VHDL Toolkit: An Abstraction Beyond Logic Synthesis

Shridhar K. Mukund and Jayaram Kalpathy
Cirrus Logic, Inc.
Fremont, CA 94538

Abstract

Logic synthesis and optimization derive their strength from assuming a restrictive model of the logic design world. When "tamed" and applied to parts of the chip design problem, they do significantly improve productivity. However, any design of reasonable complexity necessitates the application of "design knowledge" beyond the ability of logic synthesis and optimization.

We have developed a set of VHDL packages called "VHDL Toolkit" that lead to a very expressive and technology transparent abstraction for design capture in VHDL. Here technology transparency does not merely mean the ability to target to the synthesizable subset of any cellset, but it includes 1) the ability of controlled mapping to other context specific efficient combinational and storage cells, 2) the ability to capture data flow for data path compilation to a layout technology that benefits from it, 3) the ability to capture asynchronous parts and "logic tricks", 4) the ability to partition design parts that need to be implemented at the layout level and more. In this paper, we delineate the need for an expressive abstraction, illustrate the benefits with examples and discuss the language and the tool mechanisms that make such an abstraction possible.

1.0 Introduction

The merit of any design entry medium is simply the efficiency with which the design can be "human compiled", or in other words, it's goodness as a document. It is not surprising that the primary intent of VHDL was design documentation, and that simulation and synthesis are only incidental. In the real world, simulation efficiency and synthesizability are very crucial. However, in our hurry to "tape out" another chip, let us not forget that human element and take a productivity hit in the long run.

Like any decent programming language, VHDL offers a medium for building layers of abstractions such that the application layer is expressive and efficient for a given application; digital design in our case.

An analogy: When the C language was introduced, we would probably have written the following piece of "procedural" code to determine the length of a string:

```c
for( len = 0, p = name; *p != '\0'; p++) len++;
```

As we realized that this was a frequently required task, soon a library of string functions was devised and the code started looking like:

```c
len = strlen( name );
```

Well, now a programmer could get away even without knowing the concept of a pointer in C. As time went by and graphical user interface(GUI) became a popular application, another abstraction called
X Toolkit was devised. Well, now the GUI developer can "think" windows through C!

The above analogy sets the tone for the discussion to follow: an abstraction to let the logic designer think design through VHDL. The area of interest and the context of this paper, Synthesizable VHDL, is relatively new and has not taken enough "beating" to learn and refine from practice.

In section 2.0, we recapitulate some mechanisms provided in the language for design abstraction, including one that is seemingly unexploited. We then discuss the "implication" capability of the synthesis tool and generalize its use in a way as to create a dimension for abstraction.

In section 3.0, we illustrate how the implication capability can be used to capture the design in an expressive data flow style for parts, where procedural abstraction simply does not apply, and also capture "design knowledge" for implementation control.

In section 4.0, we discuss some of the synthesis tool issues that could further the cause of technology transparent design capture and controlled refinement to an implementation.

### 2.0 Synthetic Implementation

#### 2.1 The Data Flow Abstraction

Component instantiation is a very useful mechanism for abstraction. At the top levels of the design, where the design is thought of as a block diagram of interconnected complex modules, structural abstraction works very well. Procedural thinking applies well at the lower most level, namely RTL components. Procedural thinking also applies well at inter-

*. Throughout this paper the synthesis tool in question is that of Synopsys. The "implication" capability of the tool is crucial to the discussion. However, since the need for such a capability is a direct result of the nature of VHDL, any reasonable VHDL synthesis tool should have a similar capability in some form.

mediate levels for certain types of task, like state machine design and sometimes more in specific design domains, like digital signal processing. On the other hand, data flow abstraction spans the continuum simply because it is akin to digital design. One can always fall back on the data flow abstraction for addressing any design problem. Stylization requirements of the synthesis tools may further force the designer in thinking data flow, to be able to write synthesizable code.

While components could be among the building blocks of the data flow, the ones that contribute most to the expressiveness are operators, functions and procedures.

