

ASIC Sign-Off in VITAL

Ravikumar S V

Texas Instruments (India) Pvt. Ltd.

150/1, Infantry Road, Bangalore, INDIA 560 001.

Abstract

This paper highlights the various aspects of ASIC gate level modeling and how VITAL addresses the ASIC sign-off requirements. While designs are becoming larger, the models are also becoming complex with the need to model accurately the various technological effects in the submicron era. While performance and accuracy are contradictory requirements in an ASIC library, the modelers maintain a compromise between the two while developing the library. Some of the complex behavior in ASIC cells and the modeling practise that can be used in VHDL using VITAL specification is discussed. Also some of the requirements for implementation in packages are listed.

1: Introduction

VHDL has been promoted as a language for multi-level design since its inception. VHDL is extensively being used for simulation of behavioral and RTL descriptions. The biggest impediment to VHDL is the lack of back annotation standard and ASIC libraries. The accuracy and acceleration are two prime requirements for any ASIC sign-off simulation library. VITAL standardisation efforts address various aspects of accuracy and acceleration in ASIC designs by way of using an accurate defined timing and primitive package for model library build and a back-annotation standard following OVI SDF. A model is a representation of the behavior of a digital semiconductor device that can be used for logic simulation. Models are represented by behavior and state. An accurate digital simulation model will reflect on the turnaround time for creating ASICs by facilitating first pass silicon. The key to first pass silicon is to have models which mirror 'silicon' behavior. The following sections discuss the various behavior effects

that needs to be modeled and how VITAL addresses them.

2: Model Functionality

A comprehensive list of various silicon behavior in ASIC cells is described in reference[1]. The basic functionality of the cell can be as simple as an AND gate or as complex as a FIFO or complex testability cells like LSSD scan latches or a digital phase locked loop delay element with 24 delay stages. VHDL provides various options to model the functionality. The function can be written as a single process or can be abstracted by instantiating components in a structured form to implement the same function. While a single process behavior may be the best in term of performance various other criteria like ease of modeling, capability to model all timing constraints etc determines what style of modeling is used in VHDL. Example1 shows a combinatorial function realisation using VITAL primitives and tables. A basic D Flip-Flop can be realised either by using a two latch configuration with clock being used as level sensitive inputs on the latches or using as an edge triggered approach. Depending on the basic realisation of Flip-Flop the transition '0'-->'X'-->'1', needs to be defined in VitalStateTableType for correct functionality realisation. A two latch configuration realisation is shown in Example2. Example3 shows how to model strengths using ResultMap type. Currently passing a 'Z' based on a control signal needs to be realised using behavioral code. This type of macro is heavily used for parallel module testing.

2.1: Timing

Simulation models have to comprehend at least two set of timing parameters viz the pin-to-pin path delays and the constraint timing. The model

complexity depends upon on the number of factors these timing numbers depend on. In a generic case we can model the path delay or the constraint as a generic equation with dependency on design, process and operation parameters. The two important design parameters are the output load and the input pin slew while the process parameters can be process variation, the operating temperature and voltages. In case of memories and larger complex cells the timing numbers may also depend on the mux factor, the data pin width etc. So in the most generic case the timing can be modeled as a general N term equation. The larger N is the, difficult it is to model the timing numbers within the model. The problem gets aggravated with simulators not providing easy access to some of the parameters which are required to compute the timings. In VHDL there is no method provided by the language to access from within the model the load values at the output pins. Efforts to make this information available will generally lead to the use of attributes and separate communication protocol between the models. Complex data records can be passed between the models and procedures can be written to get the same but it will be only academic value since the performance overhead is quite large. The generic approach now is to compute the actual timing using another tool and annotate the numbers back to the model using a annotation file. The actual values are read by the model as *generics*. VITAL[2] specification essentially uses this approach for passing timing values. The SDF (Standard Delay File Format) of OVI2.0 is taken as the baseline standard for the delay file format and the naming conventions and other standards have been established to achieve this. As can be seen in [2] special VITAL delay types are defined to ease the SDF to VHDL generic mapping. VitalPropagatePathDelay (VPPD) Procedures are used for pin-to-pin path delay modeling.

Silicon mimicking demands for using four pulse handling options viz. Pulse Pass, Pulse Suppress, Glitch On Event and Glitch On Detect. The default OnEvent Glitch kind in VPPD correctly takes care of Silicon Portrayal. VITAL Timing Check Procedures implicitly use OnDetect GlitchKind type. This type correctly takes care of constraint violations. PATHPULSE SDF construct handling is a modeling requirement for differential and digital phase locked loop cells. VITAL currently doesn't address this requirement. The skew

