

VHDL Model Generation in an ASIC Design Environment

Stephen R. Burket
Defense Systems & Electronics Group
Texas Instruments Inc.

David A. Kluver Sr.
Information Technology Group
Texas Instruments Inc.

Abstract

This paper describes the design and implementation of an automated VHDL model and testbench generator within an ASIC design environment. This capability allows a designer to generate a behavioral and/or gate-level VHDL description of a design by exploiting the environment's high-level function generators. This description can then be exported to a commercial VHDL environment for simulation and/or synthesis. The testbench provides automated functional verification.

Introduction

During the design cycle of an ASIC, a normal requirement is that the ASIC be simulated at the system-level since many ASICs that simulate correctly on their own will fail when simulated within the system. In our ASIC design environment, there was no ready path to a system-level simulation environment. In addition, present users of this ASIC design environment have customer requirements for VHDL documentation of their designs.

We therefore developed a VHDL Export capability to generate a VHDL representation of a design. This capability meets the VHDL documentation requirements of our customers and provides a path from the ASIC design environment to the system-level simulation environment. After a brief description of our ASIC design environment, Behavioral Level ASIC Design Environment (BLADE), we discuss the process used to develop the components of the VHDL Export capability. These components include a VHDL netlist generator, a VHDL testbench generator, a VHDL symbol generator, and libraries of behavioral and gate-level VHDL models.

The Design Environment

The BLADE system includes a commercial schematic editor, simulator, and synthesis tool along with the netlist and vector translators required to drive foundry tools. The schematic editor contains high-level function generators such as ALU, counter, multiplexer, barrel shifter, and multiplier, along with test modules such as RAM Built-In Self Test (BIST) and Linear Feedback Shift Registers (LFSR). These behavioral generators are user-configurable with respect to register width, architecture, algorithm, control line polarity, etc. They can be simulated at the behavioral level and synthesized into foundry gates for gate-level simulation. With this capability a designer can quickly capture a behavioral-level design which can be synthesized into technology gates. See figure 1 for a block diagram of the BLADE system.

