

# An Automatic Model Generator for VITAL Compliant ASIC Libraries

Shirley Lu  
Synopsys Inc.  
Mountain View, CA 94043

## Abstract

VHDL Initiative Towards ASIC Libraries (VITAL) Model Development Specification (MDS) v2.2b has passed the ballot and has been turned over to IEEE for standardization. In the meantime, numerous technical issues surrounding VITAL MDS and packages start to be uncovered. An automatic model generator which creates accurate VITAL models comes in handy. ASIC vendors can generate VITAL compliant models with little effort. Generated VITAL models can be simulated and the results can be verified against expected simulation results. This will help in rapid identification of potential enhancements in current VITAL MDS and packages to achieve sign-off quality. Generated VITAL models can also help standardization process as “testcases” for the standard can be easily created. The generated VITAL libraries have very low maintenance cost. They can be re-generated as soon as the model generator is updated for VITAL issue resolution and updates to the MDS.

This paper presents an automatic VITAL model generator. It creates pin-to-pin delay style level-1 compliant models for ASIC libraries. This paper describes in detail the different strategies applied in the model generator for modeling library cells. The correlation between technology library modeling styles and the generated models is also discussed. This model generator provides a good tool for supporting industry-standard VHDL models at very low cost.

## 1. Introduction

The VITAL model generator creates level-1 compliant simulation models from the technology libraries written in Synopsys Library Description

Language. Synopsys Library Compiler parses information in the technology library source file into a library database. The VITAL model generator then takes information from the library database to create simulation models. Figure 1 shows the data flow of the VITAL generator with Library Compiler.

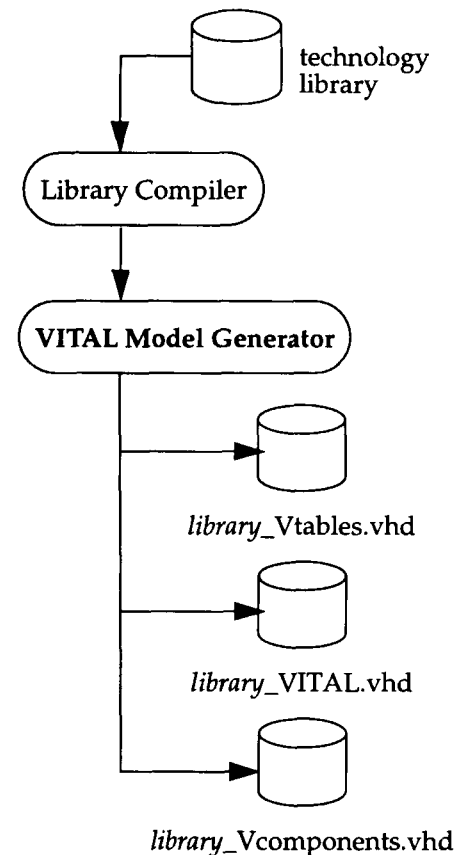


Figure 1. the VITAL model generator

VITAL model generator creates three VHDL files for each ASIC technology library:

*library\_Vtables.vhd:*

VTABLES package containing VITAL truth table and state table constants declaration;

*library\_VITAL.vhd:*

VITAL entity / architecture body;

*library\_Vcomponents.vhd:*

VITAL component package.

As Synopsys Library Description Language supports pin-to-pin delay style, the VITAL model generator creates pin-to-pin delay style level-1 architecture.

## 2. VITAL level-0 Generic Parameters

VITAL level-0 specification defines an interface standard for the names of ports and generics such that back-annotation via Standard-Delay-Format (SDF) mapping can be accomplished. The generated VITAL models are level-0 compliant. All ports use STD\_LOGIC data type. VITAL generic parameters and their corresponding types used in generated models are listed in Table 1.

