

VMS: A VHDL Modeling System

Ch Lakshmikantam and S Manohar

Texas Instruments (India) Pvt. Ltd.,
71, Miller Road, Bangalore 560052, INDIA

Abstract

VMS (A VHDL Modeling System) is a tool which automates the model generation of ASIC gate level cells for VHDL logic simulators. VMS is a rule based system and has built into it the expertise of model developers in the form of rules. VMS handles many of the complex and intricate model behavior and is a high productivity tool.

1: Introduction

In today's competitive ASIC market, a short design cycle time gives a clear edge over competitors for any ASIC vendor. A low design cycle time dictates the use of the cell-based design methodology or, what is now more commonly termed, ASIC design methodology. Here the design consists of instances of cells from the library provided by the ASIC vendor. This methodology is supported by many sophisticated tools which span from the logic design to the physical design environment. Each of the tools supporting the ASIC design methodology requires data which describe the technology, characteristics of the cells as well as data on speed and area. ASIC libraries are repositories of this information. All tools used in the ASIC design methodology like logic simulators, synthesizers, timing analyzers, electrical rule checkers and layout tools require cell libraries. For example, a logic simulator requires logic models for each of the cells in the design, and a synthesis tool requires libraries which provide information on cell functionality, its area, power consumption, drive factor etc. To stay in competition, silicon vendors need to provide and support libraries for many different kind of tools and in all the major commercial CAD software systems. Though VHDL, being the IEEE standard hardware description language, is gaining popularity in the design community, still the biggest impediment to VHDL designs is the lack of ASIC libraries. With the VITAL's evolving standard to model ASIC gate level cells, ASIC vendors are all set to

provide VHDL simulation libraries to the designers. A short cycle time in providing a VHDL library for a new, or an existing, ASIC technology is again an important factor for an ASIC vendor. VMS automates the VITAL-compliant model generation for ASIC libraries, and is a high productivity tool which brings down the library generation and release cycle time from 2 months (approximately – for an expert model developer) to as low as 2 weeks.

2: VMS flow

Fig. 1 shows the VMS model generation flow. The user provides two main inputs to the system: the functional description in equation format, and a timing database containing all the timing information for the cell. VMS processes the input equation, synthesizes, optimizes, and then generates the full timing VITAL-compliant VHDL [1,2] model. Description of various modules in the flow is given in the following sections.

2.1: Functional specification

The equation format used to specify the functionality of a cell is an extension of the equation format accepted by MIS [3]. In addition to the constructs provided by the MIS equation format, the extended equation format has constructs to specify the functionality of sequential primitives and other special functions. Fig. 2 shows the boolean operators and extended constructs available in the extended equation format, and Fig. 3 shows the syntax of the functional specification in the extended equation format. For example, the extended equation construct FLIPFLOP has the fields to specify DATA, CLOCK, SET and RESET signals. The SET_RESET_ACTIVE field allows the user to specify the behavior of the flipflop when both the SET and RESET signals are active. For instance, a value 10 assigned to SET_RESET_ACTIVE implies that SET dominates when both the asynchronous inputs are active. The COMPLEMENT field accommodates the complementary output of the flipflop, if any.

