1. Introduction

Due to the efforts of Rick Munden founder and CEO of the Free Model Foundation (FMF) and Russ Vreeland Vice President of Engineering of the FMF, a certain style for writing VHDL models for board level simulation has been developed. This paper outlines the methods used and some of the reasoning behind them. The topics covered are: header, libraries, aesthetics, file organization, entity, single-bit models, architecture, and modeling negative timing constraints. It is important that the reader has a working knowledge of the *VITAL ASIC Modeling Specification* (IEEE P1076.4). Also, another important document is a paper Russ Vreeland presented at the 1995 Cadence International Users Group entitled: *Tricks and techniques for Writing VITAL 3.0 compliant ECL models.* That paper deals with the peculiar challenges encountered in writing ECL models.

2. Header

The header for an FMF model contains the following: the file name, copyright agreement, modification history and the description of the part being modeled. The file name is chosen to match the entity name. The copyright information is based on the GNU General Public License. The modification history is a revisions list. The part description section lists the library to which the part belongs, the technology of the part, the part name or number, and a brief description of the part.

Before choosing a new name for a library or an entity the modeler should be aware that the FMF has already established several libraries for different groups of parts. Some of the libraries are: CLOCK, ECLPS, ECL10, ECL100,FIFO, IF, NLB, NLG, STND, STNDH, STNDS and SPECL. Table 1 lists the naming convention for the entity for each of the libraries and the family of parts that are modeled. Additional

Library	Entity	Parts Family
CLOCK	various	clock drivers and generators
ECLPS	eclpsxxx	Motorola MC10/100EXXX and Synergy SY10/100EXXX ECL
ECL100	ecl100xxx	100K & 100KH series ECL
ECL10	ecl10xxx	10K & 10KH series ECL
FIFO	fifoxxx	multi-sourced FIFOs
IF	ifxxx	line drivers & receivers
NLB	nlbxxx	NTT SELIC EC
NLG	nlxxx	NLG series
STND	stdxxx	54/74XXX series TTL or CMOS
STNDH	stdhxxx	54/74XXX series with bus hold
STNDS	stdsxxx	74XXX bus switches
SPECL	vxxxx	SONY SPECL ECL

TABLE 1.

libraries will be defined as needed.

The copyright note and modification history are self explanatory in style. Version numbers in the modification history should start with “1.0” and count up.

The part description section has four fields: library, technology, part, and description. The contents of the first three fields are written in all capital letters. The Description field is written out like a title. The library field contains the name of the library. If no library already exists then the name “MISC.” is used. If the model writer has a new family of parts to be added to the FMF data base then a new library should be cre-

ated. The technology field is used to indicate the technology or the family of the parts. In the case of the STD library the technology field contains “54/74XXXX” since this family of parts can be functionally equivalent but be either TTL or CMOS technologies. The part field contains the name of the part (entity) and may also have a specific part number or numbers to help clarify its identity. The description field is usually taken directly from the first line of the data sheet but may contain more information if needed for clarity. Some examples are shown below.

```
--------------------------------------------------------------------------------
--  PART DESCRIPTION:
--
--  Library:     STD
--  Technology:  54/74XXXX
--  Part:        STD869
--
--  Description: Synchronous 8-Bit Up/Down Counter
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--   PART DESCRIPTION:
--
--   Library:     SPECL
--   Technology:  ECL
--   Part:        V1136 (CXB1136)
--
--   Description: 8-Bit Universal Up/Down Counter
--
--------------------------------------------------------------------------------
```

3. File Organization

Since the entity is part of the VHDL code and VHDL does not allow entity names to begin with a number, FMF file names do not begin with a number. The file name uses the standard .vhd extension to the entity name i.e. <entity name>.vhd. Each file has exactly one entity and one architecture per model. Since there is only one architecture no configurations are used.

4. Libraries

The two libraries needed for FMF models are the IEEE library and the FMF library. The IEEE library contains the common nine value logic package and the two VITAL packages: VITAL_timing and VITAL_primitives. The other useful package for models that are not Level 1 is the std_logic_arith package which has arithmetic operators that are overloaded for use with the std_logic types. The FMF library contains the packages that have predefined constants and types. The constants define the default values for generics as well as conversion and state tables. The ff_package package contains state tables for most of the common flip-flops that can be modeled. The state_tab_package package contains state tables for counters up to three bits. These are useful for clock dividing portions of a model.

Two special packages within the FMF library are the ecl_utils and the gen_utils. Both packages contain the default definitions for constants. The ecl_utils package also contains special tables and maps for the unique needs of ECL models (see *Tricks and techniques...).* The gen_utils package should be used for most other models. If another family of parts exists that also needs a set of special constants and tables then a new package for that family should be defined.

