Module generation for VHDL synthesis

Rob Dekker and Michiel Ligthart
Exemplar Logic
2550 Ninth Street, suite 101, Berkeley, CA. 94710
(510) 849 0937
rob@exemplar.com

Abstract

VHDL does not allow component overloading of operators, or component instantiation in any function or procedure for that matter. As a result, when synthesizing operators in VHDL that should imply structured logic, a designer has the limited choice between component instantiation in the dataflow, or relying on the default implementation provided by the synthesis tool. This paper presents a methodology that allows VHDL designers to utilize technology specific macros, while maintaining a technology independent design style.

Introduction

VHDL, although originally intended as a simulation language, has found its way into the synthesis world as a viable RTL language for digital design specification. Mainly used for mask programmable gate arrays, it is now also increasingly used for FPGA design starts. Because of the variety of basic building blocks in FPGAs, technology dependent optimization is essential in FPGA synthesis. For random logic, this goal is achieved by implementing technology specific optimization algorithms in the synthesis tool. For structured logic synthesis, however, a flexible mechanism to implement arithmetic and relational operations in a technology dependent way is highly desirable.

FPGA vendors supply such implementations in pre-defined hard and soft macros. From a VHDL perspective, these macros can always be utilized using component instantiation with hardwired connectivity. However, this removes the higher level of abstraction that the HDL was supposed to provide in the first place and essentially reduces HDL design to textual schematic capture.

This paper is organized as follows. We first establish the need for technology dependent structured logic synthesis in FPGAs, and discuss the related problems when using VHDL. The section on module generation presents our solution to this problem, while the remaining sections discuss related issues to this solution. We finally present some results of VHDL designs synthesized with and without module generation.

FPGAs

Field Programmable Gate Arrays (FPGAs), contrary to traditional gate arrays, comprise a wide variety of building blocks, or cells. Xilinx' Configurable Logic Block (CLB, see figure 1), for instance, contains programmable Look-Up Tables and storage registers. The 4000 architecture implements any function of up to 5 inputs, and some functions of up to 9 inputs. In this sense, a 5-input XOR function is as expensive in area and delay as a 2-input NAND. This is in contrast with ASIC transistor counts when implementing these same

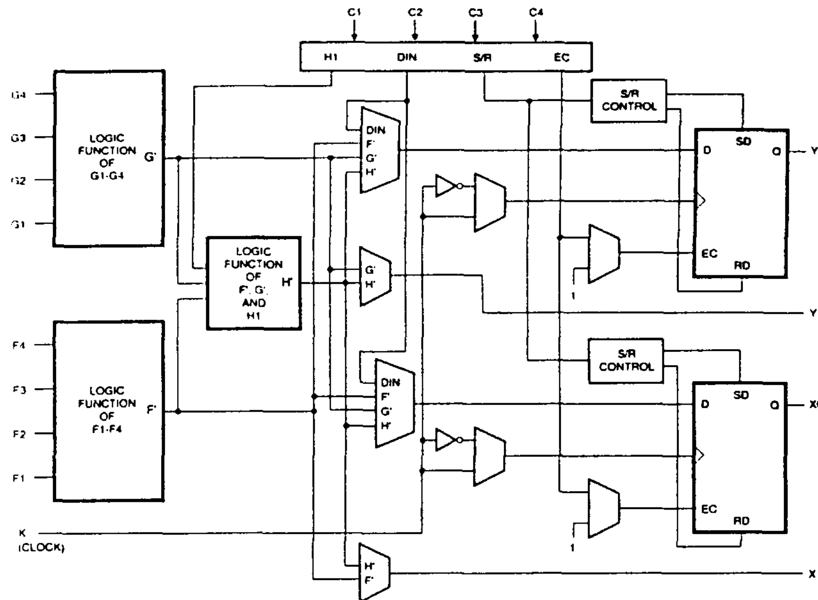

Figure 1. Simplified Block Diagram of XC4000 Configurable Logic Block.

functions.

The Actel technology, on the other hand, has a completely different basic cell (see figure 2). Based on a multiplexor architecture, the Act2 c-module implements as diverse gates as 2-input AND gates, as well as AND-OR gates with 5 inputs with the same area and delay penalty.

As a final example, the Altera MAX family (figure 3) is representative of the so-called complex PLDs. The basic structure here is the Logic Array Block (LAB), which consist of a macrocell array, an expander product term array, and an i/o control block. The LAB implements 16 functions of up to 30 product terms.