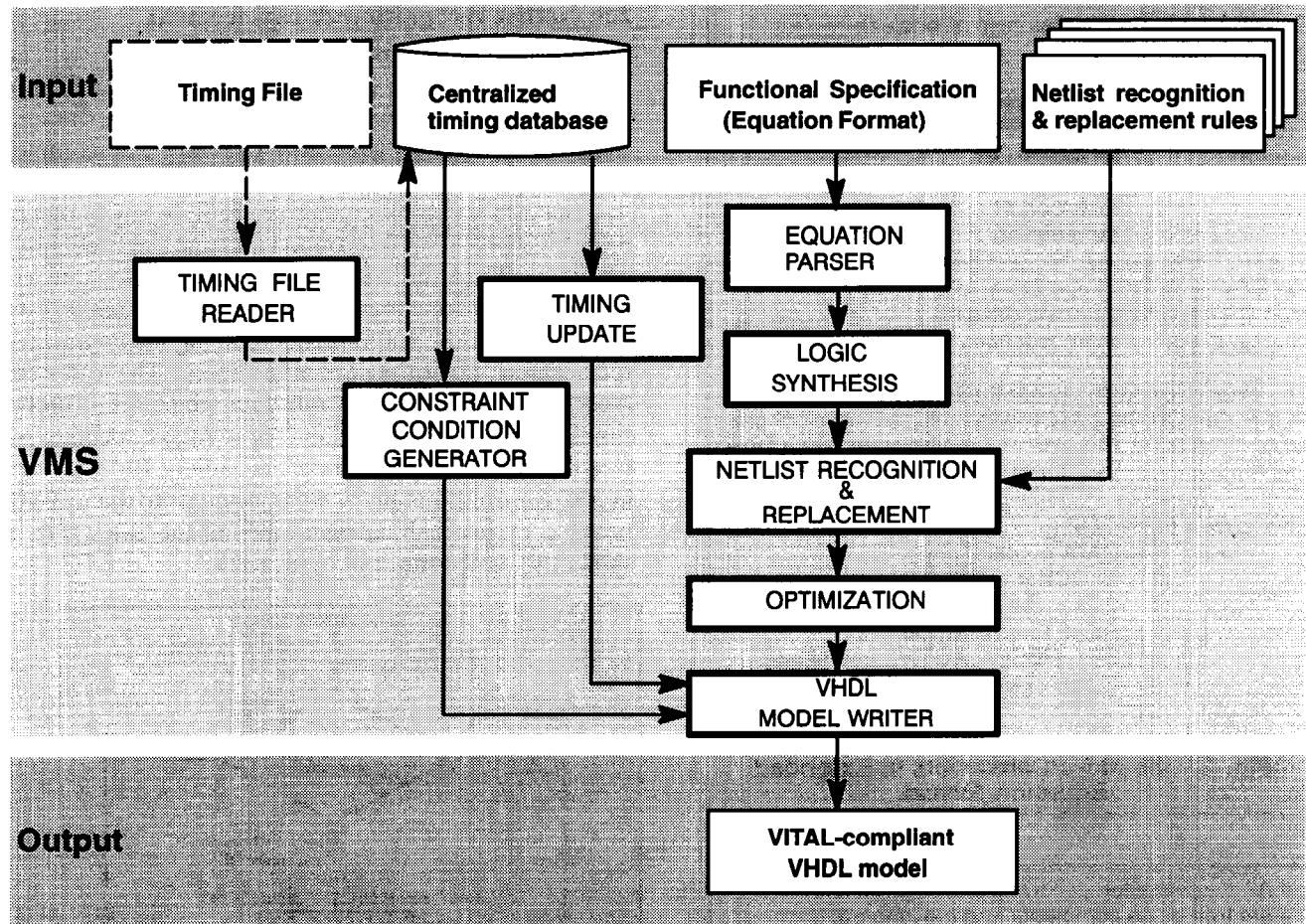


Fig. 1. VMS Flow.

2.2: Timing database

Usually, an object oriented timing database is used to store the characterized timing information for all the cells in an ASIC library. The procedural interface to such a database allows the *timing update* module to retrieve the timing information for any given cell.

2.3: Equation parser

Equation Parser is the front-end for the VMS flow. It parses the extended equation specification of the cell, and partitions it into combinational part and sequential part.

2.4: Logic synthesis

Fig. 4 shows the flow chart for logic synthesis. VMS has a primitive library associated with it. The primitive library is developed such that all the constructs in the extended equation format can be internally mapped to this primitive set. Various *boolean gates, multiplexers, latches, flipflops, pullups, pulldowns, busholders*, etc. are present

in the primitive library. Any functional specification is synthesized using this set of primitives. The combinational part of the output of the equation parser, if any, is fed to MIS [3] for synthesis. MIS synthesizes this part, and maps its input to a set of primitives available in the library provided to the tool. The output of MIS is an ASCII file which gives the netlist description of the cell in terms of VMS primitives. The netlist of the primitive cells present in this ASCII file is then translated and stored in a generic design database. If the output of the equation parser is only combinational, we have the generic netlist of the cell in terms of the primitives of the VMS system. If the output of the equation parser is only sequential, then the generic netlist is obtained by directly instantiating VMS primitives, corresponding to each statement that use extended equation construct, in the design database. If the output of the equation parser consists of both the combinational and sequential statements, then the databases from both the paths are merged to get the complete design database for the cell.

The allowed logical operators in all boolean expressions are:

```
!           for logical NOT
* (or &)   for logical AND
+ (or |)   for logical OR
^ (or !=)  for EX-OR
==         for EX-NOR
(          for grouping
```

The NOT operator has highest precedence, followed by the EX-OR/EX-NOR operators, followed by AND and then OR operators. Parentheses may be freely used to change the order of precedence.

The extended constructs are the functions such as FLIPFLOP, LATCH, LATCH with multiple *data-clock* pairs, MUX, TRISTATE, PULLUP, PULLDOWN, BUSHOLDER, etc. For example, the FLIPFLOP construct is as follows.

```
<out_pin> = FLIPFLOP (
    DATA = <data_pin>,
    CLOCK = <clock_pin>,
    SET = <set_pin>,
    RESET = <reset_pin>,
    SET_RESET_ACTIVE = <value>,
    COMPLEMENT = <comp_out_pin>
);
```

Fig. 2. Operators/Constructs in Extended Equation format.