5. Aesthetics and Maintainability

It may not always seem important, but since FMF models are to be distributed as source code, the end user will appreciate being able to read and follow the code easily. So, keeping the code neat, orderly, and of a consistent style will aid those who need to make modifications and/or fix possible bugs. Also, there are times when a global change needs to be made throughout the library. When this happens it is impor-

tant that all the source code has the exact same tabular alignment in order to facilitate automated updates to the whole library. As an example, suppose that a new generic is made available in a new version of the VITAL spec. It could be added to all the models in a library automatically by a computer program that reads and writes to the source code files. Appendix A shows the tabular alignment that should be used.

In general the look of an FMF model follows the look of the code written for IEEE packages. Reserved words should be in all capitals letters. Sections of code should be separated by comments delimited by lines above and below the comment. When doing this type of comment, it looks best if the line is continuous and extents to the right margin as in the example below.

Do this:

```
-------------------------------------------------------------------------
-- Timing Check Section
-------------------------------------------------------------------------
```

As opposed to this:

```
-- ------------------------------------
-- Timing Check Section
-- ------------------------------------
```

The file width should be limited to 80 characters. Indentation should be four spaces deep. If tabs are used they should be replaced by spaces before delivering the model. This will make it easier for those who view the model or print it out for the first time from having to worry about changing the default tab width (typically 8 spaces).

6. Entity

Besides declaring the generics and ports, the entity of a VITAL compliant model has a special attribute to indicate its compliance. The following line is placed just before the end statement of the entity.

```
ATTRIBUTE VITAL_level0 of <entity> : ENTITY IS TRUE;
```

6.1 Generics

The two types of VITAL generics are for timing and for control parameters. FMF models also add a special control generic that defines the default timing model. In general, for consistency, the timing generics are listed first and the control generics are listed last. In particular, the tipd and tpd generics are at the top and the “TimingModel” generic is at the bottom. Each group of generics is separated by a one line comment. The following example is typical.

```
GENERIC (
    -- tipd delays: interconnect path delays
    tipd_CLK            : VitalDelayType01 := VitalZeroDelay01;
    tipd_CLKNeg         : VitalDelayType01 := VitalZeroDelay01;
    tipd_D              : VitalDelayType01 := VitalZeroDelay01;
    tipd_DNeg           : VitalDelayType01 := VitalZeroDelay01;
    -- tpd delays: propagation delays
    tpd_CLK_Q                     : VitalDelayType01 := ECLUnitDelay01;
    -- tsetup values: setup times
    tsetup_D_CLK                  : VitalDelayType := ECLUnitDelay;
    -- thold values: hold times
    thold_D_CLK                   : VitalDelayType := ECLUnitDelay;
    -- tperiod_min: minimum clock period = 1/max freq
    tperiod_CLK_posedge           : VitalDelayType := ECLUnitDelay;
    -- tpw values: pulse widths
    tpw_CLK_posedge     : VitalDelayType := ECLUnitDelay;
    tpw_CLK_negedge     : VitalDelayType := ECLUnitDelay;
    -- generic control parameters
    InstancePath        : STRING  := DefaultECLInstancePath;
    TimingChecksOn      : BOOLEAN := DefaultECLTimingChecks;
    MsgOn               : BOOLEAN := DefaultECLMsgOn;
    XOn                  : BOOLEAN := DefaultECLXOn;
    -- For FMF SDF technology file usage
    TimingModel          : STRING  := DefaultECLTimingModel
```

;

Certain rules must be followed for the way timing generics are declared in the model if the data from an SDF file is to map successfully to the VHDL generics. The rules for this mapping can be found in Clause 5.2.7 of the VITAL specification and will be reviewed briefly here. Each generic starts with one of the VITAL predefined prefixes followed by the names of the ports to which it applies. For some generics the port name is followed by an edge condition. Sometimes special conditions can be placed on a generic which map to the SDF keyword COND. Below is a list of the generics and their formats.

tipd_<input port>
tpd_<input port>_<output port>[_<condition>]
tsetup_<tested port>_<reference port>[_<condition>]
thold_<tested port>_<reference port>[_<condition>]
trecovery_<tested port>_<reference port>[_<condition>]
tpw_<input port>_<edge condition>
tperiod_<input port>_<edge condition>
ticd_<input port>
tisd_<tested port>_<reference port>

It is important to note that the port names used in the generic match an actual port name as it is declared. Often the case arises where one generic applies to multiple numbered ports. In that case, the generic should contain the port name with the lowest alphanumeric weight. To illustrate this point, suppose the following ports were declared:

```
DA     : IN    std_logic := '0';
DB     : IN    std_logic := '0';
DC     : IN    std_logic := '0';
```

If the tpd generic was common to all of them it would be declared like this:

```
tpd_DA_Q        : VitalDelayType01 := ECLUnitDelay01;
```

but not like this:

```
tpd_D_Q         : VitalDelayType01 := ECLUnitDelay01;
```

since “D” is not a port name.

