Using VHDL for Board Level Simulation

SANDI HABINC
PETER SINANDER
European Space Agency

Prototyping is necessary for successful development of printed circuit boards built with complex components such as microprocessors, ASICs, and ASSPs. The European Space Agency uses VHDL models for board level simulation, optimizing such models for high functional accuracy and simulation performance.

MOST SERIOUS ERRORS encountered in ASIC (application-specific integrated circuit) developments originate from unclear or incorrectly implemented specifications. To allow independent evaluation of a device's functionality, the European Space Agency (ESA) normally requests a VHDL model before a company starts the detailed design. This allows ESA or another company to verify the functionality. (For more about ESA and its choice of VHDL, see the box.)

A logical follow-on activity to using VHDL modeling for ASIC design verification is to model and simulate complete board designs.

Board level simulation

At this level we simulate the functionality of one or several printed circuit boards built with standard components, possibly incorporating ASICs and application-specific standard products (ASSPs). Board level simulation is also called rapid or virtual prototyping[1] and sometimes system simulation. Board level simulation's purpose is to verify the correct behavior of the board—that components operate as intended in selected configurations.

When board designs contain processors, it is possible to verify the hardware-software interaction, for example verifying the programmability of ASIC registers, the operation of software drivers, and so on. In addition, we can evaluate the performance of the processor board. Board level simulation will also yield information about timing correctness, though it can probably not fully replace worst-case timing analysis for high-reliability applications such as those in spacecraft.

To avoid unreasonable expectations of board level simulation, we must understand what it does not comprise. It does not simulate system performance, including aspects in which neither accurate data nor clock behavior are essential, such as throughput and latency. Neither does it comprise explorative simulation for defining system baselines.

By employing board level simulation, designers can integrate and test printed circuit boards early, since the first design and verification loop does not need manufactured hardware. Designers can therefore postpone manufacture until the specifications settle and they have verified all interfaces. Board level simulation supports a top-down methodology allowing simulation of unimplemented boards, which enables designers to work with incomplete specifications of a system or component and to verify functionality

0740-7475/96/$05.00 © 1996 IEEE

European Space Agency

ESA has 14 European member states; Canada is an associate member. It promotes cooperation among the member states in space research and technology and their space applications, in particular for scientific purposes and space application systems. Since ESA's charter is to develop the member states' space industry, it awards contracts for major technical work to external companies that function under its supervision.

ESA has several establishments in Europe, one of which is the European Space Research and Technology Centre (ESTEC) located in the Netherlands. ESTEC's Microelectronics and Technology section (WSM) deals with design methodology in microelectronics, development of very large scale integration (VLSI) components, and related topics.

Electronic equipment and components are an increasingly important part of spacecraft, as in most other modern, complex systems. Such electronics frequently employ special system concepts, integrated circuits, and process technologies due to the harsh environment in space, particularly the radiation effects. It is also practically impossible or extremely expensive to service a spacecraft after launch. Thus, special ICs are developed by the space industry and, in Europe, often under ESA contract.

Companies working with ESA range from large, multinational corporations with decades of relevant experience to small companies entering new business areas. In comparatively new fields such as ASIC design and VHDL modeling, ESA must therefore often actively support the development work. ESA thus tries to establish concepts and methodologies helping companies gain proficiency and increase their efficiency. Implanting such knowledge in companies is beneficial to ESA, ultimately reducing development risk, cost, and schedule.

Since various companies design ASICs, ESA must allow them to use different design tools. It is neither desirable nor feasible to force all companies to use a specific vendor's tools. In the early 1990s, when the agency was formalizing the ASIC design process for ESA developments, VHDL was the only hardware description language supported by multiple tool vendors and therefore the natural choice.

VHDL 93[1] is the baseline today to reduce long-term maintenance costs for VHDL models. After delivery, ESA cannot require the original model developer to update the model when needed. Although the agency initially intended to use a VHDL 87 baseline, we changed to VHDL 93 when it became clear that updating practically every VHDL 87 model would be necessary for compatibility with the current standard. Syntax and semantics differences exist between the two versions, and although these changes are technically minor, it would be a major task to update and subsequently verify a large number of VHDL models from various companies.

Unfortunately, few vendors have adopted VHDL 93. For the standard's next update, we consider it extremely important to ensure that only features likely to gain vendor support become part of the update. An update without supporting tools would be a failure.

Reference

1. *IEEE Std 1076-93, IEEE Standard VHDL Language Reference Manual*, IEEE, Piscataway, N.J., 1994.

earlier. In large developments with boards from several different subcontractors, designers can supply users with simulation models of a board design for early system verification. Board level simulation also allows designers to prototype manageable parts of larger systems. When designing an ASIC, designers can verify its operation in a board design before manufacture. Using models for board level simulation before first silicon can save significant schedule time.

Designers can simulate situations that are difficult to capture in real hardware due to timing synchronization and so on. This results in a more thorough verification of the board design and also provides the designer with unlimited probing and acquisition points, which is not always possible in hardware. It can also provide visibility into the internal state of different components, such as registers and state machines.

Models for board level simulation can provide limited simulation support during parallel development of hardware and software.[2] This type of simulation usually takes a long time to perform with such models, but delivers high functional and timing accuracy. However, by carefully selecting which software parts to simulate, we can reduce the simulations to manageable times. For instance, it may not be feasible to boot a complete operating system, but we could verify all the firmware and hardware drivers.

