



**Behavioral Modeling in VHDL Simulations -
The Benefits of Higher Levels of Abstraction in Complex Simulations**

Conference Presentation
Lattice FAE
DesignCON, 1999

Presentation Introduction

Note: This paper was originally prepared for a presentation given at PLDCon '99. The format of the paper follows the structure of the foil presentation that was given at the conference. The section headings actually represent the headings that were used in each foil. This allows you to follow the presentation with graphics from the presentation foils where appropriate.

Objectives

The objective of this paper and the associated presentation are to:

- Identify the basic requirements for simulating programmable logic designs
- Show common mistakes made in simulation
- Demonstrate how simple test bench techniques exacerbate many errors
- Propose methods based on behavioral modeling that provide superior simulation capabilities
- Show how VHDL is well suited for behavioral modeling
- Demonstrate an example project based on modeling

It is not the purpose of this presentation to be an exhaustive treatise on methods and techniques, but a review of the code for the included example will go a long way toward instructing the motivated reader.

Agenda

The agenda of the presentation includes the following:

- Acknowledgements
- Overview of Simulating FPGA and CPLD Designs
- Survey of Common Methods
- VHDL Features that Support Modeling and Abstraction
- Designing a Test Bench using Models
- Evaluating the Benefits
- References

Acknowledgements

I would like to thank the following individuals and acknowledge their contributions to this effort: from Vantis: Doug Hardman, Andy Robin, Tim Holland, Bob Klein and Kayla Kurucz; from the Ramix Corporation: Saeed Karamooz, who allowed me to use his Mach CPLD design as an example for this paper; and finally my wife, Robin, whose patience and encouragement enabled me to write this paper.

Overview of Simulating FPGA and CPLD Designs

Reasons to Simulate

Improve product quality and reliability

Controllability and observability are two key factors in enabling proper testing of a design. Many times a simulation can test situations that are difficult to generate in a prototype debug environment. Boundary conditions can be easily checked when you have complete control of the environment. Worst case conditions may include the simultaneous occurrence of independent functions that are very difficult to achieve in an actual system.

These conditions can be easily made to coincide in a simulation that is completely controllable. In addition to having more control over the circuit, simulation also enables more complete observations of the design. Internal signals are available regardless of whether there are spare I/O pins on the device. With a few keystrokes, a 64-bit data bus can be probed. Try that with a logic analyzer! It could take the better part of a day in the lab to get a complex design instrumented for monitoring.

Decrease time-to-market by shortening the debug and integration cycle

The same of simulation advantages of controllability and observability allow errors to be caught more quickly and easily than with hardware debugging. Even more important, the design can be changed much more quickly in a simulation than on a hardware prototype. This shortens the debug cycle and makes integration easier, especially if the different modules have been simulated together before integration.

Reduce the risk

Not only is the debug cycle decreased when simulation is used, the number and magnitude of the design changes decreases as well. This decreases the chance that the changes to the hardware will be extensive enough to require a new printed circuit board design. Turning a new PCB easily adds weeks to a schedule. When pushing performance and density limits, confidence may be low that the available programmable devices will perform the desired functions. Simulation can answer these questions before expending the time and expense of building a hardware prototype.

When a design breaks new ground, it may not even be known early in the design cycle whether or not the basic design approach is feasible. Simulation is an inexpensive way of determining this **IF** the simulation techniques are powerful enough to simulate the system level aspects of the design.

Reasons Engineers Avoid Simulation

Lack of discipline or poor habits

Many engineers love to tinker. These will often forgo a rigorous analysis of design assumptions because they feel more comfortable tweaking a design into submission. Early SRAM FPGAs required tweaking because the timing models were so variable. Rigorous analysis up front was difficult to do and difficult to achieve in the actual devices. The "blow and go" methodology was developed into a high art in those days. With new FPGA families like VF1 from Vantis, deterministic, delays for high-speed designs are achievable and dependable. The development tools that Vantis and some other vendors have brought to the market also are now supporting ASIC-like methodologies that support rigorous analysis, modeling and simulation.