An edge condition is added to the tpw and tperiod generics in order to distinguish high going pulses from low going pulses. For high going pulses the extension “_posedge” is used. For low going pulses the extension “_negedge” is used. As an example table 2 charts the mapping of the tpw and tperiod timing generics for a port called CLK. Figure 1 shows how the VHDL constructs refer to a waveform. Please note that SDF entries are case sensitive and VHDL is not. In order to avoid confusion, however, the cases should match between the SDF file and the VHDL file.

SDF Entry	VHDL Construct
(WIDTH (posedge CLK)...	tpw_CLK_posedge
(WIDTH (negedge CLK)...	tpw_CLK_negedge
(PERIOD (posedge CLK)...	tperiod_CLK_posedge

TABLE 2.

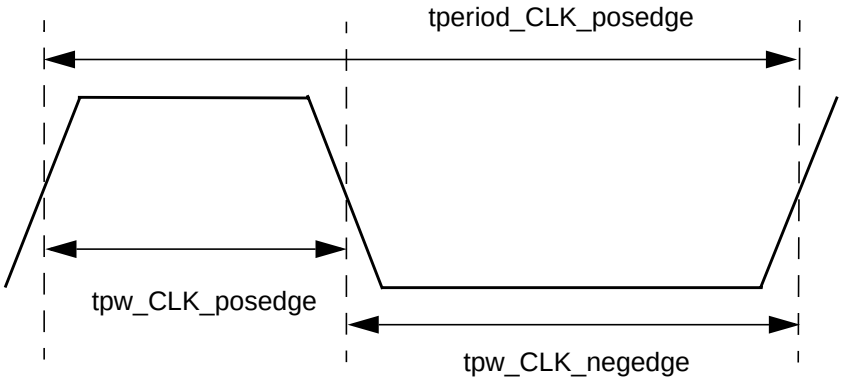


Figure 1

When the situation arises where the timing generic for a single port can take several values depending upon the inputs to other ports, a lexical suffix that describes the dependency is appended to the timing generic. This suffix maps to the SDF conditional construct that uses the key word COND. For every SDF conditional expression there is a VHDL lexical representation to which the expression is mapped. Table 3 is an abbreviated comparison of the most likely expressions that might be encountered. The complete list can be found in Clause 5.2.7.3.2 of the VITAL specification. The suffix is constructed by replacing the SDF expressions with the equivalent VHDL lexical representations while separating them with an underscore.

SDF	VHDL	Description
0	0	Logical Low
1	1	Logical High
==	EQ	Equal to
!=	NE	Not Equal to
&&	AN	Logical AND
||	OR	Logical OR

TABLE 3.

As an example, suppose a port named D had a setup specification with respect to a port called CLK that varied depending upon the input values at the ports S0 and S1. The SDF file might have the following expression:

```
(SETUP (COND S0 == 0 && S1 == 0 D) CLK ( ...
```

The VHDL representation to which the above maps would be:

```
tsetup_D_CLK_S0_EQ_0_AN_S1_EQ_0
```

As another example, suppose the propagation delay through a device from port A to port Y depended upon the values at two ports called S0 and S1. The SDF expression:

```
(COND S0 || !S1 == 0 (IOPATH A Y ( ...
```

The VHDL representation:

```
tpd_A_Y_S0_OR_NT_S1_EQ_0
```

All generics are given a default value in the models utility package (ecl_utils or gen_utils). The control generic “TimingChecksOn” is only declared in an entity if the architecture has a timing check section. All other VITAL control generics are always declared in the entity. The TimingModel generic is used to specify which timing model applies to which instantiation of the model in the final net list.

6.2 Port Declarations

Port names are chosen based on the data sheet for the part. For low active signals, the port name is appended with “Neg”. Table 4 lists the preferred names for ports that should be used if the names from a data sheet might be misleading or too lengthy. Shorter names are better because they reduce the length of the generic names.

Port Name	Use
CLK; CLK1, CLK2...	Clock
D; D0, D1...	Synchronous Inputs
Q; Q0, Q1...	Memory Element Outputs
A, B, C...; A1, A2...	Combinational Logic Inputs
Y; Y1, Y2..	Combinational Logic Outputs
RST	Reset
SEL; S0, S1...	Select
EN	Enable

TABLE 4.