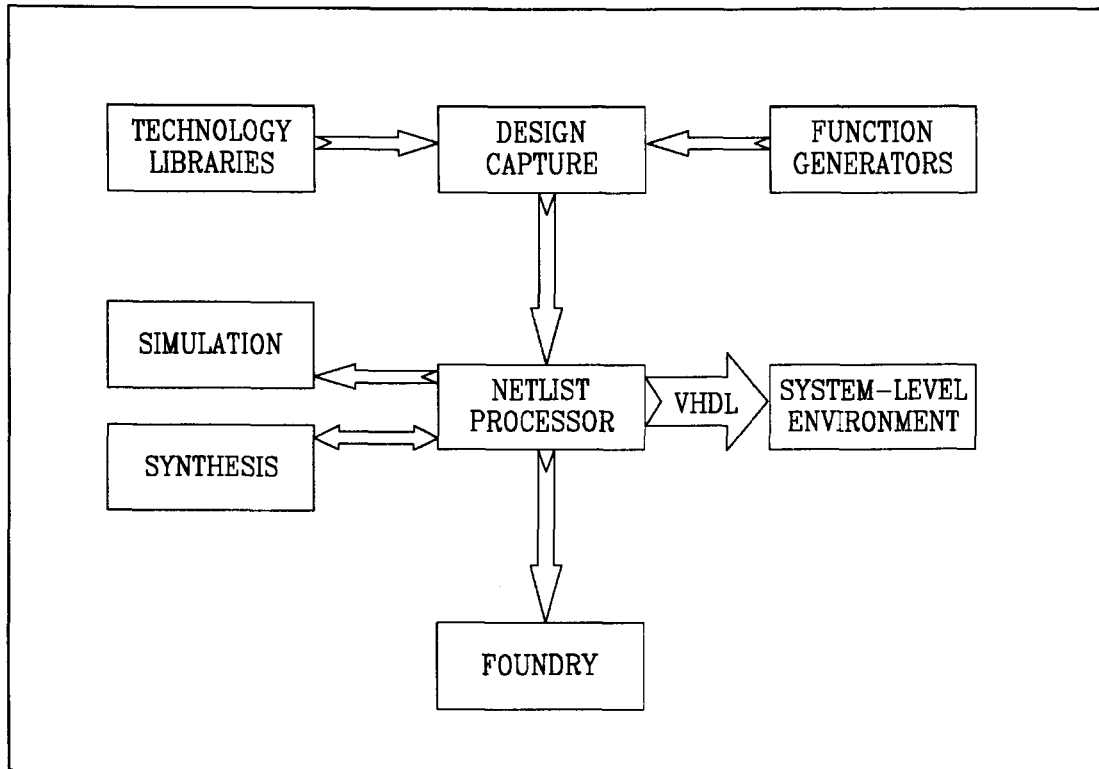


Figure 1. BLADE Block Diagram

VHDL Netlist Generator

Prior to starting work on the VHDL Netlist Generator we determined a list of initial requirements. The Netlist Generator should function in a manner consistent with the other tools already in use in the BLADE system. It should have the ability to generate a behavioral and/or structural netlist representation of a design. It should also allow the designer to make full use of the high-level function generators within BLADE. The netlist produced by the Netlist Generator should be an exact hierarchical representation of the design in IEEE Std 1076-1987 [1] VHDL. The complete VHDL capability in BLADE should be based on the IEEE 9state logic system [2], as this is the standard VHDL logic system for TI. A single VHDL target environment should be selected to avoid inconsistencies in syntax between available commercial VHDL tools.

One of the main drivers of the methodology used for the VHDL Export capability was the high-level function generators present in BLADE. To make full use of these function generators, a behavioral-level VHDL model was created for each generator. The parameters required to configure these models would be passed by the netlist. After analyzing a number of models we decided to pass these parameters as generics, by reference, and that one enumerated type would be set up to handle all non-integer parameters.

One of the areas where we ran into problems was in the use of generics to pass the required function generator parameters. A number of the models require passing an integer of size greater than 2^{32} , which exceeds the integer space of our workstation. To resolve this, the netlist represents this integer as an array of integers which are the remainders of successive divisions by 2^{31} . The models then reconstruct the appropriate value. Also, some of the VHDL models for the function generators originally used generics to define other generics. This capability was removed from our target VHDL environment in a later release, the result of a clarification of the VHDL specification. Unconstrained arrays replaced this capability.

The need for component declarations for each model instantiation in the netlist was expected to have a large impact on the development of the Netlist Generator. We were concerned about the amount of code that would be required in the Netlist Generator to generate this data. This was not an issue, however, because VHDL permits placing pre-written component declarations in a side package. The Netlist Generator simply references this component package with a 'use' clause.

A design goal for the BLADE VHDL Export capability was to have only one VHDL model for each high-level function generator. Since the generators are configurable, a different set of ports may be present for each instantiation of the generator model in the design. To accommodate this, the Netlist Generator maintains a list of all optional ports. When they are not present for a specific instantiation, an actual of "open" is assigned.

We initially used configuration specification statements to bind entities to architectures. This configuration scheme failed for hierarchical designs because the Netlist Generator did not build a bottom-to-top hierarchical netlist. Since building a bottom-to-top netlist proved to be a difficult task in BLADE, we changed to a four line "default" configuration declaration. This worked for all models except one which had a parameter our target simulator environment was unable to pass as a generic. We ended up using a fully-elaborated configuration declaration statement. See figure 2 for an example of a BLADE VHDL netlist.

A number of the BLADE high-level function generators have a variable number of input busses of variable width. For these cases we were forced to use a two-dimensional array for the port definition. This is because VHDL does not allow defining an array of arrays using generics, but does allow generics to be used when defining two-dimensional arrays.

We decided to map busses in port maps bit-by-bit to allow for assembly of busses from differently-named sources. This worked for all cases except for one model that used two-dimensional arrays. The problem was caused by the fact that for some instantiations of this model, some of the input busses are masked (not present). For example, one instantiation might have three input busses IN0, IN1, IN2, where IN1 is masked. Assigning 'open' for the masked bus bits failed. A two-dimensional signal was defined to resolve this problem. These signal bits were assigned to the bus inputs bit-by-bit, ignoring the masked bits, and then the signal was assigned to the two-dimensional port of the model through the port map. Looking back, it appears possible that the failure of the "open" assignments was due to an incorrect default assignment for the two-dimensional port.

In BLADE, ports and their signals have the same name. This presents a problem in VHDL syntax for the case where the signal driving a port also drives another component. Buffer mode can be used to resolve this, but cannot be used for output ports with multiple sources. To avoid requiring analysis of the design by the Netlist Generator, a signal is automatically created for every output port and inserted between the port driver and the port. This same solution is implemented for input ports to deal with the possibility of input pull-ups.

A recurring issue was the difference in syntax capabilities of BLADE and VHDL. BLADE syntax is more flexible than VHDL syntax. One difference is that BLADE has separate name spaces for each design object type, such as signals and components. To handle the single name space allowed in VHDL, name hashing was required. The synthesis capability of BLADE presented a second problem. During synthesis, portions of signal busses may be eliminated, resulting in non-contiguous bus ranges. To resolve this illegal VHDL condition, bus bits are re-ordered into contiguous arrays. Other syntax problems were resolved by adding a three letter prefix to VHDL reserved words found in the design, and hashing name sizes to match VHDL target system name length limitations.

