

# A COMPARISON BETWEEN VERILOG AND VITAL MODELING IN ASIC LIBRARY TOWARD SIGN-OFF

MAY HUANG  
VLSI Technology, Inc.  
and  
SHU-PARK CHAN  
International Technological University

## ABSTRACT

The two major Hardware Description Languages are Verilog and VHDL. Both languages allow high-level and gate-level descriptions. Verilog has been generally supported by ASIC vendors at the gate-level for sign-off simulation; whereas, VHDL is used mainly for system design at behavioral and RTL levels. With the introduction of VITAL specifications, using VHDL for gate-level simulation has become effective. This paper examines the features of Verilog and VITAL gate models. The structure and character of modeling in the two environments are presented. Furthermore, the different descriptions on modeling for timing violation handling, state table usage, output assignment, edge sensitive delay, back-annotation, and other factors are discussed along with the examples. The strategy is targeted at the sign-off quality of VITAL model generation.

## 1. VITAL INTRODUCTION

VHDL, the only IEEE standard hardware description language, has increased its market very quickly. Many EDA vendors developed VHDL simulators. VHDL shows its advantages as being powerful and flexible for system design. But due to the lack of ASIC libraries and slower gate-level simulation performance, people use VHDL mainly for behavioral simulation, then synthesize or translate the design to another simulation environment to run gate-level sign-off simulation[3]. The biggest impediment to the development of ASIC libraries in VHDL is the lack of model uniformity required for acceleration [2]. VITAL (VHDL Initiative Toward ASIC

Libraries) is designed to solve this key problem. Since VITAL is still under development, the comparison presented in the paper is based on the version, VITAL 2.2B.

VITAL defines a model description in terms of functionality, timing and back annotation. Timing information is introduced via SDF. Model structure and parameters are defined rigidly to ensure that VITAL models are portable; i.e. simulator independent.

VITAL has two levels of model compliance: level 0 and level 1. Level 0 defines name and type conversion for timing information and addresses back annotation. Level 1 defines functions and procedures for timing check, design functional primitives and glitch detection. It also defines the model architecture with coding style guidelines and addresses simulation performance [2].

## 2. CONSTRUCTION OF MODELS

### 2.1 Verilog Model

The construction of a Verilog cell model is fairly straightforward. It generally consists of the following parts:

- \* ports, variables, and registers to be defined
- \* functionality description
- \* specify block
  - timing parameters to be specified
  - path delays
  - timing checks

## 2.2 VITAL Model

Unlike the Verilog cell model, a VITAL cell model consists of at least two declarations:

### Entity and Architecture.

An entity declaration includes two parts: generic and port. Generic specifies timing and control parameters. Port specifies input and output signals. Since VHDL is a strongly typed language, all signals and parameters must have defined types. VITAL level 0 defines compliance with respect to the entity declaration [1].

A VITAL architecture declaration supports two choices of modeling style: pin-to-pin delay modeling and distributed delay modeling. Pin-to-pin delay modeling is more commonly used. The distributed delay style is based on the structure of the physical model. Focusing on the pin-to-pin delay style, the architecture consists of three parts[1]:

1. Declaration: Attributes, signals, variables, constants, flags, and others are defined.
2. Input Path Delay: VITAL defined procedure, `VitalPropagateWireDelay`, handles wire delay.
3. Behavior Process: It consists of the following three sections:

### (a). Timing Check Section:

The procedure, `VitalTimingCheck`, can be called repeatedly to construct setup and hold checks.

The procedure, `VitalPeriodCheck`, can be called repeatedly to construct pulse width and period checks.

### (b). Functionality section:

Function description can be one or a combination of the three structures:

- VITAL Primitives
- VITAL Truth Table
- VITAL State Table

### (c). Path Delay Section:

The VITAL defined procedure, `VitalPropagatePathDelay`, processes pin-to-pin delays. Each output signal will be driven by a call to `VitalPropagatePathDelay`.

