Practical Experiences in Modeling ASIC Libraries for Sign-Off using the IEEE VITAL Standard

Ekambaram Balaji, Design Engineer
LSI Logic Corp.
1501, McCarthy Blvd, M/S E-192
Milpitas, CA 95035
ebalaji@lsil.com

Ajayharsh Varikat, Member of Technical Staff
Cadence Design Systems (India) Pvt. Ltd.
SDF # A1/B8, NEPZ,
Noida - 201 305, UP, India
ajay@cadence.com

Abstract

The IEEE VITAL standard provides modeling guidelines and a set of predefined packages which facilitate the acceleration of designs during simulation using models from VITAL compliant library. In this paper, we highlight the various VITAL features used in a typical ASIC library, the associated modeling trade-offs, a development approach based on existing source libraries and related issues. The paper also describes few limitations of VITAL in addressing sign-off requirements and alternative implementations developed for addressing such requirements.

1. Introduction

The IEEE VITAL specification addresses issues of interoperability, backannotation and high performance simulation for sign-off quality ASIC libraries in VHDL. Even while using the coding styles provided in the specification, there are various ways in which a VITAL compliant model can be written. This paper brings out the experiences and findings of the authors while developing VITAL models for ASIC macrocells available in LSI Logic's array and cell based technologies. It gives a detailed description of the various implementation strategies and issues involved in efficient modeling of ASIC macrocells using VITAL.

In order to accelerate the availability of VITAL macrocell libraries, it is a logical choice for ASIC foundries to adopt the approach of producing models from one of their existing source libraries. We present some of our experiences in using this approach to develop VITAL libraries. The development strategy at LSI was to use existing sign-off quality verilog models for translation to VITAL models. Finally, we show how certain intricate modeling issues can be addressed at the cost of performance and Level-1 compliance. The paper also brings out some of the limitations and modeling issues that could be addressed in future enhancements to the VITAL standard.

2. LSI Logic VITAL Libraries

LSI Logic offers technology libraries corresponding to the two common design methodologies: array-based (ica) and cell based (icb). The ASIC macrocells can be broadly classified as internal cells and IO cells. The internal cells comprise of combinational gates, internal buffers, flip-flops, latches, clock drivers, oscillators etc. The IO cells consist of unidirectional & bidirectional IOs and pad cells. The complexity of these cells range from simple logic gates to more complex sequential cells with multiple clocks. All the above functions have to be represented in VITAL models. In addition, they have to meet sign-off requirements for glitch detection, functional correlation with LSI's internal simulator, simultaneous transitions, timing violation handling and behavior of special cells.
The LSI Logic VITAL libraries are fully compliant with VITAL standard as specified in the VITAL ASIC modeling specification, IEEE 1076.4. These models use the VITAL Level-1 and Level-0 modeling styles and the standard packages. The timing calculations are done outside the model using LSI’s delay calculator (ls delay) and introduced into the model using VITAL-SDF back-annotation scheme. The next section lists the features of LSI Logic VITAL models to give the readers an idea about the typical usage scenarios of the VITAL standard. Not all features provided by the VITAL standard are used in these models.

3. Features of VITAL Models

A VITAL model is a VITAL compliant VHDL representation of an ASIC cell. This section gives a detailed description of the contents of a VITAL model including macrocell support package used within the models.

3.1 VITAL Compliance

All the models are VITAL Level-0 compliant and most of them are Level-1 compliant. The non Level-1 compliant models need special modeling techniques (behavioral modeling) which are illegal according to the VITAL specification. Certain special cells with analog nature and cells with skew check may fall into this category. SDF 2.1 is used for delay back-annotation.

3.2 Interface Declaration

Interface declaration consists of entity definition specifying input/output ports and VITAL timing generics to hold the back-annotated delay information. All ports are scalars of standard 1164 base type std_uglogic. Normally the inputs are initialized to a ‘Z’ the outputs are initialized to ‘X’ values. Ports involving strength specifications such as pullup/pulldown are initialized with appropriate values, e.g. and ‘H’ or an ‘L’.