Only one technology library is allowed in an ASIC design. However, in the system-level environment multiple technologies may be present. This requires that the VHDL description for each ASIC must reference the appropriate technology library. This requires that the Netlist Generator detect the technology for the design and add the proper library references to the netlist.

```

library ieee;
use ieee.std_logic_1164.all;
library blade;
use blade.compiler_pkg.all;
use blade.compiler_components.all;

entity EXAMPLE is
  port (
    DOUT: out std_logic_vector(1 downto 0);
    INBUS1: in std_logic_vector(1 downto 0);
    INBUS2: in std_logic_vector(1 downto 0);
    TOUT: out std_logic_vector(2 downto 0)
  );
end EXAMPLE;

architecture structure of EXAMPLE is

  signal JJ_DOUT: std_logic_vector(1 downto 0);
  signal JJ_INBUS1: std_logic_vector(1 downto 0);
  signal JJ_INBUS2: std_logic_vector(1 downto 0);
  signal JJ_TOUT: std_logic_vector(2 downto 0);

begin

  TOUT(2) <= JJ_TOUT(2);
  TOUT(1) <= JJ_TOUT(1);
  TOUT(0) <= JJ_TOUT(0);
  JJ_INBUS2(1) <= INBUS2(1);
  JJ_INBUS2(0) <= INBUS2(0);
  JJ_INBUS1(1) <= INBUS1(1);
  JJ_INBUS1(0) <= INBUS1(0);
  DOUT(1) <= JJ_DOUT(1);
  DOUT(0) <= JJ_DOUT(0);

  TALLY1: LL_TALLY
    generic map (
      LL_WO => 3,
      LL_W => 4
    )
    port map (
      MJR_IN(3) => JJ_INBUS2(1),
      MJR_IN(2) => JJ_INBUS2(0),
      MJR_IN(1) => JJ_DOUT(1),
      MJR_IN(0) => JJ_DOUT(0),
      MJR_OUT(2) => JJ_TOUT(2),
      MJR_OUT(1) => JJ_TOUT(1),
      MJR_OUT(0) => JJ_TOUT(0)
    );

  INV1: LL_BUFF
    generic map (
      LL_W => 2,
      LL_invert_output => SRB_YES,
      LL_invert_input => SRB_NO,
      LL_output_type => SRB_DIRECT
    )
    port map (
      MJR_IN(1) => JJ_INBUS1(1),
      MJR_IN(0) => JJ_INBUS1(0),
      MJR_OUT(1) => JJ_DOUT(1),
      MJR_OUT(0) => JJ_DOUT(0),
      CLK => open,
      EN => open
    );
end structure;

configuration EXAMPLE_config of EXAMPLE is
  for structure
    for all: LL_TALLY use entity blade.ll_tally(behavior); end for;
    for all: LL_BUFF use entity blade.ll_buff(behavior); end for;
  end for;
end EXAMPLE_config;

```

Figure 2. BLADE VHDL Netlist Example

VHDL Model Libraries

The VHDL models for the function generators were developed concurrently with the Netlist Generator, with the goal to have just one VHDL model for each generator. Writing the models so that their behavior was identical to their respective function generators was facilitated because no timing was required (generators are functional only). The effort was hindered, however, by the lack of documentation detailing the functionality of the generators. This proved especially difficult when trying to model the handling of unknowns. Simulation was required in BLADE, to determine the complete functionality of the generators, and in the VHDL environment, to verify the functionality of the VHDL models. This simulation effort helped furnish an impetus to develop the VHDL Testbench Generator.

Our original plan was to fully implement the IEEE 9state logic system within the models, however, the complexity of implementing all nine logic states inside the models proved unjustifiable. We decided to convert the nine state inputs into the three state subtype (X01) for use inside the models.

The basic structure of our VHDL models falls into two categories. Most of the models were written in an entirely behavioral style. A hardware substructure style was used in the other models where the generator was built in a known hardware implementation. For example, the code for an adder was written behaviorally as $(a + b)$, where the code for the LFSR contains calls to a mux procedure along with calls to a dff procedure.

