

Practical Experiences of VITAL Modeling

Anne Margaret Crow, VEDA Design Automation

Abstract

During Autumn 1994, engineers at VEDA Design Automation created Level 1 VITAL compliant versions of LSI Logic's LCA300k and LCB300k libraries, VLSI Technology's VSC670 library and Xilinx's 2000, 3000 and 4000 libraries. This paper aims to communicate some of the 'do's and don'ts' of VITAL modeling we have encountered.

We begin by discussing what makes a model Level 1 compliant, focusing on two tricky topics, writing state-tables and resolving ambiguities in version 2.2b of the standard. The discussion of state tables leads onto one of the more controversial aspects of VITAL, whether or not to allow multiple processes in models. We then consider how to write memory and PCB models, which are currently not covered by the VITAL standard. Finally we look at the software VEDA used to automate the task of model generation.

Level 1 VITAL Compliance

There are four elements to VITAL, a package of primitives, a package of procedures for calculating timing violations, a precise specification of how to use generics so that they map onto SDF constructs, and documentation explaining how to use the other three elements.

Diagram 1 shows how SDF mapping onto generics works. An external delay calculator takes the load capacitances and track capacitances and calculates the timing delays, using Cadence's Standard Delay Format (SDF) for the output. Each SDF construct is mapped onto a VHDL generic construct. Here the SDF

description of a minimum setup time of 1700ps for datapin D with respect to a rising edge on clock pin CLK is mapped onto a generic referring to pins D and CLK. During elaboration, the generic changes from its default value of 1670, to 1700. This value is then used as input to the VitalTimingCheck procedure deep in the body of the model architecture.

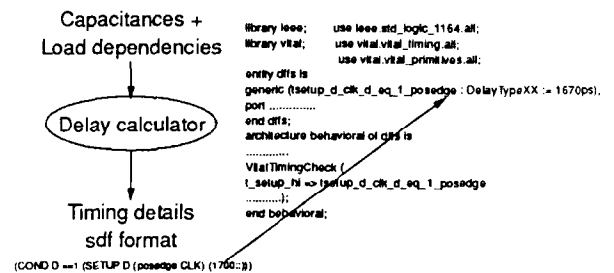


Diagram 1 : SDF mapping onto generics

There are two levels of VITAL compliance, Level 0 and Level 1. To be Level 0 compliant, a model must support back-annotation from SDF. It does not have to use the VITAL packages, so Level 0 models may not be portable across simulators and are not amenable to accelerated simulation techniques. Level 1 compliance means that a model supports SDF back-annotation, and MUST use both the VITAL packages, ensuring portability across simulators and offering the possibility of accelerated simulation.

Simulation users and modelers should aim for Level 1 compliance with the following caveat. VITAL compliance does not

guarantee sign-off quality, or even correct functionality, just conformance to a particular modeling style.

Delay Modeling Styles

Level 1 VITAL compliance specifies two ways of modeling delay, distributed delay and pin-to-pin delay. These styles must not be mixed if you want to be able to accelerate simulation of the models.

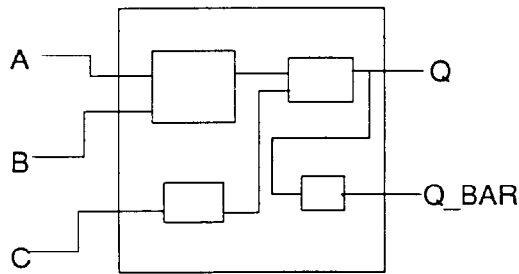


Diagram 2 : Distributed delay

Distributed delay is very simple in concept. A model is represented by a netlist of VITAL primitives, with the timing delay divided among the internal nets. The alternative modeling style, pin-to-pin delay modeling, uses a single process containing a state-table to represent the cell's functionality. The timing is contained in a block with individual procedure calls for each output port.

A Level 1 VITAL compliant model using the pin-to-pin modeling style has the following structure:

library declarations

entity declaration
 generic list
 port list

architecture declaration
 ATTRIBUTE VITAL_LEVEL1 of
 behavioral : architecture is TRUE;
 signal declarations

begin (architecture)

WIRE DELAY: BLOCK Procedural
 calls to VitalPropagateWireDelay

VITALBehavior : process
 variable declarations
 state table declaration

begin (VITALBehavior process)
 TIMING CHECKS
 perform violation checks
 set violation flags
 instantiate state tables for
 functionality
 Path Delay Section
 perform path delay
 calculation
 end process; (VITALBehavior)

end behavioral; (architecture)

First the IEEE and VITAL timing and primitive libraries are declared in the library declaration.