## 3. COMPARISON OF MODELS

The comparison of modeling in Verilog and VITAL is illustrated in three tables followed by a detailed analysis.

TABLE 1. Comparison of the Verilog and VITAL models

	<b>Verilog</b>	<b>VITAL</b>
Cell & Port Names:	Similar	Similar
Data Type:	Verilog HDL value set	Strictly Defined
Case Sensitive:	Yes	No
Control Structure:	Ok	Not allowed in process
Output Assignments:	Directly	Indirectly
Output in StateTable:	One	Multiple
Parameters:	InCap, OutCap, MaxLoad, R_Ramp, F_Ramp, MaxLoadRamp, cell_count, Transistors, Power,	None
Function description:	Primitives Truth Table State Table	Primitives Truth Table State Table

	<b>Verilog</b>	<b>VITAL</b>
<b>Timing Check:</b>	\$setup \$hold \$width \$period \$skew N/A \$recovery \$setuphold \$nochange N/A N/A	tsetup thold tpw tperiod tskew release recovery setup, thold N/A tdevice tpulse
<b>Timing Check Control:</b>	Available	Available
<b>Timing Violation Mesg.:</b>	Available	Available
<b>Violation Highlight:</b>	Flag	Flag
<b>Wire Delay:</b>	None	Each input pin
<b>Pin-to-pin Delay:</b>	min, Typ, max	One choice
<b>Edge-Control Spec.:</b>	Available	Available
<b>Glitch Handle:</b>	Inertial delay is ignored	Glitch free Glitch on Event Glitch on Detect

**TABLE 2. Comparison of symbols Used in State Table**

Case sensitive	Yes	Yes (type Of charactor string)
0	0	00, 10, X0
1	1	11, 01, X1
x	x	XX
?	Iteration of 0, 1, x	N/A
b	Iteration of 0,1	N/A
-	No change	Don't care
(vw)	Value change from v to w	N/A
*	Any value change	Any value change
r	01	0X
f	10	1X
p	01, 0x, x1	01, X1
n	10, 1x, x0	10, 1X
S	N/A	No change

TABLE 3. Some symbols owned by Vital only [2]

/	01
\	10
X	0X, 1X, XX
P	01, 0X
R	01, X1, 0X
F	10, X0, 1X
... ..	

Please note that there are several differences between the symbols used in Verilog and VITAL.

1) The Verilog model uses two characters to describe edge events and VITAL uses only one. For example, the Verilog model uses "?0" to match events "00", "10" and "x0"; VITAL uses "0" to represent the same thing.

2) Characters, "0", "1" and "x" are used to represent the states of a signal in Verilog. In VITAL, "0", "1" and "X" can be used as either signal states or signal transitions.

3) Character "-" represents "no change" of an output signal in Verilog, but in VITAL it represents "don't care", and "no change" is represented by symbol "S".

4) VITAL model uses "X" instead of "x" to represent a state unknown or a transition to unknown.

5) Each Verilog UDP table allows only one output signal; VITAL State Table can specify more than one output signal.

