Towards Maximising the use of Structural VHDL for Synthesis

Kevin O'Brien, Anne Robert, Serge Maginot
LEDA S.A.

35, av du Granier, 38042 Meylan, France
tel: (+33) 76 41 92 43 fax: (+33) 76 41 92 44
e-mail: kevin@leda.fr, anne @leda.fr, serge@leda.fr

Abstract

In this paper we show that by performing some VHDL elaboration transformations before synthesis we can extend the synthesis subset to include complex structural and hierarchical statements. This in turn means that:

- *Design, debug and simulation times are reduced*
- *Designs are more accessible (readable, modifiable, portable, reusable)*
- *Design prototyping can be speeded up*

All of this can be achieved without the need to modify existing synthesis tools.

1. Introduction

VHDL has for some time been accepted as one of the more popular description languages for high-level and register transfer level synthesis. This is the case even though no standardised synthesis semantics exist for the language. Unfortunately, the lack of a true standard means that many different *de facto* standards have been developed, especially at the register transfer level. This implies that there is no guarantee that a VHDL description will generate the same hardware on (or even be accepted by) two different synthesis tools. Thus, one of the advantages of using VHDL - the fact that it is a standard - has been lost.

With a few exceptions [1,2,10], it seems that almost everyone is content with the status quo. Circuit designers at the register transfer level adapt their writing styles to the particular tool they are using. This may be at the expense of hardware efficiency. Tool manufacturers also seem quite happy to ignore the front end of their products because changes at this level may have serious repercussions throughout the product.

In high level synthesis, the acceptable VHDL subset has increased but only in specific areas such as describing control flow and synchronisation.

In the rest of this paper, we describe **HELIOS**, a tool that extends the structural VHDL that can be used for synthesis. By doing so, we can:

- Reduce the time required to write, debug and synthesise VHDL descriptions of complex circuits
- Improve the readability, portability and reusability of the resulting description
- Speed up prototyping

These improvements apply equally to high level and register transfer level tools. We also show that, through partial elaboration of the description, the model can be transformed into one accepted by existing synthesis tools. This means that the VHDL subset accepted by such tools can immediately be increased without the need to modify the tools themselves.

2. VHDL Subsets for Synthesis

Figure 1 presents some of the more common behavioural and structural limitations imposed by commercial logic synthesis tools. For each VHDL construct shown, a given VHDL subset will either reject it completely or impose limitations on its use.

Some constructs are rejected because they simply have no synthesis semantics. These include file types or access types, after clauses, wait for clauses and disconnect statements.

Other constructs are rejected or limited due to tool limitations, not because there is no hardware equivalent. For example, real types are usually not allowed due to encoding problems. Multi-dimensional arrays are normally rejected because of the addressing problems they introduce. Loops must have static bounds and processes must also adhere to certain restrictions.

In the structural VHDL world, other restrictions are imposed such as limiting configurations to default configurations, limiting the types of generic parameters, insisting that generate statements only have locally static bounds or conditions, rejecting deferred constants and so on.

With the advent of VHDL-based high-level synthesis tools [5,6,7] and their gradual acceptance by the design community, the length of time required to write and debug

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00

VHDL construct	Status	No synthesis semantics	Tool Limitation	Comments
Real types	Rejected		X	Encoding problems
File types	Rejected	X		May be used in declarative part
Access types	Rejected	X		
N-dim arrays	Rejected		X	Addressing problems
Resolved signals	Limited		X	Catered for by HLS tools
/, mod rem	Limited		X	RHS must be power of 2
exponentiation	Limited		X	LHS must be 2
after/reject clause	Rejected	X		
wait for clause	Rejected	X		
Assert	Ignored	X		Passive
Loops	Limited		X	Limited to locally static bounds
Process	Limited		X	1 synchronisation expression
Deferred constants	Rejected		X	Removed at elaboration
Config decl	Limited		X	Resolved at elaboration
Config spec	Limited		X	Resolved at elaboration
generic	Limited		X	Propagated at elaboration
generate	Limited		X	Unrolled at elaboration

Figure 1: VHDL constructs generally limited by RTL synthesis tools

complex circuits has fallen considerably [8,9]. This is due to the expansion of the VHDL subset for describing circuit behaviour. Most of these tools overcome the behavioural limitations shown in figure 1. In other words, they accept several **wait** statements per process, loops with dynamic bounds, **exit** and **next** statements and so on.

