VHDL: A Call for Standards

David R. Coelho

Vantage Analysis Systems, Inc.
Fremont, California 94538

Abstract

With the introduction of the IEEE 1076 version of VHDL, an excellent industry standard hardware description language is now available. VHDL is an extremely flexible and versatile language. As a consequence, the language reference documentation is not sufficient to insure that models written by one hardware designer will be compatable with another's models. What is required is a set of VHDL modelling conventions and standard packages which structure the usage of VHDL modelling approaches. This paper will discuss the issues inherent in VHDL in regards to model compatability, and will propose a number of solutions to this problem.

1 Introduction

VHDL (VHSIC Hardware Description Language)[VHDL86,87] has evolved over the last several years into a very effective tool for describing electronic hardware systems. The scope of the VHDL language is very wide, allowing the description of systems ranging from the microcode and architectural levels down to the gate level. As a result of this multi-level capability, the VHDL language has considerably more flexibility and power than most other hardware description languages. For the vast majority of hardware designers, this flexibility and power is not required, and in fact can result in the creation of models that have severe compatability problems when mixed during simulation.

An analogy exists to the VHDL language in the area of programming languages. With a language such as C or Pascal, unless the use of these languages is restricted according to a strict methodology, a team of programmers will have severe difficulties during integration of a large programming effort. As a result, one of the key characteristics of an effective software engineering effort is the introduction of methodology including:

- Code Sharing; Effective use of Libraries
- Standardized Coding Conventions
- Proper Documentation of Code
- Effective Modularization of Code
- Standardization of Coding Practices including handling of errors and messages

These good engineering practices are well accepted in the software area, but are not as well understood or adopted in the hardware modelling area. With a language such as VHDL, the use of structured software engineering practices is critical in order to insure proper compatability of models during simulation.

Another motivation for the above practices relates to one of the primary requirements which influenced the development of the VHDL language, the need to effectively document hardware designs. The VHDL language in this context is useful as a mechanism to describe the operation of a hardware system, and to allow this information to pass from one designer to another, or from one organization to another. In this regard, many of the aspects mentioned above become even more important.

From this perspective, it is clear that the VHDL Language Reference Manual is not sufficient to insure the creation of compatable and documentation quality models. The remainder of this paper will outline the specific problems which must be addressed in regards to VHDL modelling practices and will propose specific solutions and approaches to these problems.

2 Model Compatability

The following sections discuss issues related to insuring the compatability between models during simulation.

2.1 Generic Parameters

Generic parameters in VHDL allow specific instantiations of a model to have different characteristics. The most obvious use of this capability relates to timing characteristics of devices. In a typical design, although the basic function of a nand gate may remain the same for all instances of the gate, the timing can change from one instance to another. Generics allow the timing associated with each gate to be passed as a parameter.

```
-- Nand gate interface with a single propagation
  -- delay for the output port
  ENTITY nand_gate IS
    GENERIC (y_prop : TIME);
    PORT (a, b : IN logic;
          y : OUT logic);
    USE standardlogic.ALL;
    END nand_gate;

  -- Nor gate interface with input port delays,
  -- and different delays on output for rising
  -- and falling values
  ENTITY nor_gate IS
    GENERIC (a_in, b_in, y_f, y_t : TIME);
    PORT (a, b : IN logic;
          y : OUT logic);
    USE standardlogic.ALL;
    END nand_gate;
```

Figure 1: Incompatability between Generic Parameters

When many models are combined during simulation, a consistent methodology must be used in each model. If timing parameters are handled in an inconsistent fashion, it may not be possible to effectively back–annotate delays from the layout of a design. Figure 1 illustrates this problem. In this example, a model of a nand gate and a nor gate are shown. Two different conventions were used in these models in regards to the information passed as generic parameters. In the nand gate, a single generic parameter **y_prop** which represents the propagation delay associated with the device in all cases was used. In the nor gate, several time values were passed; **a_in** and **b_in** represent input port delays, **y_t** and **y_f** represent rising and falling propagation delays associated with the output port. The problem in this situation is that a back–annotation program must provide information that varies depending on what model the information is passed to. This problem is compounded severely when more than two models are used during simulation.