Difficulty in justifying simulation phase to management

Many managers do not understand the benefits of simulation. They are unwilling to allow enough time in a schedule for something that they do not understand. This can be a difficult situation, but it is better to have an engineer that understands the importance of simulation, even if management requires some education. It is helpful to utilize methods that increase productivity in simulation to enable more to be done in less time.

Difficulty in generating successful simulations

Some engineers are trapped by the low-level methods of antiquated simulation techniques, and they have never made the jump to successfully simulating complex systems efficiently. By using modeling techniques at higher levels of abstraction, they can simulate their designs more efficiently and achieve their objectives more easily.

Common Simulation Mistakes

Using simple methods for complex designs

Simple methods become tedious and ineffective

As designs become more complex, more powerful simulation techniques are needed because they actually decrease the complexity of design verification. A wise engineer once said that there are only two methods for handling complexity, hierarchy and abstraction. Modeling techniques make use of both of these to allow the simulation of a complex system to be both manageable and efficient.

Low fault coverage

Low fault coverage allows design errors to be missed during simulation. One reason is that the test case was not considered properly during the planning stages. While this is more of a specification issue than an implementation issue, more efficient simulation techniques enable an individual or team to be more aggressive in specifying test scenarios. Another cause of low fault coverage is that a test scenario may be considered, but not properly implemented. When low level methods are used which do not adequately handle complexity in the design, the engineer can lose the forest in the trees.

Not qualifying circuit responses properly

As a design increases in size, it becomes more and more difficult to characterize the behavior of the circuit by visual inspection of the waveforms. Unless the test bench validates circuit responses automatically, the burden for finding each discrepancy rests with the engineer. While this can work well for simple circuits, as designs get more complex, the engineer is no longer capable of keeping up with all the vectors that get generated.

Goals for a Good Test Bench

Completeness

High fault coverage is essential for an effective simulation project. A good simulation strategy will facilitate the development of a test bench that covers all the pertinent operational scenarios. This is best achieved by using techniques that allow the simulation designer to work at a fairly high level of abstraction. It is easier to follow the flow of a simulation if you can specify the individual steps at a macro level rather than at the atomic level. "Perform this operation" is more easily understood and validated than is "Twiddle these bits". A good simulation technique should support a high level of abstraction. VHDL, with many features that support behavioral modeling at a high level, easily satisfies this requirement.

Ease of use

It almost goes without saying that a simulation method should be easy to use. There is a trap here, however. The methods that are easy to use for a small design do not scale well to a larger design. Modeling allows you to build a "virtual system" to check out the design under test ("DUT") by exercising it in a realistic operational environment. This has some overhead that makes it impractical for small projects, especially if models are not yet built. However, if verified models are already available, even small designs can be done efficiently with the virtual system approach.

Flexibility

A good simulation methodology must be flexible to allow changes to the simulation as knowledge is gained or as requirements change. Object oriented techniques offer this kind of flexibility to programmers. Embedding functionality into system level components has some similar benefits for the engineer who designs a simulation of a complex chip or system.

Reusability

Design reuse is becoming a major driver in closing the productivity gap in deep submicron VLSI design methodologies. High density FPGAs and CPLDs that promise fast time to market will require similar techniques if these goals are to be realized.

Can Modeling Meet the Goals?

These goals are largely complementary. A simulation strategy that is easy to use makes completeness easy to achieve. A flexible implementation method is generally easy to use. Flexibility generally helps reusability, allowing components to be used in a variety of environments. Reuse of familiar components has a tremendous impact on ease of use. Modeling techniques should meet all these goals.