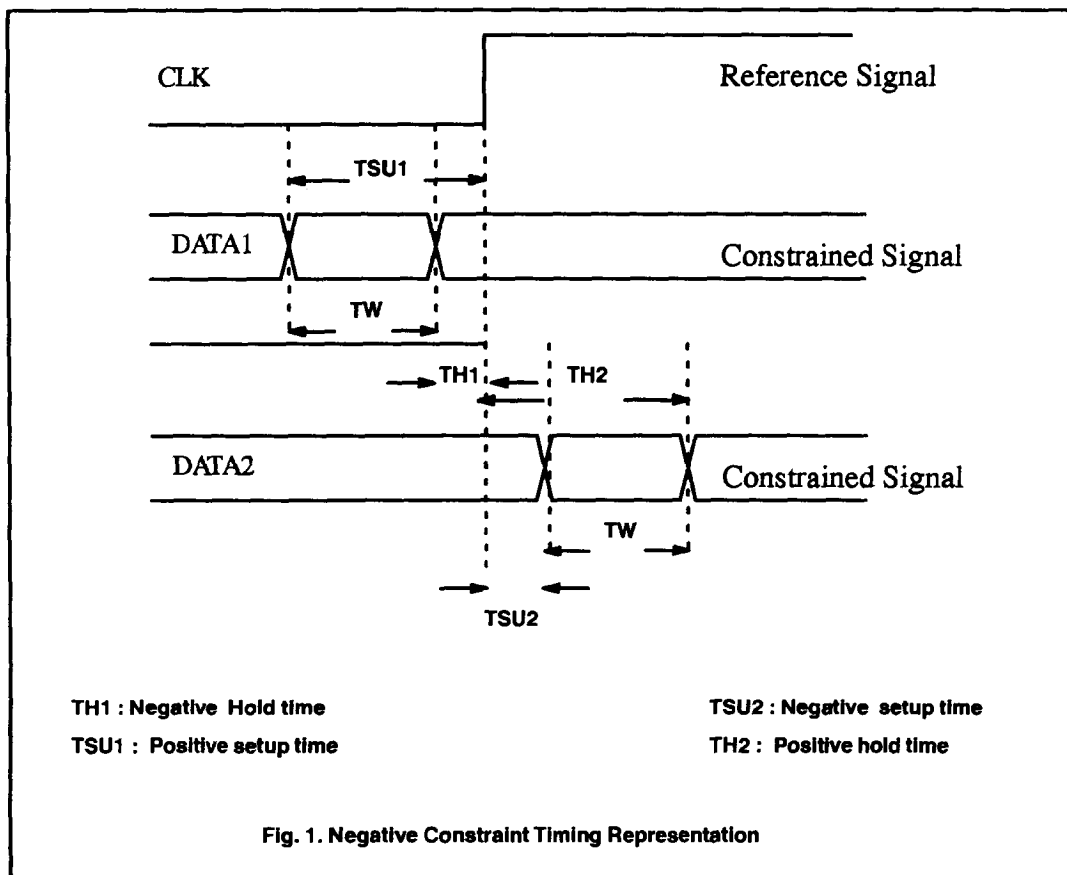
handling is another requirement for phase detector cells which VITAL has not addressed. This requirement is depicted in Fig. 4.