All the characters following # on a given line # are treated as comments and are ignored.

Give the name of the cell.
NAME = <cell_name>;

Give all the input port names.
The names are separated by blanks.
INORDER = <name1> [<name2> ...];

Give all the inout port names.
IOORDER = <name1> [<name2> ...];

Give all the output port names.
OUTORDER = <name1> [<name2> ...];

Give all the temporary signal names.
TEMP = <name1> [<name2> ...];

EQUATION

The first statement below shows a boolean # statement, and the second one shows the # syntax for the usage of an extended construct.

```
<signal> = <boolean_expr>;
<signal> = <function_name>
           ({<keyword>=[!]<name>}
           {,<keyword>=[!]<name>}*)
           );
```

ENDEQUATION;

Fig. 3. Extended Equation Format.

2.5: Netlist recognition & replacement

This is the most important part of the VMS flow. The role of this module is to optimize the netlist present in the design database, which in turn determines the efficiency of the models generated by VMS. The complex netlist recognition algorithm works on the generic netlist (obtained in the previous step) to replace any netlist of primitives which can be grouped into one single complex primitive. Then the netlist replacement algorithm replaces each of the complex primitives in the netlist by a specific netlist. Both the recognition algorithm and the replacement algorithm are rule driven. The rules for these are provided by the rule files which are, usually, supplied as part of the system. These rules basically capture the expertise of the model developers. Rule files are in ASCII format and many enhancements to the tool are possible by just enhancing/modifying the rules. Fig. 5 shows the basic format of input rules.

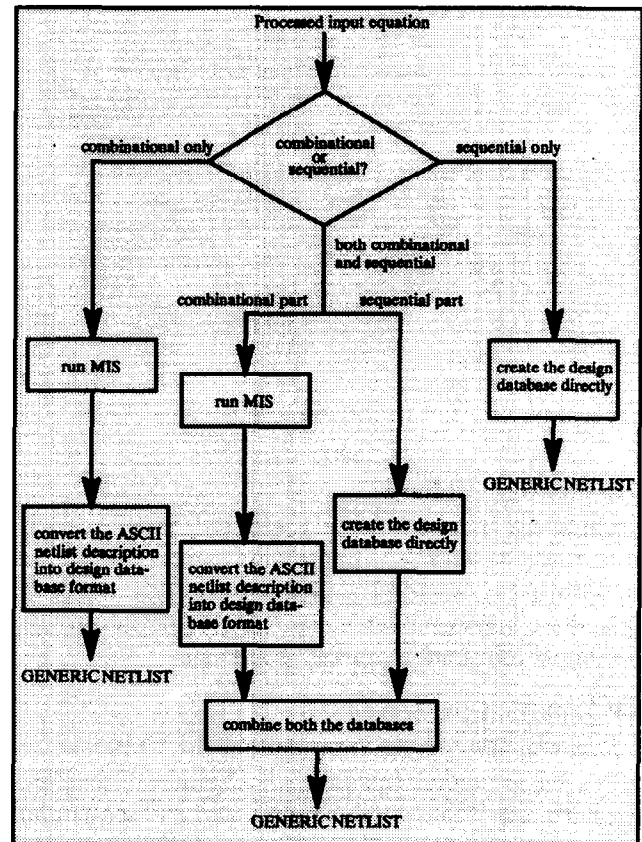


Fig. 4. Logic Synthesis.

2.6: Optimization

A great deal of the netlist optimization is achieved through *netlist recognition & replacement* using input rules to the VMS system. Further optimization can be achieved by removing unnecessary buffers or inverters in

the design netlist. Fig. 6 shows a simple example case of the inverter optimization. When multiple inverters are used to drive the *active-low reset* inputs of a series of latches, the inverters can be replaced with a single inverter as shown in the figure.

```

COMPLEX_CELL: <inputs>:<outputs>:<inouts>;
{
  Netlist to be recognized
}:=
{
  Netlist to be used in replacement
};

An example rule to recognize three 2X1 multiplexers and
replace them with a single 4X1 multiplexer:

VMS_MUX_COMP; sel1 sel2 d1 d2 d3 d4 : out ::
{
  VMS_MUX1; sel1 d1 d2 : tmp1 ::
  VMS_MUX1; sel1 d3 d4 : tmp2 ::
  VMS_MUX1; sel2 tmp1 tmp2 : out ::
}:=
{
  VMS_MUX2; sel1 sel2 d1 d2 d3 d4 : out ::
};

```

Fig. 5. Format of input rules.