High levels of abstraction should aid focus on completeness

Simulation methods based on modeling of system level components naturally work at a high level of abstraction. Having a model of a CPU allows the types of bus cycles to be specified. If the model is complete, then C or assembly language code could actually be used to drive the simulation. This allows the engineer to focus on the functional requirements of the test, leaving the twiddling of bits to the hardware model.

Benefits should outweigh effort of model generation and validation

The benefits of using models should outweigh effort of model generation and validation. After a brief survey of modeling techniques, the reader will be able to decide if this is the case.

Modularity and encapsulation of components should help flexibility

Properly designed components should be mimic the system and should be able to be modified as the system is modified. Proper modeling should yield very flexible simulation architectures.

Behavioral models should be reusable like integrated circuits

Components imply reusability if they are made to be generic. The hardware components that are typically used in an electronic board or system are made to be as widely useful as possible. It will be easy to reuse the VHDL components that model these same hardware devices.

Survey of Common Methods

Typical Simple Test Bench

(Block Diagram Included in Presentation)

Review Test Bench

In a typical test bench, the structure is simple and flat. The DUT is instantiated as a component, and a single process controls the main test sequence. Circuit stimulus is generated by in-line code and multiple wait statements control the main sequence timing. Alternatively, multiple "after" conditions in a single transport statement for each signal could be used to control timing. Most novice users of VHDL simulation validate circuit responses by inspecting the resulting waveforms. The test bench does little or no qualification of circuit performance.

Simple Test Bench Coding Example

```
-- main stimulus process
  p_stim: process
  begin
-- main stimulus process
  p_stim: process begin
```

```

    csn <= '1';
    wen <= '1';
    end_in <= (others => '0');
    rst <= '1';
    wait for clk_period*2;
    rst <= '0';
    wait for clk_period;
    assert (out_bus = x"5") report "Out did not reach count 5"
        severity error;
    waitn <= '0';
    wait for clk_period*2;
    waitn <= '1';
    wait for clk_period*20;
    assert false
        report "Reached end of stimulus."
            severity note;
end process;

```

Simple Test Bench Coding Analysis

No abstraction leads to flat, difficult implementation

Because of the flat, simple structure of this test bench, the implementation is difficult for all but the simplest simulations. For example, one might want to vary the timing of control signals in order to verify that the DUT will tolerate all the allowed variation on its inputs. This can be very difficult to achieve if all the function and timing for all the input signals are put into a single large process. There is no separation between function and timing, so flexibility is compromised.

Wait statements vs. transport statements

Multiple wait statements can be used to control the sequencing of various signals. Unfortunately this cause timing of individual signals to interact and this can become quite complex. It is virtually impossible to vary the signals intelligently, for example to allow the use of min/max timing. The interaction of signals that are all controlled by common wait states would make this a terribly complex problem. One alternative is to use individual transport statements for each signal. This allows a series of delays to be specified for each individual signal. While this decouples signal timing, multiple wait statements are easier to step through during debugging than transport statements. A third alternative is to use a separate process for each signal and have them synchronized by a master process that marks the start of each cycle.

High level and low level controls are intermixed

It is difficult to concentrate on too many issues at one time. A simple structure intermixes the main test sequencing code with the intricacies of individual signal timing. This is a difficult model to control, interpret and modify. A partitioning of high level and low level code is needed for large, complex designs.

Repetitive code difficult to manage

With no abstraction, code is often cut and pasted to implement repetitive operations, as in the case of CPU bus cycles. This is very difficult to manage. A better approach is to use a loop to repeat the steps, but this is a flat model that gets complex if several functions or processes need to be repeated interactively.

Interactive code not intuitive

Often the best way to check an interface is to be interactive. If a device interface has a handshake, then it is reasonable to have a process in the test bench that responds to that handshake. The normal way to handle this is with an independent process that participates in the handshaking and throttles the transfer of data. This is efficient, but it moves away from mainline code. It is not designed in a way that structure and function are unified.

