1.0 Appendix B: VITAL_PRIMITIVES package

1.1 Purpose:

The purpose of the primitives package is to define a standard set of acceleratable routines to facilitate the development of ASIC library models. By defining a standard set of primitives the following objectives are achieved:

- Reduce the effort required in developing a model library.
- Support fast simulation by defining a base set of functions and models which can be optimized / accelerated by simulation vendors.
- Support a standardized modeling style through usage of a common set of modelling tools (functions, procedures,...).

1.2 Primitive Set

The Primitive package contains the following set of primitives:

VitalAND	VitalAND2	VitalAND3	VitalAND4
VitalOR	VitalOR2	VitalOR3	VitalOR4
VitalXOR	VitalXOR2	VitalXOR3	VitalXOR4
VitalNAND	VitalNAND2	VitalNAND3	VitalNAND4
VitalNOR	VitalNOR2	VitalNOR3	VitalNOR4
VitalXNOR	VitalXNOR2	VitalXNOR3	VitalXNOR4
VitalBUF	VitalBufIf0	VitalBufIf1	
VitalINV	VitalInvIf0	VitalInvIf1	
VitalMux	VitalMux2	VitalMux3	VitalMux4
VitalDecoder	VitalDecoder2	VitalDecoder4	VitalDecoder8
VitalIDENT	VitalTruthTable	VitalStateTable	

Sequential (or clocked) primitives are not included in that: a) sequential cell models would be best supported by Tables, b) the variations in sequential cells from one cell library to another would either require a large set of sequential ‘primitives’ or arbitrarily favor particular behaviors.

Each of the primitives is provided both as a Function and as a Procedure. The primitive functions are provided to support behavioral modelling styles. The primitive procedures are provided to support structural modelling styles.

1.3 Primitive Functions (Behavioral Style)

The primitive functions are provided to support Behavioral modelling styles. These functions compute the defined function and return immediately (that is, the primitive functions do not contain a WAIT statement). Primitive functions may be referenced from either sequential or concurrent statements.

All parameters of the primitive functions are CONSTANTs of mode IN. This allows the usage of Signal, Variable, Constant or expression values as input values.

All the primitive functions return TYPE std_ulogic or std_logic_vector. By default, only values ‘U’, ‘X’, ‘0’, ‘1’, ‘Z’ are returned by primitive functions. These values may be mapped to user specified values (strengths) using the optional ‘ResultMap’ parameter (See “Establishing Output Strengths” on page 59.). All input parameters (other than ‘ResultMap’) of the primitive functions are of TYPE std_ulogic or std_-logic_vector.

Examples of primitive function usage (Acceleratable Behavioral Style):

```
PROCESS (a, b, c )
    VARIABLE AND1_Out, AND2_Out, OR_Out: UX01;
    VARIABLE GlitchData : GlitchDataType;
BEGIN
    AND1_Out  := VitalAND2 ( a, b );
    AND2_Out  := VitalAND2 ( c, d );
    OR_Out    := VitalOR2  ( AND1_Out, AND2_Out);
    VitalGlitchOnEvent ( q, "q", GlitchData, OR_Out, delay, Xonly );
END PROCESS;
```

1.4 Standardization Issues

a. The VITAL_Primitives package shall be compiled into the VITAL library.

b. The standard for this package consists of (a) the declarations of the VITAL_Primitives package and (b) the semantics of the VHDL code provided in the VITAL_Primitives package body, (c) the text of this document,. In the event of a conflict or misunderstanding of interpretations, the order of precendence just described shall prevail.

1.5 Primitive Procedures (Concurrent Style)

The procedure primitives are provided to support Structural modelling styles. These procedures execute in a manner similar to that of a separate process. The procedures internally WAIT for an event on the input SIGNALs, compute the new result, schedule transactions on the output SIGNALs, and WAIT for further input events. A VITAL primitive procedure should only be called as a concurrent procedure call.

The formal parameters of the procedure primitives are either SIGNALs or CONSTANTs. All the functional (logic) input or output parameters of the procedures are SIGNALs. All other parameters are CONSTANT.

Examples of primitive concurrent procedure usage (Distributed Delay Style):

```
VitalAND2 (AND1_Out, a, b);
VitalAND2 (AND2_Out, c, d);
VitalOR2  (Q, AND1_Out, AND2_Out, delay, delay);
```

The procedure primitives are parameterized for:

- separate path delays from each input SIGNAL. (See “Propagation Delay Selection” on page 57.). All path delays default to 0 ns.
- reporting of Glitches and X generation for glitches (See “VitalGlitchOnEvent Handles Glitch Propagation on Events” on page 47.)
- Output strength mapping (See “Establishing Output Strengths” on page 59.)

Note: The CONSTANT option parameters are evaluated on the initial call to the procedure. This occurs after elaboration, and so may facilitate backannotation.

In the case where all path delays are 0 ns, no glitch detection or reporting will occur.

1.5.1 Propagation Delay Selection

The following section describes the propagation delay selection algorithm used by the VITAL primitive procedures. The propagation delay selection algorithm determines the delay value used in scheduling transactions on the output SIGNAL(s).

The delay selection is based on the following general selection criteria:

- If the new output value is dependent on multiple input VALUES, the delay selected is the maximum of the delays from the dependent inputs.
- If the new output value is determined by either of two (any of multiple) input VALUES, the delay selected is the minimum of the delays from these inputs.

For example, in order for the output of a NAND primitive to be ‘0’, all the inputs must be ‘1’. The ‘0’ will be scheduled after the maximum delay from each of the inputs. However, the output of a NAND primitive will be ‘1’ if any of the inputs is ‘0’. In this case, the ‘1’ will be scheduled after the minimum delay from any of the inputs.

The delay selection has two parts:

- maintenance of separate output times from each input signal
- selection of the appropriate output time (and conversion to delay value) based on the above selection criteria.

Maintenance of output time values.

For each input a set of (absolute) output time values is maintained. These time values are: Ti0 - the time the output will change based on a falling input, Ti1 -the time the output will change based on a rising input, Tix - the time the output will change based on the input changing to unknown. These time values are computed each time the input changes. An input is considered to be falling for a change from any value to either ‘0’ or ‘L’. A signal is considered rising for a change from any value to ‘1’ or ‘H’. All other input changes are changes to unknown.

Since the propagation delay parameters (tpd_A_Q) are indexed by output value, and the calculation of output times is based on the input value (change), the calculation must take into account whether the logic path from the input to the output is an inverting or non-inverting path. If the logic path may be either inverting or non-inverting (as is the case for XOR inputs, or select input on a MUX), a set of output times for both a non-inverting and an inverting path are maintained.

Lastly, in order to simplify the delay selection, the Ti0 time value (for a particular input) is set to TIME’HIGH whenever the input is not ‘0’ or ‘L’ and the Ti1 time value is set to TIME’HIGH whenever the input is not ‘1’ or ‘H’.

Selection of Output Time (and corresponding delay)

The delay selection is dependent on the function of the primitive.

AND (NAND) primitives:‘1’ (‘0’) scheduled at maximum of Ti1 times
‘0’ (‘1’) scheduled at minimum of Ti0 times

OR (NOR) primitives:‘1’ (‘0’) scheduled at minimum of Ti1 times
‘0’ (‘1’) scheduled at maximum of Ti0 times

XOR(XNOR) primitives:‘1’ (‘0’) scheduled at max(min(Ti0 times), min(Ti1 times))
‘0’ (‘1’) scheduled at min(max(Ti0 times), max(Ti1 times))

Note: XOR delay selection is same as (A AND B) NOR (A NOR B)

MUX2 primitives: delay selection is (d1 AND dsel) OR (d0 AND (NOT dsel))

MUX primitives delay built on MUX2

DECODER2 primitive result(1) scheduled as: enable AND data
result(0) scheduled as: enable AND NOT data

DECODER primitives built up as using AND, OR, DECODER2

Note: Since the BUF, INV, IDENT primitives have a single input, the output can be scheduled directly from the input event without the need to select from multiple delay paths.