The VITAL timing generics used are tp, tpd_and tsetup/thold/trecovery/tpw generics. The naming conventions are according to the VITAL specifications. The vital delay type for timing generics is one of VitalDelayType, VitalDelayTypeEnum, or VitalDelayType01Z. The control generics are defined to control the usage of the model. The default values for all these generics are defined in the global macrocell package. These generics are defined as follows:

\[
\begin{align*}
\text{MsgOn} & \quad \text{BOOLEAN} \quad \text{DefTimingMsgOn}; \\
\text{XOn} & \quad \text{BOOLEAN} \quad \text{DefTimingXOn}; \\
\text{TimingChecksOn} & \quad \text{BOOLEAN} \quad \text{DefTimingChecksOn}; \\
\text{InstancePath} & \quad \text{STRING} \quad \text{***};
\end{align*}
\]

The models provide the capability to turn on/off certain control generics on a per instance or global basis. Features like constraint checking, timing check messages and X-generation can be controlled according to user requirements.

In addition to these the entity declaration also contains the VITAL Level0 compliance attribute. This attribute is set to TRUE in order for VITAL compliant simulation tool to perform VITAL compliance checks.

\text{ATTRIBUTE Vital_level0 OF FD3: ENTITY IS TRUE;}

3.3 Functional Implementation

A VITAL Level 1 architecture declaration section may consist of signals, attributes, constants and aliases. It must contain the attribute specification

\text{ATTRIBUTE Vital_level1 OF FD3_ARCH: ARCHITECTURE IS TRUE;}

This attribute when set to TRUE indicates adherence to the VITAL Level 1 specification. The declaration section contains all internal signals including those coming from wire delay blocks and signal delay block. All these signals are of the type std_uglogic. Sample declaration is as shown below:

\[
\begin{align*}
\text{SIGNAL CP_ipd} & \quad \text{std_uglogic} \quad \text{:= 'X';} \\
\text{SIGNAL CD_dly} & \quad \text{std_uglogic} \quad \text{:= 'X';}
\end{align*}
\]

A VITAL process is a key building block of a Level-1 architecture. A process consists of declarative part and statement part. The declarative part consists of variable declarations and the statement part consists of statements that describe the timing constraint checks, cell function and path delay selection. Although each section is optional, a process must contain at least one of these sections. All VITAL processes must have a sensitivity list. Each and every signal read in a process must appear in the sensitivity list of that process. The sequence of various sections within the process is significant. The number of process blocks will be determined based on the cell characteristics, but in general a single process model is adopted for most of the LSI Logic cells. Multiple process models may be used in some cases such as master-slave flip-flops or multi-storage devices like quad D latch.

The functionality section defines logical function of the model. This computes the new values of the output based on the input values in zero delay. This section consists of a sequence of variable assignment statements and/or calls to VitalStateTable/VitalTruthTable.
Combinational models are implemented using a network of VitalPrimitives. The function form of primitives are used from the VITAL_Primitives package. In some cases like multiplexors the VitalTruthTable definitions are used. Sequential cells are implemented using VitalStateTable definitions. Most of the cells are implemented using single output tables while some of the complex multiple stage storage devices may use intermediate state and output columns in VITAL state table description.

The combinational and sequential UDPs in the input verilog model are translated as VitalTruthTable and VitalStateTable respectively. The following is a sample implementation of functionality:

\[
\text{CDSD} := \text{VitalAND2}(\text{CD} \_ \text{dly}, \text{SD} \_ \text{dly}) ;
\]

\[
\text{VitalStateTable} \left(\begin{array}{l}
\text{StateTable} \Rightarrow \text{LSIPD3Q}, \\
\text{PredDataIn} \Rightarrow \text{PredDataIn} \_ \text{V}, \\
\text{Result} \Rightarrow \text{Q} \_ \text{V}, \\
\text{DataIn} \Rightarrow (\text{D} \_ \text{dly}, \\
\text{CP} \_ \text{dly}, \\
\text{CD} \_ \text{dly}, \\
\text{SD} \_ \text{dly}, \\
\text{notifier})
\end{array}\right)
\]

Additional gate level logic is needed for timing check conditioning. The VitalStateTables are coded to reduce pessimism as far as possible. The functional implementation ensures correlation with LSI's internal simulator. The VitalStateTable includes the handling of violation flags used in VITAL timing check procedures.