Board level simulation enables hardware and software designers to work together at an early stage and solve interface problems before committing the hardware to manufacture. Doing so is especially important with board designs

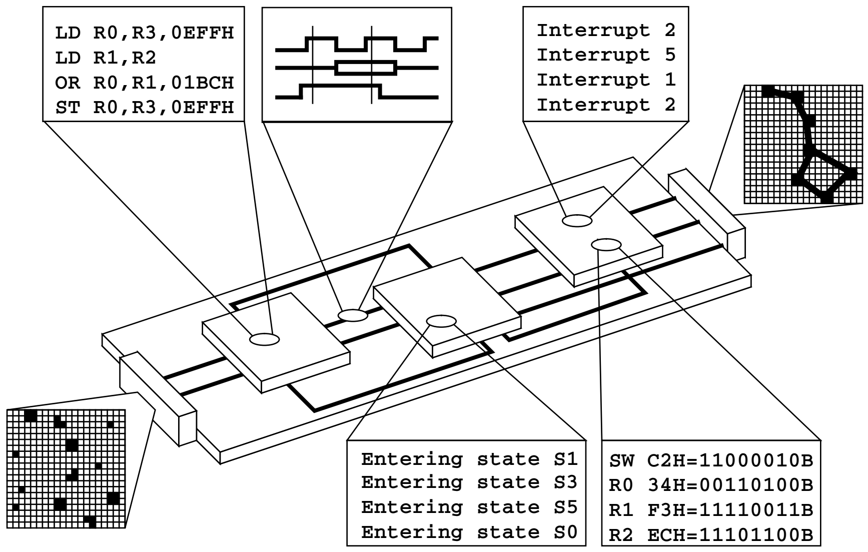


***Figure 1.** Designer's view of a board design when using board level simulation.*

incorporating ASICs because changes late in the design process cause significant schedule impacts and additional, nonrecurring engineering costs. This type of simulation will become more attractive with the continuously increasing speed of workstations and simulators. Figure 1 shows some features that concern designers at this simulation level; these include

- the functional interaction and timing aspects between components
- component internal states, for example, for finite-state machines
- occurrences of interrupts, exceptions, and other asynchronous events in the components
- instruction tracing, disassembly, and register access for processors on the board

A major issue for board level simulation is the availability of simulation models for board components. Despite the commercial models available for many standard components, board designs increasingly incorporate ASSPs, ASICs, and other unusual devices. VHDL models are therefore an interesting alternative when no models are available, the typical case for almost all components used in a spacecraft. Using VHDL greatly reduces the effort required to support several platforms and simulators, since VHDL models require only minor modifications, if any, for each new simulator. ESA has therefore chosen VHDL as the language for board level simulation in ESA-funded developments. See the Case studies box for examples of board level simulation.

Modeling for functional accuracy

Component behavior modeling, simulation performance, and ease of use for board designers characterize a model for board level simulation. The model's behavior as viewed from the outside should be the same as that of the actual component and include the component's full functionality. Such models need not implement specific test modes used only for manufacturing test, but their interface signals should have exactly the same waveform behavior as the component. Models for board level simulation must have a common interface, model timing features (setup and hold times, output delays, and so on), and handle unknown values for the model's inputs and outputs. These requirements also apply to modeling in VHDL.

We consider bus functional models (sometimes called bus interface models) as reduced models for board level simulation because they model only the timing and behavior of component interfaces. Designers typically use them for very complex devices, such as processors, in which case instruction execution, interrupts, and so on are not available or not implemented in the model. However, they do not model the component's internal functionality and thus do not provide the full potential of board level simulation (such as for software execution).

Models for board level simulation must implement the functional behavior of components accurately enough to allow verification of board designs for functionality and timing. Simulating boards using models with high accuracy will reduce the errors in the manufactured board. However, a model error can cause verification to miss board design errors. Or, an erroneous model can also indicate errors in an ASIC or printed circuit board when none actually exist. This can cause designers to introduce design errors into an originally correct board.

There are two major modeling approaches: independently developing the model from a functional specification or data sheet, and enhancing an existing model at the register-transfer level (RTL) or higher. Board level developers must use the first approach when no RTL model or only a gate level model is available. Figure 2 illustrates some sources of modeling information.

Developers should take care when developing a model with only a data sheet as input, because it may not fully describe the component. The data sheet writers may have left out details or introduced errors, and more chances for misunderstandings occur. Or, they may have simplified the data sheet information. For example, an interface

Case studies

Two ESA studies demonstrate board level simulation.

Data-handling computer

This activity, begun in 1994, aimed to design and simulate a fictitious computer for data handling on a spacecraft (Figure A). The board design incorporates a MIL-STD-1750 processor, memories, a bus interface for accessing other units on the spacecraft, four ASSPs implementing the up- and down-link communication protocols, and some glue logic. Two contracting companies wrote models for the processor, memories, and glue logic based on data sheets and optimized them for high simulation performance. In addition, they wrote basic software for board design verification in assembly code.

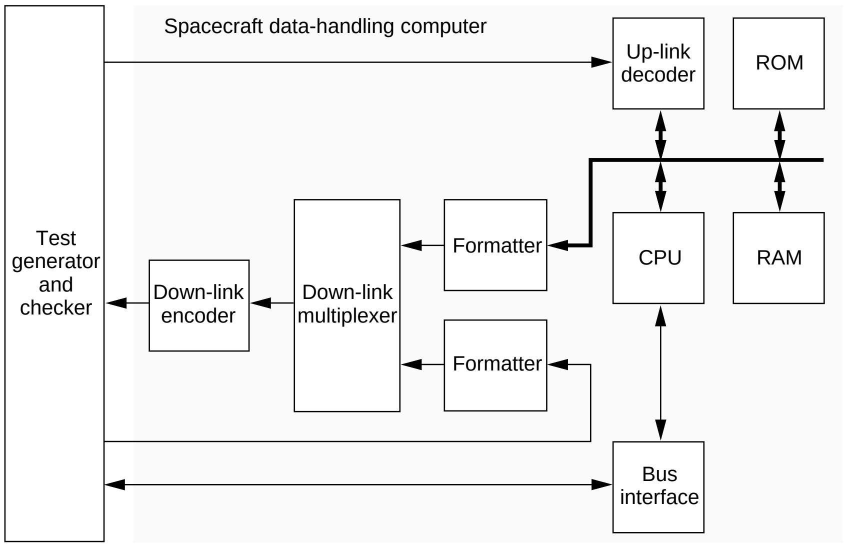


Figure A. *Fictitious computer for spacecraft data handling.*

The board level simulation detected about 10 errors in the hardware design and in the software. Hardware errors ranged from devices not being reset and incorrectly activated buffers to omissions in the board specification. While the software was intentionally very simple, it discovered several errors in the hardware-software interaction.