In our work we extend the VHDL *structural* subset. That is, we allow models to contain full configurations, generic parameters of any type, generate statements with globally static bounds, constant declarations initialised by complex and impure expressions etc.

Elaboration transformations are performed on this model to remove the complex structure thereby making the model synthesisable without modifying the synthesis tool.

3. Adding Complex Structure to the VHDL Synthesis Subset

The main transformations that can be carried out by HELIOS are:

- Elaboration of declarative part of a VHDL system
- Execution of configuration specifications
- Propagation of generic parameter values
- Execution of generate statements
- Generation of elaborated VHDL source code

The elaboration of the declarative part consists of verifying typing, assigning an initial value to every object and so on. As such it is more relevant to simulation than to synthesis.

However, we shall see that this part of the elaboration process enables us to take advantage of other powerful but non-synthesisable features such as the TEXTIO package to initialise constant declarations.

In the rest of this paper we will concentrate on showing the benefit of the last three transformations.

3.1 Configurations

Most synthesis tools allow some form of configuration. Indeed, they have to so as components can be bound to entities. However, most tools impose constraints on their use such as only allowing default configurations. In other words, configurations where the component must have the same name and interface as the entity it is instantiating. There is absolutely no reason, from a synthesis point of view, why these constraints should be imposed. In our proposed subset, configurations are fully accepted. Before synthesis, the elaboration performed by HELIOS will flatten the hierarchical configuration maintaining the compatibility with existing synthesis tools.

3.2 Generics and Generates

Configurations themselves are of limited value from the synthesis point of view unless we can combine them with other structural features such as the use of **generic** parameters and **generate** statements. **Generate** statements

are usually accepted with the restriction that the bounds of a **for-generate** statement or the value of the condition of an **if-generate** statement must be locally static (known at compile time). Generics are partially accepted also. Usually their types are quite restricted.

These restrictions are unfortunate because the use of such statements greatly improves the reusability of a VHDL description. What makes these restrictions harder to accept is the fact that both generic parameters and generate statements disappear at elaboration time. Generate statements are flattened (unrolled) and the values of generic parameters are propagated. This in turn means that, rather than restricting the types of generic parameters, why not allow all types, even those that are non-synthesisable. These will be replaced by static values before the start of synthesis.

For example, when describing a ROM for synthesis, the contents normally have to be declared as a constant value within the architecture. This is very cumbersome if the ROM is large, as shown in figure 2. It is also prone to error and difficult and time consuming to modify.

```
entity ROM is
    port ( ADDR   : in      BIT_VECTOR(0 to 15);
           CS     : in      BIT;
           DATA : out STD_LOGIC_VECTOR( 0 to 7));
end ROM;
architecture BEHAVIOUR of ROM is
    type DATA_ELEMENT is BIT_VECTOR(0 to 7);
    type ROM_MATRIX is array(0 to 2**15-1) of
                                    DATA_ELEMENT;
    constant CONTENTS : ROM_MATRIX :=
                ("01010101", "00110011"… --and so on
```

Figure 2: Describing a ROM using synthesisable VHDL

A more convenient way of doing this would be to write the ROM contents to a file (this could even be automatically generated and use a generic parameter to load the file. This not only removes the need to modify the code each time new data are to be stored in the ROM, it also makes the description cleaner and easier to read. The VHDL for doing this is shown in figure 3.

There are a few things worth noting in this example. The first is the use of a string type as a generic parameter. Register transfer level synthesis tools usually only accept integer and enumerated types as generic parameters. This means that we cannot pass commonly-used objects such as BIT_VECTORs without performing type transformation functions.