3.4 Timing Implementation

VITAL Wire delay blocks are used to model interconnect delays. This block includes calls to the procedure VitalWireDelay to propagate wire delays. The tpd_* generics of the type VitalDelayType01Z are used to back-annotate wire delay information. The following shows a sample wire delay block:

\[
\text{WireDelay} := \text{BLOCK}
\]

\[
\begin{align*}
\text{BEGIN} \\
\text{VitalWireDelay(D_ipd, D, tpd\_D);} \\
\text{VitalWireDelay(CP_ipd, CP, tpd\_CP);} \\
\text{END \ BLOCK;}
\end{align*}
\]

VITAL Signal delay blocks are created for modeling negative timing constraints(NTCs) in all sequential models. This is done for easier maintainability of these models. This block includes calls to the procedure VitalSignalDelay to propagate signal delays for adjusting negative constraint timing. The delayed signals are used throughout the model. Negative timing constraints are used irrespective of whether the model has negative timing values or not. This is done so as to be able to develop and maintain technology independent VITAL functional templates. This may have some performance implications.

\[
\text{SignalDelay} := \text{BLOCK}
\]

\[
\begin{align*}
\text{BEGIN} \\
\text{VitalSignalDelay(D\_dly, D\_ipd, tisd\_D\_CP);} \\
\text{VitalSignalDelay(CP\_dly, CP\_ipd, tisd\_CP);} \\
\text{END \ BLOCK;}
\end{align*}
\]

A pin-to-pin delay model is used in all VITAL models. The VitalPathDelay procedures are used for signal scheduling inside a VITAL Level 1 process. All paths to a given output are represented using a call to the procedure VitalPathDelay. Depending on the delay type for a given path one of VitalPathDelay01 or VitalPathDelay01Z are used. These procedures perform transition dependent delay selection. After functional computation of the output value in zero delay, the path delay schedules the output value after the time specified by the tpd_* generics. All delay paths including combinational cells have input edge dependency. Depending on which edge transition occurred at the input, the appropriate delay is used.

Glitch handling in a VITAL Level 1 model is incorporated into the signal scheduling mechanism. The mode of glitch detection used is OnDetect which is consistent with LSI's sign-off requirements. The "X-generation" and glitch messages during glitch detection can be controlled globally via constants defined in the macrocell package.

\[
\text{VitalPathDelay01}(
\begin{align*}
\text{OutSignal} \Rightarrow \text{Q}, \\
\text{OutSignalName} \Rightarrow \text{"Q"}, \\
\text{OutTemp} \Rightarrow \text{Q} \_ \text{V}, \\
\text{Paths} \Rightarrow (0 \Rightarrow ( \\
\text{InputChangeTime} \Rightarrow \\
\text{CP\_dly}'\text{Last}_\text{Event}, \\
\text{PathDelay} \Rightarrow \\
\text{tpd\_CP\_O\_posedge,} \\
\text{PathCondition} \Rightarrow ( \\
\text{posedge}(\text{CP\_dly}'\text{Last}_\text{Value}, \text{CP\_dly}) \\
\text{)}) \\
), \\
\text{GlitchData} \Rightarrow \text{Glitchdata0}, \\
\text{Mode} \Rightarrow \text{OnDetect,} \\
\text{MagOn} \Rightarrow \text{MagOn,} \\
\text{XOn} \Rightarrow \text{XOn}
\end{align*}
\)

One VitalPathDelay procedure is used for each output of the macrocell. The 'posedge' and 'negedge' are two dimensional constants declared in the LSI Logic macrocell support package.
3.5 Timing Constraint Handling

The VITAL timing check procedures are used to model timing constraints. Each setup/hold check is combined into one VitalSetupHoldCheck() procedure. Similarly each recovery/hold check pair is combined into one VitalRecoveryHoldCheck(). A call to VitalPeriodPulseCheck() is used to perform minimum pulse width checks.

