

PRACTICAL VITAL 2.2B MODEL WRITING

Anne Margaret Crow
VEDA Design Automation,
Fareham, England. PO16 8XT

Abstract

The ultimate aim of the VITAL standard is to encourage the adoption of VHDL by accelerating the creation of ASIC libraries suitable for sign-off simulation. This paper looks at the practical aspects of using VITAL for sign-off simulation, writing the models themselves and developing a supportive EDA design environment for maximum productivity.

1 Introduction

The objective of the VITAL initiative is to provide a standard to describe sign-off quality ASIC simulation libraries using the VHDL language.

However the existence of a standard is not enough. EDA vendors must provide tools which harness the power of that standard for the benefit of designers. In the case of the VITAL standard, this means that vendors must supply tools which automate the creation of VITAL models and simulators with enough power to perform complete sign-off simulations of large deep sub-micron ASICs.

This paper begins by looking in detail at how to write VITAL models, illustrating each concept through reference to VITAL models which are part of a commercial VITAL compliant library. It then discusses the availability of EDA software for creating a successful design environment which supports VITAL compliant simulation, focusing on software for the automatic creation of VITAL compliant models and software for simulating VITAL models.

2 Writing VITAL Models

We will begin by looking in detail at how to write VITAL compliant models. The key objective behind the VITAL initiative is the need to provide sufficiently accurate delay information to model complex deep sub-micron processes. Therefore, we will start by listing the timing details which the VITAL standard must be able to model, and then examine how this timing information is represented within the VITAL standard. At this point the differences between the Level 0 and the Level 1 modeling styles will be made clear. We will then look at the two different styles for modeling delays : the distributed delay style and the pin-to-pin timing style and proceed to discuss how these techniques may be used to create models which give sign-off accuracy and accelerated simulation times. The 2.2b version of the VITAL standard is used throughout.

2.1 Timing Information Needed

Let's look at the timing information which must be handled by the VITAL standard.

Intrinsic delay (delay in the transistors between A and D when unloaded)

Marginal capacitance (how changes in the capacitance at D, which is a function of the net DE and the input capacitance E, change the intrinsic delay A to D)

Cross-marginal capacitance (how changes in the capacitance at C, change the intrinsic delay A to D)

Input slew (how changes in the slope of input signal A change the intrinsic delay)

State-dependent delays (how the intrinsic delay changes depending on what the current state of the device is when an input change occurs)

Timing violation checks are also needed. These include set-up and hold checks and minimum pulse width checks.

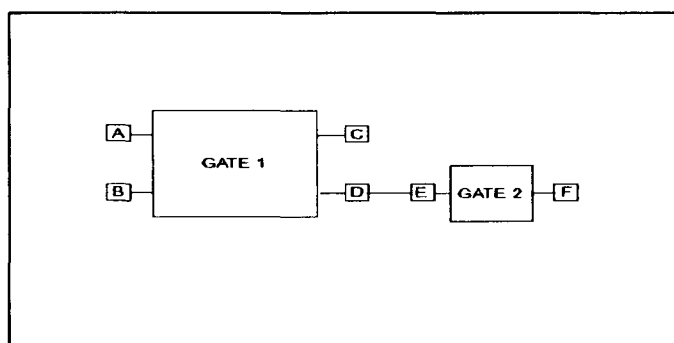


Diagram 1 : Definition of timing delays

2.2 Back-annotation support

A VITAL model only contains intrinsic delays. Everything else, marginal, cross-marginal and input slew effects or temperature, voltage and process effects must be calculated outside the model using a standard delay calculator which outputs the timing information as SDF files. A VITAL compliant model (Level 0 or Level 1 compliant) uses generics to pass this SDF timing information into the model architecture. For this mapping to work, the generics must conform to the SDF-map specification, and the delay types, signal types and port types used in the entity description are limited to those contained in the VITAL timing package (VITAL_TIMING). Before simulation starts, timing information from the SDF file is used to overwrite the default timing information in the VITAL model.

