Compiling VHDL into a High-Level Synthesis Design Representation

Petru Eles* Krzysztof Kuchcinski† Zebo Peng† Marius Minea*

* Computer Science and Engineering Department
Technical University of Timisoara
Romania

† Dept. of Computer and Information Science
Linköping University
Sweden

Abstract

This paper presents an approach to use VHDL as input specification to the CAMAD high-level synthesis system. In particular, it describes a synthesis-oriented compiler which takes a subset of VHDL as input and compiles it into the internal design representation of CAMAD, which can then be synthesized into register-transfer level design. Since CAMAD supports the design of hardware with concurrency and asynchrony, our VHDL subset includes the concurrent features of the language. We present also in the paper some important conclusions concerning how to deal with signals, wait statements, structured data, and subprograms.

1. Introduction

VHDL [5] is one of the most widely used languages in digital circuit design. The existing IEEE standard defines a very rich language for hardware description and simulation. However, the problem of extending the use of VHDL to the field of hardware synthesis does not have any definitive solutions yet. There are even discussions on to what extent VHDL is adequate as a synthesis language. The main difficulty of using VHDL for synthesis is resulted from the simulation-oriented semantics of standard VHDL; some of the VHDL features cannot be synthesized and others can only be synthesized using very sophisticated hardware. Therefore, it is quite obvious that a useful and efficient high-level synthesis system should accept only a subset of VHDL, possibly with some synthesis-oriented extensions which can be ignored when simulation is carried out.

In this paper we present a synthesis-oriented compiler based on a broad subset of VHDL. We describe the language subset, the internal design representation based on an extended timed Petri net model, and the implementation of the compiler as part of the CAMAD high-level synthesis system developed at Linköping University [9]. One of the main issues addressed in this paper is how to capture the VHDL semantics by our design representation. Our approach differs from most of the other VHDL synthesis projects by considering a wider class of VHDL descriptions and dealing with synthesis of concurrent processes.

The paper is divided into six sections. Section 2 discusses some previous work on VHDL synthesis and outlines our approach. Section 3 presents the internal design representation used in our synthesis system. Section 4 describes the compilation of the selected VHDL subset into the internal representation together with a discussion of some semantic aspects of the VHDL subset. Finally section 5 deals with the implementation of the compiler and section 6 presents our conclusions.

2. Background and Related Work

The most difficult and widely discussed features of VHDL, from the point of view of synthesis, are the timing model, concurrency and synchronization, and subprograms. However, most previous work makes use of only the sequential aspects of VHDL. For example, in [3], Camposano assumes that the behavioral hardware specification captured by VHDL is sequential and only a synchronous hardware is synthesized. Thus, sensitivity clauses of the wait statements are ignored and wait on signals is not allowed. VSYNTH, the behavioral synthesis system described in [4], is another example of using a VHDL subset that is restricted to a purely sequential description. As in Camposano's work, the architecture body may only contain a single process. No wait statement is accepted.

Bender and Stevens [1,2] point out that a VHDL description is difficult to synthesize efficiently, mainly because of the low level synchronization and communication concepts based on signals. To overcome

This work has been partially sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK)

0-8186-2780-8/92 $3.00 © 1992 IEEE

this difficulty they replace the signal concept by several other concepts representing different synchronization and communication facilities. By doing this, practically a new language with a different semantic is defined.

In [10] Postula describes SynVHDL, a subset of VHDL for high-level synthesis. It is defined based on the assumption that a design will be described as a set of processes to be synthesized one at a time. Each process will result in a separate synchronous hardware module. In SynVHDL an architecture body contains a single process, and wait statements are allowed to have only a clock signal on their sensitivity lists. No subprograms are allowed.

In [6] Lis and Gajski propose a methodology to cope with the difficulties of VHDL synthesis. Their approach differs in some way from those discussed above; instead of imposing restrictions directly on the language, they define four design models and recommend for each model a corresponding description style. The compiler embodied in their system works in four different modes according to the corresponding VHDL descriptions.

The work discussed above usually integrates VHDL into a high-level synthesis environment by imposing restrictions on the language. The authors try to solve the basic problem of interpreting VHDL's semantics in the world of synthesis, by excluding some strictly simulation-oriented facilities from the language. Their methods usually synthesize only one part of the VHDL description without considering its relation with the rest. Most of these systems restrict themselves to a practically sequential subset of VHDL, with a very restricted use of signals.