**An expression is data flow**: The beautiful thing about VHDL is that it does not treat an operator any different from a function. In fact every instance of an operator maps to a corresponding function; for example, the operator '+' maps to the function "+"(). The user has as much control in defining the functionality corresponding to an operator as he/she has on user defined functions and procedures. Hence any reasonable VHDL synthesis tool would only translate an expression into a data flow graph, the building blocks of which are defined only by the user.

Given that the building blocks are user visible, if the user had the additional capability of replacing these building blocks with equivalent logic implementations then he/she literally has the capability of synthesizing the data flow to "a" logic implementation, ready for optimization.

#### 2.2 Persistent Concurrent Procedure

Concurrent procedure calls make it possible to declare procedures representing commonly used processes and easily create such processes by merely calling the procedure as a concurrent statement. Such a procedure will persist over time (and thus the variables will retain state over time) if it's outermost statement is a
-- start and stop are synchronous with the rising edge of clk
-- start takes the ring counter to the state 00...01 and stop to 00...00
-- else the state bit pattern rotates left by one bit at every rising edge of clk
procedure ring_count ( signal clk, start, stop: in std_logic; signal q: out std_logic_vector ) is
  variable state : std_logic_vector ( q'length -1 downto 0 );
begin
  loop begin
    -- simulatable behavior goes here
    q <= state;
    wait on clk; -- the sensitivity list
  end loop;
end;

**FIGURE 1.** An example of a persistant concurrent procedure

loop statement, and the loop contains a wait statement[2].

Figure 1. describes a procedure equivalent of a ring counter. The procedural behavior is encapsulated in a loop statement. At the end of the behavior is the wait statement that determines the sensitivity of the equivalent process. It has access to external signals through the signal arguments.

Using this feature of the language one could define a procedure for any RTL object that cannot be described either as an operator, a function or a sequential procedure. These could be all sorts of commonly used building blocks like counters, FIFO, stack, flops, latches and even RAM, if you may. In essence, one could create a whole set of libraries of reusable design parts in the form of sub-programs.

To a designer, the procedural description of the ring counter may seem a little cryptic. But then given the interface of the library procedure, all the designer has to deal with is the expressiveness of the application VHDL. Figure 2. illustrates a call to ring_count at the application layer. Notice that the ring_count is implicitly parameterized, i.e. the width of the ring counter is determined by the width of the bus signal, 'slot_enable'.

Strangely enough this capability seems to be under exploited by the design community, probably because of the confusion caused by the restrictive stylization requirements of HDL synthesis tools.

### 2.3 Synthetic Modules

#### 2.3.1 The need for synthetic modules

The significant part of logic optimization is combinational logic optimization, where

```vhdl
signal slot_enable : std_logic_vector (7 downto 0 );
...
arbiter : ring_count ( clk => slot_clk, start => restart, stop => suspend, q => slot_enable );
```

**FIGURE 2.** A call to the persistant concurrent procedure
"an" implementation of a combinational logic block is translated to a canonical form and then algorithmically transformed to a "supposedly optimal" logic implementation. However elegant it may seem, state-of-the-art technology mapping algorithms are not capable of mapping to certain types of cells, probably because the exploration space is far too large for algorithmic search. Good examples of the above are the frequently used arithmetic and relational operations. For example the \texttt{}``\texttt{*}``\texttt{}`` operation, when used to represent an arithmetic adder, is best implemented using full adder cells in one of several ways in a trade off between speed and area.

A quick solution to the problem is to let the synthesis tool replace the arithmetic and relational building blocks in the data flow of an expression with corresponding well-known implementations. Of course, logic optimization can then be applied to rest of the combinational logic block and probably arithmetic optimization to the data flow. In essence, this defines a mechanism for capturing "design knowledge" into the synthesis system in such a way that one can still reap significant benefits from algorithmic optimization techniques.

2.3.2 The implication mechanism
To reiterate, VHDL does not define the functionality corresponding to an operator and it can be overloaded to be context sensitive. An operator can map to one of several functions within the same package, overloaded on the operand types. Across packages, it can once again be overloaded to mean something different, for example all the operators in a special package could correspond to some finite field operations. Hence the synthesis tool by itself cannot presume implementations corresponding to operators. Instead, it has to provide hooks for the user to "imply" a set of external implementations corresponding to the functions that an operator maps to.