All timing checks are performed on the delayed signal after the signal delay block. However the names of the test and reference signal in the call to the VITAL timing check procedures refer to the port names. The timing check procedures are conditioned to reduce pessimism using the CheckEnable parameter in the vital procedure. Additional gates are used to derive the conditioning signal. Following illustrates a typical call to the VITAL setup/hold check procedure:

```c
VitalSetupHoldCheck(
    TestSignal => D_dly,
    TestSignalName => "D",
    RefSignal => CP_dly,
    RefSignalName => "CP",
    setupHigh => tsetup_D_CP_posedge_posedge,
    setupLow => tsetup_D_CP_negedge_negedge,
    holdHigh => thold_D_CP_posedge_posedge,
    holdLow => thold_D_CP_negedge_negedge,
    CheckEnabled => (
        and (CDSD = '1' or CDSD = 'H')
    ),
    RefTransition => 'R',
    TimingData => marker1,
    MsgOn => MsgOn,
    XOn => Xon,
    Violation => violationFlag1
);
```

Following the timing check portion, is the timing check violation handling. If there is any timing violation, the appropriate violation flag is set by the VITAL timing check procedure and during a timing violation a violation flag is set to "X", and as a result the VitalStateTable() produces an "X" on the output. A subsequent set of valid inputs causes the output to come out of the X. Violation flags are ORed together to realize a toggling effect. If the ORed value is "X" then a single violation flag called 'notifier' is toggled. The following piece of code illustrates this:

```c
notifier := XTo1(violationFlag1 OR violationFlag2) XOR XTo0(notifier);
```

This transition on the notifier object is detected by the VitalStateTable() procedure and the appropriate action for "X-Handling" is taken.

4. VITAL Macrocell Package

The VITAL macrocell support package has been developed with the intent of keeping global and technology independent information centralized. It consists of non-standard gates (switch level primitives), signals, constants for initialization and specialized edge detection. The global constant definitions denote the default values for the generic parameters used in the model. This package also contains all the function table definitions used by the procedures, VitalTruthTable and VitalStateTable. Tables are kept in separate package for efficiency reasons. Defining tables within models would set aside that much memory for every instantiation of the model. Referencing it from a package avoids this.

5. VITAL SDF Backannotation

The timing values in the VITAL models are introduced by using the VITAL-SDF back-annotation scheme. All the macrocell models are VITAL-Level0 compliant and hence follow the VITAL naming conventions for naming timing generics. The VITAL-SDF files are generated by running the delay calculator (Isidelay) on the design. The VITAL Level-0 compliant annotation tool was used to back-annotate the SDF files. The timing arcs in SDF must be consistent with the timing generics in the VITAL cell models and should have a one-to-one mapping. There are no restrictions on the usage of the generic. Also conditions can be omitted from the generic parameter while the path delay or timing check procedure could have conditions.

Delay triplets(min:typ:max) are produced in the SDF file for all timing values even if the delays are the same. The VITAL compliant annotation tool will provide the flexibility to choose from one of min:typ or max delay values. State dependent delays will not be supported in the LSI VITAL models. The output-to-output timing arcs are not supported directly. Following is the list of SDF constructs and the corresponding generics used in the LSI Logic VITAL models.

All interconnect delays are represented using 'tipd_' generics. These generics do not support edge specifiers. All propagation path delays are represented using 'tpd_' generics. These generics support edge specifiers on the input port in LSI libraries. These edges are always specified even if delays from different edges of input to the specified output are same. The setup, hold, recovery, width times are represented using 'tsetup_*/thold_*/trecovery_*/tpw_*/' generics. Skew timing checks are not yet fully supported by VITAL. They are supported only for back-annotation.
6. Verilog To VITAL Translation

Producing high quality models in a short time is a key factor for the success of VITAL. Verilog libraries are used for sign-off and are considered golden by a majority of ASIC vendors including LSI Logic. This was a strong motivation to adopt a solution to convert existing Verilog libraries to VITAL. This was done using the VITALizer tools from Cadence. This translator generates VITAL models that have the same functionality as the input Verilog models. The following section describes some of the salient features of the translation tool.

6.1 Features of Translation

The translator from Cadence is not a strict one-to-one translation from Verilog, instead it was found to have intelligence to perform certain optimizations. This tool was qualified to meet the requirements set forth by LSI Logic. The translator provides high coverage which means almost all the cells in the library can be automatically translated from Verilog to VHDL.

