APPLICATION OF OBJECT-ORIENTATION TO HDL-BASED DESIGNS

David Cabanis

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

Doctor of Philosophy

Bournemouth University

September

2000

DAVID CABANIS

APPLICATION OF OBJECT ORIENTATION TO HDL-BASED DESIGN

ABSTRACT

The increase in the scale of VLSI circuits over the last two decades has been of great importance to the development process. To cope with this ever-growing design complexity, new development techniques and methodologies have been researched and applied. The early 90's have witnessed the uptake of a new kind of design methodology based on Hardware Description Languages (HDL). This methodology has helped to master the possibilities inherent in our ability to manufacture ever-larger designs. However, while HDL based design methodology is sufficient to address today's standard ASIC sizes, it reaches its limits when considering tomorrow's design scales. Already, RISC processor chip descriptions can contain tens of thousands of HDL lines.

Object-Oriented design methodology has recently had a considerable impact in the software design community as it is tightly coupled with the handling of complex systems. Object-Orientation concentrates on data rather than functions since, throughout the design process, data are more stable than functions. Methodologies for both hardware and software have been introduced through the application of HDLs to hardware design. Common design constructs and principles that have proved successful in software language development should therefore be considered in order to assess their suitability for HDLs based designs.

A new methodology was created to emphasise on encapsulation, abstraction and classification of designs, using standard VHDL constructs. This achieves higher levels of modelling along with an improved reusability through design inheritance.

The development of extended semantics for integrating Object-Orientation in the VHDL language is described. Comparisons are made between the modelling abilities of the proposed extension and other competing proposals.

A UNIX based Object-Oriented to standard VHDL pre-processor is described along with translation techniques and their issues related to synthesis and simulation. This tool permitted validation of the new design methodology by application to existing design problems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This research was sponsored by the School of Design, Engineering and Computing at Bournemouth University and IBM Havant, UK.

The author would like to acknowledge the help and support given by his supervisor Doctor David Long and Nick Weavers.

The author would also like to acknowledge the support of the fellow researchers and members of staff of the School of Design, Engineering and Computing at Bournemouth University.

GLOSSARY

ASIC. Stands for Application Specific Integrated Circuit. This is usually used to refer to an integrated circuit (IC) that is designed to implement the functionality required for a popular product as opposed to standard ICs that implement a more diverse function that can be used in wide range of products. An ASIC solution reduces the number of ICs required for a product and so reduces the production cost. ASICs are one of the major growth areas in electronics.

Abstract Data Type. An abstract data type is a type, which defines a set of visibility rules and interfaces for accessing its internal properties. ADTs represent the back-bone of Object-Oriented programming.

Abstraction Level. Term used in the latest Hardware Description Languages to indicate the level of details used to describe a design. Commonly, three main levels are specified: Behavioural, Register Transfer, and Structural. The Behavioural level represents the most abstract form of description, mainly focusing on the behaviour of a design. The Register Transfer defines the implementation details of a design. The structural level is mainly used for assembling hierarchical blocks of a design.

Attribute. Also referred as feature, name given to the composing elements of an abstract data type such as access messages and internal instance variables.

BNF. Stands for Backus Naur Form. A context-free method for expressing the syntax and structure of a programming language. Used extensively in the IEEE VHDL language reference manual.

FPGA. Stands for Field Programmable Gate Array. Also referred as Complex Programmable Logic Devices (CPLD). This is the cost effective alternative to an ASIC for smaller production quantities. FPGAs benefits form a fast design cycle and on site re-programmability due to their static RAM based architecture. FPGA are generally more expensive than ASICs on a unit basis. This is caused by their complex coarse-grained architecture.

Genericity. Also referred as tailorability, term used to qualify the level of reusability of a design. A design is said to be generic if it provides the user with a number of modifiable parameter to allow for context sensitive adaptations.

HDL. Stands for Hardware Description Language. Depending on their complexity, HDLs will support different levels of abstraction. The latest HDLs languages are based on complex sequential programming languages and allow high abstraction levels systems description. HDLs are used both for simulation and implementation (synthesis) purpose.

Inheritance. Inheritance is an important feature of object-orientation. Inheritance provides a mechanism for design reuse by deriving functionality from an already existing one. Inheritance is one of the mechanisms supporting polymorphism.

Logic Synthesis. The logic synthesis operation consists in transforming register transfer language description in an optimised set of Boolean equations. The resulting equations are then mapped into ASIC library component primitives following some user-defined constraints (mainly related to execution speed and size of the design). High-level synthesis tools also include a primary operator optimisation stage.

Object-Orientation. The object-orientation is a collection of means for organising a description (e.g. specification, implementation). From this perspective, object-orientation may be regarded as a methodology. Two of the most important mechanisms introduced by object-orientation are inheritance and polymorphism. These two techniques when used in disciplined fashion will promote design maintainability and reusability.

Polymorphism. Object-oriented mechanism also known as late binding. Polymorphism characterises the dynamic *type* of an element. Polymorphism due to inheritance is often considered as an important contributor to object-orientation.

Register Transfer. The register transfer terminology is used to characterise a description's level of abstraction. This level is situated under the behavioural level (also known as executable specifications) and above the gate level (net-list format). The register transfer level description of a system specifies in detail the logic building blocks composing it and their connectivity.

Simulator. A simulator can evaluate a system consisting of components described at different levels of abstraction (e.g. a mixture of behavioural, register transfer language, and gate level). The evaluation of each of the system's node status is generally event based. The input stimuli required for the execution of the simulation are often referred as test bench or test harness.

VHDL. Stands for Very High Speed Integrated Circuit Hardware Description Language. This language was designed by the Institute of Electrical and Electronic Engineers (IEEE), primarily as a specification/simulation language. A subset of this language can be translated into a physical circuit layout by synthesis tools.

VLSI. Stands for Very Large Scale Integration. The technology that enables integrated circuits (ICs) containing millions of deep sub-micron transistors to be fabricated.

Chapter 1

OVERVIEW AND REQUIREMENTS SPECIFICATIONS

1.1 Introduction

This thesis describes the research work performed towards the application of the Object-Oriented paradigm to the Very High Speed Integrated Circuit Hardware Description Language VHDL (DASC 1993). This chapter summarises the contents of this thesis through taxonomy of the chapters along with a definition of the project background and objectives.

1.2 Rationale

The rapid advance in integrated circuit technology over the past two decades has driven the need for improved design entry tools. There are a number of advantages to be gained from the introduction of HDLs in ASIC design. Amongst them, is the ability to describe systems at different levels of abstraction: from a structural to behavioural description of the design. This allows faster and more reliable design capture, hence improving time to market. HDL based ASIC designs represents one of the main growth areas in the field of the digital electronics. However, although well suited for today's design sizes, Hardware Description Languages such as Verilog or VHDL already show some limits (Weiss 1994) for handling multi-designer large-sized designs. This is related to the bulk philosophy on which both languages have been built. This research investigates the limits found in the most popular HDLs for handling the concepts of design abstraction, reusability, extendibility and maintainability in an effort to develop a better suited language based on the Object-Oriented paradigm (Khoshafian 1989) (Booch 1991).

The notion for reusability and extendibility are primordial in the design of any application specific integrated circuit since most projects either represent a

cut-down or improved version of existing products. This is mainly motivated by the need to limit development costs for these projects.

As a result, it is crucial to specify and design electronic systems with reuse and maintainability issues in mind.

The two principal Hardware Description Languages (VHDL and Verilog) do not represent the ideal solution for the specification and building of large designs, as both languages are based on the widely used structured and obsolete programming methodology. Although structured programming represented a reliable methodology for designing small to medium sized systems it has long been acknowledged for its limits in the software world. A new hybrid language has therefore been created along with a pre-processor to address those limits. This language is enhanced with a layer of constructs favouring the use of object-oriented techniques for code structuring. Numerous extensions have been proposed by the design community to handle large descriptions (Vista) (Shumacher 1995)(Cabanis 1995)(Ashenden 1997)(Radetzki 1997).

An ideal hybrid language should be an orthogonal extension to an already existing and well-structured language, thus allowing optimal integration within the type system and philosophy of the original language. Furthermore, the new capabilities of the language should allow modern analysis and design methodologies to be used.

With procedural languages such as Verilog or VHDL, a multi-abstraction level paradigm is used to specify, simulate, design and implement a system. The Behavioural Level represents an abstraction of the functionality described during a functional decomposition. Consequently, at this stage of the design process, only functionality issues are addressed. The Register Transfer Level (RTL) also known as Data Flow Level, implements the physical aspects of the required functionality taking into account phenomena such as timing and

power consumption along with cost related issues such as the design size. The Gate Level will represent the result of an automated translation of an RTL description into a net list format.

The approach taken in most HDL based designs relies on a functional decomposition stage, followed by a behavioural specification for the functional blocks. The behavioural description is used with a test bench to ensure the consistency and correctness of the design throughout the construction of the system. The same test bench is then used with the RTL and gate level versions of the design as a means to cross-check the functionality during the different stages of the system's development.

For each level of abstraction, designers are required to build an interface in order to adapt the test bench to the abstraction of the information exchanged. The functional blocks interface types are usually different and not compatible from one level of abstraction to the next. The behavioural level will usually maintain user-defined types for interfacing. The register transfer level and gate level will be using more hardware-related types such as "bit" based or "standard logic" based types (either scalar or composite versions).

A problem with this approach is that the communication channels used to exchange information throughout functional blocks are often prone to changes during the life of a project. As a result, any minor changes in the system specification will have an important effect on the amount of work required to update the implementation. Furthermore, as can be seen from Figure 1-1. the structure of any functional block is very rigid and often cannot be reused for other applications without requiring significant reengineering effort in terms of design debugging.

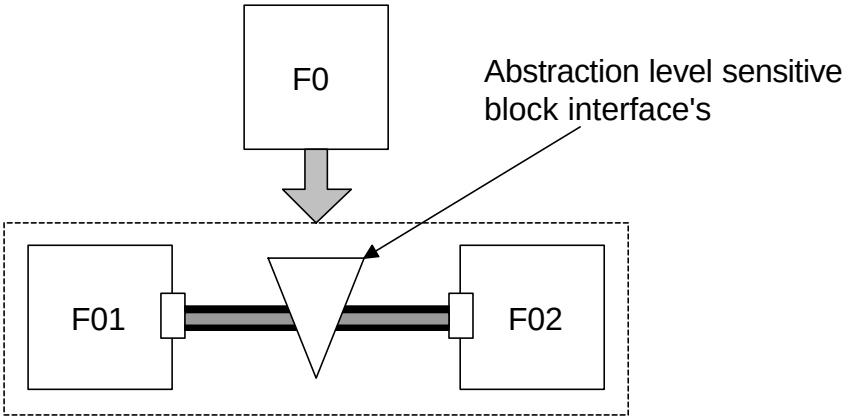


Figure 1-1. Rigid Functional Decomposition.

The need for an improved method for modelling is increasing as the size and complexity of new IC's designs steadily grows. On-going work on the subject is being carried out by an IEEE committee (Berger 1995) with the intention of developing a solution for the next generation of HDL. However, the best alternative is not clear, as there is no one solution to address all aspects of current design methodologies' limits. In terms of modelling, a number of issues that should be resolved can be listed. These include:

- Which design methodology ensures better reusability, maintainability and extendibility of hardware designs.

- To what extent a better design methodology could be implemented with already existing HDLs.

- What would be the requirements for the implementation of an improved Hardware Description Language.

- How to maintain a consistent link with already existing designs and CAD tools.

- To what extent reusability could be achieved with a new language when compared to existing HDLs.

- What should be the range of applications of any new language? E.g. specification, simulation, synthesis or formal verification.

Object-Orientation has been used in a number of concurrent processing applications since the encapsulated nature of an object provides a good mechanism for representing independent asynchronous processing units. Consequently, this infers that Object-Oriented modelling might be beneficial for the description of hardware systems.

Pragmatically, there are three main domains in hardware design where the use of Object-Orientation might prove beneficial. Primarily, enabling HDLs to support Object-Oriented constructs would narrow the gap between system analysis and system design. Both stages could be developed around an object-based methodology, hence, reducing the costly iterations that typically occur between the design stage and the analysis stage. This line of thought is related to the use of abstract interfaces for inter-component communication. For instance, as designs become more complex, it becomes almost mandatory to work at different levels of description (Behavioural, RTL, and Gates) for the different building blocks. As a result this task requires a significant effort in terms of project planning. Different parts of a system will be developed at a different pace using a variety of capture tools and means of verification. In the example of a common design, some parts might be bought from an external supplier (Intellectual Property) with fixed specifications, and interfaces. The level of details for those components will inevitably be low (RTL or Gate) since these models are designed for synthesis. Other hierarchical blocks of the designed system may require a “top-down” approach before getting to the implementation stage. With those blocks, implementation decisions may be delayed until a later stage. As a result, means of abstraction are needed for the overall design capture.

The second benefit of the use of object-orientation is related to the concept of inheritance. The inheritance represents a powerful means for categorising elements of systems and expanding their capabilities by derivation from already existing element attributes. Most modern ASIC designs are based on improved versions of existing designs, inheritance if used with hardware design would simplify the building of the new design and shorten time to market.

The third aspect is related to the modelling of large designs. Historically, software and hardware languages have been improved to further extend their abstraction capabilities. For instance, in the software world, structured languages such as C and Pascal that replaced assembly code have in turn, been replaced by Object-Oriented languages (namely C++ and other implementations of Object-Orientation). A similar analogy can be found in hardware design where gate level design and logic level description languages such as ABEL or PALASM have now been replaced by more abstract languages, namely, Verilog and VHDL.

Although a higher level of abstraction in a design commonly results in a less efficient implementation, the trade-off has proven worthwhile when the time to market, design maintainability and design extendibility represent important criteria.

To some extents, VHDL can be seen as an improved version of the Verilog language for its support for complex data structure, encapsulation and strong typing. If inheritance and polymorphism were to be added to the language to improve its abstraction capabilities, changes would have to be made to the VHDL encapsulation and type mechanisms to cope with the new designs inheritance and dynamic types.

This project focuses on improving the modelling capabilities of the VHDL language by increasing its abstraction level, thereby bringing hardware

modelling to higher grounds. As a result this should be better suited for the building and verification of large reusable designs. To achieve the announced objective, language extensions have been researched and added to the VHDL language with three essential criteria in mind: orthogonality of the new constructs with the existing ones, respect of the VHDL language philosophy (strong type system) and minimal changes to the language. Hardware models based on the new semantics and design methods have been developed, leading to the creation of a demonstration pre-processor tool to prove the validity of the research.

1.3 Aims and Objectives

To design a HDL for modelling large reusable hardware/software systems that is compatible with already used languages and proven design methodologies namely VHDL and the Object-Orientation.

To develop models in order to validate the proposed semantics and to illustrate the benefits of object-orientation applied to hardware description languages when dealing with behavioural and register transfer description level.

To create an experimental compiler/pre-processor to evaluate and validate the benefits of the proposed semantics over standard VHDL.

1.4 Taxonomy of Chapters

1.4.1 Chapter 1

This chapter introduces the research work carried out to achieve the project objectives and main issues involved. A brief description of the thesis layout is also given.

1.4.2 Chapter 2

An analytical study of the work carried out on design methodologies and proposed solutions to Object-Orientation applied to Hardware Description Languages is carried out in this chapter. This review is based on the material published in a range of software and hardware technical publications, books, journals and conference proceedings.

1.4.3 Chapter 3

The definition of the new semantics is described in this chapter along with the motivations for selecting the classification orientation versus other available solutions for achieving Object-Orientation.

1.4.4 Chapter 4

This chapter evaluates the validity and superiority of the newly proposed syntactical constructs by comparing our proposed extension to other existing proposals. The comparison uses a common set of benchmarks derived from the language design objectives set in chapter 2.

1.4.5 Chapter 5

This chapter demonstrates the proposed extension abilities for implementing design abstraction, reusability and maintainability. This is done by implementing a real-life application using our proposed extension. In order to contrast the proposed extension with standard VHDL code, the studied design will also be implemented using the VHDL RTL style. The comparison results will be used to draw conclusions on the benefits of our proposed extension over conventional VHDL code.

1.4.6 Chapter 6

This chapter presents the overall conclusion of this thesis. It also makes recommendations for further work and applications for an Object-Oriented Hardware Description Language.

1.4.7 Appendix A

This describes an experimental pre-processor that has been developed to demonstrate how a pre-processor handling inheritance, dynamic polymorphism and encapsulation could be developed to support the semantics developed in this thesis.

1.4.8 Appendix B

This appendix contains the code for the Object-Oriented description of the supporting classes for the design of the Edge Filter.

1.4.9 Appendix C

This holds both the Object-Oriented and RTL Code for the design of the Edge Filter.

Chapter 2.

REVIEW OF VLSI DESIGN METHODS

2.1 Introduction

VLSI design and software engineering are two key areas for which a rise in productivity is needed in the near future. There has been an interesting awareness of similarities in the two fields (Kumar 1993). The willingness to transfer technologies that have proved to be successful from one area to the other became stronger because of the limits of the schematic capture based methodology (Smith 1986), (Wirth 1998), (Shelor 1994). Designers who have experience in both areas have always spontaneously done technology transfer between the two disciplines. Some of the fields where technology transfer has worked successfully are in silicon compilation, in which part, or all of the process of translating a VLSI design concept into mask level layout instruction, has been automated. Like software language translators, which range from assembler to fourth generation languages systems, silicon compilers automate translations. Similarly, the assembling of predefined standard cells on a chip using traditional programming concepts to create a procedural VLSI design methodology has proved to be a success, applying to a VLSI design program, transformation approaches originally developed for software. These conclusive experiments led the designers to assume that a software like capture based methodology could be beneficial to the VLSI design process and to conceive Hardware Description Languages.

The next step hardware designers took in favour of technology transfer was the adoption of software analysis and design methodologies. Although hardware designers valued ASIC design, functional decomposition and structured analysis/ structured design methodologies, the concept of Object-Orientation, originally a software concept, was gradually gaining support within their community. The notion of objects was first introduced in the

language Simula (Dahl 1966), designed in the late 1960's. However, Object-Oriented programming did not emerge as a new analysis and programming paradigm until Smalltalk (Goldberg 1989) came along by the end of the 70's. Many Object-Oriented programming languages are widely used today. Although some of them have been designed from scratch, most of them are hybrid languages, that is to say conventional languages with added Object-Oriented concepts. This chapter will examine the limits of the current methods and identify requirements for an improved Hardware Description Language.

2.2 Schematic Capture Based Methodology

Critical time to market constraints and hardware costs have led ASIC development to rely on efficient design methodologies for decades [Figure 2.1]. Mask design was the first methodology used for the production of LSI designs; it was later replaced by logic level schematic capture. Schematic capture has been the main design entry technique used until the mid-eighties when hardware description languages appeared (Barbacci 1981).

Although the HDL based technique for developing hardware is similar to high level language to develop software, a number of issues have motivated the move from a visual graphical based methodology to a language based one.

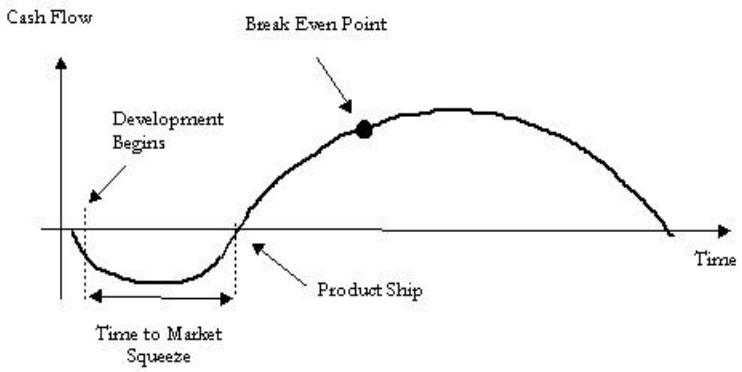


Figure 2-1. Time Related Cash Flow.

In a graphical based methodology, some information is provided to infer and clarify the functionality described in the schematic. This information will relate to specific data flow in the circuit, feed back data path, data dependencies or sequential and parallel operations. This means of representing a design is very convenient, as it does not just show how electric components connect but also how they operate. However, as component complexity increases, representing hierarchy via graphical means becomes limited. An example of this limitation can be found with the standard ANSI/IEEE standard symbols such representations are effective for small and limited functionality components but fail to address LSI or VLSI types of components, which are too complex to be represented in an efficient way.

Commonly, hardware designers add a number of indications on a design, such as specific vertical/horizontal alignments (for parallel or sequential executions), a data flow operating from left to right and others [Figure 2.2]. In most cases, a schematic should provide information that can be understood by a person reading it. The interpretation of this information has not been clearly defined and has been adopted by tradition. All these implicit means of giving information do not have their counterpart in language based designs; this justifies the popularity of graphical entry tools as "front ends" to HDL designs. Design tools such as state diagrams, flow diagrams, logic tables and others, support different design entry formalisms.

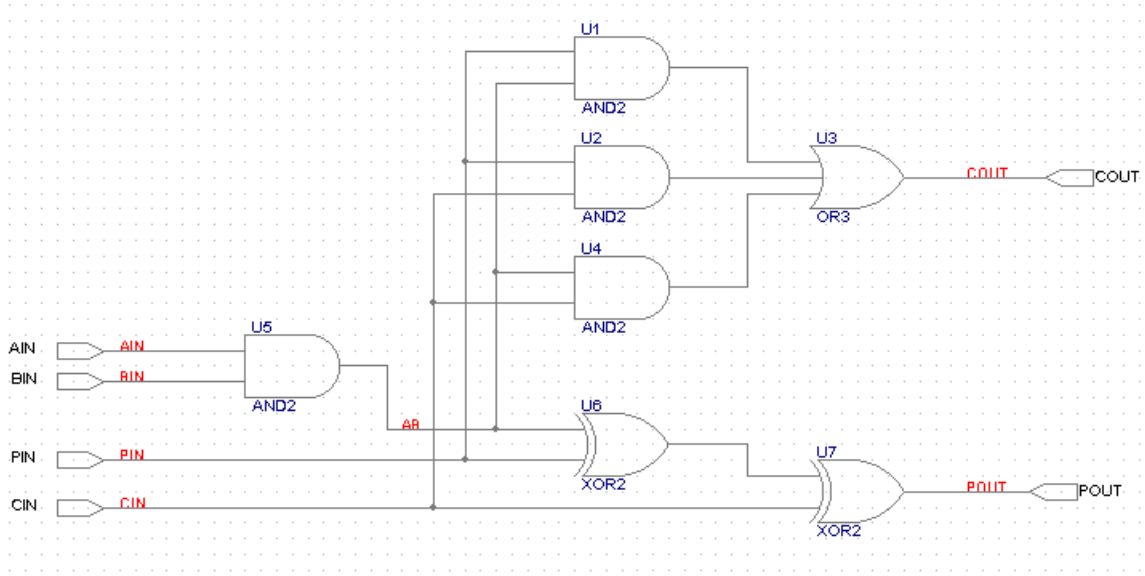


Figure 2-2. Schematic Based Design Capture.

To overcome this lack of information in HDL designs, run time assertions and inline comments can be used to clarify the code functionality. For example, information regarding the use of a specific architecture or sequential statement such as 'if' or 'case' infers the use of either multiplexor or decoder

based structures. Well-commented code can often overcome some schematic shortcomings.

Furthermore, HDL based descriptions free the designer from unnecessary details by abstracting ASIC library components. FPGA designs represent an exception to the rule. Due to their coarse grain architecture: internal features (IO blocks, internal clock buffers, etc.) need to be explicitly inferred to achieve satisfying synthesis results (Xilinx 1997). Ideally these problems should be solved as synthesis tools improve in the same way as for high level language software compilers.

In any kind of engineering construction, the most important factor for productivity is the reuse of prior effort (Keating 1998). This has greater significance in ASIC design, where getting the design right first time is a primary goal and design mistakes can jeopardise a project or increase the cost and shipment time drastically. Consequently, every reuse of well-tested and validated components increases the overall design reliability and reduces the cost and effort required. These factors apply equally to schematic or HDL based design methodology but, the use of the latter maximises its effect. When designers migrate from a schematic environment to a language environment, their motivations are often tied to the needs of creating larger designs within smaller time scales. Therefore the opportunity for reuse of larger designs increases with larger ASICs. To be able to apply design reuse, various elements need to be taken into account: the existence of functionality; what it provides; what it needs; its limitations and whether the level of testing has been thorough enough. Schematic capture does not guarantee the development of reusable parts in terms of genericity. Even though provisions for reuse can be built in, changes in ASIC technology cannot ensure direct mapping across ASIC libraries. Unlike schematic capture, with a HDL based design, a switch from one technology to another is a straightforward

operation, due to the technology independent nature of HDLs, thus preventing design obsolescence.

Hardware Description Languages can be classified according to their levels of abstraction (see Figure 2.3). Most of the early HDLs such as CDL (Chu 1965), DDL (Duley 1968), were designed specifically for description at the Register Transfer Level. Today, these languages represent the lowest entry level before the structural level. With these languages, the system had to be described in depth, using rigid expressions to assist the synthesis tools.

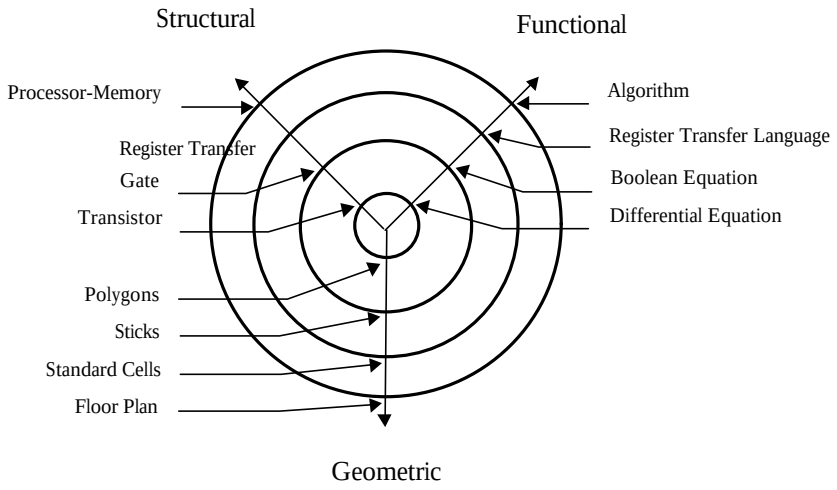


Figure 2-3 Applications of Abstraction Levels.

Later HDLs such as ARCHI (Nixon 1986), ADL (Leung 1979), S*A (Dargupta 1981), ZEUS (Lieber herr 1981) were aimed at micro-architecture level designs. With these HDLs, the description was based around a set of predefined architectures to make the design process easier, but design possibilities were restricted by the rigidity of the predefined architectures. Besides, these languages are not adequate for describing the digital system

design process, as they are often restricted in their ability to support different levels of abstractions.

In recent years, much research has been carried out on Multi-level HDLs. One of the solutions for this problem was to develop a single language that could support a large class of architectures. Some languages which fall into this category are ELLA (Morison 1985), Modlan (Pawlak 1981), UDL/I (Kartsu 1991), Trio (Coen-Porisini 1991), VHDL (DASC 1993) and Verilog (Verilog std 1364-1995). Three levels of description are currently supported by these languages: Structural, Data Flow (or Register Transfer Language) and Behavioural. Although these languages offer an efficient way of modelling and designing digital systems, few of them offer links from system specifications to Register Transfer Level, which represents the closest level to layout. Consequently, the resulting code for a complex ASIC (Application Specific Integrated Circuit) will become large. Apart from these languages, some research has been carried out in the field of specification languages. They are either a formal semantic for specifications such as ESP (Chu 1993), or a syntactic scanning, semantic analysis, interpretation generation and model integration of the English language (Cyre 1989).

Nevertheless, no specification language can be considered complete if it does not support any development techniques for analysis and reasoning. Typically, the phases of a development process include various stages: requirement analysis, design, implementation, test and integration stages. Hardware Description Languages such as VHDL were primarily designed for addressing those aspects; although nowadays VHDL cannot be used for all stages of a design, its use covers most of them. Within one development environment, engineers are able to specify, design and implement a circuit validating each level of the hierarchy via high level VHDL test benches. In comparison, schematic based designs do not allow such abstraction; the highest level often considered is the top-level block diagram. Furthermore, since a VHDL design

methodology is language based, this indirectly provides a convenient means of documentation.

VHDL, like most of the modern hardware description languages, is based on a structured programming technique. Structured programming was designed to organise complex programs more efficiently by linking the processing functions to the data structures. The method of functional decomposition characterises the steps that must be taken to reach a particular goal. These steps may be presented by functions that may take arguments in order to deal with data that is shared between the successive steps of the computation. In general, this method is not very good for data hiding. Another problem is that new designers may not be familiar with viewing their problems in terms of computation steps. Also, the method does not result in descriptions that are easy to alter. This leads us to further examine the limitations of current hardware system design methods in order to offer alternatives.

2.3 Current Hardware System Design Methods and their Limits

The choice of analysis and design methods is often motivated by the designers' need to manage design complexity and to keep development and maintenance costs down. Numerous analysis and design methodologies have been used since the 70s, contributing to a better definition of design processes. Besides, there has been an attempt at standardising the system representations used by designers. Electronic engineers successfully use two main methods: Functional Decomposition and System Based Analysis.

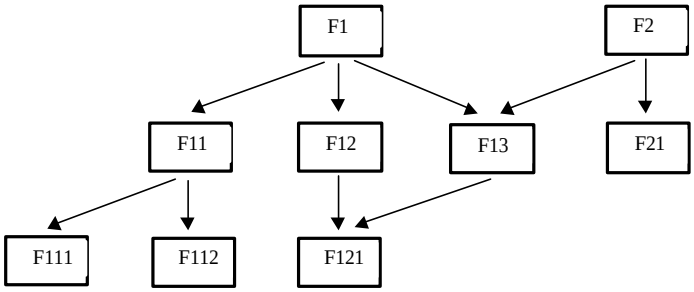


Figure 2-4. Functional Decomposition.

Functional decomposition treats one function at a time, and hierarchically decomposes it into a set of sub-functions. This process continues until the decomposition level is low enough to be implemented by a set of simple functions as shown in figure 2-4. This method is based on the principle that a problem can be divided into smaller manageable problems to reduce the complexity of the design. Consequently, the functional decomposition is seen as a simple approach towards achieving a sensible solution. Nevertheless, with this approach, the analysis effort is mainly based on creating design functions at the expense of data consistency. Furthermore, function decomposition rules are not clearly stated and therefore the decomposition hierarchy of a system will vary from one designer to another. Finally, the non-hierarchical interconnections encountered in the design of complex systems (between different taxonomies or within the different levels of the same tree) are difficult to represent with the functional decomposition paradigm.

System based analysis is inspired from the system theory by (Lanffy 1968); the system is seen as a complex and active object. The structural and functional aims of such an object are described at the System Analysis stage. Both Data models and Process models therefore represent the modelling of a system. Design rules are also provided for a better data consistency and process description. This approach appears to be an improvement over the functional

decomposition method. However, in this paradigm, the data and processes description techniques are not closely related.

An alternative approach to the system level description is based on graphical methods. Like system based analysis, graphical tools rely on an algorithmic description of the system. One of the most efficient but cumbersome methods is the Petri net description (Gourgand 1993). By using this paradigm, the concurrency within a design, as well as the data exchange protocol, can be expressed in a precise way [Figure 2-5]. As the description gets larger, the Petri net is more likely to make errors when events drive the system to a dead end or an infinite loop. Techniques based on matrix resolutions have been developed to overcome these problems but the description still rapidly becomes unmanageable.

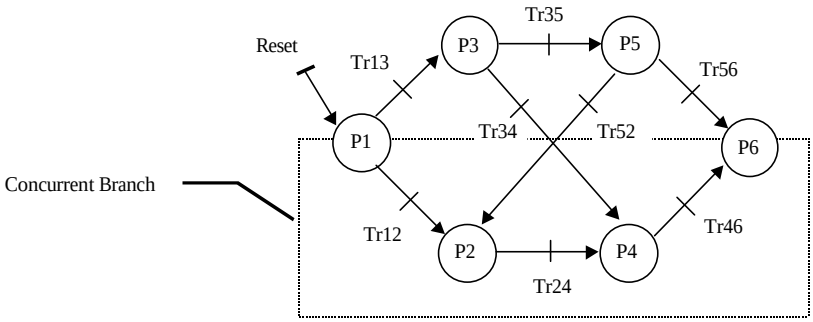


Figure 2-5. Petri Net Description.

Mixed language and graphical tools based on state diagrams are derived from the Petri net theory. Some languages that fall into this category are SpecCharts (Vahid 1991) and StateChar (Harel 1987). These languages represent an extended way of describing a system's behaviour at the algorithmic level. Nevertheless, the progress in system level description tools research (Kurup 1998) has brought new problems to light. For instance, as the level of abstraction increases, lower level models can no longer be expressed.

Conversely, it may not be possible to directly implement expressions in higher abstraction levels onto lower level models.

As a result, engineers commonly use finite [Figure 2-6] or algorithmic state machine descriptions, allowing low level modelling at the expense of abstraction level.

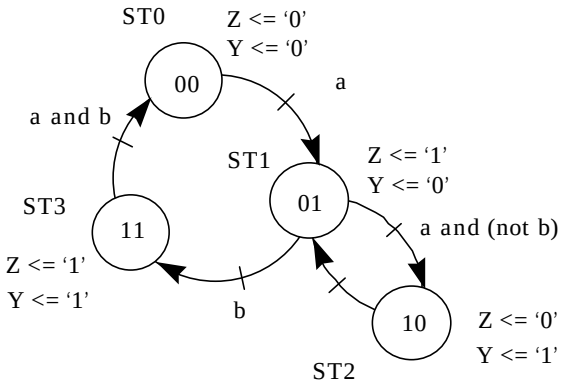


Figure 2-6. State Machine Graphical Description.

In addition to Petri net theory derived descriptions, there has been research reported on system level description using the well supported structured and functional analysis techniques (Lahitis 1991), (Bakowski 1992). These consist of hierarchical data-flow diagrams, which describe the flow of data and control signals inside the system, in response to external stimuli. It provides a system level description that is suitable for synthesis, as the method allows high level representation as well as low level modelling. Furthermore, such methods are based on reasoning techniques for which the designer follows clearly defined steps.

Although the latter techniques seem to be the way ahead for system level synthesis, they soon become inefficient when designs grow in complexity.

With large designs, work is usually allocated among different people. Ideally, designers should not have to know complex details concerning units built or modified by other people, only how the units they are responsible for interact with the rest of the system. However, with methodologies that do not emphasise encapsulation and information hiding, multi-person development team issues are more inclined to occur.

In addition to the design consistency issues, the development cost of large designs is usually very high. The cost of developing could be reduced, if some of the units the system is built from could be taken from already existing systems. Similarly, it would be beneficial if some parts could be reused in future projects. Nevertheless, neither functional decomposition nor System Analysis/ System Design (SA/SD) methods clearly emphasise the reuse and extendibility aspects of a design.

In addition to the discussed design method limits, the building of large systems using the functional decomposition or SA/SD methods do not facilitate the creation of components that closely relate to the application domain. The reason is that design based on such methods identifies two kinds of entities: the data, which are passive and represent the system, and the functions, which manipulate the data.

As a result, the designer of a system written in a procedural language such as Verilog or VHDL will face two alternatives. Either map the problem into a set of functions and define the data structure needed by the functions or map the problem to a set of data and write the functions that transform the input data into output data.

These observations lead us to look for an improved analysis and design technique providing an easier development of reusable units (genericity, tailorability), and a higher level of modelling, together with a better object abstraction.

2.4 Object-Oriented Methodology for HDL's Based Designs

The history of programming languages may be regarded as a progression from low level constructs towards high level of abstractions, that enable the programmer to specify programs in a more abstract manner thus allowing problem related abstractions to be captured more directly in a program. This development towards high level languages was partly motivated by the need to verify that a program adequately implemented a specification.

Object orientation is one of the most commonly used terms in recent design methodology projects, as it is closely associated with the handling of complex systems. Object-oriented methods address aspects such as encapsulation, reuse and derivation techniques to name but a few. Encapsulation is a technique used to hide the implementation of a design. The designer uses high level constructs to describe a system's behaviour without concerns about the final implementation of the construct. Reusability is achieved through message passing techniques to get information or to modify the values of an object. Consequently, the design of the function remains safe. Finally, derivation or inheritance is one of the most promising features of Object-orientation. This concept is based on the ability to design new functions from existing ones. To achieve this, a new design will inherit all the methods of its ancestors. All these attributes are highly beneficial when applied to hardware description languages (Takeuchi 1981).

Object-orientation has proven useful for numerous purposes within HDL. It can be useful for configuration management and macro modelling of analogue devices (Mammen 1994). It also provides a means of higher level description. Some of the hardware description languages that apply Object Oriented technology are Odice (Muller 1990), Loglan (Pawlak 1987), OODE (Takeuchi 1981) and SDL (Glunz 1998). These languages adopt different approaches for the semantic and object-orientation philosophies. Pascal,

Algol and LISP are used as a basis for the semantics, Simula67, Modula2 and Smalltalk for the Object Oriented Implementation.

The notion of objects, originally introduced in Simula, has significantly influenced the design of many subsequent languages (e.g. Modula and Ada). The first well-known object oriented language was Smalltalk. In Smalltalk, the data hiding aspect of objects has been combined with the mechanism of inheritance, allowing the reuse of code defining the behaviour of objects.

Both information hiding and data abstraction relieve the programmer's task when using existing code; they are no longer distracted by irrelevant implementation details. On the other hand, the code developer may benefit from information hiding as well, since they will be able to optimise the implementation without interfering with the client code.

The advantages of Object-Orientation are clear when considering the gap between system design and implementation. In a number of instances, Object-Oriented analysis has been regarded as a solution to this problem of communication. According to Coad and Yourdon (Coad 1991), Object-Oriented techniques allow us to capture the system requirements in a model that directly corresponds to a conceptual model of the problem domain. Furthermore, proponents of Object-Oriented Programming (OOP) claimed (Cox 1990) that an Object-Oriented approach enables a more seamless transition between the respective phases of the software life cycle. If this turns out to be true, changing user requirements could be more easily discussed as these changes would be less disruptive for the system and they could in principle be more easily applied to the successive phases of the development. One of the basic ideas underlying Object-Oriented analysis is that the abstractions achieved in developing a conceptual model of the problem domain will remain stable over time. Consequently, rather than focusing on specific functional requirements, attention should be given to modelling the problem domain by means of high level abstractions. Due to

the stability of those abstractions, the results of analysis are likely elements for reuse.

To achieve reusability, extendibility and reliability, the principles of Object-Oriented design provides the best known technical answer. Object-Orientation is a way of organising designs by focusing on a fundamental construct: the Object (Booch 1991). A design therefore represents a collection of discrete objects that incorporate both data structure and behaviours. This is often referred to as “tight binding”, as opposed to the “loose binding” found in common design methodologies. The main characteristics used to support this paradigm are classification, inheritance and polymorphism. The classification enables the designer to identify groups of objects with commonalties and combine them into distinct categories. Objects are therefore seen as instances of a given class, while a class can gather an infinite number of objects exhibiting identical attributes. Figure 2-7 illustrates the Object-Oriented view of a design containing a number of FIFO (First In First Out) and LIFO (Last In First Out) objects gathered in two distinct groups. The action of grouping similar objects into classes enables further higher abstraction in the design process as illustrated by the queues class.

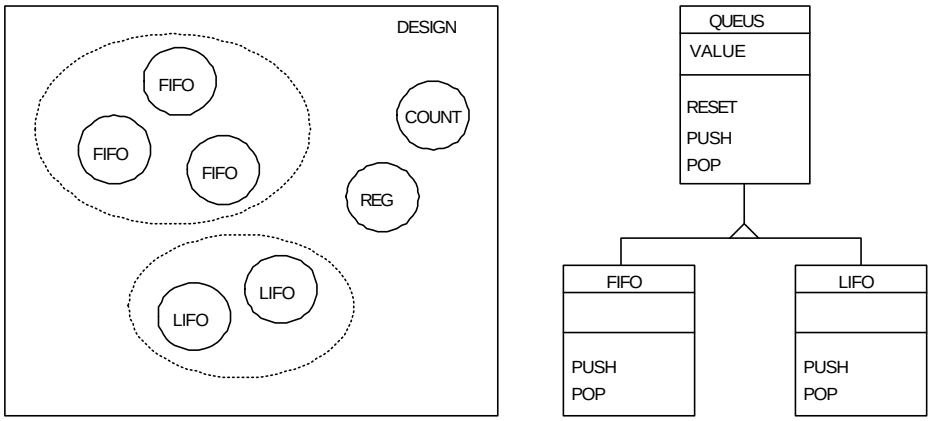


Figure 2-7. Object Based Design Representation.

The inheritance would also bring an extra level of organisation flexibility to hardware designs, as it refers to the action of creating a hierarchy of classes sharing common data structure and behaviours. A child class will be a refinement of a parent class, which transfers its unique properties to the inherited class. Inheritance is used to implement code sharing amongst several similar classes, therefore reducing code redundancy. In Figure 2-8 an example of inheritance is given with the queues class.

One can imagine that the two child classes FIFO and LIFO are using the same code for implementing the reset functionality. As a result, this operation would be defined at the parent class level thereby promoting code sharing. Inheritance is not just a module combination and enrichment mechanism, it also enables the definition of flexible entries that may become attached to objects of various forms whilst an HDL simulation is executing (referred as: run time), a property known as polymorphism. A polymorphic operation is characterised by its ability to adopt different behaviours depending on the object addressed.