Additionally, most FPGA architectures come with a variety of pre-defined technology specific hard and soft macros that can be utilized by advanced synthesis systems. (Hard macros differ from soft macros in the fact that they are pre laid out and routed, sometimes in different aspect ratios.) Hard and soft macros are especially popular to implement arithmetic and relational logic, such as addition, subtraction, incrementation, and comparisons.

Technology specific synthesis

The differences in cell granularity among different FPGAs make it a challengeboth for human designers andfor automated synthesis tools to design circuits in an optimal fashion for every architecture. For random logic, synthesis can use architecture specific optimization algorithms. To stick with the three technologies discussed before, one can use fanin-limited optimization for Xilinx, mux-based optimization for Actel, and cube-limited optimization for Altera [1]. These optimization techniques, however, operate only on the Boolean gate level, and are not capable of resynthesizing larger structures, like n-bit adders, as a structure in itself. In fact, once the n-bit adder is implemented in a certain way, no synthesis tool will change its basic structure, say from carry ripple into carry look ahead.

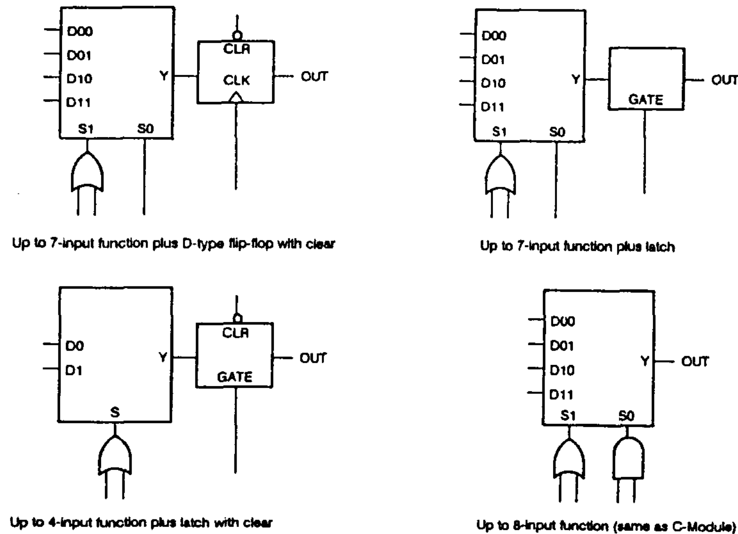

Figure 2. Combinational and sequential configurations of the Act2 cell.

Structured logic synthesis

From the discussion above, it is obvious that the way one implements structured logic functionality in an FPGA differs drastically from technology to technology. As an example, 16 bit addition in the Xilinx 4000 family is best implemented using hard macros exploiting the architectures fast carry chain, but these hard macros do not exist in the 3000/3100 family. Actel, on the other hand, has a vast variety of full-adder cells that can be configured in carry-look ahead, or ripple carry adders.

Not only target architecture, but area versus delay trade-offs can also play an important role in how to implement structured logic. To stay with the adder example, a 16-bit implementation in Act1 can be done in 24 modules as a carry-ripple, or in 78 modules as a carry-select adder for approximately half the delay.

In a schematic entry methodology, the designer can always utilize the foundry provided hard and soft macros by interconnecting them in the schematic drawing. He or she can even design proprietary macros and use those whenever appropriate. The drawback, besides using the schematic entry, is that the designer has to be familiar with the target technology and should know in detail when to use which macro.

From a VHDL perspective, these macros can be utilized using component instantiation with hardwired connectivity. For instance, when addition is required in the Xilinx 4000 family, the designer can instantiate the ADSU8H for an 8-bit implementation or the ADSU16H for a 16-bit implementation using a port map:

```
component adsu8h
   port ( a    :in  std_logic_vector(7 downto 0);
          b    :in  std_logic_vector(7 downto 0);
          add  :in  std_logic;
          s    :out std_logic_vector(7 downto 0);
```