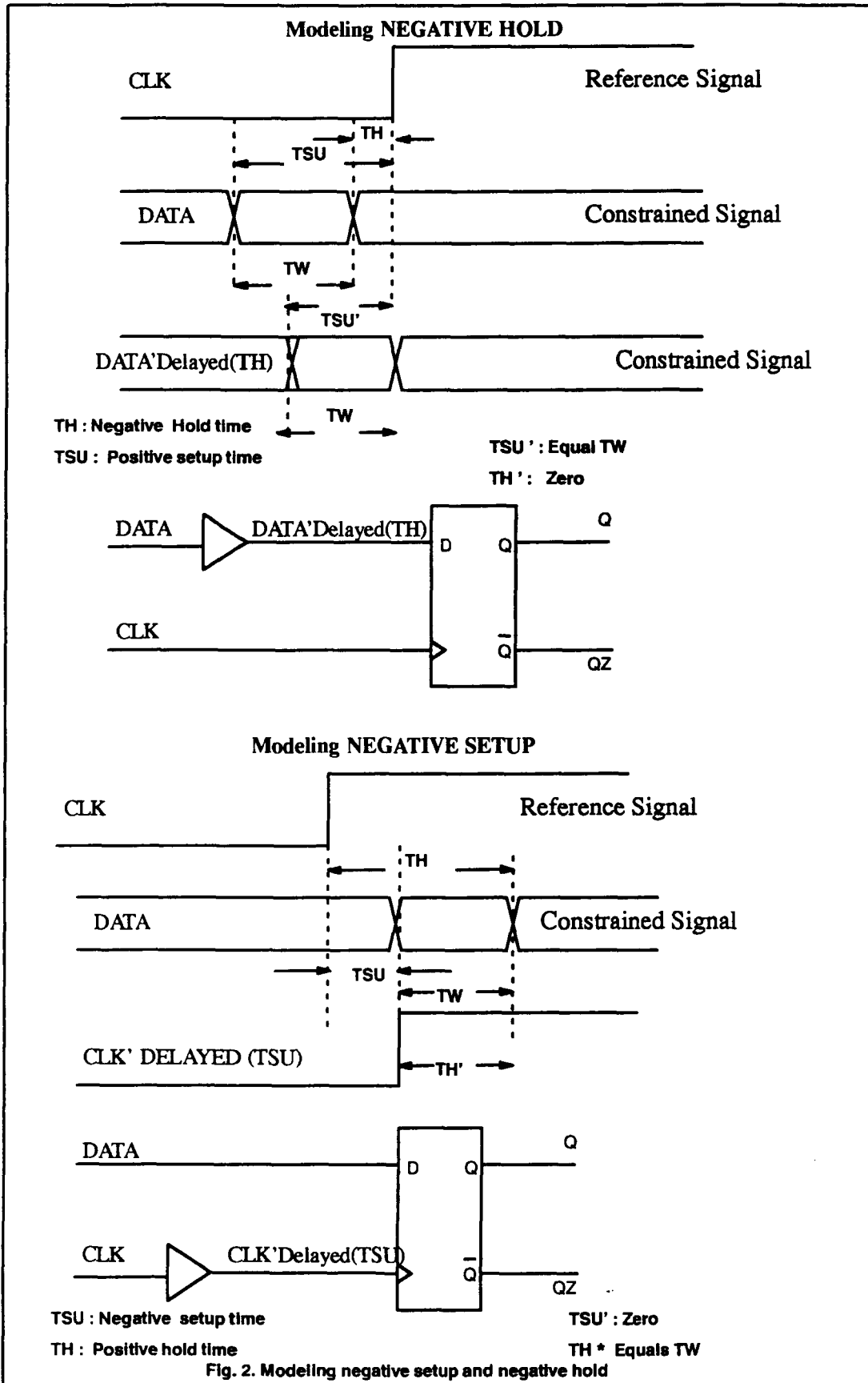
2.1.1: Constraint Timing

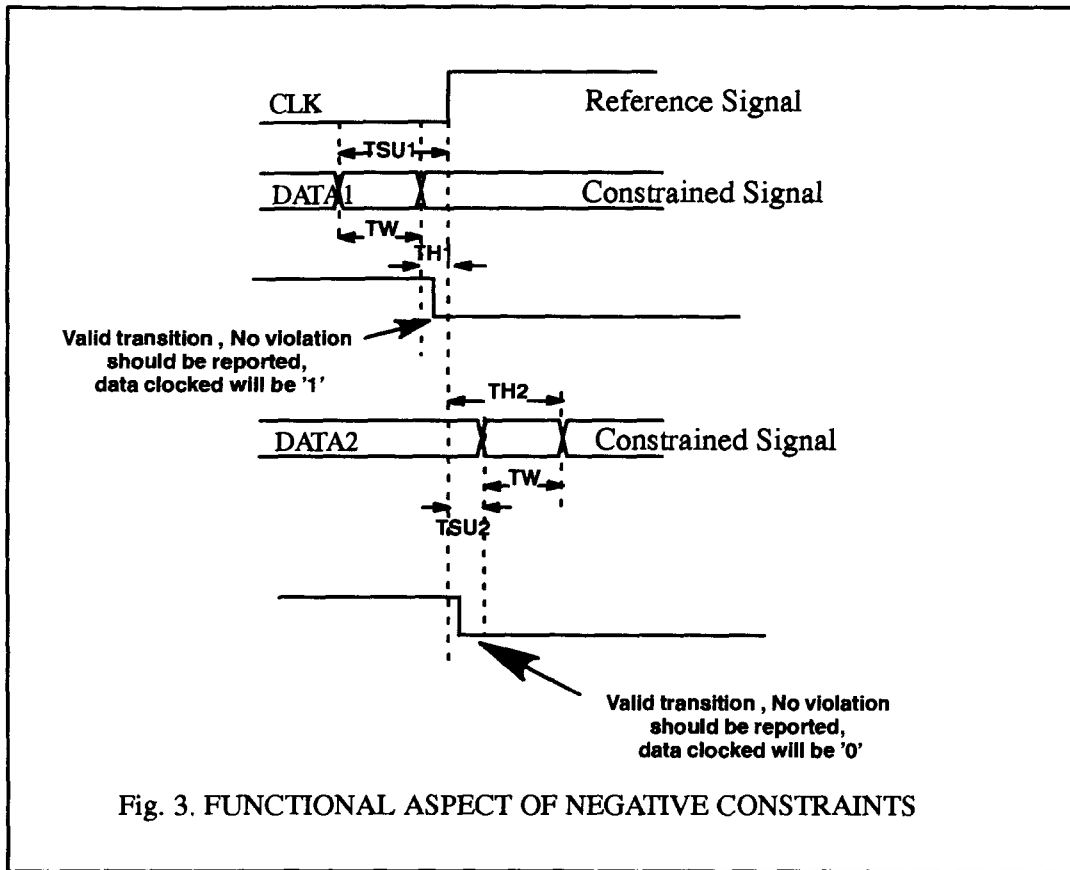
The most difficult problems in modeling different timings in a ASIC model is to comprehend the negative constraints. By constraints we mean the setup and hold time between a reference pin and the constrained pin. ASIC modeler currently needs to supply 3 libraries for the minimum, nominal and maximum operating conditions if the constraints are not annotated and are fixed parameters for these conditions. Now more ASIC cells are designed such that one of the constraint timings becomes negative. Fig. 1 provides a clear picture of different negative constraints. It should be noted however the arithmetic sum of the two constraint timings is always a non-negative number. Modeling negative constraints are not usually easy. The most common method is to move the constraint window such that the reference edge is in line with one of the edges of the window. This essentially means delaying the constrained pin or the reference pin as the case may be. Fig. 2 shows how moving a constraint window will modify the actual setup and hold times. The signal attribute DELAY in VHDL is commonly used to get the derived signal which will model the effect of moving the constraint window. This abstraction if taken into the physical model will translate to insertion of a delay buffer before the reference or constrained pin. While there are other methods to model negative constraints the basic philosophy will remain the same as mentioned above. There is also a subtle functional aspect which needs to be taken care while modeling negative constraint. Fig. 3 shows data signals with a valid transition. In the case of negative hold the data transition to '0' before the CLK edge. In this case there should not be any violation message. Also if it is assumed that this is a signal for a edge sensitive D flipflop then the CLK should have clocked to Q, the output of the flipflop, a value '1' and not '0', the value of the DATA at the CLK edge. A scenario in case of a negative setup is shown completely in the figure3. If the intent is to just provide a warning message on the violation it would have been possible to use a VHDL procedure to handle the negative numbers. It is this functional aspect in negative constraints which neatly fits into the delay buffer

modeling. If we need to model the negative constraint using the above scheme then the actual delay value for each of the pins needs to be determined before hand. If these numbers are to be annotated through an external tool then it may not be possible to find the actual delay values and use it within the model. The complexity arrives when the model have multiple set of constraint timings arising because of many constrained pins with reference to the same reference pin. In some cases the delay values through this buffers cannot be determined at all. If the constraint timings are dependent on design parameters like slew and load then the constraint value itself may not be unique and can vary for every execution of the simulation. Most ASIC vendors even though characterize the path delays with slew and load dependency have however built models with only fixed constrained values. However with emerging new technologies there may be a need to model constraint timings which is characterized as an equation! Example4 shows on how to model negative constraints. VitalPropagateWireDelay procedure can be used to model negative hold only cells but for cells having both

negative setup and negative hold combinations, currently no procedure exists to handle. While constraint timings are one of the complex issues in modeling, a new set of modeling constraints called skew constraints are emerging as challenges in the simulation modeling domain. The recent development of Asynchronous Transfer Mode (ATM) and SONET based communication protocols, ASIC libraries are being designed to support these new protocols. In this protocol all data is available as pairs in the normal and its complement form. Since no synchronization data is available between different communicating systems, a data is presumed to have changed only when both the normal and its complement pair change within a given time. A skew constraint is introduced between the pair and the data and the output is considered as unknown when the pair does not transition within the skew time. A simple AND gate in the ATM technology is shown in Fig. 4. The timing diagram explains the waveforms and its output in the ATM cells. It can be easily seen from this diagram that there is no easy method to take care of the functionality arising out of the skew constraints. Compromises are done to model this behavior and







timing accuracy is lost in the process. It is also not possible in these cells to model the exact timings.