In addition to demonstrating board level simulation, this activity identified several issues related to complex models for board level simulation. For one, it currently seems difficult to reach more than 1,000 simulated instructions per second for a timing-accurate VHDL model of a processor. This low limit seems due to limitations in the signal scheduling of current VHDL simulators. As reference, an instruction simulator in VHDL (one without accurate timing simulation or signal scheduling) has roughly an order of magnitude higher performance. An instruction simulator written in C has up to two orders of magnitude higher simulation performance. This suggests that high-level optimization of current VHDL simulators should yield significant improvements.

We also found that developing complex models from their data sheets requires large modeling and verification efforts. As an example, the original verification for the processor model was extensive. It included test stimuli for each instruction as well as verification using part of the functional production test vectors. Nevertheless, we found many errors when later validating the model with the device's original design database, for which we generated pseudorandom sequences of several million instructions.

Spacecraft interface ASIC

In another case, a European space company successfully demonstrated the benefits of board level simulation in designing an ASIC for use by several satellites. The ASIC interfaces to several components using different protocols in an on-board data-handling system, and converts between these protocols.

Simulating the board containing the ASIC as well as components on other boards interfacing with it verified the design. Models developed according to the ESA guidelines represented two of these components. A second company developed one of these models, and ESA developed the other in house.

The ASIC simulations and these models revealed two errors in the ASIC design that designers could correct prior to manufacture. The errors occurred in the ASIC subblock interfacing the models developed for board level simulation, and undocumented component behavior caused them. Accurate modeling of the reset sequence for the two models and the start-up procedure for one of the protocols enabled designers to discover the errors.

Such success demonstrates the importance of modeling for functional accuracy. These models also allowed the company to generate test vectors for the printed circuit board containing the ASIC. Automatic test equipment uses the vectors to test the board during manufacture.

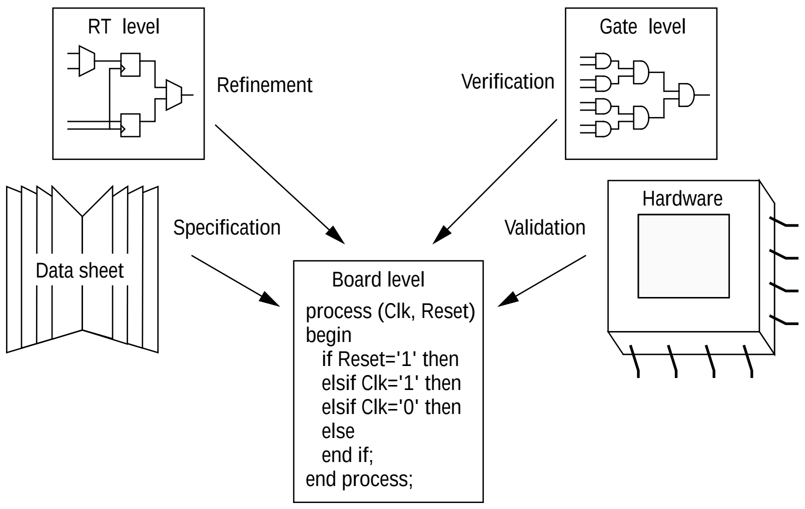


Figure 2. *Modeling information sources.*

protocol description may be more constraining than the actual design requirements, resulting in a partially correct model.

Designers normally write RTL models for synthesis, which conflicts with the simulation performance required for board level simulation. When revising an RTL model, we must also protect the design information because the resulting model might still be synthesizable. However, many simulation performance enhancements will reduce the probability that others can reverse-engineer a component. Thus, the main development effort would be to optimize the model for simulation performance, implement the model interfaces, and develop the verification stimuli.

Occasionally, the behavior that the RTL model describes and the data sheet are inconsistent. The data sheet may omit or simplify some of the device's functionality, such as proprietary design features. In such a case, we recommend modeling the full functionality and issuing a warning when an inconsistency with the data sheet occurs. We prefer this to excluding the function and consequently having an incorrect simulation. Also, the inclusion of unsupported or undocumented component functionality in the model for board level simulation could simplify its comparison with an RTL model during verification with the same set of stimuli. The more accurately the model reflects actual hardware, the more likely it is that simulation will detect problems in a board design.

Modeling for simulation performance

Present workstation and VHDL simulator performance provides a means for simulating board designs comprising several complex components such as microprocessors and ASICs. However, to tap this simulation performance, we must efficiently code models for simulation. Though defining an absolute requirement for the simulation performance of a model for board level simulations is not easy, we want to avoid unnecessarily slow or cumbersome implementations.

We based the guidelines presented next on our experience and on preliminary results from an ESA activity. This is not an exhaustive list of issues to address when tuning a model for simulation performance, and each suggestion might not apply to all situations and simulators. The best advice is to use common sense in cases of uncertainty. A good way to choose between two approaches is to simulate both and then select the most efficient one. Developers should base the stimuli for such comparative simulations on possible and probable model inputs. Different simulators have different performance characteristics; to obtain a complete picture, developers should perform simulations with all usable simulators.

Many rules and techniques for optimizing software (such as loop unrolling, code inlining, and so on) also apply to models with good simulation performance, since VHDL has many programming-language characteristics.[3] In general, VHDL simulators have less built-in optimization capabilities than state-of-the-art optimizing compilers for software. It is therefore often beneficial to perform optimization manually at the source code level. In addition, calls to subprograms declared in packages are difficult for the analyzer to optimize, since the analyzer may reanalyze the package body without reanalyzing the code making the call. Model developers could therefore manually replace subprogram calls by in-line coding in the source code to optimize a model for simulation performance.

Standard packages, such as IEEE.Std_Logic_1164[4] and IEEE.Vital_Timing,[5] are sometimes accelerated for simulation performance. But since this is not always the case, it may be necessary to use other subprograms when simulation performance is an issue.

Only execute code when necessary. A fundamental rule when modeling for simulation performance is to execute code only when necessary. Therefore, such models should use conditional statements to reduce unnecessary code execution. An outer conditional statement should reduce the necessity of evaluating enclosed conditional statements. We accomplish this with nested if and case statements, ordering them so that the highest probability branches execute first and minimizing the complexity of conditional expressions. We can assess the efficiency of the conditional-statement ordering with a code coverage utility and then often improve performance by reordering the code or modifying the structure of conditional statements. We may also identify redundant or unnecessary code.