Ports are initialized to a value appropriate for the technology of the device being modeled. Since ECL devices usually have pull down resistors at the inputs, and they can be left open in a circuit, the input ports of the model are initilized to ‘0’. For TTL, CMOS or NMOS models the input ports are initilized to ‘X’ since the real device should not have inputs that are left open in a circuit. During a simulation the ‘X’ will propagate if the design has this error. All output ports should be initialized to ‘U’.

For maintainability and clarity, all ports for a model should be declared as scalars. There should be no vectored ports. Because the VITAL signal delay, timing check and delay procedures can only have single bit data passed to them, it is much clearer and easier to maintain when the ports are declared as scalars. Otherwise the procedures must be called with vector elements which makes them harder to read and doesn’t save any work or space.

7. Single-Bit Models

Models will usually be “single-bit.” That is, if an IC has repetitive sections in a single package, only one section is to be modeled. For board level simulation, it is left to the CAD tool to take the individually placed symbols from schematic capture and instantiate the component as many times as necessary in the final VHDL netlist. As an example, the 74XXX377 is an 8-bit register with a common clock and enable. The model (std377) is written for only one of the eight sections as is evident be the following port declaration:

```
PORT (
    Q               : OUT   std_logic := 'U';
    D               : IN    std_logic := 'X';
    CLK             : IN    std_logic := 'X';
    GNeg            : IN    std_logic := 'X'
);
```

This approach has the advantage of making models easier to read and maintain. If the above model were written for eight bits there would need to be seven more redundant timing check and propagation delay procedures.

It may not be practical to model complex parts as single bits. FIFOs and complex memories are examples of parts that should be modeled as whole parts rather than slices.

8. Architecture

All architecture declarations should begin with the following line:

```
ARCHITECTURE vhdl_behavioral of <entity> IS
```

The next line indicates whether the architecture is Level 1 compliant and takes the following form:

```
ATTRIBUTE VITAL_LEVEL[1 or 0] of vhdl_behavioral : ARCHITECTURE IS true;
```

where VITAL_LEVEL is specified to be 1 or 0 and ARCHITECTURE is always “true.” The VITAL specification defines VITAL Level 1 compliance as follows:

> A VITAL Level 1 Architecture shall adhere to the Level 0 specification, except for the declaration of the VITAL _Level0 attribute.
>
> The entity associated with a Level 1 architecture shall be a VITAL Level 0 entity. Together these design units comprise a *Level 1 design entity.*
>
> The only signals that shall be referenced in a Level 1 design entity are entity ports and internal signals. References to global signals and signal-valued attributes are not allowed. Each signal declared in a Level 1 design entity shall have at most one driver.
>
> The use of subprogram calls and operators in a Level 1 architecture is limited. The only operators or subprograms that shall be invoked are those declared in package **Standard,** package **Std_Logic_1164**, or the VITAL standard packages. Formal subelement associations and type conversions are prohibited in the associations of a subprogram call.

What this means is that a Level 1 model can not use free form VHDL to describe its behavior. The behavior must be described using VITAL logic and state table primitives. Fortunately, the VITAL primitives package contains a wide variety of functions and procedures that can be used to describe combinational logic and the FMF ff_package contains state tables for most of the common types of flip-flops. In general, if a part’s data sheet has a clear and easily understood logic diagram that represents the part correctly (**warning: sometimes the logic and/or timing diagram in a data sheet can be misleading or even dead wrong**), the model should be written for Level 1 compliance. Besides primitives, VITAL has a package that contains all the timing check and delay procedures that a model writer would need. Therefore, a model should only depart from Level 1 compliance in its behavioral description. All timing checks and delays should be done using VITAL procedures when possible even if the model is not Level 1 compliant. Furthermore, all models should have a Wire Delay Block so that backannotation of wire delays can be done for board level simulations.

Some parts, such as those incorporating internal phase locked loops, may have timing based on an internal clock rather than an event on a pin. In such cases, timing checks and delays may be based on a mix of calculations performed inside the model and backannotated values.

8.1 Signal Declarations

Certain signal suffixes are common to many models. Table 5 lists the naming conventions that should be used. Wire Delay signal is the result of a VITAL Wire Delay procedure in the Wire Delay Block. Signal Delay signal is the result of a VITAL Signal Delay procedure in the Signal Delay Block. Internal signals are those signals that connect processes and primitives which are not ports. The name chosen for an internal

signal should be such that the closest port to the internal signal is used. All signals are declared as “std_ulogic” and are initialized to ‘X’.