A new range of cells which are exclusively used as variable delay buffers are being available now which is used in communication ASICs. Examples of these are variable duty factor clock drivers, digital delay drivers, digital PLLs etc. These cells have a set of input pins which are exclusively used to program the delay value from an input to an output pin. In case of a clock drivers the value in these pins determine the actual clock period as well as the duty cycle. These kind of cells are classified in the modeling parlance as *state dependent delay* cells since the delay through the cell is dependent on the state of the set of pins. While it is possible to model this in VHDL for lack of a standard naming conventions for the delay identifiers the models are not portable across VHDL simulators. VITAL standards is the first such effort which if implemented will make VHDL models portable across simulators.

2.2: Interconnect Modeling

VitalPropagateWireDelay procedure needs to be used to model interconnect delays. Example4 illustrates how to model interconnects.

3: Issues

The various ASIC VHDL Modeling issues are described as part of the requirements realisation description. VITAL standardisation efforts currently include on resolving many of the following issues:

- Allowing timing checks for internal signals
- Missing VitalPropagatePathDelay for std_logic_vector
- Level 1 Compliance Rules
- Internal signal necessity for complex cell functionality realisation
- Ability to enable/disable Interconnect Delays
- Passing a 'Z' based on a control signal for parallel module test functionality
- Allowing strengths as part of VitalTruthTables
- Handling the cases of Single input event causing multiple output events (Pullup/Pulldown cells)
- Negative Constraints handling by packages

- OVI SDF to VHDL mapping issues
- Other accuracy/performance issues

4: Conclusions

The paper describes how to meet ASIC sign off requirements using VITAL2.2b specification. Many of the common issues in the ASIC modeling world were discussed. Examples are given on modeling for

accuracy using VITAL2.2b specification. With embedded core designs the problem to be confronted will be much wider than just ASIC modeling complexity. We also did not touch upon the other models required for the ASIC design flow such as synthesis library, timing simulator libraries, fault libraries etc.