Figure 3. illustrates a mechanism for implying a parameterized entity through an instance of an operator in the application VHDL\cite{1}. The function corresponding to the particular instance of \texttt{}``\texttt{*}``\texttt{}`` has a synthesis directive which points indirectly to the entity \texttt{tk_add}. In the process, the required parameters are derived. The entity could have several implementations to suit the problem. The synthesis tool can make the choice of the implementation or the designer can explicitly exercise control.

A synthetic module is simply a parameterized entity, typically implied through such a mechanism.

2.3.3 The generalization
We generalize this capability for operators, in a straightforward way, to "any" user defined function or procedure. The mechanism that works for operators and functions which are essentially combinational, then applies to any subprogram, synchronous, asynchronous or what have you. We could now have the toolkit to support commonly used operators and RTL objects in the data flow abstraction to imply synthetic modules.

3.0 The Designer Abstraction

3.1 VHDL Toolkit
The VHDL toolkit comprises of a set of VHDL packages that define overloaded operators, functions and procedures that relate well with the world of logic design. These subprograms obviously have simulatable behavior, and hence the application VHDL code along with these packages are simulatable by any VHDL simulator.

The library system also carries the synthetic modules that the subprograms imply. Since every synthetic module has at least one architecture that has synthesizable behavior, we are set for logic syn-

---

* Synopsys-DesignWare Developer provides a detailed mechanism for synthetic module implication for operators. For the sake of brevity, our discussion here is limited to a very simplistic view.
thesis and optimization. Of course, it is technology independent.

To support efficient mapping of arithmetic and relational operators, the corresponding synthetic modules also have several "tool specific", but technology independent implementations.

In addition, the design community using the toolkit can now define technology specific implementations, for several of the synthetic modules to get control on several aspects of design refinement to logic, as discussed later. The important thing to note here is that the library has a relatively small set of reusable parts under centralized control. Migration to a new technology is automatic, but if refinement capabilities beyond the scope of logic synthesis and optimization are desired then some amount of library enhancement may be required. The library enhancement can be graceful and on a need basis. Application VHDL does not change and the vast number of user designers are completely spared of all the trouble.

3.2 The Register Inference Problem

Commercial HDL synthesis tools have significant limitations in inferring registers from procedural VHDL, which are politely called "synthesis policies".

In fact, it is strongly recommended that one should restrict each process to a single type of memory-element inferring: latch, latch with asynchronous set, flop etc[3]. As a very direct result, the designer is expected to visualize the general structure of the circuit, such as where the flip-flops go and what type of reset logic is required[4]. In essence, the designer is

---

*. Synopsys defines a set of synthetic modules in DesignWare that are built using a generic technology called the GTECH library. As long as the tool knows how to map to equivalent cells in the target technology, the mechanism is technology independent.
forced to think RTL or at best data flow, where registers are involved.

Having thought out the register, the designer is then required to "encipher" his/her thought into a "template" that the tool will supposedly "infer" from. A template for a flip-flop with reset may look like:

```vhdl
process ( clk, r_n )
begin
  if r_n = '0' then
    a <= '0000000000000000';
  elsif clk'event and clk = '1'
    a <= b;
  end if;
end process;
```

It is a template because the synthesis tool expects a well-defined form for something whose behavior can otherwise be described in very many ways. Since the designer thought out a flop with a reset, there is no inference here, it is simply another form of implication. It is a cipher because the description is relatively complex for the simple thought: "flop with a reset". It could as well have been a concurrent procedure call like:

```vhdl
flop ( clk=>clk, reset_n=>r_n, d=>a, q=>b );
```

Note that the width of the flop is implicit in the width of a and b.

In a desperate attempt to reduce the number of lines in the code, one could imply several registers in a template. What happens to the abstraction? Do we want to think out several disconnected registers in the data flow and stuff them into one template?