<b>generic</b>	<b>type</b>
TimingChecksOn	Boolean
XGenerationOn	Boolean
InstancePath	String
tipd	DelayType01
tpd (non-3state)	DelayType01
tpd (3stateable)	DelayType01Z
tsetup	DelayTypeXX
thold	DelayTypeXX
trecovery	DelayTypeXX
trelease	DelayTypeXX
tpw_hi_min	DelayTypeXX
tpw_lo_min	DelayTypeXX
tperiod	DelayTypeXX

**Table 1.** generics and corresponding types in generated models.

The values of interconnect (tipd) and path delay (tpd) generics are extended to type DelayType01Z using VitalExtendToFillDelay() function. The return values of VitalExtendToFillDelay() are then

associated with VitalPropagateWireDelay() and VitalPropagatePathDelay() formalis. As a result, simulation performance can be degraded (VITAL Issue Report #5).

The VITAL generator implements one non-level-0 workaround for conditional path delay generic names. The VITALMDS has not defined a complete mapping from SDF conditional expression to VITAL conditional path delay name (VITAL Issue Report #68). The VITAL generator translates all unmapped SDF operators to "IR68" in generic names as temporary workaround. For example, the SDF conditional path delay

```
(COND !B (IOPATH A Z (1:1:1)(2:2:2)))
```

is mapped to identifier

```
tpd_A_Z_IR68_B
```

because mapping for the operator '!' is not defined.

## 3. VITAL Level-1 Compliancy

VITAL level-1 specification provides a usage model for constructing complete cell models suitable for high-performance sign-off simulation. The generated models are compliant with VITAL level-1 specification except in 2 cases described below.

### Case 1: black box cells (level-0 compliant)

The model generator creates empty architecture body, called black box cells, for library cells when errors are detected in the cell description. An assert statement is put in to the architecture body of each black box cell to warn users about the usage during simulation.

### Case 2: tie-off cells (level-0 compliant)

Level-1 pin-to-pin delay style cannot model tie-off cells because tie-off cells do not have the input/inout ports required for the process sensitivity list. Since acceleration is not a major concern for tie-off cell models, generated VITAL models simply use concurrent signal assignments to tie those output ports to their logic values. For example,

```
HI <= '1';
LO <= '0';
```

### 3.1 Interconnect Delay Section

VITAL level-1 interconnect section models simple wire delay. Interconnect delay section of the generated models is level-1 compliant. Each input port and inout port of a library cell is connected with a `VitalPropagateWireDelay()` routine for interconnect delay modeling.

### 3.2 Timing Check Section

VITAL level-1 timing check section models all timing constraint checks associated with the library cells. The generated VITAL models use two timing violation checking procedures, `VitalTimingCheck()` and `VitalPeriodCheck()`, for constraint checking. Both procedures return Violation flag of type X01.

In the generated VITAL models, each `setup&hold` timing check is modeled by a `VitalTimingCheck()` call. Each `recovery&removal` (`recovery&release`) timing check is modeled also by a `VitalTimingCheck()` call. However, "setup" rather than "recovery", and "hold" rather than "release", become the keywords when recovery and release timing violations are reported. The current VITAL\_Timing package does not support skew checker. Therefore, the model generator ignores skew checker defined in the technology library.

For each clock pin, the clock period constraint and the pulse width constraints, if defined, are bundled into one `VitalPeriodCheck()` call. Pulse width constraints for asynchronous pins use a `VitalPeriodCheck()` call with period checks disabled.

All constraint check enabling conditions are derived from the sequential function defined in the technology library description. Those conditions are actuals in `VitalTimingCheck()` and `VitalPeriodCheck()` procedure calls determining whether constraint checking is enabled. The current timing check enabling conditions are defined in Table 2.

Non-positive setup and hold timing checks are not handled at this moment. The current VITAL MDS does not model non-positive setup and hold timing checks correctly (VITAL issue report #28). The generated models will support non-positive setup and hold time constraints once these are fully supported by VITAL.