1.5.2 Glitch Detection scheme

GlitchDetect is a deferred constant defined in the VITAL_Primitives package. Its' value is set as follows: GlitchDetect := OnEvent; Therefore for this version v2.2b of the specification, the VITAL primitives will only operate with OnEvent Glitch handling as defined in the VITAL_Timing package and specification.

PrimGlitchMode is also a deferred constant defined in the VITAL_Primitives package. Its' value is set as follows: PrimGlitchMode := XOnly; Therefore for this version v2.2b of the specification, the VITAL primitives will only operate with Xgeneration only and issue no corresponding messages as defined in the VITAL_Timing package and specification.

In addition to the glitch detection on the output assignments performed by GlitchOnEvent, the primitives detect glitch conditions caused by (simultaneous) changes on multiple inputs. These additional glitch conditions are passed on to the VitalGlitchOnEvent procedure using the 'GlitchDelay' parameter on that procedure.

1.5.3 Establishing Output Strengths

Each of the primitives is capable of producing an output value from the set of values { 'U', 'X', '0', '1', 'Z'}. Yet certain technologies (e.g. ECL) may require a more expanded set of logic strengths to be provided. In response to this requirement, a provision is made to allow rapid conversion of any of the five output values above to any other desired output value through the use of a simple table lookup.

Using the types below, strength conversion can be accomplished by creating a CONSTANT of your choosing similar to the Default* constants below and substituting your own set of table lookup values.

TYPE ResultMapType is ARRAY (UX01) of std_ulogic;

TYPE ResultZMapType is ARRAY (UX01Z) of std_ulogic;

CONSTANT DefaultResultMap : ResultMapType := ('U','X','0','1');

CONSTANT DefaultResultZMap : ResultZMapType := ('U','X','0','1','Z');

Example:

CONSTANT DefaultECLMap : ResultMapType := ('U', 'X', '0', 'H');

In this case, '1' is mapped to 'H' on the output of the primitive which utilizes this constant as the ResultMap actual parameter.

N-input Logical Gates . . . . N-input Reduction Gates

PURPOSE:

Perform logical reduction on an n-bit wide input port

DECLARATION:

The following declarations are used for behavioral or concurrent modeling where timing is handled independently from functionality.

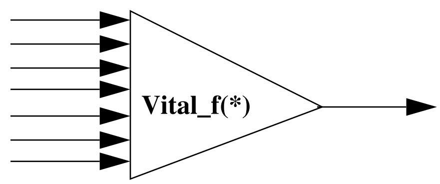
```
FUNCTION VitalAND      ( CONSTANT data       : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;

FUNCTION VitalOR       ( CONSTANT data       : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;

FUNCTION VitalXOR      ( CONSTANT data       : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;

FUNCTION VitalNAND     ( CONSTANT data       : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;

FUNCTION VitalNOR      ( CONSTANT data       : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;

FUNCTION VitalXNOR     ( CONSTANT data       : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;
```

Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalAND   (  SIGNAL             q      : OUT std_ulogic;
                        SIGNAL             data   :  IN std_logic_vector;
                        CONSTANT tpd_data_q       :  IN DelayArrayType01 ;
                        CONSTANT ResultMap        : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalOR    (  SIGNAL             q      : OUT std_ulogic;
                        SIGNAL             data   :  IN std_logic_vector;
                        CONSTANT tpd_data_q       :  IN DelayArrayType01 ;
                        CONSTANT ResultMap        : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalXOR   (  SIGNAL             q      : OUT std_ulogic;
                        SIGNAL             data   :  IN std_logic_vector;
                        CONSTANT tpd_data_q       :  IN DelayArrayType01 ;
                        CONSTANT ResultMap        : IN ResultMapType := DefaultResultMap) ;
```

```
PROCEDURE VitalNAND   ( SIGNAL            q        : OUT std_logic;
                        SIGNAL            data     :  IN std_logic_vector;
                        CONSTANT tpd_data_q        :  IN DelayArrayType01 ;
                        CONSTANT ResultMap         : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNOR    ( SIGNAL            q        : OUT std_logic;
                        SIGNAL            data     :  IN std_logic_vector;
                        CONSTANT tpd_data_q        :  IN DelayArrayType01 ;
                        CONSTANT ResultMap         : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalXNOR   ( SIGNAL            q        : OUT std_logic;
                        SIGNAL            data     :  IN std_logic_vector;
                        CONSTANT tpd_data_q        :  IN DelayArrayType01 ;
                        CONSTANT ResultMap         : IN ResultMapType := DefaultResultMap) ;
```

DESCRIPTION:



N-input, single output reduction logic.

ASSUMPTIONS / BUILT IN ERROR TRAPS:

none.

2-input Logical Gates

PURPOSE:

Perform logical reduction on an 2-bit wide input port

DECLARATION:

The following declarations are used for behavioral or concurrent modeling where timing is handled independently from functionality.

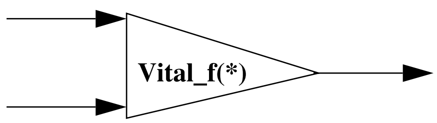
```
FUNCTION VitalAND2      ( CONSTANT a, b : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalOR2       ( CONSTANT a, b : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalXOR2      ( CONSTANT a, b : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalNAND2     ( CONSTANT a, b : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalNOR2      ( CONSTANT a, b : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalXNOR2     ( CONSTANT a, b : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;
```

Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalAND2  (   SIGNAL       q : OUT std_ulogic;
                         SIGNAL     a, b :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalOR2   (   SIGNAL       q : OUT std_ulogic;
                         SIGNAL     a, b :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

```
PROCEDURE VitalXOR2  (      SIGNAL        q : OUT std_ulogic;
                            SIGNAL      a, b :  IN std_ulogic   ;
                            CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                            CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01 ;
                            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNAND2 (      SIGNAL        q : OUT std_ulogic;
                            SIGNAL      a, b :  IN std_ulogic   ;
                            CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                            CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01 ;
                            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNOR2  (      SIGNAL        q : OUT std_ulogic;
                            SIGNAL      a, b :  IN std_ulogic   ;
                            CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                            CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01 ;
                            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalXNOR2 (      SIGNAL        q : OUT std_ulogic;
                            SIGNAL      a, b :  IN std_ulogic   ;
                            CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                            CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01 ;
                            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

DESCRIPTION:



2-input, single output reduction logic.

Each gate maintains a separate tp01,
tp10 propagation delay from each input to the output.

ASSUMPTIONS / BUILT IN ERROR TRAPS:

none.