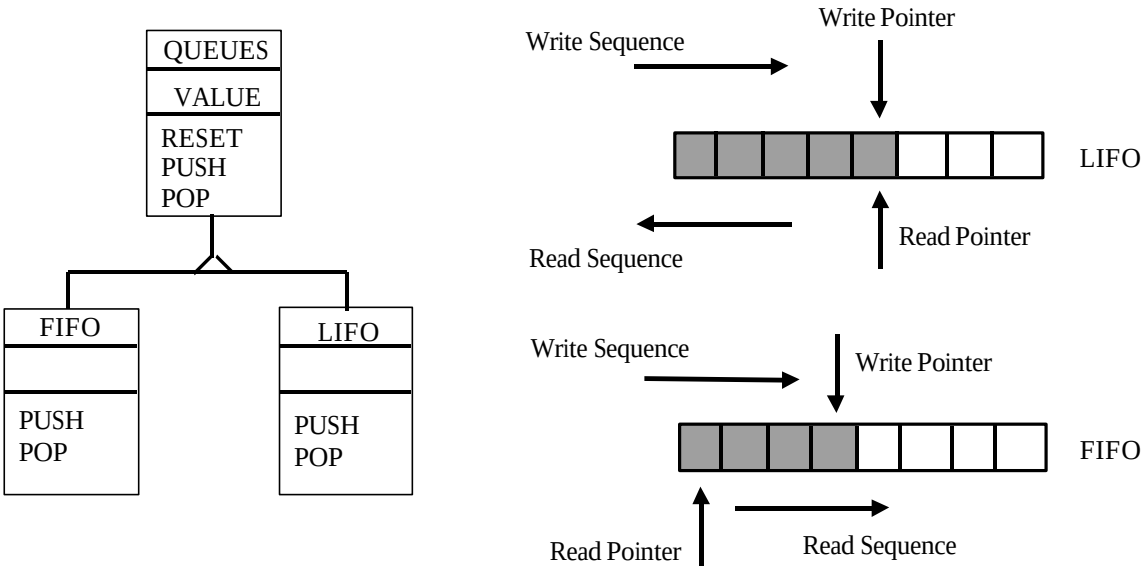


Figure 2-8. Queues Implementation.

Considering the example given in Figure 2-8, the operation PUSH could be identical for the two child classes FIFO and LIFO and could consequently be defined at the parent class level. However, the POP operation, although having the same identifier, must behave differently. To do so, the user will "overload" the two different behaviours for the two different classes and rely on the execution program (HDL simulator) to select the appropriate one depending on the class of the selected object. As can be seen, polymorphism together with encapsulation allows a higher level of system design abstraction by postponing implementation decisions until later in the development process. Although it is difficult to find equivalents to some of these new principles in hardware, it is still sensible to search for similarities between software design and hardware design methodologies. It is even more relevant since the introduction of HDLs for hardware design. Indeed, a structure of hardware entities exchanging signals closely resembles the object model. This latest advance in the hardware design process is comparable to the introduction of structured procedural languages in software. Such an increase

in the abstraction level was motivated by an increase in design complexity and design effort. Already, the hardware community faces identical problems when considering large HDL descriptions. Design organisation, maintainability and reusability are some of the main issues that have to be studied.

In an effort to provide an object-oriented version of the VHDL language, the IEEE OOVHDL Study Group (www.vhdl.org) committee has produced a requirement and design objectives document. This document identifies the needs and main requirements for the building of an object-oriented extension. The requirements are related to many aspects of the language and the design methodology. However, from this document, eight main points have been identified.

1) To target higher levels of modelling in order to ease the changes to the design specifications and make designs less technology dependent.

2) To simplify and speed-up the process of specifications. This should overcome the challenges faced during the specification of ever increasingly complex systems.

3) To ease the addition of new functionality thereby reducing the needs for restructuring large parts of a design.

4) To improve the verification process by bridging the gap between implementation and specifications.

5) To further the level of reusability; This in turn should yield better design productivity along with higher design quality.

6) To cater for better documentation capabilities, to help the designs of growing reusable components libraries.

7) To provide better consistency for the creation of Hardware/Software co-designs.

8) To make the extension easy to learn and apply; Hence allowing a better acceptance of the new semantics by the designers' community.

Along with the above guidelines, the IEEE Object-Oriented study group enumerates a collection of concepts that the object-oriented extension has to include. Most of these concepts merely characterise the main features of an object-oriented language. These are:

1) Add better abstract data typing to VHDL.

2) Add inheritance/multiple inheritance.

3) Support for method calls or message passing.

4) Add polymorphism.

5) Add Dynamic creation/removal of objects.

6) Add documentation mechanism.

Having defined the requirements and design objectives for the creation of an extension to VHDL, we will now examine to what extent VHDL supports those objectives. We will also identify which of the VHDL constructs and mechanisms could be used as a base for the building of the language extension.

2.5 Object-Orientation in Standard VHDL

2.5.1 VHDL Encapsulation and Design Abstraction

Commonly, in most HDL based designs, not all features of the system need to be accessible by an end user of each component. The designer of a

component may therefore want to keep some features private or available only to specific users (e.g. because they are for internal use only and subject to changes so letting all users access them directly could endanger further evolution of the component). This is especially true of features (sub-programs or attributes) that reflect alternative implementation details of a design rather than its primary functionality. By keeping such features private, the component designer protects users against the effect of changes in the final implementation. This policy is known as information hiding. When elaborating a design, one may achieve different levels of information hiding. For immediate functionality (those introduced in the design itself), one may specify any required encapsulation restrictions by using the public or private mechanisms as found in C++. Obviously, when considering component design, a public feature will only be accessible through the ports of the given component. Consequently, any feature not listed in the port list will be seen as private. This component approach is somewhat limited and the VHDL language offers other constructs to achieve an even smoother encapsulation.

VHDL is derived from the ADA language. As such, it supports abstract data types by means of a syntactic package construct (DASC 1993). To a larger extent, this is equivalent to the class abstract data type found in most Object-Oriented programming languages. However, the benefits achieved through abstraction can be obtained only when specific coding style is used. Such a coding style involves the use of VHDL objects such as block statements, package design units and configuration design units. In many VHDL designs, a system can be characterised as a combination of several architectures, each organised as a collection of processes and related signals. A number of packages are often used throughout the design to gather common declarations. Finally, a further level of flexibility is often added to the VHDL description through the use of configuration statements. From this observation, we can now examine issues regarding the VHDL code maintainability and reusability as it stands, and derive an Object-Oriented

implementation method to achieve better encapsulation and abstraction. We will use simple cases with clearly defined goals in order to discuss the various implementation possibilities.

The first case implements an abstract data type for a register. An inheritance mechanism is then used to create a counter abstract data type. The description of the two abstract data types uses a generic algorithmic language. Figure 2-9 shows the implementation of the base register abstract data type. An attribute *value* is defined to hold the internal value of any instance of the register "ADT". Methods are used to implement the *reset*, *read*, and *write* behaviours. This model does not consider any synthesis issues and thus focuses only on behavioural aspects. Furthermore, the type of assignment is left undefined. In Figure 2-10, a counter abstract data type is designed by inheriting the capabilities of the previously defined register ADT. This new abstract data type needs only to implement the count operation to describe a complete counter. In both examples, the messages will be considered to be directly accessible by client applications – i.e. they are public.

Although these two examples do not display the full range of the Object-Orientation paradigm capabilities, they enable us to understand two of the main features: encapsulation and inheritance. To compare the two examples to standard VHDL, a simple direct analogy can be made between the abstract data type concept and the entity-architecture pair. The entity can be seen as the 'shell' of the class and performs the encapsulation; the architecture part defines the implementation of the class. Similarly, VHDL signals can be used to communicate between entities as well as to implement data attributes. Figure 2-11 demonstrates how the register class can be coded in standard VHDL with a message oriented communication scheme.

```
Body of Register
(
      attribute value: bit_vector(0 to 7)
      method reset
      (
            value becomes “00000000”
      )

      method read (v: out bit_vector)
      (
            v becomes value
      )

      method write (v: in bit_vector)
      (
            value becomes v
      )
)
```

Figure 2-9. Algorithmic Description of the Register ADT.

```
Body of counter inherits register
(
      method count
      (
            value becomes value + “00000001”
      )
)
```

Figure 2-10. Algorithmic Description of the Counter ADT.

In this example, a user-defined type is created to list the different message that will be sent to the interface (entity). The message decoding is performed via the use of a “case” statement and the data attribute takes the form of the

The entity ports are defined for input values by the write method argument and output values by the read method argument. Using the same approach,

the counter is implemented with standard VHDL in an Object-Oriented fashion [Figure 2-12]. Here, new functionality is added to support the 'count' operation.

```
package reg_ADT_pkg is
      type   message_type is (reset, read, write);
end reg_ADT_pkg;

use work.reg_ADT_pkg.all;
entity reg is
      port (message: message_type;
            write_param_in: in bit_vector(0 to 7);
            read_param_out:  out bit_vector(0 to 7)
            );
end reg;

architecture OO of reg is
      signal value: bit_vector(0 to 7);
begin
      message_exec: process(message'transaction)
      begin
            case message is
                  when reset =>
                        value <= (others =>'0');
                  when read =>
                        read_param_out <= value;
                  when write =>
                        value <= write_param_in;
                  when others => null;
            end case;
      end process message_exec;
end OO;
```

Figure 2-11. Message Oriented Communication Scheme.

```
package counter_ADT_pkg is
      type   message_type is (reset, read, write, count);
end counter_ADT_pkg;

use work.counter_ADT_pkg.all;
entity counter is
      port (message: message_type;
            write_param_in: in bit_vector(0 to 7);
            read_param_out:  out bit_vector(0 to 7)
           );
end counter;

architecture OO of counter is
      signal value: bit_vector(0 to 7);
begin
      message_exec: process(message'transaction)
      begin
            case message is
                  when reset =>
                        value <= (others =>'0');
                  when read =>
                        read_param_out <= value;
                  when write =>
                        value <= write_param_in;
                  when count =>
                        value <= value + "00000001";
                  when others => null;
            end case;
      end process message_exec;
end OO;
```

Figure 2-12. Register Code Re-use.

Although very similar to the register in its structure, the counter [Figure 2-12] requires a number of modifications to implement the correct functionality. The message type had to be changed as well as part of the case statement. This simple example already highlights the weaknesses of VHDL when considering code reuse.

Having looked at the most basic way of implementing an abstract data type in standard VHDL (the component object), we will now consider other options

offered by the language in an effort to demonstrate their benefits over the component object method.

A good design technique recommends the use of local variables instead of signals whenever it is possible, thus enforcing the encapsulation aspect. However, large architectures (4000 to 5000-logic gates equivalent) will contain a significant number of signals, typically implementing control functions or data path functions. These signals should be rationalised in the same way as local variables: by logically grouping related signals within the architecture. Similarly, related processes should be gathered logically to simplify any future code reuse. The VHDL language provides the “block” construct to perform this encapsulation.

Although the block construct has not generated a significant interest within the design community, we believe that if adequately used, it should prove an efficient way of increasing designers’ productivity.

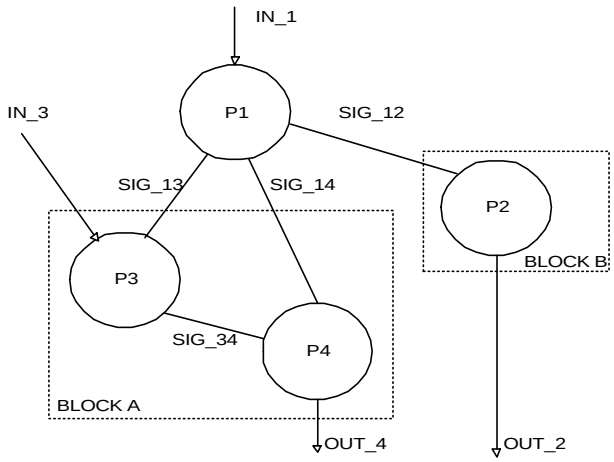


Figure 2-13. Block-Based Design Structure

In Figure 2-13, we show an architecture containing four processes that represents the design of a display driver. Process P1 is a master controller

taking synchronisation signals as inputs and generating appropriate sequence of control signals for the dependent processes. Process P2 is a lower level controller generating synchronisation signals for the external display and processes P3 and P4 are for error correction and “de-serialisation” respectively. The conventional way of designing this system consists of defining types for the signals, declaring control and data path signals within the architecture declarative part and finally defining processes P1-P4 (in any order). However, alterations are often required throughout the design life cycle to cope with specification changes or design updates. Consequently, conventional design methods are not adapted for design maintainability and design reuse. To illustrate this remark, consider a design change such that the code implementing the data path is replaced. With a conventional coding style, such operations will require the identification of declarations and processes to be removed or renamed in order to customise the architecture for the new application. This operation might represent a long and error prone process. However, by using the encapsulation capability of the block, the alteration impact is minimised and simplified. The block encapsulating P3 and P4 contains local declarations regarding the processes (signal SIG_34) and will simply need to be cut out of the description. Furthermore, the replacement block will be able to map its interface to the rest of the architecture blocks through the use of “port map” and “generic map” constructs. Figure 2-14 describes the structure of the suggested coding style. By gathering related processes into sets of blocks, the designer is able to localise the effect of potential alterations and thus build robust code. Furthermore, the encapsulation capability of the blocks gives new opportunities for design reuse. Consequently, the structuring of architectures around blocks has a significant benefit and should not be eluded. Unfortunately, the benefits of blocks are constrained to the architecture level of a VHDL description. However, packages can be employed as abstract data types to achieve encapsulation at the entity level.

```
architecture rtl of display is
   -- global declarations
   type t_state is (idle, rd_wr, int, load);
   signal state : t_state;
   signal sig_12, sig_13, sig_14 : bit;
begin
   -- process p1 description
block_a: block
   -- local declarations
   subtype data is bit_vecor(size-1 downto 0);
   signal sig_34 : data;
begin
   p3: process(in_3);
   begin
      -- process description
   end process p3;
   p4: process(sig_34)
   begin
      -- process description
   end process p4;
end block block_a;
block_b: block
   -- local declarations
   type t_state is (idle, write, refresh, sync);
   signal state t_state;
begin
   p2: process(clk, reset)
   begin
      -- process description
   end process p2;
   -- concurrent statements
end block block_b;
end display;
```

Figure 2-14. Use of the VHDL Block Statement

So far, packages in VHDL have been described as a convenient means for gathering shared declarations, using them as abstract data types can add a new dimension to the design process (Willis 1994). To implement an abstract data type, a package will have to contain the data structure and various associated behaviours of an object. The data structure can be implemented via signals or "shared variables" and the behaviours via subprograms (including "impure" functions). Figure 2-15 gives the VHDL code used to design the counter using a package as an abstract data type, while Figure 2-16 demonstrates its use.

```
package  counter is
 shared variable value : integer;
 procedure reset;
 procedure count_up(signal clk: in bit);
 procedure count_down(signal clk: in bit);
 procedure read(read_return: out integer);
end counter;

package  body counter is
procedure reset is
begin
   value := 0;
end reset;
procedure count_up(signal clk: in bit) is
begin
   if clk'event and clk = '1' then
      value := value + 1;
   end if;
end count_up;
procedure count_down(signal clk: in bit) is
  begin
   if clk'event and clk = '1' then
      value := value - 1;
   end if;
end count_down;
procedure read (read_return: out integer)is
begin
  read_return := value;
end read;
end counter;
```

Figure 2-15. Counter Described Using The 'Abstract Component' Style

Although a component instantiation would have achieved the same results in terms of functionality, the use of the package as an 'abstract component' achieves better results on numerous aspects.

As shown, the design of a counter using conventional coding styles requires the use of an entity, architecture and a process. It is possible that after compilation, this counter could be reused as a component in another design. Unfortunately, if the new design requires only a part of the functionality of the component, the redundant functionality will still be implemented. However, with the 'abstract component' approach, only those procedures used by the new design will be implemented: the synthesis tool will ignore the rest. Furthermore with conventional coding styles, if changes are required to a function of the instantiated component, the designer will either have to create a new version of the component (reducing design reliability) or create additional functions in the new design (introducing redundant logic). The use of subprogram overloading simplifies this operation with the 'abstract component' solution and does not add redundant logic.

```
process
   variable result : integer;
begin
   work.counter.reset;
   wait for 20 ns;
   for i in 0 to 10 loop
      wait on clk;
      work.counter.count_up(clk);
   end loop;
   work.counter.read(result);
   sig_result <= result;
   wait for 20 ns;
   wait;
end process;
```

Figure 2-16. Use of an 'Abstract Component'

The use of packages as abstract data types enables design maintainability through the use of feature calls as illustrated in Figure 2-16 Due to the syntactic dot notation used by most Object-Oriented programming languages, feature calls used as expressions benefit from an important property: the notation is the same for a call involving a function with no arguments as one

involving an attribute. So for instance, the expression *UART.data* is applicable whether the feature *data* is an instance variable or a method. This property of uniform access assists the smooth evolution of a design by protecting abstract data types from internal implementation changes in the objects they are referencing.

A number of styles for message passing between package based abstract data types can be considered:

A message for linear structures represents the addition or removal of an information.

- `Fifo.addFirst(data)` : adds the element data as the first element of the 'first-in first-out' linear data structure.

- `Fifo.removeLast` : remove and return the last element of the 'first-in first-out' linear data structure.

A message to associative data structures representing the addition or removal of information.

- `BitNumber.set(controlReg,8)` : associate the value 8 to the *controlReg* field in the *bitNumber* data dictionary.

- `Addresses.get( UART )` : look up the address associated with *UART* in the dictionary called addresses.

Messages to address counters representing position enquiries and calculations.

- `AddressCounter.base` : answers the value of the starting address for the counter named *addressCounter*.

- `AddressCounter.isContained (addressValue)`: answers true if the location named *addressValue* is contained within the counter address space and false otherwise.

- `AddressCounter.intersect(addressValue)`: computes the value that separates the location named *addressValue* and the current address value of the counter called *addressCounter*.

Clearly, the use of packages to build abstract components significantly outperforms the capabilities of the component instantiation when considering code reusability and maintainability aspects. Nevertheless, when used as such, the VHDL abstract data type reveals a lack of encapsulation capabilities over its Object-Oriented counterpart: the class.

The package construct enables access to data structures without resorting to subprograms. Consequently, the data-function binding is loose. To illustrate the deficiencies of the package in terms of data hiding, we will use new examples. In Figure 2-17, a signal is used as an attribute 'data'. A public procedure (visible outside the package) is declared and implements the behaviour of the abstract data type.

From this abstract data type, we will now consider the different means of performing operations and assessing their pertinence. Figures 2-18, 2-19 and 2-20 illustrate the different levels of data hiding that can be achieved when using the package as an abstract data type. In the first instance [Figure 2-18], the data hiding is minimal as the package is instantiated before the entity design unit. As a result, any attributes of the package can be accessed with no restrictions throughout the dependent architectures, with the exception of the *calculate* function.

```
package ADT_pkg is
        signal data : integer;
        procedure set_value(   signal clk : in bit;
                               signal value_in : in integer;
end ADT_pkg;

package body ADT_pkg is
        procedure set_value(   signal clk : in bit;
                               signal value_in : in integer;
                               signal value_out : out integer) is
        begin
                if clk'event and clk='1' then
                        value_out <= value_in;
                end if;
        end set_value;
        function calculate(value_in : in integer) return integer is
        begin
                return value_in*value_in;
        end calculate;
end ADT_pkg;
```

Figure 2-17. Use of a Package for Building Abstract Data Types

In the package body, the function ‘calculate’ is described but has not been declared as a part of the package header. Therefore, this simple mechanism achieves a private encapsulation. Thus, the function ‘calculate’ is not accessible outside the package.

Although this technique represents the most common way of using packages, it cannot be considered as a proper means of encapsulation. The realisation of abstract data types as modules with functions requires additional means to hide the representation of the *data* instance variable. By contrast, with an Object-Oriented approach, employing the encapsulation facilities of classes effects data hiding. To restrict the access to the package elements, the designer can alter the code, as shown in Figure 2-19.

Through the use of blocks for instance, the designer is able to restrict the visibility of a package attributes to a part of an architecture.

```
use work.ADT_pkg.all;
entity test is
        port(   input_value : in integer;
        clk : in bit);
end test;

architecture ex_1 of test is
        signal temp_value_1 : integer;
        signal temp_value_2 : integer;
begin
        set_value(clk, input_value, data);
        temp_value_1 <= calculate_value(data); -- invalid statement.
        temp_value_2 <= data+data;             -- direct access to data.
end ex_1;
```

Figure 2-18. Package, Visibility throughout the Entity / Architecture

```
entity test_2 is
        port(   input_value : in integer;
        clk : in bit);
end test_2;

architecture ex_1 of test_2 is
        signal temp_value_1 : integer;
        signal temp_value_2 : integer;
        signal temp_value_3 : integer;
begin
        encapsulation: block
                use work.ADT_pkg.all;
        begin
                set_value(clk, input_value, data);
                temp_value_1 <= calculate_value(data); -- invalid statement.
                temp_value_2 <= data+data;              -- direct access to data.
        end block encapsulation;
        temp_value_3 <= data+data;                    -- invalid statement,
end ex_1;                                             -- data not visible.
```

Figure 2-19. Narrowing the Accessibility of a Package

To conclude on the encapsulation capabilities of the package abstract data type, an even more rigorous way of performing data hiding is to consider packages as the implementations of single objects, i.e. packages represent only one instance. The use of this ADT is outlined in Figure 2.20.

```
entity test_3 is
        port(   input_value : in integer;
                clk : in bit);
end test_3;

architecture ex_1 of test_3 is
        signal temp_value_1 : integer;
        signal temp_value_2 : integer;
        signal temp_value_3 : integer;
begin
        encapsulation: block
        begin
                work.ADT_pkg.set_value(clk, input_value, work.ADT_pkg.data);
                temp_value_1 <= work.ADT_pkg.calculate_value
                                    (work.ADT_pkg.data); -- invalid statement.
                temp_value_2 <= data + data; --invalid statement ta not visible
        end block encapsulation;
        temp_value_3 <= data+data; -- invalid statement, data not visible
end ex_1;
```

Figure 2-20. Selective Access to Package Instances

In this example, specifying the full logical path to it accesses any attribute of the package. The ‘use’ statement then becomes unnecessary.

Such a representation is too cumbersome to be considered for real coding use and the loss of code robustness is compensated by the code legibility.

Packages seem to be the best contenders to implement abstract data type in VHDL. However, in order to decide whether package is powerful enough to support Object-Orientation, other issues still have to be considered.

Presently, the actual benefits of Object-Oriented programming are difficult to distinguish from existing VHDL language features. From a pragmatic point of view, Object-Oriented programming offers two main concepts used in program development: data hiding and inheritance. Encapsulation provides modularity and, as we have demonstrated in this section, can be achieved in standard VHDL to a certain extent. One of the limitations is that multiple instances of an ‘abstract component’ in a design will result in the creation of multiple occurrences of the required package. However, once a proper modularisation has been achieved, the designer of the abstract data type may

postpone any final decisions concerning the implementation at will. This feature allows quick programming.

Another advantage of the Object-Oriented approach, often considered as the main advantage, is the reuse of code. Inheritance is an invaluable mechanism in this respect, since the code that is reused offers all necessary elements. The inheritance mechanism enables the programmer to modify the behaviour of a class of objects without requiring access to the source code. The next section will focus on how to achieve inheritance in standard VHDL and assess current capabilities.

2.5.2 Limited Inheritance in Standard VHDL

The basic features of Object-Oriented programming consist of encapsulation and inheritance. In the previous section, we have shown how encapsulation is used to support the realisation of abstract data types in a VHDL based design flow. Inheritance would complement the encapsulation by providing a mechanism for sharing code. This ultimately provides a mechanism for defining polymorphic types. The VHDL language does not support inheritance although the combination of sub-typing and overloading can be regarded as a step towards it. As a result, this section will focus on identifying means of approaching inheritance using standard VHDL constructs and identify the limitations found with the existing semantics of the VHDL type system.

From a pragmatic point of view, derivation can be assimilated to the operation of “sub-typing” in procedural languages, the parent type being the base class and the sub-type being the child class equivalent in the Object-Oriented domain. From this observation, we will examine the capabilities of VHDL sub-types and types.

Three main cases can be studied from the definition of a parent type. The first case extends the type capabilities by defining a composite of the parent type as demonstrated in the following statement:

```
type array_int is array range (0 to 10) of integer;
signal sig_1 : integer;
signal sig_2 : array_int;
```

This operation is an extension of the parent type capabilities since an object declared of the derived type will, in this case, represent a collection of elements of the parent type. This mechanism can be compared to Object-Oriented inheritance although the terminology “aggregation” would be more accurate.

The second case consists of refining a type capability though the use of the range restriction or by declaring a “resolution function”. Using the range restriction on the parent type forces users of the derived sub-type to assign values only contained within the specified boundaries. This sub-typing operation is commonly used throughout VHDL designs to increase the level of dynamic error checking. This sub-typing is illustrated in the following statements:

```
subtype natural is integer range (0 to 2^32);
signal sig_3 : natural;
```

Similarly, the association of a resolution function, although adding an extra level of information to the type, restricts its use in an interesting way when compared to the Object-Oriented derivation. The next statements show the declaration of a sub-type called res_integer.

```
subtype res_int is resolved integer;
signal sig_4 : res_int;
```

This sub-type represents an integer with a specific property: a resolution function *resolved* is associated with it. This is recognisable as an extension of

the capabilities of the standard integer type. However, as for the ranged sub-type, this represents not only a restriction but also an inconsistency in the inheritance conformance rule (Eliens 1994). This rule states that for an assignment such as:

```
variable x : C_Parent;
variable y : C_Child;

x := y;
```

For class *C_Child* to conform to *C_Parent*, the base class of *C_Child* must be a descendant of the base class of *C_Parent*.

These remarks lead us to conclude that the VHDL sub-typing operation, as it stands, is not suitable to implement the inheritance mechanism. To achieve a proper inheritance, we therefore have to find a VHDL mechanism that complies with the conformance rule and allows, by its nature, the extension of the parent type capabilities instead of restricting them. To some extent, the record (DASC 1993) composite type allows such flexibility.

A primitive form of inheritance can be found in standard VHDL. This limited inheritance is applied to the record composite types. The record type in VHDL allows the grouping of elements of different types in a single object, they are often used for modelling of abstract elements. In Figure 2-21, we demonstrate how, in standard VHDL, an extended data structure can be defined by re-using an existing basic data structure. The *basic_structure* type is the equivalent of the parent class and the *extended_structure* is the equivalent of the child class. The *extended_structure*, which features a complex field structure (record of records), inherits, to some extent, all the attributes defined in the *basic_structure*. Moreover, multiple inheritance can easily be achieved by creating record types with two or more complex fields.

This inheritance process is interesting from many aspects. As opposed to the sub-typing operation, the derived record structure respects the inheritance rule since the following assignment is valid:

```
sig_1 <= sig_2.basic_structure;
```

However, this inheritance process is limiting. Due to the intrinsic nature of the record statement, functions (behaviours) cannot be included in the data structure.

```
type basic_structure is record
      coefficient: integer;
      data_1: bit_vector(7 downto 0);
end record;

type extended_structure is record
      basic_element : basic_structure;
      data_2: bit_vector(7 downto 0);
      data_3: bit_vector(7 downto 0);
end record;

signal sig_1 : basic_structure;
signal sig_2 : extended_structure;
```

Figure 2-21. Data Structure Inheritance in Standard VHDL

Composite types are powerful tools for modelling abstract data types and although they cannot encapsulate behaviours (functions or procedures) well, it is worthwhile investigating their abstraction capabilities with a simple design example.

Figure 2-22 shows the construction of an abstract data type using inheritance and overloading mechanisms. The base type creates a simple counter with a single operation: count. To achieve this, a record type called *attributes* is created containing the element *value* of type integer. To perform operations on this data structure, the *count* procedure is then created. This completes the counter abstract data type.

Figure 2-23 shows the creation of a bounded counter from the previously designed counter. To perform this operation, a new record type called *extended_attributes* is created containing within its data structure one complex element of type *attributes*. Another element of type integer is added to the data structure to implement the bounded counter: *boundary*. Finally, the procedure *count* is overloaded to take elements of type *extended_attibutes* as its parameter. The new procedure *count* inherits the behaviour of the previous *count* procedure by merely calling it using the appropriate type in its parameter list. This example is interesting as it shows that inheritance along with abstract data typing can be achieved in standard VHDL (encapsulation is not taken into account). This method of coding can therefore be regarded as a first step towards Object-Oriented design structuring in standard VHDL.

```
type attributes is record
      value: integer;
end record;

procedure count(value_in : inout attributes) is
begin
      value_in.value = value_in.value + 1;
end count;
```

Figure 2-22. Abstract Data Type with Records

In structural VHDL designs, a system consists of numerous instantiated components composing a hierarchical structure. This design method relates to aggregation techniques and, to some extent, to inheritance.

```
type extended_attributes is record
      basic_attributes : attributes; -- Inherits attributes.
      boundary: integer;
end record;

procedure count(value_in : inout extended_attributes) is
begin
if(value_in.basic_attributes.value<value_in.boundary) then
  count(value_in.basic_attributes);  -- Inherits count.
end if;
end count;
```

Figure 2-23. Abstract Data Type and Inheritance with Records

A component, as demonstrated in a previous section, can be seen as a means of realising an abstract data type: the entity represents the abstract interface part and the architecture represents the implementation part. As a result, instantiation of components is equivalent to Object-Oriented aggregation (often compared to inheritance in the Object-Oriented design literature (Eliens 1994)). From a pragmatic point of view, when instantiating an object as part of another design, all the properties of the given object are inherited by the new design. To illustrate this, Figure 2-24 uses a register component to create a counter via component instantiation.

The register in Figure 2-24 has an abstract interface composed of an input, output, *clk* and *reset* ports and its functionality is merely to latch the value of the input on the output on the positive edge of the *clk* signal

The counter behaviour is very similar to the register's behaviour, except that the output is fed back to the input via some logic to perform an add operation. By instantiating the register as part of the counter structure, the new design inherits all the capabilities of the register, consequently the rest of the design requires minimal coding. This scheme is adequate provided the inherited object does not need to be altered and all its functionality is required in the derived design. Obviously, if a change is required in the instantiated

object, this introduces potential hazard within the design flow since, design units will have to be recompiled in very specific orders.

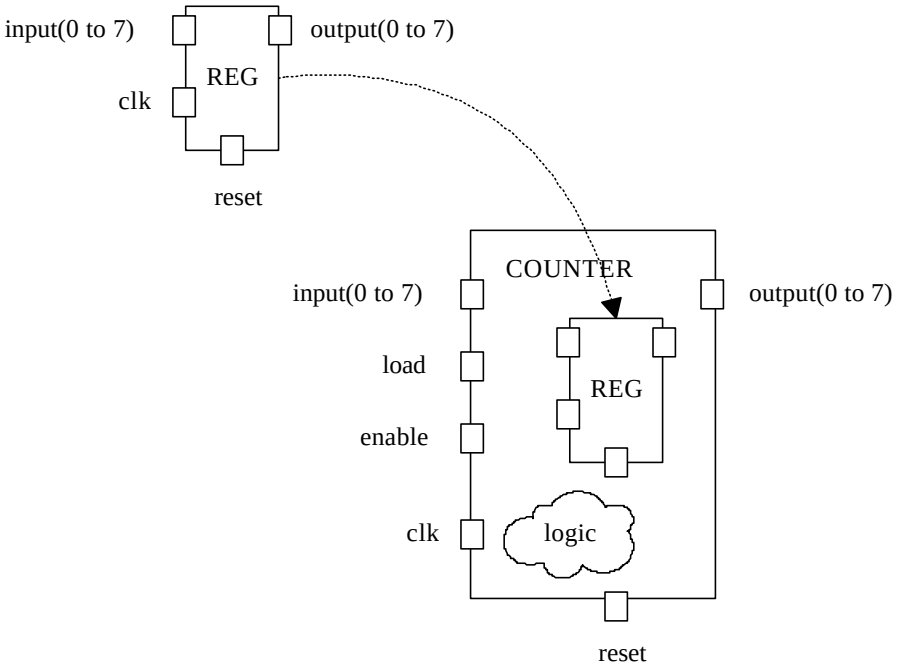


Figure 2-24. Inheritance via Component Instantiation

When taking a closer look at abstract data types, one has to realise that, they were at first developed with correctness and security in mind and not so much from a concern with extensibility and reuse. Nevertheless, it is interesting to compare the standard VHDL approach for realising abstract data types with packages and the Object-Oriented approach, with regards to the extensibility of a specification, either by adding an enquiry function or a modification function.

```
package counter is
  shared variable value : integer;
  procedure reset;
  procedure count(signal clk: in bit);
  procedure read(read_return: out integer);
end counter;

package body counter is
  procedure reset is
  begin
    value := 0;
  end reset;
  procedure count(signal clk: in bit) is
  begin
  if clk'event and clk = '1' then
    value := value + 1;
  end if;
  end count;
  procedure read (read_return: out integer)is
  begin
    read_return := value;
  end read;
end counter;
```

Figure 2-25. Initial Code for Abstract Data Type Based Component

Let us first look at what happens when we add a new modification function to the abstract data type. To illustrate this, we will once again, consider the example of a simple counter. This time, we will create a bounded counter with a lowest and highest boundary (Figure 2-25 gives the initial code for the counter).

To give an example of this new property, two instance variables of *natural* type need to be created. Furthermore, for the realisation of the new package, a function *boundary(lowest, highest)* is also required to extend the functionality of the ADT. As a consequence, to add the new function *boundary*, it is now necessary to redefine the sub-program *count* to take into account the new restriction introduced by the boundaries.

Clearly, unless special constructs are provided, the addition of a new sub-program and the extension of the *count* operation require a disruption of the

code when implementing the given abstract data type. In contrast, not surprisingly, when we wish to alter or redefine the functionality of *count* to agree with the new specification of our abstract data type, the Object-Oriented implementation does not require a disruption of the given code. We can simply add the definition of the new *count* function in a child class (we are thus performing a sub-typing operation). From that respect, adding a new *count* method corresponds to the implementation of a virtual (defined but not yet implemented) method defined as part of the abstract interface of a virtual class which gives a method interface which its sub-classes must respect. In Figure 2-26, we represent the class structure for the implementation of the *counter* abstract data type. The class *generic_counter* defines common properties found in all counters, which are *read*, *reset* and *count*. The method *count* is defined as virtual (v) to be further refined in the *generic_counter* sub-classes by its defined (d) version. The definition of virtual operations is a powerful mechanism in Object-Oriented languages with no equivalent in standard VHDL. In Object-Oriented languages such as Eiffel or C++, it is possible to declare an operation without choosing an implementation by making it a virtual operation. This transfers to proper descendants the responsibility for providing an implementation though a new declaration. Such a feature is often used when the designer of a parent class cannot yet provide a default implementation or does not want to for design methodology reasons.

For the complementary case, when adding an enquiry function (for example *get_range*) to the abstract data type, neither the package version nor the Object-Oriented class is better.

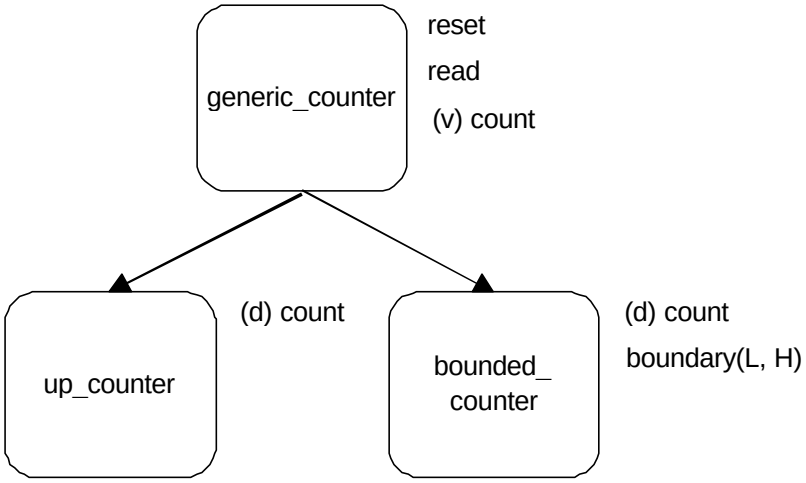


Figure 2-26. Class Structure for the Counter Abstract Data Type

Since, in a package realisation of abstract data type, the code is organised around enquiry functions, adding a new function amounts simply to a creation operation. When looking at the Object-Oriented solution, the realisation for the enquiry function is identical to the package approach.

A subtle difference may be stated though: when considering accessibility, the package-based solution offers more flexibility as instance variables can be accessed on an element basis using constructs such as:

```
counter_range := lib.Coun_package.high –lib.Coun_package.low
```

Although types were originally considered as convenient means to assist the compiler in producing efficient code, types have rapidly been recognised as a way to capture the meaning of a program in an implementation independent way. In particular, the notion of abstract data types has become a powerful device to structure large software systems. The type system introduced by VHDL through types and sub-types definitions represents an efficient means to support a procedural language. Nevertheless, as we showed in this section, such a type system does not comply with the inheritance rule and is an obstacle to the realisation of an Object-Oriented implementation.

Although the inheritance can be, to some extent, manually inferred in standard VHDL, it is not yet sufficient to implement a useful Object-Oriented language. To grasp the full extent and practicality of the techniques introduced, we need to understand polymorphism and dynamic binding. Some of the most powerful characteristics of the Object-Oriented method result from these two notions.

2.5.3 Polymorphism

Typed languages like VHDL impose quite severe constraints on the programmer. It may require considerable effort to arrive at a consistently typed system and to deal with the additional notational complexity of defining appropriate types. In practice, programmers seem to prefer working in a non-typed formalism. Languages such as Verilog are popular precisely because of the flexibility they offer since static type checking is virtually non-existent.

However, working in a non-typed formalism is often considered as unsatisfactory, as regards software reliability. To make typing practical, the support for well-understood mono-morphic typing (such as overloaded sub-programs) is provided in VHDL. More importantly, to achieve a flexible object-orientation, one must provide controlled forms of polymorphism.

Polymorphism is defined as the ability of a set of different objects to display a unique behaviour in response to the same message. This differs from overloading in standard VHDL in two ways: firstly the overloading of a sub-program is based on the definition of a set of different arguments for each new overloaded sub-program and secondly, overloading is a static operation. This section focuses on the limitations of the VHDL type system when considering polymorphism and suggests how to achieve it.

As in languages such as Pascal, C or ADA, VHDL relies on a strong type system. The VHDL language contains a variety of types that can be used to create objects (signals, variables, constants and files). The language syntax

requires that every object declared must have a type. Also, the type of two objects must be the same on both sides of an assignment statement at compile time (statically) and at execution time (dynamically). In practice, strong typing is an efficient means of ensuring code correctness. However, it is rather cumbersome when designing large systems. In a system with an important number of objects and data types, it is difficult for the designer to manage the large variety of types necessary for the description of the system. Type casting can be used to ease the assignment of related types; However, its application scope is limited.

Another construct that VHDL provides to soften its strong type nature is overloading. Sub-program overloading allows the designer to write multiple versions of a sub-program with the same name, but the number of arguments and return value can be different. This ability prevents the designer from having to find a unique name for equivalent sub-programs dealing with different types. Overloading is an interesting feature for building abstraction and conciseness in the code but, as for the previously considered constructs in VHDL, the evaluation for the matching sub-program is performed at compile time. To that extent, true polymorphism (dynamic types) cannot be achieved in standard VHDL.

Ecker W. has suggested in (Ecker 1996), that only a form of static polymorphism can be achieved in standard VHDL (see Figure 2-27). A component can be seen as a link to an actual entity-architecture pair relying on the configuration statement to mimic polymorphic behaviour. This polymorphic behaviour is said to be static since the binding of the actual entity-architecture pair is performed at elaboration time instead of execution time. In the case of the shift register, a design using this device has to declare a component to materialise an instance of this class. The next step is to create a configuration for an instantiated component (object) by selecting an entity-architecture pair. This last operation is carried out after the design has been

compiled and any architecture association can be made at this point. Furthermore, the association of a different entity remains possible provided that the component and the associated entity have compatible interfaces.

By allowing different versions of a design to be attached to a given component interface, we achieve a polymorphic operation. However, since this selection via the configuration is unique for a given design simulation or synthesis, this polymorphic operation is static instead of dynamic. This difference is of significant importance when considering the advantages of dynamic polymorphism combined with inheritance over static polymorphism.

As stated earlier, abstract data types are rather inflexible and inconvenient for specifying complex systems. To achieve flexibility, one has to organise types in a design by expressing the commonality between them. This is achieved through classification. The notion of class in Object-Orientation supports such an organisation though the inheritance mechanism. Although, until now we presented inheritance as a combination and enrichment mechanism, it also enables the definition of flexible entities that may become attached to objects of various forms at run time, thus introducing dynamic polymorphism.

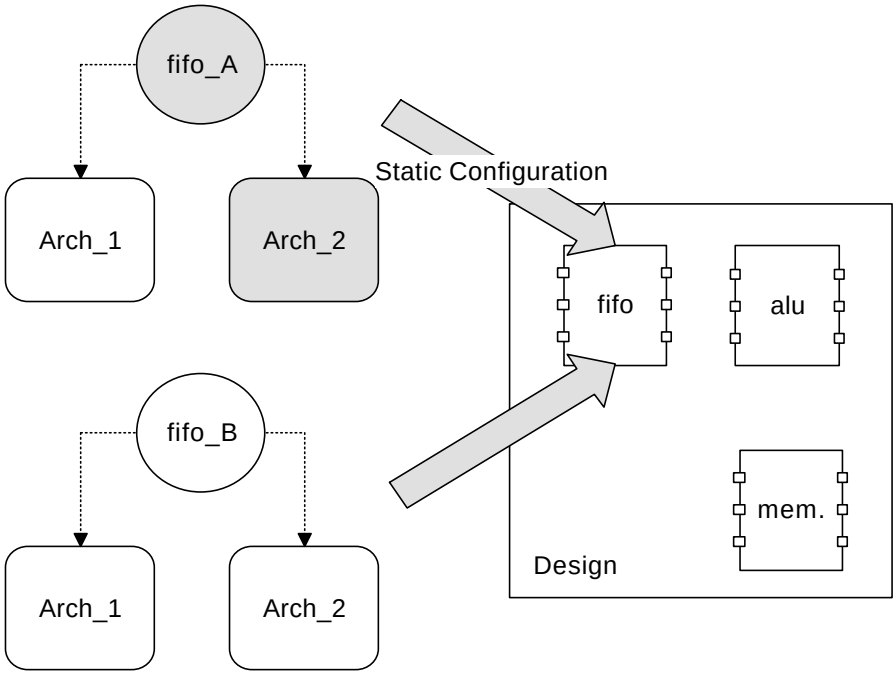


Figure 2-27. Static Polymorphism through Configuration

This characteristic can be compared to the static typing rule in standard VHDL in the sense that, by convention, an assignment of the form *a := b* is permitted if a and b are of the same type. Nevertheless, dynamic polymorphism brings a new aspect to the assignment convention by stating that "if a and b are of different types, but class b is a descendant of class a, the assignment is still valid". This corresponds to the idea that a value of a more specialised type may be assigned to an entity of a less specialised type, but not the reverse. This possibility becomes even more powerful when combined with the redefinition of inherited methods. The following example (shown in Figure 2-28), demonstrates the advantages of dynamic polymorphism. In this example, three objects are declared: o1, o2, o3; the last two objects are of a derived type of *reg*, respectively *odd_reg* and *even_reg*.