Further, there are only so many templates that a synthesis tool provides. What if the designer wanted a tristatable register and actually wanted to map to such cells, because a particular design significantly benefits from it? Well, how about a concurrent procedure like:

```vhdl
regist ( wr=>wr, re=>re, din=>a, dout=>b );
```

The corresponding synthetic module, in addition to synthesizable behavior implementation will now have a special implementation that uses those cells instead.

For efficient synthesis, state machines need to be described in the Mealy or Moore form. The designer once again may have to think out the register, the next-state logic and the output-logic. Most often, next-state logic and output-logic are good candidates for procedural description in VHDL and so are many other large combinational logic blocks.

Synthesis tools could have different synthesis policies with respect to register inference. With the synthetic module approach, since the knowledge of how a register is "inferred" by "a" synthesis tool is packaged into the library, the application VHDL is now independent of the synthesis tool! The library can be ported to support a new synthesis tool with little effort.

3.3 What Beyond Logic Synthesis?

To explore the capabilities of this dimension of abstraction, consider the VHDL code fragment in figure 4. The data flow is described in concurrent VHDL. The function `multiplex()` selects the bus argument 'if0', if the condition 'cond' is evaluated to 0 and the bus argument 'if1' otherwise. The function `transmit()` returns the bus argument 't', if en is asserted, else it returns "Z...ZZ". The instance of "xor" operator here is overloaded to be a scalar-vector logic operation that performs the equivalent of:

```vhdl
b xor ( cmpl, cmpl, ..., cmpl )
```

It is a very useful operation and applies to other logic operators such as "and", "or" etc.

3.3.1 A mix of abstractions

Concurrent VHDL statements are sequence independent and the only motivation for a general top-down data flow sequence is for readability. Just like a designer may want to organize schematic
... 

signal a, b, c : std_logic_vector ( 15 downto 0 );
signal en, cs, cmpl, oe_n : std_logic;
...

my datapath : block

signal x, y : std_logic_vector ( 15 downto 0 );
begin

x <= multiplex ( cond => en or cs, if0 => a + b, if1 => b xor cmpl );
my_flop : flop ( clk => my_clk, d => x, q => y );
c <= transmit ( en => not oe_n, i => y );
end block my datapath;
...

FIGURE 4. VHDL code fragment written in the data flow style

data flow from left to right. As we said earlier, different types of abstractions, namely structural, procedural and data flow, all have their importance in some context. In the example above, we might have a state machine described in procedural VHDL right on top of the block, my datapath and may be we have a component instantiation of a ROM below the block statement. The key is in mixing the abstractions to maximize readability.

Given the constraints of synthesizability, the most expressive abstraction is simply what ever is closest to the designer’s thought. In other words, if you think structural then write structural, if you think data flow then write data flow and if you think procedural then write procedural.

Entities are hard partitions, typically used to reflect partitions in the design specification, to abstract large reusable design parts or sometimes to break up a very large design to reduce complexity, there by splitting the design task. Blocks on the other hand are soft partitions that come in handy for quickly manipulating design partitions. Signals that are used only in a particular block can be made local to that block, there by making the architecture more readable. Further, functions, procedures and processes themselves define design hierarchies under the synthesis tool control.

3.3.2 The graphical abstraction

Graphical abstraction is often a good medium for design documentation. The merits of HDL design capture has been questioned time and again[5].

Graphical abstraction can range anywhere from brute force gate level schematics to a hierarchical schematic system with a rich set of libraries of RTL and higher level components such as bubble diagrams for state machines. Similarly VHDL description can range from the netlist form, to brute force machine generated synthesizable VHDL, to a higher level of abstraction supported with a rich set of libraries of RTL functions, procedures, large reusable design parts and procedural abstraction for the likes of state machines.

The comparisons are solely based on the availability of commercial tools, while actually both the mediums are at best equally expressive. The biggest factor actually being the libraries and the library support system. Whether VHDL should be machine generated from a graphical
abstraction or vice versa is only incidental in the choice of the tool.