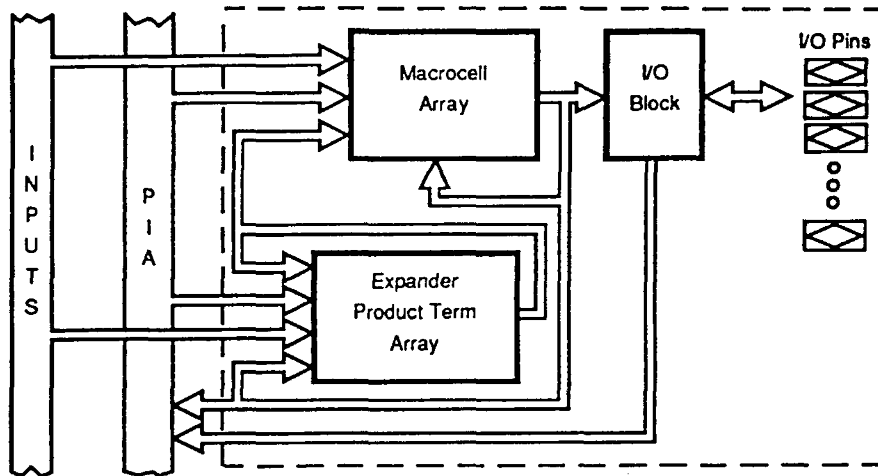

Figure 3. Simplified Block Diagram of Altera MAX5000 Logic Array Block.

```
          ofl  :out std_logic );
end component;
signal busa, busb, sum : std_logic_vector(7 downto 0);
signal pwr, ofl        : std_logic;
   ...
g0: adsu8h port map (a=>busa,b=>busb,add=>pwr,s=>sum,ofl=>ofl);
```

However, this removes the higher level of abstraction that the HDL was supposed to provide in the first place and essentially reduces VHDL design to textual schematic capture. Component instantiation is also limited to the dataflow, and as such prevents the use of these macros in concurrent processes. But at the same time, it is the only way to assure the utilization of the fast-carry chain in this particular architecture.

Assuming that the overflow signal (ofl) is not required in this implementation, a typical VHDL statement do achieve addition would have been

```
sum <= busa + busb ;
```

For the Xilinx 4000 architecture, the designer wants to see the '+' operator implemented by the ADSU8H hard macro, but cannot enforce this as VHDL does not allow for component overloading of any operator or function. One can overload functions and operators in VHDL, but the body of a function, being a sequential operation, cannot contain components. This makes it impossible for the VHDL designer to enforce a particular implementation on an operator.

Closed solutions

A straightforward solution to the above problem is to have the synthesis tool decide what type of implementation to use when encountering operators in the VHDL source

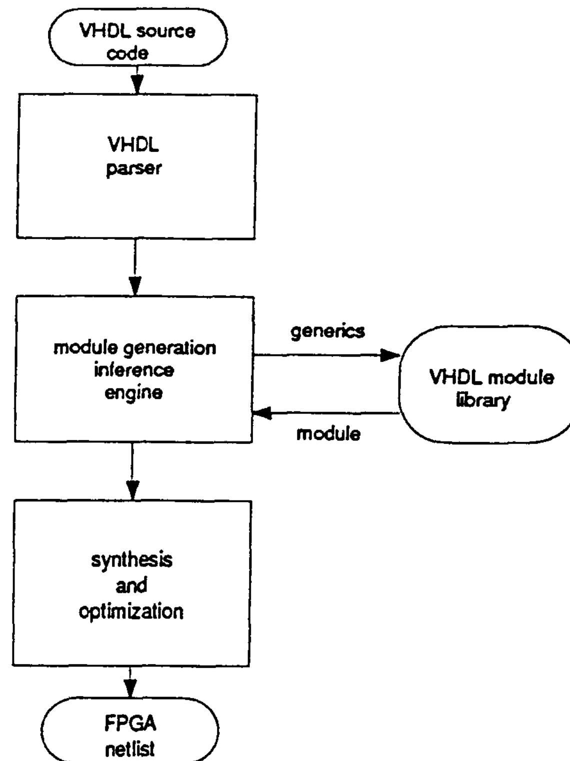

Figure 4. Module Generation Data Flow from VHDL to FPGA netlist.

description. It is easy to envision a tool that automatically instantiates (hardcoded) an ADSU8H macro for an 8-bit addition for the Xilinx 4000 family, or an FADD8 macro for a similar Act2 implementation. Such an approach seems to be in agreement with the technology dependent optimization for random logic. However, as it is not a generic algorithm it is not generally applicable. In fact, its is a discrete solution, limited to one incidental problem. It is also limited to the creativity and bandwidth of the tool vendor, and as that may well be out of sync with the requirements of the design community. Finally, it is not applicable for design environments where designers can build their own specialized macros, or utilize vendor provided macro generators [2]. Therefore, the closed solution is considered unacceptable.