Signal Name	Use
<input port>_ipd	Wire delay signal
<input port>_dly	Signal delay signal
<port>int	Internal signal

TABLE 5.

8.2 Wire Delay Block

The Vital Wire Delay Block contains the calls to the VitalWireDelay procedures which use the tipd_<port> generics. Every input port should have an associated VitalWireDelay procedure and wire delay generic. Each procedure call begins with a label. The labels are numbered starting with 1 and counting up. Each label begins with: “w_”. Below is an example wire delay block.

```
-------------------------------------------------------------------------
-- Wire Delays
-------------------------------------------------------------------------
WireDelay : BLOCK
BEGIN

    w_1: VitalWireDelay (CLK_ipd, CLK, tipd_CLK);
    w_2: VitalWireDelay (CLKNeg_ipd, CLKNeg, tipd_CLKNeg);
    w_3: VitalWireDelay (D_ipd, D, tipd_D);
    w_4: VitalWireDelay (DNeg_ipd, DNeg, tipd_DNeg);

END BLOCK;
```

8.3 Concurrent Procedures Section

All concurrent procedure calls should immediately follow the wire delay block (unless there is a Signal Delay Block -- see the Negative Timing Constrains section). Each procedure call begins with a label. The labels are numbered starting with 1 and counting up. Each label begins with: “a_”. Concurrent procedures are used for logic primitives that operate outside of a process. Some examples of uses include: Or’d clock, multiplexed inputs, 3-state buffers and output buffers. Output buffers are useful for modeling “wired-OR” and “wired-AND” capable outputs. An input parameter of the VITAL primitives called ResultMap is used for this purpose. To model the output, one of the strong logic levels that would normally be passed to an output port needs to be mapped to a ‘Z’. In the case of an open collector of a TTL (or drain of a CMOS) output, it is a ‘1’ that needs to be mapped. In the case of an open emitter of an ECL output it is a ‘0’ that needs to be mapped. There are two FMF packages that contain the needed maps. The gen_utils package contains the map (STD_wired_and_rmap) for an open collector (drain). The ecl_utils package contains the map (ECL_wired_or_rmap) for an open emitter. Below is an example of the Concurrent Procedure Calls section of an FMF model. In this case it is an ECL differential output that is being modeled.

```
-------------------------------------------------------------------------
-- Concurrent Procedures
-------------------------------------------------------------------------
a_1: VitalBUF (q => Q, a => Qint, ResultMap => ECL_wired_or_rmap);
a_2: VitalINV (q => QNeg, a => Qint, ResultMap => ECL_wired_or_rmap);
a_3: VitalOR2 (q => LENint, a => LEN1_ipd, b => LEN2_ipd);
```

Since VITAL primitives procedures do not allow for the control of X outputs in the event of a glitch, they should generally not be used with propagation delays. Functions that incorporate backannotated delays are better modeled as processes with VitalPathDelays.