Our approach is to accept for synthesis a larger subset of standard VHDL, which we called S'VHDL. When defining the subset we eliminate first of all facilities which are ambiguous (for instance those related to timing) or irrelevant (those connected to structural description, access types, etc.) from the point of view of high level synthesis. The overall structure of a program in S'VHDL comprises entity declarations, architecture bodies, package declarations and package bodies with the following properties:

- an architecture body may contain any number of concurrent statements;
- scalar and composite types, with the exception of access and file types, are accepted;
- signals can only be of scalar or bit-string type;
- recursive calls are not allowed in procedures;
- all sequential statements, with the exception of the assertion statements, are accepted; and
- the structural aspects (such as component instantiation or generate statements) are excluded.

We have implemented a compiler which takes a digital system specification in S'VHDL and generates an internal design representation. The compiler is designed as part of the CAMAD system, which synthesizes the internal representation into a register-transfer level design.

3. The ETPN Design Representation

The internal design representation of CAMAD is called ETPN (extended timed Petri net) [7, 8], which has been developed to capture the intermediate results during the high level synthesis process. The representation model is based on two separate but related parts: control and data part. The representation uses Petri nets to provide a *concurrent and asynchronous* description of control.

The *data path* of the design representation is represented as a directed graph with nodes and arcs. The nodes are used to capture data manipulation and storage units. The arcs represent the connections of the nodes. The *control part* of the design representation, on the other hand, is captured as a timed Petri net with restricted transition firing rules. These two parts are related by the control signals coming from the control part to the data path, and the conditional signals traveling in the opposite direction.

In the examples throughout the paper, data path nodes will be represented as rectangles with labels indicating the functions of the nodes or their names (if the node is a register). The arcs of the data path represent the data flow between function nodes. Communication of data from one node to another is controlled by the control signals coming from the control part. The control relation is indicated by using control state labels to guard arcs. When a control state S_i in the Petri net representing the control flow, holds a token, its associated arcs in the data path (arcs guarded by the corresponding label) will be open for data to flow.

Control states or places of the control Petri net will be depicted in our examples as circles. The transitions of control states are represented as firings of one or several transitions of the Petri net, which are depicted as bars. To express that the control flow can be guarded by results of internal computations, we use conditional signals to guard the control flow. A transition may be guarded by one or more conditions produced from the data path. A transition may be fired when it is enabled (all its input places have a token) and the guarding condition is true. If a transition has more than one guarding condition and at least one of them is true, the transition's guarding condition is true.

4. Compilation of S'VHDL to ETPN

In this section, we present the compilation of the selected

VHDL subset into the ETPN design representation by discussing how certain basic S'VHDL constructs can be represented in the ETPN model.

4.1 Wait Statements and Signal Assignments

In VHDL the wait statement can have three basic forms. Two of them, namely "wait for" and "wait until" are relatively easy to be represented in ETPN while "wait on" statement cause problems. In this section we will concentrate mainly on "waiting on events and transactions" while other forms will be briefly discussed later.

A wait statement on a signal in a S'VHDL process will result in suspending the process until an event on the specified signal occurs. Such an event occurs when the signal changes its value as the result of an assignment statement. In Figure 1, we show how the wait statement (a) and the signal assignment (c) are represented in ETPN. Waiting for an event is solved by associating the condition C_s to a transition in the waiting process. The condition C_s will be produced as result of an assignment to the respective signal, if the value of the signal changes. For reasons of simplicity we will use a compressed representation for signals (equivalent to that in Figure 1(c)) as illustrated in Figure 1(d).