Let us take an example to illustrate the equal ability of Verilog and VITAL at representing a partial state table for a DFF with set and reset signals.

```

Verilog state table (UDP)

primitive UDP_DFF(q, d, cp, set, clr, notify);
    output q;
    input d, cp, set, clr, notify;
    reg q;
    table
// d    cp    set    clr    notif    pre    q
  1     r     1     1     ?       :?     : 1; // d -> q at cp rising edge, set and clr not active.
  0     r     1     1     ?       :?     : 0; // d -> q at cp rising edge, set and clr not active.
... ..
  1     (x1)  1     1     ?       : 1    : 1; // q=1 if d and pre both 1, cp from x to 1.
  1     (0x)  1     1     ?       : 1    : 1; // q=1 if d and pre both 1, cp from 0 to x.
  0     (x1)  1     1     ?       : 0    : 0; // q=0 if d and pre both 0, cp from x to 1.
  0     (0x)  1     1     ?       : 0    : 0; // q=0 if d and pre both 0, cp from 0 to x.
  ?     ?     0     1     ?       :?     : 1; // q=1 if set.
  ?     ?     1     0     ?       :?     : 0; // q=0 if clear.
  ?     ?     0     0     ?       :?     : 0; // q=0 if both set and clr active.
  ?     (?0)  1     1     ?       :?     :-; // q is nochange at falling of cp.
  ?     (1x)  1     1     ?       :?     :-; // q is nochange when cp from 1 to x
... ..
  ?     ?     (?1)  ?     ?       :?     :-; // q is nochange at rising edge of set.
  ?     ?     ?     (?1)  ?       :?     :-; // q is nochange at rising edge of clr.
  ?     ?     ?     ?     *       :?     :x; // output x on any notifier change
    endtable

```

#### VITAL state table

```
CONSTANT UDPDF : VitalStateTableType (1 to 19, 1 to 8) := (  
-- d cp set clr notif pre q qn  
  (('1', '1', '1', '1', '-', '-', '1', '0'),  
   ('0', '1', '1', '1', '-', '-', '0', '1'),  
   ... ..  
-- The following line corresponding two lines in Verilog as "cp" changing from x1 or 0x  
  ('1', 'R', '1', '1', '-', '1', '1', '0'),  
-- The following line corresponding two lines in Verilog as "cp" changing from x1 or 0x  
  ('0', 'R', '1', '1', '-', '0', '0', '1'),  
  ('-', '-', '0', '1', '-', '-', '1', '0'),  
  ('-', '-', '1', '0', '-', '-', '0', '1'),  
  ('-', '-', '0', '0', '-', '-', '0', '0'),  
-- The following line corresponding two lines in Verilog as "cp" falling edge  
  ('-', 'F', '1', '1', '-', '-', 'S', 'S'),  
  ... ..  
  ('-', '-', 'P', '-', '-', '-', 'S', 'S'),  
  ('-', '-', '-', 'P', '-', '-', 'S', 'S'),  
  ('-', '-', '-', '-', 'X', '-', 'X', 'X')));
```

VITAL defines more symbols to represent a state table efficiently. In the above example, "qn" is not simply the inverse of "q", since when clear and set are both active, both "q" and "qn" are low. Furthermore, due to the consideration of unknown handling, "qn" should be represented separately. Verilog UDPs are limited to one output per table, hence a UDP table needs to be called twice to complete the functional description. Comparatively, VITAL presents "q" and "qn" in one table so that the table need be called only once.

## 4. TECHNICAL DISCUSSIONS

There are some interesting points which are discussed as follows:

### 4.1 Strongly Typed Language

Data Type is strictly defined in the VITAL model. For example, if a delay, or a timing constraint is defined as DelayType01 (rising, falling), but an SDF file has six field values for the corresponding delay, then an error will be given by the compiler.

Both VITAL procedures, VitalPropagateWireDelay and VitalPropagatePathDelay, are

required DelayType01Z (01,10, 0Z, Z1, 1Z, Z0). The function, VitalExtendToFillDelay, is used to translate any other type to DelayType01X.

The following example illustrates a matching problem of data type.

```
Function VitalMux4 (  
  CONSTANT data : IN std_logic_vector4;  
  CONSTANT dselect : IN std_logic_vector2;  
  CONSTANT ResultMap : IN ResultMapType :=  
    DefaultResultMap  
) RETURN std_ulogic;
```

If using a function call to VitalMux4 like:

```
z_zd := VitalMux4 (i0_ipd, i1_ipd,  
                  i2_ipd,i3_ipd, s0_ipd, s1_ipd);
```

an error message will be given during the compilation. The correct usage is to identify the subtype of the argument:

```
z_zd := VitalMux4 ((i0_ipd, i1_ipd,  
                  i2_ipd,i3_ipd), (s0_ipd, s1_ipd));
```