3-input Logical Gates

PURPOSE:

Perform logical reduction on an 3-bit wide input port

DECLARATION:

The following declarations are used for behavioral or concurrent modeling where timing is handled independently from functionality.

```
FUNCTION VitalAND3      ( CONSTANT a, b, c : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalOR3       ( CONSTANT a, b, c : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalXOR3      ( CONSTANT a, b, c : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalNAND3     ( CONSTANT a, b, c : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalNOR3      ( CONSTANT a, b, c : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalXNOR3     ( CONSTANT a, b, c : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;
```

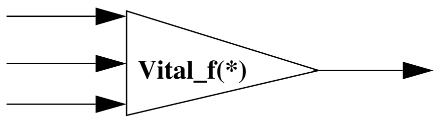
Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalAND3  (   SIGNAL       q : OUT std_ulogic;
                         SIGNAL    a, b, c :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalOR3   (   SIGNAL       q : OUT std_ulogic;
                         SIGNAL    a, b, c :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

```
PROCEDURE VitalXOR3  (   SIGNAL        q : OUT std_ulogic;
                         SIGNAL    a, b, c :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNAND3 (   SIGNAL        q : OUT std_ulogic;
                         SIGNAL    a, b, c :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNOR3  (   SIGNAL        q : OUT std_ulogic;
                         SIGNAL    a, b, c :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalXNOR3 (   SIGNAL        q : OUT std_ulogic;
                         SIGNAL    a, b, c :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

DESCRIPTION:

3-input, single output reduction logic.
Each gate maintains a separate tp01,
tp10 propagation delay from each input to the output.



4-input Logical Gates

PURPOSE:

Perform logical reduction on an 4-bit wide input port

DECLARATION:

The following declarations are used for behavioral or concurrent modeling where timing is handled independently from functionality.

```
FUNCTION VitalAND4      ( CONSTANT a, b, c, d : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalOR4       ( CONSTANT a, b, c, d : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalXOR4      ( CONSTANT a, b, c, d : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalNAND4     ( CONSTANT a, b, c, d : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalNOR4      ( CONSTANT a, b, c, d : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;

FUNCTION VitalXNOR4     ( CONSTANT a, b, c, d : IN std_ulogic ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                        ) RETURN std_ulogic;
```

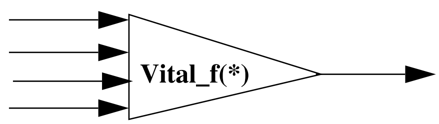
Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalAND4  (   SIGNAL      q : OUT std_ulogic;
                         SIGNAL a, b, c, d :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_d_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalOR4   (   SIGNAL      q : OUT std_ulogic;
                         SIGNAL a, b, c, d :  IN std_ulogic   ;
                         CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01;
                         CONSTANT  tpd_d_q :  IN DelayType01 := DefDelay01 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

```
PROCEDURE VitalXOR4  (    SIGNAL        q : OUT std_ulogic;
                          SIGNAL a, b, c, d :  IN std_ulogic   ;
                          CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_d_q :  IN DelayType01 := DefDelay01 ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNAND4 (    SIGNAL        q : OUT std_ulogic;
                          SIGNAL a, b, c, d :  IN std_ulogic   ;
                          CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_d_q :  IN DelayType01 := DefDelay01 ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalNOR4  (    SIGNAL        q : OUT std_ulogic;
                          SIGNAL a, b, c, d :  IN std_ulogic   ;
                          CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_d_q :  IN DelayType01 := DefDelay01 ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalXNOR4 (    SIGNAL        q : OUT std_ulogic;
                          SIGNAL a, b, c, d :  IN std_ulogic   ;
                          CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_b_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_c_q :  IN DelayType01 := DefDelay01;
                          CONSTANT  tpd_d_q :  IN DelayType01 := DefDelay01 ;
                          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

DESCRIPTION:

4-input, single output reduction logic.
Each gate maintains a separate tp01,
tp10 propagation delay from each input to the output.



ASSUMPTIONS / BUILT IN ERROR TRAPS:

none.

Buffer .... Tri-State and regular Buffers

PURPOSE:

All of the buffers are non-inverting. BUFIF0 and BUFIF1 are three-state buffers.

DECLARATION:

The following declarations are used for behavioral or concurrent modeling where timing is handled independently from functionality.

Non-strength Preserving Buffers:

```
FUNCTION VitalBUF      ( CONSTANT data : IN std_ulogic ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                       ) RETURN std_ulogic;

FUNCTION VitalBUFIF0   ( CONSTANT data, enable : IN std_ulogic ;
                        CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap
                       ) RETURN std_ulogic;

FUNCTION VitalBUFIF1   ( CONSTANT data, enable : IN std_ulogic ;
                        CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap
                       ) RETURN std_ulogic;
```

Strength Preserving Buffer

```
FUNCTION VitalIdent    ( CONSTANT data : IN std_ulogic ;
                        CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap
                       ) RETURN std_ulogic;
```

Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalBUF      (
          SIGNAL       q : OUT std_ulogic;
          SIGNAL       a :  IN std_ulogic   ;
          CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01 ;
          CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalBUFIF0      (
          SIGNAL        q    : OUT std_ulogic;
          SIGNAL        data   :  IN std_ulogic;
          SIGNAL       enable     :  IN std_ulogic;
          CONSTANT   tpd_data_q :  IN DelayType01  := DefDelay01;
          CONSTANT tpd_enable_q :  IN DelayType01Z := DefDelay01Z ;
          CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap) ;

PROCEDURE VitalBUFIF1      (
          SIGNAL          q : OUT std_ulogic;
          SIGNAL        data :  IN std_ulogic;
          SIGNAL       enable :  IN std_ulogic;
          CONSTANT   tpd_data_q :  IN DelayType01  := DefDelay01;
          CONSTANT tpd_enable_q :  IN DelayType01Z := DefDelay01Z ;
          CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap) ;

PROCEDURE VitalIdent      (
          SIGNAL       q : OUT std_ulogic;
          SIGNAL       a :  IN std_ulogic   ;
          CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01Z ;
          CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap) ;
```

DESCRIPTION:

BUF .... standard non-inverting buffer

BUFIF0 .... non-inverting buffer Data passes thru if (Enable = '0')

BUFIF1 .... non-inverting buffer Data passes thru if (Enable = '1')

Ident .... non-inverting buffer which preserves the strength of the input.

ASSUMPTIONS / BUILT-IN ERROR TRAPS:

none

Inverter .. . . Tri-State and regular Inverters

PURPOSE:

All of the buffers are inverting. INVIF0 and INVIF1 are three-state inverters.

DECLARATION:

The following declarations are used for behavioral or concurrent modeling where timing is handled independently from functionality.

```
FUNCTION VitalINV          ( CONSTANT data : IN std_ulogic ;
                             CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                           ) RETURN std_ulogic;

FUNCTION VitalINVIF0       ( CONSTANT data, enable : IN std_ulogic ;
                            CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap
                           ) RETURN std_ulogic;

FUNCTION VitalINVIF1       ( CONSTANT data, enable : IN std_ulogic ;
                            CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap
                           ) RETURN std_ulogic;
```

Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalINV      (
            SIGNAL       q : OUT std_ulogic;
            SIGNAL       a :  IN std_ulogic   ;
            CONSTANT  tpd_a_q :  IN DelayType01 := DefDelay01;
            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalINVIF0      (
            SIGNAL         q    : OUT std_ulogic;
            SIGNAL         data   :  IN std_ulogic;
            SIGNAL       enable     :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN DelayType01   := DefDelay01;
            CONSTANT tpd_enable_q :  IN DelayType01Z := DefDelay01Z;
            CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap) ;

PROCEDURE VitalINVIF1      (
            SIGNAL           q : OUT std_ulogic;
            SIGNAL         data :  IN std_ulogic;
            SIGNAL       enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN DelayType01   := DefDelay01;
            CONSTANT tpd_enable_q :  IN DelayType01Z := DefDelay01Z;
            CONSTANT ResultMap : IN ResultZMapType := DefaultResultZMap) ;
```

DESCRIPTION:

INV   .... standard inverting buffer

INVIF0 .... inverting buffer Data passes thru if (Enable = '0')

INVIF1 .... inverting buffer Data passes thru if (Enable = '1')

Multiplexor . . . . N-bit multiplexors

PURPOSE:

N-bit and commonly used 2,3 and 4-bit multiplexors.