### Timing Check

### Enabled If

<code>data setup&amp;hold</code>	clear disable & preset disable
<code>clear recovery&amp;release</code>	preset disable
<code>preset recovery&amp;release</code>	clear disable
<code>clock period&amp;pulsewidth</code>	clear disable & preset disable
<code>clear pulsewidth</code>	always
<code>preset pulsewidth</code>	always

Table 2. Timing Check enabling condition

### 3.3 Functionality Section

VITAL level-1 functionality section models the functional behavior of a library cell. In the generated VITAL models, combinational functions are modeled primarily using overloaded operators in the `std_logic_1164` package and `VitalMUX()`. VITAL primitives such as `VitalBUFIF0()`, `VitalIdent()` are used for three-state modeling and strength mapping. Sequential functions are modeled using `VitalStateTable()`.

For a combinational output, the main functionality is modeled by one boolean equation using overloaded `std_logic_1164` operators. The equation is translated directly from the function defined in the technology library without boolean optimization for predictable ambiguity resolution (e.g. X propagation).

When a combinational cell is recognized as a MUX, the `VitalMux()` primitive, instead of overloaded `std_logic_1164` operators, is used for accurate ambiguity resolution. This replacement can result in differences in ambiguity resolution. In the following example, the three implementations of a 2-to-1 MUX yield different output values when select line S is unknown:

```
-- Input values ----- S := X; A := 1; B := 1
-- Implementation 1 -- Z := X
Z := ((NOT S) AND A) OR (S AND B);
-- Implementation 2 -- Z := 1
Z := ((NOT S) AND A) OR (S AND B)
    OR (A OR B);
-- Implementation3 -- Z := 1
Z := VitalMUX ( data => (B, A),
                dselect => (0 => S));
```

Implementation 2 and 3 resolve ambiguity in select pin S, while implementation 1 is too pessimistic.

VitalBUFIF0(...) is used in the generated models for three-state output functions. The expression for the three-state condition is modeled by a std\_logic\_1164 boolean equation. For example, if data passes through when three-state enable pins B and C are both 0, the output function is modeled as follows:

```
Z := VitalBUFIF0 ( data => A,
                  enable => (B AND C));
```

Sequential functions are modeled by VitalStateTable() calls. Currently the VITAL model generator supports only single-stage flip-flops and latches.

When a VitalTruthTable() or a VitalStateTable() is used to model functionality, the truth table / state table must explicitly define output behavior for all input state X01 combination. Precise ambiguity resolution is required as automatic X reduction is not performed in VitalTruthTable() and VitalStateTable(). The generated tables have don't-care's merged to reduce final table size. They also imply output 'X' assignments by leaving those input combinations out of the generated tables.

As the generated tables are "reduced" tables, the order of table entries becomes very significant, because the table evaluation algorithm searches the table "starting at the top (entry 0) and continues searching until either a level or edge match is found, whichever occurs first", and "the search terminates with the first level or edge match".