Timingcheck and delay generics are generated from the information provided in the Verilog model. It provides the option of leaving out condition or edge part from timing generics for greater SDF compatability. Additionally generics to support NTCs are also generated. The number of delay elements in pathdelay generics (1, 2 or 6) is decided based on the need to select one of VPD, VPD01 or VPD01Z to the given output. The number of delays selected would be the maximum specified among all path delays to the given output port.

Verilog models would typically use a hold check inplace of a removal check. The translator recognizes this situation and generates removal generics and merge the recovery/hold pair in the Verilog model to generate a VitalRecoveryRemovalCheck() in the VITAL model. Separate setup/hold, period/width checks in the verilog model are also combined into single check in VITAL.

The translator supports different modeling styles such as the single process and the concurrent style. In a single process model the timing check procedures, functionality and path delays are specified in that order within a single VHDL process. In a concurrent style model, zero delay primitives and one VHDL process each for timing checks and path delays are used. For verilog models that have internal feedback, only the later approach is possible. For models without feedback, it is generally found better to use one process for each parallel module path (a parallel module path here refers to a maximal subset of the netlist that does not interact with any other part of the netlist). For example a Quad-D latch should be implemented as a multi-process model. In single process style primitives have to be correctly ordered and the functionality will have to be implemented in a reader-writer order. This is done by doing a topological sort on the netlist.

The translator performs a one-to-one translation of the verilog primitive to corresponding VITAL primitive. Special gates (pmos,nmos etc),signals (trireg, triand etc) are handled by tables specified in the support package. The translator does not produce any redundant code in the VITAL model.

6.2 Issues with Verilog to VITAL Translation

The above approach does introduce few modeling issues that need to be resolved due to the inherent differences in the semantics of the two languages viz., Verilog and VHDL. We discuss in this section some of the translation and related issues.

- Gates like pullup, tri1 etc can be implemented with the resolution function and by initializing the corresponding signal to suitable values. e.g. if a pullup has multiple drivers, it can be implemented by having an additional 'H' driving it in the resolution function. If it has only a single driver, then only an initialization of the signal to 'H' is required. In general all multiple drivers in Verilog have to be resolved in VITAL using the IEE resolution function.

- The Verilog libraries had to be pre-processed before being input to the translator. This is done to take advantage of certain features supported in VITAL but not supported in Verilog such as input edge sensitivity in propagation delay timing arcs. If edges are specified in verilog models, these are implemented in the VITAL models, even though verilog ignores the edge specifications.

- Redundant gates have been used in the verilog model to support MIPD back-annotation for input/inout pins that have no fanout. These are not required for VITAL and so the verilog libraries have to be pre-processed for these before feeding it to the translator.

- Reading output ports: This is done by driving the delayed output signal onto a temp signal. The temp signal is then driven onto the output port through a VitalDENT primitive. The temp signal can then be read within the functionality as the delayed value of the output port. In single-process style, this signal should appear in the sensitivity list of the process.
• There are semantic differences in the way Verilog UDPs work and the way in which VITAL tables function. Simultaneous transitions in sequential UDPs is highly implementation dependent in Verilog and may produce simulation mismatches in VITAL. Another significant difference is the concept of dominance in verilog where a level sensitive entry dominates an edge sensitive entry. VITAL table entries do not have any dominance defined and during simulation it chooses the first matching entry in the table. During translation this problem is eliminated by having the rows with level sensitive entries placed at the beginning of the VITAL table.

• To get simulation behavior matching with verilog (so that udps work correctly), the notifier is toggled every time there is an 'X' on any of the violation flags in the VITAL procedure. This may have some performance implications.

• In verilog, separate conditions can be specified on the test and reference signals in timing checks. VITAL timing check procedures allow conditions only on later /time check event.

7. VITAL Modeling Issues

VITAL is an emerging standard. The IEEE VITAL specification addresses a majority of the issues encountered while developing ASIC libraries. This section describes the issues in the existing VITAL standard. These issues could be addressed in future enhancements to the standard to make it much more robust.

7.1 Issues with VITAL standard

• Verilog uses concurrent UDPs which get triggered only when an input changes. In the single-process style in VITAL, a process that has a state table can get triggered even when none of the state table input change. In order to avoid an 'X' on output in such cases, a no-change entry has to be added in the VITAL table. This is eliminated by having an all 'S' entry at the end of the table. The 'S' entry currently matches "00" or "11" transitions only. It does not include "X-to-X" transitions. This causes the output to go to 'X', if there is a steady input at X value and the table is evaluated without any change in its inputs.

