

VHDL Sign-off Simulation : What Future?

Przemyslaw Bakowski^(a), Frédérique Bouchard^(a), Jean-Paul Caïsson^(b), Frédéric Igier^(b)

^(a)IRESTE, La Chantrerie, CP 3003, F-44087 NANTES Cedex 03, France

^(b)MATRA MHS, La Chantrerie, CP 3008, F-44087 NANTES Cedex 03, France

Abstract : We are convinced that a universal VHDL gate library for ASIC sign-off simulation can be developed, though the optimized VHDL code for various target simulators may differ. Our solution is based on VHDL models written with a unique entity declaration and various architecture bodies targeted at simulators. We concentrate here on VITAL compliant architectures as VITAL should soon become a standard.

I Introduction

One important issue in the development of ASICs is that the use of VHDL in the design flows currently stops at the logic synthesis level (figure 1). The use of a large number of netlist languages and the range of various corresponding simulators leads to the development of a gate library for each simulator. Therefore, it is clear that the main problems of such design flows are maintenance (e.g. any change has to be reported on every simulator) and distortion of information when translated from one language to another in the design flow. That is the reason why the idea of a complete VHDL design flow arose (figure 2). But still one problem remains : it is not thinkable to simulate efficiently an ASIC described in a full VHDL at the gate level. Thus, tool vendors are now integrating built-in primitives in their simulators in order to accelerate simulations and increase the number of simulated gates within a design.

In the meantime, model development specifications of VITAL (VHDL Initiative Toward ASIC Libraries) are trying to standardize the way of writing VHDL models in order to make them compatible, while standardization of back-annotation should come with widespread use of SDF (Standard Delay File) format.

This paper concentrates on the development of a VHDL gate library for the Sea of Gates technology, taking into account the previous considerations. Section II gives our

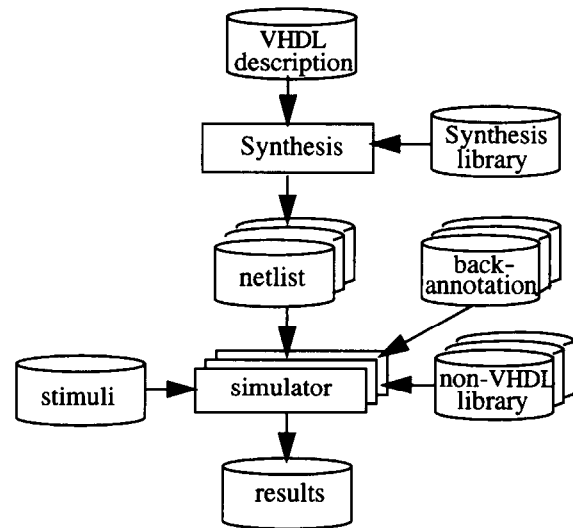


Fig. 1 : Current design flow

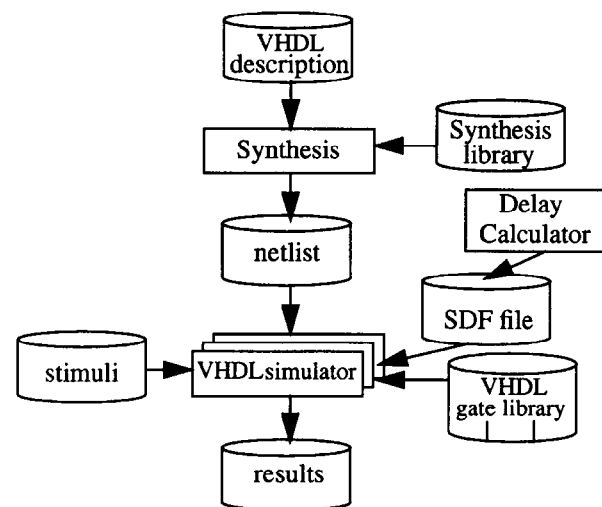


Fig. 2 : Design flow using VHDL from description to gate level

motivation to start the development of a VHDL sign-off library. Section III presents the strategy for the development of such a library. The development itself is detailed in section IV. Experimental results are provided in section V. Finally, some conclusions are drawn.

II Motivation

Customers (ASIC designers) demand performance and above all quality from ASIC vendors. That is the reason why the design flow of figure 2 does not seem realistic for some people.