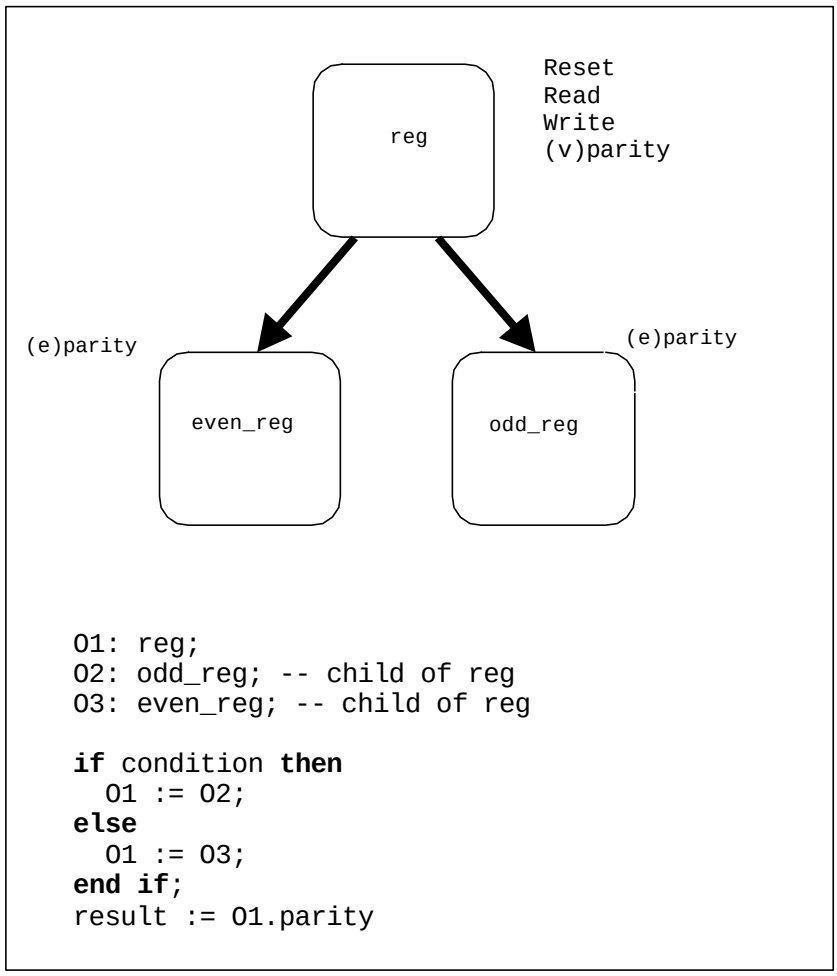


Figure 2-28. Benefits of Dynamic Polymorphism

Both derived types define a version of *parity* to perform an even or odd parity depending on the object type; the version to use in any call is determined by the run-time form of the target.

The assignment *o1:= o2* is valid because of the assignment rule defined for polymorphism; if *condition* is false, o1 will be attached to an object of type *even_reg* for the computation of the operation *c1.parity*, which will thus use a even parity check. In the opposite case, c1 will be attached to an object of type *odd_reg,* therefore the computation c1.parity will perform an odd operation. This is a dynamic binding operation.

Dynamic binding provides a high degree of flexibility in designs. The advantage for the user is the ability to request an operation without explicitly selecting one of its forms. The choice only occurs during the simulation execution (referred as 'run time'). This is an important requirement for the building of large systems, in which many components must be protected against changes to other components (abstraction). In standard VHDL, overloading and the ability to create generic designs, do not bring any improvements in this respect. They do not support a programming mechanism in which a client module may issue a request to perform an operation on a VHDL object (signal, variable) that depends on the form of this object at run time, and permitted here by inheritance, redefinition, polymorphism and dynamic binding.

These design techniques support a development mode in which every component is open and incremental. When the designer wants to reuse an existing class but needs to adapt it to a new context, one can always define a new descendant of that class (with new properties and redefined ones) without changing the original.

The power of polymorphism and dynamic binding demands adequate controls when considering type checking. Type checking is performed statically at compile time in the VHDL language. In an Object-Oriented language, dynamic type is checked at run time. When a client sends a message to an object, the object treats the call provided that it has a corresponding method defined in its class. However, if the requested object does not have the required method, a run time error occurs.

Polymorphism is not exclusive to Object-Oriented languages. For example, languages such as Pascal, and FORTRAN allow an implicit conversion between certain types. However, in VHDL, an implicit conversion is not permitted; only a type casting operation can be used on related types. Polymorphism, including such conversions, relieves the programmer from the

rigidity imposed by typing. In other words, polymorphism is used to increase the expressiveness of the type system. A possible disadvantage when adding polymorphism to a strongly typed language might be that program understanding becomes more difficult since many choices are now made by the dispatching mechanism instead of being written out explicitly. In an Object-Oriented version of VHDL, both dynamic and static type checking need to be handled to maintain an acceptable level of error recovery. The static type checking will ensure type consistency for assignments and the dynamic version will select the appropriate form of an operation depending on the run-time type of an object.

From this study of the VHDL language we highlighted the limits of the existing semantic and mechanisms for implementing the design objectives defined in section 2.5. This leads us to look at attempts from the design community to overcome the VHDL restrictions. The specific weaknesses of the proposed extensions will be discusses in chapter 4.

2.6 Existing Extension Proposals

Although the main guidelines for the definition of an Object-Oriented VHDL extension have been drawn, actual implementations have not yet been defined. As Object-Oriented VHDL is getting more popular, different suggestions for possible Object-Oriented extensions have been made (Covnot 1994), (Mills 1993), (Willis 1994), (Zippelius 1992). Along with those suggestions, a number of actual language extensions were created (Shumacher 1995), (Ashenden 1998), (Radetzki) (Cabanis 1996) . These will seek to help VHDL reach higher levels of abstraction, reusability and maintainability.

2.6.1 The Vista Proposal

As demonstrated in section 2.5, VHDL lacks adequate support for abstract data-typing operations. Consequently, all extensions to the language will

primarily focus on either extending the existing VHDL abstract data type capabilities or implementing new abstract data types.

With the Vista proposal, a new abstract data type has been added to the language in the form of a "component object" (Vista 1994), (Ramesh 1994). With the component object paradigm, the designer can build abstract data types and instantiate objects of that type within the VHDL code. The abstract data type is defined as an entity object and architecture pair. The *entityobject* will declare the interface of the abstract data type, i.e. the operations, while the architecture will declare the behaviour and attributes. Unlike standard VHDL, the proposed *entityobject* is a generalised kind of entity that can interact with standard entities from the same hierarchical level. The Vista entity does not rely on ports (of standard or enumerated types) and generics but on high level messages used for concurrent object communications.

The *entityobject* allows for inheritance: all elements of an *entityobject* are exported when inherited. This includes ports, generics, local declarations and concurrent statements found in the associated architecture. Similarly, in the proposed extension, standard entities can also be inherited in the same way as an *entityobject* (apart from operations). Operations declared within the entity declarative part are used as abstract interfaces allowing message passing. Like most object-oriented languages, operations are used for communication among objects and for access to instance variables. The communication scheme is based on a synchronisation mechanism, which is a more convenient means than the existing signal-based communication. Any caller sends a message to a client object and is blocked until the operation is completed. Messages are processed sequentially and are placed in a queue if the target object is servicing too many operations at that time. Once an operation has been processed, the following one in the queue is run.

Polymorphism is supported in the Vista proposal: different objects can have identical messages on their abstract interface.

Specific constructs have been added to the VHDL language to handle the concurrency aspect of object accessing. These concurrency control mechanisms are based on the distributed processing model and the ADA rendez-vous model. The distributed process aspect is used for the deferment and queuing of processes; furthermore, priority can be assigned to operations, in order to perform an automated sorting on incoming message calls. The ADA rendezvous model allows the selection of a specific message over a number of incoming calls, or the selection of multiple messages to be active at the same time.

A more in depth study of the language semantics and abilities will be given when we will compare our proposed extension to other existing extensions in chapter 4.

2.6.2 The Oldenburg and SUAVE Proposals

The Oldenburg and SUAVE proposals are very similar in their concept and implementation since both proposals are inspired from the ADA95 (Taft 1993) language. This is a reasonable approach since the original designers of the VHDL language borrowed many of the features and language philosophy from ADA, more specifically in areas such as sequential statements, sub-programs, design units and more importantly the type system. Similarly, concepts of operator and sub-program overloading, as well as complex data structures (such as records and access types) are part of the ADA language. Since the early 1990's, ADA has undertaken radical changes (Seidewitz 1991), (Atkinson 1990), (Nelson 1992) to adapt to new design methodologies based on Object-Orientation.

This research lead to proposals for protected objects, hierarchical libraries and support for Object-Oriented programming. A key part of the language extension for supporting object-orientation is the ability to define new types in terms of existing ones.

The new type inherits the operations and fields that are defined for its parent type. Furthermore, a new type can redefine or add features to the inherited ones. In addition to type inheritance, the Ada 95 language allows the definition of classes of types accessible by sub-programs containing arguments of these specific types. The selection of a specific operation on a specific class type is performed at run time allowing dynamic binding.

To implement type inheritance, the two proposals suggest the use of "tagged records". A tagged record has identical properties to the standard VHDL record. In addition, subtypes of tagged records can be declared and a specialisation mechanism is introduced to allow the addition of new fields. Associated with the tagged records, are procedures to implement the behaviour part of the abstract data types. Encapsulation is provided by the package visibility rules in which both tagged records and related procedures have to be declared. Procedures can also be redefined for derived types as well as added. Polymorphism, or more correctly dynamic binding, is achieved via the creation of heterogeneous object containers. These containers are defined using a class-wide type, which allows a field to adopt any type contained in the hierarchy of the class type.

More details on the two proposals will be given in chapter 4.

2.6.3 The Objective VHDL Proposal

The abstract data type in the Objective VHDL proposal differs significantly from both the VISTA and the Oldenburg/SUAVE proposals. The creation and utilisation of a *class* ADT in this proposal resembles what is found in VHDL with the *record* composite type. The *class* ADT will contain both class's attributes and method (functions and procedures). As for VHDL's records, the *class* can be declared inside architectures and packages. However, when declared, the *class* will be split in a header and body part.

The *class* supports single inheritance from which *class attributes* and methods can be inherited providing they have been declared as part of the class's header. The Objective VHDL proposal allows methods to be overridden in child classes by redefining functions or procedures using identical declarations to the one found inside the inherited class.

The instantiation of an ADT takes the form of a *variable*, *signal*, or *constant*. Each implementation has specific application restrictions.

Polymorphism is achieved through the existence of class wide types. This resembles the mechanism found in both the Oldenburg and SUAVE proposal. An object of class wide type is allowed to be assigned different class type objects that are part of its class wide tree.

Further implementation details will be given during the comparative study of all the proposed extensions in chapter 4.

2.7 Conclusions

This Chapter examined the different design methods used to specify and implement current hardware design. We demonstrated the limits of functional decomposition and SA/SD. This led us to conclude that an improved design method was required for the successful creation of multi-million gate ASICs. The IEEE OOVHDL study group has acknowledged those limits and consequently, has defined a set of requirements for the implementation of an improved design methodology. The proposed requirements are supported by a number of design objectives/changes that VHDL should undergo, to comply with the defined requirements.

A study of the VHDL semantics and mechanisms allowed us to identify the limits of the language for implementing object-orientation support along with, the areas of the language where changes would be the most suitable.

Having looked at VHDL's limits, we presented the solution proposed in the Vista, Oldenburg, SUAVE and Objective VHDL language extensions. All proposals support the design objectives set by the IEEE OOVHDL study group; However, aside from the SUAVE and Oldenburg proposals that are similar, all proposals use different semantics and mechanisms to bring object-orientation to the VHDL language.

We can conclude from this study that the creation of any language extension is not a straightforward operation. Nevertheless, it is important to establish whether a language allows a clear interpretation of the constructs introduced; whether constructs supporting Object-Orientation are independent of other constructs of the language; whether an efficient implementation of these constructs is possible; and whether the language is kept minimal, that is without superfluous constructs.

Chapter 3

LANGUAGE DESIGN

3.1 Introduction

The idea of an Object-Oriented version of VHDL is gaining interest within the design community (Oczko 1990), (Douglas 1994), (Perry 1992), (Shumacher 1996). Object Orientation is not a set paradigm but rather a particular approach to the design methodology. To some extent, Object Orientation only defines critical issues found in the process of building large designs and enables designers to deal with them.

The previous chapter examined how encapsulation, late binding and inheritance could be implemented in VHDL based design using the original semantics. We showed that this increases productivity when used on an object-based design, but also that the existing VHDL semantics are inadequate to address this level of abstraction.

Quality in hardware descriptions is a combination of several elements; the language extension design concentrated on the factors which, in the current state of industry, need to be improved. One of the main factors is reusability, or the ability to produce components that may be used in many different applications. Another is extendibility: it is notoriously difficult to modify hardware systems, especially large ones.

Among quality factors, reusability and expandability play a special role: satisfying them means having less code to write and consequently more time to devote to other important goals such as efficiency, ease of use or consistency.

The third important point is reliability. Techniques such as assertions, disciplined exception handling and static typing, enabling developers to

produce hardware descriptions with fewer errors, are part of the approach to the engineering of better designs.

In this chapter, using the requirement set by the IEEE OOVHDL study group (see section 2.4) we will examine means of extending the existing VHDL semantics in order to achieve object orientation. The first part of this chapter will explain the language design decisions; the second part will then detail the chosen semantics.

3.2 Language Design Decisions

3.2.1 Abstract Data Type Specification

Chapter 2 highlighted the weaknesses of the VHDL language in terms of data abstraction to achieve true Object-Orientation. To solve this problem, our extension offers the creation of a new kind of composite type alongside the already existing *record* and *array* types, which we have called *class*. The *class* type can be considered as a real type and takes a natural place in the VHDL type system. Regarding these new abstract data types as real types has numerous important benefits. The realisation of abstract data types means that they may be treated as any other value in the language, for instance being passed as a parameter. In contrast, syntactic solutions such as the *module* in Modula-2 or the *package* in ADA do not allow this.

Furthermore, in strongly typed languages, the objective of a type system is the prevention of errors. Therefore, if the type system lacks sufficient expression, adequate control for errors may become over restrictive. Generally, the more expressive the type system, the better is the support that the compiler may offer. In this respect, associating constructors with types may help in relieving the programmer from dealing with cumbersome tasks such as initialisation of complex data structures. Objects, in contrast to *modules* or *packages*, allow automatic initialisation of instances of abstract data types, and allow the programmer to avoid an error-prone routine.

Another area where a more complete type system may improve the designer's task, is the association of operations with objects. A polymorphic type system is needed to provide automatic selection of virtual functions and perform efficient function overloading. This is a useful mechanism to control the complexity of a program.

The VHDL language as it stands, is strongly typed to ensure readability and reliability. Our suggested extension relies on the inherent strongly-type system of VHDL but also provides a class type with more complex reference rules, since objects of class types are defined as 'referenced instances'.

Our new referenced types differ from the basic VHDL types: an object declared of a referenced type does not represent a value of type but a reference to an object containing a value. Operations on this object are defined as part of the properties of the referenced type. The declaration of a class type will consists of three parts:

1. A class name.

2. A declaration of the variables available to all instances.

3. The methods used by instances to respond to messages.

3.2.2 Visibility

The declaration part of a class will have a visibility over the declaration found within the scope of the Package, Architecture and Block depending on where the class has been defined. This visibility can be seen as going against the encapsulation principle and therefore should be used with caution.

Although it is dangerous from an encapsulation point of view, allowing visibility to span the class boundaries has numerous benefits. One the main

benefits is highlighted when using packages in a design; where common declarations such as type definition can be shared.

3.2.3 Class Genericity

The generic nature of classes is an important aspect since the designer may want to create a class that represents a group of instances but to specify size or timing information at elaboration time rather than at design time. This mechanism is similar to the mechanism used with VHDL components. Although class genericity can be achieved with *templates* like C++, the concept of generic values rather than generic types was introduced in standard VHDL. For this reason, the *template* solution is avoided to prevent language constructs from overlapping.

3.2.4 Class Tailorability

The proposal relies on a mapping mechanism to enable the designer to adapt classes to its design requirements. The use of this mapping operation is related in principle to the alias-based solution found in VHDL'93. The VHDL'93 version of the language broadens the use of aliases. However, in our opinion, aliases are only used to give a second name to an existing object rather than redefining it. This motivated us to favour the mapping construct extending it to a greater use than simply ports and generics.

As part of its declarative part, each class will be allowed to define mapping clauses to adapt names of inherited features to the local context of the new class. Name mapping is particularly useful in two cases. With renaming, it is possible to correct any name clash occurring through the multiple inheritance process. A name clash occurs when two or more inherited features have identical names, and would generally make the class invalid if the features were not renamed. Renaming also enables a class to offer its inherited features to its users and descendants under a terminology appropriate to its own

context, rather than to the context of the parents from which it inherited them. In other words, it helps make sure that, aside from offering the right features, they are presented under the appropriate feature names.

It is important to understand that the proposed renaming mechanism does not change any of the inherited features, but simply changes the name under which the clients and descendants of a class will know these features. After replacing a name of an inherited feature, the old name can be reused for various purposes. It could be used as an identifier for a new feature introduced by the new class itself. It could also be used for a feature inherited from a parent of the new class other than the present one which has a feature with the same name. In some cases, as one inherits a method or an attribute from a parent, one may wish to discard the inherited implementation. This is the process of redefinition, which turns a new feature into an effective one.

3.2.5 Class Instance Variables

Instance variables are elements used to store the class properties. For the implementation of instance variables, we will be using the VHDL's variable mechanism. Instance variables, unlike variables found in VHDL's sub-programs, will retain their values until changed. The choice of blocking assignment over non blocking assignment will be discussed in more details in chapter 4 when we will compare the different OOVHDL implementations.

3.2.6 Class Methods

Methods, in the proposed extension will be similar to VHDL subprograms. However, they will differ in the capability they offer when used together with inheritance. From a design perspective, inheritance provides a mechanism for code sharing and code reuse. Operationally, the power of inheritance in the proposed extension comes from the use of virtual methods and virtual classes.

A virtual class will be declared if the designer plans to include one or more features that are specified but not implemented; descendants of the class will later on in the design process provide the implementations. This is useful to describe groups of related concepts, or not fully understood concepts. For example, stacks describe data structures that are managed sequentially, without indicating any specific implementation. Proper descendants of this class such as FIFO will describe the concrete sequential structure. Classes and methods may in fact remain virtual for a long time, providing a high level notation for system analysis and design. Virtual methods fully exploit this principle.

3.2.7 Object Instances

Object instances might be declared in packages, architecture blocks and processes. This will allow designers to define accurately the scope of accessibility of each object inside a design. During the declaration of a specific object, the designer will be able to specify its initialisation method and configuration.

One of the most significant advantages of the use of abstract data types over conventional VHDL objects (signals and variables) is the ability to dynamically allocate or de-allocate resources for complex components. The advantage of dynamic allocations is even more obvious when considering the synthesis of dynamically re-configurable designs (Rosenberg 1994), (Vasilko 1999), (Faura 1997). With the emergence of new technologies, it is now permitted to create designs that can evolve while they are being used. In order to implement dynamic allocation and de-allocation of objects, we will use the already available VHDL access type. Using access types ensures a better integration of our semantic changes within the existing language.

3.2.8 Object Initialisation

The initialisation of an object will either be the default style (type'LEFT) found in standard VHDL or via creator methods. The latter initialisation

method can be found in programming languages such as C++. Creator methods will be used when an object needs to go through a complex initialisation process before it can be used.

3.2.9 Object Configuration

Objects instantiated inside a design, will have optional configuration parameters. Those parameters will be used to set generic values (set at elaboration time) as well as interfaces to interact with the external world. The interface map declaration is used during the object declaration process as a means of binding the argument used inside the impure methods to the argument found inside the client application. This process, although important in the simplification of concurrent Object-Oriented description, should be used with caution. The methods, impure by definition, have access to objects defined outside their parameter lists, inside the interface list. It is important to consider these interface signals as control lines as opposed to data lines. Data information should only be carried by parameter list arguments in order to ensure a proper design encapsulation.

3.2.10 Scalar Types

In standard VHDL, records of records and records containing arrays are valid constructs and are often used to build abstraction in the design. With the introduction of class types in the proposed extension, the definition of a record with a field of a class type represents an issue. Once the record type has been created, it is possible to declare either a signal or a variable of this record type. However, elements of object types do not comply with the same assignment rules as for signals and variables therefore the creation of a record type containing a element of a class type is not acceptable. Although this might appear to be a restriction in terms of how well the extension fits the 1076-93 version of the language, it is important to make a few remarks. Such a restriction in the use of certain types in VHDL is not new to the language and types such as access types already have numerous restrictions concerning

their use. Furthermore, the extension to the language offers a more powerful mechanism than the record type to build data abstraction and, ideally, the use of record should be in many cases replaced by the use of classes. The creation of arrays of ADTs will be fully supported by the proposed extension. Arrays of ADTs will be used to implement dynamic types.

3.2.11 Message Passing

Different behaviours will be achieved with message patterns depending on the method specification style. The three main behaviours of a method are identified in our proposal:

1. *Events* are used for synchronisation purposes. They are usually encoded by a message without parameters like 'reset_state'. Here the method indicates the type of event.

2. *Commands* are requests to an object to perform a local function. These may also be transferred by messages, either with or without parameters like 'StopCount' or 'load("01010")'. Again the method indicates the command given.

3. *Requests* ask an object to return some information which is present in (or can be obtained by) the receiver. They need two messages. The first issues the actual request. The second one contains the reply, going in the opposite direction. In a normal mode of operation, the sender waits for the reply. The method may or may not have arguments but will have a return type. Requests will take the form of 'GetContent' or 'evaluate("10010")'.

Some code maintainability aspects motivate the choice of the ‘dot’ notation over a standard VHDL sub-program call. When using the notation: X.state , it is not shown whether in the class of X, state is an attribute or a method without arguments. This ambiguity is intentional. A user of the class X does

not need to know how a state is obtained: the state could be stored as attribute of every object of class X, or computed via a method from other attributes. Choosing between these techniques is the role of class X. Since such implementation choices can often change over the lifetime of a project, it is essential to protect users against their effects.

Within an object-oriented environment, object can be defined as either active or passive. An active object will be able to trigger other objects' activity without having to be triggered itself. In contrast, a passive object will display its behaviour only after being triggered. The VHDL environment is based around concurrent and sequential behaviours. An important choice is whether to distinguish between active and passive objects or whether to support only one kind of object. Another important point is how the activity of objects is to be defined.

Objects in the proposed extension might be referenced in sequential statements as well as in concurrent statements. However, multiple concurrent messages are not allowed due to usability issues. In a VHDL environment, the execution of concurrent statements does not follow any order by definition (with the exception of postponed processes). This remark leads us to the study of a simple example illustrating the issues encountered with concurrent statements associated with object access.

```
object_1.write(A*4);
object_1.write(A*2);
```

Considering that these two assignments are concurrent blocking assignments, after a change of the value of A, it is impossible to determine whether the object_1 will contain the value A*4 or that of A*2.

The alternative to blocking assignments is non-blocking assignments; this time, all the assignments are performed on the last delta cycle. However, the latter example will create a situation where two commands are scheduled to

be executed at the same time on a common instance variable. This problem is solved for signals through a resolution function. However, this solution is too basic to be valid for objects with numerous messages and complex behaviours. As a result, objects will not handle multi-concurrent calls to the same-shared resources. However, concurrent calls to distinct resources within the same object are allowed.

The proposal will differentiate blocking from non-blocking method calls through the use of two distinctive call semantics. A blocking method call used within a process will have to be fully executed (including any *wait* statements) before executing the next statements in the sequence. Unlike blocking calls, non-blocking method calls when invoked will trigger concurrent activities and will not be required to be fully executed prior to moving to the following statements in the sequence. The principle of spawning concurrent activities inside a process is not foreign to the HDL world. A similar mechanism is found in Verilog with the *fork-join* mechanism.

3.2.12 Object Copy and Assignment

Unlike VHDL's signals, variables or constants, no arithmetic or logical operators are predefined for objects of class type. The user will either have to use object access methods or define operators for specific object types. Furthermore, the object assignment '==' has been defined as a clone copy of a reference as opposed to the copy of a value. The clone copy is necessary to (Meyer 1992) allow polymorphic behaviour to be expressed using dynamic typing. To perform clone copying however, simple rules for direct object reattachment have to be followed. The following example shows a limitation of direct reattachment using the clone copy.

Assuming a data structure and object declaration of the form:

```
type counter_array is array(natural range <>) of counter ;
object counter_list : counter_array(0 to 3) ;
```

Where *counter* is the parent class of the *wrapping_counter* and the *modulo_counter* class. If the first instance of the list is an instance of the *modulo _counter* type and the designer needs to obtain its boundary value (*max_value*), the following code might be written:

```
signal max_value_tmp : integer ;
object mod_counter_1,
       mod_counter_2, mod_counter_3 : modulo_counter ;
object wrap_counter_1, wrap_counter_2 : wrapping_counter ;
....
counter_list == mod_counter_1 &  mod_counter_2 &
                wrap_counter_1 & wrap_counter_2 ;
mod_counter_3 == counter_list(0); -- line 1
max_value_tmp <= mod_counter_3.max_value ;  -- line 2
```

However, since the assignment rule defined in section 2.5.3 states that only objects of an identical class or derived class can be assigned to a target object, line 1 is illegal. In line 1, the designer is assigning an object *counter_list(0)* of a type counter (from its original definition) onto an object of type *modulo_counter*, lower in the class hierarchy. This creates a type mismatch error.

Since such examples are easy to produce, one might tend to think that strong typing represents an annoying problem. This is inaccurate: in well-structured Object-Oriented designs, the problem only arises when dealing with complex systems. If it is known that an element of such data structure has special properties, it is then possible to access it separately through an object of the correct type.

In order to perform the required operation in the example, the designer can merely replace line 1 by the statement:

```
mod_counter_3 == mod_counter_1 ;
```

From this operation, both *counter_list(0)* and `mod_counter_3` are pointing to the *mod_counter_1* data structure as required and the assignment is valid. Similarly, the object equality in our proposed extension is used to determine if two objects are field-to-field compatible. The result of the expression:

`object_1 = object_2` will be valid if *object_2* is of the same type or a descendent of *object_1*. The result will return Boolean *true* if each field of the *object_2* data structure is identical to those in the *object_1* data structure (following the standard VHDL comparison mechanism). When considering complex objects, for efficiency reasons, the expression will be *true* if the two fields are pointing to the same object. However, the designer (to meet alternative requirements) can redefine the equality with the class that defines *object1*.

Besides the equality operator, two methods will be defined by the system for every created class: *Copy* and *DeepCopy*. All calls to `Object_1.Copy(Object_2)` will copy every field of *Object_2* onto the corresponding fields of *Object_1*. *Object_2* must conform to the type of *Object_1*. Thus, in general, *Object_2* will contain the same amount or more fields than *Object_1*.

In order to perform a copy of all the fields of an object (across the hierarchy), the *DeepCopy* mechanism is available. A deep copy will replicate the complete data structure, starting from the top level, down to the lowest level.

3.2.13 Encapsulation Control and Visibility

Encapsulation will be used by the designer to specify/control the visibility of class properties. The ability to indicate one of the three levels of encapsulation, namely *private*, *public* and *restricted* provides an improved access control over the existing mechanism (visible/not visible) in VHDL 93. These types relate to types found in Object-Oriented programming languages such as C++. The *private* type disables any access to the defined class property by any external requests or by any class methods of potential inheriting sub-classes. *Public* type declarations are the complement of *private* type declarations. A property declared of a *public* type will allow access to any external class requests as well as by any class methods of potential inheriting classes. Although dangerous as far as encapsulation is concerned, we consider

this aspect is important because it helps to reduce the code size of Object-Oriented designs. In order to assign values to *public* instance variables, only a blocking mechanism is supported. Nevertheless, multiple concurrent assignments to instance variables is not permitted. The *restricted* type is used when properties need to be hidden from external requests but accessible by any class methods of potential inheriting sub-classes. If no encapsulation type is specified for a class declaration element, a *restricted* type is assumed. However, for pragmatic reasons, the default encapsulation kind for methods is set to public.

3.2.14 Inheritance

Like any object-oriented language, the proposed extension will support single inheritance. A question of interest is whether a language must support multiple inheritance. Clearly, there is some disagreement on this issue (Waldo 1991), (Cargill 1991), (Armstrong 1994). For example, Smalltalk-83 does not support this type of inheritance. The Eiffel language, on the other hand, supports it. For C++, multiple inheritance was introduced at a later stage. At first, it was thought to be expensive and not really necessary. Closer analysis on the issue revealed that the cost was not excessive. The issue of multiple inheritance is still not resolved completely. Generally, it is acknowledged to be a powerful and, at the same time, a natural extension of single inheritance. For that reason, the proposed extension will support multiple inheritance along with powerful adaptation mechanisms.

As in common OOPLs, the encapsulation control will also be used during the inheritance process. The intention of encapsulation control at this stage is to restrict the visibility of any inherited *public* and *restricted* defined properties. In designs, it is often useful to use inheritance for the purpose of code sharing only, with no intention to declare sub-typing relations between two classes. In this extension, one will declare a class to be either publicly inherited or privately inherited. The latter will be used when only code sharing is intended

and does not affect the type system, whereas the former will be used when a subtype relation is needed.

A *private* inheritance will lead *public* and *restricted* inherited (parent) class properties to become *private* to the inheriting (child) class. A *public* inheritance will assimilate any *public* inherited properties as *public* and *restricted* as *restricted*. A *restricted* inheritance will lead *public* inherited (parent) class properties to become *restricted* to the inheriting (child) class. It should be understood that a well-designed inheritance hierarchy would include few occurrences of classes hiding some of their parent's features. If one has to constantly work accordingly to the parent designer's decisions, then it is necessary to consider improving the inheritance structure (assuming that it is allowed). To overwrite the inheritance default kind (*restricted*), the user defines the kind before the name of the inherited class.

Other Object-Oriented languages such as Eiffel allow the redefinition and removal of inherited features in the child class. This represents a powerful means of specialising a child class to its use. However, the redefinition as introduced by Eiffel does not agree with the VHDL language philosophy and is replaced in the proposed extension by a mapping construct. The removal of definitions is not directly supported in this extension, for the same reasons as for the redefinition; although from a methodology point of view, we believe that a controlled inheritance mechanism agrees better with the object-oriented idea of class inheritance and specialisation.

3.2.15 Dynamic Types and Polymorphism

The reuse of code is one of the most important aspects of inheritance. The principle underlying the efficient reuse of code consists of stating the difference when programming. This means that one has to redefine features of the derived class, which are added to, or different from what is provided by the base class. To fully exploit this principle, we need to introduce the

extension “virtual methods”. Virtual methods are methods for which dynamic binding, or more precisely dynamic types, apply.

The concept of dynamic types will be added to the language to soften the limiting of VHDL’s strong typing nature. The dynamic type of an object or an expression, at some stage of execution, is the type of the object to which it is attached. This should not be confused with the type called the static type. This will provide the ability to have more than one dynamic type. The new type rule means that possible dynamic types for an object must conform to the static type of this object. This is how polymorphism is implemented and kept under control by the type system.

A possible disadvantage of this new type rule might be that program understanding becomes more difficult since many choices are now implicitly made by the dispatching mechanism instead of being written out explicitly.

3.2.16 Documentation

The use of attributes will be extended to provide users with information regarding the object’s state. Using attributes, designers are able to inquire whether an object is currently processing any information or whether it is idle. Attributes will also allow designers to inquire whether a particular method is in use, waiting for a valid pre-execution condition or whether it is idle.

Attributes in the proposed extension provide a powerful means for achieving formal specification when combined with assertions by expressing correctness conditions. Assertions in the proposed extension can play several roles: they help in the production of correct and robust code, supply high level documentation, provide debugging support, and serve as a basis for exception handling. Assertions will be used to express the specification of components: to give an indication of what a component does rather than how it do it. This information is essential to build a component that will perform reliably (Meyer 1992), to use it and to validate it. In addition to the basic assertion

mechanism, the proposed extension will feature pre assert and post assert statements.

Pre and post conditions will be used to define the requirements that clients of an object must meet and obligations an object has when executing a method. This might be seen as a contract between the object and its potential clients.

3.3.1 Abstract Data Type Specification

3.3.1.1 Class Specification

Class declarations in the proposed extension, are similar to VHDL package declarations since both class declaration and package declaration define an abstract interface. The definition of a class requires a class declaration having the syntactic form:

```
type_definition ::=
      scalar_type_definition
      |composite_type_definition
      |access_type_definition
      |file_type_definition
      |class_type_definition

class_type_definition::=
      class [ use ( class_list ) ]
            class_declarative_part
      begin
            class_assertions
            |method_specification
            |method_body
      end class [ class_simple_name ];

class_list::= [ encapsulation_kind ] class_name { ,
              [ encapsulation_kind ] class_name }

class_declarative_part::=
      class_header
      class_declarative_item
```

An example class declaration with a declarative part is given in Figure: 3-1.

```
type decoder is class
      type opcode is (add, sub, mult, fetch);
      instance variable data : opcode;
begin
end class decoder;
```

Figure 3-1. Class Declaration

Although it is intentionally similar to a package declaration, the class declaration differs from the package declaration insofar as elaboration may result in multiple instances of a class type.

The class header of a class type definition declares objects used for communication between the abstract data type and its environment. A class header may contain a generic clause as well as feature mapping constructs.

The class header in the solution we suggest has the syntactic form:

```
class_header::=
      [generic_clause]
      [interface_clause]
      [feature_map_clause]

generic_clause ::=
      generic ( formal_generic_list );

interface_clause ::=
      interface ( formal_interface_list );

feature_map_clause ::=
      feature map ( feature_association_list );
```

The formal generic list in the generic clause defines generic constants whose values may be determined by the environment. An example class declaration with a generic declaration is shown in Figure 3-2:

```
     type counter is class
        generic (size: integer := 8);
         instance variable value : bit_vector(size-1 downto
0);
     begin
     end class counter;
```

Figure 3-2. Class Declaration with a Generic Declaration

The generic of an abstract data type is defined by a generic interface list (defined in the LRM section [1.1.1.1]). Each interface element in such a generic interface list declares a formal generic. Several rules govern the use of generics. The corresponding actual in a generic association list may specify the value of a generic constant. If no such actual is specified for a given formal generic (either because the formal generic is unallocated or because the actual is *open*), and if a default expression is specified for that generic, the value of this expression is the value of the generic. It is an error if no actual is specified for a given formal generic and no default expression is present in the corresponding interface element. Similarly, it is an error if some of the sub-elements of a composite formal generic are connected and others are either unconnected or unallocated. During the inheritance process, generics of parent classes are inherited by their child classes. If two generics of the same name are given a default value, the latest definition will be assimilated as the valid one unless it is re-mapped via the feature map. These properties regarding the inheritance of previously defined generics allow the designer to build complex and fully parameterisable elements through a simple mechanism.

Following the generic declaration is the interface declaration. Interfaces are defined as signals that are used for control purposes. The interface declaration is a convenient means of declaring external parameters that will be accessible

by all internal methods without further declaration as part of their parameter lists. Generally, these interfaces will be used to declare synchronisation signals. The BNF of the interface declaration is defined as follows:

```
formal_interface_list::= interface_signal_declaration

interface_signal_declaration ::=
```
(see section [4.3.2] LRM)

An example of a class interface declaration is shown in Figure 3-3.

```
type counter is class
   interface (signal CLK: bit);
   instance variable value : bit_vector(7 downto 0);
begin
end class counter;
```

Figure 3-3. Class Declaration with an interface Declaration

The third part of a class header is the feature map construct. The feature map aspect associates new feature names with formal super-class (*public*/*restricted*) feature names. This is used to avoid feature name conflicts during the multiple inheritance process.

Feature map aspects appearing immediately within the class header associate actuals with formals of the abstract data type interface implied by the binding indication immediately enclosed. This mechanism does not allow a formal to be associated with more than one actual. The feature map construct is not required if no feature clashes occur during the multiple inheritance process. The feature mapping technique can also be used to adapt inherited features to the purpose of a newly created child class. If a feature clash occurs and no mapping has been specified, local declarations will take precedence.

The BNF of the feature map aspect is defined as follows:

```
feature_map_aspect ::=
     feature map ( feature_association_list );

feature_association_list::= feature_association_element { ,
                            feature_association_element }

feature_association_element::= formal_feature_name =>
                               actual_feature_name

feature_name::=
      feature_designator [ (parameter_list) ]
      [ return type_mark ] of class_name [ (parameter_index)
]
```

The feature designator is the identifier name for a: method, variable, constant, type, sub-type, alias, attribute, file, group or object. The `parameter_index` is used to identify a given class in the class_list if a given class is inherited more than once. This situation should occur if repeated inheritance is used. An example of feature mapping is shown in Figure 3.4.

```
type up_down_counter is class use (up_counter, down_counter)
         feature map (count of up_counter => count_up,
                      count of down_counter => count_down);
      begin
      end class up_down_counter;
```

Figure 3-4. Class Declaration with a Generic Declaration

The use of `parameter_list` and `type_mark` is only allowed for method features. This should be used when name conflicts are encountered through the use of a method overloading.

After the definition of a class header is the class declarative part. The class declarative part of a given class declaration declares items that are common to all class methods.

The BNF of the class declarative items is defined as follows:

```
class_declarative_item ::=
      [ encapsulation_kind ] subprogram_declaration
      |[ encapsulation_kind ] type_declaration
      |[ encapsulation_kind ] subtype_declaration
      |[ encapsulation_kind ] constant_declaration
      |[ encapsulation_kind ] instance_variable_declaration
      |[ encapsulation_kind ] file_declaration
      |[ encapsulation_kind ] alias_declaration
      |[ encapsulation_kind ] attributes_declaration
      |[ encapsulation_kind ] attribute_specification
      |[ encapsulation_kind ] group_template_declaration
      |[ encapsulation_kind ] group_declaration
      |[ encapsulation_kind ] object_declaration
```

As for any declarations in VHDL, the item contained in a class declarative part is local to that particular class. However, all the listed declarations have been updated to contain an extra `encapsulation_kind` property. Encapsulation kind will be examined in section 3.3.7. Sub-program declaration might seem controversial for Object-Orientation purists. However, we believe that although made redundant by the existence of methods, it will occasionally be easier for a trained VHDL designer to relate to functions and procedures, rather than methods to perform local, hidden operations (local operator overloading may be one of them).

3.3.1.2 Class body

A class body will only exist if a class is declared within a package. The package header will contain the class header and the package body will contain the class body.

When declaring a class inside a package, only method specifications and feature mapping are allowed inside the class header. The method body is implemented inside the class body located within the package body.

Class bodies have the syntactic form:

```
class_body ::=
      type class_simple_name  is class
            class_body_declarative_part
      begin
            class_assertion
            |method_body
      end  class [ class_simple_name ];

class_body_declarative_part ::=
      [ encapsulation_kind ] subprogram_declaration
      |[ encapsulation_kind ] subprogram_body
      |[ encapsulation_kind ] type_declaration
      |[ encapsulation_kind ] subtype_declaration
      |[ encapsulation_kind ] constant_declaration
      |[ encapsulation_kind ] instance_variable_declaration
      |[ encapsulation_kind ] file_declaration
      |[ encapsulation_kind ] alias_declaration
      |[ encapsulation_kind ] attributes_declaration
      |[ encapsulation_kind ] attribute_specification
      |[ encapsulation_kind ] group_template_declaration
      |[ encapsulation_kind ] group_declaration
      |[ encapsulation_kind ] object_declaration
```

The ability for the designer to split a class definition into a specification and a body is crucial for a flexible design organisation. By splitting the class into an abstract interface (specification) and an implementation part, the designer benefits from the flexibility of the package definition in terms of primary and secondary design unit dependency and analysis sequence.

The instance variables declared within the class specification are equivalent to the shared variable defined in the 1076-93 standard and are used inside the abstract data type. These variables can be declared of any VHDL types. An encapsulation property allowing derived classes or outside objects to monitor and alter their state is also added. The encapsulation property might also be used to hide information from a derived class or outside objects. The default encapsulation kind (if not specified) is *restricted*. A restricted encapsulation kind will hide the instance variable from the rest of the world, but will still permit the instance variable to be inherited in child classes.

```
encapsulation_kind ::= private | public | restricted

instance_variable_declaration ::=
      instance variable identifier_list : subtype_indication
```

Instance variables are only accessible by an external object if they are declared as public. The *private* encapsulation kind will forbid both the inheritance and external access to the instance variable. Encapsulation kind will be discussed in details in section 3.3.7

3.3.1.3 Methods

Methods in the proposed extension resemble the mechanisms found in C++ or Eiffel. Unlike ADA95 methods can be used as both expressions and statements depending on the style used for the method call. This represents in our opinion a more pragmatic approach to message passing furthermore, the use of methods instead of VHDL's sub-programs clearly states the distinction in between the two mechanisms. Each method specified within a class specification defines an abstract operation, which operates atomically on a unique object of the associated class type. The following BNF description shows the definition of a method specification.

```
method_specification ::=
            [encapsulation_kind] method designator [
            (formal_parameter_list) ] [ return type_mark ]
```

The user of a class's object will see methods as messages. These messages will be used to access or modify the state of this object. Methods will support overloading therefore, when a message is sent, the method with matching message pattern is selected from the class of the receiver. Expressions in the selected method are evaluated one after another. Once all the expressions are evaluated, a value is returned to the sender of the message.