• While modeling negative timing constraints it is not possible to use the same signal as both test and the reference signal. Hence, additional internal signals have to be derived from the same signal source to circumvent this problem. Also it is observed that if NTCs are specified in the VITAL model, the timing checks will be reported interchangeably for example: a hold for setup and setup for hold will be reported as a setup in the negative domain is same as a hold in the positive domain. Timing checks on the output ports does not work properly with negative constraints.

• Due to different internal delays produced for test and reference signals in negative timing constraints the timing check condition may be false within a small window of time but the check will still be performed.

• VITAL restricts the use of deferred constants. The VITAL table definitions reside in a package and even if one entry in the VitalStateTable () changes, the package has to be recompiled and consequently, the whole library has to be recompiled. If deferred constants were allowed only the package body needs to be recompiled.

• Conditioning is possible only on the later of the two events. There is an inconsistency here with generic name rules. Conditions can be specified for each signal in the generic names for timing check generics, while the corresponding timing checks do not support conditions on both signals.

• If data and clock change simultaneously and one of the setup or hold timing limit happens to be zero, VITAL always reports a setup violation. Hence if the setup value is zero no violation will be reported. This is different from Verilog behavior where a hold violation is reported in such cases.

• In a multi process model the flow of data between processes was achieved by having zero delay VPDs. This could have some performance implications. If VITAL could provide a specialized zero delay signal assignment statement (e.g. VitalAssign) then communication between processes could be simpler.

7.1 Issues in Efficient VITAL Modeling

7.1.1 Signals vs. Variables

A well known characteristic of gate-level simulation is that less than 2% of events get propagated to the next level of logic. Primitives or other procedures which are instantiated concurrently get activated only on signal value changes on their inputs. Therefore, a model having a set of concurrent primitives representing a netlist incurs a reduced amount of evaluation because of this characteristic.

On the other hand, using signals as the means of propagating values has a significant overhead in terms of scheduling and descheduling of events. In general, signals are expensive to use than variables in VHDL.
As an alternative to modeling with a network of concurrent primitives, one can use variables to communicate values inside the VITAL behavior process. This style reduces the overhead of scheduling and descheduling signal values.

7.1.2 Single process vs. Multiple processes

For modeling and simulation efficiency, the choice of using one or more processes should be dictated by the circuit topology. While a single process enables an efficient representation for combinational and single-stage sequential circuits, multiple processes are very useful in modeling multi-stage sequential circuits, such as master-slave flip-flop, and sequential circuits with feedback. Also, modeling these classes of circuits in a single process leads to designing large and complex state-tables which are error-prone and also result in inefficient simulation. Similarly, for cells where all input signals do not affect all outputs, use of multiple processes is more efficient because it ensures that only the minimum of computation is done for the relevant part of the circuit whenever an input changes.

Due to the VITAL restriction that all signals read in a process appear in the sensitivity list, multi-process models perform worse than single-process ones. If this restriction is removed and we place only the zero-delay signals in the pathdelay process, concurrent style models can outperform sequential style models. This is because most of the input changes typically die out before reaching the output. Concurrent style models would then call pathdelay procedures only for those cases where the output actually changes. This is an important efficiency consideration.

7.1.3 State/Truth Table vs. VITAL Primitives

Within a VITAL process, there are alternate ways of modeling functionality. One can use a Truth/State table or can use a sequence of IEEE operators and/or VITAL primitive function calls. When choosing either methodology the trade-off to be considered are - ease of modeling, performance and memory usage. For combinational logic, using operators/logic primitives is simple and intuitive and is more or less equivalent to writing the corresponding boolean equation. However, even for combinational logic, a truth table can be made. For sequential cells, expressing in terms of state tables is usually easy.

Modeling using tables typically requires more skill and care has to be taken to make sure that the tables do not result in too optimistic/pessimistic outputs. However, this can offer significantly higher performance because of less number of events generated and propagated. The other aspect to be considered is memory usage. While small optimized tables can be implemented with acceptable memory overheads, larger tables often result in excessive memory overheads.