As a matter of fact, it was not realistic a few years ago because of performances and because of portability.

Platforms provided by tool vendors were certainly not powerful enough to simulate complex ASICs at the gate level.

Moreover, two kinds of simulators were available : full-VHDL simulators, i.e. simulators developed from scratch to implement the IEEE-1076 standard, and VHDL-adapted simulators which accept VHDL descriptions but whose kernels have not been changed and therefore use their own built-in resources during compilation and simulation. A study presented in [GCG91] shows that there is a 100% portability between full-VHDL simulators, whereas no portability exists between VHDL-adapted simulators or between VHDL-adapted and full-VHDL simulators.

Those main limitations explained why some ASIC designers were suspicious of a complete VHDL design flow. But nowadays, those objections are not justified anymore. Most tool vendors are able to supply ASIC designers and foundries with platforms and simulators for sign-off quality. Furthermore, VITAL -which is an initiative coming from the industrial side- serves as an efficient link between ASIC vendors and tool vendors, this link being regulated by customers.

The goal of this paper is to show our conviction that VHDL sign-off simulation is becoming possible.

III Development strategy

As shown on the VHDL design flow in figure 2, a unique library has to be developed at the gate level and should fit every simulator, though different simulators may not accept the same VHDL source code for a defined gate, e.g. some of these tools include directly usable built-in primitives and others do not.

We suggest the process presented in figure 3 : the entity

declaration of a model can be the same for all target simulators, whereas one architecture body has to be written for each tool. The generic parameters in the entity declaration have therefore to embrace all parameters possibly needed by the different models of the same component, even if some models do not use them all.

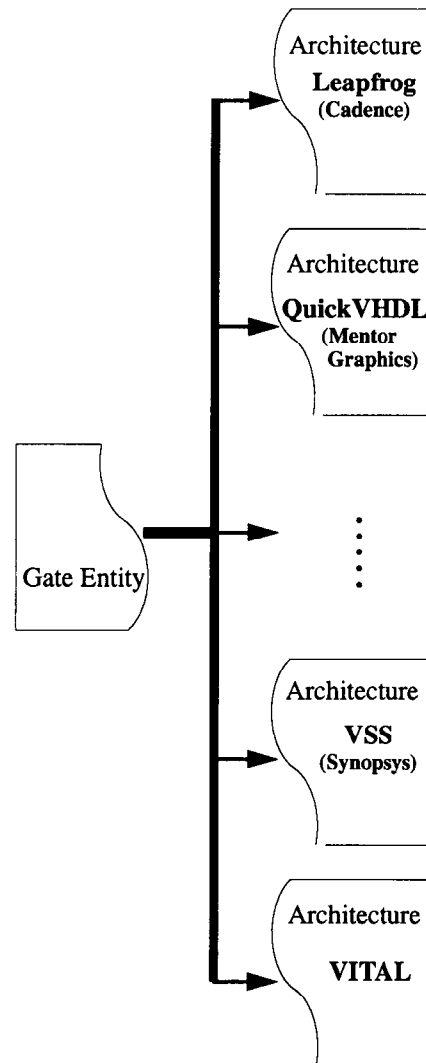


Fig. 3 : Structure of a component stored in the library

The idea of this strategy is derived from the study reported in [GCG91] and was suggested in [CH93], where a slightly modified concept was presented : the native (targeted) level was lower, in the architecture body, it concerned the call of targeted primitives. The difference was due to the unique simulation environment we had

available at that moment - System HILO from Genrad which did not accept several architectures for an entity.

Knowing that the goal of the project is to simulate ASICs of more than 100,000 gates, the basic models described in the library have to be really highly simulation-performant, and consequently have to adhere to a number of strict rules. As a big endeavour [VITAL] towards standardization of those rules was simultaneously taking shape we decided to adopt it as soon as possible.

Our strategy is first to develop some well chosen models manually in order to understand and extract the rules of derivation for each category of models (e.g. combinational gates, sequential gates...) and then to develop a generation tool well adapted to our needs. This paper puts the emphasis on the first part, i.e. the writing of models.