For example, the interconnect delay between D and E is modeled by including a generic in the entity description of the second gate :

```
tpd_E : DelayType01 := (0 ps, 0 ps);
```

This VHDL construct is equivalent to the sdf construct

```
INTERCONNECT TOP/GATE1/D TOP/GATE2/E (10::) (210::)
```

A more complex example is the case of a D-type flip-flop, which has conditional timing paths. The propagation delay *tpd* from clock input CLK to data output Q is dependent on the value of the signal at data input D. *tpd* is represented by the generic :

```
tpd_CLK_Q_D_EQ_1_POSEDGE : DelayType01 := (300 ps, 350 ps);
```

which is equivalent to the sdf construct

```
COND D=1 IOPATH (POSEDGE CLK) Q (320::) (370::)
```

Timing checks are also mapped. For example, a minimum setup time on data pin D is represented by the generic :

```
tsetup_D_Clk : DelayType01 := (1670 ps, 1790 ps);
```

which is equivalent to the sdf construct

```
SETUP CLK D (1700::) (1820::)
```

The use of external SDF files to model interconnect effects means that the accuracy of your VITAL solution depends not only on the VITAL models themselves, but on the accuracy provided by the delay calculator.

2.3 Some VHDL is more VITAL than others

To be Level 0 compliant, models need only to support back annotation from SDF. All other timing and function packages are outside the specification and may be written by the user. A Level 0 simulator *may* give VITAL accuracy, but this depends on the quality of these custom packages.

Level 0 does not define the modeling style and therefore cannot assure both portability and performance across simulators.

As Level 0 models do not use the VITAL packages, all the VHDL procedures must be written by the model writer, which will slow down model development dramatically.

Most significant of all for users of the models, Level 0 models cannot be accelerated. This is particularly important for modeling deep sub-micron processes. For example, a simple flip-flop modeled according to the VITAL standard contains almost 200 lines of code, all of which must be executed at run-time.

To be a practical solution for sign-off simulation, a simulation model must be Level 1 compliant. Level 1 compliant models must use the VITAL timing package (VITAL_TIMING) and the VITAL primitives package. The VITAL timing package contains data types and subprograms for delay selection, timing violation checking and reporting, and glitch detection. The VITAL primitives package contains combinational primitives, a truth table facility to describe more complex combinational parts and state tables to describe sequential parts.

Using the VITAL timing packages ensures that the models are accurate enough to describe the nuances of deep sub-micron behavior, speeds up the development of models and provides portability across simulators. Level 1 compliance mandates that these packages are used in accordance with one of two delay modeling styles, distributed delay or pin-to-pin timing delay. The difference between these two styles will be covered shortly. Most important of all, when developing a practical VITAL design environment, models which meet all the Level 1 compliance criteria can be simulated very much faster by a state-of-the-art simulator such as Vulcan, which gives accelerated simulation of VITAL primitives.