All state tables and truth tables are defined, in the generated models, as constants in a separate package VTABLES. This package is used in the generated VITAL architecture. Example 1 shows the generated state table constant for a D flip-flop with rising-edge triggered clock CP, asynchronous clear CD (active high) and preset SD (active high). This cell has two output pins Q and QN, where Q is a non-inverted output and QN, an inverted output. Both outputs remain "nochange" when both clear and preset pins are active.

```
---
--- '0' -- low level
--- '1' -- high level
--- '-' -- don't care on any level (match any level)
--- 'S' -- (output) retains its present value
--- '/' -- 0->1 rising edge
--- 'X' -- unknown level
--- 'B' -- '0' or '1'
---
Constant L: VitalTableSymbolType := '0';
Constant H: VitalTableSymbolType := '1';
Constant x: VitalTableSymbolType := '-';
Constant S: VitalTableSymbolType := 'S';
Constant R: VitalTableSymbolType := '/';
Constant U: VitalTableSymbolType := 'X';
Constant V: VitalTableSymbolType := 'B';
```

```
Constant dfcs_tab : VitalStateTableType := (
--
-- (alg#) refers to the index in state table
-- algorithm following this example
--
-- |----- in ----- | -- out -- | - index -
--
-- Vio CD SD CP D IQ Q QN (entry)(alg#)
--
( U, x, x, x, x, x, U, U ),-- (1) #1
( x, H, H, x, x, L, L, H ),-- (2) #2
( x, H, H, x, x, H, H, L ),-- (3) #2
( x, H, L, x, x, x, L, H ),-- (4) #3
( x, L, H, x, x, x, H, L ),-- (5) #4
( x, x, H, x, x, H, H, L ),-- (6) #5c
( x, H, x, x, x, L, L, H ),-- (7) #6c
( x, L, L, R, L, x, L, H ),-- (8) #7
( x, L, L, R, H, x, H, L ),-- (9) #7
( x, L, L, V, x, x, S, S ),-- (10) #8
( x, L, L, x, L, L, L, H ),-- (11) #9
( x, L, L, x, H, H, H, L ),-- (12) #9
( x, x, L, x, L, L, L, H ),-- (13) #10
( x, L, x, x, H, H, H, L ),-- (14) #11
);
```

Example 1. State table constant for D Flip-Flop

The following algorithm determines how state tables are generated for sequential cells. Assume that a Flip-flop or latch is defined in the technology library by the following attributes:

```
Data: synchronous input function
Clock: clock function (for Flip-Flop)
Enable: synchronous enable function (for latch)
Clear: asynchronous Reset function
Preset: asynchronous Set function
F11: non-inverted output value if both
Clear and Preset are active
```

Fn11: inverted output value if both  
Clear and Preset is active

the choices for F11 and FN11 are '0', '1', 'X',  
nochange (same value as the previous output  
value), and toggle (inversion of the previous  
output value).

The generated state tables implement the following  
algorithm. The numbers in parentheses following  
each item refer to the state table entries in Example  
1.

1. If Violation flag is set (to 'X'),  
assign all outputs to 'X'. (1)
2. Else if Clear is active and Preset is active,  
assign non-inverted outputs to F11 value and  
inverted outputs to Fn11 value. (2) (3)
3. Else if Clear is active and Preset is inactive,  
assign non-inverted outputs to '0's and  
inverted outputs to '1's. (4)
4. Else if Clear is inactive and Preset is active,  
assign non-inverted outputs to '1's and  
inverted outputs to '0's. (5)
5. Else if Clear is unknown and Preset is active,
  - a. if F11 is '1',  
assign non-inverted outputs to '1's;
  - b. else if Fn11 is '0',  
assign inverted outputs to '0's;
  - c. else if F11 is nochange and  
the previous non-inverted outputs are '1's,  
assign non-inverted outputs to '1's;  
if Fn11 is nochange and  
the previous inverted outputs are '0's,  
assign inverted outputs to '0's. (6)
6. Else if Clear is active and Preset is unknown,
  - a. if F11 is '0',  
assign non-inverted outputs to '0's;
  - b. if Fn11 is '1',  
assign inverted outputs to '1's;
  - c. else if F11 is nochange and  
the previous non-inverted outputs are '0's,  
assign non-inverted outputs to '0's;  
if Fn11 is nochange and  
the previous inverted outputs are '1's,  
assign inverted outputs to '1's. (7)
7. Else if Clear is inactive, preset is inactive and  
Enable(latch) / Clock(ff) is active,  
assign non-inverted outputs to Data value and  
inverted outputs to inverted Data value. (8) (9)
8. Else if Clear is inactive, Preset is inactive and  
Enable(latch) / Clock(ff) is inactive,

all outputs remain stable (assigned 'S'). (10)

9. Else if Clear is inactive, Preset is inactive,  
Enable(latch) / Clock(ff) is unknown and Data  
equals to the non-inverted previous output  
value,  
assign output values to the previous output  
value. (11) (12)
10. Else if Clear is unknown, Preset is inactive and  
Data and the non-inverted previous output  
value both equal to '0' and the inverted  
previous output value equals to '1',  
assign non-inverted outputs to '0's and  
inverted outputs to '1's. (13)
11. Else if Clear is inactive, Preset is unknown and  
Data and the non-inverted previous output  
value both equal to '1' and the inverted  
previous output value equals to '0',  
assign non-inverted outputs to '1's and  
inverted outputs to '0's. (14)
12. All missing input combination implies  
outputs assigned 'X's.