Next we have the entity declaration with the model's name, the list of generics to handle timing delays and violations and port list.

Next we have the model architecture. This begins with a declaration of the architecture name followed by the Level 1 attribute, which indicates to a simulator that it can attempt to accelerate this model.

Next are signal declarations for each input and bi-directional port. These signals are created to model timing delays. Each signal is delayed with respect to a change on the respective signal. This delay represents the delay caused by the interconnect. The VitalPropagateWireDelay procedure reads the value of the appropriate generic and assigns that delay to the internal signal.

Inside the Level 1 process we have the model functionality and timing. Any state-tables and variables to be used inside the process are declared before the beginning of the process. The first section inside the process contains the timing checks. The VITAL procedures VitalPeriodCheck and

VitalTimingCheck perform violation checks and set violation flags which force the state-tables to set the affected outputs to 'X'. The second section contains the model's functionality. For sequential models, one or more state-tables are instantiated. For combinational models, Boolean equations are preferred, although some modelers use truth-tables. The third section calculates the path delays, through calls to the VitalPropagatePathDelay procedure. Note : the order of expressions and procedure calls within the process matters, as they are executed in the order in which they appear.

Let's look at some sections of this model in more detail.

State tables

The state table construct is a very compact way of representing the functionality of a sequential cell. It can be considered to be a truth-table with an extended symbol set. For example, the symbol 'S' represents a steady value. The functionality is defined in the state table declaration. This is instantiated in the process itself. During simulation, the current input values, including the violation flag, are compared with each line of the truth-table in turn, until a match is found. If more than one line matches, the first line to match is taken.

The VITAL specification does not define how a simulator should perform X-handling (unlike System HILO) If no line matches, then all outputs are set to 'X'. To avoid undue pessimism, with all outputs being set to 'X' every time one of the inputs becomes 'X', the modeler must define X handling within the table. All likely input combinations which do not cause all the outputs must be in the table. This can cause the table to become large, but is identical to the way 'X' values are modeled in Verilog. However, unlike Verilog, the memory requirements of large state-tables, are not infeasibly big.

```

CONSTANT StateTab : VitalStateTableType (1 TO
14, 1 TO 9) := (
-- pr clr ck d violation q nq q nq
(' ', ' ', ' ', ' ', 'X', ' ', ' ', 'X', 'X'),
('0', '0', ' ', ' ', ' ', ' ', ' ', '1', '1'),
('0', '1', ' ', ' ', ' ', ' ', ' ', '1', '0'),
('1', '0', ' ', ' ', ' ', ' ', ' ', '0', '1'),
('1', '1', '/', '0', ' ', ' ', ' ', '0', '1'),
('1', '1', '/', '1', ' ', ' ', ' ', '1', '0'),
('1', '1', '0', ' ', ' ', ' ', ' ', 'S', 'S'),
('1', '1', 'S', ' ', ' ', ' ', ' ', 'S', 'S'),
('0', 'X', ' ', ' ', ' ', ' ', ' ', '1', 'X'),
('X', '0', ' ', ' ', ' ', ' ', ' ', 'X', '1'),
('X', '1', '0', ' ', ' ', ' ', '1', '0', 'S', 'S'),
('X', '1', 'S', ' ', ' ', ' ', '1', '0', 'S', 'S'),
('1', 'X', '0', ' ', ' ', ' ', '0', '1', 'S', 'S'),
('1', 'X', 'S', ' ', ' ', ' ', '0', '1', 'S', 'S')
);

```

The preceding code shows the state table describing the functionality of a d-type flip-flop with preset and clear. This state table is then instantiated in the body of the architecture as follows :

```

VitalStateTable ( StateTable => StateTab,
                  DataIn =>
(pr_ipd,clr_ipd,ck_ipd,d_ipd,violation),
                  NumStates => 2,
                  Result => Res,
                  PreviousDataIn => Prv
);

```

Memos on the VITAL reflector suggest that there are limitations to the complexity of parts which can be modeled using the state table construct. VEDA's modeling team have not found this to be the case. They have created more than 3000 Level 1 compliant models for a variety of sub-micron processors, including a 0.5 micron process. Models have been for a complete spectrum of ASIC cell types, from simple combinational gates, right through to multiple clock scan flip-flops. Each cell was modeled using state tables in a single process.

Obviously the more inputs and outputs a cell has, the larger the state-table needed to model it. Large VITAL state-tables can be cumbersome to write. (unless the data book contains a suitable truth-table you can adapt!) The simple solution is to break the functionality down into a set of smaller

state-tables, connected through variables, all instantiated within a single process.