The example in Figure 3.5 demonstrates the creation of methods within our proposed extension.

```
type reg is class
  instance variable value : bit_vector(7 downto 0);
begin
  public method reset is
  begin
    value := (others => '0');
  end method reset;
  public method write(signal clk: bit; data_in:
                      bit_vector(7 downto 0)) is
  begin
    if clk'event and clk = '1') then
      value := data_in;
    end if;
  end method write;
  method read return bit_vector is
  begin
     return value;
  end method read;
end class reg;
```

Figure 3-5. Class with Two Methods and an Execution Priority

Method formal parameters may be *constants*, *signals*, *variables*, *files* or *objects* (the default being constant). The only mode that is allowed for the formal parameters of a method is the mode *in*. In the case of files and objects, the parameters have no mode.

Two categories of methods can be identified: standard methods and virtual methods. The most straightforward category is the standard and merely defines an implementation for a given method specification. Unlike standard methods, virtual methods only define a specification, leaving the method body to be defined in child classes after the inheritance process. Figure 3.6 demonstrates the creation of a virtual method.

Method bodies are similar to procedure bodies insofar as they will accept sequential statements (including the wait statement).

that refers to the message receiver itself.

'*this*' and the formal parameters are available only during the execution of a local method. In addition, a method may obtain two different kinds of variables. These kinds of variables differ in terms of how widely they are available (their scope) and how long they persist.

Instance variables exist for the entire lifetime of the object. Temporary variables (local to methods) are created for specific activity and are available only for the duration of the activity.

Including a temporary variable declaration within the declarative part of a method indicates temporary variables. These variables are dynamically elaborated at run time. A temporary variable declaration consists of a variable name, its type and an optional default assignment. Instance variables represent the current state of an object. Temporary variables represent the transitory state necessary to carry out some activity. Temporary variables are typically associated with a single execution of a method: they are created when a message causes the method to be executed and are discarded when the method is completed and returns a value. Figure 3.6 demonstrates the creation of a virtual class.

```
type counter is class
      instance variable value: integer;
begin
      public method reset is
      begin
            value := 0;
      end method reset;

      public method read return integer is
      begin
            return value;
      end method read;

      public method count;
end class counter;
```

Figure 3-6. Virtual Class Definition

This example illustrates the creation of a virtual class with a virtual method *count* that is defined to be refined in child classes of the *counter* class.

Method names are used to identify methods. There are two kinds of feature names: identifiers and operator names. The difference between identifiers and operator names does not affect any properties of the methods, only the way that a client object may call them. For methods with identifier names, calls use the ‘dot notation’ as in: `obj1.read` or obj2.puch(`“0001”`). In contrast, calls

to methods using operator name will be written as: `obj3 + obj4`. This notation is more convenient than a more traditional one using the notation: `obj3.plus(obj4)`.

3.3.1.4 Class Definition

The definition of a class will be permitted within numerous constructs such as the package, architecture and block. Their respective declarative items are extended to include the class specification body. Type definition is extended to include object type. The BNF representation of class definition is detailed in Figure 3-7

Class definition consists of a class specification and a class body. Each class specification will be associated with exactly one class body. It is important to notice that the package declarative item list does not contain the class body declaration since this declaration is performed at the package body level.

3.3.2 Declaring Objects

Before it can be used, an object must be declared either inside or outside a class declaration part. If outside, an object of a class might be declared in the declarative region of packages, architectures or blocks.

Objects declared at the package level will be accessible throughout any design hierarchy referencing this package. When declared at the architecture level it will only be accessible within this architecture. Finally, objects declared at the block level will only be accessible by elements inside the block.

The object declarations are of the syntactic form:

```
outside_class_object_declaration::=
    object identifier_list : class_indication
   [ (formal_parameter_list) ]
   [ generic map ( generic_association_list ) ]
   [ interface map ( interface_association_list ) ];
```

When building complex objects, objects are declared inside a class declarative part. These objects obey the same encapsulation rules as normal instance variables.

The syntactic form of such a declaration is:

```
inside_class_object_declaration::=
      [ encapsulation_kind ] object identifier_list :
class_indication [ (formal_parameter_list) ]
      [ generic map ( generic_association_list ) ]
      [ interface map ( port_association_list ) ];
```

```
architecture_declarative_item::=
      common_declarative_item
   | subprogram_body
      | component_declaration
      | attribute_declaration
      | attribute_specification
      | configuration_specification
      | class_body
block_declarative_item::=
      common_declarative_item
      | subprogram_body
      | class_body
      | component_declaration
      | attribute_declaration
      | attribute_specification
      | configuration_specification
package_declarative_item::=
      common_declarative_item
package_body_declarative_item::=
      common_declarative_item
      | subprogram_body
      | class_body
common_declarative _item::=
      type_declaration
      | subtype_declaration
      | class_declaration
      | constant_declaration
      | file_declaration
      | alias_declaration
      | subprogram_declaration
      | use_clause
      | object_declaration
```

Figure 3.7. Class Definition

3.3.2.1 Creators

A successful creation always performs a default initialisation (which may then be overridden by the creation method) on the variable attribute fields of the resulting object.

A creator method will have the same name as the class, has optional parameters and can be overloaded. The creator method of an object will only

be called when the object needs to be initialised. It is possible to overload creator methods to allow a powerful and flexible initialisation of complex structures. As demonstrated in the following example, this mechanism relieves the user from an error-prone process.

In the example Figure 3-8, a creator method is defined for the *memory* class with an initial value formal parameter. When the user defines the object *mem_1*, the value to be set in the memory element at initialisation time is set to

```
architecture OO of example is
      type memory is class
            type mem_array is array (0 to 2) of integer;
            instance variable mem_array_1 : mem_array;
      begin
      public method memory(initial_value : integer);
            begin
                mem_array_1 := (others => initial_value);
            end method memory;

            public method read.....
            end method read;

            public method write.....
            end method write;
      end class memory;
      object mem_1 : memory(255); -- creation
begin
      wait for 10 ns;
      data_out <= mem_1.read(address);
end OO;
```

255.

Figure 3-8. Class Declaration with a Creator Method

Wait statements are not allowed as part of a creator method since these functions are executed during initialisation time (before run time).

Objects can be dynamically created and removed using the VHDL access type. This mechanism is identical to the one that exists for other VHDL

types. The example in Figure 3-9 demonstrates the creation of object pointers.

```
process
      type counter_access is access counter;
      variable counter_1 : counter_access;
begin
      counter_1 := new counter;
      for I in 0 to 10 loop
            counter_1.count;
            wait until clk ='1';
      end loop;
      deallocate(counter_1);
end process;
```

Figure 3-9. Dynamic Creation and Removal of an Object

3.3.2.2 Generic Map Aspect

The generic map aspect is identical to the existing generic map aspect of the VHDL standard 1076-1993. Its use has been extended to classes. A generic map aspect will be used to associate actual values with the formal generics of class.

```
architecture OO of register_16 is
      object reg_1 : reg generic map (size => 16);
begin
      A(15 downto 0) <= reg_1.read;
end OO;
```

Figure 3-10. Object Instantiation with a Generic Map

In Figure 3-10, it is assumed that a class reg with a generic *size: integer* has already been defined. An object *reg_1* is created and the generic *size* of this object of *reg* class is set to 16 at elaboration time.

3.3.2.3 Interface Map Aspect

The interface map semantics is defined as follows :

```
interface_map_aspect::=
      interface map (association_list);

association_list ::=
   association_element, { association_element }

association_element ::=
      formal_idenfier_name => actual_idenfier_name
```

An example of interface mapping is given in Figure 3-11.

```
architecture RTL of FIFO is
      signal : clk4: bit;
      object counter_1 : counter interface map (clk =>
clk4);
begin
   -- some code
end RTL;
```

Figure 3-11. Object Instantiation with an Interface Map

3.3.2.4 Composite Types

An object type can be declared as an array element. Each element can be accessed individually on an element by element basis or as a whole. *Event* and *Command* behaviours can be directly applied to the whole array by using the *.all* notation.

Figure 3-12 gives an example of declarations applied to arrays. This example assumes that a class *Counter* with a method *write* and *reset* has already been defined within a package.

Arrays are important elements in the proposed extension as late binding can take advantage of these complex data structures.

```
architecture Example_1 of Example
    type Counter_array is array (3 downto 0) of Counter;
    object Single_counter : Counter;
    object Complex_counter : Counter_array;
begin
    wait for 10 ns;
    Single_counter.reset;
    Complex_counter.ALL.reset;  -- reseting the array of
                                -- counters
    wait for 10 ns;
    Complex_counter(2).write("1010"); -- access to a single
                                      -- element
end Example_1;
```

Figure 3.12. Declaration of an Array of Objects.

3.3.3 Concurrent and Sequential Messages

Objects can be used within a sequential part in three main ways: message passing, object assignment and object comparison. For message passing, methods defined in the class specification (non-virtual) operate on objects of that class via automatic selection. The prefix denotes the object of the class. The suffix denotes the method call with optional parameters passed as part of the call. The syntactic form of a method call has the form:

```
    object_name{.object_name}.method_name[ (argument_list) ]
    object_name{->object_name}->method_name[ (argument_list)
]
```

The target and argument lists are optional; the method name is required. In the case of a call using the following form: `alu_1.multiply(3,5)`, the call uses one of the syntactical possibilities, dot notation. The target of the call is *alu_1*, the method name of the call is multiply and the actual argument list is 3 and 5.

The target is separated from the method name of the call by a period or an arrow. The *this* keyword, represents the current object executing, it is possible to write the call as: `this.store(my_value)`. However, in this case, it is also possible to write only `store(my_value)`. The *this* notation will only be found inside a method body.

A call can also be of multiple dots such as in:

```
NetCell(2).Processor(1).instruct_reg.set(jmp, 255)
```

For features without arguments, the actual argument list will be absent, as in: `acknowledge := dma.status` , where the status must either be an instance variable or a method without arguments.

```
process
      object counter_1 : upcounter;
      object counter_2 : downcounter;
      object fifo_1 : fifo;
begin
      if reset = '1' then
            fifo_1->initialise;
            counter_1->reset;
            couter_2->reset;
      elsif clk'event and clk ='1' then
            counter_1->count;
            counter_2->count;
            fifo_1->push("0001");
      end if;
end process;
```

Figure 3-13. Non Blocking Object Calls

The use of an arrow in a message infers a non-blocking call. Consequently, the call is scheduled to occur when the process is suspended. This allows the user to spawn operations on numerous objects without having to wait for the completion of an operation before moving on to the next sequential statement. If multiple non-identical blocking calls are scheduled onto one

object inside the process, the last call will prevail. Figure 3-13 shows the use of the non-blocking call.

3.3.3.1 Object Assignment and Copy

The object assignment is defined as the == sign. This refers to a clone copy of an objects reference. Examples of the object assignment can be seen in section 4.3.8.

The object copy is of two forms: as shallow and deep. The shallow copy uses the keyword *copy*. The deep copy uses the keyword *deepcopy*. Examples of are shown in Figure 3-14. The Deep Copy mechanism is used to copy all the values of the instance variables contained within an object including those originating from aggregated objects. Unlike the deep copy, the shallow copy is restricted to instance variables declared at the object level (not including aggregated instance variables).

```
Process(transfer, counter_2)
   object counter_1, counter_2 : counter;
begin
   if transfer = true then
      counter_1.copy(counter_2);
   end if;
end process;
```

Figure 3-14. Object Shallow Copy

3.3.4 Encapsulation Control and Visibility

Encapsulation is used to control the visibility of a class properties.

The BNF form for the encapsulation kind is as follows:

```
encapsulation_kind ::= private | public | restricted
```

The example in Figure 3-15 demonstrates the use of encapsulation.

```
type reg is class
      private content: bit_vector(7 downto 0);
begin
        public method reset is
            begin
            this.write("00000000");
        end method reset;

       public method read return bit_vector  is
       begin
            return content;
        end method read;

       restricted method write(data : bit_vector) is
        begin
            content := data;
        end method write;

        public method write(data : bit_vector;
                            signal clk : bit) is
        begin
            if (clk'event and clk = '1') then
                  this.write(data);
            end if;
        end method write;
end class reg;
```

Figure 3-15. Class Encapsulation Levels

This class definition will be part of a package body, block or architecture declaration. In this example, a class representing a register abstract data type is created, an attribute content is declared to implement the stored value in the register. This element is declared as private since the contents of the *reg* class should not be accessed directly but via access methods.

Methods which implement the class features, *read*, *write* and *reset*, are public (default) to be accessed by client objects. However, an asynchronous version of *write* is provided for internal use only so it is declared as restricted. The restricted declaration will permit the method to be inherited in child classes as opposed to the private one. In most cases, the private declaration will be used to specify a property that is only valid for a specific class and not for any of its descendants.

3.3.5 Inheritance

The *shift_register* class in Figure 3-16 illustrates a public inheritance. This class definition will be part of a package body, block or architecture declaration. It is assumed that the parent class *reg* has been defined at an earlier stage. This example defines a set of new methods for performing shift right and shift left operations. An error will be produced when compiling this design since the *content* attribute was declared as *private* in the *reg* class (see Figure 3-15) and both methods *shr* attempt to access it although it has not been exported during the inheritance mechanism.

```
type shift_register is class use (public reg)
begin
    public method shr(signal clk : bit) is
    begin
        if (clk'event and clk = '1') then
            this.write('0' & content(BitsNbr-1 downto 1));
        end if;
    end method shr;
    public method shl(signal clk : bit) is
    begin
    if  (clk'event and clk = '1') then
        this.write(content(BitsNbr -2  downto 0) & '0');
    end if;
   end method shl;
end class shift_register;
```

Figure 3-16. Public Encapsulation during the Inheritance Process

Multiple inheritance is achieved by listing the inherited classes in the type definition, after the class keyword.

An example of multiple inheritance is shown in Figure 3-17.

```
type shifting_counter is class use(public shift_reg,
                                   public counter) is
begin
end class shifting_counter;
```

Figure 3-17. Multiple Inheritance Example

In this example, it is assumed that a class shift_reg and counter have already been defined. The *shifting_counter* class is a new abstract data type merging the functionality of a shift register class and a counter class. Merging is often useful when inheriting virtual methods with identical notions in the descendant.

This example suggests that possible conflicts may arise due to merging during the inheritance process. Such a naming conflict can be removed by the mapping construct as shown in Figure 3-18.

```
type UpDownCounter is class use (UpCounter, DownCounter);
    generic (delay: time : 3 ns);
    feature map (Count of DownCounter => CountDown);
end class UpDownCounter;
```

Figure 3-18. Class Declaration with a Feature Map

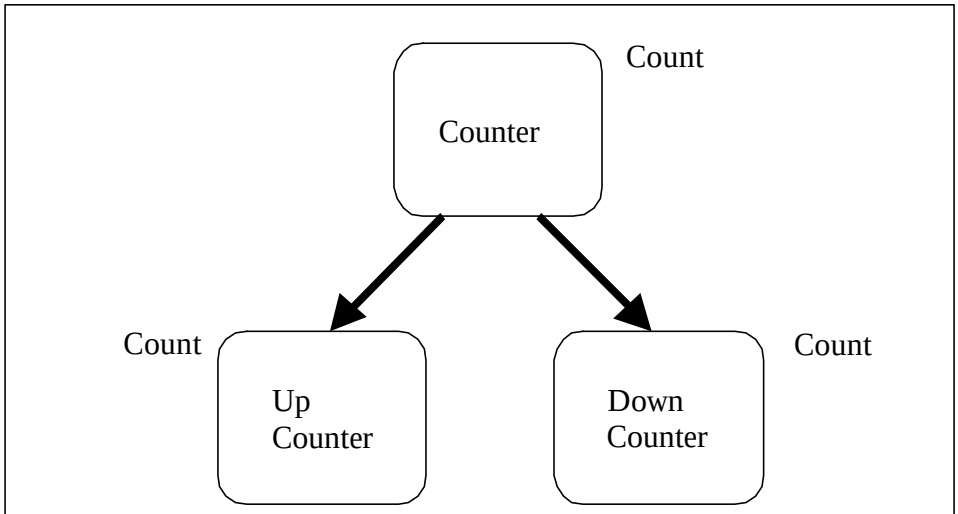
As part of the declarative part, each class can define mapping clauses to adapt names of inherited features to the local context of the new class.

3.3.6 Polymorphism

The proposed semantics fully support polymorphism. The example shown in Figure 3-19 demonstrates how polymorphism can be achieved. This example assumes that a class counter is created with a virtual method *count*. From this class *counter*, two sub-classes are derived: *up_counter*, *down_counter* in which the method *count* is defined (becomes defined). Figure 3-19 shows the definition of three objects, one for each class. On a change of the value of *A*, the *counter_1* object will change its dynamic type from *counter* to either *up_counter* or

down_counter. After the selection has been performed, the operation *counter_1.count* is carried out. This displays a polymorphic behaviour in the sense that the dynamic type of *counter_1* is only defined at run time and can change during the execution. Depending on that type, the *count* message, will either perform a count up operation or a count down operation. The '==' symbol is not a copy but a pointer to the same object.

The assignment attempt might in some cases require knowledge of the type of the source. What is needed in this case is a way to perform the assignment, but conditional on its applicability: if the type of the source object turns out not to be compatible with the type of the target object, then no reattachment should occur. The *'class* attribute provides information regarding the dynamic type of an object and can therefore be used to perform a validity check before assignments.




```
architecture polymorphic of example is
      object counter_1 : counter;
      object counter_2 : up_counter;
      object counter_3 : down_counter;
begin
      process
      begin
            if A = true then
                  counter_1 == counter_2;
            else
                  counter_1 == counter_3;
            end if;
            counter_1.count;
            wait on A;
      end process;
end polymorphic;
```

Figure 3-19. Polymorphic Behaviour

3.3.7 Visibility

The declaration part of a class has a visibility over the declaration found within the scope of the Package, Architecture and Block depending on where the class has been defined.

For the example in Figure 3-20, a class uses an enumerated type called opcode as part of its method execution and parameter list. This user-defined type, if only declared inside the class declarative part, could not be used in its method

argument list since the client object would not know about its declaration. Allowing it to be declared outside the class definition allows both the class and the design instantiating the package to use the same common type declaration.

```
package p_alu is
   type opcode is (add, sub, div, mult);
   type alu is class
      content : integer;
   begin
       method send(operation : opcode; value : integer)
              return integer;
   end class alu;
end package;

package body p_alu is
   type alu is class
   begin
       method send(operation : opcode; value : integer)
              return integer is
       begin
                        -- some statements
       end method send;
   end class alu;
end package p_alu;

use work.p_alu.all;
entity dsp is
            -- some ports declaration
end dsp;
architecture OO of dsp is
      signal command : opcode;
      object alu_1 : alu;
      signal result : integer;
begin
      result <= alu_1.send(command, 7);
end OO;
```

Figure 3-20. Class Using a Type Declared Within a Package

3.3.8 Attributes

A list of attributes allowed with objects and methods follows:

1. **'event** : Will return true when a method call occurs on an object or the referenced method.

2. **'last_event** :Will return the time since a previous method call occurred on an object or the referenced method.

3. **'active** :Will return true when an object has at least one or more methods or the referenced method currently processing.

4. **'last_active**: Will return the time since the last method processing activity occurred on an object or the referenced method.

5. **'stable [(time)]** : Will return true whenever the referenced object or method has had no call for the time specified by the optional time expression.

6. **'quiet [(time)]** : Will return true whenever the referenced object or method has not been processing any methods for the time specified by the optional time expression.

7. **'transaction**: Will create a signal of type bit that toggles its value for every method call that occurs on the referenced object or method.

8. **‘class**: This attribute will only be used by objects and will return the class type of the referenced object.

```
process
begin
    DMA_1->Start_transfer;-- DMA_1 is an object
                          -- of class DMA
    wait until DMA_1'stable(135 ns);
    DMA_1.Load_source("0001");
    wait;
end process;
```

Figure 3-21. Wait Statement Synchronised on a 'stable Attribute

Figure 3-21 illustrates the use of the *'stable* attribute. This example assumes that a class *DMA* with a method *Start_transfer* and *Load_source has* been defined earlier in the code. The attribute *'stable* is used on an object of abstract data type *DMA* and is performing a test on the non-activity of the object *DMA_1*. Once the period of inactivity has elapsed, the message *Load_source* is sent to the object DMA_1 with a given argument.

Like VHDL's signals, object activities can also be monitored as part of a sensitivity list.

3.3.9 Assertions

The improved assertion mechanism uses the following syntactic form:

```
class_assertion_statement::=
      [ contract_condition ] assert condition
            [ report expression ]
            [ severity expression ];

contract_condition::=
       pre
      |post
```

The example in Figure 3-22 shows the use of a pre assert statement.

```
type counter is class
   value : integer range 0 to 9;
begin
   pre assert (value <= 9)
   report "You have reached the maximum value"
   severity warning;

   method count is
   begin
      count := count + 1;
   end method count;
end class counter;
```

Figure 3-22. Use of Pre Assertions

For the example in Figure 3-22, a pre-assertion is used to check the status of the instance variable *value* for each access to the class. When *value* reaches 10, the assert statement issues a warning message.

3.4 Conclusions

Three main objectives are addressed in this proposal: a better encapsulation mechanism, inheritance and late binding. Although a number of different styles can be used to implement the latter mechanisms, we believe that classification orientation represents the most appropriate style in terms of its integration within both the language philosophy and the Object-Oriented paradigm. The proposed semantics illustrated in this chapter rely on a three-level encapsulation control which extends the VHDL visible/non visible mechanism and provides a flexible means of selection for the inheritance process.

Inheritance will here be selective, single or multiple and enhanced by a powerful redefinition construct called the feature map in addition to genericity. Furthermore, creator methods are added to classes allowing a straightforward initialisation sequence for instances of classes.

The proposed extension redefines the assignment rules to allow more flexibility in the type system, i.e. objects of a derived class type can be assigned to objects of an ancestor class type thereby allowing dynamic binding and therefore polymorphism to take place.

This chapter defined the language constructs created to tackle the three main requirements set in section 2.4 namely: encapsulation, inheritance and polymorphism. The following chapters will highlight the flexibility of these constructs to build complex hardware descriptions and will compare the proposed extension with other proposed Object-Oriented extensions to VHDL.

Chapter 4

COMPARATIVE STUDY OF PROPOSED EXTENSIONS

4.1 Introduction

Having introduced the proposed extensions (see section 2.6 and chapter 4) we will now compare the extensions according to the language design objectives set by the IEEE OOVHDL study group (detailed section 2.4). The extensions that will be examined are the VISTA, the Oldenburg, the SUAVE the Objective VHDL and our proposal (Cabanis 1995).

The VISTA proposal (Ramesh 1994.2) is the earliest extension to the language and to some extents the less sophisticated too. Although this proposal has been abandoned, we will briefly study the mechanisms used for implementing data abstraction, inheritance, and polymorphism.

The SUAVE proposal is in effect a superset of the Oldenburg (Shumacher 1995) proposal. Consequently, in an effort to reduce the amount of redundancy in this comparative study, we will focus on the SUAVE extension and mention the areas where the two proposals differ from one another.

4.2 Study of the VISTA Proposal

4.2.1 Abstract Data Type

The Vista proposal is based on a component like abstract data type: the *EntityObject*. This represents a sensible and straightforward implementation of an abstract data type. This approach is similar in its principle to other existing object-oriented programming languages such as C++ or Eiffel (Meyer 1992).

```
EntityObject <identifier> is [ new <identifier> ]
            [ generic <generic_list> ]
            [ port <port_list> ]
            { operation <operation_name> <interface_list> }
            [ begin ]
                  [ entity_statement_part ]
end EntityObject [ <identifier> ]
```

Figure 4-1. The EntityObject Abstract Data Type

The *EntityObject* represents a real abstract data type, in the sense that attributes and related operations are gathered in one design entity. The splitting of the abstract interface and its associated implementation is an important feature for achieving design maintainability. The Vista proposal achieves this here by the separation of the *EntityObject* and the *Architecture*. However, we believe that the Vista extension is restricted in its scope of use. The creation of ADTs based on a type instead of a design unit is more suited to dynamic typing and encapsulation control. Furthermore, the creation of a new design unit makes design organisation and management far more complex. The VHDL language (DASC 1993) consists of five design units comprising three primary and two secondary units. The concept of design units represents a powerful yet complex means of structuring a design. The use of design units requires significant efforts in design management; they are still not fully understood by most experienced designers, nor supported by a wide number of synthesis tools. HDLs like the Verilog language (Verilog std 1364-1995) owe their popularity to their simplified use and also partly to the absence of design units. Furthermore, while the VHDL'93 version of the language allows designers to reduce the number of design units via the direct instantiation mechanism, the Vista proposal introduces one more design unit, bringing the total to six. Although we do not contest the importance of design units, we believe that the addition of more design units will increase the design organisation complexity for no extra gain. The splitting of the abstract data type into an interface is necessary, but the use of multiple architectures

conflicts with the Object-Oriented inheritance principle. Different versions of a design should not have a common *EntityObject* but rather a common generalised parent class in order to allow an extended version of the language to better map to Object-Oriented design methodologies.

In addition to declarations of operations, the *EntityObject* can hold the declaration of ports and generics. Port declarations are used to create global signals throughout the abstract data type. These signals are generally reserved for synchronisation. This feature is relevant since it simplifies the modelling of synchronous devices; for example the clock signal of a synchronous design does not have to be part of the sensitivity list in a call but merely declared as a common port for the given abstract data type.

4.2.2 Inheritance with the EntityObject.

The Vista Object-Oriented extension supports a public single inheritance mechanism. Both instance variable and operations are exported to the inheriting *EntityObject*.

Neither a clear inheritance control mechanism or redefinition mechanisms have been implemented. With the Vista proposal, redefinition is merely achieved through a new declaration and definition of a given operation. Furthermore, with the Vista proposal, an operation parameter list can only contain VHDL standard types. This limitation, which is not an issue in all applications, comes from the use of *ComponentObjects* as abstract data type instances instead of type representations.

In addition to this limitation, the use of a *super* mechanism to access previous versions of redefined operations does not exploit the full capabilities of the inheritance hierarchy. This highlights two issues: no support for multiple inheritance and loss of previous versions of redefined operations. To illustrate the latter issue, one can imagine a situation where a given operation has been defined at three levels of the inheritance hierarchy: in such a case, the lowest

level will not be able to access the first definition but only the definition of its direct parents.

4.2.3 Polymorphism and Object Handles

EntityObject handles represent a convenient means of handling dynamic types. An *EntityObject* handle as defined in the Vista proposal can take any *EntityObject* type at run-time, allowing polymorphic operations to be implemented.

However, the scheme proposed in the Vista extension, undermines and to some extent jeopardises, the VHDL type system. Maintaining the consistency of the VHDL type system is a crucial aspect of the building of an Object-Oriented extension to the language. The *EntityObject* handle does not respect any sub-typing relationship; thus it is possible to assign objects of any type (abstract type) to a handle at run time. For instance, an array of *EntityObject* handles could contain a shift register, a counter and an ALU object at the same time. Although the counter and the shift register may have a common root (such as a register), the ALU has little in common with the other devices in the *EntityObject* handles array. Allowing such a grouping of abstract data types could lead to a situation where the array's elements contained inconsistent operations. This is only one example of the issues related to the phenomenon of 'direct object reattachment'. In general, it is conceptually wrong to allow the grouping of non-related types in an array. This can be illustrated by, considering the example in Figure 4-2. In this example, the first two elements of the *EntityObject* handle array are assigned to a basic counter and a bounded counter.

```
variable Counter_list is array (0 to 2) of EO_Handle ;
begin
      counter_list(0) := Basic_counter ;
      counter_list(1) := Bounded_counter ;
      send Counter_list(0) Reset ;   -- line 1
      send Counter_list(1) Reset ;   -- line 2
      send Counter_list(0) load_last (15); -- line 3
      send Counter_list(1) load_last (15); -- line 4
```

Figure 4-2. Grouping of Un-related Types in an Array.

On line 1 and line 2, *Reset* operations are successfully sent to both counter elements. Line 3 performs a *load_last* operation on the element zero of the *EntityObject* Handle: this is an illegal operation since the element zero is of *basic_counter* type and does not implement a *load_last* operation. Unlike line 3, line 4 successfully performs the *load_last* operation since, this time, the element one of the *EntityObject* handle array is of type *bounded_counter*. Although the expected behaviour is achieved, the drawback of this method is that the operation *load_last* is allowed at compile time and the actual error would not be reported until run time (if ever). Allowing each object of the *EntityObject* handle array to run specific commands allows bad coding styles and compromises the VHDL type system. In a well-planned Object-Oriented design, these cases should only arise when dealing with complex problems. In most cases, the rationale for using a generic structure (array) is to abstract the details of specific variants, and concentrate on what is common to all. If it is known that an element of such a structure has special properties, then it is possible to access it separately, through an object of the right type. Ideally, both line 3 and line 4 (in Figure 4-2) should be disallowed, assuming that the common set of array elements is made of basic counters. This limitation of the *EntityObject* handle results from the limitations of components being used as abstract data types in the proposed Vista extension.

4.2.4 Vista Study Conclusions

The Vista proposal implements all the required objectives defined by the OOVHDL study group. However, because of its choice of ADT implementation (component object), this proposal offers very limited scope for abstraction. Furthermore the creation of an extra VHDL design unit makes design organisation and management a more complex task. The use of object containers for implementing dynamic type is valid; however since no assignments rules are defined for the *ObjectHandle*, this will lead to some design inconsistencies. This extension was a good first attempt for the implementation of an object-oriented extension to the VHDL. It showed the limits of a component-like ADT and allowed the research community to focus on composite type based (record like) ADT alternatives.

4.3 Study of the Active Proposals

This analysis of competing proposal is conducted using a number of comparison points derived from the OOVHDL study group requirements list. These are: Class interface definition, method definition, abstract classes, encapsulation and visibility control, inheritance, instantiation, initialisation, assignment and copying, aggregation, method invocation, polymorphism and genericity.

4.3.1 Class Interface Definition

The SUAVE (Ashenden 1997.2)/Oldenburg proposal uses the VHDL language feature namely: The package. Furthermore the functionality of the package is modified to suit the requirements of a flexible abstract data type. The package in the SUAVE extension no longer needs to be a design unit and can be declared in other declarative parts. An example of a class describing a counter is given in Figure 4-3:

```
package counter is
   -- Declarations
end package counter;
```

Figure 4-3. ADT Declaration in SUAVE/Oldenburg.

Packages in the SUAVE proposal are no longer design units when they are used as ADTs. As a result, one can question the need for using the existing package since, such a language decision will create an overlap with the existing VHDL semantics as well as inconstancies. We do not believe that any benefits are grained from selecting existing language constructs if they are not adapted to the implementation of an orthogonal extension to the VHDL.

The Objective VHDL (Radetziki 1997.2) proposal relies on the creation of a new kind of composite type: The class. A class type is parented to the VHDL record type and contains both fields (instance variables) and their associated subprograms. The example in Figure 4-4 describes a counter using both the Objective VHDL and our proposal:

```
type counter is class
   -- Declarations
end class counter;
```

Figure 4-4. ADT Declaration in Objective VHDL and our Proposal.

All proposals provide similar mechanisms for data encapsulation and method implementation. The SUAVE and Oldenburg proposals are alike and the Objective VHDL class is identical to our proposal.

4.3.2 Method Definition

While SUAVE/Oldenburg as well as Objective VHDL use the VHDL subprograms, their formal parameter list differs. For the SUAVE/Oldenburg proposal, any method requires at least one formal parameter that carries the

object (or data structure) on which the operation is to be performed. The example in Figure 4-5 demonstrates the declaration of two methods:

```
package counter is
type value_t is tagged record
   value: integer;
end record;
function read(val : value_t) return integer;
procedure write(val: inout value_t;
                signal clk: in std_logic;
                value_in: in integer);
end package counter;
```

Figure 4-5. Method Declarations in SUAVE/Oldenburg.

The objective VHDL gives the methods, implicit access to the attributes of the class that declares them. As a result the formal parameter list only contains useful input/output arguments. This style of method declaration is the most commonly implemented in other object-oriented programming languages. An example of method declaration in Objective VHDL is given in

```
type counter is class
   class attribute  value : integer;
   function read return integer;
   procedure write(signal clk: in std_logic;
                   value_in: in integer);
end class counter;
```

Figure 4-6:

Figure 4-6. Method Declaration in Objective VHDL.

Our proposed extension does not rely on subprogram calls for implementing method calls. Although this requires the addition of the new keyword ‘method’, the motivations are justified. In VHDL and numerous other programming languages there are two kinds of subprograms: functions and procedures. While both subprograms share a large amount of common features, their small differences justify their unique identity. The main

differences between functions and procedures are execution time, number of output parameters and their usage as part of an expression. The procedure permits the suspension of processes (to perform signal update) whilst executing. In other words, one can use the VHDL *wait* statement inside a procedure body. The execution of a function is considered instantaneous; therefore no waiting is permitted inside the function body. Furthermore the function is not allowed to call internally a procedure contains a wait statement. Similarly, the proposed 'method' construct gathers some of the singular properties of both VHDL subprograms and rejects others. The method accepts any number of input parameters of signal constant and file class; however only one return value is allowed. This resembles the function call and was implemented to allow method calls to be used as part of expressions. Unlike Objective VHDL or SUAVE, there are no distinctions made on which type of subprogram to use (function or procedure): depending on the application (statement or expression) the method constructor is used independently. This method style simplifies the creation of classes and broadens the class' applications. Furthermore this relates directly to most object oriented programming languages. A 'method' in the proposed extension allows the use of *wait* statements. This feature is akin to the VHDL procedure and allows expressions/statements to be executed in no time or after a user definable delay. Consequently, the functionality required for the creation of a well-suited method implementation justifies the building of a new separate language mechanism. The solutions offered by the SUAVE and Objective VHDL represent a compromise that promotes ambiguities with existing and different VHDL constructs.

An example of a counter class with the proposed extension is given in Figure 4-7:

```
type counter is class
   instance variable  value : integer;
   method read return integer;
   method write(signal clk: std_logic;
                value_in: integer);
end class counter;
```

Figure 4-7. Method Declaration in our Proposal.

4.3.3 Non Instantiable (abstract) Classes

All proposals support the creation of abstract classes. Whereas SUAVE requires an explicit declaration using the keyword 'abstract' (not specified in the Oldenburg proposal), both our proposed extension and the Objective VHDL one use an implicit definition. When a class is defined with methods that have not yet been defined, the class becomes an abstract class. An example of an abstract class declaration in SUAVE is given in Figure 4-8:

```
package packet_t is
   type abstract new data_t with record
      parity: boolean;
   end record value_t;
   procedure process_packet(value: inout data_t);
end package packet_t;
```

Figure 4-8. Abstract Class Declaration in SUAVE .

The benefit of using an extra keyword to qualify if a class is 'abstract' or not is minor. With SUAVE, the user may get an error earlier in the compilation sequence since the compiler will check if all methods are defined in a non-'abstract' class. However, with the Oldenburg, Objective VHDL and our proposed extension, the error issued when the user attempts to instantiate objects of an abstract class type. Similar behaviour is found in standard VHDL when using a non-fully defined 'deferred' constant. An example of an abstract class declaration in Objective VHDL is given in Figure 4-9:

```
type packet is class
   class attribute parity: integer;
   procedure process_packet;
end class packet;
```

Figure 4-9. Abstract Class Declaration in Objective VHDL.

The equivalent code using our proposed extension is given in Figure 4-10:

```
type packet is class
   instance variable parity: integer;
   method process_packet;
end class packet;
```

Figure 4-10. Abstract Class Declaration in our Proposal.

4.3.4 Encapsulation Control and Visibility Control

Encapsulation control is implemented at different levels in the three proposals. The SUAVE/Oldenburg version relies on two separate mechanisms: the package body and the definition of a private area in the package header (not in the Oldenburg proposal). The package body is used to hide sub-programs from the class user. The private part of a package header is used to hide the implementation of a type. Types and sub-programs declared outside the private part of a package header are visible by the class user without any restrictions. The declaration of a class containing both visible and non-visible declarations is given in Figure 4-11:

```
package fifo is
   type fifo_t is tagged record
      value : std_logic_vector(7 downto 0);
      is_full : boolean;
      is_empty : boolean;
   end record fifo_t;
   procedure push(val: inout fifo_t;
                  signal clk: in std_logic;
                  value_in: in std_logic);
end package fifo;
package body fifo is
   procedure push(val: inout fifo_t;
                  signal clk: in std_logic;
                  value_in: in std_logic) is
      -- some code
   end procedure push;
   procedure async_write(val: inout fifo_t) is
      -- some code
   end procedure async_write;
end package fifo;
```

Figure 4-11. Visible an Non-Visible Declaration in SUAVE.

In this example, the type *fifo_t* and the procedure push are visible, whereas the procedure *async_write* isn't.

The semantics in Figure 4-12 would hide the type *fifo_t*:

```
package fifo is
   type fifo_t is tagged private;
   procedure push(val: inout fifo_t;
                  signal clk: in std_logic;
                  value_in: in std_logic);
   private
   type fifo_t is tagged record
      value : std_logic_vector(7 downto 0);
      is_full : boolean;
      is_empty : boolean;
   end record fifo_t;
end package fifo;
```

Figure 4-12. Hidden Type Declaration in SUAVE.

For the example in Figure 4-12, the type *fifo_t* is declared as a private type and defined in the private part of the package header.

The Objective VHDL proposal implements encapsulation via: an encapsulation rule and the use of a class body. The encapsulation rule in Objective VHDL defines that none of the 'class attributes' are visible outside the class that defines them. Any access to the 'class attributes' has to be made through the use of sub-programs. However, unlike the SUAVE/Oldenburg proposals, the designer can specify if a given 'class attribute' is to be visible inside child classes or not. The visibility of a 'class attribute' after inheritance is achieved by declaring the 'class attribute' inside the header part of a class. The declaration of a 'class attribute' inside the body part of a class will cause the 'class attribute' to be lost during the inheritance process. Similarly, a sub-program in Objective VHDL is visible if declared inside the class header. Alternatively, a sub-program can be hidden if declared in a class body. This mechanism is identical to the existing package header / package body in standard VHDL. Figure 4-13 demonstrates the declaration of visible and non-visible sub-programs and class attributes.

```
type fifo is class
   class attribute value: std_logic_vector(7 downto 0);
   class attribute is_empty: boolean;
   class attribute is_full : boolean;
   procedure push(signal clk: in std_logic;
                  value_in: in std_logic);
end class fifo;

type fifo is class body
   procedure push(signal clk: in std_logic;
                  value_in: in std_logic) is
      -- some code
   end procedure push;
   procedure async_write(value_in: in std_logic);
end class body;
```

Figure 4-13. Visible and Non Visible Declarations in Objective VHDL.

For the example in Figure 4-13, the sub-program push is visible and the sub-program *async_write* is hidden. None of the 'class attributes' are visible outside the class *fifo*; however, since they have been declared inside the class header, they will be inherited by all child classes. The code in Figure 4-14 illustrates how to exclude the 'class attributes' *is_empty* and *is_full* during derivation.

```
type fifo is class
   class attribute value: std_logic_vector(7 downto 0);
   procedure push(signal clk: in std_logic;
                  value_in: in std_logic);
end class fifo;

type fifo is class body
   class attribute is_empty: boolean;
   class attribute is_full : boolean;
   procedure push(signal clk: in std_logic;
                  value_in: in std_logic) is
      -- some code
   end procedure push;
   procedure async_write(value_in: in std_logic);
end class body;
```

Figure 4-14. Hidden Attribute Declarations in Objective VHDL.

Both 'class attributes' have been moved to the class body part hence they will not be inherited by any of the child classes.

The proposed extension implements a thorough encapsulation mechanism for ‘instance variables’, method access and inheritance control. Encapsulation uses three identifiers ‘public’, ‘private’ and ‘restricted’. As with Objective VHDL and SUAVE (Oldenburg relies only on VHDL’s package visibility mechanisms), ‘instance variables’ can be hidden from the class user by using the keyword ‘private’ in front of the variables’ definition. Nevertheless, unlike Objective VHDL any ‘instance variables’ can be made visible to the class user by using the keyword ‘public’ instead of ‘private’. Furthermore the proposed extension allows each individual ‘instance variable’ of the class to have a different encapsulation control; this cannot be done with the SUAVE/Oldenburg proposal. In SUAVE, all or none of the fields of the record type are set to be visible or hidden. The example in Figure 4-15 illustrates the declaration of ‘instance variables’ and methods with different encapsulation controls:

```
type fifo is class
   public instance variable is_full : boolean;
   public instance variable is_empty : boolean;
   restricted instance variable value :
                          std_logic_vector(7 downto 0);
   public method push(signal clk : std_logic;
                      value_in :  std_logic) is
      -- some code
   end method push;
   restricted method async_write(value_in : std_logic) is
      --some code
   end method async_write;
end class fifo;
```

Figure 4-15. Encapsulation Control in our Proposal.

Figure 4-15 demonstrates the flexibility of the proposed extension over competing proposals. As shown in this example, both variables *is_full* and *is_empty* have been declared as *public* hence visible outside the class that defines them; whereas the variable *value* is hidden inside the class *fifo*. We believe that this flexibility, only present in our proposal, is essential for the

construction of a versatile ADT. Figure 4-16 demonstrates the use of a public instance variable:

```
architecture rtl of data_line is
   object fifo_1 : fifo;
begin
   process
   begin
      wait until clk = '1';
      if fifo_1.is_empty then
         fifo_1.push(clk, in1);
 -- more code
```

Figure 4-16. Use of Public Instance Variables in our Proposal.

In Figure 4-16, the state (*is_full*) of the *fifo_1* object is accessed through a 'method-like' message. This style presents two main benefits: consistency and increased maintainability level. Since the access to a public instance variable is semantically identical to a method call, object access remains consistent. Furthermore, if the user decides at a later stage to implement the query (*is_full*, *is_empty*) using methods instead of 'instance variables', the supporting code used to access an instance of the *fifo* class is not required to be updated since the calls will be identical.