Figure 5. depicts the schematic generated by elaborating the code segment in figure 4. The block encapsulation is used to quickly define a hierarchy, hence the design my_datapath. The hierarchical modules TK_ADD, TK_CXOR, TK_MUX2, TK_FLOP and TK_TRANSMIT are synthetic modules that are simply placeholders for one of several implementations to be refined later. Rest of the design is synthesized to logic equivalent, hence the inverter and the OR gates. The fact that makes it expressive is simply the nice set of graphic symbols associated with the synthetic modules in the library. Machine generated schematics are quite expressive when there are fewer elements on a page, which is the case with synthetic modules. The interface of the design is automatically generated. A little change in VHDL will reflect instantaneously in the schematics. Most of all, the whole design can be quickly reorganized into another set of schematic pages by simply moving the concurrent statements and blocks statements around. The schematic pages for synthetic modules are also machine generated, but for most part are not even required for documentation.

A picture of a rose or Mona Lisa is very readable and expressive if you may[5], but how many persons have the talent to capture them? In other words, the expressiveness of manually generated schematics is only as good as the artistic talent of the designer and the library of building blocks. It is a lot of work to make refinements, especially across schematic pages. A well-sketched bubble diagram is only as expressive as a well-written procedural description of a state machine.

The big idea behind synthesizable VHDL is to define an industry standard portable design capture medium, and tools come into play only for refinement to an implementation. Graphical tools that help in organizing VHDL design capture, especially with the structural aspects, are welcome. But schematic based tools that generate cryptic synthesizable VHDL could as well generate netlists in some generic technology and still be technology independent.

3.3.3 The Interpretation problem
Logic synthesis refines VHDL code to an equivalent logic implementation, which need not necessarily be optimal. Logic optimization further refines it to a supposedly optimal logic implementation.
process ( rst_n , clk )
    variable cnt : std_logic_vector ( 7 downto 0 );
begin
    wait until clk'event and clk = '1';
    if rst_n = '0' then
        cnt := "00000000" ;
    else
        cnt := cnt + 1 ;
    end if ;
    x_en <= decode ( cnt , "−10010−" );
end process ;

**FIGURE 6.** Potential interpretation problem during logic synthesis

Like we discussed earlier, where register inferring is concerned, logic synthesis is subject to interpretation and can potentially generate implementation not equivalent to the behavior. Consider the example in figure 6. In addition to the eight flops required for the counter, the synthesis tool may produce an additional flop for the 'x_en' assignment, which is inconsistent with the behavior. Here decode() is an utility function that returns '1' when 'cnt' takes a particular value. Further, the synthesis tool may altogether ignore the sensitivity list. If logic synthesis does not produce equivalent implementation, it renders behavioral simulation effort useless and can cause very serious verification problems. In the data flow abstraction this could have been described using the flop() procedure or it could as well be:

```plaintext
count ( clk => clk , reset_n => rst_n , q => cnt );
```

```plaintext
x_en <= decode ( cnt , "−10010−" );
```

Using the implication technique, we can abstract out the error prone, dirty mechanics of register inference from the world of the logic designer.

### 3.3.4 The implementation space

We could potentially take the data flow implementation in figure 5, and let the synthesis tool synthesize the synthesizable behavior implementations of the synthetic modules, thereby producing a logic implementation, ready for optimization.

The paradigm of logic optimization is algorithmic and is based on certain assumptions. Depending upon the trade-off requirements in the design situation, it may not always generate good results. We have already seen the problem of optimizing arithmetic and relational operators. In fact, the primary limitation comes from the fact that technology mapper may not like multiple output combinational cells. Which means that in our example, we will have to let the tool pick up one of the better structural implementations for the adder and optimize rest of the logic around it. In fact, quite often there are very efficient technology dependant implementations for such operators. Further, synthesis tools support arithmetic/algebraic optimization at the data flow level, that could be made use of.

**Where are my muxes?** Unbuffered multiplexer cell is a favorite one among CMOS designers because it is small and fast, when used with caution. Ironically, technology mappers are biased towards certain types of combinational cells. In fact, one can start with a good implementation
3.3.8 Routability
Logic optimization assumes that layout area is proportional to the gate count. While it is a good first order approximation, it can lead to less routable logic, especially when optimizing large circuits and data path portions of the design. Very often, retaining the general structure of the data flow parts and functional partitions during optimization leads to better implementation. The designer can start off with the synthetic implementation and explore the implementation space by performing logic optimization, arithmetic/algebraic optimization, resource sharing etc. to parts of the design.