Here we have three small state-tables being instantiated to describe a complex scan cell, fd1s2s from LSI Logic's LCA300k library.

```
VitalStateTable ( StateTable => StateTab0,
                 DataIn =>
(sck2_ipd,si_ipd,violation0),
                 NumStates => 1,
                 Result => Res0,
                 PreviousDataIn => Prv0
                 );
```

```
VitalStateTable ( StateTable => StateTab1,
                 DataIn =>
(te_ipd,cp_ipd,d_ipd,ti_ipd,violation1),
                 NumStates => 1,
                 Result => Res1,
                 PreviousDataIn => Prv1
                 );
```

```
VitalStateTable ( StateTable => StateTab2,
                 DataIn =>
(sck1_ipd,cp_ipd,w1_int,w2_int,violation2),
                 NumStates => 2,
                 Result => Res2,
                 PreviousDataIn => Prv2
                 );
```

Single or multiple processes?

Showing how state-tables can be used to describe any complexity of cell, leads us to a fairly controversial subject, the use of multiple processes. Version 3.0 of the VITAL standard allows the use of multiple processes within Level 1 architectures. So a cell can be represented by a network of processes, similar to a network of VITAL primitives.

Supporters for this modeling style claim that it is easier to write than a single process containing state-tables, and is applicable to a wider range of cell types. I think that the foregoing examples have shown that this is not the case.

In addition, supporters of multiple processes claim that state-tables are simulation and memory intensive.

Obviously this is very simulator dependent. Verilog is known to have difficulties with large truth-tables. But a simulator implemented according to the VITAL standard need not encounter memory or performance problems with state-tables. Indeed, if you consider the simulation and memory overhead associated with each process, a small cell implemented using three processes will require three times more memory than its single process counterpart and need about double the simulation power.

Resolving Ambiguities in the VITAL Standard

Version 2.2 of the VITAL standard is ambiguous in places. For example, Section 2.7.3 of the specification gives several alternative forms of the generic for Propagation Delay, including a form (tpd) with neither input nor output port specified, so that it can be used as a default, when a path between two ports has not been given its own unique generic. The examples given in Appendix C, always specify both input and output ports (tpd_CLK_Q). VEDA decided to always specify both ports, as this fits in better with the method for acquiring timing data, which always provides a separate timing delay for each input-output pair.

Section 2.7.3 specifies a generic for Input Release Time, trelease, which cannot be annotated, as there is no equivalent construct in SDF. As back-annotation is not supported, VEDA models do not contain this construct.

Section 2.7.3 specifies a generic called trecovey, which is used to describe input removal time, for which the generic terminal is used. VEDA models do not contain this construct anyway, as it cannot be back-annotated.

The crucial factor when interpreting ambiguities in the standard, is to ensure that the resultant models can be annotated from SDF. For example, setup times are modeled using the generic tsetup_d_cp. Setup times vary according to whether the

data pin is high or low, so setup time takes two values. The VITAL specification states that a two valued setup generic should be of type DataType01 and a single valued generic of type DelayTypeXX.

VEDA opted for DelayType01, but the models did not back-annotate successfully. This is because SDF format has a single delay value for setup times, where this single value is then conditional on the value of the data pin. VEDA rewrote the models defining tsetup_d_cp to be of type DelayTypeXX. Delays of this type have a single value which may be conditional on the value of another signal. This is consistent with the SDF approach, so back-annotation was successful.

VITAL 3.0

Maybe we should have just waited for Version 3.0 to come along. We were not the only modelers experiencing problems! Version 3.0 resolves the ambiguities associated with generics.

When creating generics for propagation delay, input setup time, input hold time and recovery time, you must specify both port names, instead of leaving either or both port names blank, to denote wild-cards. This avoids the possibility of any mismatching, and saves time during back-annotation, as the back-annotation does not have to search through the whole design to find ports which might match.

When creating generics for period and pulse width, you can no longer create separate generics for minimum and maximum values. This is because the SDF format always specifies only the minimum values.

The generics for removal time, trecovey and release time, trelease, are no longer used as they will never be back-annotatable from SDF.

Version 2.2b also had several minor bugs in the VITAL packages. This didn't affect VEDA, as we used the Issue Reports to avoid the problems.

Memory Modeling

There is nothing in the VITAL standard specifically related to modeling memory.

The first memory models VEDA wrote were one bit RAMs for a LSI Logic library. These were treated as d-type flip-flops and modeled using state-tables. This technique is not appropriate for larger models, as the state-tables become unwieldy. With 62 PCB models to write, ranging from a 16 x 1 static RAM to a 4M x 36 dynamic RAM, a different approach was required.