DECLARATION:

```
FUNCTION VitalMUX  (     CONSTANT data    : IN std_logic_vector;
                         CONSTANT dselect : IN std_logic_vector ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                         ) RETURN std_ulogic;

FUNCTION VitalMUX2 (     CONSTANT data1, data0 : IN std_ulogic;
                         CONSTANT dselect     : IN std_ulogic ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                         ) RETURN std_ulogic;

FUNCTION VitalMUX4 (     CONSTANT data    : IN std_logic_vector4;
                         CONSTANT dselect : IN std_logic_vector2 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                         ) RETURN std_ulogic;

FUNCTION VitalMUX8 (     CONSTANT data    : IN std_logic_vector8;
                         CONSTANT dselect : IN std_logic_vector3 ;
                         CONSTANT ResultMap : IN ResultMapType := DefaultResultMap
                         ) RETURN std_ulogic;
```

Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalMUX  (
            SIGNAL        q : OUT std_ulogic;
            SIGNAL     data :  IN std_logic_vector;
            SIGNAL     dsel :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN DelayArrayType01;
            CONSTANT tpd_dsel_q :  IN DelayArrayType01;
            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalMUX2  (
            SIGNAL        q : OUT std_ulogic;
            SIGNAL   d1, d0 :  IN std_ulogic;
            SIGNAL     dsel :  IN std_ulogic;
            CONSTANT   tpd_d1_q :  IN DelayType01 := DefDelay01;
            CONSTANT   tpd_d0_q :  IN DelayType01 := DefDelay01;
            CONSTANT tpd_dsel_q :  IN DelayType01 := DefDelay01;
            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalMUX4  (
            SIGNAL        q : OUT std_ulogic;
            SIGNAL     data :  IN std_logic_vector4;
            SIGNAL     dsel :  IN std_logic_vector2;
            CONSTANT tpd_data_q :  IN DelayArrayType01;
            CONSTANT tpd_dsel_q :  IN DelayArrayType01;
            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

```
PROCEDURE VitalMUX8  (
        SIGNAL       q : OUT std_ulogic;
        SIGNAL       data :  IN std_logic_vector8;
        SIGNAL       dsel :  IN std_logic_vector3;
        CONSTANT tpd_data_q :  IN DelayArrayType01;
        CONSTANT tpd_dsel_q :  IN DelayArrayType01;
        CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

DESCRIPTION:

```
MUX   .......... result := data(dselect)
MUX2  .......... 2-input mux;
        result := data0 when (dselect = '0'),
                  data1 when (dselect = '1'),
                    'X' when (dselect = 'X') and (data0 /= data1)
MUX4  .......... 4-input mux; result := data(dselect)
MUX8  .......... 8-input mux; result := data(dselect)
```

ASSUMPTIONS / BUILT-IN ERROR TRAPS:

none

Decoder .. . . N-bit Decoder

PURPOSE:

N-bit and commonly used 2,3 and 4-bit decoders.

DECLARATION:

```
FUNCTION VitalDECODER   (
                CONSTANT        data   : IN std_logic_vector;
                CONSTANT        enable : IN std_ulogic;
                CONSTANT        ResultMap : IN ResultMapType := DefaultResultMap
) RETURN std_logic_vector;

FUNCTION VitalDECODER2  (
                CONSTANT data   : IN std_ulogic;
                CONSTANT enable : IN std_ulogic;
                CONSTANT        ResultMap : IN ResultMapType := DefaultResultMap
) RETURN std_logic_vecto2r;

FUNCTION VitalDECODER4  (
                CONSTANT data   : IN std_logic_vector2;
                CONSTANT enable : IN std_ulogic;
                CONSTANT        ResultMap : IN ResultMapType := DefaultResultMap
) RETURN std_logic_vector4;

FUNCTION VitalDECODER8  (
                CONSTANT data   : IN std_logic_vector3;
                CONSTANT enable : IN std_ulogic;
                CONSTANT        ResultMap : IN ResultMapType := DefaultResultMap
) RETURN std_logic_vector8;
```

Use the following declarations whenever distributed delay modeling styles are appropriate.

```
PROCEDURE VitalDECODER   (
                SIGNAL           q : OUT std_logic_vector;
                SIGNAL         data :  IN std_logic_vector;
                SIGNAL       enable :  IN std_ulogic;
                CONSTANT   tpd_data_q :  IN DelayArrayType01;
                CONSTANT tpd_enable_q :  IN DelayType01 := DefDelay01;
                CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalDECODER2  (
                SIGNAL           q : OUT std_logic_vector2;
                SIGNAL         data :  IN std_ulogic;
                SIGNAL       enable :  IN std_ulogic;
                CONSTANT   tpd_data_q :  IN DelayType01 := DefDelay01;
                CONSTANT tpd_enable_q :  IN DelayType01 := DefDelay01;
                CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

```
PROCEDURE VitalDECODER4  (
            SIGNAL          q : OUT std_logic_vector4;
            SIGNAL       data :  IN std_logic_vector2;
            SIGNAL     enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN DelayArrayType01;
            CONSTANT tpd_enable_q :  IN DelayType01 := DefDelay01;
            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;

PROCEDURE VitalDECODER8  (
            SIGNAL          q : OUT std_logic_vector8;
            SIGNAL       data :  IN std_logic_vector3;
            SIGNAL     enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN DelayArrayType01;
            CONSTANT tpd_enable_q :  IN DelayType01 := DefDelay01;
            CONSTANT ResultMap : IN ResultMapType := DefaultResultMap) ;
```

DESCRIPTION:

General Algorithm :

- Result(...) := '0' when (enable = '0')
- Result(data) := '1'; all other subelements = '0'
- Result array is decending (n-1 downto 0) order

ASSUMPTIONS / BUILT-IN ERROR TRAPS:

none

2.0 Table Primitives

VITAL supports Truth Table and State Table primitives with powerful edge detection and pessimism reduction mechanisms.

2.1 VITAL_Tables Data Types

The rich symbol set shown below is used to develop high accuracy State Tables. A limited set of values (i.e. ‘X’ through ‘Z’) are used to develop Truth Tables. Users may define their own aliases for their programming ease if desired.

```
TYPE TableSymbolType IS (
'/',-- 0 -> 1                                             (input)
'\',-- 1 -> 0                                             (input)
‘P’,-- Union of ‘/’ and ‘^’                               (input)
‘N’,-- Union of ‘\’ and ‘v’                               (input)
‘r’,-- 0 -> X                                             (input)
‘f’,-- 1 -> X                                             (input)
‘x’,-- Unknown edge
‘U’,-- Uninitialized level
‘X’,-- Unknown level                                      (input, output)
‘0’,-- low level                                          (input, output)
‘1’,-- high level                                         (input, output)
‘-', -- don’t care on any LEVEL (will match any level) (input, output)
‘B’,-- 0 or 1                                             (input)
‘Z’,-- High Impedance                                     (output)
‘p',-- ‘/’ and ‘r’ (any edge from 0)                      (input)
‘n’,-- ‘\’ and ‘f’ (any edge from 1)                      (input)
‘R',-- ‘^’ and ‘p’ (any possible rising edge)             (input)
‘F’,-- ‘v’ and ‘n’ (any possible falling edge)            (input)
‘^’,-- X -> 1                                             (input)
‘v’,-- X -> 0                                             (input)
‘E’,-- ‘v’ and ‘^’ (any edge from X)                      (input)
‘A’,-- 0 -> X, X -> 1                                     (input)
‘D’,-- 1 -> X, X -> 0                                     (input)
‘*’,-- ‘R’ and ‘F’ (any edge)                             (input)
'S' -- if used as an input it represents a steady value(input, output)
     -- if used as an output, the output retains its present value
);
```

```
SubType EdgeType is TableSymbolType range ‘/’ to ‘1’;
SubType VitalTruthTableSymbol is TableSymbolType range ‘X’ to ‘Z’;
```

The following table shows the TableSymbolType elements and the levels and edge transitions they represent.