Improved Simple Test Bench

(Block Diagram included in Presentation)

One can improve this simple model slightly by adding statements that will check the responses of the DUT. As will be seen later, the simple structure associated with this kind of test bench does not support very well the addition of the code to check responses.

Improved Test Bench Analysis

Checking data automatically is a mixed blessing

Adding data checking to the simple test bench can improve efficiency by relieving the engineer of some of the tedium of checking simulation results. It is difficult with the simple test bench to have enough control to do all the checking that is required to make sure that the hardware circuit will perform the same way the simulated model does. Just sampling the data at the correct time can be difficult. Checking setup and hold times and all the other parameters of importance is a daunting task when all the code for the simulation is forced to reside in a single process or in a fairly flat model. The situation can be remedied somewhat by setting up multiple process to check signals, parameters or interfaces, but again it is an ad-hoc process, often with little guidance or standardization.

Test Bench Using Modeling

(Block Diagram Included in Presentation)

Improved Modeling Test Bench

(Block Diagram Included in Presentation)

Key Characteristics of Modeling

The basic idea behind a test bench based on modeling is to create functional models of all the board level components that surround the design under test, or DUT. The top-level file is usually composed of structural VHDL code that ties the various models together as instantiated components. One (or perhaps more) of the components is the DUT. The other components are functional equivalents of the devices that surround the DUT on the actual board. In our example, a CPLD design destined for a Vantis M4A-192/96-5 needs to be simulated. The design is an interface between a 33 MHz i960RP microprocessor and dual banks of SRAM memory. It allows the memory to be accessed at half the bus speed by alternating access to two banks of SRAM. The CPLD is the DUT, but the CPU, clocks, transceivers and SRAM must be modeled. VHDL is a modeling language that readily supports such device modeling.

Models Generate Stimulus

Since the behavior of actual components is modeled in the test bench, the models themselves generate the stimulus to the DUT. This frees the engineer from having to explicitly define each vector. Instead, the simulation is driven by high level operations that are built into the models. In our example, both the memory and the SRAM controller are slave devices. Therefore the CPU model drives the simulation. The designer needs only to specify the sequence of bus cycles with a method to route data into and out of the design. The CPU model will generate all individual bus signals. By handling the simulation at this higher level, the designer can be concerned with system level functions and let the models handle the details. Once models are generated, a complex functional simulation can be generated much faster than with low level methods.

Models Check Responses

In our earlier example, a check function could have been included in the test bench to check for address a data hold time. This checking is done better inside an SRAM model because it can be coded in one place and will be active any time the SRAM is accessed. Control path timing and function can be easily

implemented in behavioral models. Some data path functions can be tested internal to the models, as in the case of a CRC generator/checker, which can internally verify data, patterns. Other solutions can be implemented as well. A CPU model might perform write and readback cycles to verify the operation of the memory subsystem. The SRAM model might be implemented as an array, simply storing data in array locations according to the address. Another choice would be to build a model that could transfer address and memory data to a disk file for verification later by inspection or even have the model read expected values from a file and compare them on the fly to the values received from the simulation. In our example, the memory was implemented as a small array because there was not requirement to pass large amounts of data through the design. The CPU model is responsible for verifying data integrity, and the SRAM model checks for proper signal timing coming from the DUT.

Main control process may not be at the top level test bench

The main difference of the modeled test bench from the simple case is its lack of functional code. It is primarily a structural module, and the mechanism for controlling the simulation sequence is not immediately evident. In our example, the CPU is the master in the system, and it is driven by the program code. How can we control the simulation in this case? A straightforward way is to encode into the CPU model the bus operations that are required for the tests. This allows the test sequence to be specified by the necessary bus operations, writing and reading data to verify proper operation.

