the 3rd International Forum on Design Languages (FDL), Sept 2000, pp 53-59.

110. Kreyszig, E., "Advanced Engineering Mathematics", 7th Ed., John Wiley & Sons, 1993, ISBN: 0-471-59989-1.

111. Kolson, D. J. - Nicolau, A. - Dutt, N., "Elimination of Redundant Memory Traffic in High-Level Synthesis", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 11, Nov 1996, pp 1354-1364.

112. Gupta, S. - Savoiu, N. - Kim, S. - Dutt, N. - Gupta, R. - Nicolau, A., "Speculation Techniques for High Level Synthesis of Control Intensive Designs", Proceedings of the 38th Design Automation Conference, 2001, Session 18, pp 269-272.

113. Grun, P. - Dutt, N. - Nicolau, A., "Memory aware compilation through accurate timing extraction", Proceedings of the 37th Design Automation Conference, 2000, Session 19, pp 316-321.

114. "VGA timing information", http://www.hut.fi/Misc/Electronics/Docs/pc/vga_timing.html.

115. Nicoud, J. -D., "Video RAMs: Structure and Applications", IEEE Micro, Feb 1998, pp 8-27.

116. Foley, J. D. - Van Dam, A. - Feiner, S. K. - Hughes, J. F., "Computer Graphics Principles and Practice", Addison Wesley, 1996, ISBN: 0-201-84840-6.

117. Bishop, L. - Eberly, D. - Whitted, T. - Finch, M. - Shantz, M., "Designing a PC Game Engine", IEEE Computer Graphics and Applications, Jan/Feb 1998, pp 46-53.

118. Cignoni, P. - Puppo, E. - Scopigno, R., "Representation and Visualization of Terrain Surfaces at Variable Resolution", Scientific Visualisation 98, World Scientific, pp 50-68.

119. "The TTL Data Book for design engineers, 2nd Ed.", Binary to BCD conversion, Texas Instruments, 1977.