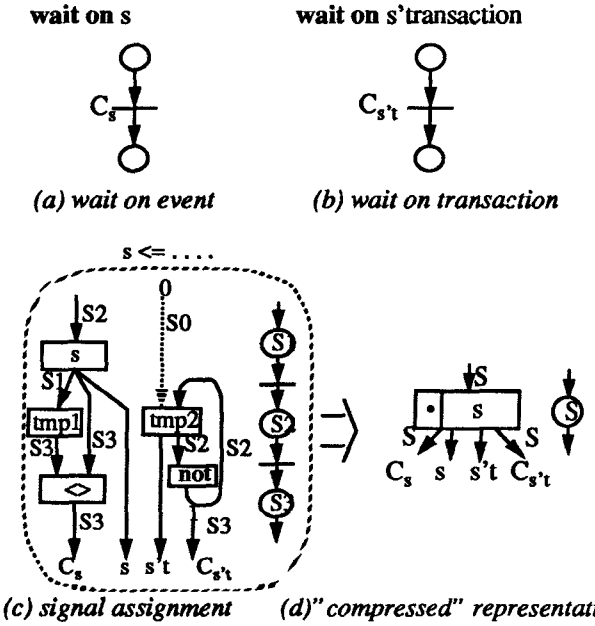

Fig. 1. Wait on signals and signal assignments

A process can also wait for a transaction occurring on a signal. A transaction happens whenever a value is assigned to a signal regardless if its value changes or not. This is done, in VHDL, by referring explicitly to the attribute *transaction* of the given signal, as illustrated in Figure 1(b).

Figure 1(c) shows also how the condition $C_{s't}$, corresponding to *s'transaction*, and the value of *s'transaction* (*s't*) are generated. The condition $C_{s't}$ is *true* (*1*) whenever a value is assigned to the signal. When generating the value of the attribute *transaction* of a signal *s*, we followed the semantics defined for VHDL. It means that if *s* is a signal of any type, than *s'transaction* is a signal of type *bit*, whose value changes to the inverse of the previous one whenever a value is assigned to signal *s*. In Figure 1(c) we depicted also the initialization of *s'transaction*, by an initial state *S0*.

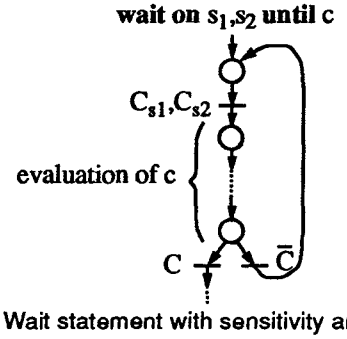

Fig. 2. Wait statement with sensitivity and condition clause

A process can wait not only on a sensitivity list consisting of one or more signals, but also on a condition. Sensitivity lists and condition clauses can appear together or alone in a wait statement. An example of such wait statements is given in Figure 2, where the control Petri net corresponding to a wait on a sensitivity list containing two signals, and a condition clause is shown. According to the VHDL semantics the ETPN representation will have the transition guarded by conditions C_{s1} and C_{s2} corresponding to the signals *s1* and *s2* followed by the transitions guarded by *C* and *not C*, which correspond to the condition clause.

4.2 Subprograms

There exist different approaches to treat subprograms in a high level synthesis system. The main problem is wether to preserve their structural meaning and synthesize them as separate pieces of hardware, or to expand them in the calling process. We used both approaches in our system.

In Figure 3, we show the control Petri net (a) and the interface data path (b) for a VHDL example containing two successive calls of the same procedure *p*. The dotted lines indicate the control transfer at procedure call and return. The data path corresponds to the VHDL semantics of parameter transfer for variable and constant parameters. It represents the assignment from the actual parameters to the formal ones at the call for in and inout parameters, and the assignment from the formal parameters to the actual ones at return for out and inout parameters. Since procedures are called sequentially from one process the data path will not

be accessed at the same time and do not need to be duplicated.