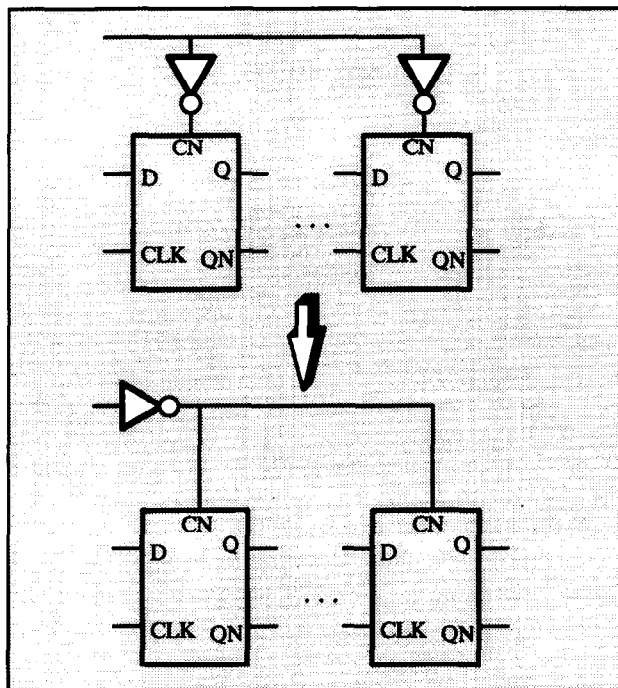


Fig. 6. Inverter Optimization Example.

2.7: Constraint condition generator

Various constraint checking (between the input signals of a sequential cell) statements are the important

part of any logic simulation model. But, all the constraints need not be checked always. For example, in a flipflop the setup/hold checks of the data w.r.t. the clock signal need not be performed when any of the asynchronous pins *set* or *reset* is active. Not having the constraint conditions leads to pessimistic model behavior. Such conditions are automatically generated by this module in the flow. Again, the expertise of the model developers is built into this module.

```

# In the timing file, the constraint condition description begins
# with the keyword CONSTRAINTS and ends with
# ENDCONSTRAINTS. Declaration of interface section
# (i.e., NAME, INORDER, IOORDER, and OUTORDER
# statements) is same as that present in the equation format.

CONSTRAINTS
# Setup times:
TSU on <signame> [opt_edge] before <signame>
      [opt_edge] [when [!] <boolean expression> ],
      forcetoX (<pinlist>);

# Hold times:
TH on <signame> [opt_edge] after <signame>
      [opt_edge] [when [!] <boolean expression> ],
      forcetoX (<pinlist>);

# Low pulse width:
MPWL on <signame> [when [!] <boolean
      expression> ], forcetoX (<pinlist>);

# High pulse width:
MPWH on <signame> [when [!] <boolean
      expression> ], forcetoX (<pinlist>);

ENDCONSTRAINTS;

TSU, TH, MPWL, MPWH, on, before, after, when and forcetoX
are keywords. <signame> is one of the signals described in the
interface section. opt_edge is optional and can be:
(01) for Low to High
(10) for High to Low
(0) for Anything to Low
(1) for Anything to High
(?0) for Ambiguous to Low
(?1) for Ambiguous to High
(1?) for High to Ambiguous
(0?) for Low to Ambiguous.

<boolean expression> is optional and defines the condition
under which the constraint check should be made.
<pinlist> is a comma separated list of output pins that need to be
forced to X (unknown logic state) if the constraint is violated.
If not given, no action is taken by the model when constraints are
violated.

```

Fig. 7. Timing File Format.

2.8: Timing file reader

Although the *constraint condition generator* can generate constraint conditions for most of the cells, it may not be able to handle very complex cells like variable delay elements, etc. In such cases, the constraint conditions can be optionally given through an input timing file. The *timing file reader* reads the timing file and updates the timing database with the constraint condition

information present in the file. The format of the input timing file to specify various constraint conditions is summarized in Fig. 7. The set of output ports that should be set to unknown logic value on a particular constraint violation can also be specified through this file.

2.9: Timing update

The function of this module is to read the timing information such as pin-to-pin delays, constraint values, etc. from the centralized timing database and pass it to the *VHDL model writer*.

2.10: VHDL model writer

This module translates the optimized netlist of the cell to a VHDL model. Depending on the options given to VMS, this module can generate VITAL Level-0 compliant behavioral models, or VITAL Level-1 compliant models which use primitives and truth/state tables recommended by VITAL. While the *Functional* block in the model is completely written out by this module, all the relevant information to write out *InputPortDelay*, *PathDelay* and *TimingCheck* blocks is obtained from the *timing update* module. A modification in the modeling style only necessitates changes to this module, leaving the other part of the flow intact.