Preliminary results showed that a signal assignment is between one and two orders of magnitude more costly in simulation time than reading a signal or variable in an if statement. This suggests that it is possible to use large structures of if statements to prevent unnecessary signal assignments and still gain in simulation performance.

Regions of related code, such as timing and unknown checkers disabled via generics, can benefit from placement in separate processes. We use generate statements to prevent them from executing when not necessary, as Figure 3 shows. This is better than placing such functions in a process and enabling or disabling them with a conditional statement, since the model will still invoke the process for each event on signals in the sensitivity list. The generate statement around the process will exclude the process from the simulation when it's disabled and thus eliminate its cost when not in use. However, this approach will not always reduce the memory usage, indicating that it does not completely exclude the code.

```
TimingGenerate : if TimingChecksOn generate
    TimingCheck : process(Clk, D, CS_N, RW_N)
    begin
        VitalPeriodPulseCheck(...);          -- CS_N width for write access
        VitalSetupHoldCheck(...);            -- D setup & hold w.r.t. CS_N
    end process  TimingCheck;
end generate TimingGenerate;
```

Figure 3. *Enabling and disabling of a process with a generate statement and the generic TimingChecksOn.*

Use processes wisely. Each process invocation has a cost in terms of simulation performance, thus we should keep the number of processes small.

Other literature suggests that sensitivity lists have better simulation performance than wait statements, since an analyzer can more easily optimize a sensitivity list than a wait statement. A wait statement would require more handling by the simulator kernel since the core monitors signals listed in such statements only after suspending the process at that particular statement. The overhead of enabling and disabling the signal monitoring would thus not be necessary for a sensitivity list. The simulator would always suspend and resume a process from the same statement, that is, at the end of the process. Even using only one wait statement in the process does not ensure that the analyzer will implement the process in the same way as one with a sensitivity list. Moreover, for the same reasons, sensitivity lists would thus be more suitable than wait statements for optimization by the simulator. However, preliminary study results have not shown any substantial difference in simulation performance between the two constructs.

There is also the question of whether to use sensitivity lists or wait statements when tracking where to resume the simulation of a process after its suspension. With a sensitivity list, we must provide a structure of conditional statements to ensure that the simulation will resume at the relevant code location, since the process execution will always resume at the begin statement. On the other hand, using multiple wait statements will inherently track the relevant code location, since the simulation of the process will resume just after the wait statement. This trade-off is not uncommon when developing board level simulation models written on high abstraction levels and is well worth considering when selecting a modeling approach.

When using sensitivity lists, we should group functionalities sensitive to the same signals in the same process. This reduces the number of processes to invoke for each signal event. Following this approach, we would group all functions related to the same clock region in one process. Functions related to different clock regions should be located in different processes to avoid invoking the process for each event on the unrelated clocks. Including only necessary signals in the sensitivity list minimizes process invocation. Finally, simulators treat most concurrent statements in the same way as a process, and they incur penalties when used.

Minimize the number of signals. Code optimized for simulation performance should use variables instead of signals wherever possible, since each signal requires specific handling (event scheduling) that takes more memory storage and instructions to execute. In fact, preliminary results show that signals contribute significantly more to the simulation time than variables do (see Table 1). For these simulations we used simulators based on native code generation, interpreted VHDL, and VHDL translated to C and then compiled.

Optimized code should only use signals for communication between processes. We could substitute VHDL 93 shared variables for signals, but should not do so, since they could introduce indeterministic behavior into the simulation. Merging processes also reduces the number of signals used for

Table 1. Ratio between simulation time for signal and variable assignments.

Data type	Native code	Compiled C	Interpreted VHDL
Bit	50	200	2
Std_ULogic	100	200	7
Bit_Vector, 32-element width	7	60	10
Std_ULogic_Vector, 32-element width	7	60	10
Integer	60	200	7

communication. We should avoid signal-generating attributes such as 'Stable, since a scheduler must also handle implicit signals and instead use the 'Event attribute where possible.

When moving a concurrent signal assignment into a process, model developers must ensure that it is not updated more often than it would have been as a concurrent assignment. Developers should avoid reassigning a signal its current value, since each such assignment requires scheduling a transaction for that signal. It is best to avoid recalculating a signal value expression too often when moving the signal assignment into a process (say, calculating only when the relevant input signals change rather than for each clock cycle). Similarly, removing static signals that seldom change will not significantly improve performance and may even decrease it.

Select the right types. Numerical data types such as integer normally result in better simulation performance than arrays such as Std_Logic_Vector and Bit_Vector. We could use numerical data types for extensive calculations directly using the integer instructions of the host machine's processor. However, when a simulation requires the bit field information (as during instruction decoding in microprocessor models), we must use numerical data types with care. Retrieving such information from an integer could be more costly than using an array in the first place. We must assess the cost trade-off between directly performing calculations on Std_Logic_Vector or performing type conversions between Std_Logic_Vector and integer accompanied by subsequent integer calculations.

Preliminary findings suggest that converting a Std_Logic_Vector signal to an integer variable, adding two integer variables, and converting the result to a Std_Logic_Vector variable can be as fast as performing an addition between two Std_Logic_Vector signals using unaccelerated add subprograms. It will be fruitful to readdress this issue in the future because of the ongoing work to define a standard arithmetic package, which will provide future simulator optimizations of arithmetic operations on Std_Logic_Vector data types.

In general, a model should perform type conversion and checking for unknown values on inputs only when it uses the data. However, adopting the most efficient modeling strategy requires knowledge of whether an input changes frequently but is seldom read by the model, or if it seldom changes but is often read. A model should convert the latter signal (a static mode pin, for example) on each signal event. A model should convert a signal that often changes but is only read under certain conditions (such as a data bus) only when it actually needs the value.

This also avoids unnecessary assertion reports, since the model checks for unknown values only when it uses them. For example, a conversion between Std_Logic_Vector and integer combined with a check for unknown values on data and address buses are only necessary when the model reads the values. This scheme has better simulation performance than if each signal is converted upon each occurrence of an event—such as a transition between valid data, invalid data, and high impedance on the bus.

Enumerated types have better simulation performance than array types, especially for coding finite-state machines using case statements. In such statements, Bit_Vectors and Std_Logic_Vectors are slower than enumerated types by up to an order of magnitude.