Initially, a number of model features proved awkward to code in VHDL. These were the selectability of control and clock signal polarity, the fact that control and clock signals may or may not be present in a particular instantiation, and that control and clock signals may be controlled by an enable signal. Checking for polarity, whether a signal is present or not, and the status of enable lines whenever the signal of interest appears in the model proved inefficient, tedious, and confusing. A number of methods were used to deal with these troublesome signals. If the only problem was that the signal had user-definable polarity, the model was written with a predefined polarity. The user-defined polarity definition of the signal is checked once at the beginning of the model, and if different from the predefined polarity, the signal is inverted. For signals with a combination of these problems, such as clock, set, and reset, a variable type with values of YES, NO, and UNKNOWN is defined. The signals are converted into one of these values at the beginning of the model, and when the signal appears later in the model, this simple variable is evaluated.

VHDL Testbench Generator

As the development of the Netlist Generator and model library proceeded, we realized that the automatic testbench generator, planned to meet the requirements of our customers, would be extremely valuable for our testing of the Netlist Generator and VHDL models. We therefore moved up the development effort for this task. We decided that the testbench should be based on one source to prevent problems with multi-source conflicts. The source we chose was the BLADE simulator test vector file. This file, which contains input and output signal names, strobe times, input values, and expected output values, is produced automatically by the simulator during a simulation. No attempt was made to use the VHDL WAVES [3] format for generating the testbench as WAVES had not been released at the time.

Initially we generated an example testbench by hand to get an idea of what would be required. From this example testbench we decided that the testvector file should be parsed for the strobe time, stimulus, and expected response. We also decided that the model should have a null entity (one without a port list), and that connections should be made to the device under test (DUT) via signals. We determined that if we put all of the code in one model, the model would be very large, and the automated process to build the testbench would be inefficient. We therefore proceduralized as much of the testbench code as possible and placed this proceduralized code in a testbench package. The rest of the reusable code was placed in a lookup file for the testbench generator to reference. This allowed the testbench generator to supply just the design related specifics, pulling the non-design related code from the lookup file. We also decided to have the testbench generator do an initial parse of the testvector file to remove all extraneous information. This was to reduce the amount of

parsing the testbench would have to do during simulation. See figure 3 for a block diagram of the BLADE VHDL testbench.

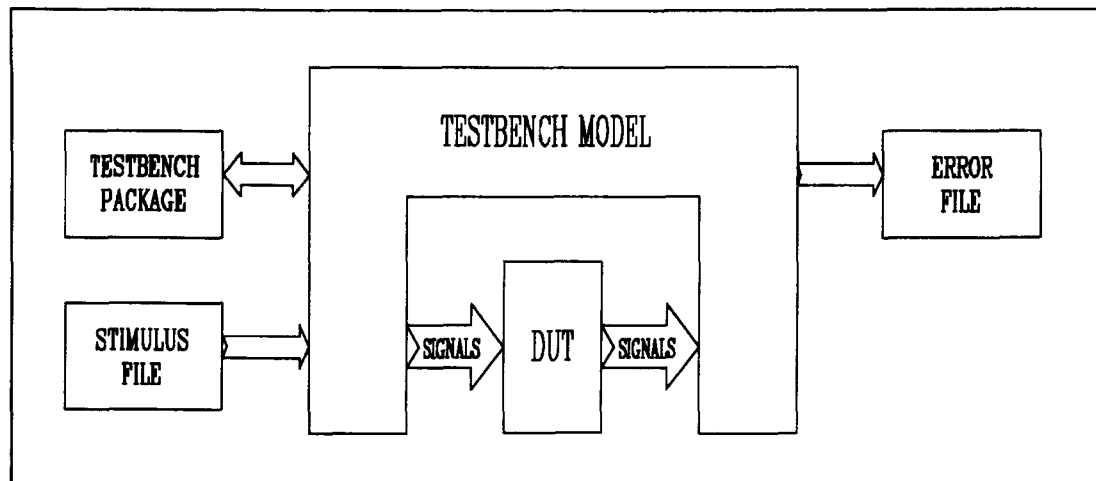


Figure 3. BLADE VHDL Testbench Block Diagram

Since our testbench stimulus originates in the BLADE simulator, which doesn't use IEEE 9state logic, conversion procedures had to be generated to convert BLADE simulator logic to IEEE 9state logic. In addition, conversion procedures for character manipulation were required for file reading and writing. We found that the limited capabilities of the TEXTIO [1] package made it very inconvenient to parse the stimulus file, as well as to write to error files.

It was a requirement for the testbench to report discrepancies to an error file, discrepancies being defined as a miscompare between the VHDL simulation and the expected response from the stimulus file. The problem was how to put the signal names in the testbench model so that the testbench could write the names to the files in the proper format. We initially tried to use arrays sized large enough to handle any set of signal names. VHDL, however, requires that all array locations be filled, which was considered unwieldy. A satisfactory solution was to use arrays of string arrays whose sizes were defined by the testbench generator based on the design, with the signal names being placed as the default values in the array declarations. See figure 4 for a partial example of a BLADE VHDL testbench model.