Generic module generation

The preferred solution is one where designers can refer to libraries where certain operators are defined in terms of a target technology. Such a library could contain n-bit adders, subtractors, accumulators, and other datapath operators. The synthesis tool chooses from this library the preferred implementation of an operator, depending on size and well-defined attributes. Such a library, which is referred to as a module generation library, can be defined in many different ways, but is preferably written in
VHDL.

Figure 4 shows the general flow of data in a module generation environment. After the VHDL source code is successfully parsed, it is passed on to an inference engine that matches supported operators like addition with preferred implementations in the module generation library. Matching is performed on three levels:

- name of the module generator
  each operator has an identifying name, e.g. 'modgen_add' for '+'.
- generic value 'size'
  module generators can be defined for any size in the natural range.
- number of ports
  the number of ports is defined by the module generator and its size.

(Other generics can be defined as well, as discussed below, but are not mandatory for a match.)

If a match is found, the module generator from the library is instantiated. Otherwise the default Boolean definition for the operation is taken.

The module generation library can define the module generators in many different ways, for instance as a bit-slice or with discrete solutions matching the size generic. The only restriction is that the module generation library cannot be recursively defined.

As an example, consider again the addition operator. Assuming Xilinx 4000 as the target architecture, a module generation library can be provided as in the following listing (the xilinx package referred to in the listing contains the component declarations).

```
use work.xilinx.all;

entity modgen_add is
  generic (size    : integer := 16 );
  port     ( a, b   : in  std_logic_vector(size-1 downto 0);
             s      : out std_logic_vector(size-1 downto 0) );
end modgen_add;
architecture eXemplar of modgen_add is
      signal add   : std_logic;
      signal ofl   : std_logic_vector(0 to 1);
begin
   m8: if (size = 8 ) generate
    p0: adsu8h  port map (a=>a, b=>b, s=>s, add=>add, ofl=>ofl(0));
   end generate;
   m16: if (size = 16 ) generate
    p0: adsu16h port map (a=>a, b=>b, s=>s, add=>add, ofl=>ofl(1));
   end generate;
end eXemplar;
```

This, limited, module generation library defines specific implementation for size=8 and size=16 generators. (Actual module generator descriptions will, in general, synthesize modules of any size.)

The fixed name for this module generator is 'modgen_add'. The generic value 'size' in its entity is also required to communicate which generator is eventually instantiated. Finally, every module generator comes with a fixed set of ports, in this case two input and one output bus. The size of the buses is determined by the 'size' generic, and is resolved by the inference machine at compile time.

Inference prevention attribute

Although module generation is the preferred way of implementing structured logic, it should not be inferred in all cases. Once a module generator is instantiated, it is considered an external component that cannot be touched by the synthesis tool anymore. However, there are situations where an operator is used in the VHDL source description that eventually will be swept away during optimization. Type conversion functions are a good example of such a situation. A std_logic_vector to integer conversion function may very well use the '+' operator to achieve the conversion, as in the next listing:

```
function vector_to_int (vect : bit_vector; size : integer)
   return integer is
      variable result : integer range (2**size)-1 downto 0 ;
begin
   for i in 0 to size-1 loop
      if (vect(size-1-i)='1') then
         result := result + 2**i ;
      end if ;
   end loop ;
   return result ;
end vector_to_int ;
```

In normal operation, the inference engine will match the '+' operator with a module generator from the library (the 'size' parameter of the function has been resolved by the VHDL parser) and a component is instantiated. However, the type conversion, from a hardware point of view, should evaluate to a set of wires, and as such not appear in the synthesized netlist. The inference engine, unfortunately, cannot make this distinction between 'real' and 'virtual' structured logic, and should be guided by the VHDL designer if such a situation occurs. Therefore, a special attribute is required to disable module generation. This attribute, called 'modgen_enable' operates on signals and variables. In the above example, one would write:

```
attribute modgen_enable : boolean ;
attribute modgen_enable of result: variable is FALSE ;
```

If the attribute is not specified , module generation is considered on by default. Notice that the attribute switches module generation off only for the variable or signal it is specified on.