Passing large data structures to subprograms as parameters decreases simulation performance as the data structure size increases. This is another consideration when deciding whether to represent data as Std_Logic_Vector or integer.

We should avoid using resolved types internally in the model, since calculations for the resulting value may call a resolution function for each event on the driving signals. Using unresolved types instead could increase simulation performance. We should therefore use resolved types only when we need the resolution function. Of course, this does not apply to variables, since they need no resolution function and thus cause no difference in simulation performance.

We have observed no significant difference between using Std_ULogic and using Std_Logic. Some simulators ensure that they do not invoke the resolution function for signals with only one driver. This is analogous to replacing such signals with their unresolved base type (for example, Std_Logic signals with one driver become Std_ULogic signals). An additional benefit of using unresolved types is the automatic detection at analysis time of unwanted short-circuit connections between signals, since a signal of an unresolved type can only have one driver.

Optimize large data structures. When modeling large memory elements, we must consider the host machine memory actually used by the simulator. This is because the cache hit ratio will decrease with more memory use and cause a consequent decrease in performance. It is not the size of the memory allocated by the simulator that is critical, but the size and distribution of the frequently accessed memory. Since memory allocation differs among simulators, operating systems, and platforms, it is difficult to determine a method's impact on simulation performance. In general, we recommend minimizing the memory usage; from that point of view, signal declarations cost more than variable declarations (see Table 2).

We should compare actual memory usage to the theoretical best value. For example, an 8-element-wide register would require at least 4 bytes of memory if modeled as a Std_Logic_Vector, since 4 bits would represent each element, covering all nine Std_Logic values. Some simulators represent each Std_Logic element as a character, which uses 8 bytes for the same register. An integer could also represent the same register contents; it would require 4 bytes in most simulators. Although this is less than what the Std_Logic_Vector representation uses, an integer cannot represent all nine Std_Logic strengths. Memory usage measurements for the two approaches together with the level of detail for the data representation should indicate how to model such registers and larger memories.

Optimize expressions. The analyzer cannot evaluate globally static constants, such as deferred constants, at analysis time. It is possible to work around this by declaring a local constant that is computed once during the elaboration from the deferred constant or constants. For example, let "constant LocalTpd: Time := GlobalTpd/2;" where GlobalTpd is a deferred constant.

Table 2. *Ratio of memory usage for array elements declared as signals and variables.*

Data type	Native code	Compiled C	Interpreted VHDL
Bit	3	30	30
Std_ULogic	3	30	30
Bit_Vector, 32-element width	100	20	20
Std_ULogic_Vector, 32-element width	100	20	20
Integer	20	10	10

```
Result0 := A+B*C;
Result1 := D–B*C;
(a)

Temp0  := B*C;
Result0 := A+Temp0;
Result1 := D–Temp0;
(b)
```

Figure 4. *Common term in two expressions expressed as a temporary variable: original **(a)** and optimized **(b)** code.*

```
Temp1 := A+B;
Temp2 := C–D;
Temp3 := E mod F;
Result2:= Temp1/Temp2*Temp3;
(a)

Result2:= (A*B) / (C*D)*(E mod F);
(b)
```

Figure 5. *Unnecessary temporary expressions merged into one: original **(a)** and optimized **(b)** code.*

Some simulators do not optimize expressions for common terms, and we must therefore manually optimize the source code, as Figure 4 shows.

On the other hand, unnecessary use of temporary expressions could reduce simulation performance, since each temporary variable assignment has a certain cost (as Figure 5 shows). For calculations performed with temporary signals instead of variables, the penalty is worse.

Since VHDL specifies short-circuit Boolean evaluation, terms that would short-circuit an expression evaluation should occur as early as possible in the expression. VHDL specifies short-circuit evaluation for Boolean and Bit types. Logical operators (**and**, **or**, and so on) do not support short-circuit evaluation for Std_Logic, but we can accomplish this in the manner illustrated by Figure 6. Signals A and B are of type Std_Logic, but each expression within parentheses results in a Boolean value. The **or** operator will thus benefit from short-circuit evaluation if its left **or** operand is true.

Evaluate simulation performance. We should evaluate a model's performance during its development to identify simulation bottlenecks and improve the code. Stimuli generation should not greatly influence performance measurements, but still reflect realistic simulation scenarios and execute large portions of the model. Appropriate stimuli are crucial in analyzing simulation performance.

A code coverage utility is useful for identifying simulation bottlenecks. Such tools report the number of times each statement has been executed during a simulation, identifying frequently executed statements.

Measure the cost of VHDL statements. Clearly, it is difficult to provide

```
signal A, B: Std_Logic;
...
if (A='1') or (B='0') then
    ...
end if;
```

Figure 6. Example of an expression benefiting from short-circuit evaluation.

modeling guidelines that always optimize simulation performance because of the differences between VHDL simulators. Writing efficient VHDL models requires knowledge of which constructs are fast and which ones to avoid. While experienced model writers often have a feel for efficient coding, little exact information is available. The model writer therefore needs a cost list ranking VHDL statements in terms of simulation performance and memory usage. Having such information based on quantitative measurements and not on assumptions can help the model writer choose among different coding variants for the best performance.

As a first step toward such a list, an ESA-supervised study measured the cost of several basic, low-level VHDL constructs, such as variable and signal assignments, processes with sensitivity lists or wait statements, and so on. At this time, only preliminary results are available, some of which we have just presented.

Practical example. When developing an ASIC or other component using VHDL and synthesis, the objective is the resulting component and not the VHDL model. Developers often split the model into parts allowing several designers to work concurrently on separate parts. To facilitate both design and verification, designers further break each part into manageable-size code chunks. They target optimizations to the synthesis results, reducing area and/or increasing the operating speed. Sometimes designers also add functionality for testability reasons. Since it is not the primary objective of these models, the resulting simulation performance is often far below that of a model coded for optimal performance. While acceptable for simulations during development, optimal simulation performance is necessary for board level simulation.

Table 3. Simulation time before and after optimization for simulation performance.