7.1.4 Use of locally static expressions

In various places VITAL Level-1 style requires that an expression be globally static. While writing models, wherever possible, the user should try and use locally static expressions.

8. Non-Level1 VITAL Models

One area that VITAL does not adequately address at present is skew checks. Owing to the specialized requirements in the LSI library, we had to develop a timer based skew-check procedure. This was used in non level1 VITAL models.

Two procedures called IsiOutPhaseSkewCheck() and IsiInPhaseSkewCheck() are defined to address skew check functionality. These procedures check for a maximum skew time between two signals. The skew check procedures are time based as opposed to event based($skew in verilog). The action on skew timing violation is taken when the skew limit transpires and subsequently when the required state of the two signal is established (when the destination event happens) the circuit returns to normal behavior. One issue in a timer based skew check was the inability to use VHDL wait statement within the procedure. This was worked around by scheduling a dummy event on a trigger signal within the procedure. The trigger signal is sensitive to the process and when the limit transpires the process is triggered for reporting a violation.

The IsiOutPhaseSkewCheck() procedure detects an out-of-phase skew violation between input signals Signal1 and Signal2. This is a timer based implementation in which a violation is detected if Signal1 and Signal2 are in the same logic state longer than the specified skew interval. The timing constraints are specified through parameters representing the skew values for the different states of Signal1 and Signal2.