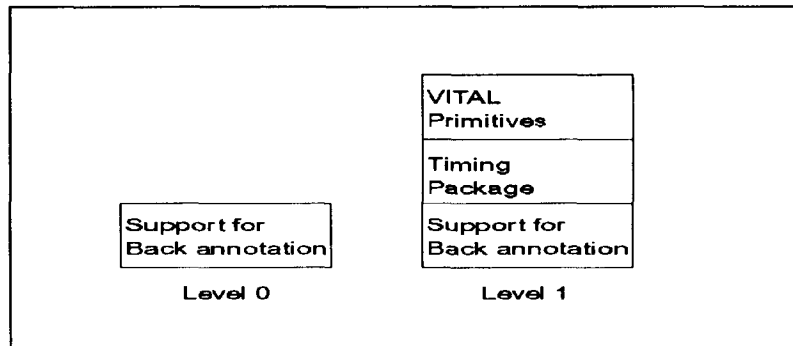


Diagram 2 : Differences between Level 0 and Level 1

2.4 Architecture of models

The VITAL standard does not mandate how model architectures are defined. Combinational parts may be described using VITAL primitives, truth-tables or Boolean equations. VEDA recommends the use of Boolean equations, as these can be most readily accelerated. Sequential parts must be modeled as state-tables which are similar to truth-tables. Example 3 shows a state-table.

```
begin
  G1 : VitalINV   ( q => G1Out, a => I1_ipd, tpd_a_q =>
                 tdevice_g1_q );
  G2 : VitalAND2 ( q => Y0, a => G1Out, b => I2_ipd,
                 tpd_a_q => tdevice_g2_q,
                 tpd_b_q => tdevice_g2_q );
end;
```

Example 1 : Part of IA2 model described using VITAL primitives

```
begin
  y_zd := ( a AND (NOT b) ) OR ( (NOT a) AND b );
end;
```

Example 2 : Part of xor2 model described using Boolean equations

The VITAL standard does not define how a simulator should perform X-handling in state and truth-tables. This means that the behavior of a part with an unknown on one of its inputs must be explicitly stated in the state-table for that part. This task is left to the modeler's discretion, as when writing Verilog models. It is particularly problematic when developing models with many inputs, any of which could be unknown. Writing a model with exhaustive coverage of all possible input conditions is not practical. In practice, the model writer decides which of the input states are likely to go into an unknown state, and models only these input conditions. (This is unlike traditional gate-level simulators such as System HILO, which support automatic X-handling, and for which a very compact state-table, or EXACT, can be provided every time.)

In the following example, when $preset = clear = 0$, $q = qb = 0$. When $clear = 1$, $preset = 0$, $qbar = 0$, $q = 1$. $qbar$ is 0 in both cases, so the state of clear does not affect $qbar$. Accordingly, if $clear = X$ and the other inputs are defined, there is no ambiguity as to the state of $qbar$, it is 0. q on the other hand is not defined and is set to X when $clear = X$. This is shown in line 11 of the state-table.

```

begin
CONSTANT StateTab : VitalStateTableType (1 TO 14, 1 TO 8) := (
-- cp      cd      d      sd      violation      qn      q
( '-', '-', '-', '-', 'X', '-', 'X', 'X' ),
( '-', '0', '-', '0', '-', '-', '0', '0' ),
( '-', '1', '-', '0', '-', '-', '0', '1' ),
( '-', '0', '-', '1', '-', '-', '1', '0' ),
( '/', '1', '1', '1', '-', '-', '0', '1' ),
( '/', '1', '0', '1', '-', '-', '1', '0' ),
( '\', 'B', '-', 'B', '-', '-', 'S', 'S' ),
( '-', '/', '-', 'B', '-', '-', 'S', 'S' ),
( '-', 'B', '-', '/', '-', '-', 'S', 'S' ),
( 'S', 'B', '-', 'B', '-', '-', 'S', 'S' ),
( '-', 'X', '-', '0', '-', '-', '0', 'X' ),
( '-', '0', '-', 'X', '-', '-', 'X', '0' ),
( 'S', '1', '-', 'X', '-', '0', 'S', 'S' ),
( 'S', 'X', '-', '1', '-', '1', 'S', 'S' ),
);
end;

```

Example 3 : Part of fd3 model described using a state-table

2.5 Level 1 delay style differences

As mentioned earlier, Level 1 compliance mandates that delays are modeled in accordance with one of two modeling styles : distributed delay or pin-to-pin timing delay.