8.4 Process Section

Immediately following the Concurrent Procedures section is the VITAL Process section. Each process should be separated by a comment title. The declaration of each process should begin with a descriptive label. The label for the main behavior process is “VitalBehavior”. For ECL models there are two other commonly used processes. One for differential clocks and one for differential inputs. Each of these has its respective label: “Dinputs” and “ECLclock”. For VITAL Level 1 compliance, a process can only assign a signal through the Path Delay Section. If the process does not need a delay then a “dummy” Path Delay must be used (see the “Multiple processes and signal assignments” section of *Trick and techniques...*). If a model is not Level 1 compliant then no process needs the “dummy” Path Delay. All processes should have a sensitivity list. Each major section of a process should be separated with a comment title. The following code from the eclpsl52 model shows the organization of multiple processes in a Level 1 compliant model.

```
-------------------------------------------------------------------------
-- D inputs Process
-------------------------------------------------------------------------
Dinputs : PROCESS (D_ipd, DNeg_ipd)

    -- Functionality Results Variables
    VARIABLE Dint_zd        : X01;

    -- Output Glitch Detection Variables
    VARIABLE D_GlitchData   : VitalGlitchDataType;

BEGIN

    -------------------------------------------------------------------------
    -- Functionality Section
    -------------------------------------------------------------------------
    Dint_zd := ECL_s_or_d_inputs_tab (D_ipd, DNeg_ipd);

    -------------------------------------------------------------------------
    -- (Dummy) Path Delay Section
    -------------------------------------------------------------------------
    VitalPathDelay (
        OutSignal       => Dint,
        OutSignalName   => "Dint",
        OutTemp         => Dint_zd,
        GlitchData      => D_GlitchData,
        Paths           => ( 0 => (0 ps, VitalZeroDelay, FALSE))
    );

END PROCESS;

-------------------------------------------------------------------------
-- ECL CLock Process
-------------------------------------------------------------------------
ECLclock : PROCESS (CLK_ipd, CLKNeg_ipd)

    -- Functionality Results Variables
    VARIABLE Mode           : X01;
    VARIABLE CLKint_zd      : std_ulogic;
    VARIABLE PrevData       : std_logic_vector(1 to 3);

    -- Output Glitch Detection Variables
    VARIABLE CLK_GlitchData : VitalGlitchDataType;

BEGIN

    -------------------------------------------------------------------------
    -- Functionality Section
    -------------------------------------------------------------------------
    Mode := ECL_diff_mode_tab (CLK_ipd, CLKNeg_ipd);

    VitalStateTable (
        StateTable      => ECL_clk_tab,
        DataIn          => (CLK_ipd, CLKNeg_ipd, Mode),
        Result          => CLKint_zd,
        PreviousDataIn  => PrevData
    );

    -------------------------------------------------------------------------
    -- (Dummy) Path Delay Section
    -------------------------------------------------------------------------
    VitalPathDelay (
        OutSignal       => CLKint,
```

```
        OutSignalName   => "CLKint",
        OutTemp         => CLKint_zd,
        GlitchData      => CLK_GlitchData,
        Paths           => ( 0 => (0 ps, VitalZeroDelay, FALSE))
    );

END PROCESS;

-------------------------------------------------------------------------
-- Main Behavior Process
-------------------------------------------------------------------------
VitalBehavior : PROCESS (CLKint, Dint)

    -- Timing Check Variables
    VARIABLE Tviol_D_CLK    : X01 := '0';
    VARIABLE TD_D_CLK       : VitalTimingDataType;

    VARIABLE Pviol_CLK      : X01 := '0';
    VARIABLE PD_CLK         : VitalPeriodDataType := VitalPeriodDataInit;

    VARIABLE Violation      : X01 := '0';

    -- Functionality Results Variables
    VARIABLE Q_zd           : std_ulogic;
    VARIABLE PrevData       : std_logic_vector(1 to 3);

    -- Output Glitch Detection Variables
    VARIABLE Q_GlitchData   : VitalGlitchDataType;

BEGIN

    -------------------------------------------------------------------------
    -- Timing Check Section
    -------------------------------------------------------------------------
    IF (TimingChecksOn) THEN

        VitalSetupHoldCheck (
            TestSignal      => Dint,
            TestSignalName  => "Dint",
            RefSignal       => CLKint,
            RefSignalName   => "CLKint",
            SetupHigh       => tsetup_D_CLK,
            SetupLow        => tsetup_D_CLK,
            HoldHigh        => thold_D_CLK,
            HoldLow         => thold_D_CLK,
            CheckEnabled    => TRUE,
            RefTransition   => '/',
            HeaderMsg       => InstancePath & "/eclpsl52",
            TimingData      => TD_D_CLK,
            Violation       => Tviol_D_CLK
        );

        VitalPeriodPulseCheck (
            TestSignal      => CLKint,
            TestSignalName  => "CLKint",
            Period          => tperiod_CLK_posedge,
            PulseWidthHigh  => tpw_CLK_posedge,
            PulseWidthLow   => tpw_CLK_negedge,
            CheckEnabled    => TRUE,
            HeaderMsg       => InstancePath & "/eclpsl52",
            PeriodData      => PD_CLK,
            Violation       => Pviol_CLK
        );

    END IF;

    -------------------------------------------------------------------------
    -- Functionality Section
    -------------------------------------------------------------------------
    Violation := Tviol_D_CLK OR Pviol_CLK;

    VitalStateTable (
        StateTable      => DFF_tab,
        DataIn          => (Violation, CLKint, Dint),
        Result          => Q_zd,
        PreviousDataIn  => PrevData
    );

    -------------------------------------------------------------------------
    -- Path Delay Section
    -------------------------------------------------------------------------
    VitalPathDelay01 (
```

```
            OutSignal       => Qint,
            OutSignalName   => "Qint",
            OutTemp         => Q_zd,
            GlitchData      => Q_GlitchData,
            Paths           => (
                0 => (InputChangeTime   => CLKint'LAST_EVENT,
                      PathDelay         => tpd_CLK_Q,
                      PathCondition     => TRUE)
            )
        );

    END PROCESS;
```

8.4.1 Variable Declarations

Certain variable suffixes are common to many models. Table 6 lists the naming conventions that should be used. There are four basic groups of variables that are declared: Timing Check, Funtionality Results, Output Glitch Detection, and No Weak Value. Each major group of variables is separated by a short comment line.