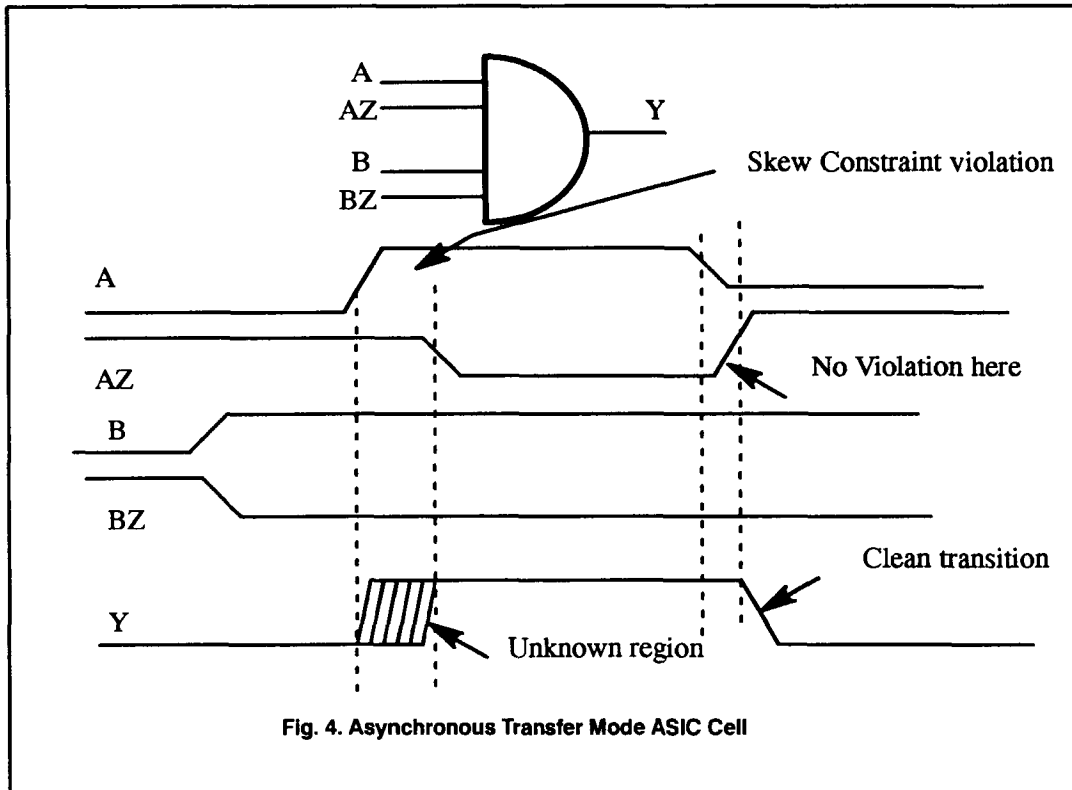


Fig. 4. Asynchronous Transfer Mode ASIC Cell

5: References

- [1] ASIC Sign-Off in VHDL by Oz Levia and Fred Abramson, Synopsys at VIUF'93 conference
- [2] VITAL : VHDL Initiative Towards ASIC libraries Specification version 2.2b 25th Mar'94
- [3] VMS – A VHDL Modeling System by Lakshmikantam Ch and Manohar S, Texas Instruments at VIUF'94 conference.

```
-- Example 1 : Usage of Vital Primitive package to realise combinatorial
--             functionality
-- The following boolean function can be realised using VitalTruthTable
-- and VITAL primitives.
-- Boolean Function : Y = (A1*A2+B1*B2+C1*C2+D1*D2+E1) ;
```

```
library IEEE;
use IEEE.Std_logic_1164.all;
library VITAL;
```

```
use VITAL.VITAL_TIMING.all;
use VITAL.VITAL_PRIMITIVES.all;
```

```
ENTITY BFEX2 IS
GENERIC(
  tipd_A1 : DelayType01Z := ZeroDelay01Z;
  tipd_A2 : DelayType01Z := ZeroDelay01Z;
  tipd_B1 : DelayType01Z := ZeroDelay01Z;
  tipd_B2 : DelayType01Z := ZeroDelay01Z;
  tipd_C1 : DelayType01Z := ZeroDelay01Z;
```

```

tipd_C2 : DelayType01Z := ZeroDelay01Z;
tipd_D1 : DelayType01Z := ZeroDelay01Z;
tipd_D2 : DelayType01Z := ZeroDelay01Z;
tipd_E1 : DelayType01Z := ZeroDelay01Z;
tpd_A1_Y : DelayType01 := (0.94 ns, 1.51 ns);
tpd_A2_Y : DelayType01 := (0.91 ns, 1.72 ns);
tpd_B1_Y : DelayType01 := (1.08 ns, 2.01 ns);
tpd_B2_Y : DelayType01 := (1.05 ns, 2.23 ns);
tpd_C1_Y : DelayType01 := (1.15 ns, 2.41 ns);
tpd_C2_Y : DelayType01 := (1.11 ns, 2.63 ns);
tpd_D1_Y : DelayType01 := (1.16 ns, 2.75 ns);
tpd_D2_Y : DelayType01 := (1.13 ns, 2.98 ns);
tpd_E1_Y : DelayType01 := (0.62 ns, 1.04 ns);
PORT(
  A1 : IN std_logic := 'U';
  A2 : IN std_logic := 'U';
  B1 : IN std_logic := 'U';
  B2 : IN std_logic := 'U';
  C1 : IN std_logic := 'U';
  C2 : IN std_logic := 'U';
  D1 : IN std_logic := 'U';
  D2 : IN std_logic := 'U';
  E1 : IN std_logic := 'U';
  Y : OUT std_logic
);
END BFEF2;

```

ARCHITECTURE arch_VITAL OF BFEF2 IS

```

attribute VITAL_LEVEL1 of arch_VITAL : architecture is TRUE;
signal A1_ipd : std_logic ;
signal A2_ipd : std_logic ;
signal B1_ipd : std_logic ;
signal B2_ipd : std_logic ;
signal C1_ipd : std_logic ;
signal C2_ipd : std_logic ;
signal D1_ipd : std_logic ;
signal D2_ipd : std_logic ;
signal E1_ipd : std_logic ;
signal E2EG_5949 : std_logic ;
BEGIN
  -- Input Port Delay Block
  WIRE_DELAY : BLOCK
  BEGIN
    VitalPropagateWireDelay(A1_ipd,A1,tipd_A1);
    VitalPropagateWireDelay(A2_ipd,A2,tipd_A2);
    VitalPropagateWireDelay(B1_ipd,B1,tipd_B1);
    VitalPropagateWireDelay(B2_ipd,B2,tipd_B2);
    VitalPropagateWireDelay(C1_ipd,C1,tipd_C1);
    VitalPropagateWireDelay(C2_ipd,C2,tipd_C2);
    VitalPropagateWireDelay(D1_ipd,D1,tipd_D1);
    VitalPropagateWireDelay(D2_ipd,D2,tipd_D2);
    VitalPropagateWireDelay(E1_ipd,E1,tipd_E1);
  END BLOCK;

```

--- BEHAVIOR SECTION

```

VITALBehavior :
PROCESS (A1_ipd, A2_ipd, B1_ipd, B2_ipd, C1_ipd, C2_ipd, D1_ipd,
D2_ipd, E1_ipd)
  VARIABLE Results : STD_LOGIC_VECTOR(1 to 1) := (others =>
'X');
  ALIAS Y_zd : STD_LOGIC is Results(1);
  VARIABLE E2EG_5925, E2EG_5926 : std_logic;
  VARIABLE E2EG_5949 : std_logic ;

```

```

VARIABLE GlitchData_Y : GlitchDataType;
CONSTANT AORUDP: VITALTruthTableType(0 to 5, 0 to 4) :=
(( '0', '-', '0', '-', '0' ),
 ( '0', '-', '-', '0', '0' ),
 ( '-', '0', '0', '-', '0' ),
 ( '-', '0', '-', '0', '0' ),
 ( '1', '1', '-', '-', '1' ),
 ( '-', '-', '1', '1', '1' ));

```

begin

--- Functionality Section

```

E2EG_5925 := VitalAND2 (B1_ipd, B2_ipd);
E2EG_5949 := VitalTruthTable (
  TruthTable => AORUDP,
  DataIn => std_logic_vector'(D1_ipd, D2_ipd, C1_ipd, C2_ipd));
E2EG_5926 := VitalAND2 (A1_ipd, A2_ipd);
Y_zd      := VitalOR4 (E2EG_5925, E2EG_5949, E2EG_5926,
E1_ipd);

```

--- Path Delay Section

VitalPropagatePathDelay (

```

  OutSignal => Y,
  OutSignalName => "Y",
  OutTemp => Y_zd,
  Paths => (
    0 => (
      InputChangeTime => A1_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_A1_Y),
      PathCondition => TRUE
    ),
    1 => (
      InputChangeTime => A2_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_A2_Y),
      PathCondition => TRUE
    ),
    2 => (
      InputChangeTime => B1_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_B1_Y),
      PathCondition => TRUE
    ),
    3 => (
      InputChangeTime => B2_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_B2_Y),
      PathCondition => TRUE
    ),
    4 => (
      InputChangeTime => C1_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_C1_Y),
      PathCondition => TRUE
    ),
    5 => (
      InputChangeTime => C2_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_C2_Y),
      PathCondition => TRUE
    ),
    6 => (
      InputChangeTime => D1_ipd'last_event,
      PathDelay => VitalExtendToFillDelay(tpd_D1_Y),
      PathCondition => TRUE
    )
  )
);