IV The development of prototype models

IV.1 Our environment for the simulation

The VHDL gate library we are developing will be used for our currently available environment, comprising the following full-VHDL simulators :

- Leapfrog v1.0 from Cadence
- QuickVHDL v8.2_1.4.1b from Mentor Graphics or V-System/Workstation v2.5 from Model Technology
- VSS v3.1 from Synopsys (gate level simulator)

We add VITAL to this list of targets for the different architectures, hoping that it will soon become a standard available with all simulators.

IV.2 The available primitives

The first two simulators mentioned do not own any built-in primitive. For them, we chose to develop the models according to VITAL rules and using VITAL packages. Therefore, VITAL target actually embraces three targets.

VSS v3.1 simulator from Synopsys includes a short number of generic and accelerated built-in primitives :

- TLU and TLU_INTEGER : combinational lookup-tables
- SEQGEN : sequential lookup-table
- WIREBUF : transport wire delay buffer

All models written using those primitives should contain, in fact, solely instantiations of the above primitives.

VITAL provides primitives in the form of procedures and functions. The primitive functions are given to support behavioural modelling styles; the primitive procedures are supplied to support structural modelling styles. The VITAL packages corresponding to VITAL specifications v2.1e [VITAL21] are not available at present. The models presented here have been written according to VITAL_Timing and VITAL_Primitives packages v0.301. Some comments will be added to show what the new specifications will bring explicitly.

IV.3 The representative models developed

Two models have been chosen to represent the two main categories : an AOI22 gate (computing $\text{not}(AB+CD)$ where A, B, C, D are inputs) for the combinational category; and a SFFRSB (Scan Flip-Flop with Reset and SetBar) for the sequential category.

Synopsys models are automatically generated from the target library defined for synthesis thanks to a tool provided with Synopsys environment, Library Compiler. The problem of compatibility between the Synopsys entity declarations and the VITAL ones will be studied later on.

The main problems we met using VITAL packages v0.301 are due to the fact that VITAL primitives do not suit exactly our needs. For our SFFRSB gate, the SetBar input is active low and Reset has higher priority than SetBar, whereas VITAL primitive DFFSC has no inversion on the Set input, and Set has higher priority than Clear. Moreover, a multiplexor has to be added to this primitive in order to obtain the Scan functionality. Further, we cannot maintain information of path delay, i.e. we cannot choose the right path delay to assign the output if we do not know which input has affected the output.

This problem is obvious for the AOI22 gate. It cannot be written in a structural way using primitives like `VitalAND2()` and `VitalNOR3()`, because intermediate delays are not available, and we cannot get the path information if we use primitive functions.

Those issues should be resolved with the next release of packages as a primitive called `VitalPropagatePathDelay()` should handle this path information at a general level. As for the inconsistency between our SFFRSB gate and VITAL's DFFSC primitive, a single primitive called `VitalStateTable()` will handle all sequential gates.

For the time being, the only possible way of writing models using VITAL packages is to write models the same

way as primitives are themselves written in VITAL_Primitives package v0.301, i.e. not using predefined component-primitives, but at least by using useful VITAL functions.

The AOI22 gate

This gate has 4 inputs declared as A, B, C, D : IN std_ulogic; and one output declared as O : OUT std_ulogic in the entity declaration. The architecture body has a single process. A large part of it is common to already existing primitives, we only have to pay attention to the inputs : are they inverting or not (or even both), and then write the functionality on a well defined way.

Here all the inputs are inverting ones (an edge on the output can only be produced by the reverse edge on an input). Therefore, there is a call to the VITAL InvPath() procedure for each input, once for the initialization of the delay schedules (1), and then in the main loop in order to re-evaluate these delay schedules each time something happens (2).

```
(1) InvPath(A_Schd,InitialEdge(A),tpd_A_O));
...
(2) InvPath(A_Schd, GetEdge(A), tpd_A_O));
...
```

The functionality is described simply. In case the switch UnitDelayOn is TRUE, we have :

```
IF UnitDelayOn THEN
L0: LOOP
O <= not((A and B) or (C and D)) AFTER
0.1 ns;
WAIT ON A, B, C, D;
END LOOP L0;
```

Otherwise, we begin with initializations (1) and enter the main loop with initializations (2). Then, the new value and the new scheduled delay are computed :

```
new_val := not((A and B) or (C and D));
new_schd := not((A_Schd and B_Schd) or
(C_Schd and D_Schd));
```

The handling of scheduled delays is then made by the VITAL GetSchedDelay() procedure and the output O is assigned through the VITAL GlitchOnEvent() procedure which also detects glitches.