For every timing check procedure, there is its associated violation variable used by the main behavior process and a Timing Data information variable used by the timing check procedure. Furthermore, there is at least one more common violation variable used by the main behavior process. This variable is called “Violation” and is used as the result of an “OR” of all the individual violations variables. All violation variables are initialized to ‘0’.

Functionality Results Variables are for both intermediate and final results of a process. Two of the more common variables are “PrevData” and “<output port>_zd”. The former is used to hold the previous data in a state table. The latter is used as the input to the Path Delay procedure. Occasionally, for a non-Level 1 compliant model, it may be desirable to declare a vectored variable to do calculations. In that case the best way to deal with the Path Delay Procedure is to alias each element of the vector as if it were an independent result variable. As an example, suppose a four bit vector was used to calculate a count and the result of the count was to be output to the output ports: Q0 through Q4. The variable declaration should look as follows:

```
-- Functionality Results Variables
VARIABLE COUNT          : std_logic_vector(4 downto 0);
ALIAS Q0_zd             : std_ulogic IS COUNT(0);
ALIAS Q1_zd             : std_ulogic IS COUNT(1);
ALIAS Q2_zd             : std_ulogic IS COUNT(2);
ALIAS Q3_zd             : std_ulogic IS COUNT(3);
```

Another situation encountered with non-Level 1 compliant models is the need to deal with weak value inputs (‘L’ and ‘H’). While VITAL procedures automatically handle them, free form VHDL code has to recognize them as valid inputs. When code tests for a ‘1’ it also has to test for an ‘H’ and for a ‘0’ it has to test for an ‘L’. The easiest way to deal with this is to declare a “No Weak Value” variable and strip weak values with the “To_UX01” function found in the IEEE 1164 package. This variable can then be used in place of an input signal. As an example, suppose a model had two control ports, S0 and S1, that needed to be tested. The variable declarations would be:

```
VARIABLE S0_nwv         : UX01 := ‘X’;
VARIABLE S1_nwv         : UX01 := ‘X’;
```

At the beginning of the process there would be:

```
S0_nwv := To_UX01 (S0_ipd);
S1_nwv := To_UX01 (S1_ipd);
```

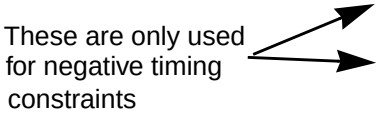
Then, wherever S0_ipd and S1_ipd would normally be used, S0_nwv and S1_nwv would take their places.

Variable Name	Use
Tviol_<tested port>_<reference port	Setup and hold check violation
TD_<tested port>_<reference port>	Setup and hold data
Pviol_<tested port>	Period pulse violation
PD_<tested port>	Period pulse data
Rviol_<tested port>_<reference port>	Recovery removal violation
RD_<tested port>_<reference port>	Recovery removal dat
Violation	“OR” of all violations variables
<output port>_zd	Result used by Path Delay procedure
<output port>_GlitchData	Used by Path Delay procedure
<input port>_nwv	Used to strip inputs of weak values

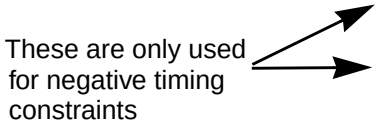
TABLE 6.