Other compatability problems can occur when some models require more detailed information than others. For example, setup and hold time constraints are useful generic parameters for models. Unfortunately, if one designer ignores these values, and another designer includes these parameters, the resulting simulation will be limited in it's effectiveness.

2.2 Constraints

Behavioral simulators and hardware description languages have a significant advantage over older gate level technologies by supporting semantic checks as part of the hardware model. The following summarizes some of the more important checks which can be performed very effectively in VHDL:

- Setup limits
- Hold limits
- Spike detection
- Special timing requirements
- Invalid data

It is important that models which are combined during simulation consistently handle constraint checks. If this rule is not followed, the effectiveness of the simulation will be reduced, and in the case of spike detection the simulation results can be erroneous. For example, if selected models suppress pass-through of spikes, but others don't, the results of the simulation can in fact be false. Although less severe, the results of a simulation can be very misleading if only selected models check setup and hold constraints. The designer may falsely assume that he has no timing errors in his design, when in fact some of his models are just not checking for these constraints.

2.3 Unknown Handling

One of the most difficult but most important aspects of writing a behavioral model is the proper handling of unknown values during simulation. Experience with simulators has shown that the introduction of an unknown state is required in order to correctly handle the following situations:

- Circuit power-up and associated simulator initialization
- Recovery from improper device use, both in timing and function

During circuit power-up, unknown values are required in order to accurately predict the state of circuit after the power-up sequence is completed. Consider the case of a flip-flop; At the end of a power-up sequence, the simulator must choose to set the flip-flop state as either true or false In reality, the state of the flip-flop is indeterminate, since the value of the flip-flop will lock in based on the current and voltage levels in the actual device which are highly dependent on the topology of the device. A more accurate reflection of the power-up sequence is to place the flip-flop initially into an unknown state, and only after a sequence of inputs to the device which force the device to a known state are received does the unknown value disappear. Proper handling of unknown values in models can be a very effective tool in diagnosing hardware designs, especially for power-up conditions. Should a design fail to properly eliminate unknown values during this stage of simulation, the designer can expect to see indeterministic behavior in the actual circuit.

A secondary use of unknowns occurs during error recovery. Consider a device which has indeterministic behavior for a given set of inputs. A J-K flip-flop is a good example. If both the J and K inputs are held high, the state of the device can not be predicted. One approach to handling this situation would be to report an error to the user, and halt the simulation. This approach is not a good one, since it doesn't give the user the option of proceeding with simulation in order to observe other effects in the circuit, and does not allow error propagation effects to be observed. An alternative would be to arbitrarily choose a true or false value. Here, the user may be deceived into believing that the simulation results are correct when in fact they may not accurately model the behavior of the actual circuit. The best solution is to place an unknown state in the flip-flop which indicates to the user that the state of the flip-flop has an indeterminate value.

From the standpoint of model compatability, it is critical that all models combined during a simulation use the conventions in regards to the handling of unknown. Clearly, if some models utilize unknowns, and others don't, improper simulation results are possible and at the minimum, any utility which might have been gained from the unknown state will be lost.

2.4 Naming Conventions

Good modelling practices dictate that consistent and uniform naming conventions be applied to the development of VHDL models. This becomes even more important when generic parameters are utilized, especially with respect to back-annotation of delays from layout. Without standardized naming conventions, it may not be possible for an automatic back-annotation facility to deposit values into the VHDL database.

Areas of concern are summarized here:

- Architectural body names
- Port names
- Generic parameter names

Clearly, maintaining a consistent mapping between schematic types and architectural bodies will make implementation of back-annotation easier. Consistency in generic parameter names makes automation of instantiation of generic parameter values possible. Since generic parameters often contain information which must ultimately be associated with port names, maintaining consistency between generic parameter names and port names will improve model readability and consistency.