After using testbenches to test a number of our models, we discovered that reporting all of the discrepancies involving unknowns made it very difficult to locate logic errors in the error files. Adding a switch to the Testbench Generator allows the designer to have unknown miscompares ignored by the testbench.

VHDL Symbol Generator

After use of the VHDL Export capability by some of our customers, a request was made for an automatic way to generate a netlist of user-created VHDL models. This request entailed two capabilities, the first to automatically generate a VHDL entity from a user-created symbol, the second to automatically generate a modifiable graphical symbol in the schematic editor from a VHDL model.

One of the problems encountered while reading a model to create a symbol was the depth of parsing needed to extract the relevant data. For example, if a port definition relies on a generic for its size, the Symbol Generator is unable to parse for this value. For this case, a default value of one is assigned and the designer is required to edit the symbol to the desired size.

```

use std.textio.all;
library ieee;
use ieee.std_logic_1164.all;
library blade;
use blade.compiler_pkg.all;
use blade.testbench_pkg.all;

entity EXAMPLE1_testbench is

    subtype vert_string is string (1 to 22);
    subtype horz_string is string (1 to 10);
    type vert_matrix is array (natural range <>) of vert_string;
    type horz_matrix is array (natural range <>) of horz_string;

end EXAMPLE1_testbench;

architecture testbench of EXAMPLE1_testbench is

    signal SRB_IN: std_logic_vector(3 downto 0);
    signal SRB_OUT: std_logic_vector(2 downto 0);

    component EXAMPLE1
        port (
            SRB_IN: in std_logic_vector(3 downto 0);
            SRB_OUT: out std_logic_vector(2 downto 0)
        );
    end component;

begin

    DUT: EXAMPLE1
        port map(
            SRB_IN => SRB_IN,
            SRB_OUT => SRB_OUT
        );

    process

        variable tb_outputs: std_logic_vector(1 to 4);
        variable tb_inputs : std_logic_vector(1 to 3);
        variable ch_outputs : char_array           (tb_outputs'RANGE);
        variable last_ch_outputs : char_array      (tb_outputs'RANGE);
        variable ch_inputs  : char_array           (tb_inputs'RANGE);

        variable error_markers : char_array        (tb_inputs'RANGE);
        variable ch_good_stuff  : char_array        (tb_inputs'RANGE);
        variable last_ch_good_stuff : char_array    (tb_inputs'RANGE);
        variable good_stuff     : std_logic_vector (tb_inputs'RANGE);
        variable last_good_stuff : std_logic_vector (tb_inputs'RANGE);
        variable last_tb_outputs : std_logic_vector (tb_outputs'RANGE);

        variable vert_names: vert_matrix (1 to 11):= (
            "          ",
            "          SSS",
            "          SSSS RRR",
            "          RRRR BBB",
            "          BBBB  ",
            "          OOO",
            "          IIII UUU",
            "          NNNN TTT",
            "          ((( ((",
            "          3210 210",
            "Strobe Time  ))) ))");

        variable horz_names: horz_matrix(1 to 4):= (
            "SRB_OUT(2)",
            "SRB_OUT(1)",
            "SRB_OUT(0)",
            "          ");

        variable test_bench_name: string(1 to 18):= "EXAMPLE1_testbench";
        variable time_unit      : time;
    end process;
end architecture testbench;

```

Figure 4. BLADE VHDL Testbench Example (Partial) Showing Signal Name Array Definitions

Although the default logic system for the VHDL Export capability is IEEE 9state, we recognize that designers may want to generate models based on their own logic system. The Symbol Generator parses the model to determine the logic type used for each port; the Netlist Generator makes use of this information when creating the netlist.

Conclusion

The BLADE VHDL Export capability is presently being used with success by projects at TI. It is allowing designers to generate VHDL netlists for their designs, verifying them at the system level. We feel that the VHDL language has enabled us to meet all of our design goals. The success of the VHDL Export capability has shown that VHDL can be a useful mechanism for communicating design data independent of a design environment. It is proving to be a key to the verification of multi-technology system-level designs.

References

1. IEEE Design Automation Standards Subcommittee "VHDL Language Reference Manual" IEEE Std 1076-1987" (1988)
2. IEEE Model Standards Group "Std_logic_1164 Multi-Value Logic System" (January 1992)
3. IEEE Design Automation Standards Subcommittee "Draft WAVES Standard PAR 1029.1" (November 1990)