For each part, the functionality was described in 'C' and compiled. This 'C' model was associated with a VHDL entity, which contained generics for timing delays and violation as usual, but with an empty architecture. During simulation, the simulator takes functionality from the compiled 'C' models, back-annotating from SDF if it exists, or using the default timing in the 'C'; models if it does not. The timing and violation semantics are provided by the standard VITAL packages.

Here is the model for a 2k x 8 Static RAM.

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

entity sram2kx8v70c is
  generic
  (
    tpd_oeb_io  : DelayType01Z := (1 ns, 10 ns,
                                   35 ns, 40 ns, 35 ns, 40 ns);
    trecovey_csb_web_negedge_posedge
      : DelayTypeXX := 65 ns;
    tsetup_a_web_csb_eq_0_posedge :
      DelayTypeXX := 65 ns;
    thold_web_a_csb_eq_0_posedge  :
      DelayTypeXX := 5 ns;
    tipd_csb      : DelayType01Z := (0 ns, 0 ns, 0 ns,
                                   0 ns, 0 ns, 0 ns, : string := "");
    INTEL_DUMP_FILE : string := "";
    TimingChecksOn : Boolean := TRUE;
    -- Rest of generics deleted);

  port
  (csb      : in      std_logic;
    -- Rest of ports deleted);
end sram2kx8v70c;
```

```

architecture behavioral of sram2kx8v70c is
  ATTRIBUTE VITAL_LEVEL1 of behavioral
    : architecture is TRUE;
-- Signal declarations
-- WIRE_DELAY BLOCK
-- VITALBehavior process declaration

BEGIN
-- TIMING CHECKS --
-- PATH DELAY DECTION
-- NO FUNCTIONALITY
end process;
end behavioral;

```

By default, all addresses will initialize to 'X'. VEDA has created several additional generics INIT_VALUE and INTEL_INIT_FILE which allow the user to initialize all locations to 0 or 1, or to initialize from data values contained in an extended Intel format file. Another generic INTEL_DUMP_FILE, reads the entire memory contents into a dump file. This is useful during debug, to check that your memory contents are as expected. It can also be used as input to proprietary programming tools to blow EPROMs, as the ultimate check on a design.

An alternative strategy currently under development makes it easier for designers external to VEDA to develop their own memory models. VEDA will supply them with a special VHDL package containing functions and procedures for memory development. Part of the package header and some sample routines are shown below.

```

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
package ram_package is
  type ramdatatype is access
    std_logic_vector;
  type ramrectype is record
    data : ramdatatype;
    length : natural;
    addr_width : natural;
    data_width : natural;
  end record;
  type ramtype is access ramrectype;

```

```

-- Function declarations
function not_same_to_x(data1, data2 :
  std_logic_vector)
  return std_logic_vector;

function ram_create(length, addr_width,
  data_width : natural;
  init_val : std_ulogic := 'X')
  return ramtype;

-- Procedure declarations
procedure ram_dump(variable ram : in ramtype;
  filename : string);

```

```
end ram_package;
```

```

-- Package body
-- not_same_to_x: given 2 std_logic_vectors of
-- the same size, return a result of the
-- same size. If the 1st element of the 2
-- vectors is the same, then the first
-- element of the result is that value,
-- otherwise it is X. The same algorithm is
-- applied to subsequent elements.

```

```

function not_same_to_x(data1, data2 :
  std_logic_vector)
  return std_logic_vector is
  alias data2_alias :
  std_logic_vector(data1'range) is data2;
  variable result :
  std_logic_vector(data1'range);
begin
  assert data1'length = data2'length;
  for i in data1'range loop
    if data1(i) = data2_alias(i) then
      result(i) := data1(i);
    else
      result(i) := 'X';
    end if;
  end loop;
  return result;
end;

```

```

--ram_dump: dump contents of a ram.
procedure ram_dump(variable ram : in ramtype;
  variable textfile : out text) is
  variable start, finish,
    data_width : natural;
  variable index : integer;
  variable L : line;
  variable data : ramdatatype;
begin
  start := 0;
  data_width := ram.data_width;
  data := ram.data;

```

```

    for i in 0 to ram.length - 1 loop
        finish := start + data_width - 1;
        write(L, i);
        write(L, ' ');
        write(L,
logic_vector_to_string(data(start to finish)));
        writeline(textfile, L);
        start := finish + 1;
    end loop;
end;

```

Here is a 16 x 8 RAM model using the special memory package.