2.5 Value System

One of the most controversial aspects of simulation relates to the value system adoped by the simulator. The following summarizes some of the most popular systems:

- 4 state system: True, False, Unknown, High-Impedance
- 12 state system: True, False, Unknown; each with a strength composed of Strong, Resistive, High-Impedance, and Indeterminate
- 15 state system: True, False, Unknown; each with a strength composed of Force, Strong, Resistive, High-Impedance, and Indeterminate

Each of these value systems has similarities that can be summarized here:

- Basic state values of True, False and Unknown
- Inclusion of a strength system with 4 or 5 values

The High-Impedance value associated with the 4-state system above is really a strength. The introduction of strengths in addition to the basic state values provides a convenient mechanism for modelling charge effects associated with switch level modelling and simulation. Although the strengths are not required for TTL and higher level simulation, the basic 4-state system can be viewed as a subset of the strength based systems.

Substantial experience has been gained in the use of these various value systems and in some cases in the mixing of value systems during a single simulation. Unfortunately, when value systems are mixed, significant technical challenges emerge including:

- Proper mapping from one value system to another
- Maintaining reasonable simulation efficiency
- Managing the increased complexity related to displaying simulation results to the user
- Managing the increased complexity of models which must deal with mixed value systems

In certain cases, simulation results can be inaccurate due to problems related to mapping from one value system onto another. In a broad sense, switch level components must be **isolated** in order to insure proper handling of charge effects. As a result, it is not possible to embed this intelligence in models, but rather the simulator kernel must have supplemental processing and data structures which reflect these isolated switch level portions of the circuit. In VHDL, this supplemental processing violates basic premises of the language and are clearly beyond the scope of a VHDL model. For this reason, mixing of value systems as part of a VHDL simulation introduces inaccuracies which result in inaccurate results.

From the above discussion, it is clear that the adoption of a single standard value system is highly advantageous as it eliminates the problems inherent in mixing such systems. Since only a minimal efficiency penalty is associated with the choice of a value system in VHDL, a superset of the most popular and effective value systems is prudent. For the reason, later sections will discuss the incorporate of the state system discussed above as part of a standard VHDL package.

3 The Solution

The solution to the problems outlined in previous sections can be addressed in the following ways:

- Provide a **set of standards and conventions** which limit the ways in which VHDL is used for typical hardware modelling tasks
- Provide one or more **standard logic modelling packages** which give the hardware modeller a framework and set of utilities which form a structured environment for model development
- Provide **standard VHDL libraries** of models which represent the more commonly used devices and parts

3.1 Modelling Standards/Conventions

Modelling standards are critical to the success of a standard library. A number of issues emerge when developing VHDL models for use by a range of users:

- **Model Value Systems** - without standardization, models may use different value systems. The net result of this is the inability to mix models efficiently or accurately during simulation. For example, if one model handles high-impedance but another model does not, the overall simulation results will suffer and in certain cases may not accurately reflect the behavior of the hardware.
- **Incompatable Generic Parameters** - generic parameters are most useful for back-annotation of timing and delays from layout. Unless all models in a library adhere to a common naming convention for generic parameters, an identical unit system for delays and consistent types of timing values, it will not be possible to back-annotate timing information.
- **Inconsistent Handling of Exceptions** - unless all models handle setup errors, hold errors and spike detection in a similar fashion, the usefulness of simulation will be affected. The important point here is that the user must have predictable and well documented simulation results and unless the exception handling in all models in consistent this won't be the case.
- **Documentation Standards** - for readability of models, the naming conventions, comment standards, and other related information should be consistent.

A *VHDL Modelling Standards Guideline* should establish conventions in the following areas:

- naming conventions regarding port names, model names, symbol names
- documentation standards regarding comments, signal names, algorithms, etc
- conventions regarding generic parameters for compatability with symbol timing attributes used in back-annotation from layout
- standard value systems
- standard bus resolution functions
- standard approaches to representing data abstraction
- standard modelling levels and associated conventions, i.e. ASICs, boards, systems, standard parts

3.2 Standard Packages

A standard logic modelling package has the potential to provide an excellent framework for VHDL model development and when combined with the previously mentioned modelling standards guideline can be effective in insuring compatability during simulation.