Although not blocking, this type of unnecessary restriction is nevertheless frustrating and only adds to the complexity of the design.

```
use STD.TEXTIO.all;
package ROM_FUNCTIONS is
    type DATA_ELEMENT is BIT_VECTOR(0 to 7);
    type ROM_MATRIX is array(0 to 2**15-1) of
                                    DATA_ELEMENT;
    impure function ReadFile(FileName : STRING)
        return ROM_MATRIX;
end ROM_FUNCTIONS;
package body ROM_FUNCTIONS is
    impure function ReadFile(FileName : STRING)
                                return ROM_MATRIX is
        file ROMFILE :  TEXT  is FileName;
        variable NEXTLINE : LINE;
        variable RESULT : ROM_MATRIX;
    begin
        for I in 0 to 2**15-1 loop
            READLINE(ROMFILE , NEXTLINE);
            READ(NEXTLINE, RESULT[I]);
        end loop;
        return RESULT;
    end ReadFile;
end ROM_FUNCTIONS;

use WORK.ROM_FUNCTIONS.all;
entity ROM is
    generic (FileName : STRING);
    port ( ADDR  : in    BIT_VECTOR(0 to 15);
           CS    : in    BIT;
           DATA : out STD_LOGIC_VECTOR( 0 to 7));
end ROM;
architecture BEHAVIOUR of ROM is
    constant CONTENTS : ROM_MATRIX :=
                                ReadFile(FileName);
```

Figure 3: Using generics to simplify, generalise and speed up the ROM model description

Another point of interest in this example is the use of TEXTIO functions to initialise the ROM. Normally, this package is completely rejected by synthesis tools at all levels. However, as we can see here, by permitting its use in the declarative region of a design, large improvements can be made in the legibility and generalisation of a design. When HELIOS elaborates the description, the function call used to assign the constant CONTENTS will be executed and the call itself will be replaced by the return value. Thus, the synthesis tool will not even be aware that TEXTIO or an impure function were used.

After elaboration, the description will be similar to that in figure 2.

4. Example

To show the benefit of these statements, suppose we wish to synthesise a system that contains a ROM. To model our ROM, we have a limited library of cells. The only memory cell we have access to is one with a 32K addresses of 8 bit data configuration. We also have other cells capable of representing the address decoding logic.

Our wish is to optimise the ROM, trading off speed and area. The quickest way to do this is by prototyping the system with different address and data sizes and comparing the results. For simplicity, we assume that the data are automatically loaded into whatever ROM configuration we specify.

Using the usual VHDL synthesis subset would mean modifying the ROM description for each new prototype. This would be the case even if the ROM is buried deep in the design's hierarchy (implies a lot of recompilation). For example, suppose our first prototype uses a 64Kx8 configuration. Using our library of functional units, the resulting model would have the structure shown in figure 4.

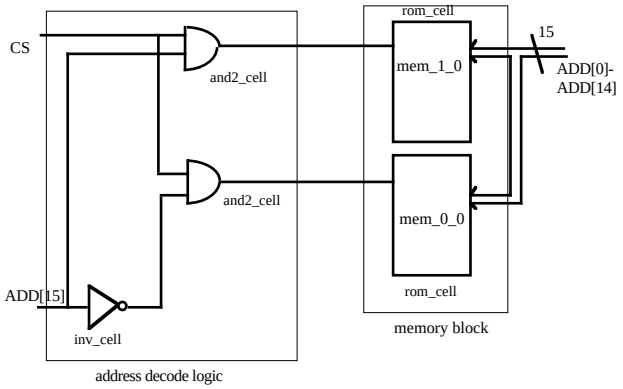

Figure 4: ROM Configuration

The ROM element of our system will have the VHDL description shown in figure 5.

```
entity ROM is
    port (  ADD   : in   BIT_VECTOR(0 to 15);
            CS      : in      BIT;
            DATA : out STD_LOGIC_VECTOR( 0 to 7));
end ROM;
use COMP_LIB.CELL_COMPONENTS.all;
architecture STRUCTURE of ROM is
    for all : inv_cell use entity LIB.INVERTER(BEH);
    for all : and2_cell use entity  LIB.AND2(BEH);
    for all : rom_cell use entity LIB.ROM_32K(BEH);
    signal NOT_ADD15 , CS_1, CS_2 : BIT;
begin
    INV1 : inv_cell  port map (ADD[15], NOT_ADD15 );
    AND2_1 : and2_cell  port map (ADD[15],CS, CS_1);
    AND2_2 : and2_cell
        port map(NOT_ADD15,CS, CS_2);
    MEM1 : rom_cell
        port map (ADD[0 to 14], CS_1, DATA);
    MEM2 : rom_cell
        port map (ADD[0 to 14], CS_2, DATA);
end STRUCTURE;
```

Figure 5: VHDL structural description corresponding to the sub-system of figure 2

This is quite straightforward and more or less what would be performed using today's synthesis tools.