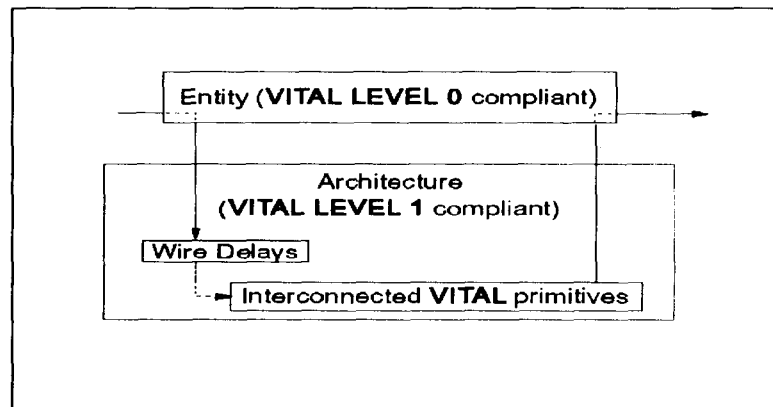


Diagram 3 : distributed delay

The distributed delay style consists of a netlist of VITAL primitives for each cell, with timing delays arbitrarily assigned to each primitive. In the days of proprietary simulators, this was the original modeling method. This style is not recommended. This is because each primitive corresponds to a separate process after elaboration, so a model written this way is usually slower to simulate than a pin-to-pin type model. It is difficult to back-annotate delays onto a model of this type as timing delays are not associated with the ports, but with the internal primitives. Finally, although the style is simple in concept, it is actually quite difficult to write because of the difficulty in judging how to split net delays between the primitives. (Refer to example 4 at the end of the paper, for an example of this modeling style.)

The pin-to-pin delay style enables the use of a single process, labelled Vital Behavior, for each cell. This process contains timing violation checks, functionality and propagation delays from that input to every output for which there is a timing path. Back annotation is straight-forward, as it the intrinsic timing delays are associated with individual ports. Moreover, by employing single processes, this style is more efficient for today's VITAL compliant simulators. (Refer to example 5 at the end of this paper, for an example of this modeling style.)

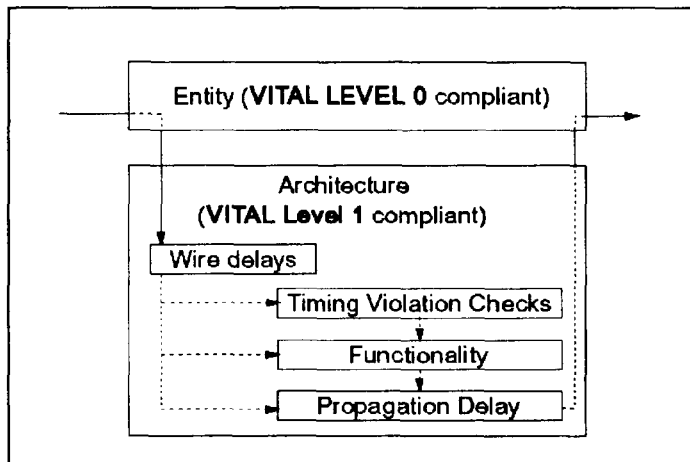


Diagram 4 : pin-to-pin timing

These two styles must not be mixed, as this prevents VITAL simulators from accelerating the models.

3 Creating a VITAL compliant EDA environment

Having established how to write VITAL models, we will turn our attention to the creation of an EDA environment which supports the VITAL standard. The main problem here is the sheer amount of data which must be processed in order to model the complexities of deep sub-micron processes with sufficient accuracy for sign-off simulation. Existing EDA software techniques cannot handle the amount of data needed without seriously degrading performance, often beyond acceptable limits. Until recently, the lack of EDA tools sophisticated enough to handle the VITAL standard looked like being a major barrier to the widespread acceptance of the emerging standard. Fortunately, some EDA vendors have met the challenge of the new standard. The second section of this paper looks at some of the new EDA software for maximizing productivity with the VITAL standard.

3.1 VITAL models and productivity

As you have seen, the VITAL format provides a compact way of sharing all the accurate timing data needed to model deep sub-micron designs. The very fact that the standard can cope with this level of detail poses serious problems for model writers - firstly, obtaining the timing data and secondly, checking that the data in the model is correct. To do this manually typically takes in excess of six man-months for a new ASIC library. This can result in an unacceptable time-lag between developing a new process and being able to create digital designs on that process. Automating model generation is the best solution.