In this paper, a minimal set definitions will be proposed for this standard logic modelling package. Figure 2 shows the package declaration along with the type definitions for the package. A type **t_logic** is defined which represents the basic value system for logic level signals. Three values, true, false and unknown are used; five strengths, force, strong, resistive, high impedance, and indeterminate are used. In addition, state qualifiers are used to handle special unknown situations for switch level modelling. Thes special qualifiers allow the simulation to avoid overly pessimistic results when unknown values are fed to transmission gate primitives. The types **t_state** and **t_strength** are defined in order to allow efficient handling of model logic as demonstrated later. An array type is defined for bus resolution **t_logic_vector** along with the associated bus resolution function **f_logic_bus**. A subtype which is associated with the bus resolution function **t_wlogic** and an array version of it **t_wlogic_bus** are also declared.

Several utility functions are provided as shown in figure 3. **f_state** and **f_strength** return the associated state and strength respectively of a given signal value. The **f_qualifier** function returns the special value qualifer if it exists for a given value. Additional functions **f_logic** and

```
PACKAGE standardlogic IS
  TYPE t_logic IS (
    U,
    Z0,Z1,ZX,
    W0,W1,QW00,QW0X,QW11,QW1X,QWXX,WX,
    R0,R1,QR00,QR0X,QR11,QR1X,QRXX,RX,
    F0,F1,QF00,QF0X,QF11,QF1X,QFXX,FX
    );
  TYPE t_state IS ('0','1','X','U');
  TYPE t_strength IS ('Z','W','R','F','U');

  TYPE t_logic_vector IS ARRAY (POSITIVE RANGE <>) OF
    t_logic;
  FUNCTION f_logic_bus(s : t_logic_vector)
    RETURN t_logic;
    -- bus resolution function

  SUBTYPE t_wlogic IS f_logic_bus t_logic;
    -- wired signal data type
  TYPE t_wlogic_bus IS ARRAY (POSITIVE RANGE <>) OF
    t_wlogic;
    -- wired signal vector data type
```

Figure 2: Standard Logic Modelling Package

```
FUNCTION f_state(lv : IN t_wlogic) RETURN t_state;
    -- return state given logic value
  FUNCTION f_strength(lv : IN t_wlogic) RETURN t_strength;
    -- return strength given logic value
  FUNCTION f_qualifier(lv : IN t_wlogic) RETURN t_state;
    -- return bus resolution qualifier given logic value

  -- LOGIC VALUE BUILDING FUNCTIONS
  FUNCTION f_logic(lvstate : IN t_state;
    lvstrength : IN t_strength) RETURN t_wlogic;
    -- return logic value given state/strength
  FUNCTION f_logicq(lvstate : IN t_state;
    lvstrength : IN t_strength;
    lvqualifier : IN t_state)
    RETURN t_wlogic;
    -- return logic value given state/strength/qualifier
```

Figure 3: Utility Functions

f_logicq are used to construct logic values given states and strengths.

A set of basic logic functions is provided in figure 4. Each of these routines performs a logic function such as NOT, AND, etc.

In order to effectively handle timing calculations, the routine shown in figure 5 is provided. The **f_assign** function calculates the delay associated with a signal assignment and performs the signal assignment. This routine provides a basis for handling a wide range of technology dependent issues. In particular, an input delay as well as rising and falling output delays are utilized during the calculation. If these values are provided by a back-annotation facility, accurate processing of the timing related to individual traces of a layout can be taken into account.

3.3 Standard Libraries

Standard libraries provide an excellent basis for getting hardware designers started. They provide good examples of how models should be developed, they give the engineer a critical mass for starting use of VHDL, and they allow leverage of engineering effort by avoiding duplicated VHDL modelling efforts.