3: Results

An example functional specification for a D-flipflop with active-low *set* and *reset*, and the corresponding output VHDL model generated by VMS are presented in *Appendix*. The example also shows a rule that is used by the netlist recognition & replacement algorithm to get the desired netlist. The *entity* part of the model consists of generic parameters which are place holders for input port delays, various path delays, and constraints. It also specifies the port interface for the cell. The *architecture* part of the model consists of four blocks: *input_port_delay* block specifying the input port delays, *functionality* block specifying the functionality of the cell in terms of boolean primitives and concurrent procedure calls to the truth/state tables, *path_delay* block specifying the pin-to-pin delay for each of the outputs, and *timing_check* block specifying various setup, hold, and pulse width constraint checks on the input signals of the flipflop.

The tool can handle any cell whose functionality can be described using the extended equation format. For example, VHDL models for *scan latches*, *mixed-flip-flops*, *registers*, etc. can be generated by VMS. The accu-

racy of the models generated by the system is as good as manually created models.

4: Future work and conclusions

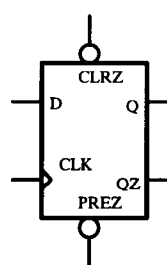
The need for the reduced cycle time for generation of VITAL-compliant ASIC simulation libraries is the driving factor for the development of VMS. As the VITAL standards are still under revision, modification of the *VHDL model writer* module is an absolute necessity in the near future. Once the VITAL standards are frozen, the VMS system becomes stable. The *netlist recognition and replacement rules*, *timing update*, and *VHDL model writer* are crucial part of the VMS flow, whose enhancement/modification gives the most optimized VHDL model in the required modeling style.

5: References

- [1] *IEEE Standard VHDL Language Reference Manual*, IEEE, NY, 1988.
- [2] *VITAL (VHDL Initiative Toward ASIC Libraries) Model Development Specification*, Dec. 1993.
- [3] R. K. Brayton et al., *MIS: Multiple-Level Logic Optimization System*, IEEE Transactions on CAD, Vol. CAD-6, No. 6, Nov. 1987, pp. 1062–1081.

Appendix

Example: A D-flipflop with active-low set and reset.



Functional specification:

```

NAME = DTB10;
INORDER = CLK CLRZ D PREZ;
OUTORDER = Q QZ;
EQUATION
  Q = FLIPFLOP(
    DATA = D,
    CLOCK = CLK,
    SET = !PREZ,
    RESET = !CLRZ,
    SET_RESET_ACTIVE = 0,
    COMPLEMENT = QZ
  );
ENDEQUATION;

```

Rule used by the netlist recognition & replacement algorithm to realize FLIPFLOP as Master-Slave FF:

```

VMS_DFF_COMP1 ; data clk pre clr : out outz ::
(
  VMS_DFF ; data clk tmpPre tmpClr : out outz :: ;
  VMS_NOT ; pre : tmpPre :: ;
  VMS_NOT ; clr : tmpClr :: ;
) :=
(
  VMS_NOT ; clk : tmpclk :: ;
  VMS_DLAUDP ; data tmpclk clr pre : tmp1 :: ;
  VMS_DLAUDP ; tmp1 clk clr pre : out :: ;
  VMS_DFFQZUDP ; clr pre out : outz :: ;
);

```

VITAL-compliant VHDL model generated by VMS:

```

library ieee;
use ieee.Std_logic_1164.all;
library work;
use work.VITAL_PRIMITIVES.all;
use work.VITAL_TABLES.all;
use work.VITAL_Timing.all;
use work.VITAL_Level1.all;

ENTITY DTB10 IS
  GENERIC(
    tiph_CLK : Time := 0.0 ns;
    tiph_CLK : Time := 0.0 ns;
    tiph_CLRZ : Time := 0.0 ns;
    tiph_CLRZ : Time := 0.0 ns;
    tiph_D : Time := 0.0 ns;
    tiph_D : Time := 0.0 ns;
    tiph_PREZ : Time := 0.0 ns;
    tiph_PREZ : Time := 0.0 ns;
    tiph_CLK_Q : Time := 0.0 ns;
    tiph_CLK_Q : Time := 0.0 ns;
    tiph_CLK_QZ : Time := 0.0 ns;
    tiph_CLK_QZ : Time := 0.0 ns;
    tiph_PREZ_Q : Time := 0.0 ns;
    tiph_PREZ_Q : Time := 0.0 ns;
    tiph_PREZ_QZ : Time := 0.0 ns;
    tiph_PREZ_QZ : Time := 0.0 ns;
    tiph_CLRZ_Q : Time := 0.0 ns;
    tiph_CLRZ_Q : Time := 0.0 ns;
    tiph_CLRZ_QZ : Time := 0.0 ns;
    tiph_CLRZ_QZ : Time := 0.0 ns;
    tsetup_D_CLK : Time := 1.35 ns;
    thold_CLK_D : Time := 0.40 ns;
    tsetup_PREZ_CLK : Time := 0.74 ns;
    thold_CLK_PREZ : Time := 0.72 ns;
    tsetup_CLRZ_CLK : Time := 0.37 ns;
    thold_CLK_CLRZ : Time := 1.3 ns;
    tpwh_CLK : Time := 0.93 ns;
    tpwl_CLK : Time := 0.93 ns;
    tpwl_PREZ : Time := 0.99 ns;
    tpwl_CLRZ : Time := 0.89 ns;
    TimingChecksOn : Boolean := TRUE;
    XGenerationOn : Boolean := TRUE;
    InstancePath : string := ""
  );
  PORT(
    CLK : IN std_logic := 'U';
    CLRZ : IN std_logic := 'U';
    D : IN std_logic := 'U';
    PREZ : IN std_logic := 'U';
    Q : OUT std_logic;
    QZ : OUT std_logic
  );
END DTB10;