3.2 Automatic VITAL model generation

VEDA Design Automation use Metasoftware's MASTER Toolbox software to automate the library generation process.

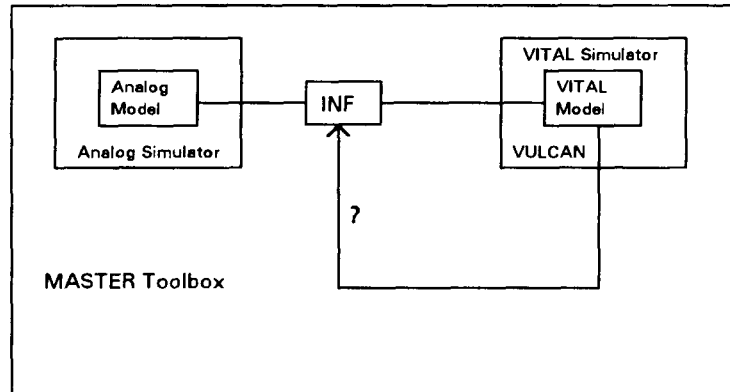


Diagram 5 : Automatic library generation software

A 'golden' analog model is iteratively simulated in its native simulation environment until all possible information is extracted from it : functionality, load-dependencies, capacitances, input thresholds, slope dependencies, state dependencies, process, temperature and voltage dependencies and set-up and hold violations. This information is stored in a compact form in Intermediate Neutral Format (INF). This information is then used to generate a VITAL model. This model is then itself repeatedly simulated using a VITAL compliant simulator (typically VULCAN), so that all possible timing information and functionality is extracted from it. This information is then compared against the results of the analog simulation as a Quality Assurance check on the final digital model. The entire closed loop process is remarkably fast - recently a complete ASIC library of about 90 cells, for a 0.5 micron process was created and validated in under a week.

3.3 Practical Simulation Times

The sheer volume of data required for modeling deep sub-micron poses problems for simulators. For example, a simple flip-flop modeled according to the VITAL standard contains almost 200 lines of code, all of which must be executed at run-time. Arguments are rife as to whether interpreted or compiled solutions are better, but these are not addressing the real issue. All the interpreted versus compiled approach can do is speed up the raw behavioral performance. There are limitations to the improvements which can be achieved in this way, and they are probably almost reached. Any improvements which can be realised in this way will shortly be eroded by requirements or more and more accuracy and thus more and more data to be executed at run-time.

What is needed is a simulator which can take VITAL constructs, VITAL primitives and state-tables and map them directly onto constructs in the simulator kernel, thus providing accelerated VITAL performance. Without accelerated performance, the standard loses its credibility, as it takes an unacceptable length of time to perform a sign-off simulation on a realistically sized (>100,000 gates) design. With accelerated performance, gate-level simulation speeds in excess of those obtainable with Verilog-XL are achievable. This makes VHDL a realistic alternative to Verilog and one welcomed by designers using VHDL for pre synthesis design entry, who can now use one simulator for all stages of their design.

A word of warning. The VITAL standard provides a mechanism for accelerating simulation at gate level, but does not mandate this accelerated performance. As a result, many simulators may claim VITAL-compliance, but do not offer accelerated VITAL simulation. This is because support for accelerated VITAL simulation requires the simulator to have been designed from the beginning with the VITAL standard and methodology in mind, rather than being a traditional style simulator kernel which just happens to be able to simulate heavily processed VITAL models. Only then is it possible to map VITAL primitives and VITAL state-tables directly to constructs in the simulator kernel. VEDA Design Automation's VHDL simulator, Vulcan, is the first to provide accelerated VITAL simulation, capable of accelerating both the VITAL primitives and VITAL state-tables.