Important candidates for standard libraries include:

- ASIC macro libraries
- standard board level component libraries

In the previous section, a standard logic modelling package was summarized. In this section, a specific example which utilizes this package will be shown.

```
-- BASIC LOGIC OPERATION FUNCTIONS
FUNCTION f_tech(lvstate: IN t_state; t : IN t_technolog
  RETURN t_wlogic;
  -- return a logic value given a state expression.
  -- This function is used in conjunction with the
  -- following logic operation functions to return an
  -- expression value. For example:
  --
  --  NMOS nand gate:
  --    f_tech(f_nand(f_state(a),f_state(b)),nmos)
  --  TTL and/or:
  --    f_tech(f_and(f_or(f_state(a),f_state(b)),
  --       f_or(f_state(c),f_state(d))),ttl)
  --
FUNCTION f_not(a : IN t_state) RETURN t_state;
  -- return logic NOT of given value
FUNCTION f_and(a,b : IN t_state) RETURN t_state;
  -- return logic AND of given values
FUNCTION f_or(a,b : IN t_state) RETURN t_state;
  -- return logic OR of given values
FUNCTION f_nand(a,b : IN t_state) RETURN t_state;
  -- return logic NAND of given values
FUNCTION f_nor(a,b : IN t_state) RETURN t_state;
  -- return logic NOR of given values
FUNCTION f_xor(a,b : IN t_state) RETURN t_state;
  -- return logic XOR of given values
```

Figure 4: Logic Functions

```
  FUNCTION f_choosedelay(newval : IN t_wlogic;
    indel, out_f, out_t : IN TIME) RETURN TIME;

  PROCEDURE f_assign(newstate : IN t_state;
    SIGNAL sig : INOUT logic;
    in_delay, out_f, out_t : IN TIME);

END standardlogic ;
```

Figure 5: Delay Functions

Figure 6 shows the VHDL model for a nand gate. The generic parameters support the passing of a separate delay for each input port, and both rising and falling delays for the output port.

Figure 7 shows the declarations for a JK flip-flop. Once again, each input has a delay associated with it, and each output has both a rising and a falling delay. Notice also, that for this model, the clock input has a setup and hold delay associated with it.

Figure 8 shows the main body of the flip-flop model, and outlines how exceptions are handled in this model. Of particular interest are the VHDL assertions which test for setup and hold errors. Also of interest is the asynchronous clear process.

Figure 9 shows the remainder of the model. A single process is used to handle clocked inputs to the model. Once again, the **assign** routine is used to perform the delay calculations and the signal assignments to the output.

```
ENTITY nand_gate IS
  GENERIC (a_in, b_in, y_f, y_t : TIME);
  PORT (a, b : IN logic;
        y : OUT logic);
  USE standardlogic.ALL;
  END nand_gate;
```

```
ARCHITECTURE behavioral OF nand_gate IS
  BEGIN
  -- main process for handling device output
  PROCESS (a,b)
    BEGIN
    -- assign output, a input changed
    IF NOT a'STABLE THEN
      f_assign(f_nand(f_state(a),f_state(b)),
               y,a_in,y_f,y_t);
    END IF;

    -- assign output, b input changed
    IF NOT b'STABLE THEN
      f_assign(f_nand(f_state(a),f_state(b)),
               y,b_in,y_f,y_t);
    END IF;
    END PROCESS;
  END behavioral;
```

Figure 6: Nand Gate Model

```
ENTITY jkff IS
  GENERIC (clr_in, clk_in, q_f, q_t, qb_f, qb_t,
                       clk_setup, clk_hold : TIME);
  PORT (clr, clk, j, k : IN logic;
        q, qb : OUT logic);
  USE standardlogic.ALL;
  END jkff;
```

Figure 7: JKFF Declarations

Both of these models are considerably more complex than might at first appear necessary. Both models include the following capabilities:

- **Accurate Handling of Timing** - in both cases, these models are sophisticated enough to handle back-annotated segment delays which can differ from one input port to another.
- **Full Constraint Checking** - this includes proper handling of spike suppression, and hold and setup checks.
- **Standardized Value System** - both models utilize a standard value system.
- **Standardized Generic Parameters** - both models have employed a standardized approach to naming generic parameters, and both have all the required parameters to allow effective automatic back-annotation of delays from layout.