### 4.2 Input / Output Assignments

Verilog model defines functionalities among the input and output signals. Sometimes internal variables are used for combinational descrip-

tions. Unlike the Verilog model, VITAL models define an internal signal for each input to hold input wire delay. Also there is no output assignment in the function description part. VITAL models do the output assignment inside procedure `VitalPropagatePathDelay`. An example is given as follows:

Verilog:

```

module fn01 (zn, a1, b1);
  input a1, b1;
  output zn;
  not g1(N2, a1);
  nand g2(zn, N2, b1);
  ...
endmodule

```

VITAL:

```

ENTITY fn01 IS
  ...
  PORT (
    A1  : IN std_logic;
    B1  : IN std_logic;
    ZN  : OUT );
END fn01d2
ARCHITECTURE fn01_arch OF fn01 IS
  Begin
  ...
  SIGNAL A1_ipd  : std_ulogic := 'X';
  SIGNAL B1_ipd  : std_ulogic := 'X';
  ...
  WIRE_DELAY : BLOCK
  BEGIN
    VitalPropagateWireDelay(A1_ipd, A1,
      VitalExtendToFillDelay(tipd_A1));
    VitalPropagateWireDelay(B1_ipd, B1,
      VitalExtendToFillDelay(tipd_B1));
  END BLOCK;
  VITALbehavior : PROCESS(A1_ipd, B1_ipd)
  VARIABLE ZN_zd : std_ulogic;
  ...
  BEGIN
    ZN_zd := VitalNAND2(B1_ipd, Vital-
      INV(A1_ipd));
    VitalPropagatePathDelay(
      OutSignal => ZN,

```

```

OutSignalName => "ZN",
OutTemp      => ZN_zd,

```

```

...
END PROCESS;
END fn01_arch;

```

In the VITAL model, input signals A1 and B1 are assigned to internal signals A1\_ipd and B1\_ipd with wire delays. The output ZN is assigned by internal variable ZN\_ipd inside procedure, `VitalPropagatePathDelay`.

### 4.3 Violation Process

Both Verilog and VITAL use a flag to identify a timing violation. But the definition of the flag may be different and therefore the violation processes may be different. For example, the timing check for setup in Verilog is:

```

$setup (data_event, reference_event, limit,
  notifier);

```

where notifier is the flag for setup violation. Verilog\_XL toggle notifier as:

Notifier Values:

Before violation:	After violation:
x	1
0	1
1	0
z	z

In a UDP table a line used for violation specification may be:

```

// data clk set reset notifier pre out
? ? ? ? * ? x;

```

which means that the output is "x" on any notifier change (a violation happened).

In VITAL the toggle for the flag is different: 0 for no violation, X for violation. If translating the above violation specification to `VitalStateTable` without proper modification, it may provide a wrong result.

```

-- data clk set reset notifier pre out
('-', '-', '-', '-', '*', '-', 'X');

```

If the notifier changes from "X" to "0" (no violation), the output is "X" since the notifier is changed. It is obviously unexpected. To correct the error, we rewrite the violation specification as:

```

-- data clk set reset notifier pre out
('-', '-', '-', '-', 'X', '-', 'X');

```

#### 4.4 Path Polarity Delay

In Verilog modeling, a path polarity delay may be used to describe a pin-to-pin delay. For example, there is a low active latch with set and reset signals. To model the latch, define the delay from input 'd' to output 'q' as follows:

```
if(!en && cdn && sdn)
  (d +=> q) = (1, 2);
```

where a positive polarity delay is used. The equation means that a rise of "d" at the source always causes a rise of "q" at the destination.

A positive polarity delay is defined as follows:

A rise (fall) at a source always causes a rise (fall) at a destination.

Similarly, a negative polarity delay is defined as follows:

A rise (fall) at a source always causes a fall (rise) at a destination.