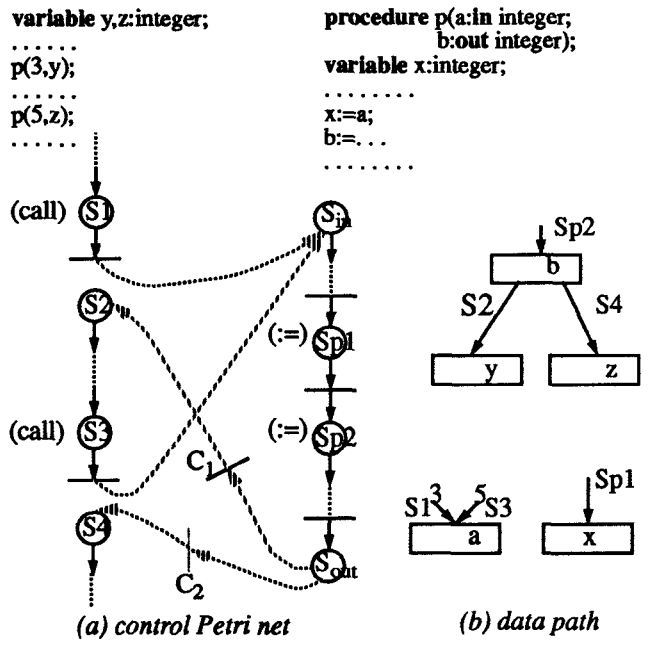

Fig. 3. Procedure calls

In the case of subprograms in-line expansion, we have in fact as many subprogram bodies as calls, and there is practically no call and return in the resulted hardware. In the other case when subprograms are synthesized separately and accessed at each call, there arise additional problems. One is related to the fact, that the return must occur to different points, depending on where the call came from. It is in fact the problem of saving the return address. In our case we solve it by using conditions C_1 and C_2 generated by the data path to guide the control transfer, as shown in Figure 3 and 4.

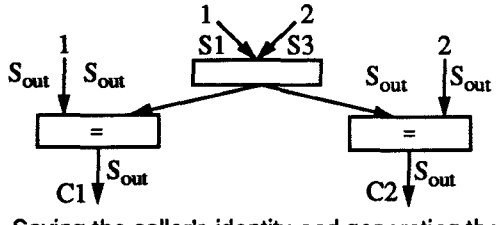

Fig. 4. Saving the caller's identity and generating the return conditions

Another problem is the concurrent procedure call when subprograms are shared between processes. In Figure 3 we considered that the two procedure calls originate from the same process and consequently are strictly sequential. This is always the case with subprograms declared inside a process, and consequently not visible outside it. If a subprogram is called by more than one process and it is expanded, each process will execute its own copy of the subprogram body and there are no problems related to sharing. If the subprogram would be synthesized as a single piece of hardware, it would be necessary to protect it against being entered by more than one process at a time. In principle, this problem can be solved by using mutual exclusion mechanisms and transforming procedures into critical resources. This can result, however, in a undesirable situation which can cause deadlock. We illustrate the problem with the following example.

```
signal s:integer:=0;
procedure p(x:integer) is
.....
    if x=1 then
        wait s;
.....
end p;

process
.....
p(1);
.....
end process;

process
.....
p(2);
s<=...
.....
end process;
```

When using VHDL for simulation, there is no problem with the above example. The simulator accepts multiple activations of a subprogram. But if subprograms are implemented as hardware modules and are protected for multiple activations, and the call *p(1)* is executed first, the two processes will deadlock. Such a deviation from the standard VHDL semantics cannot be accepted. This is a strong argument to perform in-line expansion of subprograms that are called from more than one process.

An additional care must be devoted to subprogram parameters of the signal type if the subprogram is not expanded. According to the VHDL semantics, the actual signal is associated to the formal one at the beginning of each call; thereafter, during the execution of the subprogram body, a reference and an assignment to the formal parameter is equivalent to a reference or assignment to the actual one. To resolve possible conflict due to the access to the actual signal, we can use the same method we used to solve the problem of returning from subprograms.

In our approach, the programer has the possibility to choose if he/she wants a subprogram to be expanded or to be synthesized separately. If there is no information given from the programer the system will make a decision based on program analysis results. No matter which alternative is chosen, in order to maintain the knowledge of the initial structure, from the source program to the internal representation, components of the control net and the data path are labeled with attributes indicating the processes, subprograms, and blocks they belong to.

4.3 Timing Model

Due to the simulation-oriented definition of the semantics of standard VHDL, it is assumed that all statements between two wait statements are executed in

zero time. This assumption can not be implemented in hardware. The VHDL *after* clause in a signal assignment specifies also a strict simulation delay time, which is impossible (and useless) to be synthesized exactly in hardware. The same applies to the time clause in a wait statement. To make it possible to consider timing information during the synthesis process, we interpret the timing model of S'VHDL as follows:

- the *after* clause in a signal assignment represents a maximal allowed delay of the synthesized hardware which is used to carry out the respective assignment;
- a wait with a time clause specifies the minimal delay from the previous wait statement.

The timing model of VHDL is also relevant to signals and their handling. Unlike variables, which are updated as soon as they are assigned, signals are only updated at the end of a simulation cycle. This means that, after an assignment, a signal preserves it's old value until the next wait is executed. We believe that this semantics, strictly connected to the VHDL simulation model, should not be preserved for synthesis. We have decided to update signal values after assignments in the same way as variables. This means that a S'VHDL signal assignment is semantically equivalent to a VHDL signal assignment followed by a *wait for 0* statement.