The use of three distinct encapsulation kinds in the proposed extension has added benefits during the inheritance process. By specifying if an 'instance *private*, *public* or *restricted*, the class designer has full control over what should and what should not be present in a derived child class. This inheritance control is similar to what is found in the Objective VHDL extension. However, the distinction between a *public* type and a *restricted* type in the proposed extension offers further levels of flexibility. This distinction will be detailed during the study of inheritance across proposed extensions.

In addition to the limits encountered in the definition of encapsulation in SUAVE/Oldenburg and Objective VHDL, these proposals rely on a confusing and over complicated data hiding mechanism. The encapsulation mechanism found relies on a standard VHDL package-body 'like' hiding mechanism. The standard VHDL language defines that declarations made at the package-body level are only visible within the package that declares them. This definition has an important role in the context of the separation of VHDL's primary and secondary design units and design units' compilation rules. Pragmatically this rule allows a designer to make changes to the content of a package-body, without having to recompile any other design units that might be referencing the altered package. The VHDL language permits this since the content of the package body is hidden from the rest of the design. Although, this is a complex encapsulation mechanism, it is justified by the existence of two separate design units (primary and secondary). However, the complexity involved in the use of a similar mechanism in SUAVE and Objective VHDL is not justified. In SUAVE, the new package ADT is no longer a design unit hence will not be compiled on its own if the package has been declared as part of an architecture declarative part. Consequently the need for a distinct package-body for implementing parts of an ADT is over complex and irrelevant. It can be assumed that this package-header/ package-body split has been kept in SUAVE to resemble the original standard VHDL package construct at the expense of conciseness. Furthermore the SUAVE extension adds extra overlap in the VHDL language by declaring a private part of a package-header. From these observations we conclude that the use of the package construct and encapsulation mechanism does not represent a suitable implementation of an ADT.

Unlike the SUAVE/Oldenburg extensions, Objective VHDL relies on a new ADT construct. However, the encapsulation mechanism is still similar to the SUAVE proposal with the distinction between a header and a body part of a non-design unit. For the same reasons developed for the SUAVE proposal

we do not believe that this represent a useful complication of the VHDL language.

4.3.5 Inheritance

Single inheritance and method overloading are provided in all the proposed extensions; however, multiple inheritance is only present in our proposed extension. The discussions over the benefits of single versus multiple inheritance are beyond the scope of this study. Nevertheless, we believe that the use of multiple inheritance in conjunction with flexible mapping mechanisms, offers designers a higher level of reuse than the one achieved through single inheritance. To highlight the benefits of using multiple inheritance, we will use a simple case study. The study assumes the existence of the class structure in Figure 4-17:

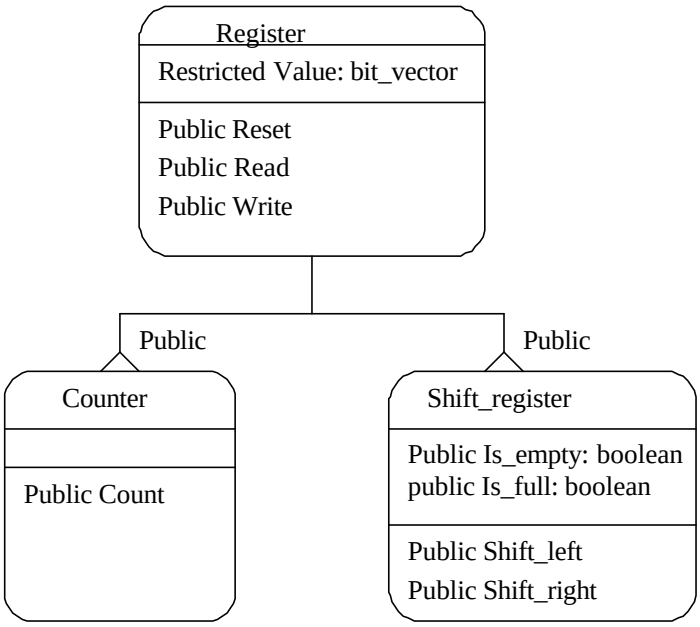


Figure 4-17. Example Class Structure.

The case study also assumes that the designer is required to create a new class called *Add_shift*. The Add_shift class is an aggregation of both, a counter and a shift register functionalities and should contain the methods and instance variables shown in Figure 4-18:

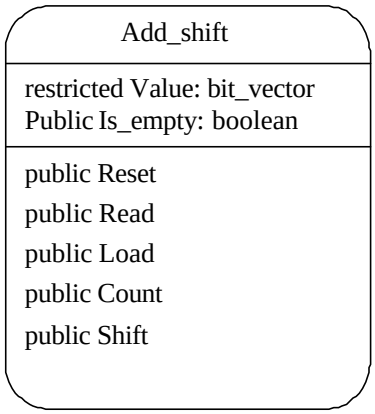


Figure 4-18. Example Add_shift Class.

Using the proposed extension, the creation of such a class requires the inheritance of the two existing classes *counter* and *shift_register*. However, a number of elements of the inherited classes are not needed by the *Add_shift* class; furthermore, some of the inherited methods require different names. As a result, the undesired elements should be hidden from the user and the methods with different identifiers should be renamed. The code in Figure 4-19 illustrates the implementation of the *Add_shift* class using our proposed extension:

```
type add_shift is class use (public COUNTER,
                             restricted shift_register) is
   feature map (WRITE => LOAD,
                is_empty => public is_empty;
                shift_left => public shift);
begin
end class add_shift;
```

Figure 4-19. Feature Mapping in our Proposal.

In Figure 4-19, *add_shift* is created from a public inheritance of the *counter* class and a restricted inheritance of the *shift_register* class. Since both the *counter* and *shift_register* class come from the same origin (*register* class), all common features are collapsed into single elements. The merging of common features insures the availability of only one version of: *value*, *read*, *write* and *reset* inside the *add_shift* class. Furthermore, the language extension defines that the encapsulation kind with the highest visibility (*public*) is chosen for collapsed features. Thus, the collapsed methods *read*, *write* and *reset* will be publicly visible due to the inheritance encapsulation kind of the *counter* class.

The inheritance encapsulation kind for the two parent classes, was chosen according the required features in the *add_shift* class. A public inheritance of the *counter* class insures the visibility and inheritance of all the *public* and *restricted* features of that class. A restricted inheritance of the *shift_register* class, sets the default encapsulation kind of all public features of that class to be restricted in the derived *add_shift* class. This allows the features *is_full*, *is_empty*, *shift_right* and *shift_left* to be hidden in the *add_shift* class.

The feature map construct in this example, is used for two purposes: name mapping and encapsulation kind overwrite. The name mapping is used to rename both *write* and *shift*_left into *load* and *shift* respectively. In addition, the keyword *public* is used in front of *shift* and *is_empty* in order to overwrite the

default encapsulation kind, set by the *restricted* inheritance kind of the *shift_register* class.

The implementation of the same *add_s*hift class with the SUAVE / Oldenburg and Objective VHDL extensions would present a number of problems. Due to their lack of support for multiple inheritance, the designer would have to select a unique parent and add manually the missing code required for the full implementation of the *add_shift* class. This will lead to code redundancy hence making the design more error prone and less maintainable. Furthermore the SUAVE / Oldenburg and Objective VHDL extensions support a limited encapsulation mechanism which does not allow them to adapt inherited features to the needs of the child class. In contrast, the proposed extension implements a feature mapping mechanism. This makes the ability to hide, make visible, or rename features before and after inheritance a simple and flexible process. The example in Figure 4-20 illustrates an implementation of the *add_shift* class using the SUAVE / Oldenburg proposals.

```
package add_shift is
   type add_shift_t is new counter_t with record
      is_empty: boolean;
   end record;
   procedure load(value : inout add_shift_t;
                  value_in : in bit_vector);
   procedure shift(value : inout add_shift_t;
                   value_in : in bit_vector);
   procedure empty(value : inout add_shift_t;
                   value_out : out boolean);
end package add_shift;

package body add_shift is
   procedure load(value : inout add_shift_t;
                  value_in : in bit_vector) is
      -- some code
   end procedure load;
   procedure shift(value : inout add_shift_t;
                   value_in : in bit_vector) is
      -- some code
   end procedure shift;
   procedure empty(value : inout add_shift_t;
                   value_out : out boolean) is
      -- some code
   end procedure empty;
end package add_shift;
```

Figure 4-20. Inheritance Limits in SUAVE/Oldenburg.

Some of the limits of the SUAVE/Oldenburg extensions inheritance/encapsulation mechanisms are demonstrated in Figure 4-20. Since the *shift_register* class could not be inherited at the same time as the *counter* class, the shift procedure had to be re-implemented in the *add_shift* class. The load procedure has been created to conform to the specifications. However, with this implementation, both the newly defined *load* procedure and the inherited *write* procedure will exist inside the *add_shift* class and have the same behaviour. Another limitation found in the SUAVE/Oldenburg extension is the non-ability to mix encapsulation kinds within the class's data structure. For this reason we had to create a procedure *empty* to access to the new *private* instance variable *is_empty*.

The example in Figure 4-21 demonstrates similar limitations found with the Objective VHDL proposal.

```
type add_shift is new class counter with
   class attribute is_empty: boolean;
   procedure load(value_in : bit_vector);
   procedure shift(value_in : bit_vector);
   function empty return boolean;
end class add_shift;
type  add_shift is class body
   procedure load(value_in : bit_vector) is
      -- some code
   end procedure load;
   procedure shift(value_in : bit_vector) is
      -- some code
   end procedure shift;
   function empty return boolean is
      -- some code
   end function empty;
end class body add_shift;
```

Figure 4-21. Inheritance Limits in Objective VHDL.

The Objective VHDL implementation has similar limitations to the SUAVE/Oldenburg extensions. The class attribute *is_empty* and the procedure *shift* were created due to the lack of multiple inheritance support. The *load* procedure was created to conform to the class's specifications: however, since feature mapping is not allowed in Objective VHDL, the *write* procedure inherited from the *counter* class becomes redundant in the add_*shift* class. As with the SUAVE/Oldenburg example, the function *empty* had to be built to allow access to the *is_empty* class attribute.

Along with multiple inheritance, our proposed extension supports repeated inheritance. Repeated inheritance occurs whenever (as a result of multiple inheritance), two or more ancestors of a class have a common parent. This class is often called a repeated descendant. Figure 4-22 (Left) shows a repeated inheritance for an *hand-bell counter*.

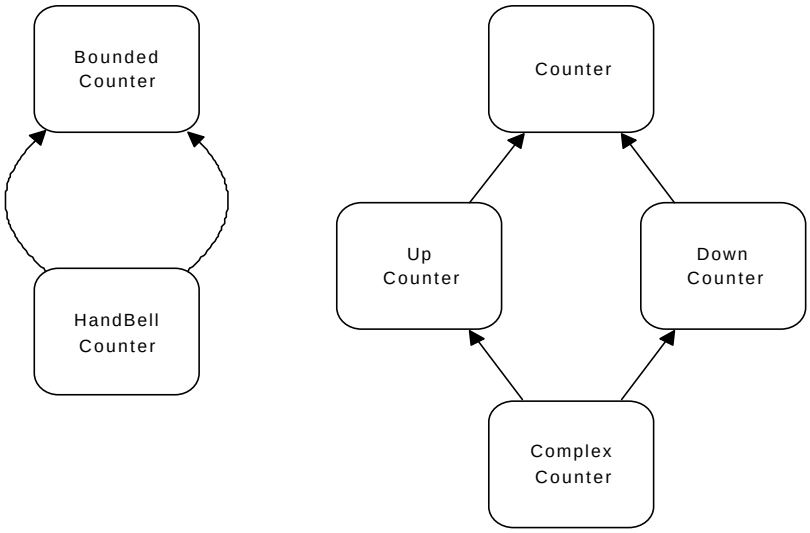


Figure 4.22. Repeated Inheritance for an Hand-bell Counter.

This simple form is called repeated inheritance and corresponds to the case when *the hand-bell counter* is a repeated child of the *bounded counter*. Another form of repeated inheritance is shown in Figure 4-22 (Right) in which the parent *UpCounter* of *ComplexCounter* is a proper descendant of the *Counter* class and one or more other parents (such as the *DownCounter* are descendants of the Counter class. With repeated inheritance, in our extension, if no feature mapping is performed, the multiple inherited features are merged in a single one.

Repeated inheritance is not supported by any of the other proposed extensions.

4.3.6 Instantiation

The instantiation of a class's object is a requirement to the implementation of an object-oriented extension to the VHDL programming language. However, the realisation of object's instantiation varies significantly between both the SUAVE/Oldenburg and Objective VHDL proposals, when compared to our extension. With the SUAVE/Oldenburg and Objective VHDL, signals, variables and constants are used for the creation of a class instance (object) whereas, our proposed extension, promotes the creation of a new VHDL building element, the 'object'. The examples in Figure 4-23 and Figure 4-24 demonstrate the instantiation of objects in the three proposals:

```
architecture rtl of edge is
   signal count1 : counter;
begin
   process
      variable fifo1 : fifo;
   begin
      -- some code
   end process;
end rtl;
```

Figure 4-23. Instantiation in SUAVE/Oldenburg & Objective VHDL.

```
architecture rtl of edge is
   object count1 : counter;
begin
   process
      object fifo1 : fifo;
   begin
      -- some code
   end process;
end rtl;
```

Figure 4-24. Object Instantiation in our Proposal.

There are numerous considerations in the choice of a mechanism for implementing classes instantiations (objects). The selection of one

implementation over another depends on the means defined within the language extension for accessing an object.

Pragmatically, objects can be considered as resources that can be queried or altered by a client. This is implemented by most programming language via two kind of resources: constants and variables. Both kinds carry mechanisms used to read and assign values; specifically, constant can only be read and variable can be read as well as altered without any notion of time delay.

Nevertheless, due to its concurrent and sequential nature, the VHDL language has to feature a new kind of assignment mechanism, which allows the description of deterministic concurrent assignments. Consequently, the signal kind was created to implement a non-blocking assignment mechanism.

The existence of the blocking and non-blocking assignment mechanisms inside the VHDL language is a determining factor in the creation of a class instance (object). The proposed instantiation mechanisms in both SUAVE/Oldenburg and Objective VHDL is identical. Objects in those two proposals can be accessed in a blocking or non-blocking fashion depending on the kind of declaration used (signal, variable or constant). The figure 4.23 illustrates the declaration of an object using a signal; therefore this implies that the ‘counter1’ object can only be assigned using a non-blocking assignment style. Similarly, if the designer needs to access to an object using a blocking assignment style, the declaration of this object should be of the variable kind. This implementation of object declaration represents a compromise between minimal language change and a flexible object-oriented extension to VHDL.

From the two existing kinds of assignment in the VHDL language, we argue that only the blocking kind is useful for the implementation of class instance. For that reason our proposed extension only features one kind of element akin to the VHDL’93 shared variable.

Although the use of signals for implementing object updates through a non-blocking mechanism is a sensible approach, we do not believe that the complexity of the non-blocking mechanism brings any significant benefits over the use of blocking assignments in the context of an object oriented extension.

In a non object oriented version of the VHDL language, non-blocking assignments are used for conveying information between concurrently interacting processes or entities (ports) in a deterministic way. This mechanism was introduced to VHDL in order to overcome the phenomenon of HDL race found in languages such as Verilog. A HDL race condition occurs on a shared resource when two distinct values are assigned (blocking assignment) at the same simulation time (eg.25 ns) but on different delta cycles. The end result is non-deterministic since the order in which the two assignments are performed can not be established. Unlike the blocking assignment, non-blocking assignments will always guarantee a deterministic value on a common resource according to the output of a resolution function. An example of the behaviour of blocking assignments is shown in Figure 4-25:

```
data <= data1 when en1 = '1' else 'Z';
data <= data2 when en2 = '1' else 'Z';
```

Figure 4-25. Multiple Drivers Assignment in VHDL.

In this example, the common resource ‘data’ is of a resolved type such as std_logic. Nevertheless, aside from this application, hardware resources are never accessed (written to) by multiple concurrent processes. This rule is dictated by safe design practice as well as synthesis tools restrictions. Figure 4-26 and Figure 4-27 demonstrate improper and proper VHDL descriptions for a counter.

```
process(reset)
begin
   if reset = ‘1’ then
      count <= 0;
   end if;
end process;

process(clk)
begin
   if clk = ‘1’ then
      count <= count + 1;
   end if;
end process;
```

Figure 4-26. Improper Description of a Counter.

```
process(clk, reset)
begin
   if reset = ‘1’ then
      count <= 0;
   elsif clk’event and clk = ‘1’ then
      count <= count +1;
   end if;
end process;
```

Figure 4-27. Proper Description of a Counter.

Although syntactically correct, the solution shown in Figure 4-26 cannot be exploited as, the style used in this example Figure 4-26 will be regarded as non-deterministic and will not be accepted by synthesis tools. This is due to the two processes are concurrently accessing a common resource *count*.

Since coding styles based on concurrent non-blocking assignments are forbidden in a VHDL design flow, we argue that objects should not be accessed (written too) by multiple concurrent processes in order to avoid non-deterministic behaviours.

Unlike non-blocking assignments, the creation of a class instance through a variable-like (blocking) declaration is supported by all proposals. The SUAVE/Oldenburg and Objective VHDL variable declarations and visibility scopes are restricted to the *process*. With our proposed extension, object declaration is permitted at the package, architecture, block, and process level, hence giving a greater variation in accessibility scopes. However, unlike VHDL 'shared variables', objects do not support multiple concurrent access (writing) to their 'instance variables'. As demonstrated, the need for multiple instantiation mechanisms found in SUAVE/Oldenburg and Objective VHDL (signal, variable and constant) is not justified. A single and versatile new mechanism provides a more suitable solution to object instantiation in an object-oriented extension to VHDL.

Besides static objects, the instantiation of dynamically allocated objects is implemented in all proposals through the use of VHDL *access* types.

4.3.7 Initialisation

Objective VHDL and our proposed extension allow class attribute initialisation. This follows the VHDL standard initialisation mechanism being either "(type'left)" or a user defined value. Nevertheless our proposed extension supports a more sophisticated initialisation mechanism with creator methods. Creator methods use the same names as the class that defines them.

A creator method can be overloaded and will be executed at elaboration time. The argument for more advanced initialisation mechanisms is based on the potential complexity of object-oriented data structures in comparison to the VHDL scalar or composite types.

Unlike Objective VHDL or our proposed extension, the SUAVE/Oldenburg extensions does not support user defined initialisation.

4.3.8 Assignment and Controlling Copying

All proposals support deep copy. The Shallow copy is implemented in both SUAVE and our proposed extension; however, the Objective VHDL proposal allows the overloading of objects' operators.

Object assignments are also supported by all proposals providing the assignments comply with the inheritance rules for implementing dynamic types. With the SUAVE/Oldenburg and Objective VHDL proposals, object assignment follows VHDL's signals, variables and constant mechanisms. With the proposed extension, only one mechanism is used: '=='. This assignment assigns references to objects as opposed to the object themselves.

4.3.9 Aggregation

Object aggregation is supported equally in all the proposed extensions. Classes are permitted to declare instance variables of other class types.

4.3.10 Method Invocation

Method invocation has been implemented differently in SUAVE/Oldenburg, Objective VHDL and our proposed extension.

The implementation of methods in SUAVE/Oldenburg relies on VHDL's subprograms. Consequently, method invocation in those proposals is in effect a VHDL subprogram call. Although this mechanism provides a solution to message passing, we argue that this style is not a practical implementation of message passing in an Object-Oriented programming language. Most OOPLs

aside from ADA 95, implement message passing through the use of the 'dot' notation. This notation is consistent and permits the use of a method call either as a statement or as an expression; this represents a significant advantage as detailed in section 3.2.11. The subprogram style of message passing forces the class designer to take decisions regarding the nature of message calls: either statements (procedures) or expressions (functions). Unlike method calls using the 'dot' notation, once the style of subprogram message call is set, the given message will only be used following the chosen definition style.

In addition to the restricted use of subprogram calls, the SUAVE/Oldenburg proposal requires the destination object for a call, to be passed as an *inout actual* parameter. Although this style follows VHDL's semantics, it represents a cumbersome implementation. Lastly, the SUAVE/Oldenburg implementation of message passing violates the VHDL LRM definition for the formal parameter class (file, signal, variable, constant). This is due to the fact that the SUAVE/Oldenburg proposals allow the creation of variable, signal and constant objects. To fix this violation, the VHDL language would have to be modified to allow the sub-programs parameter classes (signal, constant, and variable) to be part of the selection mechanism during overloading. Figure 4-28 illustrates message passing with the SUAVE/Oldenburg proposals.

```
package reg is
   type reg_t is tagged record
      value : std_logic_vector(7 downto 0);
   end record reg_t;
   procedure write(signal val: inout fifo_t;
                   signal clk: in std_logic;
                   value_in : in std_logic);
   function read(signal val: inout fifo_t)
                 return std_logic;
end package reg_t;
...
architecture rtl of sample is
   signal reg_1: reg_t;
begin
   write(reg_1, clock, 33); -- statement style
   out_a <= '1' when read(reg_1) = 10 -- expression style
            else '0'
   -- more code
```

Figure 4-28. Message Passing in SUAVE / Oldenburg .

The Objective VHDL solution to message passing resembles the SUAVE / Oldenburg insofar that it uses sub-program calls. However, this proposal relies on the 'dot' notation to associate the target object with a message. Although this style is more suited for the implementation of an Object-Oriented message mechanism, the use of sub-program calls as statements or expressions restricts the message passing abilities of the proposed extension. Furthermore, to support the use of different object implementations (signals, variables, constants), the Objective VHDL proposal requires the building of a complex object configuration mechanism. This configuration mechanism defines the behaviour of methods according to the object declaration (signal, variable and constant). This creates a significant amount of code redundancy for the creation of a versatile class.

The example in Figure 4-29 demonstrates the use of message passing with the Objective VHDL proposal.

```
type reg is class
   class attribute value : std_logic_vector(7 downto 0);
   for signal
      procedure write(signal clk: in std_logic;
                      value_in : in std_logic);
      function read return std_logic;
   end for;
end class reg;
...
architecture rtl of sample is
   signal reg_1: reg;
begin
   reg_1.write(clock, 33); -- statement style
   out_a <= '1' when reg_1.read = 10 -- expression style
            else '0'
   -- more code
end rtl;
```

Figure 4-29. Message Passing in Objective VHDL.

Unlike SUAVE/Oldenburg or Objective VHDL, our proposed extension implements a truly Object-Oriented-like message call mechanism. This is achieved by using a newly defined method mechanism as opposed to VHDL sub-programs. This style enables us to create flexible method calls that do not need to be identified and previously defined as statements or procedures. Method calls are used as both statements and expressions on a single type of element: the *object*. Unlike the SUAVE/Oldenburg or Objective VHDL proposals, the use of a unified *object* mechanism instead of signals, variables and constants largely simplifies the creation of methods. No special configuration or complex and redundant overloading operation is required for the use of objects through message passing. Furthermore, our proposed extension defines two kinds of message calls: blocking and non-blocking. These call variations present a significant modelling advantage over other proposed extensions (as detailed in section 3.3.3). This mechanism is borrowed from the Verilog world and enables sequential activities to be

spawned in parallel as well as sequentially. An example of message passing with our proposed extension is given in Figure 4-30.

```
type reg is class
   instance variable value : std_logic_vector(7 downto 0);
   method write(signal clk: in std_logic;
                value_in : in std_logic);
   method read return std_logic;
end class reg;
...
architecture rtl of sample is
   object reg_1: reg;
begin
   reg_1.write(clock, 33); -- statement style
   out_a <= '1' when reg_1.read = 10 -- expression style
            else '0'
   -- more code
end rtl;
```

Figure 4-30. Message Passing in our Proposed Extension.

4.3.11 Polymorphism/Dynamic Types

All proposed extensions support polymorphism. The SUAVE/Oldenburg and Objective VHDL, require the definition of signals, variables, constants or parameters of class wide types. With our proposed extension, no specific indications are required for implementing a class-wide type.

Errors regarding dynamic type mismatches will occur during run time for all proposed extensions.

4.3.12 Genericity.

All proposals implement class genericity. The SUAVE proposal allows creation of generic types as well as generic constants. Generic types are akin to C++ templates and can be seen as a significant benefit for the creation of reusable classes. However, we do not feel that such a mechanism is suited for a strongly typed language such as VHDL. The use of template-like mechanisms might compromise the type system of the language and confuse the language users.

4.4 Conclusions

This chapter has defined the approach taken by the main OOVHDL proposals. It highlighted the strengths and weaknesses of the languages through examples based on applications and suggested novel mechanisms to tackle modelling issues not supported by current extensions. A number of conclusions can be drawn from this assessment. Although all proposals are implementing the language design requirements specified by the IEEE OOVHDL study group, the implementation choices have a significant effect on the usability of the different proposals. The SUAVE/Oldenburg proposals require a minimum number of syntactic constructs to implement a form of Object-Orientation. However, the support is minimal: dynamic typing and single inheritance. Furthermore we demonstrated that the choice of existing VHDL constructs (Packages, sub-programs, signals, constants) was inadequate. Such constructs added an overlap with the existing VHDL mechanisms as well as restricting the implementation of Object-Orientation inside VHDL. The Objective VHDL solution takes a more pragmatic approach to the implementation of an OOVHDL. The creation of a new ADT gives an advantage to this solution over the SUAVE/Oldenburg proposals. However although this proposal demonstrates a level of independence from the ADA95, it still relies on non adapted mechanisms for object implementations. Furthermore, like the SUAVE/Oldenburg proposals no class adaptation mechanisms are provided in Objective VHDL. Our proposed extension has shown its superiority in terms of language integration and Object-Orientation implementation. In our proposal, further mechanisms are provided to implement encapsulation control, multiple inheritance, run time validity checking, feature mapping, and dynamic typing.

Chapter 5.

CASE STUDY: THE EDGE FILTER DESIGN

5.1 Introduction.

This chapter documents a test case object-oriented VHDL model, designed to include the characteristics found in a typical hardware design.

The test case implements an Edge Filtering image processing function on a raster scan input image, using 3X3 pixel masks. This chapter contains an overview of the Edge Filtering function; defines the partitioning of the design and includes a description of the supporting class structure implementation. It concludes with a discussion of the benefits of using object-oriented VHDL models compared to equivalent standard VHDL models.

5.2 Overview of Function.

An Edge Filter is an Image Processing function which identifies the edges of objects within an image, where the image is defined in terms of a pixel array of intensity values. An object within the image is defined as an area of pixels with little variation in intensity. Edges of objects are represented by pixels with a wide variation in intensity, and the direction of the object edge is perpendicular to the direction of intensity variation.

The operation of the edge filter circuit chosen is summarised below. A more detailed description is given in (IEEE 1982).
The edge strength for each pixel is calculated by comparison with all adjacent pixels in the image using a 3X3-pixel mask. Mask coefficients determine the contribution of each pixel to the central pixel's edge strength, i.e.

	c-1	c	c+1
r-1	a	b	c
r	d	e	f
r+1	g	h	i

For a particular pixel (c,r), the edge strength is given by:

$$E(c,r) = [\, a \times P(c-1, r-1) + b \times P(c, r-1) + c \times P(c+1, r-1) + d \times P(c-1, r) + e \times P(c, r) + f \times P(c+1, r) + g \times P(c-1, r+1) + h \times P(c, r+1) + i \times P(c+1, r+1) \,] / j$$

Where 'a' to 'i' are mask coefficients, 'j' is the mask divisor, and $P(x,y)$ are pixel intensities in the adjacent rows and columns to the current pixel.

Two masks are used to detect edges in the West/East direction (WE) and in the North/South direction (NS) using the following coefficients.

-1	-2	-1
0	0	0
1	2	1

NS edge mask

-1	0	1
-2	0	2
-1	0	1

WE edge mask

For both masks, the mask divisor is 4.

These masks serve to filter out low variations of intensity over the pixel area covered by the mask. The greater the intensity variation in the direction covered by the mask, the greater the edge strength generated by the mask.

Each mask (WE, NS) generates edge strengths for each pixel location. The strength can be either positive or negative depending on the direction of increasing intensity for the edge. The absolute value of edge strengths is

compared to a threshold. The binary result of the comparison signifies the presence of an edge on either the NS, WE or both directions of the image.

5.3 Filter External Interface.

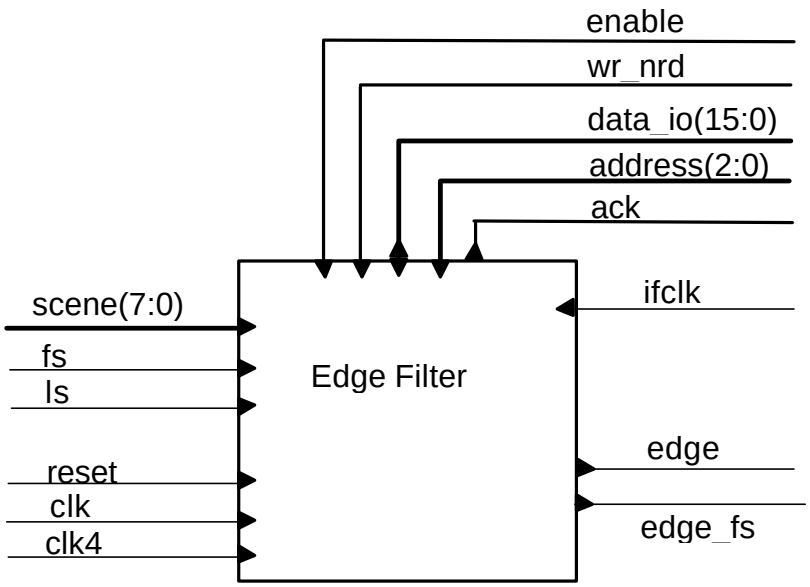


Figure 5-1. Edge Filter Top Level.

Image intensity data is input as an 8-bit unsigned bit vector (scene) in a raster scan format, synchronised to a pixel clock (clk). Also synchronised to the pixel clock are single bit frame (fs) and line sync (ls) inputs for the image data, and a quad-speed clock (clk4). A reset input bit is also used (reset).

Binary edge data (edge) is output in a raster scan format, together with a single bit frame sync output (edge_fs) which is set for the last pixel edge data in a frame. The edge data can be read out in short bursts (one line maximum) at a higher rate than the pixel clock. An output status bit (fifo_empty) is cleared when edge data is available for output.

Information is written to and read from the filter via a bi-directional 16-bit databus (data_io), using a 3-bit address input (address) to access internal registers and a read/write input bit (wr_nrd) to control dataflow direction. An

input interface enable bit (enable) and an output acknowledge bit (ack) are used to synchronise input and output of data. A separate clock (ifclk) is required by the interface.

5.4 Filter Partitioning.

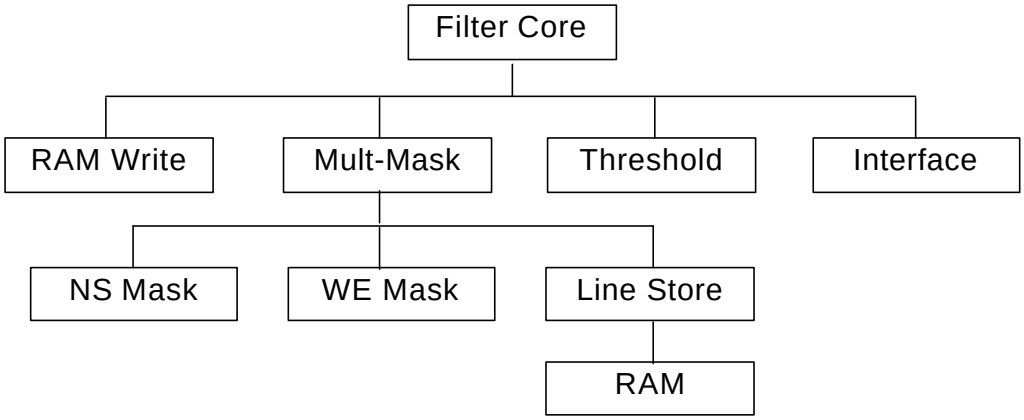


Figure 5-2. Edge Filter Decomposition.

The filter is partitioned into a mask block, a threshold and an interface block as shown in Figure 5.2. The partitioning of the filter is based on functional decomposition. This style of decomposition is the most commonly used in VLSI design and was chosen to illustrate how the proposed semantic improvements can be beneficial when used in typical VHDL based designs. Alternatively, a completely object oriented design decomposition could be used; however, we do not believe that this would be a pragmatic approach to a Register Transfer Level style of design.

5.4.1 Mult-Mask.

This block implements the two 3X3 pixel mask filters and two Line Store FIFO's. The block contains two instances of a mask model (NS Mask, WE Mask) and two instances of a generic FIFO model (Line Store). The FIFO's

are required to store the pixel intensity data from previous image lines for the top and middle rows of the mask filters.

Mult Mask also contains a small state machine which uses the frame and line sync inputs to synchronise the Edge Filter to the start of a new frame after reset, and to generate new frame and result blanking signals for the output data. Result blanking sets the mask result data to a default value over areas of the image where not all the mask data is valid, e.g. the first two lines of an image where the top and middle rows of the mask do not contain current image data.

5.4.1.1 Mask blocks (wemask, nsmask).

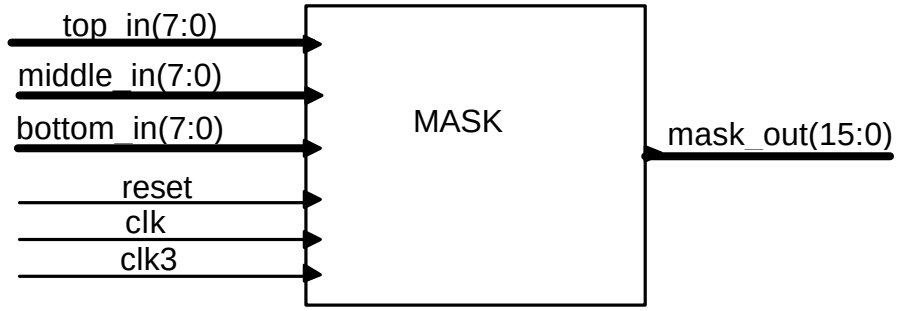


Figure 5-3. Mult-Mask Top Level.

The Mask block inputs 8-bit intensity data from the external data input and from both Line Store FIFO's and it shifts each data input through a register bank representing the three rows and three columns of the mask. On each shift, the mask will multiply each mask location by the appropriate coefficient and sum the results.

5.4.1.2 Line Store.

Figure 5-4. Line-Store Top Level.

Each Line Store is a FIFO based on a RAM with eight-bit words for the intensity data and six bit addresses representing the maximum image length allowable.

5.4.2 Threshold Multiplexor.

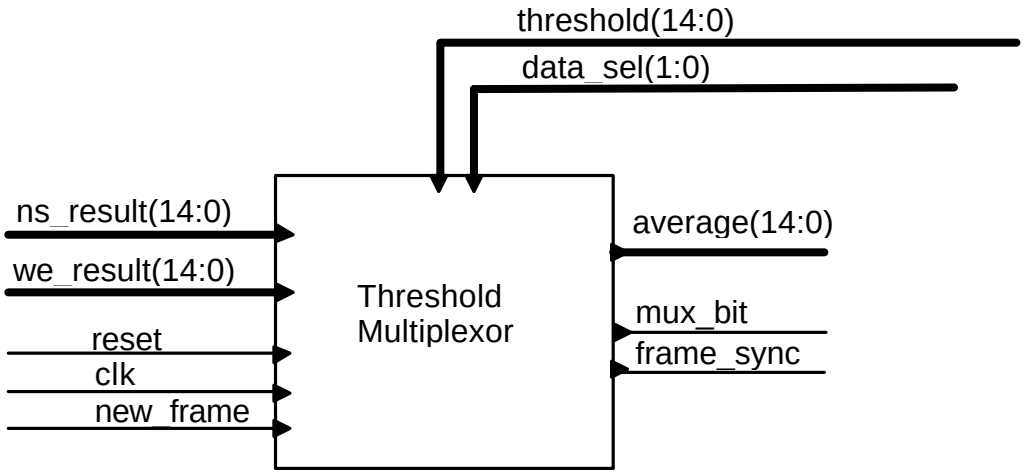


Figure 5-5. Threshold Multiplexor Top Level.

This block converts the 15 bit unsigned edge strength data, from the two mask filters, into a single binary output bit. Both mask results are compared to a threshold value to convert them into binary. The specific binary data to be output is determined by the 2-bit data_sel input according to the following table:

data_sel	Output	comment
01	NS	NS (row) edges
10	WE	WE (column) edges
11	NS OR WE	row and column edges
00	NS AND WE	"diagonal" edges

Table 5-1. Threshold Decoding Table Top Level.

This block also contains an “averager” function, which calculates a running average of the 15-bit edge strength data from the masks. The averager uses data_sel to determine which mask output to average, i.e. if only one of the mask outputs is selected, the averager only uses that output. If both masks are selected, the largest edge strength is used. In all cases, zero edge strengths are ignored to avoid skewing the average result. In a real application, average information would be important in deciding a suitable value for the threshold.

The threshold and data select values are written into the filter via a microprocessor interface. The running average value is read out via the microprocessor interface.

5.4.3 Interface.

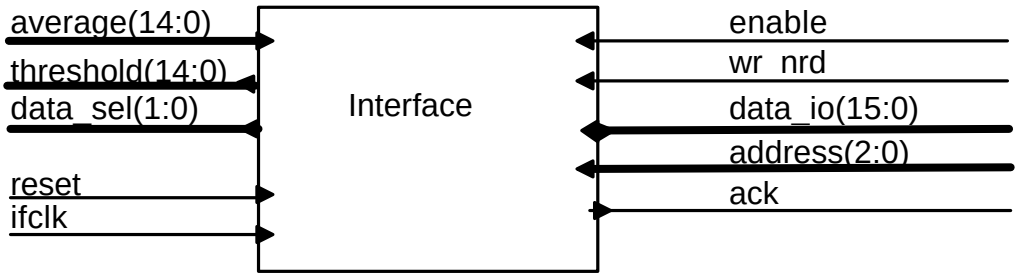


Figure 5-6. Interface Top Level.

The interface handles read and write cycles initiated by the asynchronous microprocessor interface bus. There are two write and one read cycle operations as follows:

cycle	Data	address	Databus bits
write	Threshold	010	0 to 14
	Data select	001	0 to 1
read	Average	100	0 to 14

Table 5-2. Detail of Write and Read Operations.

All input/output data communication is with the threshold multiplexor block. Interface must detect read or write cycle initiation on the microprocessor interface and input or output data as required using the acknowledge output.

5.4.4 RAM.

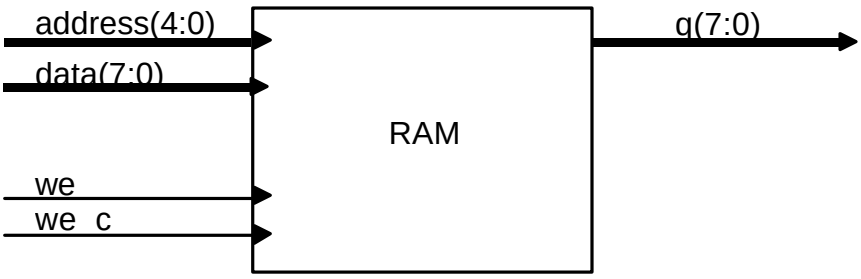


Figure 5-7. RAM Top Level.

The RAM is a synchronous memory, with registered address lines. Two separate data ports are used for reading from and writing to the RAM block. The RAM block is used as part of the Line Store block for implementing a line FIFO. Two RAM blocks are used in the Edge Filter design for the storing the top and middle lines.

5.4.5 RAM Write

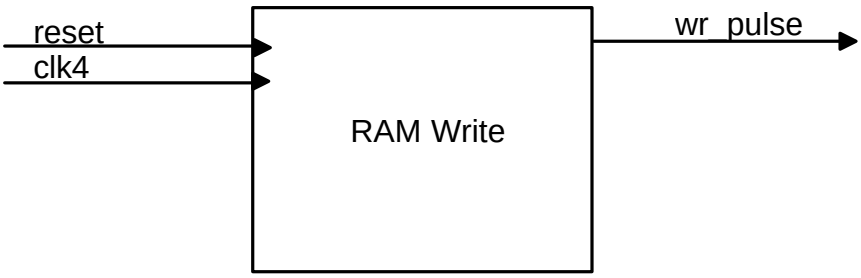


Figure 5-8. Ram Write Top Level.

The RAM-write block produces the write enabling signals for the FIFO RAMs.

5.4.6 Filter Core

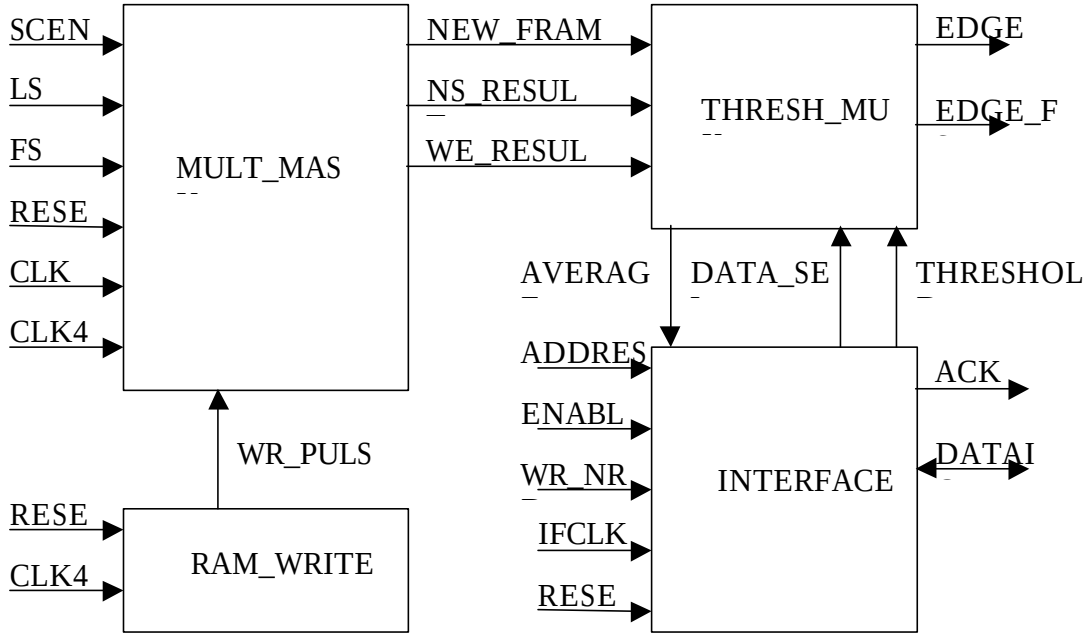


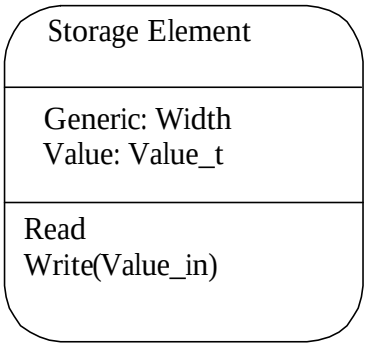
Figure 5-9. Filtercore Top Level.