```

```

),
7 => (
    InputChangeTime => D2_ipd'last_event,
    PathDelay => VitalExtendToFillDelay(tpd_D2_Y),
    PathCondition => TRUE
),
8 => (
    InputChangeTime => E1_ipd'last_event,
    PathDelay => VitalExtendToFillDelay(tpd_E1_Y),
    PathCondition => TRUE
)
),
GlitchData => GlitchData_Y,
GlitchMode => MessagePlusX
);
END PROCESS;

end arch_VITAL;

configuration CFG_BFEX2_VITAL of BFEX2 is
    for arch_VITAL
    end for;
end CFG_BFEX2_VITAL;

-- Example2 : Usage of Vital state table package to realise sequential cell
-- functionality
-- This example demonstrates how to tackle to and from 'X'
-- transitions when a FlipFlop is realised using a two latch
-- configuration

----- CELL DFF -----
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library VITAL;
use VITAL.VITAL_Timing.all;

-- entity declaration --
entity DFF is
    generic(
        TimingChecksOn: Boolean := True;
        XGenerationOn: Boolean := True;
        InstancePath: STRING := "";
        tpd_CLK_Q      : DelayType01 := (1.371 ns, 1.392 ns);
        tpd_CLK_QZ     : DelayType01 := (1.125 ns, 1.213 ns);
        tsetup_D_CLK   : DelayTypeXX := 0.980 ns;
        thold_CLK_D    : DelayTypeXX := 0.130 ns;
        tpd_D_Q_QZ     : DelayType01 := (0.00 ns, 0.00 ns);
        tpw_CLK_posedge : DelayTypeXX := 1.030 ns;
        tpw_CLK_negedge : DelayTypeXX := 0.830 ns;
        tipd_CLK       : DelayType01 := (0.00 ns, 0.00 ns);
        tipd_D         : DelayType01 := (0.00 ns, 0.00 ns));
    port(
        CLK      : in  STD_LOGIC;
        D        : in  STD_LOGIC;
        Q        : out STD_LOGIC;
        QZ       : out STD_LOGIC);
end DFF;

-- architecture body --
library VITAL;
use VITAL.VITAL_Primitives.all;
library WORK;
use WORK.VTABLES.all;
architecture VITAL of DFF is

```

```

    SIGNAL CLK_ipd : STD_LOGIC := 'X';
    SIGNAL D_ipd   : STD_LOGIC := 'X';

begin

-----
-- INPUT PATH DELAYs
-----
    WIRE_DELAY : BLOCK
    BEGIN
        VitalPropagateWireDelay (CLK_ipd, CLK,
        VitalExtendToFillDelay(tipd_CLK));
        VitalPropagateWireDelay (D_ipd, D,
        VitalExtendToFillDelay(tipd_D));
    END BLOCK;

-----
-- BEHAVIOR SECTION
-----
    VITALBehavior : PROCESS (CLK_ipd, D_ipd)

-- timing check results
    VARIABLE Tviol_D_CLK : X01 := '0';
    VARIABLE Tmkr_D_CLK : TimeMarkerType;
    VARIABLE Pviol_CLK : X01 := '0';
    VARIABLE PInfo_CLK : PeriodCheckInfoType :=
    PeriodCheckInfoInit;

-- functionality results
    VARIABLE Violation : X01 := '0';
    VARIABLE PrevData1 : STD_LOGIC_VECTOR(1 to 3) := (others =>
    'X');
    VARIABLE PrevData2 : STD_LOGIC_VECTOR(1 to 3) := (others =>
    'X');
    VARIABLE Results1 : STD_LOGIC_VECTOR(1 to 2) := (others =>
    'X');
    VARIABLE Results2 : STD_LOGIC_VECTOR(1 to 2) := (others =>
    'X');
    VARIABLE IINVnet1 : std_logic;
    ALIAS Q_zd : STD_LOGIC is Results2(1);
    ALIAS QZ_zd : STD_LOGIC is Results2(2);

-- output glitch detection variables
    VARIABLE Q_GlitchData : GlitchDataType;
    VARIABLE QZ_GlitchData : GlitchDataType;
    CONSTANT DFF_tab : VitalStateTableType := (
--
--Vio CLK D IQ Q QZ
--
    ( 'X', '-', '-', '-', 'X', 'X' ),
    ( '-', 'N', '-', '-', 'S', 'S' ),
    ( '-', '0', '*', '-', 'S', 'S' ),
    ( '-', '1', 'N', '-', '0', '1' ),
    ( '-', 'P', '0', '-', '0', '1' ),
    ( '-', '1', 'P', '-', '1', '0' ),
    ( '-', 'P', '1', '-', '1', '0' ),
    ( '-', '-', 'P', '1', '1', '0' ),
    ( '-', '*', '1', '1', '1', '0' ),
    ( '-', '-', 'N', '0', '0', '1' ),
    ( '-', '*', '0', '0', '0', '1' ),
    ( 'X', '-', '-', '-', 'X', 'X' ),
    ( '-', '1', '0', '-', '0', '1' ),
    ( '-', '1', '1', '-', '1', '0' ),
    ( '-', 'B', '-', '-', 'S', 'S' ));

begin

```

```

-----
-- Timing Check Section
-----
if (TimingChecksOn) then
  VitalTimingCheck (D_ipd, "D", CLK_ipd, "CLK",
    tsetup_D_CLK, tsetup_D_CLK,
    thold_CLK_D, thold_CLK_D,
    TRUE,
    To_X01(CLK_ipd) = '1',
    InstancePath & "/DFF",
    Tmkr_D_CLK, Tviol_D_CLK);
  VitalPeriodCheck (CLK_ipd, "CLK",
    0 ns, TIME'HIGH,
    tpw_CLK_posedge, TIME'HIGH,
    tpw_CLK_negedge, TIME'HIGH,
    PInfo_CLK, Pviol_CLK,
    InstancePath & "/DFF",
    TRUE);
end if;

-----
-- Functionality Section
-----
Violation := Tviol_D_CLK or Pviol_CLK;
IINVnet1 := VitalINV (CLK_ipd);

VitalStateTable(StateTable => DFF_tab,
  DataIn => (Violation, IINVnet1, D_ipd),
  NumStates => 1,
  Result => Results1,
  PreviousDataIn => PrevData1);
VitalStateTable(StateTable => DFF_tab,
  DataIn => (Violation, CLK_ipd, Results1(1)),
  NumStates => 1,
  Result => Results2,
  PreviousDataIn => PrevData2);

-----
-- Path Delay Section
-----
VitalPropagatePathDelay (
  OutSignal => Q,
  OutSignalName => "Q",
  OutTemp => Q_zd,
  Paths => (0 => (CLK_ipd'last_event,
VitalExtendToFillDelay(tpd_CLK_Q), TRUE),
  1 => (D_ipd'last_event, VitalExtendToFillDelay(tpd_D_Q_QZ),
TRUE)
  ),
  GlitchData => Q_GlitchData,
  GlitchMode => MessagePlusX,
  GlitchKind => OnEvent);
VitalPropagatePathDelay (
  OutSignal => QZ,
  OutSignalName => "QZ",
  OutTemp => QZ_zd,
  Paths => (0 => (CLK_ipd'last_event,
VitalExtendToFillDelay(tpd_CLK_QZ), TRUE),
  1 => (D_ipd'last_event, VitalExtendToFillDelay(tpd_D_Q_QZ),
TRUE)
  ),
  GlitchData => QZ_GlitchData,
  GlitchMode => MessagePlusX,
  GlitchKind => OnEvent);