	Simulation time (hours)		
Optimization type	Native code	Compiled C	Interpreted VHDL
Initial (no optimization)	1.7	2.5	8.2
Remove signal renaming	1.6	2.4	7.2
Merge clocked processes	1.2	2.0	6.5
Merge clocked-process code	1.0	1.3	5.5

We would like to automatically transform a VHDL RTL model into a functionally identical model with better simulation performance. With an eye toward that goal, we performed a series of experiments on a model from an ASIC development activity. The model's characteristics included

- split processes for state machines (state and next state in different processes) in a synthesizable, RTL, VHDL model
- 17,000 lines of code (including comments) and 5,000 executable lines (not including declarations, netlists, and so on)
- two hierarchy levels with one top-level entity and eight instantiated subcomponents
- 66 processes (39 clocked) and 52 concurrent signal assignments (only to rename signals)
- 670 signals mainly of Std_Logic and Std_ULogic_Vector type and no variables
- 17,000 gates in an ASIC for a complex protocol machine with low-speed serial data (50 Kbps) using a 4-MHz clock

We performed simulations (see Table 3) using simulators based on native code generation as well as interpreted VHDL and compiled C; the latter two are different modes of the same product. The simulated time was 1.4 seconds, which corresponds to 5.4 million simulated clock cycles. All times are wall clock time measured on a not otherwise loaded SparcStation 10/71 (117 SpecInt92) and include the time for the test bench. We made multiple measurements for each case to eliminate random measurements. The following describes the most significant optimization steps.

Remove signal renaming. Of 670 signals, the original model developer introduced 52 to change the name of signals or their accessibility, that is, to circumvent VHDL's inability to read a port of mode **out**. Using the same name for a signal in the whole model and introducing ports of mode **inout** allowed us to remove these unnecessary assignments. Although we removed less than 10% of the signals, the impact on simulation performance was measurable.

Merge clocked processes. We removed the hierarchy so a single architecture contained all processes. We then merged all 39 clocked processes into one, but permitted each code region to retain its surrounding clock edge conditional statement "if Rising_Edge(Clk) then." The increase in simulation performance is nevertheless significant, most likely due to the high clock-to-data ratio.

Merge clocked-process code. In this step, we replaced the 39 clock edge conditional statements with one, enclosing the code from the original 39 clocked processes. This basically eliminated 420 million calls to the Rising_Edge function.

In this experiment, performance increased by up to a factor of two, even without spending any effort on optimizing several processes and signals. Models including each state machine in a single process, rather than splitting them into two parts (as in this case), would benefit more from this optimization. The modest improvement for the interpreted-VHDL simulation probably stems from the interpreter overhead, since these code transformations have only a marginal effect on the amount of pseudocode to interpret.

We implemented these optimization steps in a way allowing automatic VHDL model translation according to a set of transformation rules. It is possible to implement such a tool as an VHDL-to-VHDL translator or as one optimization step in a simulator. The latter would of course be preferable since it would benefit all users.

ASSP example. One of ESA's contracting companies modeled a spacecraft telemetry chip at several abstraction levels, and we monitored its simulation performance throughout the development of the different models. The designers used RTL Verilog to design this ASSP, although many blocks resembled functions such as flip-flops and low-level multiplexers. They then translated the design to VHDL using semiautomatic conversion so that ESA could independently verify the design on a VHDL simulator.

They then developed a model for board level simulation for the ASSP following the ESA guidelines. The model comprised multiple processes representing the major component blocks.

In parallel, we developed a model that captures the ASSP's synchronous functionality in one process and all asynchronous interfaces in another. We further optimized this model by integrating a memory under the ASSP's exclusive control into the process containing synchronous functions. This removed any overhead from signal assignments for memory interfacing and also excluded the memory model from the simulation.

The increase in simulation performance between the different abstraction levels was remarkable (see Figure 7). The model for board level simulation was approximately 100 times faster than the RTL model, a performance increase that makes board design simulations feasible. The RTL model was only two times faster than the gate level model run on a conventional non-VHDL simulator. This model, however, was at a low abstraction level, and its translation from Verilog was perhaps not the most efficient. An RTL model written directly in VHDL should normally be much faster than the gate level model and not fully so much slower than the board level model. The model with only two processes and the integrated memory was five times faster than the board level model. Normally, spacecraft electronics incorporate four to eight ASSPs of this type per board; consequently, the model's performance greatly affects total simulation time.

Verification

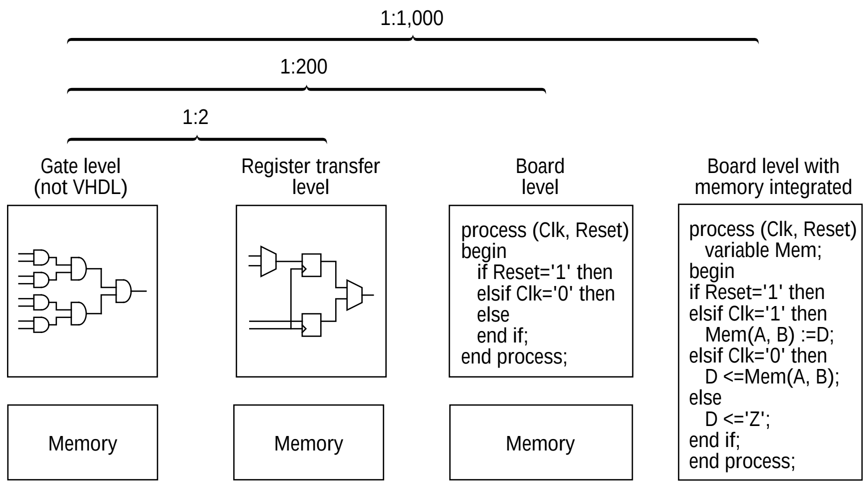
We verify models for board level simulation to ensure that the model correctly represents the actual hardware's functionality and timing. We based the method outlined here on the availability of actual hardware or a low-level design representation (such as a gate netlist) for comparison. When these are not available, we base verification on data sheet or similar information, though this increases the potential for errors and misunderstandings regarding the functionality. In this case, we emphasize performing verification independently of model development.

We have identified two categories of verification. The first ensures that the model has the same functionality as the actual hardware and occurs during model development. The second verifies that the model works for a certain combination of simulator and platform. Users perform this work with test stimuli provided by the model developer.

We should perform the first category of verification at the end of model development and include all functional test stimuli used during component development. As a last verification, we place the model in a typical board design and simulate it to detect problems not covered by other test stimuli. The set of test stimuli should serve as a maintenance vehicle for verifying the model after modifications.