For Flip-Flop cells with gated clock, the generated  
models first use a VitalTruthTable() to model the  
combinational function of the gated clock. Then the  
internal clock variable feeds into VitalStateTable()  
as the edge-triggered clock pin. Doing so assures  
accurate edge detection in VitalStateTable().

In the generated models, the number of internal  
state (NumStates) in VitalstateTable() is always 1  
since only single-stage sequential cells are  
modeled. The internal state is feedback from the  
first output pin. The internal state is named as IQ  
in state table constant if the first output has non-  
inverted function, or IQN if the first output has  
inverted function.

### 3.4 Output Driving Strength Mapping

The VITAL primitives and overloaded  
std\_logic\_1164 operators produce output values  
from the value set ['U', 'X', '0', '1', 'Z']. For certain  
technologies requiring a more expanded set of  
output strength mapping, users are allowed to  
convert any of the five output values above to other  
desired output value through a look-up table. This  
look-up table can be associated with the ResultMap  
formals in most VITAL primitives to achieve  
various output strength mapping. The default  
value of ResultMap is ('U', 'X', '0', '1', 'Z'). This

section discusses ResultMap values used in the generated models for different output driving strengths.

CMOS outputs drive strong high, strong low and strong unknown signals. The generated models use default ResultMap value ('U', 'X', '0', '1', 'Z') for strength mapping.

Wired-or ECL outputs yield strong high, weak low and weak unknown signals. The ResultMap value, therefore, is ('U', 'W', 'L', '1', 'Z') in the generated models.

Wired-and ECL outputs yield weak high, strong low and weak unknown signals. The ResultMap value, therefore, is ('U', 'W', '0', 'H', 'Z') in the generated models.

Open-drain outputs, representing pad pins without a pull-up transistor, have the strength of floating high ('Z'), strong low and strong unknown. The ResultMap value ('U', 'X', '0', 'Z', 'Z') is assigned to all open-drain outputs in the generated models.

Open-source outputs, representing pad pins without a pull-down transistor, have the strength of strong high, floating low ('Z') and strong unknown. The ResultMap value ('U', 'X', 'Z', '1', 'Z') is assigned to all open-source outputs in the generated models.

For outputs connected to a VitalBUFIF0() primitive, the ResultMap value is directly associated with the three-state buffer for strength mapping. Otherwise, the generated model adds a strength preserving buffer VitalIdent() to each output with respective ResultMap values associated for strength mapping.

Table 3 lists various technologies and corresponding strength mapping constants associated with ResultMap formal in VitalIdent() or VitalBUFIF0() primitive:

<u>technology</u>	<u>ResultMap value</u>
CMOS	('U', 'X', '0', '1', 'Z') -- default
wired-or ECL	('U', 'W', 'L', '1', 'Z')
wired-and ECL	('U', 'W', '0', 'H', 'Z')
open-drain	('U', 'X', '0', 'Z', 'Z')
open-source	('U', 'X', 'Z', '1', 'Z')

**Table 3.** ResultMap values for various driving strengths

Example 2 shows the function section of a generated model in wired-or ECL technology. This cell models a two-input AND function with inputs A and B. A VitalIdent() primitive is used in this case for output strength mapping.

```
Z_zd := VitalIdent (
    data => (B_ipd) AND (A_ipd),
    ResultMap => ('U', 'W', 'L', '1', 'Z'));
```