The Filter Core module connects all the sub-components of the Edge Filter Design.

5.5 Supporting Class Structure

In order to create the edge filter design, a generic class structure was created (see Appendix B.1). The initial classes were made as abstract as possible, to allow the class structure to grow as new objects were created during the building of the edge filter sub-blocks. Generic parameters and virtual classes were used for this purpose.

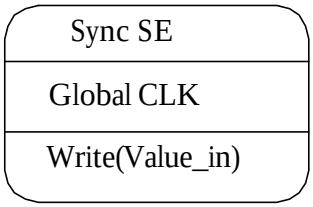
5.5.1 Storage Element



The *storage element* class is the base element of the class structure. It was decided to use the std_logic_vector type for the *value* instance variable. The integer type would have been more suited for most arithmetic operations, however using the Synopsys 'std_logic_unsigned' package (www.synopsys.com) compensates for the limitations of VHDL's vector arithmetic. A subtype *value_t* was created to simplify the sizing of the *value* instance variable throughout the entire class tree. The Generic parameter *depth* is used to set the dimension of *value* and provide a flexible data structure for the class.

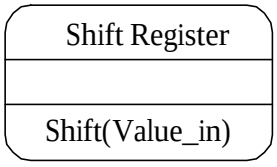
Two methods were created a *read* and *write* method both publicly accessible. The *read* method only returns the content of the instance variable *value*. The *write* method changes the content of the instance variable *value*. The default mode of operation for both *read* and *write* was set to be asynchronous to allow the definition of classes that feature both a synchronous and an asynchronous behaviour later in the design stage.

5.5.2 Synchronous Storage Element



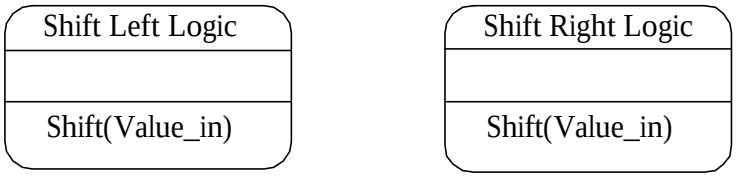
The derived synchronous storage element class is a small variation of the previous class. This time the *write* method has been overloaded to provide a synchronous version. The inherited *write* method was mapped into a new name: *write_async*; this will protect from ambiguities in the two styles of operations. The global element *clk* was created to allow the visibility of a common control signal throughout all the subsequently derived classes. Although the use of a global element can be seen as a violation of the encapsulation principle; pragmatically when modelling hardware modules, synchronisation signals are often required and do not represent information (part of the data flow) as such. For this reason, control and synchronisation signals should be declared as global.

5.5.3 Shift Register



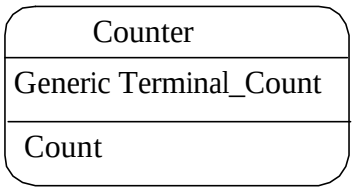
The virtual class *shift register* is one of the variations of the *sync_se* class. This class only defines one virtual method *shift*. The method is said to be virtual since its implementation will be defined in later classes. The shift register class will be used in designs where late binding is required.

5.5.4 Shift Right Logic & Shift Left Logic



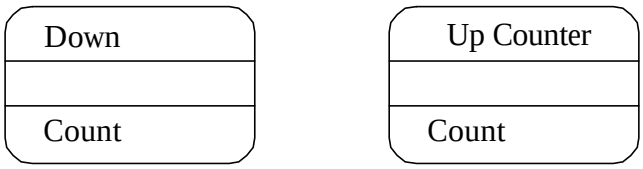
Both classes are derived from the virtual shift register class. The two classes are defined since they both implement their own version of the virtual *shift* method defined in their parent class.

5.5.5 Counter



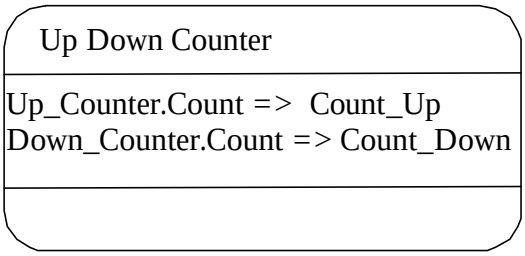
The virtual *counter* class is derived from the *synchronous storage element*. This class defines a virtual method *count* as well as a generic parameter *terminal_count*. The generic will be used to set the top boundary for the counter.

5.5.6 Up Counter & Down Counter



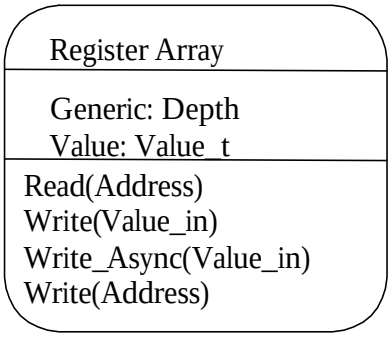
The two classes *up counter* and *down counter* are derived from the virtual counter class; Both classes implement variations of the virtual *count* method.

5.5.7 Up-Down Counter



The *up down counter* class is derived from both the *up counter* and *down counter* classes. The method map construct was used to map the *count* method of both parent classes into a *count_up* and *count_down* method. The remaining elements of both classes are merged into unique elements inside the *up down counter* class. No code is required for the creation of this class since all the required functionality is already inherited from both parents.

5.5.8 Register Array

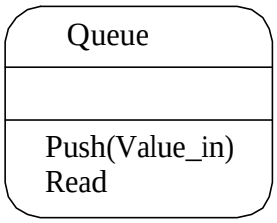


The *register array* class is based on the *synchronous element* class; it defines a new generic parameter: *depth*. In addition, the class redefines all the inherited methods as well as the instance variable *value*. The *value* instance variable is defined as an array of std_logic_vector of dimensions *width*depth*. The *write_async* method will asynchronously write its formal parameter (*value_in*) to

all locations of the array. The *write* method is overloaded and can synchronously either fill the array with the value of its formal parameter or write a single element by using the *address* formal parameter. The *read* method performs a read at a defined location of the array. The register array class should be used to model a synchronous RAM or a register bank.

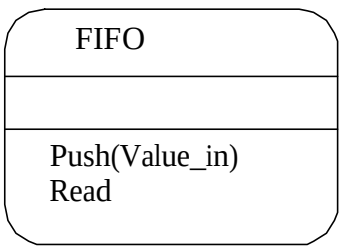
Although this class redefines all of its inherited properties, it is still useful to categorise it as a child of the synchronous storage element class since they share common behaviours such as the synchronous and asynchronous *write*.

5.5.9 Queue



The virtual *queue* class is derived from the *register array* class; it defines two virtual methods: *push* and *read*. The two methods were identified as common to most queues regardless of their implementations. The queue class will only be used in designs to promote late binding and polymorphism.

5.5.10 First In First Out (FIFO)



The *fifo* class is a child of the queue class. It defines the implementation of both *push* and *read* methods. The *push* method will write sequentially the

formal parameter *value_in* to the *value* array. The *read* method will extract the first value written to the *value* array and point to the next element in the sequence.

Two instance variables *is_full* and *is_empty* have been declared. These two public variables are used as flags to display the *fifo's* current status. The user of the FIFO class can directly interrogate (no method calls needed) the *fifo* status by reading the values of those two flags.

The *fifo* uses an *up_down_counter* object to implement the address pointer. This dependence is represented in the class structure diagram by a connection line ended by a circle.

5.6 Edge Filter implementation

5.6.1 Mult Mask

Division of the sum of the mask coefficient and pixel intensity products by the mask divisor is carried out in this model.

Two objects of the *shift left logic* class are used in the process *data_reg* for creating a latency of two clock cycles on the two signals *ls_sync* and *fs_sync.*

The two objects are defined with a generic set to 2 and a global connection to the *clk* signal. The initialisation of the objects is done asynchronously on the *reset* signal. The shift operation is done synchronously on the edge of the signal *clk*. Two concurrent method calls (last two statements) continuously update the value of the *ls_sync* and *fs_sync* signals.

5.6.2 Mask Blocks (NS mask, WE mask)

The masks must carry out nine multiplications and sum nine results within a single clock period. In order to save on arithmetical resources, the calculation is carried out over three clock periods using a quad-speed clock *clk4*, with a specific phase relationship to the input data clock *clk*. The mask calculation is

carried out on three banks of three 8 bit registers. The register banks are implemented as FIFO objects of size 3*8 bits namely: *top*, *mid* and *bot*.

The three FIFOS are gathered into an array (*fifo_array*) to facilitate global operations. The process *mask_shift* resets the FIFOs asynchronously on the *reset* signal. The reset operation is done globally by sending the *write_async* request to the *fifo_array*. Furthermore, the *mask_shift* process updates the contents of the *top*, *mid* and *bot* FIFOs on an edge of the *clk* signal.

The content of the three FIFOs is read inside the *calcomb* process in order to perform the required arithmetic operations. Although a normal FIFO would only allow read access to its first element, in our case, the *fifo* class is derived from the *register array* class which permits access to its elements by providing an address.

5.6.3 Line Store

A dual port RAM is used for the FIFO. An output from the *RAM Write* block is used as the RAM write-enable. *Read* and *write* pointers are set one line length apart on reset, and increment on the positive and negative edges respectively of the pixel clock, wrapping to zero at the end of the line. The two counters (*rd_ptr*, *wr_ptr*) are implemented using instances of the *up counter* class. The global *clk* signals are mapped to the signals *ls_clk* and *not_ls_clk* respectively. This allows the counter to increment their value on the negative and positive edges of the *ls_clk* signal. The two counters have been aggregated into a counter array *add_ptr* to facilitate the execution of common operations. The process 'read_wite_pointer' asynchronously resets the counters on the *reset* signal and increments them on a change of the *ls_clk* signal. The internal values of the two counters are then multiplexed into the *int_ad* (memory address) signal depending on the value of the *ls_clk* signal.

5.6.4 Threshold Multiplexor

The threshold and multiplexor functions are straightforward. The *averager* function is slightly more complicated, since three separate comparison

operations are required. Inequality comparators check NS and WE edge strengths are not equal to zero, and a "less than" comparator checks if NS edge strength is greater than WE. No objects were used in this block since the functionality implemented is very specific and cannot benefit from the use of predefined classes.

5.6.5 Interface

Interface uses a clock input (*ifclk*) to synchronise the read/write cycle operations. *Address*, *read/write enable*, and the *databus* are registered on the falling edge of *enable*. The registered *data*, plus *enable*, are then synchronised to the clock using double registers to avoid meta-stability problems. The interface clock is not assumed to be synchronous to the input data pixel clock, so the average data input from the threshold multiplexor block is double registered to the interface clock at the inputs. The interface block defines three *fifo* objects for the *address*, *data* and *average*. In addition, two *shift register* objects for the *write* and *enable* signals are declared. Both the *fifo* and *shift register* objects are used to introduce a latency of two clock cycles into the data path. The process *sync_reg* resets all the objects synchronously and updates their value on a positive edge of the *clk* signal. A number of intermediate signals (*wnr*, *date2*, *addr2* and *enable*) have been declared to facilitate access to objects' content. Concurrent assignments are made to the intermediate signals at the beginning of the architecture block; this allows access to part of the signals (slices of an array) when needed by the rest of the design.

5.6.6 RAM Write

The RAM Write module relies on a Gray counter active on *clk4*. The *wr_pulse* is created by combining two bits of the counter's output. This implementation did not require the use of any objects.

5.6.7 RAM

The RAM block uses a clocked process to synchronise the reading and writing to the signal *clk* The Register Array object is used to implement the RAM. Additional statements are used to enable the writing operations using the *we* signal.

5.7 Comparative Study

This study compares the implementation of the same edge filter design using both, the proposed object-oriented extension and standard RTL VHDL. This analysis highlights the differences in terms of number of lines of code per process as well as an estimate of the amount of code reuse in the object-oriented version of the design. Both design files can be found in Appendix B.

From a first look at the comparative table, it can be seen that some modules benefit more than others from the use of the proposed semantics extension. Modules with a small number of statements such as: *Filter Core*, *RAM Write* and *RAM* do not show improvements in terms of number of lines. Depending on their application specificity, those modules are either almost completely created from reusable code (RAM) or not at all. This result is to be expected of any higher-level design method. Similar results can be observed when comparing the building of low level gate primitives such as a multiplexor at both the transistor level and the gate level. Remarkably, although it would require more silicon to construct a multiplexor using a higher level modelling method (`OUT = IN1 AND NOT SELECT OR IN2 AND SELECT`), with the proposed object-oriented models, no additional implementation effort (aside from the building of the original class structure) is required.

Modules of a larger size have a more varied level of code reuse. The *Line Store* module has the highest level of reuse amongst large size modules. This level of reuse is due to efficient use of object instances, polymorphism and a low level of application specific code. In this example the number of sequential

statements is lowered by 60 percent. However, the number of concurrent statements has slightly increased in order to implement the polymorphism (array of counters).

In contrast, the level of reuse in the *Mult Mask* module is limited. This is the result of the high level of specificity of that module.

Similarly, the *Threshold* module is specific to the Edge Filter design and cannot benefit from the use of generic objects.

The remaining modules: *Interface*, *WE Mask* and *NS Mask* use an estimated 30 percent of reused code. This is an expected level of reuse for less application-specific modules. When the object oriented extensions to the semantics are used, a 30 to 50 percent code reduction is observed at the process level. However, the number of concurrent statements is constantly higher when using the new semantics. This increase in the number of concurrent statements is due to the use of temporary signals to access objects' instance variables. This can be easily avoided by changing the encapsulation control of instance variables from *restricted* to *public.* Although the change of encapsulation level would represent the simplest solution to instance variable access, this would go against the principle of data hiding.

Looking at the overall design, the size of the version using the proposed semantic enhancements has been reduced by 11 percent. Although this represents a small improvement in terms of numbers of lines of code, 22 percent of the code needed for the creation of the object-oriented version of the design is made of reusable components. This suggests a faster design capture and a higher level of reliability.

Module	Standard VHDL	OO Extension	Reuse (%)
RAM	11	11	~90
Sync_ram	10	10	
Concurrent	1	1	
Line Store	39	16	~80
Readwrite pointer	34	9	
Concurrent	5	7	
NS Mask	74	60	~30
Calcseq	25	25	
Calcomb	22	22	
Maskshift	27	13	
Concurrent	0	1	
WE Mask	66	53	~30
Calcseq	25	25	
Calcomb	14	14	
Maskshift	27	13	
Concurrent	0	1	
Mult Mask	129	129	~10
ns_complete	15	15	
we_complete	15	15	
state_reg	11	11	
state_assign	58	58	
data_reg	25	21	
Concurrent	5	9	
Threshold	69	69	None
Averager	27	27	
Cont_regs	16	16	
Thresh_reg	23	23	

Concurent	3	3	
Ram Write	10	10	None
gray_code	9	9	
Concurrent	1	1	
Interface	83	78	~30
Async	13	13	
if_wr	26	26	
Sync_reg	29	19	
Store_reg	11	11	
Concurrent	4	9	
Filter Core	4	4	None
Concurrent	4	4	

Table 5-3. Case Study Comparative Table.

5.8 Conclusions

This design study compared two functionally identical implementations of a moderately complex design using both the standard VHDL RTL style and our proposed object-oriented semantic extensions. This design represented a real example and contained a balanced amount of control logic and data path logic.

From the functional decomposition, basic building blocks were identified. A class structure was created to support the requirements of the Edge Filter design. The design of the class structure was a top-down and bottom up process. Numerous virtual classes were created to facilitate inheritance and promote the use of polymorphism and late binding. The depth of the class tree was voluntarily kept to a maximum of 4, so as to ease the use of the classes. After the creation of the design, a comparative study was carried out to highlight the strengths and weaknesses of the proposed extension. The study has shown that the benefits of using an object oriented implementation

depends greatly on the level of specificity of a given module. For the example chosen, a 30 to 50 percent code reduction was achieved at the process level. In addition, the amount of reused code reached 22 percent on the overall design. Consequently, this study demonstrated the benefits of the enhanced semantics in terms of faster design capture, better reliability and reduced verification effort.

The calculation of the given reuse figure was derived from the count of the number of both sequential and concurrent statements inside the standard VHDL description as well as the Object-oriented description. This study did not consider the creation of the test harness. Nevertheless, due to the behavioural nature of a VHDL simulation test harness, similar or further levels of reuse are expected. The creation of classes of pattern generators such as preset, random, sequential or file-based could be envisaged. Furthermore existing objects created for the edge filter design may also be used in the creation of a complex test harness.

Chapter 6.

PREPROCESSOR DESIGN

6.1 Introduction

A number of small test cases were developed during elaboration of the Object-oriented VHDL semantics. These test cases were validated by manually translating Object-oriented VHDL code into standard VHDL then simulating the results using generic VHDL simulators. Although this translation process was sufficient for validating individual test cases, it soon became apparent that the creation of an automated translation mechanism or "pre-processor" was required for larger projects. The development of the pre-processor is described in this appendix. The semantics for the proposed extension presented in this appendix use an earlier style used during the refinement process of the final language extension. However, the translation mechanism presented can equally be applied the latest version of the language extension.

6.2 Requirements

Since VHDL is a platform and tool independent language, the manual translation of Object-oriented VHDL to standard VHDL produced code suitable for any standard VHDL simulator. However, the choice of operating system and development language used for the pre-processor limited its portability. Since IBM sponsored this research project, there was a requirement for the pre-processor to be executable on RS6000 workstations, using the AIX (UNIX like) operating system. Nevertheless, since the accessibility of RS6000 workstation is scarce in the engineering world, the pre-processor also had to be developed on SUN and Hewlett Packard workstations. A PC (LINUX) based version of the pre-processor was also created at a later stage. The UNIX based development tools were: GNU C,

(f)lex , (b)yacc and sed. Lex and Yacc were used for the creation of the scanner and parser part of the pre-processor. GNU C was used for the creation of the object-oriented to standard VHDL translator. Sed was used for text file processing and formatting.

6.3 System Design

The functions required by the pre-processor can be split into three categories:

1. Creation of the abstract syntax tree (treebuild.lex, treebuild.yacc, treebuild.c).

2. Extraction/Creation of supporting design units (file_handling.c).

3. Formatting of the original code into standard VHDL code (process.c, top.c).

The activities are summarised in Figure 6-1.

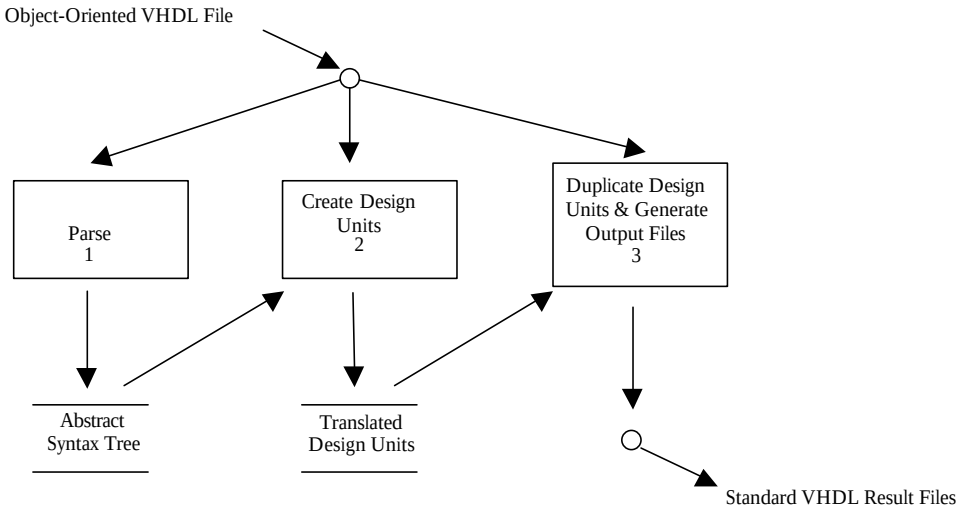


Figure 6-1. Operation of Pre-Processor

6.4 Specification of the Translation Mechanism

The main issue in the design of the pre-processor was the creation of human readable standard VHDL code. The results found from the language study (detailed in chapter 2) were used and expanded for the creation of the translator part of the pre-processor.

In order to implement classes, VHDL packages were used. Methods were translated into procedures whose visibility depended on their encapsulation (public/private/restricted). Similarly, "shared variables" were used to implement the class's instance variables. Figures 6-1 and 6-2 illustrate the translation mechanism for a basic class.

```
class SHIFT_REG is
   type vector is array (9 downto 0) of integer;
   value : vector;
begin
public method reset is
begin
   value := (others => 0);
end method reset;

public method shift(signal clk: bit; value_in: integer);

public method read return vector is
begin
        return value;
end method read;
end class SHIFT_REG;
```

Figure 6-2. Class Declaration

```
package shift_reg is
    type vector is array (9 downto 0) of integer;
    shared variable value :  vector;

    procedure  reset;
    procedure  read( return_1 : out  vector );

end shift_reg;

package body shift_reg is

    procedure reset is
    begin
        value := (others => 0);
    end reset;

    procedure read( return_1 : out  vector ) is
    begin
        return_1:= value;
    end read;

end shift_reg;
```

Figure 6-3. Translated Version of the Class Declaration.

The translation of a derived class requires information to be collected from the parents' classes (multiple inheritance). The translation is done according to the encapsulation attribute of the inherited items, as well as the type of inheritance specified. Furthermore, due to the mapping capabilities defined in the extension, the translator has to perform a number of name changes and merging (instance variables, methods, etc.) in the derived class. These changes are done according to the mappings formally specified. Table 6-1 summarises the translation process according to the encapsulation attribute and the type of inheritance.

	Inheritance Type		
capsulation Type	Public	Private	Restricted
blic	PH/PB	PB	PB
vate	PB	PB	PB
stricted	PB	PB	PB

Table 6-1. Translation Process According to Inherited Attributes

PH: Signifies the declaration of the translated element in the Package Header.

PB: Signifies the declaration of the translated element in the Package Body.

In the proposed extension, the inherited code from parent classes does not physically appear in a child class. However, when the translation of a child class is performed, all the code required for the creation of the corresponding standard VHDL package is required. This explains why, elements of private and restricted encapsulation type have to be copied to the package bodies after an inheritance takes place. The translation of a derived class is shown in

```
class ShiftRightReg use public shift_reg is
begin
public method shift(signal clk: bit; value_in: integer) is
begin
        if clk'event and clk='1' then
                value := value_in & value(9 downto 1);
        end if;
end method shift;
end class ShiftRightReg;
```

Figures 6-4 6-5.

Figure 6-4. Child Class of the ‘shift_reg’ Class.

The VHDL language reference manual (DASC 1993) specifies that a package is shared across the design units that declare it. Consequently, for a number of objects *N* of a given class declared inside a design, there should be *N*

different instances of the same package defined. The translator gathers the number of objects of the same class from the abstract syntax tree. As a result, separate copies of the original package are created, each with a unique name.

```
package shiftrightreg is
    type vector is array (9 downto 0) of integer;
    shared variable value :  vector;
    procedure  shift( signal clk: bit ; value_in: integer );
    procedure  reset;
    procedure  read( return_2 : out  vector );
end shiftrightreg;

package body shiftrightreg is
    procedure shift( signal clk:  bit ; value_in:  integer ) is
    begin
        if clk'event and clk='1' then
                value := value_in & value(9 downto 1);
        end if;
    end shift;

    procedure reset is
    begin
        value := (others => 0);
    end reset;

    procedure read( return_2 : out  vector ) is
    begin
        return_2:= value;
    end read;
end shiftrightreg;
```

Figure 6-5. Translated Version of a Child Class.

The last step of the translation consists in converting object-oriented method calls to standard VHDL procedure calls. Once the translation is performed, the remaining standard VHDL code is appended to the translated code.

An example of object use is shown in Figure 6-6.

```
process
        variable temp: integer := 33;
begin
        shift_reg1.reset;
        shift_reg2.reset;
        shift_reg3.reset;
        counter1.reset;
        wait for 30 ns;
        for i in 0 to 3 loop
                wait until clk= '1';
                shift_reg1.shift(clk, 10);
                counter1.count;
        end loop;
        -- temp := shift_reg1.read;
        shift_reg2.shift(clk, temp);
        shift_reg3.ShiftRight(clk, temp);
        wait until clk = '1';
        shift_reg3.ShiftLeft(clk, temp);
        wait for 20 ns;
end process;
```

Figure 6-6. Object-Oriented Method Calls.

```
process
        variable temp: integer := 33;
begin
        work.shift_reg1.reset;
        work.shift_reg2.reset;
        work.shift_reg3.reset;
        work.counter1.reset;
        wait for 30 ns;
        for i in 0 to 3 loop
                wait until clk= '1';
                work.shift_reg1.shift(clk, 10);
                work.counter1.count;
        end loop;
        -- work.shift_reg1.read( temp );
        work.shift_reg2.shift(clk, temp);
        work.shift_reg3.shiftright(clk, temp);
        wait until clk = '1';
        work.shift_reg3.shiftleft(clk, temp);
        wait for 20 ns;
end process;
```

Figure 6-7. Translation for Method Calls.

6.5 Full Application Example

This example features all of the mechanisms presented in this section. A base class ‘shift_reg’ is declared with two defined methods: ‘reset’ and ‘read’ and a virtual method ‘shift’. Two classes are directly derived from the virtual class ‘shift_reg’, namely: ‘ShiftRightReg’ and ‘ShiftLeftReg’. These two child classes define the virtual method ‘shift’ inherited for the ‘shift_reg’ class. Two additional child classes are defined: ‘Counter’ and ‘BidirShiftReg’ using single and multiple inheritance respectively. In addition to multiple inheritance, the ‘BidirShiftReg’ class illustrates the feature-mapping mechanism.

In this example, four different objects are declared. The declared objects are then used in a process to illustrate different method call styles.

The following code shows both the original Object-Oriented code and the translated version.

The Original Object-Oriented Code is shown in figure 6-8.

```
entity my_design is
end my_design;

architecture OO of my_design is

class SHIFT_REG is
   type vector is array (9 downto 0) of integer;
   value : vector;
begin

public method reset is
begin
   value := (others => 0);
end method reset;

public method shift(signal clk: bit; value_in: integer);

public method read return vector is
begin
        return value;
end method read;

end class SHIFT_REG;

class ShiftRightReg use public shift_reg is
begin

public method shift(signal clk: bit; value_in: integer) is
begin
        if clk'event and clk='1' then
                value := value_in & value(9 downto 1);
        end if;
end method shift;

end class ShiftRightReg;

class ShiftLeftReg use public shift_reg is
begin

public method shift(signal clk: bit; value_in: integer) is
begin
        if clk'event and clk='1' then
                value :=  value(8 downto 0) & value_in;
        end if;
end method shift;

end class ShiftLeftReg;
```

```
class counter use public shiftrightreg is
begin

public method count is
begin
       for i in 0 to 7 loop
          value(i) := value(i) + 1;
       end loop;
end method count;

end class counter;

class BidirShiftReg use public ShiftRightReg, public ShiftLeftReg
is
        generic (delay: time := 3 ns);
        method map (Shift of ShiftRightReg => ShiftRight;
                    Shift of ShiftLeftReg => ShiftLeft);
begin
        -- no delta coding required
end class BidirShiftReg;

-----------------------------------------------------------------
--
--  Objects  declaration  using  CLASS  types  and  generic  maps.
--
-----------------------------------------------------------------
--

object shift_reg1 : ShiftRightReg;
object shift_reg2 : ShiftLeftReg;
object shift_reg3 : BidirShiftReg;
object counter1 : counter;

-----------------------------------------------------------------
--
--                      Signals                       declaration.
--
-----------------------------------------------------------------
--

signal clk: bit:= '0';

begin

clk <= not clk after 10 ns;
```

```
process
        variable temp: integer := 33;
begin
        shift_reg1.reset;
        shift_reg2.reset;
        shift_reg3.reset;
        counter1.reset;
        wait for 30 ns;
        for i in 0 to 3 loop
                wait until clk= '1';
                shift_reg1.shift(clk, 10);
                counter1.count;
        end loop;

        -- temp := shift_reg1.read;
        shift_reg2.shift(clk, temp);
        shift_reg3.ShiftRight(clk, temp);
        wait until clk = '1';
        shift_reg3.ShiftLeft(clk, temp);
        wait for 20 ns;
end process;
end OO;
```

Figure 6-8. Object-Oriented Version of the Example.

The translated version of the ShiftRightReg is shown in figure 6-9.

```
package shift_reg1 is

    type vector is array (9 downto 0) of integer;
    shared variable value :  vector;

    procedure  shift( signal clk: bit ; value_in: integer );
    procedure  reset;
    procedure  read( return_2 : out  vector );

end shift_reg1;

package body shift_reg1 is

    procedure shift( signal clk:  bit ; value_in:  integer ) is
    begin
        if clk'event and clk='1' then
                value := value_in & value(9 downto 1);
        end if;
    end shift;

    procedure reset is
    begin
        value := (others => 0);
    end reset;

    procedure read( return_2 : out  vector ) is
    begin
        return_2:= value;
    end read;
end shift_reg1;
```

Figure 6-9. Translated Version of the ShiftRightReg.

The translated version of the ShifLLefReg is shown in Figure 6-10.

```
package shift_reg2 is

    type vector is array (9 downto 0) of integer;
    shared variable value :  vector;

    procedure  shift( signal clk: bit ; value_in: integer );
    procedure  reset;
    procedure  read( return_3 : out  vector );

end shift_reg2;


package body shift_reg2 is

    procedure shift( signal clk:  bit ; value_in:  integer ) is
    begin
        if clk'event and clk='1' then
                value :=  value(8 downto 0) & value_in;
        end if;
    end shift;

    procedure reset is
    begin
        value := (others => 0);
    end reset;

    procedure read( return_3 : out  vector ) is
    begin
        return_3:= value;
    end read;

end shift_reg2;
```

Figure 6-10. Translated Version of the ShiftLeftReg.

The translated version of the BidirShiftReg is shown in figure 6-11.

```
package shift_reg3 is

    type vector is array (9 downto 0) of integer;
    constant delay :  time ;
    shared variable value :  vector;

    procedure  shiftright( signal clk: bit ; value_in: integer );
    procedure  reset;
    procedure  read( return_5 : out  vector );
    procedure  shiftleft( signal clk: bit ; value_in: integer );

end shift_reg3;

package body shift_reg3 is

    constant delay :  time  :=  3 ns;
    procedure shiftright( signal clk:  bit ; value_in:  integer )
is

    begin
        if clk'event and clk='1' then
                value := value_in & value(9 downto 1);
        end if;
    end shiftright;

    procedure reset is
    begin
        value := (others => 0);
    end reset;

    procedure read( return_5 : out  vector ) is
    begin
        return_5:= value;
    end read;

    procedure shiftleft( signal clk:  bit ; value_in:  integer ) is
    begin
        if clk'event and clk='1' then
                value :=  value(8 downto 0) & value_in;
        end if;
    end shiftleft;

end shift_reg3;
```

Figure 6-11. Translated Version of the BidirShiftReg.

The translated version of the Counter is given in figure 6-12.

```
package counter1 is

    type vector is array (9 downto 0) of integer;
    shared variable value :  vector;

    procedure  count;
    procedure  shift( signal clk: bit ; value_in: integer );
    procedure  reset;
    procedure  read( return_4 : out  vector );

end counter1;

package body counter1 is

    procedure count is
    begin
       for i in 0 to 7 loop
          value(i) := value(i) + 1;
       end loop;
    end count;

    procedure shift( signal clk:  bit ; value_in:  integer ) is
    begin
        if clk'event and clk='1' then
                value := value_in & value(9 downto 1);
        end if;
    end shift;

    procedure reset is
    begin
        value := (others => 0);
    end reset;

    procedure read( return_4 : out  vector ) is
    begin
        return_4:= value;
    end read;

end counter1;
```

Figure 6-12. Translated Version of the Counter.

6.6 Conclusions

A demonstration pre-processor that runs on SUN, HP, RS6000 and LINUX workstations has been developed. This enables larger Object-Oriented VHDL descriptions to be validated in an automated way. The Object-Oriented constructs supported by the pre-processor, are limited to the capabilities of the Standard VHDL language. As a result, the pre-processor does not support polymorphism.

Further work should concentrate on creating a full compiler and simulator to enable the demonstration of all proposed enhancements

Chapter 7

OVERALL CONCLUSIONS AND RECOMMENDATION FOR FURTHER WORK

The aim of this research project was to demonstrate the benefits of applying Object-Orientation to HDL based design and to propose a new hardware description language semantics based on inheritance, encapsulation and late binding. To validate this methodology a pre-processor was created and applications were developed.

A study of design techniques used for the creation of large hardware devices was carried out. It revealed that although design methods evolved to cope with the ever-growing level of complexity of silicon chips a design methodology that focuses on aspects such as reusability and maintainability had yet to be introduced to the hardware design world. The latest design-capture methods use hardware description languages created from software languages such as C or ADA. As a result, a number of approaches for tackling reusability and maintainability in software programming languages were identified and further researched. These included the use and extension of composite data structures to create higher-level data abstraction. This approach was selected as a base for the creation of a new Objet-Oriented hardware description language semantics.

The existence and wide support for the VHDL hardware description languages in the EDA industry lead us to consider this language for the creation of an Object-Oriented hardware description language.

In order to limit the impact on existing designs and minimise the changes to the VHDL semantics, the VHDL language and its use to achieve higher levels of abstraction was studied.

Various means for achieving data abstraction with the VHDL language were found. It was determined that the ‘component’, ‘block’ and ‘package’ syntactic constructs can be suitable for implementing abstract data types when used in specific ways.

The VHDL ‘component’, although considered as the most obvious mechanism for promoting object abstraction, was also proven to be the least well suited due to its encapsulation limits. Using ‘block’ statements for building of ADTs is particularly suitable for architecture level reuse, however, our study highlighted that blocks have a very limited accessibility scope since they are not classified as VHDL design units. This led us to consider other available syntactic constructs such as the VHDL ‘package’. The package proved to outperform the other solutions when comparing the building of abstract components. We demonstrated that the use of packages allowed loose data-function binding. In addition, unlike components, packages provided a way to reach higher levels of abstraction through the separation of declarations in both the package header and the package body. We concluded that when used as abstract data types, packages represented the only substantial option for implementing an Object-Oriented-like hardware description as part of an overall object-based design methodology. Nevertheless the use of packages when considering inheritance has proven unsuccessful. Implementing Object-Orientation concepts such as encapsulation and inheritance with packages could only be achieved at the expense of code clarity. This meant compromising the concept of maintainability to the benefit of reusability.

It was demonstrated that forms of 'binding' could only be achieved in standard VHDL through the use of the configuration mechanism. As a language requirement, all VHDL design units have to be elaborated before any simulation run. Consequently, it was found that configuration only represented a static version of polymorphism and could not serve the full

purpose of a true Object-Oriented abstract data type. After study and comparison of available standard VHDL constructs, it was decided that new abstract data typing abilities were required to be added to the VHDL language to provide full Object-Orientation support.

From the study of the VHDL language and the requirement list set by the IEEE OOVHDL Study Group, we derived an extension to the language.

A new-referenced type 'class', was created to implement flexible abstract data typing. It differed from the basic VHDL types in the sense that, an object declared of a referenced type did not represent a value of type but a reference to an object containing a value. Operations on this object were defined as part of the properties of the referenced type. The class semantics were deliberately made similar to those for a package. However, from the conclusions drawn in chapter 2 it was decided not to make the class a design unit but rather a composite type. The proposed ADT supported genericity, feature mapping, virtual definition, three encapsulation levels and contract based communications support. Message passing was implemented though method calls. Methods were designed to resemble VHDL subprograms and hence supported sequential statements as well as blocking and non-blocking calls. This last feature proved to be particularly useful for modelling object activities for both the sequential and the concurrent mode. Furthermore, an extra set of attributes was added to the language to provide for better documentation.

Late binding support on the newly created ADT was added to the language, backed up by a selective multiple inheritance mechanism. Selective multiple inheritance was chosen to enable better control over encapsulation of derived features.

The new semantic for our proposed extension was evaluated by comparing it to other existing proposals. A set of benchmarks derived from the design objectives was used to that effect. Our study showed that our semantic

presented a clearer and more versatile implementation of Object-Orientation. Inheritance was restricted to single in the VISTA, Oldenburg, SUAVE, and Objective VHDL proposals and did not offer mapping capabilities. Encapsulation was only of public or private mode in the other competing proposals, hence demonstrating a weaker support for derivation operations. No contract based communication mechanisms were implemented outside our proposed extension,

In order to highlight the benefits of our proposed extension when compared the standard VHDL code, we implemented a complex case study. The chosen design was implemented using both VHDL and the proposed extension to the language. This study demonstrated an average of 22 % gain in productivity achieved with our proposal through design reuse.

A demonstration pre-processor that ran on SUN, RS6000, HP and Linux environments was created to validate the semantics and test their performance. This enabled our Object-Oriented description to be tested via conventional VHDL simulators. The generated code used shared variables and packages (ADT form) as core syntactic constructs. Consequently, the resulting VHDL code could not be synthesised. This however did not diminish the level of achievement since the main objective of this research was to demonstrate the benefits of an Object-Oriented hardware description language for addressing design maintainability and reusability issues. Late binding could not be implemented due to the existing VHDL simulator limitations.

A number of recommendations for future enhancements have been made. Key recommendations include the development of classes of components inspired from the Library Parameterisable Modules standard, to promote design reuse. An Object-Oriented VHDL to C or C++ pre-processor would have to be created to allow a more seamless integration of the proposed extension with commercial simulators and the implementation of full

polymorphism. Mechanisms such as the PLI (Programming Language Interface) or the foreign code interface would provide the required platform for such integration.

The emerging Dynamically Re-configurable Field Programmable Gate Array devices represent another area of applications. Due the nature of dynamically re-configurable FPGA, a hardware programming language providing dynamic allocation of hardware components (referred as objects) represents the ideal design approach. As a result further research is required on the creation of synthesis algorithms that will exploit scheduling and automatic resources sharing on dynamic hardware logic blocks.

Results from this research have been published at five international conference proceedings and presented to the IEEE OOVHDL study group. Furthermore, the research community has acknowledged this work though numerous references to our publications. As a longer-term goal, the recognition of the proposed extension by the IEEE Design Automation Standardisation Committee would represent the acceptance by the hardware design world of an alternative and better-suited design methodology to cope with large sub-micron VLSI chip designs. This thesis document is already being examined by the Cadence EDA research group and will be submitted to the IEEE DASC for review.

REFERENCES

Armstrong, R. M. (1994). "Uses and abuses of inheritance.", Software Engineering(January 1994), pp 19-26.

Ashenden P., Wilsey P., (1997). "SUAVE: Painless Extension for an Object-Oriented VHDL.", VIUF Fall 97 Conference, pp 60-67.

Ashenden P., Wilsey P., (1998). "SUAVE: Extending VHDL to Improve Data Modelling Support.", IEEE Design and Test of Computers 1998 , Vol.15, no 2, pp 34-44.

Ashenden P., Wilsey P., (1998.2). "SUAVE: Object-Oriented and Genericity Extensions to VHDL for High-Level Modeling" pp 34-41.

Atkinson, C. (1990). "DRAGOON: An Object-Oriented Notations supporting the reuse and distribution of ADA software." ADA Letters **X**(9): 50-59.

Bakowski, J. C. (1992). "System performance modelling with the functional scheme on VHDL.", Microprocessing, Microprogramming Conference 1992, pp 23-36.

Barbacci (1981). "Syntax and semantics of CHDLs.", IFIP Computer hardware description languages and their applications, pp 243-257.

Berger, W. N., et al. (1995). "Requirements and design objectives for an Object-Oriented extension of VHDL.", IEEE Object-Oriented VHDL Study Group (http://vhdl.org).

Booch, G. (1991). "Object-Oriented design with applications.", Benjamin/ Cummings.

Cabanis D., Medhat S. (1995). "Object-Orientation Applied to VHDL Descriptions.", VIUF Spring 95 Conference pp. 3.9-3.15.

Cabanis D., Medhat S. (1996). "Object-Orientated Extensions to VHDL: The Classification Orientation.", SIG-VHDL Spring 96, pp 9-19.

Cabanis D., Medhat S. (1996). "Classification-Orientation for VHDL: A Specification.", VIUF Spring 96 Conference pp.265-274.

Cabanis D., Medhat S. (1996). "Perspectives of Object-Oriented Technology Applied to Hardware Description Language Based Designs.", Proceedings of 3rd International Conference on Concurrent Engineering and Electronic Design Automation, Poole, UK Jan 18-19 1996, pp. 448-453.

Cabanis D. (1997) “Proposed Object-Oriented Extensions to VHDL.” Report Version 1.0, Bournemouth University available by email at dcabanis@cadence.com.

Cabanis D., Medhat S. (1997) “From Structured to Object-Oriented Design Methods” VIUF spring 1997, pp 5.11-5.22.

Cargill, T. (1991). “Controversy: The case against multiple inheritance in , Computing System **4**(1), pp 69-83.

Chu (1965). “An algol like computer design language.”, ACM 1965 pp 3.10-3.25.