TABLE 1. Truth and State Table symbol semantics

Symbol	Index	From	To
0		0	0
/	1	0	1
r	2	0	X
\	3	1	0
1		1	1
f	4	1	X
v	5	X	0
^	6	X	1
X		X	X
p	7	1 U 2	
n	8	3 U 4	
E	9	5 U 6	
P	10	1 U 6	
N	11	3 U 5	
A	12	2 U 6	
D	13	4 U 5	
R	14	6 U 7	
F	15	5 U 8	
*	16	14 U 15	
S	17		

Levels

2.2 Standardization

- Locally Static tables may be declared and used as actuals to multiple VitalTruthTable and/or Vital-StateTable calls.

VitalTruthTable .... Truth Table Primitive

PURPOSE

VitalTruthTable is designed to be used as a concurrently or sequentially called truth table.

DECLARATION:

```
FUNCTION VitalTruthTable (
      CONSTANT    TruthTable          : IN      VitalTruthTableType;
      CONSTANT    DataIn              : IN      std_logic_vector
) RETURN std_logic_vector;

FUNCTION VitalTruthTable  (
      CONSTANT    TruthTable          : IN      VitalTruthTableType;
      CONSTANT    DataIn              : IN      std_logic_vector
) RETURN std_logic;

PROCEDURE VitalTruthTable  (
      CONSTANT    TruthTable          : IN      VitalTruthTableType;
      CONSTANT    DataIn              : IN      std_logic_vector;
      SIGNAL      Result              : OUT     std_logic_vector
);

PROCEDURE VitalTruthTable  (
      CONSTANT    TruthTable          : IN      VitalTruthTableType;
      CONSTANT    DataIn              : IN      std_logic_vector;
      SIGNAL      Result              : OUT     std_logic
);
```

DESCRIPTION:

The VitalTruthTable compares the stimulus, called DataIn, with each entry in the TruthTable and returns a result. TruthTables can be made up of any value in the set {‘X’,’0’,’1’,’-’,’B’,’Z’}. However, ‘Z’ may only appear on the OUTPUT portion of the truth table.

Both the functional and procedural forms of the subprogram are provided. The function returns just a value, whereas the procedural form returns a zero delay assignment of that same value to the signal named “Result”.

TruthTable Construction

- **TruthTable** is a two dimensional array (Natural Range, Natural Range) of VitalTruthTableSymbol. The first dimension of the parameter **"TruthTable"** is the number of entries in the table (i.e. the number of input combinations that have a specified output value).
- The second dimension is the total number of inputs and outputs.
- The number of inputs is specified by the length of the parameter **"DataIn"**.
- It is an error if the Length of DataIn is greater than or equal to the size of the second dimension of **"TruthTable"**.

Each row of the truth table is formatted as follows:

Stimulus(0, DataIn'Length - 1 downto 0), Response(0, TruthTable'Length(2) - DataIn'Length - 1 downto 0)
Stimulus(1, DataIn'Length - 1 downto 0), Response(1, TruthTable'Length(2) - DataIn'Length - 1 downto 0)
...
Stimulus(N - 1, DataIn'Length - 1 downto 0), Response(N - 1, TruthTable'Length(2) - DataIn'Length - 1 downto 0)

where N is the total number of entries and is equal to TruthTable'Length(1).

2 to 4 decoder example:
|***** STIMULUS *****| ************* R E S P O N S E **************|

D1	D0	Q3	Q2	Q1	Q0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

TruthTable Algorithm

The subroutines first converts the inputs (DataIn) to the X01 subtype by applying the following mapping:

DataIn	DataInX01 := To_X01(DataIn)
'U'	'X'
'X'	'X'
'0'	'0'
'1'	'1'
'Z'	'X'
'W'	'X'
'L'	'0'
'H'	'1'
'-'	'X'

DataInX01 is then compared to the stimulus portion of the truth table according to the e following matching rules:

TruthTable Stimulus Portion	DataInX01:= To_X01(DataIn)	Result of comparison
‘X’	‘X’	‘X’ only matches with ‘X’
‘0’	‘0’	‘0’ only matches with ‘0’
‘1’	‘1’	‘1’ only matches with ‘1’
‘-’	‘X’, ‘0’, ‘1’	‘-’ matches with any value of DataInX01
‘B’	’0’, ‘1’	‘B’ only matches with ‘0’ or ‘1’

The subroutines search the TruthTable starting at the top (entry 0) and continues searching until a match is found. The TruthTable is compared row by row to the current value of “**DataIn”**. For each row, the comparison begins with the leftmost subelement of DataIn and the corresponding leftmost subelement of the compared row of the TruthTable. If all of the subelements of DataIn match, according to the matching table above, with each of the subelements of a given row of the TruthTable, then a “match” is declared.

If no “match” is found, then VitalTruthTable will return an ‘X’ or a vector of ‘X’s as the case may be.

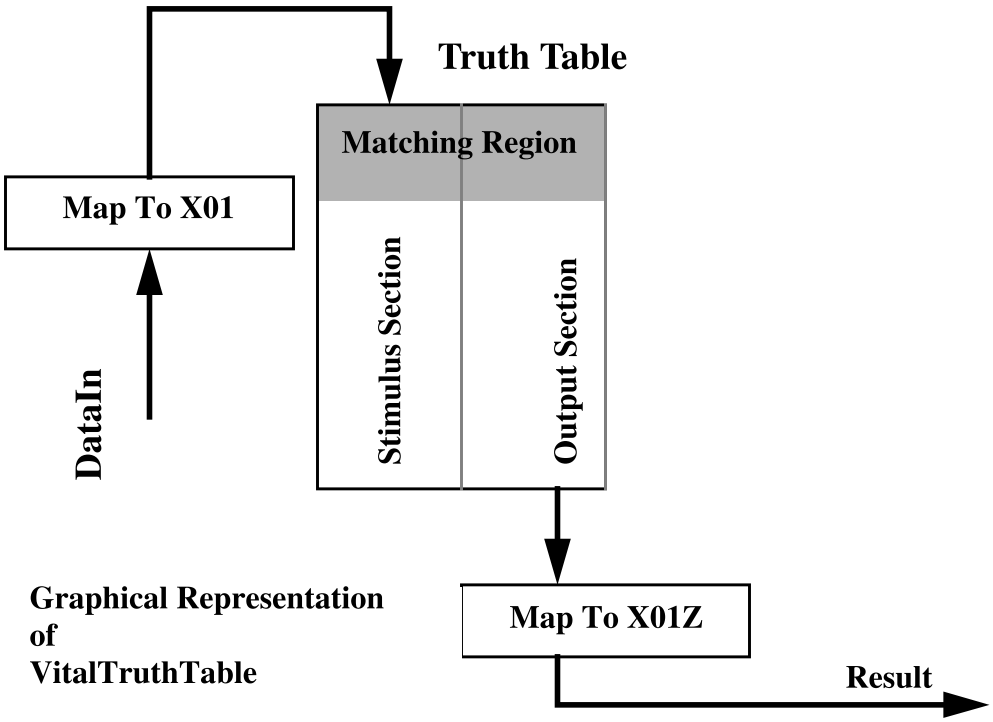
If a “match” is found, the VitalTruthTable will return the response portion of the row corresponding to the “match” subject to the following output mapping:

Once a match is found, or it is determined that no match can be made, the outputs are determined from the output section of the truth table. The number of outputs is equal to TruthTable’Length(2) - DataIn’Length. The outputs are then converted to the X01Z subtype according to the following mapping:

TruthTable Result Subelement	Result
‘X’	‘X’
‘0’	‘0’
‘1’	‘1’
‘Z’	‘Z’
‘-’	‘X’

BUILT IN ERROR TRAPS:

- It is an error to include the character ‘Z’ in the stimulus portion of a TruthTable.
- It is an error to include the character ‘B’ in the response portion of a TruthTable.
- It is an error if (DataIn’length + Result’length) /= TruthTable’Length(2).
- If any of these errors are detected, all ‘X’s are returned for the procedural versions and zero length vectors for the function versions.
- An error is reported if the size of the output vector is either too large or too small.



For the vector procedure, the outputs are placed into the actual associated with the parameter **"Result"**, starting from the right side of both the truth table and the actual associated with **"Result"**, until the actual is filled or there are no more outputs left in the truth table.

If the actual associated with the parameter **"result"** is too long then the remaining portion is filled with 'X's. The vector function behaves in a similar manner but always returns a vector with the range TruthTable'Length(2) - DataIn'Length - 1 downto 0.

APPLICABLE TYPES:

SUBTYPE VitalTruthTableSymbol IS TableSymbolType range 'X' to 'Z';

TYPE VitalTruthTableType IS ARRAY (NATURAL Range <>, NATURAL Range <>)
of VitalTruthTableSymbol;

EXAMPLE:

```
constant xor_tab : VitalTruthTableType(0 to 3, 0 to 2) := (
    ( '0', '1', '1' ),
    ( '1', '0', '1' ),
    ( 'B', 'B', '0' ),
    ( 'X', '0', 'X' )
  );
```

Concurrent Style

```
VitalTruthTable ( TruthTable => xor_tab,
              DataIn    => std_logic_vector'(IN1, IN2),
              Result    => OUT1
             );
```

Behavioral Style

```
xor_process : process (IN1, IN2)

begin
  OUT1 <= VitalTruthTable (TruthTable=>xor_tab,
   DataIn=>  std_logic_vector'(IN1, IN2)
                  );
end process;
```

Explanation: The xor_tab example is complex. In particular, the (B,B,0) entry is tricky, because it looks like it could match input (0,1), for which the output is not 0. But because the table is scanned sequentially, the (0,1,1) and (1,0,1) entries would be seen first, so the only input values that could match (B,B,0) are (0,0) and (1,1), which do indeed generate 0 output. Also, there is a (X,0,X) entry, but no corresponding (0,X,X) entry in the table; this works because failure to match will generate an X output; but that implies that the (X,0,X) entry is not needed either.

VitalStateTable . . . . State Table Primitive

PURPOSE:

VitalStateTable is designed to be used as a concurrently or sequentially called state machine.

DECLARATION:

```
Procedure VitalStateTable (
      CONSTANT   StateTable       : IN     VitalStateTableType; -- User's StateTable data
      CONSTANT   DataIn           : IN     std_logic_vector; -- Inputs
      CONSTANT   NumStates        : IN     Natural, -- number of states
      VARIABLE   Result           : INOUT  std_logic_vector; -- States and Outputs
      VARIABLE   PreviousDataIn   : INOUT  std_logic_vector
);

Procedure VitalStateTable (
      CONSTANT   StateTable       : IN     VitalStateTableType; -- User's StateTable data
      CONSTANT   DataIn           : IN     std_logic_vector; -- Inputs
      VARIABLE   Result           : INOUT  std_logic, -- States and Outputs
      VARIABLE   PreviousDataIn   : INOUT  std_logic_vector
);

Procedure VitalStateTable (
      CONSTANT   StateTable       : IN     VitalStateTableType; -- User's StateTable data
      SIGNAL     DataIn           : IN     std_logic_vector; -- Inputs
      CONSTANT   NumStates        : IN     Natural, -- number of states
      SIGNAL     Result           : INOUT  std_logic_vector  -- Outputs
);

Procedure VitalStateTable (
      CONSTANT   StateTable       : IN     VitalStateTableType; -- User's StateTable data
      SIGNAL     DataIn           : IN     std_logic_vector; -- Inputs
      SIGNAL     Result           : INOUT  std_logic -- Outputs
);
```

DESCRIPTION:

These procedures compute the value of the output (with zero delay) of a synchronous sequential circuit (Moore Machine) based upon the inputs, the present state, and a State table.

a. StateTable: Represents the Stimulus-CurrentState-NextState-OutputValue of a state machine.

b. DataIn: Specifies the data that is input to the sequential circuit.

c. NumStates: Specifies the number of state variables (i.e. flip flops) in the state machine. The left most NumStates elements of the actual associated with the parameter **"Result"** are the state variables.

d. Result: Supplies the present state and returns the next state and the outputs of the circuit.

e. PreviousDataIn:: Used with the sequential procedures only. It keeps track of the previous values of the inputs and the states. It should be declared to be a std_logic_vector of length NumInputs where NumInputs is determined by DataIn'Length. The actual associated with this parameter should be initialized to all 'X's and should never be set again by the user.

StateTable Construction

- **StateTable** is a two dimensional array (Natural Range, Natural Range) of TableSymbolType. The first dimension of the parameter **“StateTable”** is the number of entries in the table (i.e. the number of input and state combinations that have a specified output value).
- The second dimension is the total number of inputs, state variables, and outputs.
- The number of inputs is specified by the length of the parameter **“DataIn”**.
- It is an error if the Length of DataIn is greater than or equal to the size of the second dimension of **“StateTable”**.

Each row of the **StateTable** is formatted as follows:

Input(0,NumInputs - 1), ..., Input(0,0), State(0,NumStates - 1), ..., State(0,0), Output(0,NumOutputs - 1), ..., Output(0,0)
Input(1,NumInputs - 1), ..., Input(1,0), State(1,NumStates - 1), ..., State(1,0), Output(1,NumOutputs - 1), ..., Output(1,0)
...
Input(N,NumInputs - 1), ..., Input(N,0), State(N,NumStates - 1), ..., State(N,0), Output(N,NumOutputs - 1), ..., Output(N,0)

where N + 1 is the total number of entries in the state table and the number of outputs (NumOutputs) is equal to TruthTable’Length(2) - DataIn’Length - NumStates.

The elements State(x, NumStates - 1) downto State(x, 0) represent the present values of the state variables (i.e. the present state). The elements Output(x, NumOutputs - 1) downto Output(x, NumOutputs - NumStates) represent the new values of the state variables (i.e. the next state). The remaining output elements represent the new values of the outputs (if any).