The technique of path polarity delay is used in Veritime for static timing analysis. The relationship of the pins is specified to avoid non-sense path inspection in timing analysis. For the simulation part, (d +=> q) and (d => q) does not make a difference.

In VITAL modeling, there is a way to represent a conditional delay, but expression is not responsible for producing the "rise of 'd' always causes the rise of 'q' " behavior. The following is an example:

```
VitalPropagatePathDelay(
  OutSignal    => q,
  OutSignalName => "q",
  OutTemp      => q_zd,
  Paths        => (
    0=> (d_ipd'LAST_EVENT, VitalExtendToFill-
      Delay(tpd_d_q_r), To_X01(d_ipd)='1')
    GlitchData  => q_GlitchData,
    GlitchMode  => MessageOnly,
    GlitchKind  => GlitchFree);
```

#### 4.5 Edge Sensitive Delay

An edge sensitive delay is a delay from an input to an output that depends on the state of another input signal.

Use the latch again as an example. To describe the delay from an enable signal "en" to the output "q" in Verilog, we have:

```
if(cdn && sdn)
  (negedge en => (q+:d)) = (0.4, 0.5);
```

which means that the delay uses 0.4 if 'd' is '1'; otherwise it uses 0.5.

The similar delay description in VITAL can be:

```
tpd_en_q_r: DelayTypeXX := 0.4;
tpd_en_q_f: DelayTypeXX := 0.5;
... ..
VitalPropagatePath Delay (q, "q", q_zd,
  path => (0 => (en_ipd'last_event,
    VitalExtendToFillDelay (tpd_en_q_r,
      (cdn_ipd and sdn_ipd and d) = '1' ) ),
    1 => (en_ipd'last_event,
      VitalExtendToFillDelay (tpd_en_q_f),
        (cdn_ipd and sdn_ipd and not(d) ) = '1' ) ),
    GlitchData => q_GlitchData,
    GlitchMode => xonly,
    GlitchKind => Onevent);
```

Like path polarity delay, the edge sensitive delay is required by Veritime for timing analysis, and they do not contribute anything in simulation mode.

## 5. CONCLUSION

VITAL compliant gate-level models are generated to support ASIC design sign-off simulation. VITAL pledges the portability of a gate-level library among different simulators. Both Verilog and VITAL support technology-independent libraries, and possibly use an identical SDF file to introduce timing information into the models. Verilog provides the option of a technology-dependent description by the "specify" block. VITAL provides more primitives than Verilog to facilitate functionality descriptions, also to accelerate the performance. VITAL provides more glitch detections and flexible glitch processes. Compared with Verilog, VITAL models satisfy the functionality accuracy, known and unknown handling, timing check, on intra- and inter-cell delays, and therefore satisfy the sign-off requirements.

Verilog models contain information for not only simulation but also timing analysis (Veritime) and fault analysis (Verifault). VITAL models currently only focus on simulation purpose

but they possess the potential ability to support timing and fault analysis which may become part of the modeling standard in the future.

#### ACKNOWLEDGMENTS

The authors would like to acknowledge the following people for their significant assistance in understanding VITAL and VITAL models generation to meet sign-off requirements.

Michael Kohl, Paul Wiscombe, Rob Anderson, Pong Hsu and Pravin Bhusari of VLSI Technology Inc.

Jose De Castro, Dennis Brophy and Kent Moffat of Mentor Graphics Corporation.

Victor Berman, Sanjay Nayak and Ravi TR of Cadence.

Steven Schulz of Texas Instruments.

Ray Ryan, Yvonne Ryan of Ryan & Ryan.

#### REFENENCES

- [1]. VITAL Model Development Specification Version 2.2B, April, 1994.
- [2]. Ryan & Ryan, "Understanding VITAL 2.2B", Fall, 1994.
- [3]. Susan Loffredo, "Product development", Electronic Business Buyer, June, 1994, pp. 65-67.