The IsiInPhaseSkewCheck() procedure detects an in-phase skew violation between input signals Signal1 and Signal2. This is a timer based implementation in which a violation is detected if Signal1 and Signal2 are in different logic states longer than the specified skew interval. The timing constraints are specified through parameters representing the skew values for the different states of Signal1 and Signal2.
The interface specifications for two types of skew checks including their default initialized values are shown below:

```c
PROCEDURE lsiOutPhaseSkewCheck ()
VARIABLE Violation:OUTX01;
VARIABLE SkewData :INOUT lsiSkewDataType;
SIGNAL Signal1 :IN std_ulogic;
CONSTANT Signal1Name :IN STRING := "";
SIGNAL Signal2 :IN std_ulogic;
CONSTANT Signal2Name :IN STRING := "";
CONSTANT SkewS1S2RiseFall :IN TIME;
CONSTANT SkewS1S2RiseFall :IN TIME;
CONSTANT SkewS1S2FallRise :IN TIME;
CONSTANT SkewS1S2FallRise :IN TIME;
CONSTANT Signal1Delay :IN TIME := 0 ns;
CONSTANT Signal2Delay :IN TIME := 0 ns;
CONSTANT CheckEnabled :IN BOOLEAN := TRUE;
CONSTANT XonS1S2RiseFall :IN BOOLEAN := TRUE;
CONSTANT XonS1S2FallRise :IN BOOLEAN := TRUE;
CONSTANT MsgOnS1S2RiseFall :IN BOOLEAN := TRUE;
CONSTANT MsgOnS1S2FallRise :IN BOOLEAN := TRUE;
CONSTANT MsgOnS1S2RiseFall :IN BOOLEAN := TRUE;
CONSTANT MsgOnS1S2RiseFall :IN BOOLEAN := TRUE;
CONSTANT MsgOnS1S2FallRise :IN BOOLEAN := TRUE;
CONSTANT MsgOnS1S2FallRise :IN BOOLEAN := TRUE;
CONSTANT MsgSeverity :IN SEVER_LEVEL := WARNING;
CONSTANT HeaderMag :IN STRING := "";
SIGNAL trigger :INOUT std_ulogic ;
);```

9. Highlights and Lowlights of VITAL

The VITAL standard derived some of its features from existing modeling techniques. Verilog being a popular and well accepted industry standard HDL, some of the modeling principles were adopted from Verilog. It seems that in doing so, the VITAL committee wanted to eliminate some of the deficiencies found in Verilog modeling of ASIC cells. However, VITAL does have some drawbacks. This section discusses some of the positive and negative aspects of VITAL. These are solely individual opinions based on the experiences of the authors.

9.1 Highlights of VITAL

VITAL ASIC Libraries can be modelled more accurately than Verilog. For example, LSI Logic VITAL libraries have features like negative timing constraints and input edge sensitivity in path delays. Verilog currently ignores edges on path delays. Negative timing constraints is a feature which has not been supported until recently in Verilog. It is still highly a simulator dependent and is not a part of the OVI Verilog standard. VITAL support for negative timing constraints seems to be a more elegant and unambiguous solution.

ASIC libraries typically include complex scan cells with multiple clocks and multiple storage outputs. Verilog provides UDP description which are restricted to a single output. This poses a great difficulty to modelers as they have to connect UDPs in a parallel or sequential manner and deal with synchronization problems. VITAL state table have the ability to specify multiple output states. In addition, several internal storage nodes can also be represented. This is a significant advantage while modeling complex cells.

VITAL back-annotation is based on generics. This parametrized scheme of back-annotation is much more flexible. The VITAL standard only defines on how the generics should be specified, but it does not impose any restrictions on the usage of generics. This is often a problem in Verilog, where SDF back-annotation is highly dependent on the delay model. Consider the case of a RAM model where the functionality is highly dependent on timing which precludes the usage of a pin-to-pin delay model. Due to lack of a pin-to-pin model for memories and the functionality being purely behavioral, it precludes the usage of SDF for back-annotation. Instead inelegant modeling techniques have to be adopted to insert the timing values in the model.

VITAL is a standard and hence the models are portable. This means, that the models should exhibit the
same functional and timing characteristics regardless of the simulator on which they are exercised. Verilog model behavior is sometimes simulator dependent and at times different simulators may produce different results. These issues may be tool specific but at least the modelers need not be necessarily concerned about VITAL models while running various VITAL compliant simulators. This is not necessarily true in the verilog world as of today.

The VITAL standard offers flexibility of having two levels of compliance viz., Level-0 and Level-1. This enables the users to choose between performance and the desired functionality. In almost all cases the users could at least make the model Level-0 compliant and use the SDF back-annotation scheme.

Glitch handling is an aspect of model behavior which is traditionally dependent on the HDL being used and the corresponding simulator. VITAL provides different modes of glitch handling. These are glitch OnDetect and glitch OnEvent. Due to the flexibility provided by VITAL we could develop models which correlate more closely with LSI’s internal simulator.

VITAL supports various usability features which makes the model easy to maintain. For example features like turning on/off timing checks, message reporting and X-generation are provided within the model. In Verilog these features are simulator dependent and are much more restrictive in terms of usage.

9.2 Lowlights of VITAL

VITAL being a subset of VHDL language imposes strong type checking and semantic restrictions which makes it not so user friendly. Verilog on the other hand is much more easy to learn and use. This lowlight is related more to the inherent differences between Verilog and VHDL.

The VITAL standard should address some of the existing modeling issues as pointed out in this paper to make it much more usable. Features such as skew checks, zero delay assignments and more accurate state table handling should be added.

VITAL models are much more verbose than the corresponding verilog models. It is easier to write verilog models than to write a VITAL model for the same cell. Considering the complexity of typical ASIC cells, a VITAL model contains too much data. Also this may be a factor to consider for development schedules. The main concern for ASIC vendors like LSI is to accelerate the availability of VITAL ASIC libraries. It is impractical for them to consider a development approach of developing all their VITAL libraries from scratch manually. This mandates the use of model generation tools based on their existing technology library databases. Any such approach adopted may have its own set of issues. It is a trade-off between level of accuracy/optimization desired and time constraints. Verilog libraries on the other hand have a significantly faster turnaround time.

10. Conclusion

In conclusion, the paper summarizes the usage scenarios of the various features provided by VITAL. It describes the view points in favor of using a development strategy of converting existing Verilog libraries to VITAL libraries. The paper also brings out certain areas which could be addressed by VITAL in future enhancements to the standard. The main focus of the authors during the development of VITAL libraries at LSI was functionality, timing accuracy and some efficiency considerations. The testing of all the VITAL models was done using Cadence LeapFrog-XL and Mentor Graphics QuickHDL simulators. No performance benchmarks were done using VITAL simulators. In future enhancements to LSI Logic VITAL libraries we plan to achieve performance improvements and additional level of accuracy.

Acknowledgments

The authors would like to thank Raghuram Belur, Prabhu Krishnamurthy for their invaluable suggestions and constant review of technical content and Rajesh Mehta, Varalakshmi Balaji for their help during the preparation of the paper.

References