DFF Example:

RESET	SET	D	CLK	Q	QN
‘1’	,‘1’	,‘0’	,‘/’	,‘0’	,‘1’
‘1’	,‘1’	,‘1’	,‘/’	,‘1’	,‘0’
‘1’	,‘1’	,‘X’	,‘/’	,‘X’	,‘X’
‘-’	,‘-’	,‘S’	,‘/’	,‘S’	,‘S’
‘-’	,‘-’	,‘-’	,‘\’	,‘S’	,‘S’
‘1’	,‘1’	,‘-’	,‘D’	,‘S’	,‘S’
‘1’	,‘1’	,‘-’	,‘A’	,‘X’	,‘X’
‘0’	,‘0’	,‘-’	,‘*’	,‘X’	,‘X’
‘0’	,‘1’	,‘-’	,‘*’	,‘0’	,‘1’
‘1’	,‘0’	,‘-’	,‘*’	,‘1’	,‘0’

StateTable Algorithm

The subroutines first converts the inputs (DataIn) to the X01 subtype by applying the following mapping:

DataIn	DataInX01 := To_X01(DataIn)
‘U’	‘X’
‘X’	‘X’
‘0’	‘0’
‘1’	‘1’
‘Z’	‘X’
‘W’	‘X’
‘L’	‘0’
‘H’	‘1’
‘-’	‘X’

DataInX01 is then compared to the stimulus portion of the truth table according to the e following matching rules:

StateTable Stimulus Portion	DataInX01 := To_X01(DataIn)	Result of comparison
‘X’	‘X’	‘X’ only matches with ‘X’
‘0’	‘0’	‘0’ only matches with ‘0’
‘1’	‘1’	‘1’ only matches with ‘1’
‘-’	‘X’, ‘0’, ‘1’	‘-’ matches with any value of DataInX01
‘B’	’0’, ‘1’	‘B’ only matches with ‘0’ or ‘1’

The previous value of DataIn is stored in order to determine if an edge has occurred. Edges are determined by looking up the previous and current value of DataIn on the state table.

The subroutines search the **StateTable** starting at the top (entry 0) and continues searching until either a level or edge match is found, whichever occurs first. Each row in the table is compared element by element to the actual associated with the formal “**DataIn”** and to the states. For each row, the comparison starts at the left most element of both the actual associated with **“DataIn”** and the state table, and proceeds to the right. Once all of the inputs have been compared, the comparison moves to the states. Here the comparison moves from the left most index of the actual associated with the formal **“Result”** (comparing it to State(x, NumStates - 1) in the state table) and proceeds to the right.

The comparison of the entry continues until all of the inputs have been compared or a mismatch is encountered.The search terminates with the first level or edge match. (Note that for this purpose ‘S’ is understood to represent an edge.)

Once a match is found, or it is determined that no match can be made, the new values of the state variables and the outputs are determined from the output section of the state table. All of the elements of TableSymbolType that are marked as outputs are valid values to be used in this portion of the state table.

The state variables and the outputs are then converted to the X01Z subtype according to the following mapping:

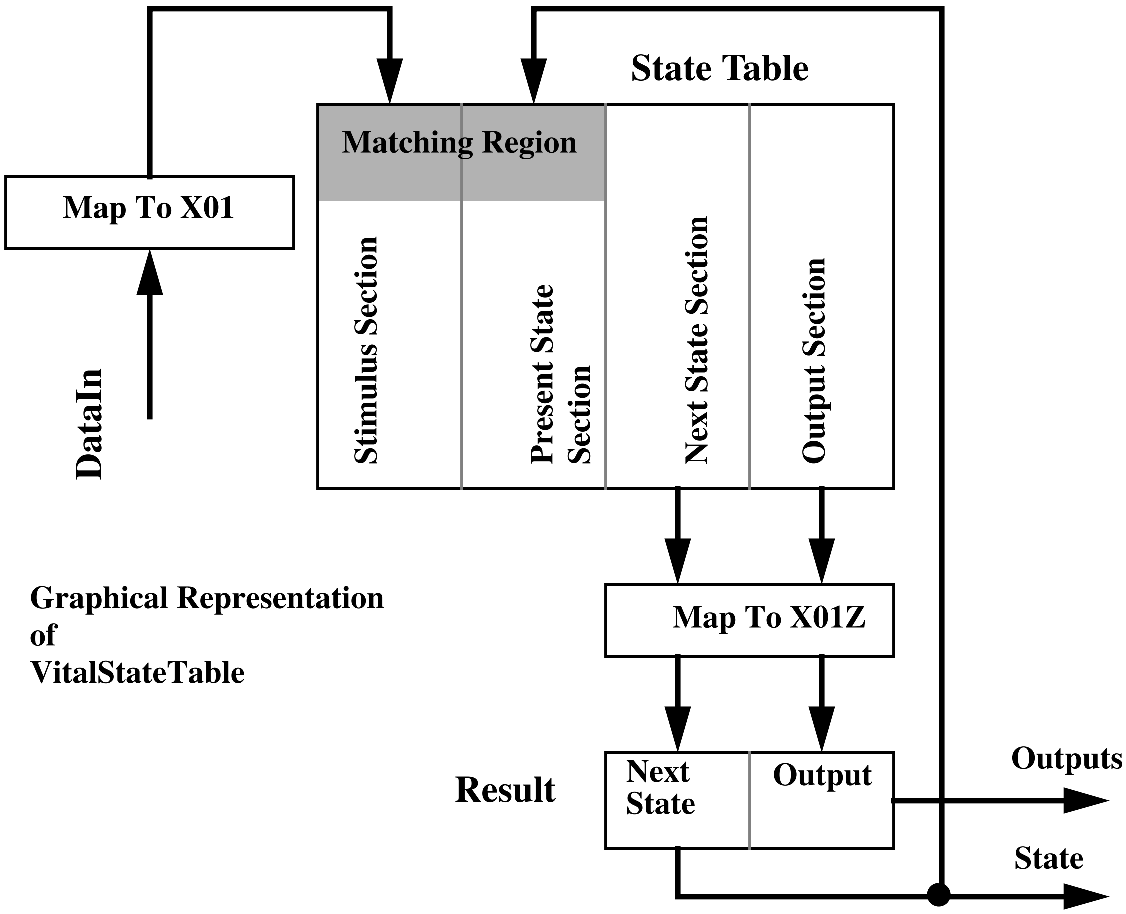
TruthTable Result Subelement	Result
'X'	'X'
'0'	'0'
'1'	'1'
'Z'	'Z'
'-'	'X'
'S'	previous OUTPUT value

If for a certain combination of input values, the state table does not specify the result, then the actual associated with the formal parameter **"Result"** will be assigned all 'X's.

The states and outputs are placed into the actual associated with the parameter **"Result"**, starting from the right side of both the state table and the actual associated with **"Result"**, until the actual is filled or there are no more outputs or states left in the state table. If the actual associated with the parameter **"Result"** is too long then the remaining portion is filled with 'X's.

BUILT IN ERROR TRAPS:

- It is an error to include the character 'Z' in the stimulus portion of a TruthTable.
- It is an error to include the character 'B' in the response portion of a TruthTable.
- An error is reported if the size of the input vector is greater than or equal to StateTable'Length(2).
- If any of these errors are detected, all 'X's are returned for the procedural versions and zero length vectors for the function versions.
- An error is reported if the size of the output vector is either too large or too small.
- It is an error if the user specifies more than one "edge entry" in a row of the state table.



ASSUMPTIONS:

For each state table it is necessary that at least one entry be made with an 'S' for the clock. This will let the procedure VitalStateTable handle the case where the procedure gets activated but the clock did not change. If this entry is not included the result will default to 'X's in this case.