Chu, H. C. (1993). “An executable specification language for mixed timing control circuits.”, CHDL and their applications, IFIP, pp 96-103.

Coad, E. Y. (1991). “Object-Oriented analysis”, 2nd Edition, Prentice Hall.

Coen-Porisini, A. M. (1991). "Specification and verification of hardware systems using the temporal logic language TRIO.", CHDL and their applications IFIP, pp 27-39.

Covnot, D. H., S. Swang (1994). "OOVHDL-An Object-Oriented VHDL.", VIUF'94, pp 126-135.

Cox, B. (1990). “Planning the software industrial revolution.”, IEEE Software, pp 25-33.

Cyre (1989). "Towards synthesis from English descriptions.", 26th ACM/IEEE Design Automation Conference, pp 5.22-5.35.

Dahl, K. N. (1966). “Simula - an algol based simulation language.” ACM 9(9), pp 671-672.

DASC, I. (1993). "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1993, IEEE Publications.

Dargupta (1981). "S*A: a language for describing computer architectures.", 5th International conference on CHDL, pp 151-159.

Douglas, D. (1994). "Object-Oriented extensions to VHDL.", VIUF'94 Fall, pp 4.43-4.51.

Duley, D. D. (1968). “A digital system design language.”, IEEE transactions on computer(September), pp 12.21-12.37.

Ecker, W. (1996). "An Object-Oriented view of structural VHDL descriptions.", VIUF'96 Spring, pp 178-190.

Eliens, A. (1994). "Object-Oriented Software Development.", Addison-Wesley.

Faura, J. M. e. a. (1997). "VHDL modelling of fast dynamic reconfiguration on novel multi-context RAM based field programmable devices.", VIUF Europe '97, Toledo, Spain, pp 34-46.

Glunz, W. (1998). "Integrating SDL VHDL for system hardware design.", CHDL and their applications IFIP 1998, pp 7.41-7.56.

Goldberg, D. R. (1989). "Smalltalk-80: the language its implementation." Reading, MA.

Gourgand, S. N. (1993). “Petri net based technology for task scheduling on , Simulations(September), pp 67-80.

Harel (1987). "STATECHARTS: a virtual formalism for complex systems.", Science of computer programming. North Holland pp 6.9-6.21.

Kartsu (1991). "UDL/I standardisation effort another approach to HDL standard.", Euro ASIC'91, pp 173-182.

Keating, P. B. (1998). "Reuse Methodology Manual for System on a Chip Design", KAP Publications.

Khoshafian (1989). "Concepts, Languages, Data Bases, User Interfaces.", New York, wiley.

Kumar, J. A., et al. (1993). “A framework for hardware/software co-design.” IEEE Computer(December) pp 9.37-9.50.

Kurup, T. A., et al. (1998). "It's the methodology stupid.", Bytek Designs, inc.

Lahitis, M. S., et al. (1991). "SADE: A graphical tool for VHDL based on system analysis.", ICCAD'91, pp 59-70.

Lanffy, L. V. (1968). "General system theory, foundation, development, applications.", New York, G. Braziller Publishing.

Leung (1979). "ADL an architecture description language for packet communication systems.", 4th International Symposium on computer hardware description languages, pp 4.93-4.104.

Lieber herr, S. K. (1981). "ZEUS: an hardware description language.", 5th international conference on CHDL, pp 3.59-3.71.

IEEE: Patern Recognition and Image Processing (1982). IEEE Publications ISBN 9993772313

Mammen, W. T. (1994). "Object-Oriented macro-modelling of analogue devices.", CEEDA'94, pp 342-354.

Meyer, B. (1992). "Eiffel: The language", Prentice, Hall.

Mills, M. (1993). "Programming Enhancements to the very high speed integrated circuit hardware description language", Wright Laboratory, can be obtained at http://vhdl.org.

Morison, N. P. (1985). "The design rational ELLA, a hardware description language.", 7th international conference on CHDL, pp 4.45-4.57.

Muller, W. (1990). "ODICE: Object-Oriented hardware descriptions in CAD environment.", CHDL and their applications, IFIP, pp 2.12-2.26.

Nelson, G. M. (1992). “Concurrent Object-Oriented programming in classic , ACM Letters XII(5), pp 77-81.

Nixon, S. W. (1986). "A micro architecture description language for retargetting firmware tools.", 19th Annual workshop on programming pp 231-243.

Oczko, A. (1990). "Hardware design with VHDL at a very high level abstraction.", European conference on VHDL methods, pp 127-136.

OOVHDL Study Group, O. S. (1995). "Participation in the definition of needs and requirements and analysis of existing proposals in the definition of Object-Oriented extensions to VHDL", IEEE DASC can be obtained at http://vhdl.org.

Pawlak, J. J. (1981). "MODLAND a language for multi-level description and modelling of digital systems.", 5th international conference on CHDL 1981, pp 6.12-6.24.

Pawlak, A. (1987). "Modern Object-Oriented programming language as an HDL." CHDLs and their applications, IFIP pp 4.45-4.57.

Perry, D. (1992). "Applying Object-Oriented techniques to VHDL.", VIUF'92 Spring pp 191- 200.

Ramesh, C. (1994). "Object-Orienting VHDL for component modelling." VIUF'94 Fall, pp 198-211.

Radetzki, M , Putzke W. (1997) "Objective VHDL Language Definition", OFFIS Research Institute, Oldenburg, Germany, REQUEST Deliverable 2.1 A (http://eis.informatik.uni-oldenburg.de/research/objective_vhdl.shtml.

Radetzki M, Putzke W, Nebel W, Maginot S, Berge J-M, Targant (1997.2) A. "VHDL Language Extensions to Support Abstraction and Reuse.", Proceedings of Workshop on Libraries, Component Modelling and Quality Assurance , Toledo, Spain 1997 pp 17-30.

Rosenberg, J. (1994). Hardware acceleration using cache logic FPGAs. Silicon Design Show. can be obtained at http://www.atmel.com.

Seidewitz, E. (1991). “Object-Oriented programming through type extension in ADA (X.)”, ADA Letters XI(2) pp 86-97.

Shelor, C. (1994). “New philosophy aids shift from schematic-based to HDL-Publication, pp 119-124.

Shumacher, W. N. (1995). "Inheritance concept for signals in Object-Oriented Extensions to VHDL.", Euro-DAC'95, IEEE Computer Society Press pp 2.12-2.23.

Shumacher, W. N., W. Putzke, M. Wilmes (1996). "Applying Object-Oriented techniques to hardware modelling-A case study." VHDL-Forum for CAD in Europe / Sig VHDL spring pp 211-223.

Smith, R. G. (1986). “Technology transfer between, VLSI design and software engineering: CAD tools and design methodologies.” Proceedings of the IEEE 74(6) pp 875-885.

Taft, S. T. (1993). "ADA 9X: From abstraction-oriented to object-oriented.", OOPSLA'93, pp 64-75.

Takeuchi, A. (1981). "Object-Oriented description environment for computer hardware." Computer hardware description languages and their applications, IFIP, pp 3.21-3.35.

Vahid, S. N. (1991). "SpecChart: A language for system level synthesis.", CHDL and their applications, IFIP pp 6.42-6.54.

Vasilko, D. C. (1999). "A technique for modelling dynamic reconfiguration with improved simulation accuracy." IEICE Transactions, pp 77-90.

Vista OOVHDL Language reference (1994.2), IEEE RASSP Technical report, can be obtained at http://www.vhdl.org.

Vista (1994). "OOVHDL: Object-Oriented Extensions for VHDL", IEEE DAC 1994 pp 33-41.

Waldo, J. (1991). “Controversy: The case for multiple inheritance in C++.” Computing systems **4**(2) pp 157-171.

Willis, J. (1994). "A proposal for minimally extend VHDL to achieve data encapsulation, late binding and inheritance." VIUF Fall, pp 118-131.

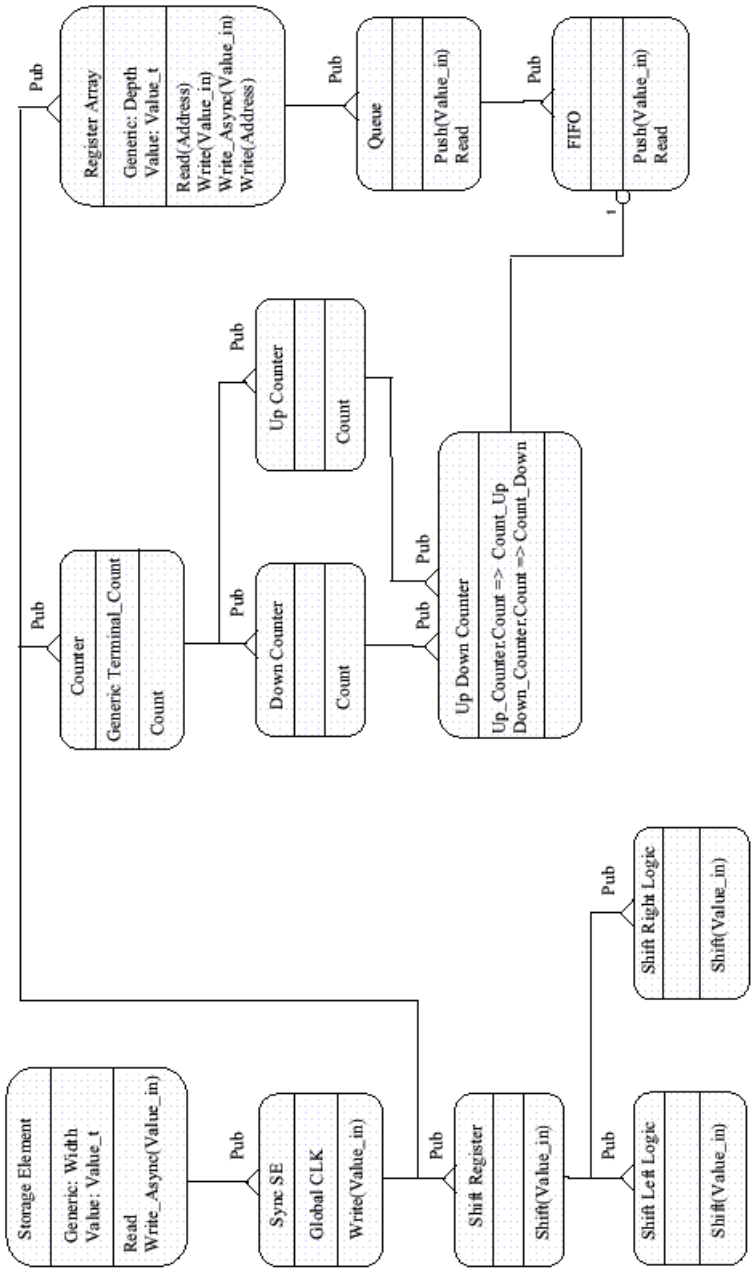
Wirth, N. (1998). “Hardware Compilation Translating Programs into Circuits.” IEEE Computer(June 1998), pp 25-31.

Xilinx (1997). "Xilinx XC6200 field programmable gate array, advanced product informations version 1.8", can be obtained at http://www.xilinx.com.

Zippelius, M. G. (1992). "An Object-Oriented extension of VHDL." VHDL Forum Europe Spring, pp 212-224.

APPENDIX A: EDGE FILTER SUPPORTING CLASSES

Appendix A-1. Edge Filter Class Structure



Appendix A-2. Class Structure Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

package STD_CLASS is

   type STORAGE_ELEMENT is class
      generic (DEPTH : integer := 1);
      subtype VALUE_T is std_logic_vector (DEPTH-1 downto
0);
   begin
      method READ return VALUE_T;
      method WRITE(VALUE_IN : VALUE_T);
   end class STORAGE_ELEMENT;

   type SYNC_SE is class use (public STORAGE_ELEMENT)
      interface (CLK : std_logic);
      feature map (WRITE => WRITE_ASYNC);
   begin
      method WRITE(VALUE_IN : VALUE_T);
   end class SYN_SE;

   type SHIF_REGISTER is class use (public SYNC_SE)
   begin
      method SHIFT(VALUE_IN : std_logic);
   end class SHIFT_REGISTER;

   type SHIFT_RIGHT_LOGIC is class use (public
SHIFT_REGISTER)
   begin
      method SHIFT(VALUE_IN : std_logic);
   end class SHIFT_RIGHT_LOGIC;

   type SHIFT_LEFT_LOGIC is class use public SHIFT_REGISTER
   begin
      method SHIFT(VALUE_IN : std_logic);
   end class SHIFT_LEFT_LOGIC;

   type COUNTER is class use (public SYNC_SE)
      generic (TERMINAL_COUNT : VALUE_T := (others => '1'));
   begin
      method COUNT;
   end class COUNTER;

   type UP_COUNTER is class use (public COUNTER)
   begin
      method COUNT;
   end class UP_COUNTER;

   type DOWN_COUNTER is class use(public COUNTER)
   begin
      method COUNT;
```

```
   end class DOWN_COUNTER;

   type UP_DOWN_COUNTER is class use (public UP_COUNTER,
                                      public DOWN_COUNTER)
      feature map (COUNT of UP_COUNTER => COUNT_UP,
                   COUNT of DOWN_COUNTER => COUNT_DOWN);
   begin
   end class UP_DOWN_COUNTER;

   type REGISTER_ARRAY is class use (PRIVATE SYNC_SE)
      generic (WIDTH : integer := 1);
   begin
      method WRITE_ASYNC(VALUE_IN: VALUE_T);
      method WRITE(VALUE_IN: VALUE_T);
      method WRITE(ADDRESS: integer; VALUE_IN: VALUE_T);
      method READ(ADDRESS: integer) return VALUE_T;
   end class REGISTER_ARRAY;

   type QUEUE is class use (public REGISTER_ARRAY)
   begin
      method PUSH(VALUE_IN : VALUE_T);
      method READ return VALUE_T;
   end class;

   type FIFO is class use (public QUEUE)
   begin
      method PUSH(VALUE_IN : VALUE_T);
      method READ return VALUE_T;
   end class FIFO;

end package STD_CLASS;


package body STD_CLASS is

   type STORAGE_ELEMENT is class
      instance variable VALUE : VALUE_T := (others => '0');
   begin
      method READ return VALUE_T is
      begin
         return VALUE;
      end method READ;

      method WRITE(VALUE_IN : VALUE_T) is
      begin
         VALUE := VALUE_IN;
      end method WRITE;
   end class STORAGE_ELMENT;

   type SYNC_SE is class
   begin
      method WRITE(VALUE_IN : VALUE_T) is
      begin
         if CLK'event and CLK = '1' then
            VALUE := VALUE_IN;
```

```
         end if;
   end class SYN_SE;

   type SHIFT_RIGHT_LOGIC is class
   begin
      method SHIFT(VALUE_IN : std_logic) is
      begin
         this.WRITE(VALUE_IN & VALUE(DEPTH-1 downto 1));
      end method SHIFT;
   end class SHIFT_RIGHT_LOGIC;

   type SHIFT_LEFT_LOGIC is
   begin
      method SHIFT(VALUE_IN : std_logic) is
      begin
         this.WRITE(VALUE(DEPTH-2 downto 0) & VALUE_IN);
      end method SHIFT;
   end class SHIFT_LEFT_LOGIC;

   type UP_COUNTER is class
   begin
      method COUNT is
      begin
         if VALUE = TERMINAL_COUNT then
            this.WRITE(to_std_logic_vector(0,
VALUE'length));
         else
            this.WRITE(std_logic_vector(unsigned(VALUE) +
1));
         end if;
      end method COUNT;
   end class UP_COUNTER;

   type DOWN_COUNTER is class
   begin
      method COUNT is
      begin
         if VALUE = 0 then
            this.WRITE(to_std_logic_vector(TERMINAL_COUNT,
                                           VALUE'length));
         else
            this.WRITE(std_logic_vector(unsigned(VALUE) -
1));
         end if;
      end method COUNT;
   end class DOWN_COUNTER;

   type REGISTER_ARRAY is class
      type REG_ARRAY_T is array (0 to DEPTH-1) of VALUE_T;
      instance variable VALUE : REG_ARRAY_T;
   begin
      method  WRITE_ASYNC(VALUE_IN : VALUE_T) is
      begin
         for I in VALUE'range loop
            VALUE(I) := VALUE_IN;
```

```
      end loop;
   end method WRITE_ASYNC;

   method WRITE(VALUE_IN : VALUE_T) is
   begin
      if CLK'event and CLK = '1' then
         for I in VALUE'range loop
            VALUE(I) := VALUE_IN;
         end loop;
      end if;
   end method WRITE;

   method WRITE(ADDRESS: integer; VALUE_IN: VALUE_T);
   begin
      if CLK'event and CLK = '1' then
         VALUE(ADDRESS) := VALUE_IN;
      end if;
   end method WRITE;

   method READ(ADDRESS: integer) return VALUE_T is
   begin
      return VALUE(ADDRESS);
   end method READ;

end class REGISTER_ARRAY;

type FIFO is class
   public instance variable IS_FULL: boolean := FALSE;
   public instance variable IS_EMPTY: boolean := TRUE;
   object READ_POINTER: UP_DOWN_COUNTER
         generic map(TERMINAL_COUNT => DEPTH-1);
begin
   method PUSH(VALUE_IN : VALUE_T) is
      variable FIRST : BOOLEAN := TRUE;
   begin
      for I in DEPTH-1 downto 0 loop
         this -> WRITE(I, VALUE(I-1), CLK);
      end loop;
      this -> WRITE(0, VALUE_IN, CLK);
      if not FIRST then
         if READ_POINTER.READ /= DEPTH-1 then
            READ_POINTER.COUNT_UP;
            IS_FULL := FALSE;
         else
            IS_FULL := TRUE;
         end if;
      else
         FIRST := FALSE;
         IS_FULL := FALSE;
      end if;
   end method PUSH;

   method READ return VALUE_T is
   begin
      return VALUE(READ_POINTER.READ);
```

```
         if READ_POINTER.READ /= 0 then
            READ_POINTER.COUNT_DOWN;
            IS_EMPTY := FALSE;
         else
            IS_EMPTY := TRUE;
         end if;
      end method READ;
   end class FIFO;

end package STD_CLASS;
```

APPENDIX B: EDGE FILTER CODE

Appendix B-1. Mult Mask Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: Dual mask, dual linestore controller
--
------------------------------------------------------------
----------
--
-- File name: mult_mask.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
-------
------------------------------------------------------------
---

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.all;
USE WORK.std_class.ALL;

ENTITY mult_mask IS
  PORT(scene   : IN slv8;
       ls,fs   : IN STD_LOGIC;
       reset   : IN STD_LOGIC;
       clk     : IN STD_LOGIC;
       clk4    : IN STD_LOGIC;
       wr_pulse  : IN STD_LOGIC;
       new_frame : OUT STD_LOGIC;
       ns_result  : OUT slv15;
       we_result  : OUT slv15;
       ls_out  : OUT slv8);
END mult_mask;

ARCHITECTURE rtl OF mult_mask IS

   -- mult mask states
constant sync_fs : std_logic_vector(11 downto 0):=
 "000000000001";
   constant sync_ls : std_logic_vector(11 downto 0):=
 "000000000010";
   constant last_pxl : std_logic_vector(11 downto 0):=
"000000000100";
   constant first_line : std_logic_vector(11 downto 0):=
"000000001000";
   constant end_first : std_logic_vector(11 downto 0):=
"000000010000";
```

```
   constant sec_line : std_logic_vector(11 downto 0) :=
"000000100000";
   constant end_line : std_logic_vector(11 downto 0):=
"000001000000";
   constant blank : std_logic_vector(11 downto 0):=
"000010000000";
   constant valid : std_logic_vector(11 downto 0):=
"000100000000";
   constant frame_sync : std_logic_vector(11 downto 0):=
 "001000000000";
   constant start_last : std_logic_vector(11 downto 0):=
"010000000000";
   constant last_line : std_logic_vector(11 downto 0):=
"100000000000";

   -- mult mask state variables
   signal L_SYNC, F_SYNC: std_logic_vector(1 downto 0);
   SIGNAL current_state,
          next_state : std_logic_vector(11 downto 0);
   SIGNAL next_new_frame, new_frame_sync:STD_LOGIC;
   SIGNAL scene_reg :image_data;
   SIGNAL top_data, middle_data:image_data;
   SIGNAL top_data_reg, middle_data_reg:image_data;
   SIGNAL blankop, blankop_cap :BOOLEAN;
   SIGNAL ns_mask_out, we_mask_out:filt_data;
   SIGNAL ns_filt, we_filt:abs_filt;
   object LS_SHIFT, FS_SHIFT : SHIFT_LEFT_LOGIC
                                    GENERIC MAP(DEPTH => 2)
                                    INTERFACE MAP (CLK =>
CLK);
   COMPONENT nsmask
     PORT(bottom_in : IN image_data;
          middle_in : IN image_data;
          top_in    : IN image_data;
          reset     : IN STD_LOGIC;
          clk       : IN STD_LOGIC;
          clk4      : IN STD_LOGIC;
          mask_out  : OUT filt_data);
   END COMPONENT;
   COMPONENT wemask
     PORT(bottom_in : IN image_data;
          middle_in : IN image_data;
          top_in    : IN image_data;
          reset     : IN STD_LOGIC;
          clk       : IN STD_LOGIC;
          clk4      : IN STD_LOGIC;
          mask_out  : OUT filt_data);
   END COMPONENT;
   COMPONENT linestore
     PORT (wr_data  : IN image_data;
           rd_data  : OUT image_data;
           reset    : IN STD_LOGIC;
           wr_pulse : IN STD_LOGIC;
           ls_clk   : IN STD_LOGIC;
           clk4     : IN STD_LOGIC);
```

```
   END COMPONENT;
BEGIN
   midrow:linestore
          PORT MAP (wr_data => scene_reg,
                    rd_data => middle_data,
                    reset => reset, wr_pulse => wr_pulse,
                    ls_clk => clk, clk4 => clk4);

   toprow:linestore
          PORT MAP (wr_data => middle_data_reg,
                    rd_data => top_data, reset => reset,
                    wr_pulse => wr_pulse,
                    ls_clk => clk, clk4 => clk4);

   nthsth:nsmask
          PORT MAP (bottom_in => scene_reg,
                    middle_in => middle_data_reg,
                    top_in => top_data_reg,
                    reset => reset, clk => clk, clk4 =>
clk4,
                    mask_out => ns_mask_out);
   wstest:wemask
          PORT MAP (bottom_in => scene_reg,
                    middle_in => middle_data_reg,
                    top_in => top_data_reg,
                    reset => reset, clk => clk,
                    clk4 => clk4, mask_out => we_mask_out);

  -- complete mask operation by normalising absolute mask
  -- result

  ns_complete:
  PROCESS (ns_mask_out, blankop_cap)
    VARIABLE ns_filt : slv16;
  BEGIN
    ns_filt := conv_std_logic_vector(ns_mask_out, 16);
    IF ns_filt(15) = '1' THEN
      ns_filt := unsigned(NOT(ns_filt)) + '1';
    END IF;
    ns_filt := "00" & ns_filt(15 downto 2);
    IF blankop_cap THEN
      ns_result <= (OTHERS => '0');
    ELSE
      ns_result <= ns_filt(14 downto 0);
    END IF;
  END PROCESS ns_complete;

  we_complete:
  PROCESS (we_mask_out, blankop_cap)
    VARIABLE we_filt : slv16;
  BEGIN
    we_filt := conv_std_logic_vector(we_mask_out, 16);
    IF we_filt(15) = '1' THEN
      we_filt :=  unsigned(NOT(we_filt)) + '1';
    END IF;
```

```
    we_filt := "00" & we_filt(15 downto 2);
    IF blankop_cap THEN
      we_result <= (OTHERS => '0');
    ELSE
      we_result <= we_filt(14 downto 0);
    END IF;
  END PROCESS we_complete;

  -- write linestore o/p to external port for RAM test read
  -- access
  ls_out <= conv_std_logic_vector(top_data, 8);

  state_reg:
  PROCESS(CLK, RESET)
  BEGIN
    IF reset = '1' THEN
      current_state <= sync_fs;
      blankop_cap <= TRUE;
    ELSIF clk'event AND clk = '1' THEN
      current_state <= next_state;
      blankop_cap <= blankop;
    END IF;
  END PROCESS state_reg;

  state_assign:
  PROCESS (current_state, fs_sync, ls_sync)
  BEGIN
    next_state <= current_state;
    blankop<=true;               -- default is input is
blanked
    next_new_frame <= '0';       -- default is not new frame
      IF current_state = sync_fs THEN
        IF fs_sync = '0' THEN
          next_state <= sync_ls;
        END IF;
      ELSIF current_state = sync_ls THEN
        IF ls_sync = '0' THEN
          next_state <= last_pxl;
          next_new_frame <= '1';
        END IF;
      ELSIF current_state = last_pxl THEN
        IF ls_sync = '1' THEN
          next_state <= first_line;
        END IF;
      ELSIF current_state = first_line THEN
        IF ls_sync = '0' THEN
          next_state <= end_first;
        END IF;
      ELSIF current_state = end_first THEN
        IF ls_sync = '1' THEN
          next_state <= sec_line;
        END IF;
      ELSIF current_state = sec_line THEN
        IF ls_sync = '0' THEN
          next_state <= end_line;
```

```
      END IF;
    ELSIF current_state = end_line THEN
      IF ls_sync = '1' THEN
        next_state <= blank;
      END IF;
    ELSIF current_state = blank THEN
      next_state <= valid;
    ELSIF current_state = valid THEN
      blankop<=false;
      IF (ls_sync = '0' AND fs_sync = '0') THEN
        next_state <= frame_sync;
      ELSIF (ls_sync = '0') THEN
        next_state <= end_line;
      END IF;
    ELSIF current_state = frame_sync THEN
      IF(ls_sync = '1' AND fs_sync = '1') THEN
        next_state <= start_last;
      END IF;
    ELSIF current_state = start_last THEN
      next_state <= last_line;
    ELSIF current_state = last_line THEN
      blankop<=false;
      IF ls_sync = '0' THEN
        next_state <= last_pxl;
        next_new_frame <= '1';
      END IF;
  END if;
END PROCESS state_assign;

data_reg:
PROCESS(clk, reset)
BEGIN
  IF reset = '1' THEN
    scene_reg <= 0;
    middle_data_reg <= 0;
    top_data_reg <= 0;
    LS_SHIFT.WRITE_ASYNC("11");
    FS_SHIFT.WRITE_ASYNC("11");
    new_frame <= '0';
    new_frame_sync <= '0';
  ELSIF clk'event AND clk = '1' THEN
    scene_reg <= conv_integer(unsigned(scene));
    middle_data_reg <= middle_data;
    top_data_reg <= top_data;
    LS_SHIFT.SHIFT(LS);
    FS_SHIFT.SHIFT(FS);
    new_frame_sync <= next_new_frame;
    new_frame <= new_frame_sync;
  END IF;
END PROCESS data_reg;

L_SYNC <= LS_SHIFT.READ;
F_SYNC <= FS_SHIFT.READ;
LS_SYNC <= L_SYNC(1);
FS_SYNC <= F_SYNC(1);
```

```
END rtl;
```

Appendix B-2. NS & WE Mask Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: customised nsmask
--
------------------------------------------------------------
----------
--
-- File name: nsmask.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
   USE flowlab.mask_types.ALL;

LIBRARY IEEE;
   USE IEEE.STD_LOGIC_1164.ALL;

USE WORK.std_class.ALL;

   ENTITY nsmask IS
      PORT(bottom_in : IN image_data;
           middle_in : IN image_data;
           top_in    : IN image_data;
           reset     : IN STD_LOGIC;
           clk       : IN STD_LOGIC;
           clk4      : IN STD_LOGIC;
           mask_out  : OUT filt_data);
   END nsmask;

   ARCHITECTURE rtl OF nsmask IS
      TYPE mask_row IS ARRAY (0 TO mask_size) OF image_data;
      TYPE FIFO_ARRAY_T IS ARRAY (0 TO 2) OF FIFO;
      OBJECT TOP, MID, BOT: FIFO
                            GENERIC MAP(DEPTH => 3, WIDTH =>
8)
                            INTERFACE map(CLK => CLK);
      OBJECT FIFO_ARRAY : FIFO_ARRAY_T;
      SIGNAL prelim, prelim_reg :filt_data;
      TYPE calc_state_t IS (init, top, middle, bottom,
pause);
      SIGNAL calc_state:calc_state_t;
   BEGIN
      FIFO_ARRAY == (TOP, MID, BOT);

      maskshift:
      -- shift mask scene data and read in new values
      -- from top and middle linestores, and scene data
input
```

```
      PROCESS(clk, reset)
      BEGIN
         IF reset = '1' THEN
            FIFO_ARRAY.ALL.WRITE_ASYNC(CONV_STD_LOGIC_VECTOR
                                       (0,8));
            mask_out<=0;
         ELSIF (clk'event AND clk = '1') THEN
            TOP->PUSH(TOP_IN);
            MID->PUSH(MIDDLE_IN);
            BOT->PUSH(MIDDLE_IN);
         -- output mask calculation result
            mask_out<=prelim;
         END IF;
      END PROCESS maskshift;

      calcseq:
      -- state machine to calculate mask arithmetic row by
row
      -- prelim to store running total
      PROCESS (clk4, reset)
      BEGIN
         IF reset = '1' THEN
            calc_state<=init;
            prelim_reg<=0;
         ELSIF (clk4'event AND clk4 = '1') THEN
            prelim_reg<=prelim;
            CASE calc_state IS
               WHEN init =>
                  calc_state<=pause;
               WHEN pause =>
                  calc_state<=top;
                  prelim_reg<=0;
               WHEN top =>
                  calc_state<=middle;
               WHEN middle =>
                  calc_state<=bottom;
               WHEN bottom =>
                  calc_state<=pause;
               WHEN OTHERS =>
                  calc_state <= init;
            END CASE;
         END IF;
      END PROCESS calcseq;

   calcomb:
      PROCESS(calc_state, FIFO_ARRAY, prelim_reg)
        VARIABLE left_v, right_v : INTEGER RANGE 0 TO
(((2**image_width)-1)*2);
      BEGIN
         left_v := 0;
         right_v := 0;
         CASE calc_state IS
            WHEN top =>
               left_v := BOT.READ(2);
               right_v := TOP.READ(2);
```

```
            WHEN middle =>
               left_v := BOT.READ(0);
               right_v := TOP.READ(0);
            WHEN bottom =>
               left_v := 2*BOT.READ(1);
               right_v := 2*TOP.READ(1);
            WHEN OTHERS =>
               left_v := 0;
               right_v := 0;
         END CASE;
         prelim <= (left_v - right_v) + prelim_reg;
      END PROCESS calcomb;
   END rtl;


------------------------------------------------------------
----------
--
-- Description: customised wemask
--
------------------------------------------------------------
----------
--
-- File name: wemask.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
   USE flowlab.mask_types.ALL;

LIBRARY IEEE;
   USE IEEE.STD_LOGIC_1164.ALL;

USE WORK.std_class.ALL;

   ENTITY wemask IS
      PORT(bottom_in : IN image_data;
           middle_in : IN image_data;
           top_in    : IN image_data;
           reset     : IN STD_LOGIC;
           clk       : IN STD_LOGIC;
           clk4      : IN STD_LOGIC;
           mask_out  : OUT filt_data);
   END wemask;

   ARCHITECTURE rtl OF wemask IS
      TYPE mask_row IS ARRAY (0 TO mask_size) OF image_data;
      TYPE FIFO_ARRAY_T IS ARRAY (0 TO 2) OF FIFO;
      OBJECT TOP, MID, BOT : FIFO GENERIC MAP(DEPTH => 3,
                                              WIDTH => 8)
                                    INTERFACE MAP(CLK => CLK);
      OBJECT FIFO_ARRAY : FIFO_ARRAY_T;
```

```
      SIGNAL prelim, prelim_reg :filt_data;
      TYPE calc_state_t IS (init, top, middle, bottom,
pause);
      SIGNAL calc_state:calc_state_t;

   BEGIN
      FIFO_ARRAY == (TOP, MID, BOT);

      maskshift:
      -- shift mask scene data and read in new values
      -- from top and middle linestores, and scene data
input
      PROCESS(RESET, CLK)
      BEGIN
         IF reset = '1' THEN
            FIFO_ARRAY.ALL.WRITE_ASYNC(CONV_STD_LOGIC_VECTOR
                                       (0,8));
            mask_out<=0;
         ELSIF (clk'event AND clk = '1') THEN
            TOP->PUSH(TOP_IN);
            MID->PUSH(MIDDLE_IN);
            BOT->PUSH(MIDDLE_IN);
                     -- output mask calculation result
            mask_out<=prelim;
         END IF;
      END PROCESS maskshift;

      calcseq:
      -- state machine to calculate mask arithmetic row by
row
      -- prelim to store running total
      PROCESS (clk4, reset)
      BEGIN
         IF reset = '1' THEN
            calc_state<=init;
            prelim_reg<=0;
         ELSIF (clk4'event AND clk4 = '1') THEN
            prelim_reg<=prelim;
            CASE calc_state IS
               WHEN init =>
                  calc_state<=pause;
               WHEN pause =>
                  calc_state<=top;
                  prelim_reg<=0;
               WHEN top =>
                  calc_state<=middle;
               WHEN middle =>
                  calc_state<=bottom;
               WHEN bottom =>
                  calc_state<=pause;
               WHEN OTHERS =>
                  calc_state <= init;
            END CASE;
         END IF;
      END PROCESS calcseq;
```

```
   calcomb:
   -- of clk4
   PROCESS(calc_state, FIFO_ARRAY, prelim_reg)
   BEGIN
      CASE calc_state IS
         WHEN top =>
            prelim <= (BOT.READ(2) - TOP.READ(2))
                       + prelim_reg;
         WHEN middle =>
            prelim <= (BOT.READ(0) - TOP.READ(0))
                       + prelim_reg;
         WHEN bottom =>
            prelim <= (2*BOT.READ(1) - 2*TOP.READ(1))
                       + prelim_reg;
         WHEN OTHERS =>
            prelim <=prelim_reg;
      END CASE;
   END PROCESS calcomb;
END rtl;
```

Appendix B-3. Line Store Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: 8 bit, 32 word FIFO, based on dual-port RAM.
--
------------------------------------------------------------
----------
--
-- File name: linestore.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

LIBRARY flowlab;
USE flowlab.mask_types.ALL;
USE WORK.std_class.ALL;

   ENTITY linestore IS
      PORT (wr_data  : IN image_data;
            rd_data  : OUT image_data;
            reset    : IN STD_LOGIC;
            wr_pulse : IN STD_LOGIC;
            ls_clk   : IN STD_LOGIC;
            clk4     : IN STD_LOGIC);
   END linestore;

   ARCHITECTURE struct OF linestore IS
      SUBTYPE line_pointer IS INTEGER RANGE 0 TO
line_length;
      SIGNAL rd_slv, wr_slv:STD_LOGIC_VECTOR(image_width – 1
             DOWNTO 0) := (others =>'0');
      SIGNAL logic_0 : STD_LOGIC;
      SIGNAL NOT_LS_CLK : STD_LOGIC;
      COMPONENT generic_ram
         port (
              d         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
              address   : IN  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
              we, clk   : IN  STD_LOGIC ;
              q         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
              );
      END COMPONENT;
      OBJECT RD_PTR : UP_COUNTER
                        GENERIC MAP(DEPTH =>
LINE_DATA_WIDTH)
                                INTERFACE MAP(CLK =>
LS_CLK);
```

```
      OBJECT WR_PTR : UP_COUNTER
                           GENERIC MAP(DEPTH =>
LINE_DATA_WIDTH)
                           INTERFACE MAP(CLK => NOT_LS_CLK);
      TYPE ADDR_PTR_T : IS ARRAY (0 TO 1) OF COUNTER;
      OBJECT ADDR_PTR : ADDR_PTR_T;
   BEGIN
      ADDR_PTR == (RD_PTR, WR_PTR);

      logic_0 <='0';

      memory:generic_ram
            PORT MAP(q => rd_slv, d => wr_slv,
                     address => int_ad,
                     we => wr_pulse, clk => clk4);

      rd_data <= conv_integer(unsigned(rd_slv));

      wr_slv <= conv_std_logic_vector(wr_data, image_width);

      NOT_LS_CLK <= NOT LS_CLK;

      read_write_pointers:
      PROCESS(LS_CLK, NOT_LS_CLK, RESET)
      BEGIN
         IF RESET = '1' THEN
            ADDR_PTR.ALL.WRITE(CONV_STD_LOGIC_VECTOR(0,
                               LINE_DATA_WIDTH));
         ELSE
            ADDR_PTR.ALL.COUNT;
         END IF;
      END PROCESS;

   int_ad <= WR_PTR.READ WHEN ls_clk = '1' ELSE RD_PTR.READ;
   END struct;
```

Appendix B-4. Threshold Mult Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: Threshold, multiplex and average
calculation..
--
--
------------------------------------------------------------
----------
--
-- File name: thresh_mux.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

LIBRARY flowlab;
USE flowlab.mask_types.ALL;
USE WORK.std_class.ALL;

ENTITY thresh_mux IS
  PORT(ns_result : IN slv15;
       we_result : IN slv15;
       new_frame : IN STD_LOGIC;
       clk       : IN STD_LOGIC;
       reset     : IN STD_LOGIC;
       threshold : IN slv15;
       data_sel  : IN slv2;
       frame_sync: OUT STD_LOGIC;
       average   : OUT slv15;
       mux_bit   : OUT STD_LOGIC
       );
END thresh_mux;

ARCHITECTURE rtl OF thresh_mux IS
  SIGNAL thresh_reg : slv15;
  SIGNAL average_i, average_reg : abs_filt;
  SIGNAL thresh_we, thresh_ns:STD_LOGIC;
  SIGNAL data_sel_reg:slv2;
  CONSTANT zero:slv15:=(OTHERS => '0');
BEGIN
  averager:
  PROCESS (ns_result, we_result, data_sel_reg, reset,
           average_reg, new_frame)
  BEGIN
    IF reset = '1' THEN
      average_i <= 0;
```

```
    ELSIF new_frame = '1' THEN
      average_i <= 0;
    ELSE
      average_i <= average_reg;
      CASE data_sel_reg IS
        WHEN nsbinary =>
          IF ns_result /= zero THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(ns_result)) / 2);
          END IF;
        WHEN webinary =>
          IF we_result /= zero THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(we_result)) / 2);
          END IF;
        WHEN OTHERS =>
          IF (ns_result /= zero)
              AND (ns_result > we_result) THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(ns_result)) / 2);
          ELSIF we_result /= zero THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(we_result)) / 2);
          END IF;
      END CASE;
    END IF;
  END PROCESS;

  -- threshold ns result
  thresh_ns <= '1' WHEN ns_result > thresh_reg
                   ELSE '0';

  -- threshold we result
  thresh_we <= '1' WHEN we_result > thresh_reg
                    ELSE '0';

cont_regs:
  -- update threshold and data_sel on new_frame
  PROCESS (reset, clk)
  BEGIN
    IF reset = '1' THEN
      thresh_reg <= zero;
      data_sel_reg <= "00";
    ELSIF clk'event AND clk = '1' THEN
      IF new_frame = '1' THEN
        thresh_reg <= threshold;
        data_sel_reg <= data_sel;
      ELSE
        thresh_reg <= thresh_reg;
        data_sel_reg <= data_sel_reg;
      END IF;
    END IF;
 END PROCESS;

  -- register mult_mask result outputs
```

```
thresh_regs:
  PROCESS
  BEGIN
    WAIT UNTIL clk'event AND clk='1';
    IF reset = '1' THEN
      average_reg <= 0;
      frame_sync <= '0';
      mux_bit <= '0';
    ELSE
      average_reg <= average_i;
      frame_sync <= new_frame;
      -- select bit value to output
      CASE data_sel_reg IS
        WHEN nsbinary =>
          mux_bit <= thresh_ns;
        WHEN webinary =>
          mux_bit <= thresh_we;
        WHEN nsorwe =>
          mux_bit <= thresh_ns OR thresh_we;
        WHEN OTHERS =>
          mux_bit <= thresh_ns AND thresh_we;
      END CASE;
    END IF;
  END PROCESS;