Now we want to try another configuration with a 16 bit data bus. This will necessitate the addition of two more memory cells. Data are split into two cells and the addressing of each pair of cells is identical. This process continues for every prototype tried, an extremely costly approach in terms of time and effort.

If we were only interested in simulating each configuration, we would write a generic model of the ROM, configure it in the top-level unit and simply change the generic parameters for each prototype. This is both faster and cleaner.

With HELIOS, the same approach can now be taken for synthesis. Because of our limited cell library, the structural architecture of our generic model is quite complicated. As we do not know the number of memory cells that will be instantiated at compile time, we must use generate statements to:

- Instantiate a new memory cell for every 32K of memory required
- Instantiate miscellaneous address decoding logic cells
- Instantiate a new memory cell for every 8 bits of data required

The generic model is outlined in figure 6.

```
entity GENERIC_ROM is
    generic (    ADDRESS_BUS_WIDTH : INTEGER;
                 DATA_BUS_WIDTH          : INTEGER);
    port (ADD : in BIT_VECTOR(0 to
                        ADDRESS_BUS_WIDTH - 1);
          CS      : in      BIT;
          DATA :out  STD_LOGIC_VECTOR(0 to
                        DATA_BUS_WIDTH - 1);
end GENERIC_ROM;
use COMP_LIB.CELL_COMPONENTS.all;
architecture STRUCTURE of GENERIC_ROM is
    constant INSTS_FOR_ADDRESS : INTEGER :=
                (ADDRESS_BUS_WIDTH / 15) + 1;
    constant TRAILING_ADD_BITS : INTEGER :=
                ADDRESS_BUS_WIDTH mod 15;
    constant INSTS_FOR_DATA : INTEGER :=
                (DATA_BUS_WIDTH / 8) + 1;

    signal DATA_OVERFLOW:BIT_VECTOR(0 to 7) :=
                (others=>'0');
    signal ADD_OVERFLOW:BIT_VECTOR(0 to 14) :=
                (others=>'0');
begin
--When instantiating memory cells we have to cater for the
--fact that there may be address bus widths that are not a
--multiple of 32K. We store these trailing values in an
--array for each bus and when instantiating memory
--cells for these extra values, we'll map these arrays to the
--ports. If the port is not used, it will have the value '0',
--otherwise it will have the corresponding input value.
```

```
    L1 : if TRAILING_ADD_BITS > 0 generate
        L2 : for I in 0 to 14 generate
            L3 : if TRAILING_ADD_BITS < I generate
                ADD_OVERFLOW(I) <=
                ADD(((INSTS_FOR_ADDRESS-1)*15)+I);
            end generate L3;
        end generate L2;
    end generate L1;
--Begin generating memory cells.

    FullADD : for I in 0 to INSTS_FOR_ADDRESS-1
    generate
        FullDATA : for J in 0 to INSTS_FOR_DATA-1
        generate
--Instantiate memory cells and required decoding logic.
        end generate FullDATA;
    end generate FullADD;
end STRUCTURE;
```

Figure 6: Outline of generic ROM model

As can be seen from figure 6, the generic model associates generic parameter values with generate statements (albeit indirectly through constants). This means that the generate bounds or conditional expressions cannot be evaluated at compile time. However, if we elaborate the entire model, starting with the top level configuration, no generics or generate statements will remain. For example, if the configuration shown in figure 7(a) is elaborated with HELIOS, the ROM description passed to the synthesis or hardware emulation tool will be that shown in figure 7(b).

```
configuration PROTOTYPE_64Kx8 of TOP_ENTITY is
    for TOP_STRUCTURE
        for all : GENERIC_ROM_MODEL use entity
            WORK.GENERIC_ROM(STRUCTURE)
            generic map (
                ADDRESS_BUS_WIDTH => 16;
                DATA_BUS_WIDTH    => 8)
            port map (ADD  => ADDRESS_BUS;
                      CS   => CHIP_SELECT;
                      DATA => DATA_BUS);
        end for;
            …
    end for;
end PROTOTYPE_64Kx8;
```