8.4.2 Timing Check Section

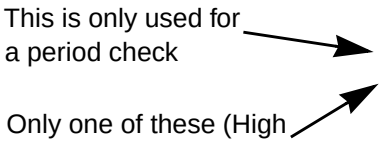
The timing check section contains all the VITAL timing check procedures inside an “if” statement. These procedures are only accessed if the generic “TimingChecksOn” is set to “TRUE.” To enhance readability, each timing check procedure call is organized as shown below.




```
VitalSetupHoldCheck (
    TestSignal      =>  <tested port>,
    TestSignalName  =>  "<tested port>",
    TestDelay       =>  tisd_<tested port>_<reference port>,
    RefSignal       =>  <reference port>,
    RefSignalName   =>  "<reference port>",
    RefDelay        =>  ticd_<reference port>,
    SetupHigh       =>  tsetup_<tested port>_<reference port>,
    SetupLow        =>  tsetup_<tested port>_<reference port>,
    HoldHigh        =>  thold_<tested port>_<reference port>,
    HoldLow         =>  thold_<tested port>_<reference port>,
    CheckEnabled    =>  TRUE,
    RefTransition   =>  '<VITAL edge symbol>',
    HeaderMsg       =>  InstancePath & "<entity name>",
    TimingData      =>  TD_<tested port>_<reference port>,
    XOn             =>  XOn,
    MsgOn           =>  MsgOn,
    Violation       =>  Tviol_<tested port>_<reference port>
);
```




```
VitalRecoveryRemovalCheck (
    TestSignal      => <tested port>,
    TestSignalName  => "<tested port>",
    TestDelay       =>  tisd_<tested port>_<reference port>,
    RefSignal       =>  <reference port>,
    RefSignalName   => "<reference port>",
    RefDelay        => ticd_<reference port>,
    Recovery        => trecovery_<tested port>_<reference port>,
    ActiveLow       => FALSE,
    CheckEnabled    => TRUE,
    RefTransition   => '<VITAL edge symbol>',
    HeaderMsg       => InstancePath & "<entity name>",
    TimingData      => TD_<tested port>_<reference port>,
    XOn             => XOn,
    MsgOn           => MsgOn,
    Violation       => Rviol_<tested port>_<reference port>
);
```




```
VitalPeriodPulseCheck (
    TestSignal      =>  <tested port>,
    TestSignalName  => "<tested port>",
    Period          => tperiod_<tested port>_posedge,
    PulseWidthHigh  => tpw_<tested port>_posedge,
    PulseWidthLow   => tpw_<tested port>_negedge,
    CheckEnabled    => TRUE,
    HeaderMsg       => InstancePath & "<entity name>",
    PeriodData      => PD_<tested port>,
    XOn             => XOn,
    MsgOn           => MsgOn,
    Violation       => Pviol_<tested port>
);
```

Note that the listing order for the parameters passed to the timing check is different than the order listed in the VITAL spec. Also, the “MsgSeverity” constant is not included and not all parameters are passed all the time as indicated by the margin notes. “<VITAL edge symbol>“ refers to the characters that are included in the “VitalEdgeSymbolType” usually a “/” for a rising edge or a “\” for a falling edge.

8.4.3 Functionality Section

The functionality section is the core functional block of the process. In the case of a Level 1 compliant model this section contains the variable assignments and calls to “VitalStateTable” procedures. For models that are not Level 1 compliant this section contains the free form VHDL code that describes the function of the process. In either case the section begins with the “Violation” variable being assigned the result of the “OR” of the timing check variables.

8.4.4 VITAL State Tables

Almost all FMF-supplied package state tables are written with one state, and are designed to be used with a call to “VitalStateTable” (VST) with a "Result" variable of type std_ulogic and the "NumStates" parameter omitted (this causes the 2nd of the 4 overloaded VST procedures to be called -- the 2nd has a scalar "Result" parameter declared). For usages of VST where either 0 or more than 1 states are defined, or the "Result" output is a vector (of size greater than 1), the call to VST must select the first of the overloaded VST procedures. The 2 concurrent VST procedures (numbers 3 and 4 in the order of their listing in the source code) have never been used in an FMF model. One problem in using them arises from the "DataIn" parameter being declared as a std_logic_vector. In the cases where FMF was considering using a concurrent VST, the inputs to the VST were sets of scalar signals. An aggregate of several signals is not a signal, and the concurrent VST requires a bona fide vector signal to be passed into it. Thus, the usual technique of using aliases to convert a scalar to a vector and vice versa, for the purposes of matching the type declarations in the VST procedure, cannot be used for an outside-the-process (concurrent) procedure. Because of this difficulty, cases where concurrent VSTs can be used are rare, and generally shouldn't be considered.

8.4.5 Path Delay Section

The Path Delay section of a process models the propagation delays through the model. To enhance readability, each Path Delay procedure call is organized as shown below.

```
VitalPathDelay01 (
    OutSignal       => <signal leaving process>,
    OutSignalName   => "<signal leaving process>",
    OutTemp         => <result variable from functional section> ,
    GlitchData      => <glitch data variable>,
    Paths           => (
        0 => (InputChangeTime   => <signal into process>'LAST_EVENT,
              PathDelay         => <delay generic>,
              PathCondition     => <boolean expression>)
    )
);
```

For each path delay another numbered path is added to the “Paths” array. The numbering starts with zero and counts up. Care must be taken when using the “PathCondition.” Unless a need arises to control different paths under different conditions, the boolean expression should always be set to “TRUE.” When the boolean expression describes a condition, and the condition is not met, the default path delay (usually zero unless explicitly passed into the procedure) is used. So, if there are multiple paths with different conditions, all of the possible conditions must be covered. Otherwise an unintentional zero delay may occur.

9. Negative Timing Constraints

(TBD)