  -- write average output
  average <= conv_std_logic_vector(average_reg, 15);
END rtl;
```

Appendix B-5. Interface Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: Asynchronous processor interface model.
--              Part of edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: interface.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
USE flowlab.mask_types.all;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY interface IS
  PORT(clk       : IN STD_LOGIC;
       reset     : IN STD_LOGIC;
       address   : IN slv3;
       dataio    : INOUT slv16;
       wr_nrd    : IN STD_LOGIC;
       enable    : IN STD_LOGIC;
       average   : IN slv15;
       ack       : OUT STD_LOGIC;
       threshold : OUT slv15;
       data_sel  : OUT slv2
       );
END interface;

ARCHITECTURE rtl OF interface IS

  SIGNAL addr_reg, addr1, addr2 : slv3;
  SIGNAL data_reg, data1, data2 : slv16;
  SIGNAL wnr_reg, wnr1, wnr2  : std_logic;
  SIGNAL enable1, enable2  : std_logic;
  SIGNAL aver1, aver2, thresh_reg, threshold_i : slv15;
  SIGNAL datasel_reg, data_sel_i : slv2;
  SIGNAL data_in : slv16;
  SIGNAL tri_bus : STD_LOGIC;

BEGIN

  -- intermediate signals used to read output ports
  threshold <= threshold_i;
```

```
  data_sel <= data_sel_i;


  async:
  PROCESS (reset, enable)
  BEGIN
   -- async reset
   IF reset = '1' THEN
     addr_reg <= (OTHERS => '0');
     data_reg <= (OTHERS => '0');
     wnr_reg <= '0';
   ELSIF (enable'event AND enable='0') THEN
   -- register input on falling edge of enable
     addr_reg <= address;
     data_reg <= data_in;
     wnr_reg <= wr_nrd;
   END IF;
  END PROCESS async;


  if_rw:
  PROCESS(wnr2, data2, addr2, enable2, thresh_reg,
datasel_reg)
  BEGIN

    ack <= '1';                   -- default is
unacknowledged
    tri_bus <= '1';               -- default is tristated
bus
    threshold_i <= thresh_reg;
    data_sel_i <= datasel_reg;

    IF enable2 = '0' THEN
      -- bus activity - inputs valid
      IF wnr2 = '1' THEN
        -- write cycle
        CASE addr2 IS
          WHEN "010" =>
            -- threshold data
            threshold_i <= data2(14 DOWNTO 0);
          WHEN "001" =>
            -- data select,
            data_sel_i <= data2(1 DOWNTO 0);
          WHEN OTHERS =>
            null;
        END CASE;
        ack <= '0'; -- acknowledge write
      ELSE
        -- read cycle
        IF addr2 = "100" THEN
          -- read average result value
          -- un-tristate buffers
          tri_bus <= '0';
        END IF;
        ack <= '0'; -- acknowledge read cycle
```

```
    END IF;
  END IF;
END PROCESS;

sync_reg:
PROCESS (clk)
  -- double buffer i/p data to sync to interface clk
BEGIN
  IF clk'event AND clk='1' THEN
     IF reset = '1' THEN
        -- reset signal sync'ed to clk, so sync reset
       aver1 <= (OTHERS => '0');
       aver2 <= (OTHERS => '0');
       addr1 <= (OTHERS => '0');
       data1 <= (OTHERS => '0');
       wnr1 <= '0';
       enable1 <= '1';
       addr2 <= (OTHERS => '0');
       data2 <= (OTHERS => '0');
       wnr2 <= '0';
       enable2 <= '1';
    ELSE
       aver1 <= average;
       aver2 <= aver1;
       addr1 <= addr_reg;
       data1 <= data_reg;
       wnr1 <= wnr_reg;
       enable1 <= enable;
       addr2 <= addr1;
       data2 <= data1;
       wnr2 <= wnr1;
       enable2 <= enable1;
     END IF;
  END IF;
END PROCESS sync_reg;

store_reg:
PROCESS (reset, clk)
  -- register threshold and data_sel between write cycles
BEGIN
  -- reset signal not sync'ed to clk, so async reset
  IF reset = '1' THEN
    thresh_reg <= (OTHERS => '0');
    datasel_reg <= (OTHERS => '0');
  ELSIF clk'event AND clk='1' THEN
    thresh_reg <= threshold_i;
    datasel_reg <= data_sel_i;
  END IF;
END PROCESS store_reg;


-- only data output is average data
WITH tri_bus SELECT
   dataio <= ('0' & aver2) WHEN '0',
             (OTHERS => 'Z') WHEN OTHERS;
```

```
  data_in <= NOT(dataio);
END rtl;
```

Appendix B-6. RAM Write Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: Read/write pulse generator for linestore and
bitstore
--              FIFO models
--
------------------------------------------------------------
----------
--
-- File name: ramwrite.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE WORK.std_class.ALL;

ENTITY ramwrite IS
  PORT(clk4  : IN STD_LOGIC;
       reset  : IN STD_LOGIC;
       wr_pulse : OUT STD_LOGIC);
END ramwrite;

ARCHITECTURE rtl OF ramwrite IS
  SIGNAL count:slv2;
  TYPE t_lookup IS ARRAY (0 to 3) of std_logic_vector(1
downto 0);
  CONSTANT table : t_lookup := ("10", "00", "11", "01");
BEGIN
  gray_code:
  PROCESS(clk4, reset)
  BEGIN
     IF reset = '1' THEN
       count <= "00";
     ELSIF clk4'event AND clk4 = '1' THEN
       count <= table(CONV_INTEGER(UNSIGNED(count)));
     END IF;
  END PROCESS gray_code;
  wr_pulse <= count(1) AND NOT count(0);
END rtl;
```

Appendix B-7. RAM Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: 8 bit, 32 word synchronous single port RAM
--
--
------------------------------------------------------------
----------
--
-- File name: generic_ram.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

Library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
--use IEEE.std_logic_unsigned.all ;

ENTITY generic_ram IS
   PORT (
         d         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
         address   : IN  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
         we, clk   : IN  STD_LOGIC ;
         q         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) :=
(others => '0')
        );
END generic_ram ;

------------------------------------------------------------
----------
--
-- Infered single port RAM models
--
--
------------------------------------------------------------
----------

ARCHITECTURE rtl OF generic_ram IS
  TYPE mem_type IS ARRAY (2**5 DOWNTO 0) OF
       STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL mem : mem_type := (others=>(others=>'0')) ;

  SIGNAL int_ad : STD_LOGIC_VECTOR(4 DOWNTO 0) :=
         (others =>'0');

BEGIN

    -- Synchronous RAM
    PROCESS (clk)
```

```
    BEGIN
      IF clk'EVENT AND clk ='1' THEN
        int_ad <= address;
        IF (we = '1') THEN
          mem(CONV_INTEGER(UNSIGNED(address))) <= d ;
        END IF ;
      END IF;
    END PROCESS;

    q <= mem(CONV_INTEGER(UNSIGNED(int_ad))) ;


END RTL ;
```

Appendix B-8. Filtercore Code Using the Proposed Extension

```
------------------------------------------------------------
----------
--
-- Description: Top level structural model for
--              edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: filtercore.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
---------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

ENTITY filtercore IS
  PORT(
        dataio : inout slv16;
        address: IN slv3;        -- proc i/f r/w address
        lsram_op: OUT slv8;   -- linestore RAM output for
test
        scene   : IN slv8;       -- input intensity data
        ifclk   : IN STD_LOGIC;  -- fast clk for i/f
        clk     : IN STD_LOGIC;
        clk4    : IN STD_LOGIC;
        edge_fs : OUT STD_LOGIC; -- o/p data frame sync
        fs   : IN STD_LOGIC;     -- image sync control
        ack  : OUT STD_LOGIC;    -- proc i/f acknowledge
        enable : IN STD_LOGIC;   -- proc i/f enable
        wr_nrd : IN STD_LOGIC;   -- proc i/f r/w
        edge : OUT STD_LOGIC;    -- o/p binary edge data
        ls   : IN STD_LOGIC;     -- image sync control
        reset : IN STD_LOGIC     -- reset control
      );
END filtercore;

ARCHITECTURE struct OF filtercore IS

  SIGNAL ns_result, we_result : slv15;
  SIGNAL new_frame, wr_pulse : STD_LOGIC;
  SIGNAL average:slv15;
  SIGNAL data_sel:slv2;
  SIGNAL data_out:slv16;
  SIGNAL threshold:slv15;
```

```
  COMPONENT ramwrite
    PORT(clk4  : IN STD_LOGIC;
         reset  : IN STD_LOGIC;
        wr_pulse : OUT STD_LOGIC);
  END COMPONENT;

  COMPONENT mult_mask
     PORT(scene   : IN slv8;
          ls,fs   : IN STD_LOGIC;
          reset   : IN STD_LOGIC;
          clk     : IN STD_LOGIC;
          clk4    : IN STD_LOGIC;
          wr_pulse  : IN STD_LOGIC;
          new_frame : OUT STD_LOGIC;
          ns_result  : OUT slv15;
          we_result  : OUT slv15;
          ls_out  : OUT slv8);
  END COMPONENT;

  COMPONENT thresh_mux
    PORT(ns_result : IN slv15;
         we_result : IN slv15;
         new_frame : IN STD_LOGIC;
         clk       : IN STD_LOGIC;
         reset     : IN STD_LOGIC;
         threshold : IN slv15;
         data_sel  : IN slv2;
         frame_sync: OUT STD_LOGIC;
         average   : OUT slv15;
         mux_bit   : OUT STD_LOGIC
         );
  END COMPONENT;

  COMPONENT interface
    PORT(clk       : IN STD_LOGIC;
         reset     : IN STD_LOGIC;
         address   : IN slv3;
         dataio    : INOUT slv16;
         wr_nrd    : IN STD_LOGIC;
         enable    : IN STD_LOGIC;
         average   : IN slv15;
         ack       : OUT STD_LOGIC;
         threshold : OUT slv15;
         data_sel  : OUT slv2
         );
  END COMPONENT;

BEGIN

  ramwr: ramwrite
       PORT MAP(clk4, reset, wr_pulse);

  double:mult_mask
       PORT MAP(scene, ls, fs, reset, clk, clk4, wr_pulse,
```

```
new_frame, ns_result, we_result, lsram_op);

  thrmux:thresh_mux
       PORT MAP(ns_result, we_result, new_frame, clk, reset,
 threshold, data_sel, edge_fs, average, edge);

  procif:interface
       PORT MAP(ifclk, reset, address, dataio, wr_nrd,
enable,
                average, ack, threshold, data_sel);

END struct;
```

Appendix B-9. Mult Mask Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: Dual mask, dual linestore controller and
instantiator.
--              Part of edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: mult_mask.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

USE flowlab.mask_types.ALL;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.all;

ENTITY mult_mask IS
  PORT(scene   : IN slv8;
       ls,fs   : IN STD_LOGIC;
       reset   : IN STD_LOGIC;
       clk     : IN STD_LOGIC;
       clk4    : IN STD_LOGIC;
       wr_pulse  : IN STD_LOGIC;
       new_frame : OUT STD_LOGIC;
       ns_result  : OUT slv15;
       we_result  : OUT slv15;
       ls_out  : OUT slv8);
END mult_mask;

ARCHITECTURE rtl OF mult_mask IS

  -- mult mask states
  constant sync_fs : std_logic_vector(11 downto 0)
                     := "000000000001";
  constant sync_ls : std_logic_vector(11 downto 0)
                     := "000000000010";
  constant last_pxl : std_logic_vector(11 downto 0)
                     := "000000000100";
  constant first_line : std_logic_vector(11 downto 0)
                     := "000000001000";
  constant end_first : std_logic_vector(11 downto 0)
                      := "000000010000";
  constant sec_line : std_logic_vector(11 downto 0)
                      := "000000100000";
  constant end_line : std_logic_vector(11 downto 0)
```

```
                    := "000001000000";
  constant blank : std_logic_vector(11 downto 0)
                    := "000010000000";
  constant valid : std_logic_vector(11 downto 0)
                    := "000100000000";
  constant frame_sync : std_logic_vector(11 downto 0)
                    := "001000000000";
  constant start_last : std_logic_vector(11 downto 0)
                    := "010000000000";
  constant last_line : std_logic_vector(11 downto 0)
                    := "100000000000";

  -- mult mask state variables
  SIGNAL current_state,
         next_state : std_logic_vector(11 downto 0);

  SIGNAL ls_cap, fs_cap, ls_sync, fs_sync:STD_LOGIC;
  SIGNAL next_new_frame, new_frame_sync:STD_LOGIC;
  SIGNAL scene_reg :image_data;
  SIGNAL top_data, middle_data:image_data;
  SIGNAL top_data_reg, middle_data_reg:image_data;
  SIGNAL blankop, blankop_cap :BOOLEAN;
  SIGNAL ns_mask_out, we_mask_out:filt_data;
  SIGNAL ns_filt, we_filt:abs_filt;

  COMPONENT nsmask
    PORT(bottom_in : IN image_data;
         middle_in : IN image_data;
         top_in    : IN image_data;
         reset     : IN STD_LOGIC;
         clk       : IN STD_LOGIC;
         clk4      : IN STD_LOGIC;
         mask_out  : OUT filt_data);
  END COMPONENT;

  COMPONENT wemask
    PORT(bottom_in : IN image_data;
         middle_in : IN image_data;
         top_in    : IN image_data;
         reset     : IN STD_LOGIC;
         clk       : IN STD_LOGIC;
         clk4      : IN STD_LOGIC;
         mask_out  : OUT filt_data);
  END COMPONENT;

  COMPONENT linestore
    PORT (wr_data  : IN image_data;
          rd_data  : OUT image_data;
          reset    : IN STD_LOGIC;
          wr_pulse : IN STD_LOGIC;
          ls_clk   : IN STD_LOGIC;
          clk4     : IN STD_LOGIC);
  END COMPONENT;

BEGIN
```

```
  midrow:linestore
         PORT MAP (wr_data => scene_reg, rd_data =>
middle_data, reset => reset, wr_pulse => wr_pulse,
                  ls_clk => clk, clk4 => clk4);

  toprow:linestore
         PORT MAP (wr_data => middle_data_reg, rd_data =>
top_data, reset => reset, wr_pulse => wr_pulse,
                  ls_clk => clk, clk4 => clk4);

  nthsth:nsmask
         PORT MAP (bottom_in => scene_reg, middle_in =>
middle_data_reg, top_in => top_data_reg,
                  reset => reset, clk => clk, clk4 => clk4,
mask_out => ns_mask_out);
  wstest:wemask
         PORT MAP (bottom_in => scene_reg, middle_in =>
middle_data_reg, top_in => top_data_reg,
                  reset => reset, clk => clk, clk4 => clk4,
mask_out => we_mask_out);

 -- complete mask operation by normalising absolute mask
result

  ns_complete:
  PROCESS (ns_mask_out, blankop_cap)
    VARIABLE ns_filt : slv16;
  BEGIN
    ns_filt := conv_std_logic_vector(ns_mask_out, 16);
    IF ns_filt(15) = '1' THEN
      ns_filt := unsigned(NOT(ns_filt)) + '1';
    END IF;
    ns_filt := "00" & ns_filt(15 downto 2);

    IF blankop_cap THEN
      ns_result <= (OTHERS => '0');
    ELSE
      ns_result <= ns_filt(14 downto 0);
    END IF;
  END PROCESS ns_complete;

  we_complete:
  PROCESS (we_mask_out, blankop_cap)
    VARIABLE we_filt : slv16;
  BEGIN
    we_filt := conv_std_logic_vector(we_mask_out, 16);
    IF we_filt(15) = '1' THEN
      we_filt :=  unsigned(NOT(we_filt)) + '1';
    END IF;
    we_filt := "00" & we_filt(15 downto 2);

    IF blankop_cap THEN
      we_result <= (OTHERS => '0');
    ELSE
```

```
      we_result <= we_filt(14 downto 0);
    END IF;
  END PROCESS we_complete;

  -- write linestore o/p to external port for RAM test
ls_out <= conv_std_logic_vector(top_data, 8);

state_reg:
  PROCESS(CLK, RESET)
  BEGIN
    IF reset = '1' THEN
      current_state <= sync_fs;
      blankop_cap <= TRUE;
    ELSIF clk'event AND clk = '1' THEN
      current_state <= next_state;
      blankop_cap <= blankop;
    END IF;
  END PROCESS state_reg;

state_assign:
  PROCESS (current_state, fs_sync, ls_sync)

  BEGIN

    next_state <= current_state;
    blankop<=true;               -- default is input is
blanked
    next_new_frame <= '0';       -- default is not new frame

      IF current_state = sync_fs THEN
        IF fs_sync = '0' THEN
          next_state <= sync_ls;
        END IF;

      ELSIF current_state = sync_ls THEN
        IF ls_sync = '0' THEN
          next_state <= last_pxl;
          next_new_frame <= '1';
        END IF;

      ELSIF current_state = last_pxl THEN
        IF ls_sync = '1' THEN
          next_state <= first_line;
        END IF;

      ELSIF current_state = first_line THEN
        IF ls_sync = '0' THEN
          next_state <= end_first;
        END IF;

      ELSIF current_state = end_first THEN
        IF ls_sync = '1' THEN
          next_state <= sec_line;
        END IF;
```

```
      ELSIF current_state = sec_line THEN
        IF ls_sync = '0' THEN
          next_state <= end_line;
        END IF;

      ELSIF current_state = end_line THEN
        IF ls_sync = '1' THEN
          next_state <= blank;
        END IF;

      ELSIF current_state = blank THEN
        next_state <= valid;

      ELSIF current_state = valid THEN
        blankop<=false;
        IF (ls_sync = '0' AND fs_sync = '0') THEN
          next_state <= frame_sync;
        ELSIF (ls_sync = '0') THEN
          next_state <= end_line;
        END IF;

      ELSIF current_state = frame_sync THEN
        IF(ls_sync = '1' AND fs_sync = '1') THEN
          next_state <= start_last;
        END IF;

      ELSIF current_state = start_last THEN
        next_state <= last_line;

      ELSIF current_state = last_line THEN
        blankop<=false;
        IF ls_sync = '0' THEN
          next_state <= last_pxl;
          next_new_frame <= '1';
        END IF;

    END if;

  END PROCESS state_assign;

data_reg:
  PROCESS(clk, reset)
  BEGIN
    IF reset = '1' THEN
      scene_reg <= 0;
      middle_data_reg <= 0;
      top_data_reg <= 0;
      ls_cap <= '1';
      fs_cap <= '1';
      ls_sync <= '1';
      fs_sync <= '1';
      new_frame <= '0';
      new_frame_sync <= '0';
    ELSIF clk'event AND clk = '1' THEN
      scene_reg <= conv_integer(unsigned(scene));
```

```
      middle_data_reg <= middle_data;
      top_data_reg <= top_data;
      ls_cap <= ls;
      fs_cap <= fs;
      ls_sync <= ls_cap;
      fs_sync <= fs_cap;
      new_frame_sync <= next_new_frame;
      new_frame <= new_frame_sync;
    END IF;
  END PROCESS data_reg;

END rtl;
```

Appendix B-10. NS Mask and WE Mask Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: customised nsmask
--
------------------------------------------------------------
----------
--
-- File name: nsmask.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY nsmask IS
  PORT(bottom_in : IN image_data;
       middle_in : IN image_data;
       top_in    : IN image_data;
       reset     : IN STD_LOGIC;
       clk       : IN STD_LOGIC;
       clk4      : IN STD_LOGIC;
       mask_out  : OUT filt_data);
END nsmask;

ARCHITECTURE rtl OF nsmask IS

  TYPE mask_row IS ARRAY (0 TO mask_size) OF image_data;

  SIGNAL topl_data, topm_data, topr_data:image_data;
  SIGNAL midl_data, midm_data, midr_data:image_data;
  SIGNAL botl_data, botm_data, botr_data:image_data;
  SIGNAL prelim, prelim_reg :filt_data;
  TYPE calc_state_t IS (init, top, middle, bottom, pause);
  SIGNAL calc_state:calc_state_t;

BEGIN

  maskshift:
  -- shift mask scene data and read in new values
  -- from top and middle linestores, and scene data input
  PROCESS(clk, reset)
  BEGIN
    IF reset = '1' THEN
      topl_data <= 0;
      topm_data <= 0;
```

```
    topr_data <= 0;
    midl_data <= 0;
    midm_data <= 0;
    midr_data <= 0;
    botl_data <= 0;
    botm_data <= 0;
    botr_data <= 0;
    mask_out<=0;
  ELSIF (clk'event AND clk = '1') THEN
    topl_data <= topm_data;
    topm_data <= topr_data;
    topr_data <= top_in;
    midl_data <= midm_data;
    midm_data <= midr_data;
    midr_data <= middle_in;
    botl_data <= botm_data;
    botm_data <= botr_data;
    botr_data <= bottom_in;

    -- output mask calculation result
    mask_out<=prelim;

  END IF;
END PROCESS maskshift;

calcseq:
-- state machine to calculate mask arithmetic row by row
-- prelim to store running total
PROCESS (clk4, reset)
BEGIN
  IF reset = '1' THEN
    calc_state<=init;
    prelim_reg<=0;
  ELSIF (clk4'event AND clk4 = '1') THEN
    prelim_reg<=prelim;
    CASE calc_state IS
      WHEN init =>
        calc_state<=pause;
      WHEN pause =>
        calc_state<=top;
        prelim_reg<=0;
      WHEN top =>
        calc_state<=middle;
      WHEN middle =>
        calc_state<=bottom;
      WHEN bottom =>
        calc_state<=pause;
      WHEN OTHERS =>
        calc_state <= init;
     END CASE;
  END IF;
END PROCESS calcseq;

calcomb:
PROCESS(calc_state, topl_data, topm_data, botl_data,
```

```
 topr_data, botm_data, botr_data, prelim_reg)
    VARIABLE left_v, right_v : INTEGER RANGE 0 TO
(((2**image_width)-1)*2);
  BEGIN
    left_v := 0;
    right_v := 0;
    CASE calc_state IS
      WHEN top =>
        left_v := botl_data;
        right_v := topl_data;
      WHEN middle =>
        left_v := botr_data;
        right_v := topr_data;
      WHEN bottom =>
        left_v := 2*botm_data;
        right_v := 2*topm_data;
      WHEN OTHERS =>
        left_v := 0;
        right_v := 0;
    END CASE;
    prelim <= (left_v - right_v) + prelim_reg;
  END PROCESS calcomb;

END rtl;

----------------------------------------------------------------
----------
--
-- Description: customised wemask
--
----------------------------------------------------------------
----------
--
-- File name: wemask.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
----------------------------------------------------------------
----------

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY wemask IS
  PORT(bottom_in : IN image_data;
       middle_in : IN image_data;
       top_in    : IN image_data;
       reset     : IN STD_LOGIC;
       clk       : IN STD_LOGIC;
       clk4      : IN STD_LOGIC;
       mask_out  : OUT filt_data);
END wemask;
```

```
ARCHITECTURE rtl OF wemask IS

  TYPE mask_row IS ARRAY (0 TO mask_size) OF image_data;

  SIGNAL topl_data, topm_data, topr_data:image_data;
  SIGNAL midl_data, midm_data, midr_data:image_data;
  SIGNAL botl_data, botm_data, botr_data:image_data;
  SIGNAL prelim, prelim_reg :filt_data;
  TYPE calc_state_t IS (init, top, middle, bottom, pause);

  SIGNAL calc_state:calc_state_t;

BEGIN

  maskshift:
  -- shift mask scene data and read in new values
  -- from top and middle linestores, and scene data input
  PROCESS
  BEGIN
    WAIT UNTIL (clk'event AND clk = '1');
    IF reset = '1' THEN
      topl_data <= 0;
      topm_data <= 0;
      topr_data <= 0;
      midl_data <= 0;
      midm_data <= 0;
      midr_data <= 0;
      botl_data <= 0;
      botm_data <= 0;
      botr_data <= 0;
      mask_out<=0;
    ELSE
      topl_data <= topm_data;
      topm_data <= topr_data;
      topr_data <= top_in;
      midl_data <= midm_data;
      midm_data <= midr_data;
      midr_data <= middle_in;
      botl_data <= botm_data;
      botm_data <= botr_data;
      botr_data <= bottom_in;

      -- output mask calculation result
      mask_out<=prelim;

    END IF;
  END PROCESS maskshift;

  calcseq:
  -- state machine to calculate mask arithmetic row by row
  -- prelim to store running total
  PROCESS (clk4, reset)
  BEGIN
    IF reset = '1' THEN
```

```
      calc_state<=init;
      prelim_reg<=0;
    ELSIF (clk4'event AND clk4 = '1') THEN
      prelim_reg<=prelim;
      CASE calc_state IS
        WHEN init =>
            calc_state<=pause;
        WHEN pause =>
          calc_state<=top;
          prelim_reg<=0;
        WHEN top =>
          calc_state<=middle;
        WHEN middle =>
          calc_state<=bottom;
        WHEN bottom =>
          calc_state<=pause;
        WHEN OTHERS =>
          calc_state <= init;
       END CASE;
    END IF;
  END PROCESS calcseq;

  calcomb:
  -- of clk4
  PROCESS(calc_state, topl_data, midl_data, botl_data,
topr_data, midr_data, botr_data, prelim_reg)
  BEGIN
    CASE calc_state IS
      WHEN top =>
        prelim <= (botr_data - botl_data) + prelim_reg;
      WHEN middle =>
        prelim <= (topr_data - topl_data) + prelim_reg;
      WHEN bottom =>
        prelim <= (2*midr_data - 2*midl_data) + prelim_reg;
      WHEN OTHERS =>
        prelim <=prelim_reg;
    END CASE;
  END PROCESS calcomb;

END rtl;
```

Appendix B-11. Line Store Code Using the RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: 8 bit, 32 word FIFO, based on dual-port RAM.
--              Part of edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: linestore.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

ENTITY linestore IS
  PORT (wr_data  : IN image_data;
        rd_data  : OUT image_data;
        reset    : IN STD_LOGIC;
        wr_pulse : IN STD_LOGIC;
        ls_clk   : IN STD_LOGIC;
        clk4     : IN STD_LOGIC);
END linestore;

ARCHITECTURE struct OF linestore IS

  SUBTYPE line_pointer IS INTEGER RANGE 0 TO line_length;

  SIGNAL wr_ad, rd_ad,
int_ad:STD_LOGIC_VECTOR(line_data_width-1 DOWNTO 0) :=
(others =>'0');

  SIGNAL rd_slv, wr_slv:STD_LOGIC_VECTOR(image_width - 1
DOWNTO 0) := (others =>'0');

  SIGNAL logic_0 : STD_LOGIC;

  COMPONENT generic_ram
   port (
         d         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
         address   : IN  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
         we, clk   : IN  STD_LOGIC ;
         q         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
```

```
  END COMPONENT;


BEGIN

logic_0 <='0';

memory:generic_ram
       PORT MAP(q => rd_slv, d => wr_slv,
                address => int_ad,
                we => wr_pulse, clk => clk4);

rd_data <= conv_integer(unsigned(rd_slv));

wr_slv <= conv_std_logic_vector(wr_data, image_width);


read_pointer:
PROCESS
  VARIABLE rd_ptr:line_pointer := 0;
BEGIN
  WAIT UNTIL (ls_clk'event AND ls_clk = '1');
  IF reset = '1' THEN
    rd_ptr:=0;
    rd_ad <= CONV_STD_LOGIC_VECTOR(rd_ptr, line_data_width);
  ELSE
  -- read cycle finished
    IF rd_ptr = line_length-1 THEN
      rd_ptr:= 0;
    ELSE
      rd_ptr:=rd_ptr + 1;
    END IF;
    rd_ad <= CONV_STD_LOGIC_VECTOR(rd_ptr, line_data_width);
  END IF;

END PROCESS read_pointer;

write_pointer:
PROCESS
  VARIABLE wr_ptr:line_pointer := 0;
BEGIN
  WAIT UNTIL (ls_clk'event AND ls_clk = '0');
  IF reset = '1' THEN
    wr_ptr:= 0;
    wr_ad <= CONV_STD_LOGIC_VECTOR(wr_ptr, line_data_width);
  ELSE
  -- write cycle finished
    IF wr_ptr = line_length-1 THEN
       wr_ptr:=0;
    ELSE
      wr_ptr:=wr_ptr + 1;
    END IF;
    wr_ad <= conv_std_logic_vector(wr_ptr, line_data_width);
  END IF;
```

```
END PROCESS write_pointer;
int_ad <= wr_ad WHEN ls_clk = '1' ELSE rd_ad;
END struct;
```

Appendix B-12. Threshold Multiplexor Code Using the RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: Threshold, multiplex and average
calculation..
--
--
------------------------------------------------------------
----------
--
-- File name: thresh_mux.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

ENTITY thresh_mux IS
  PORT(ns_result : IN slv15;
       we_result : IN slv15;
       new_frame : IN STD_LOGIC;
       clk       : IN STD_LOGIC;
       reset     : IN STD_LOGIC;
       threshold : IN slv15;
       data_sel  : IN slv2;
       frame_sync: OUT STD_LOGIC;
       average   : OUT slv15;
       mux_bit   : OUT STD_LOGIC
       );
END thresh_mux;

ARCHITECTURE rtl OF thresh_mux IS

  SIGNAL thresh_reg : slv15;
  SIGNAL average_i, average_reg : abs_filt;
  SIGNAL thresh_we, thresh_ns:STD_LOGIC;
  SIGNAL data_sel_reg:slv2;

  CONSTANT zero:slv15:=(OTHERS => '0');

BEGIN

  averager:
  PROCESS (ns_result, we_result, data_sel_reg, reset,
           average_reg, new_frame)
```

```
  BEGIN
    IF reset = '1' THEN
      average_i <= 0;
    ELSIF new_frame = '1' THEN
      average_i <= 0;
    ELSE
      average_i <= average_reg;
      CASE data_sel_reg IS
        WHEN nsbinary =>
          IF ns_result /= zero THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(ns_result)) / 2);
          END IF;
        WHEN webinary =>
          IF we_result /= zero THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(we_result)) / 2);
          END IF;
        WHEN OTHERS =>
          IF (ns_result /= zero) AND
             (ns_result > we_result) THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(ns_result)) / 2);
          ELSIF we_result /= zero THEN
            average_i <= (average_reg / 2) +
(conv_integer(unsigned(we_result)) / 2);
          END IF;
      END CASE;
    END IF;
  END PROCESS;

  -- threshold ns result
  thresh_ns <= '1' WHEN ns_result > thresh_reg
                   ELSE '0';

  -- threshold we result
  thresh_we <= '1' WHEN we_result > thresh_reg
                    ELSE '0';

cont_regs:
  -- update threshold and data_sel on new_frame
  PROCESS (reset, clk)
  BEGIN
    IF reset = '1' THEN
      thresh_reg <= zero;
      data_sel_reg <= "00";
    ELSIF clk'event AND clk = '1' THEN
      IF new_frame = '1' THEN
        thresh_reg <= threshold;
        data_sel_reg <= data_sel;
      ELSE
        thresh_reg <= thresh_reg;
        data_sel_reg <= data_sel_reg;
      END IF;
    END IF;
```

```
 END PROCESS;

  -- register mult_mask result outputs
thresh_regs:
  PROCESS
  BEGIN
    WAIT UNTIL clk'event AND clk='1';
    IF reset = '1' THEN
      average_reg <= 0;
      frame_sync <= '0';
      mux_bit <= '0';
    ELSE
      average_reg <= average_i;
      frame_sync <= new_frame;
      -- select bit value to output
      CASE data_sel_reg IS
        WHEN nsbinary =>
          mux_bit <= thresh_ns;
        WHEN webinary =>
          mux_bit <= thresh_we;
        WHEN nsorwe =>
          mux_bit <= thresh_ns OR thresh_we;
        WHEN OTHERS =>
          mux_bit <= thresh_ns AND thresh_we;
      END CASE;
    END IF;
  END PROCESS;

  -- write average output
  average <= conv_std_logic_vector(average_reg, 15);

END rtl;
```

Appendix B-13. Interface Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: Asynchronous processor interface model.
--              Part of edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: interface.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
USE flowlab.mask_types.all;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY interface IS
  PORT(clk       : IN STD_LOGIC;
       reset     : IN STD_LOGIC;
       address   : IN slv3;
       dataio    : INOUT slv16;
       wr_nrd    : IN STD_LOGIC;
       enable    : IN STD_LOGIC;
       average   : IN slv15;
       ack       : OUT STD_LOGIC;
       threshold : OUT slv15;
       data_sel  : OUT slv2
       );
END interface;

ARCHITECTURE rtl OF interface IS

  SIGNAL addr_reg, addr1, addr2 : slv3;
  SIGNAL data_reg, data1, data2 : slv16;
  SIGNAL wnr_reg, wnr1, wnr2  : std_logic;
  SIGNAL enable1, enable2  : std_logic;
  SIGNAL aver1, aver2, thresh_reg, threshold_i : slv15;
  SIGNAL datasel_reg, data_sel_i : slv2;
  SIGNAL data_in           : slv16;
  SIGNAL tri_bus           : STD_LOGIC;

BEGIN

  -- intermediate signals used to read output ports
  threshold <= threshold_i;
```

```
  data_sel <= data_sel_i;


  async:
  PROCESS (reset, enable)
  BEGIN
   -- async reset
   IF reset = '1' THEN
     addr_reg <= (OTHERS => '0');
     data_reg <= (OTHERS => '0');
     wnr_reg <= '0';
   ELSIF (enable'event AND enable='0') THEN
   -- register input on falling edge of enable
     addr_reg <= address;
     data_reg <= data_in;
     wnr_reg <= wr_nrd;
   END IF;
  END PROCESS async;


  if_rw:
  PROCESS(wnr2, data2, addr2, enable2, thresh_reg,
datasel_reg)
  BEGIN

    ack <= '1';                   -- default is
unacknowledged
    tri_bus <= '1';               -- default is tristated
bus
    threshold_i <= thresh_reg;
    data_sel_i <= datasel_reg;

    IF enable2 = '0' THEN
      -- bus activity - inputs valid
      IF wnr2 = '1' THEN
        -- write cycle
        CASE addr2 IS
          WHEN "010" =>
            -- threshold data
            threshold_i <= data2(14 DOWNTO 0);
          WHEN "001" =>
            -- data select,
            data_sel_i <= data2(1 DOWNTO 0);
          WHEN OTHERS =>
            null;
        END CASE;
        ack <= '0'; -- acknowledge write
      ELSE
        -- read cycle
        IF addr2 = "100" THEN
          -- read average result value
          -- un-tristate buffers
          tri_bus <= '0';
        END IF;
        ack <= '0'; -- acknowledge read cycle
```

```
      END IF;
    END IF;
  END PROCESS;

  sync_reg:
  PROCESS (clk)
    -- double buffer i/p data to sync to interface clk
  BEGIN
    IF clk'event AND clk='1' THEN
   IF reset = '1' THEN
                        -- reset signal sync'ed to clk, so
sync reset
                        aver1 <= (OTHERS => '0');
                        aver2 <= (OTHERS => '0');
                        addr1 <= (OTHERS => '0');
                        data1 <= (OTHERS => '0');
                        wnr1 <= '0';
                        enable1 <= '1';
                        addr2 <= (OTHERS => '0');
                        data2 <= (OTHERS => '0');
                        wnr2 <= '0';
                        enable2 <= '1';
             ELSE
                        aver1 <= average;
                        aver2 <= aver1;
                        addr1 <= addr_reg;
                        data1 <= data_reg;
                        wnr1 <= wnr_reg;
                        enable1 <= enable;
                        addr2 <= addr1;
                        data2 <= data1;
                        wnr2 <= wnr1;
                        enable2 <= enable1;
             END IF;
    END IF;
  END PROCESS sync_reg;

  store_reg:
  PROCESS (reset, clk)
    -- register threshold and data_sel between write cycles
  BEGIN
    -- reset signal not sync'ed to clk, so async reset
    IF reset = '1' THEN
      thresh_reg <= (OTHERS => '0');
      datasel_reg <= (OTHERS => '0');
    ELSIF clk'event AND clk='1' THEN
      thresh_reg <= threshold_i;
      datasel_reg <= data_sel_i;
    END IF;
  END PROCESS store_reg;


 -- only data output is average data
  WITH tri_bus SELECT
     dataio <= ('0' & aver2) WHEN '0', (OTHERS => 'Z')
```

```
                WHEN OTHERS;
  data_in <= NOT(dataio);
END rtl;
```

Appendix B-14. RAM Write Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: Read/write pulse generator for linestore
--              FIFO model
--              Part of edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: ramwrite.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY ramwrite IS
  PORT(clk4  : IN STD_LOGIC;
       reset  : IN STD_LOGIC;
       wr_pulse : OUT STD_LOGIC);
END ramwrite;

ARCHITECTURE rtl OF ramwrite IS

  SIGNAL count:slv2;
  TYPE t_lookup IS ARRAY (0 to 3) of std_logic_vector(1
downto 0);
  CONSTANT table : t_lookup := ("10", "00", "11", "01");

BEGIN

gray_code:
  PROCESS(clk4, reset)
BEGIN
  IF reset = '1' THEN
    count <= "00";
  ELSIF clk4'event AND clk4 = '1' THEN
    count <= table(CONV_INTEGER(UNSIGNED(count)));
  END IF;
END PROCESS gray_code;

wr_pulse <= count(1) AND NOT count(0);

END rtl;
```

Appendix B-15. RAM Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: 8 bit, 32 word synchronous single port RAM
--
--
------------------------------------------------------------
----------
--
-- File name: generic_ram.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

Library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
--use IEEE.std_logic_unsigned.all ;

ENTITY generic_ram IS
   PORT (
         d         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0) ;
         address   : IN  STD_LOGIC_VECTOR(4 DOWNTO 0) ;
         we, clk   : IN  STD_LOGIC ;
         q         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) :=
(others => '0')
        );
END generic_ram ;

------------------------------------------------------------
----------
--
-- Infered single port RAM models
--
--
------------------------------------------------------------
----------

ARCHITECTURE rtl OF generic_ram IS
  TYPE mem_type IS ARRAY (2**5 DOWNTO 0) OF
STD_LOGIC_VECTOR(7 DOWNTO 0) ;
  SIGNAL mem : mem_type := (others=>(others=>'0')) ;

  SIGNAL int_ad : STD_LOGIC_VECTOR(4 DOWNTO 0) := (others
=>'0');

BEGIN

    -- Synchronous RAM
    PROCESS (clk)
```

```
    BEGIN
      IF clk'EVENT AND clk ='1' THEN
        int_ad <= address;
        IF (we = '1') THEN
          mem(CONV_INTEGER(UNSIGNED(address))) <= d ;
        END IF ;
      END IF;
    END PROCESS;

    q <= mem(CONV_INTEGER(UNSIGNED(int_ad))) ;


END RTL ;
```

Appendix B-16. Filtercore Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: Top level structural model for
--              edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: filtercore.vhd
-- Version  : 1.0
-- Author   : David Cabanis
--
------------------------------------------------------------
---------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

LIBRARY flowlab;
USE flowlab.mask_types.ALL;

ENTITY filtercore IS
  PORT(
   dataio     : inout slv16;
   address    : IN slv3;        -- proc i/f r/w address
   lsram_op   : OUT slv8;       -- linestore RAM output
access
   scene      : IN slv8;        -- input intensity data
   ifclk      : IN STD_LOGIC;   -- fast clk for i/f
   clk        : IN STD_LOGIC;
   clk4       : IN STD_LOGIC;
   edge_fs    : OUT STD_LOGIC;      -- o/p data frame sync
   fs         : IN STD_LOGIC;       -- image sync control
   ack        : OUT STD_LOGIC;      -- proc i/f acknowledge
   enable     : IN STD_LOGIC;       -- proc i/f enable
   wr_nrd     : IN STD_LOGIC;       -- proc i/f r/w
   edge       : OUT STD_LOGIC;      -- o/p binary edge data
   ls         : IN STD_LOGIC;       -- image sync control
    reset     : IN STD_LOGIC        -- reset control
      );
END filtercore;

ARCHITECTURE struct OF filtercore IS

  SIGNAL ns_result, we_result : slv15;
  SIGNAL new_frame, wr_pulse : STD_LOGIC;
  SIGNAL average:slv15;
  SIGNAL data_sel:slv2;
  SIGNAL data_out:slv16;
  SIGNAL threshold:slv15;
```

```
  COMPONENT ramwrite
    PORT(clk4  : IN STD_LOGIC;
         reset  : IN STD_LOGIC;
        wr_pulse : OUT STD_LOGIC);
  END COMPONENT;

  COMPONENT mult_mask
     PORT(scene   : IN slv8;
          ls,fs   : IN STD_LOGIC;
          reset   : IN STD_LOGIC;
          clk     : IN STD_LOGIC;
          clk4    : IN STD_LOGIC;
          wr_pulse  : IN STD_LOGIC;
          new_frame : OUT STD_LOGIC;
          ns_result  : OUT slv15;
          we_result  : OUT slv15;
          ls_out  : OUT slv8);
  END COMPONENT;

  COMPONENT thresh_mux
    PORT(ns_result : IN slv15;
         we_result : IN slv15;
         new_frame : IN STD_LOGIC;
         clk       : IN STD_LOGIC;
         reset     : IN STD_LOGIC;
         threshold : IN slv15;
         data_sel  : IN slv2;
         frame_sync: OUT STD_LOGIC;
         average   : OUT slv15;
         mux_bit   : OUT STD_LOGIC
         );
  END COMPONENT;

  COMPONENT interface
    PORT(clk       : IN STD_LOGIC;
         reset     : IN STD_LOGIC;
         address   : IN slv3;
         dataio    : INOUT slv16;
         wr_nrd    : IN STD_LOGIC;
         enable    : IN STD_LOGIC;
         average   : IN slv15;
         ack       : OUT STD_LOGIC;
         threshold : OUT slv15;
         data_sel  : OUT slv2
         );
  END COMPONENT;

BEGIN

  ramwr: ramwrite
       PORT MAP(clk4, reset, wr_pulse);

  double:mult_mask
       PORT MAP(scene, ls, fs, reset, clk, clk4, wr_pulse,
```

```
 new_frame, ns_result, we_result, lsram_op);

  thrmux:thresh_mux
       PORT MAP(ns_result, we_result, new_frame, clk, reset,
threshold, data_sel, edge_fs, average, edge);

  procif:interface
       PORT MAP(ifclk, reset, address, dataio, wr_nrd,
enable,
average, ack, threshold, data_sel);

END struct;
```

Appendix B-17. Mask Types Code Using RTL VHDL

```
------------------------------------------------------------
----------
--
-- Description: Package of constant and type declarations
for
--              edge filter testcase model.
--
------------------------------------------------------------
----------
--
-- File name: mask_types.vhd
-- Version  : 3.0
-- Author   : David Cabanis
--
------------------------------------------------------------
----------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE mask_types IS

  CONSTANT line_data_width:NATURAL:=5;
  CONSTANT line_length :NATURAL:=2**line_data_width;
  CONSTANT frame_length:NATURAL:=line_length;

  CONSTANT image_width:NATURAL:=8;
  SUBTYPE image_data IS INTEGER RANGE 0 TO (2**image_width)-
1;

  CONSTANT mask_length:NATURAL:=3;
  CONSTANT mask_size:NATURAL:=(mask_length-1);

  CONSTANT mask_divisor:NATURAL:=4;

  SUBTYPE filt_data IS INTEGER RANGE -32768 TO 32767;
  SUBTYPE abs_filt IS INTEGER RANGE 0 TO 32767;

  SUBTYPE slv2 IS STD_LOGIC_VECTOR(1 DOWNTO 0);
  SUBTYPE slv3 IS STD_LOGIC_VECTOR(2 DOWNTO 0);
  SUBTYPE slv8 IS STD_LOGIC_VECTOR(7 DOWNTO 0);
  SUBTYPE slv15 IS STD_LOGIC_VECTOR(14 DOWNTO 0);
  SUBTYPE slv16 IS STD_LOGIC_VECTOR(15 DOWNTO 0);

  -- data select multiplexor values
  CONSTANT nsbinary:slv2:="01";
  CONSTANT webinary:slv2:="10";
  CONSTANT nsorwe  :slv2:="11";
  CONSTANT nsandwe :slv2:="00";

END mask_types;
```