The SFERSB gate

This gate has 6 inputs and one output :

```
Q : OUT std_ulogic; -- D Flop out
D : IN std_ulogic; -- Data
CK : IN std_ulogic; -- Clock
R : IN std_ulogic; -- Reset
SB : IN std_ulogic; -- SetBar (active low)
SCAN: IN std_ulogic; -- Scan Enable
SIN : IN std_ulogic; -- Scan Input
```

The functionality and timing aspects of the gate are done through a single call to the procedure VitalSFERSB() we have defined in the architecture body. As VITAL recommends the use of tables to support sequential cells, the definition of VitalSFERSB() is based on the following "state table":

```
CONSTANT CondTable : ConditionTableT(1 TO
14, 1 TO 6) :=
-- "clock scansel scanin data reset setbar"
(1 => "----1-", --0
2 => "----X-", --X
3 => "----00", --1
4 => "----0X", --X
5 => "/0-001", --0
6 => "/0-101", --1
7 => "/0-X01", --X
8 => "/10-01", --0
9 => "/11-01", --1
10 => "/X--01", --X
11 => "X---01", --X
12 => "----1-", --X
13 => "----V01", --X
14 => "-----" --q
);
```

The first part of the architecture concerns the computation in case of a unit delay simulation, i.e. where all delays are equal to 0.1 ns and no timing check is performed. Otherwise, the delay schedules are evaluated (use of InvPath() for the inverting inputs reset and setbar, and use of both InvPath() and BufPath() for the clock-to-output delay). Then, the timing checks are performed by calling specific VITAL predefined procedures or functions (e.g. SetupHoldCheck()). We find then the computation of functionality and propagation delay : the VITAL predefined function SelectCondition() indicates which line of the above table is relevant at a given time and a CASE statement

specifies the action to be taken and updates the scheduled times :

```

CASE SelectCondition (EdgeArray6'
(clock_edge, scansel_edge, scanin_edge,
data_edge, reset_edge, setbar_edge),
CondTable) IS
  WHEN 1      -- ----1- [reset]
    => q_val := '0';
        q_dly := sch_reset_q_inv.inp1 - NOW;
  WHEN 2      -- ----X- [unknown reset]
    => q_val := 'X';
        q_dly := sch_reset_q_inv.inpx - NOW;
  WHEN 3      -- ----00 [setbar]
    => q_val := '1';
        q_dly := MAXIMUM(sch_setbar_q_inv.inp0,
                          sch_reset_q_inv.inp0) - NOW;
  ...
  WHEN 7-- /0-X01 [rising clock, unknown data]
    | 10-- /1X-01 [rising clock, scan,
                  unknown scanin]
    | 11-- /X--01 [rising clock,
                  unknown scansel]
    |12-- X---01 [unknown clock]
    |13-- ---V01 [Timing Violation]
    => q_val := 'X';
        q_dly := sch_clock_q_buf.inpx - NOW;
  WHEN OTHERS -- [No reset/setbar, No clock
                  edge, No errors]
    => q_dly := -1 ns;
END CASE;

```

The output is assigned, through the `GlitchOnEvent()` procedure, with the new output value `q_val` and the new delay `q_dly`.

V Experimental results

A single AOI22 model and a single SFFRSB model have first been checked. Then, tests have been made in order to find out whether our goal can be reached, namely to simulate a network of more than 100,000 gates in a reasonable lapse of time. Up to now, the circuits we developed were no larger than about 70,000 gates.

A Sea of Gates chip is an array of basic frames. Each frame is composed of PMOS and NMOS transistors. The interconnections between these transistors transform a basic frame into a basic gate. All the components of a chip

are built up from associating these basic gates. An SFFRSB component does not comprise the same number of basic gates as an AOI22 component. Therefore, in order to ease the comparison, the number of gates provided in the tables is in fact the number of basic gates.

Simulations have been run on a SPARC 10 workstation with QuickVHDL v8.2_1.4.1b from Mentor Graphics. We made some comparisons with classical simulations, here with QuickSim II v8.2_6.1 from Mentor Graphics.