As a result, these models will be very effective when combined during simulation.

```
ARCHITECTURE behavioral OF jkff IS
  BEGIN
  -- assertion to check for setup errors
  ASSERT (clk'EVENT) AND
         (f_state(clk) = '1') AND
         ((NOT j'STABLE(clk_setup)) OR
          (NOT k'STABLE(clk_setup)))
    REPORT "Setup error";

  -- assertion to check for hold errors
  ASSERT (j'EVENT) AND (NOT clk'STABLE(clk_hold))
    REPORT "Hold error";
  ASSERT (k'EVENT) AND (NOT clk'STABLE(clk_hold))
    REPORT "Hold error";

  -- process to handle asynchronous clear
  PROCESS (clr)
    BEGIN
    IF f_state(clr) = '0' THEN
      f_assign('0',q,clr_in,q_f,q_t);
      f_assign('1',qb,clr_in,qb_f,qb_t);
    END IF;
    END PROCESS;
```

Figure 8: JKFF Exception Handling

```
-- process to handle clocked input
PROCESS (clk)
  BEGIN
  -- check for a down transition in clock
  IF f_state(clk) = '0' THEN

    -- watch out for unknown inputs
    IF (f_state(j) = 'X') OR (f_state(k) = 'X') THEN
      f_assign('X',q,clk_in,q_f,q_t);
      f_assign('X',qb,clk_in,qb_f,qb_t);

    -- calculate the next output state
    ELSE
      CASE f_state(j) IS
        WHEN '0' =>
          CASE f_state(k) IS
            WHEN '0' => -- do nothing
            WHEN '1' =>
              f_assign('0',q,clk_in,q_f,q_t);
              f_assign('1',qb,clk_in,qb_f,qb_t);
          END CASE;
        WHEN '1' =>
          CASE f_state(k) IS
            WHEN '0' =>
              f_assign('1',q,clk_in,q_f,q_t);
              f_assign('0',qb,clk_in,qb_f,qb_t);
            WHEN '1' =>
```

```
            f_assign(f_state(f_not(f_state(q))),
                    q,clk_in,q_f,q_t);
            f_assign(f_state(f_not(f_state(qb))),
                    qb,clk_in,qb_f,qb_t);
          END CASE;
        END CASE;
      END IF;
    END IF;
    END PROCESS;
END behavioral;
```

Figure 9: JKFF Clocked Input

4 Conclusion

Experience with VHDL and other related languages such as HHDL[COEL83,85] and ADLIB[COEL87] have clearly demonstrated that without a set of modelling guidelines and standard packages, a wide variance in coding quality and compatability will exist from one designer to another. For this reason, a standard logic modelling package and associated guidelines are proposed. Although this package is primarily targeted at logic and functional modelling problems, this area does represent the vast majority of usages for HDL's and is the area which most critically needs standardization.

By leveraging existing knowledge in the development of behavioral models, a standard VHDL package such as the one proposed in this paper can be thought of as a **knowledge base** from which other designers can learn and leverage. Further, VHDL users will experience an increase in productivity by following standard practices and utilizing a standard package.

5 References

[COEL83] D.Coelho, "HELIX, A Tool for Multi-Level Simulation of VLSI Systems", International Semi-Custom IC Conference, November 1983

[COEL85] D.Coelho, "High-Level Design Using HELIX", ACM Computer-Science Conference, March 1985

[COEL87] D.Coelho, D.Hill, "Multi-Level Simulation for VLSI Design", Kluwer Academic Publishers, 1987

[VHDL86] IEEE, "VHDL Language Reference Manual", Draft Standard 1076/A, December 31, 1986.

[VHDL87] CAD Language Systems, "VHDL Tutorial for IEEE Standard 1076 VHDL", Draft, May 1987.