Using VHDL for Modeling

VHDL originally defined as a pure modeling language

VHDL was originally conceived to be a pure modeling language. This led to some difficulties early on for those who wanted to use VHDL for synthesis because many useful modeling constructs were not appropriate for synthesis. The full range of VHDL's modeling capability can be used for generating behavioral models to be used in simulation. Very complex behavior can be specified without regard to adhering to the RTL standard for synthesis. Even analog and mechanical systems can be modeled using the mathematical functions available. The analog extensions being considered now for VHDL are primarily extensions to the math functions that are currently available in the language. Because the simulation models do not have to be synthesized, many high level behavioral constructs can be used to simplify the generation of functional models. Defining functions according to the components that perform them yields a functional decomposition that encourages a higher level of abstraction. This decouples the functional specification from the low-level operational details and simplifies the task of writing the test program.

Structure Begets Structure

One beauty in getting started with a modeling test bench is that no abstract or arcane method is needed to figure out how to start. It is really quite straightforward. The structure of the hardware design is the structure of the test bench. It is not like object oriented programming which required a wholesale paradigm shift and which was challenging to programmers who had been steeped in structured programming methods. On the contrary, building a test bench with functional models requires the same analytical view than engineers have developed through years of working with schematics and lab equipment. One simply replaces the components on a board with functional components in a VHDL design, and then figures out how to model each component. Planning a simulation is much like planning the early stages of prototype debug when the hardware engineer is often the one who writes the diagnostic code to do the first checkout. Planning the control mechanism may be a little trying, because it is so different from simpler methods. But once the method for controlling the master is established, exercising the slave devices in the system is a simple matter.

In our example, the SRAM is a slave, so it merely responds to the memory controller. The CPU model is another story. It must be controlled in order to control the flow of the simulation. But it would be impractical to fully model a complete CPU. A simpler mechanism must be devised. The control functions could be encoded as actual assembly language directives, but this would be overkill and would be difficult to use. A more reasonable approach would be to implement the basic bus operations that are used to

transfer data, and encode these into special codes that could be used to control the flow of data through the system. For example, the CPU must execute single and burst mode read and write operations, so these would be the place to start. The codes can be stored in the VHDL code of the CPU model, or could be read by the model from a text file that has the instructions listed in the desired order of execution. Driving the simulation with a sequence of high level commands satisfies our need to control the simulation flow from a high level of abstraction, making it easy to focus on the system level considerations associated with covering all the pertinent test scenarios.

Behavior is Encapsulated

Localizing function enhances flexibility

The function of the SRAM model is defined only once, inside the SRAM model. The timing of the SRAM is independent of the CPU bus timing, the chip select decoder, and all the other components in the system. Therefore the other parts of the system can be modified independently without regard to the SRAM. This is a key to achieving flexibility. If the SRAM needs to be modified, its definitions are all localized, so they need to be changed in only one area. In the earlier example, timings interacted, and were spread

Model characteristics are defined independently

The functions of each model, and thus each interface, are more easily defined because they don't interact with characteristics of other models. When defining the SRAM, you need only be concerned with the details of the SRAM itself. The speed of the processor or the characteristics of the state machine are of no consequence when defining the SRAM model.

Validation of Models

Models must be validated at a low level of abstraction, looking explicitly at the detailed function and timing. It is best done by a different engineer than the one who will use the models to prevent "self-fulfilling prophecies". If one engineer develops the models, validates them, and uses them, the same assumptions or blind spots may be perpetuated throughout the whole process. Encapsulation allows the model to be validated independently of the final application. Generally speaking, if the model is correct, the model will work in any valid application.

VHDL Features that Support Modeling and Abstraction

Multiple Architectures

Why would you want to use multiple architectures for a simulation? There are many good reasons. These are just a few:

To re-implement difficult structures just for simulation

Often a structure in a design can cause problems for a simulation. Consider the case of a long counter, which requires many thousands of simulation cycles before anything interesting happens. You can instantiate a counter component that has one architecture for synthesis that would require the long timeout. A second architecture, used only for simulation, could have a much shorter timeout cycle.