**Example 2.** AND gate with wired-or output mapping

### 3.5 Pull-up / Pull-down Resistor Modeling

The model generator models pull-up / pull-down resistors on ports by assigning ResultMap values to the VITAL primitives. Pull-up ports are modeled by mapping output value 'Z' to 'H'. Pull-down ports are modeled by mapping output value 'Z' to 'L'. The resulting ResultMap values are listed in Table 4:

<u>technology</u>	<u>Output with pull-up resistor</u>
CMOS	('U', 'X', '0', '1', 'H')
wired-or ECL	('U', 'W', 'L', '1', 'H')
wired-and ECL	('U', 'W', '0', 'H', 'H')
open-drain	('U', 'X', '0', 'H', 'H')
open-source	('U', 'X', 'H', '1', 'H')

<u>technology</u>	<u>Output with pull-down resistor</u>
CMOS	('U', 'X', '0', '1', 'L')
wired-or ECL	('U', 'W', 'L', '1', 'L')
wired-and ECL	('U', 'W', '0', 'H', 'L')
open-drain	('U', 'X', '0', 'L', 'L')
open-source	('U', 'X', 'L', '1', 'L')

<u>input / inout with pulled resistor</u>	
Pull-up:	('U', 'X', '0', '1', 'H')
Pull-down:	('U', 'X', '0', '1', 'L')

**Table 4.** ResultMap values for pull-up and pull-down ports

For an output (or inout) port with a pull-up / pull-down resistor, it is assumed that the output (or inout) is three-stateable port. Therefore, the respective pull-up / pull-down ResultMap value is associated with the VitalBUFIF0() primitive connected to the output (inout) port. Example 3 illustrates three-stateable output Z with a pull-up resistor.

```
Z_zd := VitalBUFIF0 (
    data => A_ipd,
    enable => EN_ipd,
    ResultMap => ('U', 'X', '0', '1', 'H'));
```

**Example 3.** Output with a pull-up resistor

For an inout port with pull-up (or pull-down) resistors, a VitalIdent() primitive maps input strength 'Z' to 'H' (or 'L') by assigning ResultMap to appropriate value as described in Table 4. The intermediate signal is then connected to the combinational or sequential function. Example 4 illustrates three-stateable inout port IO with a pull-up resistor.

```
IO_ipd2 := VitalIdent(
    data => IO_ipd,
    ResultMap => ('U', 'X', '0', '1', 'H'));
IO_zd := VitalBUFIF0 (
    data => A_ipd,
    enable => EN_ipd,
    ResultMap => ('U', 'X', '0', '1', 'H'));
...
```

**Example 4.** Output with a pull-up resistor

Input ports should not have pull-up or pull-down resistors in the models because they should be visible outside of the cell.

### 3.6 Path Delay section

In the generated models, all path delays related to one output signal are packed into a VitalPropagatePathDelay() procedure call. Non-conditional path delays always have the path-delay condition set to TRUE. Conditional path delays have the path-delay conditions set according to the declaration in the technology library. In the technology library definition, users are required to enter a "default" path delay for those conditions not satisfying any of the specified, especially when certain input values are unknown. In the generated models, a "none-of-the-above" condition is derived from all declared conditions as the default path delay condition. For example, a conditional path delay has the following state dependency in SDF file:

```
(DELAY
  (ABSOLUTE
    // condition #1
    (COND (B==1'B1 || C==1'B1)
      IOPATH A Z (:1:) (:4:))
    // condition #2
    (COND (B==1'B0 && C==1'B0)
      IOPATH A Z (:3:) (:2:))
    // default path delay
    (IOPATH A Z (:3:) (:4:))
    ...
  )
)
```

these path delay conditions are modeled in VitalPropagatePathDelay() as:

```
cond. 1: (To_X01(B OR C) = '1')
cond. 2: (To_X01((NOT B) AND (NOT C)) = '1')
default: (To_X01((B OR C) OR ((NOT B) AND
  (NOT C))) /= '1')
```