```

ARCHITECTURE LOGIC OF DTB10 IS

```

attribute VITAL_LEVEL1 of LOGIC : architecture is TRUE;
signal CLK_ipd : std_logic;
signal CLRZ_ipd : std_logic;
signal D_ipd : std_logic;
signal PREZ_ipd : std_logic;
signal Q_zd, Q_d : std_logic;
signal QZ_zd, QZ_d : std_logic;

```

```

CONSTANT DLAUDPTable : VITALStateTableType(0 to 33, 0 to 5) :=
( ('?', '?', 'N', '?', '?', '0'),

```

```

(**, '?', '0', '?', '?', '0'),
('?', '**, '0', '?', '?', '0'),
('?', '?', '0', '*', '?', '0'),
('?', '?', '1', 'N', '?', '1'),
(**, '?', '1', '0', '?', '1'),
('?', '**, '1', '0', '?', '1'),
('?', '?', 'P', '0', '?', '1'),
('?', 'N', '1', '1', '?', '$'),
('?', '0', 'P', '1', '?', '$'),
('?', '0', '1', 'P', '?', '$'),
(**, '0', '1', '1', '?', '$'),
('N', '1', '?', '1', '?', '0'),
('0', 'P', '?', '1', '?', '0'),
('0', '1', '?', 'P', '?', '0'),
('0', '1', '*', '1', '?', '0'),
('P', '1', '1', '?', '?', '1'),
('1', 'P', '1', '?', '?', '1'),
('1', '1', 'P', '?', '?', '1'),
('1', '1', '1', '*', '?', '1'),
('P', '?', '1', '?', '1', '1'),
('1', '?', 'P', '?', '1', '1'),
('1', '?', '1', '*', '1', '1'),
('1', '*', '1', '?', '1', '1'),
('?', 'N', 'X', '1', '0', '0'),
('?', '0', 'X', 'P', '0', '0'),
(**, '0', 'X', '1', '0', '0'),
('N', '?', '?', '1', '0', '0'),
('0', '?', '?', 'P', '0', '0'),
('0', '*', '?', '1', '0', '0'),
('0', '?', '*', '1', '0', '0'),
('?', 'N', '1', 'X', '1', '1'),
('?', '0', 'P', 'X', '1', '1'),
(**, '0', '1', 'X', '1', '1') );

```

```

CONSTANT DFFQZUDPTable : VITALTruthTableType(0 to 3, 0 to 3) :=

```

```

( ('?', '0', '?', '0'),
('0', '1', '?', '1'),
('?', '1', '0', '1'),
('1', '?', '1', '0') );

```

```
BEGIN
```

```
-- Input Port Delay Block
```

```
INPUT_PORT_DELAY : BLOCK
```

```
BEGIN
```

```
  PropagateInputPortDelay(CLK_ipd,CLK,ExtendToFillDelay((tiph_CLK,tiph_CLK)));
```

```
  PropagateInputPortDelay(CLRZ_ipd,CLRZ,ExtendToFillDelay((tiph_CLRZ,tiph_CLRZ)));
```

```
  PropagateInputPortDelay(D_ipd,D,ExtendToFillDelay((tiph_D,tiph_D)));
```

```
  PropagateInputPortDelay(PREZ_ipd,PREZ,ExtendToFillDelay((tiph_PREZ,tiph_PREZ)));
```

```
END BLOCK;
```

```
-- Zero Delay Functionality Block
```

```
FUNCTIONALITY : BLOCK
```

```
  signal IINVnet1 : std_logic;
```

```
  signal r1,r2,r3 : std_logic_vector(0 to 0);
```

```
BEGIN
```

```
IINVnet1 <= VitalINV(CLK_ipd);
```

```
VitalStateTable (
```

```
  NumInputs => 4,
```

```
  NumOutputs => 1,
```

```

    StateTable => DLAUDPTable,
    Data=>std_logic_vector'(D_ipd,IINV-
net1,CLRZ_ipd,PREZ_ipd),
    Result => r1
);

VitalStateTable (
    NumInputs => 4,
    NumOutputs => 1,
    StateTable => DLAUDPTable,
    Data=>std_logic_vec-
tor'(r1(0),CLK_ipd,CLRZ_ipd,PREZ_ipd),
    Result => r2
);

VitalTruthTable (
    NumInputs => 3,
    NumOutputs => 1,
    TruthTable => DFFQZUDPTable,
    Data => std_logic_vector'(CLRZ_ipd,PREZ_ipd,r2(0)),
    Result => r3
);

Q_zd <= r2(0);
QZ_zd <= r3(0);
END BLOCK;