User defined attributes

In order to allow for as much user creativity as possible, the module generation environment lets designers specify user-defined attributes on the module generators. The attributes should be defined as generics with a default value in the entity interface of the module generator as follows:

```
type speed_grade_t is (slow, medium, fast);
entity modgen_add is
```

```
    generic ( size    : integer := 16 ;
              speed   : speed_grade_t := medium );
    port    ( a, b    : in  std_logic_vector(size-1 downto 0);
              s       : out std_logic_vector(size-1 downto 0) );
end modgen_add;
```

Once the attribute is declared as a generic, it can be used in the architecture body to specify different implementations.

```
m0: if size = 16 generate
    if speed = slow generate
       g0 : slo_macro port map ( ... );
    end generate ;
    if speed = medium generate
       g0 : med_macro port map ( ... );
    end generate ;
    if speed = fast generate
       g0 : fas_macro port map ( ... );
    end generate ;
end generate ;
```

The VHDL source code has the attributes operate on the left hand side signal of the assignment:

```
type speed_grade_t is (slow, medium, fast);
attribute speed_grade: speed_grade_t ;

result <= bus_a + bus_b ;
attribute speed_grade of result:signal is fast ;
```

The inference engine will match the speed_grade attribute, as it matches the mandatory generics, with the generator in the module generation library.

Simulation issues

A potential drawback of the module generation paradigm is that the module generator that eventually is implemented by the inference engine is not the same as the operator that was simulated at the behavioral level. In principle, this can lead to a mismatch between pre-and post synthesis simulation.

The operators are matched by name , not by functionality, and because of that, the body of the generator can contain just anything. The designer linking in a specific module generation library is solely responsible for verification of either the module generator, or the synthesized solution.

This in itself is not a new problem. Behavioral simulation libraries have exactly the same problem, where the match between hardware component and software description is not guaranteed by construction. Whenever the designer would instantiate a component using a port map, he or she can still design with the wrong instance when a mismatching behav-

ioral description of that component is used.

Prototype implementation

The above described system has been implemented in Exemplar Logic's VHDL environment. It currently recognizes the following operations:

'+'	addition
'-'	binary subtraction, unary negation
'+1'	increment by 1
'-1'	decrement by 1
'='	equal
'\='	not equal
'>'	greater then
'>='	greater then or equal
'<'	less then
'<='	less then or equal

Partial module generation libraries for Xilinx 3000, Xilinx 4000, Act1, and Act2 families have been developed for up to 32-bit implementations.

Experimental results

Table 1 shows results for VHDL synthesis with and without module generation. The target architecture was Xilinx 4000. The source VHDL file contained a simple n-bit addition. Results include i/o buffers and pad delays and are after place and route.

	with module generation		without module generation	
design	area	delay	area	delay
adder8	13	30ns/32MHz	14	54ns/18MHz
adder16	21	36ns/28MHz	30	107ns/9MHz

Table 1. VHDL synthesized circuits with/without module generation.

Summary

We have presented a methodology that allows VHDL designers to enforce the use of technology dependent implementations for arithmetic and relational operators, while preserving the benefits of technology independent specification. This concept, dubbed generic module generation, matches specific arithmetic or relational operations with predefined implementations from a VHDL library file. Generic module generation is proposed as a truly open system, and allows designers to create their own custom generators

based on custom defined generics.

The benefits of generic module generation are manyfold. Not only has the designer the choice of many different structured logic implementations for a design, but also once a decision is made to change the target technology, only the linked module generation library needs to change in order to obtain an equally optimal design in the new technology.

Generic module generation has been implemented in Exemplar Logic's CORE, a logic synthesis tool aimed at the FPGA design community.

Acknowledgments

Bob Condon, principal engineer at Exemplar Logic, suggested the use of a VHDL side file to implement the module generators, and in that sense can be considered the spiritual father of the concept.

References

[1] 'ELSS: A Logic Synthesis Tool for FPGAs', R.P Ranauro and M.M. Ligthart, proceedings of the 4th annual IEEE Asic Conference and Exhibit, 1991, pp. 13.2.1-13.2.4.

[2] 'Shortening the Design Cycle for Programmable Logic Devices', J.Seidel and S. Kelem, IEEE Design and Test, pp. 40-50, December 1992.