Table 1 provides results obtained for an array of AOI22 models. We ran simulations on 2 ns. The typical delay times are : propagation delay for a rising edge $t_{plh}=0.47$ ns, propagation delay for a falling edge $t_{phl}=0.24$ ns. We did not provide simulations run in unit-delay ($t_{plh}=t_{phl}=0.1$ ns) for VHDL simulation as results are almost identical on 2 ns.

Table 1: Simulation times for arrays of AOI22 models

nb of basic gates	VHDL simulation	classical simulation	
	typical times	unit-delay	typical times
100,040	58 min		
150,040	2 h 09 min	1 min 37 s	3 h 07 min

Table 2 provides results obtained for an array of SFFRSB models. We ran simulations on 8 ns with a clock period of 4 ns, i.e. on two clock rising edges. We give here results for unit-delay simulations and for typical time simulations as a big difference appears whether we use or not the timing checks. Moreover, contrary to the array of AOI22 models, we could easily find out the limit of the number of simulatable VHDL SFFRSB models. This limit corresponds to a number of 384,000 basic gates which is almost four times our goal.

For combinational gates, the tremendous number of events explains the long simulation times compared to those for sequential gates activated only on clock edges. Table 1 and Table 2 would imply that classical simulations are really efficient in case of unit-delay simulations, whereas VHDL-VITAL simulations seem to be better when typical times are used, unless we meet the limits of the VHDL simulator (384,000 basic gates for a SFFRSB model). Nevertheless, those conclusions do not match the experiments found in Table 3.

Table 2: Simulation times for arrays of SFFRSB models

nb of basic gates	VHDL simulation		classical simulation	
	unit-delay	typical times	unit-delay	typical times
150,600	12 min 34 s	18 min 38 s	1 min 57 s	22 min 02 s
225,600	27 min 35 s	41 min 18 s		
300,600	48 min 32 s	1 h 13 min		
384,000	1 h 18 min	1 h 58 min	7 min 05 s	1 h 01 min

Table 3 shows simulation times for a shift register composed of 2,000 SFFRSB models in the Scan mode. Here, at least 2,000 clock rising edges are needed to complete the simulation. With a clock period of 4 ns, we ran the simulations on 8.01 μ s.

Table 3: Simulation times for a shift register of 2,000 SFFRSB gates

VHDL simulation		classical simulation	
unit-delay	typical times	unit-delay	typical times
7 h 13 min	11 h 40 min	1 min 37 s	8 min 24 s

The topology of the circuit and the number of events seem to be decisive factors.

VI Conclusions

We have presented here the foundations for the development of a universal gate library for the Sea of Gates technology. We foresee a growing industrial interest for the new-born VITAL specifications. Therefore, we have decided to concentrate on the development of models according to VITAL rules.

However, we have to be careful with first experiments which show that VHDL simulations sometimes give better results than classical simulations. As a matter of fact, we

did not use the packages corresponding to the VITAL specifications v2.1e, so primitives have not completely been exploited. Moreover, the packages were neither built-in nor accelerated.

On the other hand, we did not use back-annotation, and the present models are valid for VITAL only, some additional code may be added for adaptation to other target simulators in order to complete a universal gate library. Further experiments are necessary and we are working towards more precise conclusions.

Acknowledgements

We would like to thank Ludovic Larzul for considerable help in writing VITAL models.

References

- [CH93] J-P. Caïsson, J-L. Hallé
A Universal Structural VHDL Library for ASIC Simulations
Proceedings of the VHDL-Forum for CAD in Europe, pp. 185-190, Innsbruck, March 1993
- [GCG91] P. Guiraudou, J-P. Caïsson, E. Garcia
ASIC Design and VHDL, Library Portability : A Standard Approach?
Proceedings of the VHDL-Forum for CAD in Europe, pp. 127-132, Marseille, April 1991
- [VITAL] VITAL: VHDL Initiative Toward ASIC Libraries, **Model Development Specification, Version 2.0, May 1993 and Version 2.1e, December 1993**
- [VITAL21] VITAL: VHDL Initiative Toward ASIC Libraries, **Model Development Specification, Version 2.1e, December 1993**