3.3.9 Global implementation control
Parameterized module generation along with the implication capability provides a mechanism for capturing all sorts of design knowledge into the synthesis environment, only some of which could be discussed in this section. Depending upon the design situation, one may be more important than the other. Modularization is also a way of automatically tagging generic design parts. Interactive mechanisms provided by the synthesis tool can then be used for querying and making incremental refinements to the design.

3.3.10 Hardware composition rules apply
Looking at the behavioral aspect of a signal, it could be represented either by a wire or a latch[6]. The fact that the behavior of registers are packaged away in a library and that the library elements are coded by VHDL users expected to be more than novice, signals in effect are abstracted to look like wires in the designers world. Of course, a significant contribution comes from the resolved std_logic_1164 type, which captures the notion of tristatability of wires.

If the library elements truly model the behavior of their equivalent hardware building blocks, then the designer can compose the design in exactly the same way it is done in the logic design world. While this may not sound like a big deal to the community of "VHDL modelers" or VHDL simulation guru's, it means a lot to logic designers whether novice or experienced in using VHDL. Consider for example the puzzle of how to "model" a bi-directional bus[4]. In synthesizable procedural VHDL, one may need to write as follows:

```vhdl
if ( data_enb = '1' ) then
  data <= int_data;
  int_data <= "ZZZZZZZZ"
else
  int_data <= data;
  data <= "ZZZZZZZZ"
end if;
```

In the logic design world, a bi-directional bus can be created simply by connecting tristatable buffers back-to-back. For example consider the following hardware composition:

```
  a[7:0]  b[7:0]
     |      |
     |      |
    wr     re
```

Going by the same composition rule, it can be "captured" in concurrent VHDL as:
```
signal a, b : std_logic_vector ( 7 downto 0 );
signal we, re : std_logic;
...
  a <= transmit ( en => wr, i => b );
  b <= transmit ( en => re, i => a );
...
```

In the above example, if 'wr' and 're' are asserted simultaneously, it will result in logic collision in hardware, exactly the
same way the std_logic resolution function will complain during simulation.

With the data flow abstraction, logic designers can gracefully migrate from the RTL design world without having to learn complex simulation semantics of procedural VHDL, synthesis policies and the specialized art of writing synthesis smart VHDL.

3.4 The Myth

Register/tristate inference and other stylization limitations of synthesis tools typically result in VHDL code that consist of lots and lots of little fragmented processes. The fact that all the lines are stuffed into processes does not constitute procedural behavior. On the contrary, since the predominant inter process communication is data flow, in aggregate, it is still a data flow abstraction. In fact, one can decompose every concurrent statement in the data flow style, to an equivalent process to get the same apparent effect.

Decomposing "natural" building blocks into processes is as criminal as C programming without modularization and libraries.

3.5 Library Engineering Issues

Since the library is intended for use by many users, care has to be exercised to ensure that it is effective in its purpose and robust at the same time. Although a detailed discussion is out of scope, it deserves some treatment at least because the number of lines in the library code for error checking and robustness are typically more than that for the behavior itself.

As far as possible, subprogram and argument names need to relate well with the logic design context. Overloading can be used to advantage for subprograms of similar functionality, but different numbers and types of arguments. The multiplex() operation could be 2-to-1, 4-to-1 or for that matter even 3-to-1 and more. The regist() operation could either be a single tristatable register or a whole register file depending on whether the control signals are bit wide or bus wide. One could have several flavors of latch() and flop() etc., as long as it does not make the application VHDL look cryptic. It is a good practice to use explicit argument association to be precise and readable.

To prevent the designer from making usage errors, the subprograms should have elaborate static checks. For example, the bus arguments ‘[0]’ and ‘[1]’ of the multiplex() function need to be checked for equal lengths. Very often, run-time checks need to be performed because the underlying implementation may dictate certain constraints on the input. The ring_count() procedure in figure 1 may need to check against simultaneous ‘start’ and ‘stop’ assertion.