```

library ieee;
use ieee.std_logic_1164.all;
library vital;
use vital.ram_package.all;

entity ram16x8 is
    port (web : std_logic;
        waddr : in std_logic_vector(3 downto 0);
        raddr : in std_logic_vector(3 downto 0);
        d_in : in std_logic_vector(7 downto 0);
        d_out : out std_logic_vector(7 downto 0));
end;
architecture a of ram16x8 is
    attribute vulcan_ram : boolean;
    attribute vulcan_ram of a : architecture is
        true;
begin
    -- VITAL TIMING block here
    process(web, waddr, raddr, d_in)
        variable ram : ramtype :=
            ram_create(16, 4, 8, 'X');
        variable r_data : std_logic_vector
            (7 downto 0);
    begin
        -- VITAL TIMING section here
        -- RAM functionality section
        if x01_value_change(web, '1', '0') then
            ram_write(ram, waddr, d_in);
        elsif x01_has_changed(web) and web =
            'X' then
            ram_write_same(ram, waddr, d_in);
        elsif web = '0' and
            x01_has_changed(d_in) then
            ram_write(ram, waddr, d_in);
        elsif web = '0' and
            x01_has_changed(waddr) then
            ram_set_to_X(ram);
        end if;
        ram_read(ram, raddr, r_data);
        -- PATH-DELAY section
    end process;
end;

```

PCB Modeling

PCB models are not particularly suitable for modeling in strict accordance with the VITAL standard. The models become very large. Moreover, high quality 'C' models and hardware models already exist.

VEDA has adopted an approach similar to that described above for memory models. A VHDL entity describes the interface in a style which supports back-annotation from SDF. The functionality is described using a corresponding 'C' model.

Key to this technique is a strategy for ensuring the SDF contains accurate delays. For example, the delay calculator should be able to model non-linear loading effects for pin-to-pin delays. VEDA's delay calculator uses a piece-wise linear equation to model this.

Automating Model Generation

As I mentioned earlier, VEDA's modeling team has written over 3000 VITAL models in the space of about three months. This level of activity was only possible by using MetaSoftware's MASTER Toolbox software to automate the process. For example, VEDA's library for LSI Logic's 0.6 micron process was generated and validated in 4 man-weeks. It contained 1,964 cells.

MASTER Toolbox works as follows. A 'golden' model, typically Verilog or HSPICE, is iteratively simulated in its native simulation environment until all possible information is learnt from it: functionality, timing delays and violations. This information is stored in a compact form in Intermediate Neutral Format (INF) and used to generate a VITAL model. This model is then itself repeatedly simulated using a VITAL compliant simulator so that all possible information is extracted from it. This information is then compared against the results of the 'golden' simulation, as a Quality Assurance check on the final digital model.

The task was straightforward. Only one cell, a 16 input NAND gate, could not be

'learnt' by the MASTER Toolbox, because of the large number of possible input combination needing to be exercised to learn the functionality. The only other problems experienced were with five complex scan cells. MASTER Toolbox learnt all the timing information, but could not convert this information into INF for the validation stage. It was a simple task to convert the learnt timing data into INF and thus ensure that every cell was validated with the MASTER Toolbox.

It is necessary to automate the modeling process for several reasons. First, you need many lines of timing information to model a deep sub-micron cell. It takes a long time to develop this information manually. Second, the timing information contained in the digital models must correspond to the actual timing behavior. MASTER Toolbox automatically obtains the timing information from a 'golden' model of the cell, so no errors are introduced through manual intervention. In addition, the new models are compared back against the 'golden' model, to make sure that they exhibit the same behavior. Third, as we have discussed, VEDA's models have had to change their implementation of the VITAL specification, to work-around some of the ambiguities. These modifications were easy to implement, by modifying the MASTER Toolbox VITAL modeler, and then regenerating and verifying the models

Summary

Summarizing, VITAL 2.2b standard was sufficient to create several sign-off quality libraries, though version 3.0 will be easier to use.

We found using MetaSoftware's MASTER Toolbox software invaluable for creating accurate, validated models in a short space of time

VEDA's modeling team recommend the use of single process models containing one or more state-tables. This style gives superior memory utilization and simulation performance.

The standard does not currently address memory parts or PCB models, so we have developed our own solution, based on compiled 'C' models, with standard VITAL timing and violations, back-annotated from SDF. VEDA welcome any extensions to the VITAL standard which address these areas.

Acknowledgements

The author would like to thank Gavin Davis and Dr Alan Mayes, both of VEDA Design Automation for their contribution to this paper.