```

```

END PROCESS;

end VITAL;

configuration CFG_DFF_VITAL of DFF is
  for VITAL
    end for;
end CFG_DFF_VITAL;

-- Example 3: PULLUP Resistor. This example demonstrates the use of
"ResultMapType"
-- to realise strengths.

library IEEE;
use IEEE.STD_LOGIC_1164.all;
library VITAL;
use VITAL.VITAL_Timing.all;

-- entity declaration --
entity PULLUP is
  generic(
    tpd_PWRDN_TAP      : DelayType01z :=
      (0.000 ns, 0.000 ns, 0.000 ns, 0.000 ns, 0.000 ns, 0.000 ns);
    tipd_PWRDN        : DelayType01 := (0.000 ns, 0.000 ns);
    tipd_TAP          : DelayType01 := (0.000 ns, 0.000 ns);

    port(
      PWRDN            : in STD_LOGIC;
      TAP              : inout STD_LOGIC);
  end PULLUP;

-- architecture body --
library VITAL;
use VITAL.VITAL_Primitives.all;

architecture VITAL of PULLUP is
  SIGNAL PWRDN_ipd : STD_LOGIC := 'X';
  SIGNAL TAP_ipd  : STD_LOGIC := 'X';

begin

  -----
  -- INPUT PATH DELAYS
  -----
  WIRE_DELAY : BLOCK
  BEGIN
    VitalPropagateWireDelay (PWRDN_ipd, PWRDN,
VitalExtendToFillDelay(tipd_PWRDN));
    VitalPropagateWireDelay (TAP_ipd, TAP,
VitalExtendToFillDelay(tipd_TAP));
  END BLOCK;

  -----
  -- BEHAVIOR SECTION
  -----
  VITALBehavior : PROCESS (PWRDN_ipd, TAP_ipd)

  -- timing check results

  -- functionality results
  VARIABLE TAP_ipd2 : STD_LOGIC := 'X';
  VARIABLE Results : STD_LOGIC_VECTOR(1 to 1) := (others =>
'X');
  ALIAS TAP zd : STD_LOGIC is Results(1);

```

```

-- output glitch detection variables
VARIABLE TAP_GlitchData : GlitchDataType;

begin

-----
-- Functionality Section
-----
TAP_ipd2 := VitalIdent (data => TAP_ipd,
    ResultMap => ('U','X','0','1','H'));
TAP_zd := VitalBUFIFO (data => '1',
    enable => PWRDN_ipd,
    ResultMap => ('U','X','0','1','H'));

-----
-- Path Delay Section
-----
VitalPropagatePathDelay (
    OutSignal => TAP,
    OutSignalName => "TAP",
    OutTemp => TAP_zd,
    Paths => (0 => (PWRDN_ipd'last_event,
VitalExtendToFillDelay(tpd_PWRDN_TAP), TRUE)),
    GlitchData => TAP_GlitchData,
    GlitchMode => MessagePlusX,
    GlitchKind => OnEvent);

ENDPROCESS;

end VITAL;

configuration CFG_PULLUP_VITAL of PULLUP is
    for VITAL
    end for;
end CFG_PULLUP_VITAL;

-- Example4 : Negative constraint Modeling Example - Limitations
-- The following example demonstrates the use of internal signals
-- The use of internal signal is to realise the functionlity due
-- negative constraints

library IEEE;
use IEEE.STD_LOGIC_1164.all;
library VITAL;
use VITAL.VITAL_Timing.all;

-- entity declaration --
entity MUXSCANFF is
    generic(
        TimingChecksOn: Boolean := True;
        XGenerationOn: Boolean := True;
        InstancePath: STRING := "";
        tpd_CLK_Q      : DelayType01 := (1.371 ns, 1.392 ns);
        tpd_CLK_QZ     : DelayType01 := (1.125 ns, 1.213 ns);
        tsetup_D_CLK   : DelayTypeXX := 1.070 ns;
        thold_CLK_D    : DelayTypeXX := 0.020 ns;
        tsetup_SCAN_CLK : DelayTypeXX := 1.920 ns;
        thold_CLK_SCAN : DelayTypeXX := -0.210 ns;
        tsetup_SD_CLK  : DelayTypeXX := 3.470 ns;
        thold_CLK_SD   : DelayTypeXX := -1.540 ns;
        tpw_CLK_posedge : DelayTypeXX := 1.030 ns;
        tpw_CLK_negedge : DelayTypeXX := 0.830 ns;
        tpd_CLK        : DelayType01Z := ZeroDelay01Z;

```

```

tipd_D      : DelayType01Z := ZeroDelay01Z;
tipd_SCAN   : DelayType01Z := ZeroDelay01Z;
tipd_SD     : DelayType01Z := ZeroDelay01Z;

```

```

port(
    CLK      : in  STD_LOGIC;
    D        : in  STD_LOGIC;
    SCAN     : in  STD_LOGIC;
    SD       : in  STD_LOGIC;
    Q        : out STD_LOGIC;
    QZ       : out STD_LOGIC;
end MUXSCANFF;