Since the inner most behavior of all the RTL elements are packaged in the library, we have good centralized control on their simulation models. Elaborate checks for violations can be made and can be quickly changed to suit the conditions. The behavioral models need to be as simulation efficient as possible. For the sake of simplicity std_logic type has been used in all the examples. In several places, one may have to use std_ulogic type.

3.6 Verification

A significant component of synthesis validation, is the verification of alternative implementations of synthetic modules against the corresponding simulation behavior. Typically, synthetic modules are parameterized. Hence smaller test cases can be generated and even exhaustive simulation is feasible. Automating this level of verification into the library system is crucial to quality.
4.0 Some thoughts

4.1 VHDL Issue
As per the proposal VHDL92-DR-6, one could associate an arbitrary expression to the port of type IN of an instantiated component[7]. For example:

regcell port map ( we => reg_en and clk, ...)

This significantly enhances readability. On similar lines, the language could let an expression be associated with an IN argument of a subprogram whose formal is declared to be of type SIGNAL. Then one could say:

my_reg : regist ( we => reg_en and clk, ...)

Of course, this means that there is an implicit signal from the simulation perspective.

4.2 Synthesis Issues
In the interest of the design community there is surely a need for standardization on how synthesis tools implement the design refinement step of mapping subprograms to synthetic modules.

Synthesis tools should support the capability of associating graphic symbols with user defined synthetic modules and generate pre-optimization schematics where word operations are preserved. Apart from algorithmic optimization capabilities, it is very important that synthesis tools provide interactive capabilities for querying the design and making incremental refinements to the synthetic implementation.

Although multiplexers, tri-state drivers, latches, flip-flops etc., are special cases of conditional and guarded signal assignments, there is significant advantage in recognizing and mapping these constructs to synthetic modules before optimization to gates.

VHDL provides a mechanism for labeling a statement. There is no reason why a synthesis tool should not use these labels as instance names or instance name prefixes for the synthetic modules and logic components generated by that statement. Names are often important for navigation.

Synthesis tools should provide the ability of passing user-defined VHDL attributes to the hierarchical VHDL netlist, or maybe in the form of properties in the industry standard EDIF format. Significant amount of knowledge can then be passed from the VHDL libraries down to the back-end layout synthesis and place and route tools.

The success of any design capture medium is largely based on the availability of libraries, whether these are simple utility functions or large reusable design parts. VHDL library vendors can help enhance productivity by letting the designers get away from low level design using raw VHDL. The use of VHDL in synthesis involves defining subsets with specific semantics and a set of conventions. Until there is a standard, it helps a lot to keep the application VHDL, "synthesis-tool"-independent. Let the complex simulation semantics of the language reside in libraries and simulation environments where they belong.

4.3 In The Future
When synthesis tools mature and high level synthesis along with related verification technology becomes a commercial reality, it will be reasonable to write several parts of the design in procedural VHDL, especially in certain domains like digital signal processing. High level synthesis tools will generate data flow refinement; the components of which will still be the very same synthetic modules. To be realistic, it will take quite some time before the design community switches from data flow thinking to procedural thinking for larger parts of the design.

Bottom line is that the best abstraction is the one that is closest to the thought process of an average designer. An expressive data flow abstraction significantly helps in making the paradigm shift, graceful, with-
out having to compromise with the current inadequacies of synthesis tools.

After all, the ability to write procedural abstraction need not be the primary motivation for switching to VHDL. When it comes to design capture, it can be as good as any other medium, plus it is also a medium for simulation, module generation, interchange and what have you. The fact that it is an emerging industry standard allows us to reap productivity benefits from off-the-shelf tools for simulation, logic optimization, arithmetic optimization, register re-timing, and in the future, higher levels of optimization through high level synthesis.

5.0 Conclusion

By exploiting the use of subprograms as building blocks, we are able to address the shortcomings of logic synthesis and optimization where capturing "design knowledge" is concerned, providing true technology transparency and most of all, an expressive data flow abstraction for design capture in synthesizable VHDL.

Acknowledgments

The authors would like to thank Oscar Saldhana, Jeff Mitchem and Bob Alvin for helpful suggestions and discussions.

References