4 Conclusion

To conclude, the VITAL standard defines the techniques for creating sign-off quality ASIC models and their techniques are outlined above. It is important to comply with Level 1 of the standard to obtain high performance, accurate and portable models. But to achieve the primary goal of the VITAL initiative - the widespread adoption of VHDL for sign-off simulation - specialist software tools and training are needed to create and use these VITAL models efficiently.

Example 4 : Distributed Delay Style - Simple Combinational Cell

```

LIBRARY          IEEE;                USEIEEE.Std_logic_1164.all;
LIBRARY VITAL;          USE VITAL.VITAL_Timing.all;
                                USE VITAL.VITAL_Primitives.all;

entity IA2 is
  generic (tdevice_g1_q : DelayType01 := ( 1 ns, 1 ns );
          tdevice_g2_q : DelayType01 := ( 1 ns, 1 ns );
          tipd_I1      : DelayType01Z := ( 0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns );
          tipd_I2      : DelayType01Z := ( 0 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns );

          TimingChecksOn : Boolean := true;
          XGenerationOn  : Boolean := true;
          InstancePath   : string := "");

  port      (Y0 : out std_logic;  I1, I2 : in std_logic );
end;

architecture Level1DistributedDelay of IA2 is
  Attribute VITAL_LEVEL1 of Level1DistributedDelay :architecture is TRUE;
begin
  WIRE_DELAY : block
    signal I1_ipd, I2_ipd : std_logic;

    begin
    VitalPropagteWireDelay (Outsig => I1_ipd, Insig => I1, twire => tipd_I1);
    VitalPropagteWireDelay (Outsig => I2_ipd, Insig => I2, twire => tipd_I2);
    end block;

  VITAL_NETLIST : block
    signal G1Out : std_logic;
    begin
    G1 : VitalINV ( q => G1Out, a => I1_ipd, tpd_a_q => tdevice_g1_q );
    G2 : VitalAND2 ( q => Y0, a => G1Out, b => I2_ipd,
                   tpd_a_q => tdevice_g2_q, tpd_b_q => tdevice_g2_q );
    end block;
end;

```

Example 5 : Pin-to-pin delay style - sequential cell

```
library ieee;                use ieee.std_logic_1164.all;
library vital;              use vital.vital_timing.all;
                             use vital.vital_primitives.all;

entity dffs is
  generic
    (tpd_clk_q      : DelayType01 := (350 ps, 350 ps);
      tpd_prez_q    : DelayType01 := (350 ps, 10 ps);
      tpd_clk_qn   : DelayType01 := (350 ps, 350 ps);
      tpd_prez_qn  : DelayType01 := (10 ps, 350 ps);
      tpw_clk      : DelayType01 := (350 ps, 350 ps);
      tpw_prez     : DelayType01 := (350 ps, 10 ps);
      tsetup_d_clk : DelayType01 := (350 ps, 350 ps);
      tsetup_prez_clk : DelayType01 := (350 ps, 10 ps);
      thold_clk_d  : DelayType01 := (350 ps, 350 ps);
      TimingChecksOn : Boolean    := TRUE;
      XGenerationOn  : Boolean    := TRUE;
      InstancePath   : string     := "");

  port
    (q      : out      std_logic;
     qn     : out      std_logic;
     d      : in       std_logic;
     clk    : in       std_logic;
     prez   : in       std_logic );
end dffs;

architecture behavioral of dffs is
  ATTRIBUTE VITAL_LEVEL1 of behavioral : architecture is TRUE;

begin
  VITALBehavior : process (d,clk,prez)
    VARIABLE q_GlitchData : GlitchDataType;
    VARIABLE qn_GlitchData : GlitchDataType;
    VARIABLE Prv : std_logic_vector(1 TO 4) := (OTHERS=>'X');
    VARIABLE Res : std_logic_vector(1 TO 2) := (OTHERS=>'X');
    ALIAS qn_zd : std_logic is Res(1);
    ALIAS q_zd : std_logic is Res(2);
    CONSTANT StateTab : VitalStateTableType (1 TO 7, 1 TO 7) := (
      -- clk d prez violation - qn q
      ( '-', '-', '-', 'X', '-', 'X', 'X'),
      ( '-', '-', '0', '-', '-', '0', '1'),
      ( '/', '0', '1', '-', '-', '1', '0'),
      ( '/', '1', '1', '-', '-', '0', '1'),
      ( '\', '-', 'B', '-', '-', 'S', 'S'),
      ( '-', '-', '/', '-', '-', 'S', 'S'),
      ( 'S', '-', 'B', '-', '-', 'S', 'S')
    );
    VARIABLE violation : X01 := '0';
    VARIABLE pw_clk    : BOOLEAN;
    VARIABLE pwchk_clk : delayarraytypexx(0 TO 1);
    VARIABLE pw_prez   : BOOLEAN;
    VARIABLE pwchk_prez : delayarraytypexx(0 TO 1);
    VARIABLE suh_d_clk : X01 := '0';
    VARIABLE tsuh_d_clk : timemarkertype;
    VARIABLE suh_prez_clk : X01 := '0';
    VARIABLE tsuh_prez_clk : timemarkertype;

  BEGIN

    -- TIMING CHECKS --
    if (TimingChecksOn) then
      VitalPeriodCheck (
        testport      => clk,
        testportname  => "clk",
        pw_hi_min     => tpw_clk(tr01),
        pw_lo_min     => tpw_clk(tr10),
        Info          => pwchk_clk,