And the complete VitalPropagatePathDelay() call is generated as follows.

```
VitalPropagatePathDelay(
  OutSignal => Z,
  OutName => "Z",
  OutTemp => Z_zd,
  Paths => (
    0 => (A_ipd'last_event, -- condition #1 --
      VitalExtendToFillDelay(
        tpd_A_Z_OP_B_EQ_1_TI_B1_OR_C_EQ_1_TI_B
        1_CP),
        (To_X01(B_ipd OR C_ipd) = '1')),
    1 => (A_ipd'last_event, -- condition #2 --
      VitalExtendToFillDelay(
        tpd_A_Z_OP_B_EQ_1_TI_B0_AN_C_EQ_1_TI_
        B0_CP),
        (To_X01((NOT B_ipd) AND (NOT
        C_ipd))='1')),
    2 => (A_ipd'last_event, -- default --
      VitalExtendToFillDelay(tpd_A_Z),
        (To_X01((B_ipd OR C_ipd) OR
        ((NOT B_ipd) AND (NOT C_ipd)))
        /= '1'))),
  GlitchData => Z_GlitchData,
  GlitchMode => MessageOnly,
  GlitchKind => OnDetect);
```

As a result, only one condition will be active in any given input conditions.

### 3.7 Output-to-output Delay Modeling

VITAL v2.2b specification does not support output-to-output delays in level-1 pin-to-pin delay modeling (VITAL Issue Report #16). It is consistent with SDF v2.1 specification, which does not support output-to-output path delays. Therefore, current generated models do not support output-to-output delays.

### 3.8 Bus / Bundle Port Modeling

All bus / bundle ports are bit-blasted in the generated models. As a result, all ports in generated models are scalar.

## 4. Testbench Generation

The model generator creates testbenches for different levels of model validation and verification. Individual cell testbenches as well as whole library testbench are both created.

## 5. Limitations and Future Work

The current release of VITAL model generator has the following restrictions:

1. The VITAL model generator is intended for creating level-1 compliant models. Non-compliant workaround is generally not offered to temporarily solve VITAL open issues. Workaround, however, is provided under special circumstances when simulation speed (acceleration) is not a concern. The generated models, therefore, provide true implementation of VITAL v2.2b MDS where unresolved issues can be observed. The model generator will be kept up to date with approved issue resolution.

2. The current VITAL model generator does not generate distributed delay style models.

3. Current release of the VITAL model generator only handles single-stage sequential cells. Work is under way to handle multi-stage sequential cells such as Master-slave latches and Flip-Flops.

4. Currently the VITAL model generator uses overloaded `std_logic_1164` operators to model combinational functions. A potential enhancement for the model generator is to allow users to choose

between using overloaded `std_logic_1164` operators and using `VitalTruthTable()` to model combinational functions.

5. The VITAL model generator does not model Flip-Flops as back-to-back latches. It models Flip-Flop functions strictly as edge-triggered devices. As a result, ambiguity resolution may not be satisfactory. If back-to-back latches are preferred as the Flip-Flop models, manual modification is required. A potential enhancement for the model generator is to allow users to specify the preference of back-to-back latches for Flip-Flop functions.

6. The VITAL generator does not model non-positive setup / hold time constraints pending resolution approval on VITAL Issue Report #28.

## Summary

A VITAL model generator has been presented in this paper. This level-1 compliant model generator provides a useful tool for:

- superb data integrity and accuracy;
- fast industry adoption of VITAL standard;
- efficient testcase creation for VITAL package verification;
- VITAL standard validation;
- fast identification of sign-off modeling issues;
- rapid turn-around for future packages and standard update.
- low maintenance cost.

## References

VITAL Model Development Specification, Version v2.2b, March 23, 1994

IEEE Standard VHDL Language Reference Manual (IEEE std 1076), 1987

Standard Delay Format Specification, Version 2.1, February 1994

Synopsys Library Compiler Reference Manual, Volume 1, Version 3.1a