To use gate level vs. behavioral models

Another use for multiple architectures could be applied to a board level simulation of multiple chips. Gate level models simulate much more slowly than do behavioral models. One could substitute behavioral models for all devices but the one of interest, and then perform simulations much faster. Once the design looks sound, the gate level models could be used.

To iterate test bench design

Often the simplest type of simulation to implement the first time is not the most efficient. In our example, the first CPU model that was developed used a simple process which had the control functions implemented directly in VHDL code. This was less efficient than a model that used a test language parser to drive the simulation, but it was simpler to bring up the first time. While bringing up the more complex parsing routine, I could keep the original model as an alternative architecture to use for comparison with the new architecture.

Attributes

VHDL has a rich set of attributes that can extract information from signals and other objects that are extremely useful in simulations, especially for checking functional and timing operation. They are universally attached to a prefix, as in the case of the familiar "clk'event". The most useful are listed below:

<i>Signal</i> 'event	Returns a true value if the signal had an event (changed value) during the current simulator tick. This is useful for synchronizing events with a strobe signal. For example, when the WR~ goes high on an SRAM, then a write cycle may have taken place. This would be an appropriate time to execute the appropriate code inside a process that models an SRAM.
<i>Signal</i> 'last_value	Returns the value of the signal prior to the last event.
<i>signal</i> 'last_event	Returns the time elapsed since the most recent event on a signal. This could be useful in an SRAM model to see if the address had been stable throughout the duration of the write cycle.
<i>signal</i> 'last_active	Returns the time elapsed since the most recent transaction (scheduled event) on a signal
<i>signal</i> 'active	Returns a true value if any transaction (scheduled event) occurred on a signal during the current simulator tick.
<i>type</i> 'image(<i>expression</i>)	Creates a text image of the specified expression. This is useful for text reports. Expression must be of the same type as the <i>type</i> to which the attribute was attached.
<i>type</i> 'value(<i>string</i>)	Returns a value of the attached type
<i>signal</i> 'delayed(<i>time</i>)	Creates a signal identical to the attached signal only delayed by <i>time</i> .
<i>signal</i> 'stable(<i>time</i>)	Creates a Boolean signal that becomes true when the signal is stable for the specified time period. This could be used for checking setup and hold times.
<i>signal</i> 'quite(<i>time</i>)	Creates a Boolean signal that becomes true when the signal is quite (having no events scheduled) for the specified time period.
<i>signal</i> 'transaction	Creates a bit signal that toggles when a transaction or event occurs on a signal
<i>type</i> 'pos(<i>value</i>)	Returns the position number of the particular value for the specified enumerated type. This is useful to generate an integer that references the value of an enumerated type.
<i>type</i> 'val(<i>integer</i>)	Returns the value of an enumerated type from the specified integer value. The first enumerated value has a position number of 0, and all subsequent values have numbers of 1,2,3... respectively.

Control Structures

The **if-then-else**, **case**, and looping structures that VHDL provides as sequential statements are extremely useful for controlling the flow of a simulation program. Remember that if-then-else statements can cause quite a bit of logic to be generated in a synthesis program, but this is of no consequence when developing models for simulation only. The if-then-else structure can model very complex behaviors in a natural style that is easy to define. For more structured conditional structures, the **case** statement should be used. The various loop structures can be quite useful for repetitive operation. One should remember that in VHDL, a process automatically restarts once it has completed (subject to whatever WAIT statements or sensitivity lists are included), so an explicit loop structure may not be needed.

Wait Statements

Wait statements are extremely useful for controlling the timing of scheduling events and assertions. The three uses for the wait statement are listed below:

Wait for (<i>time</i>)	Suspends execution until the specified time period has elapsed
Wait until (<i>condition</i>)	Suspends execution until the specified condition is true
Wait on (<i>sensitivity list</i>)	Suspends execution until an event occurs on one of the signals listed in the sensitivity list

Transport Delay Specifications

Easier than wait statements?