```

```

-- architecture body --
library VITAL;
use VITAL.VITAL_Primitives.all;
library WORK;
use WORK.VTABLES.all;
architecture VITAL of MUXSCANFF is
    SIGNAL CLK_ipd : STD_LOGIC := 'X';
    SIGNAL D_ipd   : STD_LOGIC := 'X';
    SIGNAL SCAN_ipd : STD_LOGIC := 'X';
    SIGNAL SD_ipd  : STD_LOGIC := 'X';

```

```
begin
```

```
-----
-- INPUT PATH DELAYs
-----
```

```

WIRE_DELAY : BLOCK
    signal SD_ipd1,SCAN_ipd1 : STD_LOGIC;
BEGIN
    VitalPropagateWireDelay (CLK_ipd, CLK, tipd_CLK);
    VitalPropagateWireDelay (D_ipd, D, tipd_D);
    VitalPropagateWireDelay (SCAN_ipd1, SCAN, tipd_SCAN);
    SCAN_ipd <= transport SCAN_ipd1 after (0 ns - thold_CLK_SCAN);
    VitalPropagateWireDelay (SD_ipd1, SD, tipd_SD);
    SD_ipd <= transport SD_ipd1 after (0 ns -thold_CLK_SD);
END BLOCK;

```

```
-----
-- BEHAVIOR SECTION
-----
```

```
VITALBehavior : PROCESS (CLK_ipd, D_ipd, SCAN_ipd, SD_ipd)
```

```
-- timing check results
```

```

VARIABLE Tviol_D_CLK : X01 := '0';
VARIABLE Tmkr_D_CLK : TimeMarkerType;
VARIABLE Tviol_SCAN_CLK : X01 := '0';
VARIABLE Tmkr_SCAN_CLK : TimeMarkerType;
VARIABLE Tviol_SD_CLK : X01 := '0';
VARIABLE Tmkr_SD_CLK : TimeMarkerType;
VARIABLE Pviol_CLK : X01 := '0';
    VARIABLE PInfo_CLK : PeriodCheckInfoType :=
PeriodCheckInfoInit;

```

```
-- functionality results
```

```

VARIABLE Violation : X01 := '0';
VARIABLE PrevData : STD_LOGIC_VECTOR(1 to 5) := (others =>
'X');
    VARIABLE Results : STD_LOGIC_VECTOR(1 to 2) := (others =>
'X');
    ALIAS Q_zd : STD_LOGIC is Results(1);
    ALIAS QZ_zd : STD_LOGIC is Results(2);

```

```

-- output glitch detection variables
VARIABLE Q_GlitchData : GlitchDataType;
VARIABLE QZ_GlitchData : GlitchDataType;

begin

-----
-- Timing Check Section
-----
if (TimingChecksOn) then
  VitalTimingCheck (SD_ipd, "SD", CLK_ipd, "CLK",
    tsetup_SD_CLK + thold_CLK_SD, tsetup_SD_CLK +
thold_CLK_SD,
    ((0 ns - thold_CLK_SD) + thold_CLK_SD), ((0 ns -
thold_CLK_SD) + thold_CLK_SD),
    ((CLK_ipd = '1') AND (((SCAN_ipd /= '0')))),
    To_X01(CLK_ipd) = '1',
    InstancePath & "/MUXSCANFF",
    Tmkr_SD_CLK, Tviol_SD_CLK);
  VitalTimingCheck (D_ipd, "D", CLK_ipd, "CLK",
    tsetup_D_CLK, tsetup_D_CLK,
thold_CLK_D, thold_CLK_D,
    ((CLK_ipd = '1') AND (((SCAN_ipd /= '1')))),
    To_X01(CLK_ipd) = '1',
    InstancePath & "/MUXSCANFF",
    Tmkr_D_CLK, Tviol_D_CLK);
  VitalTimingCheck (SCAN_ipd, "SCAN", CLK_ipd, "CLK",
    tsetup_SCAN_CLK + thold_CLK_SCAN, tsetup_SCAN_CLK
+ thold_CLK_SCAN,
    ((0 ns - thold_CLK_SCAN) + thold_CLK_SCAN), ((0 ns -
thold_CLK_SCAN) + thold_CLK_SCAN),
    ((CLK_ipd = '1') AND (((D_ipd /= SD_ipd)))),
    To_X01(CLK_ipd) = '1',
    InstancePath & "/MUXSCANFF",
    Tmkr_SCAN_CLK, Tviol_SCAN_CLK);
  VitalPeriodCheck (CLK_ipd, "CLK",
    0 ns, TIME'HIGH,
tpw_CLK_posedge, TIME'HIGH,
tpw_CLK_negedge, TIME'HIGH,
PInfo_CLK, Pviol_CLK,
InstancePath & "/MUXSCANFF",
TRUE);
end if;

```

```

-----
-- Functionality Section
-----
Violation := Tviol_D_CLK or Tviol_SCAN_CLK or Tviol_SD_CLK
or Pviol_CLK;
VitalStateTable(StateTable => MUXSCANFF_tab,
  DataIn => (Violation, CLK_ipd, D_ipd, SCAN_ipd, SD_ipd),
  NumStates => 1,
  Result => Results,
  PreviousDataIn => PrevData);

-----
-- Path Delay Section
-----
VitalPropagatePathDelay (
  OutSignal => Q,
  OutSignalName => "Q",
  OutTemp => Q_zd,
  Paths => (0 => (CLK_ipd'last_event,
VitalExtendToFillDelay(tpd_CLK_Q), TRUE)),
  GlitchData => Q_GlitchData,
  GlitchMode => MessagePlusX,
  GlitchKind => OnEvent);
VitalPropagatePathDelay (
  OutSignal => QZ,
  OutSignalName => "QZ",
  OutTemp => QZ_zd,
  Paths => (0 => (CLK_ipd'last_event,
VitalExtendToFillDelay(tpd_CLK_QZ), TRUE)),
  GlitchData => QZ_GlitchData,
  GlitchMode => MessagePlusX,
  GlitchKind => OnEvent);

ENDPROCESS;

end VITAL;

configuration CFG_MUXSCANFF_VITAL of MUXSCANFF is
  for VITAL
  end for;
end CFG_MUXSCANFF_VITAL;

```