Users perform the second category of verification when installing the model in their particular simulation environment. They should use one or more test stimuli provided by the model developer, reporting whether the model passed or failed the test. Users do this after installing a model because of differences between VHDL simulators and between different releases of the same simulator. In addition, users tend to first suspect the models when a problem occurs during simulation. Users are more likely to employ test benches that support automatic verification and require minimum involvement on their part. Such test benches may reduce the problems reported to model developers or distributors, eliminating problems not related to the model itself.

After the model's delivery, the user must verify that the model's exact behavior remains unchanged. A useful method involves sampling the outputs on the test multiple input signature register (MISR). We compare the result to



```
process (Clk, Reset)
begin
  if Reset='1' then
  elsif Clk='0' then
  else
  end if;
end process;
```

```
process (Clk, Reset)
  variable Mem;
begin
if Reset='1' then
elsif Clk='1' then
  Mem(A, B) :=D;
elsif Clk='0' then
  D <=Mem(A, B);
else
  D <='Z';
end if;
end process;
```

Figure 7. *Example simulation performance acceleration due to increase in abstraction level for an ASSP.*

a predetermined signature to check whether a model has passed or failed a test. Compressing the output data eliminates large reference files.

Regression tests performed during model or ASIC development could also use this approach. Since such data compression is most feasible for synchronously sampled data, it is better to use these regression tests for test benches verifying functional behavior on a per-clock-cycle basis.

The MISR needs sufficient stages to allow for long test stimuli without significant risk of error masking. A primitive binary polynomial implementing a maximum-length linear feedback shift register provides the best structure. Detailed information on these shift registers, the underlying theory, lists of polynomials, and so on are available in Bardell et al.[6]

Quantitative measurement methods such as code coverage and gate level fault coverage can evaluate a test stimulus' efficiency. A code coverage utility can help verify that a test stimulus executes every line of source code in a model, but does not tell how much of the model functionality the stimulus covers. We can better approximate the functional coverage of a test stimulus. We do this by applying the same stimulus to a fault simulation of the gate level model and calculating its fault coverage. This is of course only feasible when a gate level model exists.

During the fault simulation, the test stimulus should not activate functions used for internal testability, such as built-in self-test, when the test object does not fully implement them. Testability logic could cover faults without these parts being verified for functionality. This would result in fault coverage not corresponding to the functional coverage.

Verification's purpose should always be to ensure the model's complete functionality, and not solely to satisfy code and fault coverage goals. These are merely inexact measurements of the verification's efficiency.

Maintenance

For five years, ESA has received VHDL models developed by contractors. These models helped us identify new error types and unusual coding styles to avoid. Although we could regard some of these as minor coding anomalies, they create long-term maintenance and code reuse problems, among others. Such problems become significant in handling a large number of models from many different companies. Thus, in situations where we employ VHDL for portability reasons, modeling guidelines are essential.

Before creating ESA-specific guidelines, we sought existing modeling guidelines. Unfortunately, the guidelines we found were not very useful, containing mainly general and high-level requirements. Their scope is often too vast to be practical, for example, covering both modeling issues and requirements for electronic data sheets. Such guidelines would require each user to spend significant effort in requirements analysis and developing technical solutions. In addition, the design community seems to not have widely accepted any of these guidelines, despite their several years of existence.

We issued the ESA *VHDL Modelling Guidelines*[7] in September 1994, after a review by companies working with the agency. ESA intends this as a requirement document for contracts involving VHDL modeling. In line with ESA's philosophy of not mandating specific design tools, companies may use our guidelines with most VHDL simulators or synthesis tools. The only firm requirement is that the tools must support IEEE.Std_Logic_1164 types. Figure 8 shows a component declaration that complies with ESA guidelines.

As a complement to the *VHDL Modelling Guidelines*, we have prepared *VHDL Models for Board-level Simulation*.[8] While the guidelines largely contain requirements, this document shows how to implement the guidelines in a practical way, explaining underlying issues and trade-offs. It focuses on VHDL models for board level simulation. But many techniques are useful in other types of developments, such as ASIC verification.

ESA's Web site (http://www.estec.esa.nl/wsmwww) includes more information on microelectronics and VHDL simulation.

Distributing models

In 1996, ESA will have received about 20 VHDL models of complex ICs for spacecraft electronics. A logical step would be to actively introduce board level simulation to the companies working with ESA. Not only will board level simulation improve design quality and reduce development schedules, it will also allow new companies more opportunities to design spacecraft electronics, thus increasing the competition among equipment suppliers. The major task is ensuring the availability of simulation models.

The market for simulation models of space-specific components is too small to be of economic interest to established companies. Therefore, our strategy is to reuse the VHDL models developed during component design. Using VHDL greatly reduces the effort of supporting several platforms and simulators, since VHDL models require only minor modifications (if any) for each new simulator. Commercial models also follow this trend; as an example, Synopsys Logic Modeling offers VHDL models of lower complexity components.

Commercial efforts have often used C models for complex components for historical reasons and claimed performance gains. However, the scheduling cost for a model's external signals is the major limiting factor for simulation performance. Until simulator vendors introduce significant improvements in signal scheduling, there should be little difference in simulation performance between VHDL- and C-based models running on a VHDL simulator.

Distributing VHDL source code would require a minimum effort since the user could be responsible for adapting code to different simulators. However, such an approach would be unacceptable because it does not protect design information. Specifically, the company that designed a component is understandably unwilling to make that information available to its competitors, especially in cases of synthesizable VHDL code. Furthermore, the availability of VHDL source code would encourage redevelopment of similar devices. This in turn would increase costs for ESA as well as significantly decrease the interest in foundries to support devices such as ASSPs.

Source code encryption or scrambling could achieve an initial level of protection. This technique requires removing comments, replacing identifiers with meaningless names, and removing code structures such as indentation. Although the encrypted source code is difficult to read directly (see Figure 9), an automatic process could increase its readability. However, the model is still identical, that is, a synthesizable model will remain synthesizable. As an experiment, we distributed encrypted VHDL source code along with a nondisclosure agreement. While encryption tools are commercially available,[9] we developed a simple tool in house for this experiment.