NOTE:

a. NumStates may be assigned the value 0. In this case, the next state is not dependent upon the present state (e.g. a D flip flop).

b. To model Mealy machines it is recommended that VitalStateTable be used to model the state transition and VitalTruthTable be used to model the output function.

APPLICABLE TYPES:

```
TYPE VitalStateTableType IS ARRAY ( NATURAL Range <>, NATURAL Range <> )
                                  of TableSymbolType;
```

EXAMPLE 1: JKSR Flip-Flop

```
VITALBehavior : PROCESS ( J, K, PN, CN, CLK )

  -- Timing Check results
  VARIABLE Tviol_J, Tviol_K, Tviol_PN, Tviol_CN : X01 := '0';
  VARIABLE Pviol_CLK    : X01 := '0';
  VARIABLE PeriodCheckInfo_CLK : DelayArrayTypeXX(0 TO 1) :=
      PeriodCheckInfo_Init;

  -- Functionality results
  VARIABLE violation : X01 := '0';
  VARIABLE  PrevData : std_logic_vector(1 TO 6) := (OTHERS=>'X');
  VARIABLE   Results : std_logic_vector(1 TO 2) := (OTHERS=>'X');
  ALIAS Q_zd    : std_ulogic IS results(1);
  ALIAS QN_zd   : std_ulogic IS results(2);

  -- Output Glitch detection support variables
  VARIABLE Q_GlitchData    : GlitchDataType;
  VARIABLE QN_GlitchData   : GlitchDataType;

  -- State Table Declaration (Possibly moved to a package? )
  CONSTANT L : VitalTableSymbolType := '0';   -- a low value
  CONSTANT H : VitalTableSymbolType := '1';   -- a high value
  CONSTANT x : VitalTableSymbolType := '-';   -- ignore / any value
  CONSTANT U : VitalTableSymbolType := 'X';   -- Unknown value
  CONSTANT B : VitalTableSymbolType := 'B';   -- non-X clock value
  CONSTANT V : VitalTableSymbolType := 'X';   -- Violation
  CONSTANT S : VitalTableSymbolType := 'S';   -- Same value (no output change)
  CONSTANT R : VitalTableSymbolType := '/';   -- Rising clock edge

  CONSTANT JKSRFF_tab : VitalStateTableType (1 TO 8, 1 TO 9) := (
  ------------------------------------------
  -- viol PN  CN CLK  J   K   Q  |  Q' QN'
  ------------------------------------------
    ( V,  x,  x,  x,  x,  x,  x,    U,  U ),   -- Timing violation
    ( x,  L,  x,  x,  x,  x,  x,    H,  L ),   -- Preset (overrides clear)
    ( x,  H,  L,  x,  x,  x,  x,    L,  H ),   -- Clear
    ( x,  H,  H,  R,  L,  x,  L,    L,  H ),
    ( x,  H,  H,  R,  H,  x,  L,    H,  L ),
    ( x,  H,  H,  R,  x,  L,  H,    H,  L ),
    ( x,  H,  H,  R,  x,  H,  H,    L,  H ),
    ( x,  H,  H,  B,  x,  x,  x,    S,  S ) ); -- non-rising / non-X clock
```

```
BEGIN
  --------------------------------------------------------------------------
  -- Timing Check Section
  -- Note: Variation from 2.1e spec, returned violation is of type X01
  --------------------------------------------------------------------------
  IF (TimingCheckON) THEN
      -- Setup J high or low before CLK rise
      -- Hold  J high or low after  CLK rise
    VitalTimingCheck ( J, "J",  CLK, "CLK",
                       tsetup_J_CLK(tr01), tsetup_J_CLK(tr10),
                       thold_J_CLK(tr01), thold_J_CLK(tr10),
                       (CLK='1') AND (To_X01(PN)/='0') AND (To_X01(CN)/='0'),
                       InstancePath & "/JKSRFF", Tviol_J );
      -- Setup K high or low before CLK rise
      -- Hold  K high or low after  CLK rise
    VitalTimingCheck ( K, "K",  CLK, "CLK",
                       tsetup_K_CLK(tr01), tsetup_K_CLK(tr10),
                       thold_K_CLK(tr01), thold_K_CLK(tr10),
                       (CLK='1') AND (To_X01(PN)/='0') AND (To_X01(CN)/='0'),
                       InstancePath & "/JKSRFF", Tviol_K );
      -- Setup PN high before CLK rise
      -- Hold  PN low after   CLK rise
    VitalTimingCheck ( PN, "PN",  CLK, "CLK",
                       tsetup_PN_CLK(tr01), 0 ns              ,
                       0 ns                 , thold_PN_CLK(tr10),
                       (CLK='1') AND (To_X01(CN)/='0'),
                       InstancePath & "/TDN12", Tviol_PN );
      -- Setup CN high before CLK rise
      -- Hold  CN low after   CLK rise
    VitalTimingCheck ( CN, "CN",  CLK, "CLK",
                       tsetup_CN_CLK(tr01), 0 ns              ,
                       0 ns                 , thold_CN_CLK(tr10),
                       (CLK='1') AND (To_X01(PN)/='0'),
                       InstancePath & "/TDN12", Tviol_CN );
    VitalPeriodCheck ( CLK, "CLK",
                       tperiod_CLK, TIME'HIGH,
                       tpw_CLK(tr01), TIME'HIGH, tpw_CLK(tr10), TIME'HIGH,
                       PeriodCheckInfo_CLK, Pviol_CLK,
                       InstancePath & "/JKSRFF", TRUE );
  END IF; -- Timing Check Section
```

```
        --------------------------------------------------------------------------
        -- Functionality Section
        --------------------------------------------------------------------------
        violation := Tviol_J OR Tviol_K OR Pviol_CLK OR Tviol_PN OR Tviol_CN;

        VitalStateTable ( StateTable => JKSRFF_tab,
                              DataIn => ( violation, PN, CN, CLK, J, K ),
                           NumStates => 1,
                              Result => Results,
                      PreviousDataIn => PrevData );

        --------------------------------------------------------------------------
        -- Path Delay Section
        --------------------------------------------------------------------------
        VitalPropagatePathDelay ( Q, "Q", Q_zd,
            Paths(0)=>( PN 'LAST_EVENT,
                   VitalExtendToFillDelay(tpd_PN_Q  ), (To_X01(PN )/='1') ),
            Paths(1)=>( CN 'LAST_EVENT,
                   VitalExtendToFillDelay(tpd_CN_Q  ), (To_X01(CN )/='1') ),
            Paths(2)=>( CLK'LAST_EVENT,
                   VitalExtendToFillDelay(tpd_CLK_Q ),
                   (To_X01(PN ) ='0') AND (To_X01(CN ) ='0') ),
            GlitchData=>Q_GlitchData, GlitchMode=>Xonly, GlitchKind=>OnEvent );

        VitalPropagatePathDelay ( QN, "QN", QN_zd,
            Paths(0)=>( PN 'LAST_EVENT,
                   VitalExtendToFillDelay(tpd_PN_QN ), (To_X01(PN )/='1') ),
            Paths(1)=>( CN 'LAST_EVENT,
                   VitalExtendToFillDelay(tpd_CN_QN ), (To_X01(CN )/='1') ),
            Paths(2)=>( CLK'LAST_EVENT,
                   VitalExtendToFillDelay(tpd_CLK_QN),
                   (To_X01(PN ) ='0') AND (To_X01(CN ) ='0') ),
            GlitchData=>QN_GlitchData, GlitchMode=>Xonly, GlitchKind=>OnEvent );

      END PROCESS;
    END Behavior;
```