```

```

-- MORE TIMING CHECKS --
Violation      => pw_clk,
HeaderMsg      => InstancePath,
Condition      => (prez='1')
);
VitalPeriodCheck (
testport       => prez,
testportname   => "prez",
pw_hi_min      => tpw_prez(tr01),
Info           => pwchk_prez,
Violation      => pw_prez,
HeaderMsg      => InstancePath,
Condition      => TRUE
);
VitalTimingCheck (
testport       => d,
testportname   => "d",
refport        => clk,
refportname    => "clk",
t_setup_hi     => tsetup_d_clk(tr01),
t_setup_lo     => tsetup_d_clk(tr10),
t_hold_hi      => thold_clk_d(tr01),
t_hold_lo      => thold_clk_d(tr10),
CheckEnabled   => (prez='1'),
RefTransition  => (clk='1'),
HeaderMsg      => InstancePath,
TimeMarker     => tsuh_d_clk,
Violation      => suh_d_clk
);
VitalTimingCheck (
testport       => prez,
testportname   => "prez",
refport        => clk,
refportname    => "clk",
t_setup_hi     => tsetup_prez_clk(tr01),
CheckEnabled   => TRUE,
RefTransition  => (clk='1'),
HeaderMsg      => InstancePath,
TimeMarker     => tsuh_prez_clk,
Violation      => suh_prez_clk
);
end if;

violation := suh_d_clk OR suh_prez_clk;

VitalStateTable ( StateTable => StateTab,
                  DataIn => (clk,d,prez,violation),
                  NumStates => 1,
                  Result => Res,
                  PreviousDataIn => Prv
                  );
-----
--Path Delay Section
-----
VitalPropagatePathDelay (q, "q", q_zd,
Paths(0)=>(0 ns,VitalExtendToFillDelay(tpd_clk_q),clk'EVENT),
Paths(1)=>(0 ns,VitalExtendToFillDelay(tpd_prez_q),prez'EVENT),
GlitchData=>q_GlitchData, GlitchMode=>Xonly,GlitchKind=>OnEvent );
VitalPropagatePathDelay (qn, "qn", qn_zd,
Paths(0)=>(0 ns,VitalExtendToFillDelay(tpd_clk_qn),clk'EVENT),
Paths(1)=>(0 ns,VitalExtendToFillDelay(tpd_prez_qn),prez'EVENT),
GlitchData=>qn_GlitchData,GlitchMode=>Xonly,GlitchKind=>OnEvent);
end process;
end behavioral;

```