```

```

PATH_DELAY : BLOCK
BEGIN

```

```

    path_Q: process (Q_zd)
        variable GlitchData_Q : Glitchdatatype;
        variable tpd_CLK_Q : DelayType01;
        variable tpd_PREZ_Q : DelayType01;
        variable tpd_CLRZ_Q : DelayType01;
    begin
        tpd_CLK_Q(tr01) := tplh_CLK_Q;
        tpd_CLK_Q(tr10) := tphl_CLK_Q;
        tpd_PREZ_Q(tr01) := tplh_PREZ_Q;
        tpd_PREZ_Q(tr10) := tphl_PREZ_Q;
        tpd_CLRZ_Q(tr01) := tplh_CLRZ_Q;
        tpd_CLRZ_Q(tr10) := tphl_CLRZ_Q;
    end process;

```

```

    PropagatePathDelay (
        OutSignal => Q_d,
        OutSignalName => "Q",
        OutTemp => Q_zd,
        Paths => (
            0 => (
                InputChangeTime => CLK_ipd'last_event,
                PathDelay => ExtendToFillDelay(tpd_CLK_Q)
            ),
            1 => (
                InputChangeTime => PREZ_ipd'last_event,
                PathDelay => ExtendToFillDelay(tpd_PREZ_Q)
            ),
            2 => (
                InputChangeTime => CLRZ_ipd'last_event,
                PathDelay => ExtendToFillDelay(tpd_CLRZ_Q)
            )
        ),
        GlitchData => GlitchData_Q
    );
end process;

```

```

    path_QZ: process (QZ_zd)
        variable GlitchData_QZ : Glitchdatatype;
        variable tpd_CLK_QZ : DelayType01;
        variable tpd_PREZ_QZ : DelayType01;
        variable tpd_CLRZ_QZ : DelayType01;
    begin
        tpd_CLK_QZ(tr01) := tplh_CLK_QZ;
        tpd_CLK_QZ(tr10) := tphl_CLK_QZ;
        tpd_PREZ_QZ(tr01) := tplh_PREZ_QZ;
        tpd_PREZ_QZ(tr10) := tphl_PREZ_QZ;
        tpd_CLRZ_QZ(tr01) := tplh_CLRZ_QZ;
        tpd_CLRZ_QZ(tr10) := tphl_CLRZ_QZ;

        PropagatePathDelay (
            OutSignal => QZ_d,
            OutSignalName => "QZ",
            OutTemp => QZ_zd,
            Paths => (
                0 => (
                    InputChangeTime => CLK_ipd'last_event,
                    PathDelay => ExtendToFillDelay(tpd_CLK_QZ)
                ),
                1 => (
                    InputChangeTime => PREZ_ipd'last_event,
                    PathDelay => ExtendToFillDelay(tpd_PREZ_QZ)
                ),
                2 => (
                    InputChangeTime => CLRZ_ipd'last_event,
                    PathDelay => ExtendToFillDelay(tpd_CLRZ_QZ)
                )
            ),
            GlitchData => GlitchData_QZ
        );
    end process;

```

```

    Q <= Q_d;
    QZ <= QZ_d;
END BLOCK;

```

```

timchk : IF (TimingChecksOn) GENERATE
    TIMING_CHECK : BLOCK
    BEGIN
        timingCheck_D_CLK:process
        (D_ipd,CLK_ipd,CLRZ_ipd,PREZ_ipd)
            variable violation : Boolean := FALSE;
        begin
            TimingCheck (TestPort => D_ipd,
                TestPortName => "D",
                RefPort => CLK_ipd,
                RefPortName => "CLK",
                t_setup_hi => tsetup_D_CLK,
                t_setup_lo => tsetup_D_CLK,
                t_hold_hi => thold_CLK_D,
                t_hold_lo => thold_CLK_D,
                condition => ((CLK_ipd = '1') AND (CLRZ_ipd /= '0')
                AND (PREZ_ipd = '0')),
                Violation => violation,
                OutSignal(0) => Q_zd,
                OutSignal(1) => QZ_zd,
                XOnCondition => True,
                XOffCondition => (rising_edge(CLK_ipd) OR (CLRZ_ipd
                = '0') OR (PREZ_ipd = '0')),
                HeaderMsg => InstancePath
            );
        end process;
    END IF;