5. Implementation of the S'VHDL Compiler

The S'VHDL compiler is implemented as part of the CAMAD system. CAMAD generates the structural description of a digital system, starting from a high level specification. CAMAD acts like the front-end of a complete VLSI synthesis system, as shown in Figure 5(a). The S'VHDL compiler described here is intended to work as a front-end for the CAMAD. It translates a S'VHDL description into an ETPN representation.

The high level description of a digital system in S'VHDL represents a behavioral specification which describes the functionality of the circuit. Figure 5(b) shows how the S'VHDL source program is successively translated to the ETPN internal form. Between the external source and the final ETPN, the program passes two internal intermediate forms: the program graph and the data flow description. These successive representations are illustrated for the following program:

```
package pac is
    procedure p;
end pac;

package body pac is
    constant a:integer := 3;
    procedure p is
        variable x,y:integer;
    begin
        x:=a+1;
        y:=x*2;
        .....
    end p;
end pac;

entity ent is
    port (pin:integer; pout:out bit);
end ent;
```

```
architecture a of ent is
    procedure p(i:in integer) is
    begin
        pac.p;
        .....
        end p;
        signal s:integer := 0;
    begin
        process (s)
            variable v:integer;
        begin
            v:=s-1;
            p(v);
            pout<="1";
            .....
        end process;
        s<=pin+1;
        .....
    end a;
```

The first pass of the compiler does a complete syntactic and semantic analysis of the S'VHDL specification and transforms it into an internal form called program graph. A program graph consists of trees corresponding to the syntactic structure of the statement parts in the program and additional nodes. These nodes represent declared objects on which the statements operate.

The second and third pass of the compiler perform structural transformations on the generated program graph and, at the same time, some high level optimizations. The dataflow description generated by the second pass reflects

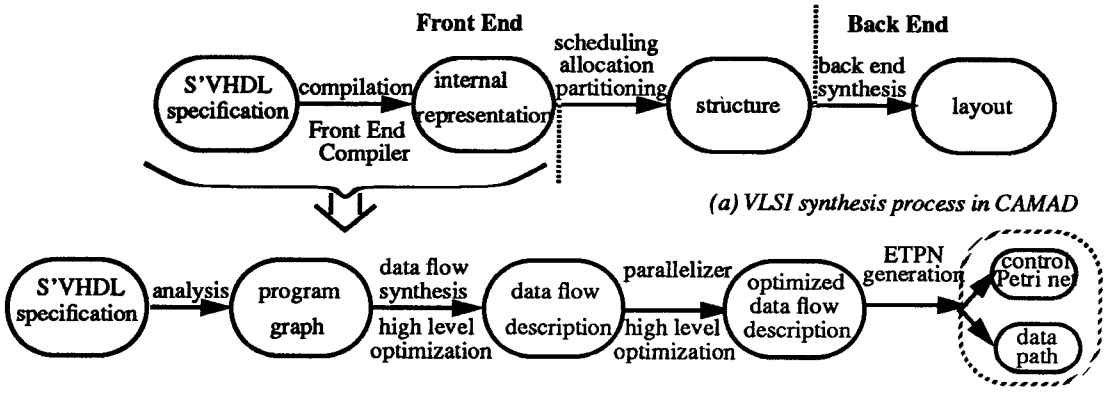

Fig. 5. Overview of CAMAD and the S'VHDL compiler

the flow of data through the program at the level already corresponding to the constructs supported by ETPN. As such, the transformations performed in the second pass are aimed at eliminating some high-level constructs still present in the program graph. One example of such transformation is to expand processes into Petri net loops. Passes two and three perform also, to some degree, extraction of parallelism and high-level optimizations (such as constant folding, eliminating common subexpressions, and code motion).