The solution we envision for future, large-scale model distribution is to supply the models in an analyzed format, supporting those simulators that sufficiently protect the VHDL code. Further, it must be possible to remove most of the remaining source code information. To increase the protection level, we could combine the analysis with source code encryption to rename units and signals internal to the model. In specif-

```
component BitMod
   generic (
      SimCondition:       SimConditionType := WorstCase;
      InstancePath:       String           := "BitMod:";
      TimingChecksOn:     Boolean          := False;
      tperiod_Clk:        TimeArray        := tperiod_Clk;         -- TClk
      tpw_Clk_posedge:    TimeArray        := tpw_Clk_posedge;     -- TCHi
      tpw_Clk_negedge:    TimeArray        := tpw_Clk_negedge;     -- TCLo
      tpw_CSN_negedge:    TimeArray        := tpw_CSN_negedge;     -- T1
      tsetup_D_CSN:       TimeArray        := tsetup_D_CSN;        -- T2
      thold_D_CSN:        TimeArray        := thold_D_CSN;         -- T3
      tpd_A_D:            TimeArray        := tpd_A_D);            -- T4
   port (
      -- System signals (2)
      Clk:       in     Std_Logic;                                 -- Master Clock
      Reset_N:   in     Std_Logic;                                 -- Master Reset
      -- Interface to internal registers (12)
      A:         in     Std_Logic_Vector(0 to 1);                  -- Address bus
      CS_N:      in     Std_Logic;                                 -- Chip select
      RW_N:      in     Std_Logic;                                 -- Read/write
      D:         inout  Std_Logic_Vector(0 to 7);                  -- Bidir. bus
      -- Serial Interface (3)
      SClk:      in     Std_Logic;                                 -- Serial clock
      Sdata:     in     Std_Logic;                                 -- Serial input
      Mdata:     out    Std_Logic);                                -- Serial output
end component;
```

Figure 8. A typical component declaration for a model complying to ESA guidelines.

```
process bEgiN WAIT on llllll1ll ; if llllll1ll = '1' AnD llllll1ll'EvEnt ANd
llllll1ll'laST_VaLUE = '0' THeN l1l1111l1 <= l11lllll1 ; IF l1l1ll1l1 = '0' thEn
l1111llll <= "00111111" ; lll1l1ll1 <= "11111111" ; ElsIf ( ll1l111l1 = '1' anD
ll1111l1l = '1' ) ThEn l111ll1ll <= l111ll1ll + "00000001" ; eNd IF ; iF ( l11l1l11l
= '1' ) THEn ll11llll1 <= l1111llll ; eNd if ; ENd iF ; EnD iF ; eND pRocESS ;
```

Figure 9. Encrypted VHDL source code.

ic cases, we could first modify the VHDL models and render them more difficult to synthesize, for example, by eliminating hierarchy and merging processes. A side benefit would be increased simulation performance.

Ongoing work seeks to demonstrate the feasibility of distributing analyzed VHDL models. We have created scripts that automatically analyze a VHDL model for a specific simulator version and purge unnecessary source code information. We then use automatic test benches to verify the analyzed model without manual inspection.

ESA WILL CONTINUE to receive VHDL models of ASICs, ASSPs, and some standard components that contractors develop under ESA contract. The next major step will be to establish a viable distribution scheme for VHDL models for board level simulation. Major challenges are

- intellectual property issues and the related issue of preserving the market for ASSPs and standard components, which an easily synthesizable VHDL model could destroy
- the relatively small market, estimated to be less than 30 customers
- customer expectations for high levels of maintenance and bug correction coupled with an unwillingness to pay more than a minimum price for the service
- market size and expectations, and finding a suitable distribution company

The final decision on whether to launch a service for this type of model availability will depend on many factors. In parallel with and as a complement to offering simulation models of ASSPs and standard components, synthesizable cores or super cells will form an important part of increasing the effectiveness of ASIC design. D&T

Acknowledgments

Although not mentioned by name, we thank those companies we worked with that performed the basic work referenced in this article. We also thank all those discussing simulation performance in their papers and on the Internet from whom we have gathered input for our work.

References

1. C. Hein et al., "RASSP VHDL Modeling Terminology and Taxonomy—Revision 1.0," *Proc. Second Annual RASSP Conf.,* 1995; http://rassp.scra.org.
2. T. Egolf et al., "Experiences with VHDL Models of COTS RISC Processors in Virtual Prototyping for Complex System Synthesis," *Proc. VHDL Int'l Users' Forum,* 1995.
3. O. Levia, "Writing High Performance VHDL Models," *Proc. EuroVHDL,* IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 119-127.
4. *IEEE Std 1164-93, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164),* IEEE, Piscataway, N.J., 1993.
5. *IEEE Std 1076.4-95, IEEE Standard VITAL ASIC Modelling Specification,* IEEE, 1995.
6. P.H. Bardell et al., *Built-In Test for VLSI: Pseudorandom Techniques,* John Wiley & Sons, New York, 1987.
7. P. Sinander, *VHDL Modelling Guidelines,* European Space Agency, the Netherlands, 1994, ftp://ftp.estec.esa.nl/pub/vhdl/doc/ModelGuide.ps.
8. S. Habinc, *VHDL Models for Board-level Simulation,* European Space Agency, the Netherlands, 1996, ftp://ftp.estec.esa.nl/pub/vhdl/doc/BoardLevel.ps.
9. K. O'Brien and S. Maginot, "Non-Reversible VHDL Source-Source Encryption," *Proc. EURO-DAC'94 with EURO-VHDL'94,* IEEE Computer Society Press, Los Alamitos, Calif., 1994, pp. 480-486.

Sandi Habinc is a microelectronics engineer in the Microelectronics and Technology Section of the European Space Research and Technology Centre. He holds an MS in computer science and engineering from the Chalmers University of Technology, Gothenburg, Sweden.

Peter Sinander is leading the work in the Microelectronics and Technology Section at the European Space Research and Technology Centre. He holds an MS in electrical and electronic engineering from the Chalmers University of Technology.

Direct questions concerning this article to Sandi Habinc, Microelectronics and Technology Section (WSM), European Space Research and Technology Centre, Postbus 299, NL-2200 AG Noordwijk, The Netherlands; sandi@ws.estec.esa.nl.