(a)

```
entity GENERIC_ROM is
    port (  ADD  : in  BIT_VECTOR(0 to 15);
            CS   : in  BIT;
            DATA : out STD_LOGIC_VECTOR(0 to 7);
end GENERIC_ROM;
use COMP_LIB.CELL_COMPONENTS.all;
architecture STRUCTURE of GENERIC_ROM is
    constant INSTS_FOR_ADDRESS : INTEGER := 2;
    constant TRAILING_ADD_BITS : INTEGER := 1;
    constant INSTS_FOR_DATA : INTEGER := 1;
    signal DATA_OVERFLOW : BIT_VECTOR(0 to 7)
        := (others=>'0');
    signal ADD_OVERFLOW : BIT_VECTOR(0 to 14)
        := (others=>'0');
--Signals declared in the generate statements executed
    signal NOT_ADD15  : BIT;
    signal CS_1  : BIT;
    signal CS_2  : BIT;
begin
    ADDRESS_OVERFLOW(0) <=  ADD(15);

--Instantiate memory cells and decoding logic for I=0.
    AND2_1 : and2_cell  port map(
        ADDRESS_OVERFLOW(0),CS, CS_1);
    MEM1 : rom_cell port map (
        ADD[0 to 14], CS_1, DATA);

--Instantiate memory cells and decoding logic for I=1.
    INV1 : inv_cell  port map(
        ADDRESS_OVERFLOW(0), NOT_ADD15 );
    AND2_2 : and2_cell  port map (
        NOT_ADD15 ,CS, CS_2);
    MEM2 : rom_cell port map (
        ADD[0 to 14], CS_2, DATA);
end STRUCTURE;
```

(b)

Figure 7: (a) Example of top level configuration,
(b) ROM description of figure 6 after elaboration of top level configuration.

Figure 7(b) is very similar to the hand-built architecture shown in figure 5. The main difference lies in the fact that this result was generated by elaborating a generic description and is now acceptable by most commercial logic synthesis tools. The use of a generic model means that we can build different ROM models faster and safer.

In summary, using VHDL elaboration as well as other, synthesis-oriented transformations enables us to make full use of the VHDL language when designing large, complex circuits. The design time is reduced as the number of lines of code needed is decreased. The description itself is more readable as the detail concerning a particular configuration is added at elaboration time. The subset extensions shown also vastly improve the prototyping time as it becomes trivial to change a system's configuration. Finally, by using HELIOS the designer no longer needs to maintain different models for simulation and synthesis.

5. Conclusions

We have presented HELIOS, a tool that elaborates VHDL descriptions for synthesis. By performing these transformations, we have shown that system level models can be written clearly in VHDL speeding up design, simulation, debug and prototyping time. In addition, the transformations executed by HELIOS ensure that the resulting model is acceptable to current synthesis tools.

References

[1] IEEE, "IEEE Draft Standard VHDL Synthesis Packages"
IEEE P1076.3-199X, 1995

[2] Villar E, "The Level 0 VHDL Synthesis and Semantics, Parts 1 and 2", **VHDL Newsletter #19 and #20**, 1995.

[3] Airiau R, Bergé J-M, Olive V, "Circuit Synthesis with VHDL", Kluwer Academic Publishers, 1994.

[4] Postula A, "VHDL Specific Issues in High Level Synthesis", in Mermet J (ed), **VHDL for Simulation, Synthesis and Formal Proofs of Hardware**, Kluwer Academic Publishers, 1992.

[5] Kission P, Ding H, Jerraya A, "Structured Design Methodology for High-Level Design", **Proc. 31st DAC**, pp466-471, San Francisco, June 1994.

[6] Synthesia "The Synthesis VHDL Design System - SYNT 1.5 User's Guide", Synthesia AB, 1995.

[7] Gajski D, Dutt N, Wu A, Lin S, "High-Level Synthesis: Introduction to Chip and System Design", Kluwer Academic Publishers, 1991.

[8] Bourbon B, "On System-Level Design", **Computer Design**, December 1990.

[9] O'Brien K, "Compilation de Silicium : du Circuit au Système", Ph.D dissertation, INPG, Grenoble, France, March 1993.

[10] IEEE, "IEEE Draft Standard VITAL ASIC Modeling Specification" I**EEE P1076.4-199X**, 1995