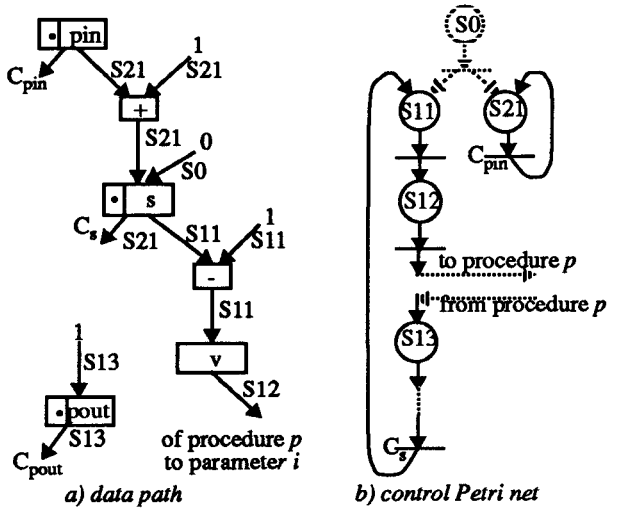

Fig. 6. Example of ETPN generated by the compiler

Finally, the last pass of the compiler transforms the dataflow description into ETPN, generating the corresponding data path and the control Petri net. Figure 6 shows the ETPN representation corresponding to the process and the concurrent signal assignment in the given program example. Some of the solutions discussed in section 4 for wait statements, signal assignments and procedure calls can be recognized here. Note the initial state *S0* in Figure 6 controls initialization of global variables and signals, and provides the initial token to all processes.

6. Conclusions

This paper presents a VHDL compiler for the CAMAD high-level synthesis system. We have described the language subset accepted by the compiler, the internal representation used, and the overall structure of the implementation. Based on the fact that CAMAD supports the design of hardware with concurrency and asynchrony, our VHDL subset includes the concurrent features of the language. We have found that ETPN, the internal representation defined for CAMAD, can be used as a target representation for VHDL constructs. Some important problems and solutions concerning how to deal with signals, wait statements, and subprograms, from the specific point of view of synthesis, are presented.

Certain aspects of VHDL semantics are strictly simulation-oriented and should be redefined or ignored when dealing with synthesis. They include the *after* clause in signal assignments, the time clause in wait statements, and the way signal values are updated after assignments. We have presented several modifications of VHDL semantics in respect to these constructs in the paper, which makes it possible to synthesize them directly into hardware.

Acknowledgments

Peter Eles and Marius Minea are grateful for the financial support from Linköping University. It offered them the opportunity to accomplish this work in the wonderful environment provided by IDA (Dept. of Computer and Information Science) and especially by CADLAB.

References

[1] Benders, L.P.M., Stevens, M.P.J., *Task Level Behavioral Hardware Description*, Proc. Euromicro 91 Hardware and Software Design Automation, 1991; Microprocessing and Microprogramming, Vol. 32, Nr. 1-5, 1991, pp. 323-331.

[2] Benders, L.P.M, Stevens, M.P.J., *Petri Net Modelling of Task Level Behavioral VHDL for VLSI*, Proc. Euro-VHDL'91, 1991, pp. 180-183.

[3] Camposano, R., Saunders, L.F., Tabet, R.M., *VHDL as Input for High-Level Synthesis*, IEEE Design & Test of Computers, March 1991, pp 43-49.

[4] Harper, P., Krolikoski, S., Levia, O., *Using VHDL as a Synthesis Language in the Honeywell VSYNTH System*, in J.A. Darringer, F.J. Rammig (Editors), *Computer Hardware Description Languages and their Applications*, North Holland, 1990, pp 315-330.

[5] *IEEE Standard VHDL Language Reference*, IEEE Std. 1076-1987, IEEE Computer Soc. Press, 1987.

[6] Lis, J.S., Gajski, D., *VHDL Synthesis using Structured Modeling*, Proc. IEEE/ACM Design Automation Conference, 1989, pp 606-609.

[7] Peng, Z., *A formal Methodology for Automated Synthesis of VLSI Systems*, Ph.D. Dissertation, Dept. of Computer and Information Science, Linköping University, No. 170, 1987.

[8] Peng Z., *Semantics of a Parallel Computation Model and its Applications in Digital Hardware Design*, Proc. International Conference on Parallel Processing,1988, pp 69-73.

[9] Peng, Z., Kuchcinski, K., Lyles, B., *CAMAD: a Unified Data Path/Control Synthesis Environment*, in D.A. Edwards (Editor), Design Methodologies for VLSI and Computer Architecture, North-Holland, 1989, pp 53-67.

[10] Postula, A., VHDL Specific Issues in High Level Synthesis, Proc. Euro-VHDL'91, 1991, pp 70-77.