```

```

timingCheck_PREZ_CLK:process
(PREZ_ipd,CLK_ipd,CLRZ_ipd)
variable violation : Boolean := FALSE;
begin
TimingCheck (TestPort => PREZ_ipd,
TestPortName => "PREZ",
RefPort => CLK_ipd,
RefPortName => "CLK",
t_setup_hi => tsetup_PREZ_CLK,
t_setup_lo => 0 ns,
t_hold_hi => 0 ns,
t_hold_lo => thold_CLK_PREZ,
condition => ((CLK_ipd = '1') AND (CLRZ_ipd /= '0')
AND (D_ipd /= '1')),
Violation => violation,
OutSignal(0) => Q_zd,
OutSignal(1) => QZ_zd,
XOnCondition => True,
XOffCondition => (rising_edge(CLK_ipd) OR (CLRZ_ipd
= '0') OR (PREZ_ipd = '0')),
HeaderMsg => InstancePath
);
end process;

timingCheck_CLRZ_CLK:process
(CLRZ_ipd,CLK_ipd,PREZ_ipd)
variable violation : Boolean := FALSE;
begin
TimingCheck (TestPort => CLRZ_ipd,
TestPortName => "CLRZ",
RefPort => CLK_ipd,
RefPortName => "CLK",
t_setup_hi => tsetup_CLRZ_CLK,
t_setup_lo => tsetup_CLRZ_CLK,
t_hold_hi => thold_CLK_CLRZ,
t_hold_lo => thold_CLK_CLRZ,
condition => ((CLK_ipd = '1') AND (PREZ_ipd /= '0')
AND (D_ipd /= '0')),
Violation => violation,
OutSignal(0) => Q_zd,
OutSignal(1) => QZ_zd,
XOnCondition => True,
XOffCondition => (rising_edge(CLK_ipd) OR (CLRZ_ipd
= '0') OR (PREZ_ipd = '0')),
HeaderMsg => InstancePath
);
end process;

pulse_CLK:process(CLRZ_ipd,CLK_ipd,PREZ_ipd)
variable PeriodCheckInfo_CLK : DelayArrayTypeXX(0 to 1)
:= PeriodCheckInfo_Init;
variable violation : Boolean := FALSE;
begin
PeriodCheck (TestPort => CLK_ipd,
TestPortName => "CLK",
pw_hi_min => tpwh_CLK,
pw_lo_min => tpwl_CLK,
info => PeriodCheckInfo_CLK,
condition => ((CLRZ_ipd /= '0') AND (PREZ_ipd /= '0')

```

```

AND (Q_d /= D_ipd)),
Violation => violation,
OutSignal(0) => Q_zd,
OutSignal(1) => QZ_zd,
XOnCondition => True,
XOffCondition => (rising_edge(CLK_ipd) OR (CLRZ_ipd
= '0') OR (PREZ_ipd = '0')),
HeaderMsg => InstancePath
);
end process;

pulse_PREZ:process(PREZ_ipd,CLRZ_ipd,CLK_ipd)
variable PeriodCheckInfo_PREZ: DelayArrayTypeXX(0 to 1)
:= PeriodCheckInfo_Init;
variable violation : Boolean := FALSE;
begin
PeriodCheck (TestPort => PREZ_ipd,
TestPortName => "PREZ",
pw_hi_min => 0 ns,
pw_lo_min => tpwl_PREZ,
info => PeriodCheckInfo_PREZ,
condition => ((CLRZ_ipd /= '0') AND (Q_d /= '1')),
Violation => violation,
OutSignal(0) => Q_zd,
OutSignal(1) => QZ_zd,
XOnCondition => True,
XOffCondition => (rising_edge(CLK_ipd) OR (CLRZ_ipd
= '0') OR (PREZ_ipd = '0')),
HeaderMsg => InstancePath
);
end process;

pulse_CLRZ:process(CLRZ_ipd,PREZ_ipd,CLK_ipd)
variable PeriodCheckInfo_CLRZ: DelayArrayTypeXX(0 to 1)
:= PeriodCheckInfo_Init;
variable violation : Boolean := FALSE;
begin
PeriodCheck (TestPort => CLRZ_ipd,
TestPortName => "CLRZ",
pw_hi_min => 0 ns,
pw_lo_min => tpwl_CLRZ,
info => PeriodCheckInfo_CLRZ,
condition => ((PREZ_ipd /= '0') AND (Q_d /= '0')),
Violation => violation,
OutSignal(0) => Q_zd,
OutSignal(1) => QZ_zd,
XOnCondition => True,
XOffCondition => (rising_edge(CLK_ipd) OR (CLRZ_ipd
= '0') OR (PREZ_ipd = '0')),
HeaderMsg => InstancePath
);
end process;
END BLOCK;
END GENERATE;

END LOGIC;

```