A transport delay specification can be used to time signal transitions in much the same way that wait statements are used. The difference is that wait statements advance the simulation time, affecting the timing of all signals together. Transport delays control the timing of a series of transitions on a single signal, independent of all other signals. A wait statement is still required to advance the simulation time, but the timing sequence specified in the transport statement will roll off the simulation event wheel as time advances. It is much easier to specify signal transitions individually with transport statements, but it is generally more difficult to unravel the timing when stepping through code during debug. An alternative is to use separate processes for each signal using wait statements. This can be easily debugged and also easily specified. This spreads the code over many processes that may be difficult to follow for a large number of signals, but the benefits may be worth it.

Transport Example:

```
SIG0 transport '1' after 0 ns
      '0' after 10 ns
      '1' after 25 ns;
```

Assert Statements

Generate messages during simulation

Assert statements check whether a certain condition is true and generates a message if the condition is not true. This enables self-checking models that will report violations automatically. They can be used to check that data is correct. Using the signal attributes, they can also validate that required specifications are met. For example, an SRAM requires data and address to be set up to the write pulse. The data must be set up to the end of the write pulse, but the address must be stable throughout. Assert statements can be used within the SRAM model to check all these conditions and to report an error if they are not met. Remember that the **'image'** attribute can be used to generate string representations of expressions that can be displayed in a message.

File I/O in 1076-1993

The file input/output functions of the 1987 standard were somewhat limited. The 1993 version added syntax enhancements and new features to extend the file I/O capabilities of the language. The file input/output functions of VHDL can be used in various ways. The primary uses are:

- Application of stimulus vectors
- Storage of response vectors
- Storage of expected responses for comparison (using **assert**)
- Formatted reports

- Reading custom test language files (which are then parsed)

This last technique can be used with a higher level of abstraction to control the simulation. Since the commands are specified at the functional level, the detailed implementation is left to the functional models, which respond to each command as they have been programmed to do. This is a very powerful technique for driving simulations at a high level. The designer's system-level knowledge is leveraged to the greatest degree because the control is exercised at the level that is most convenient for checking system function.

File I/O Commands

File_open(<i>file,name,mode</i>)	Opens file object with the specified name for a mode of read, write or append.
File_open(<i>status,file,name,mode</i>)	Opens file object with the specified name for a mode of read, write or append, and assigns value to status of type file_open_status that has a value of OPEN_OK, STATUS_ERROR, MODE_ERROR or NAME_ERROR
File_close(<i>file</i>)	Closes specified file
Read(<i>file,object</i>)	Reads from specified file into specified object.
Readline(<i>file,line</i>)	Read a line from the specified file
Write(<i>file,object</i>)	Writes specified object to specified file
Writeline(<i>file,line</i>)	Writes a line to the specified file
Endfile(<i>file</i>)	Returns boolean true value if file pointer of specified file is at the end of the file

Text I/O

Additional text handling commands added by the standard textio package	
Apply_exponent	This procedure reads in numeric characters and the '_' character and uses them as an exponent for the rval parameter. It indicates the success of the operation through the ok parameter.
Apply_exponent	This procedure reads in numeric characters and the '_' character and uses them as an exponent for the rval parameter. It indicates the success of the operation through the ok parameter.
Apply_fraction	This procedure reads numeric characters and the '_' character from a line variable and converts them into a fractional number. It indicates the status of the conversion through the ok parameter.
Apply_mantissa	This procedure reads numeric characters and the '_' character until encountering a '.' character. It converts these characters into a real number and indicates any problems through the ok parameter.
Extract_integer	Once the optional leading sign is removed, an integer can contain only the digits '0' through '9' and the '_' (underscore) character. VHDL disallows two successive underscores, and leading or trailing underscores.
Extract_real	This procedure reads numeric characters and the '_' character until encountering a '.' character. It converts these characters into a real number and indicates any problems through the ok parameter.
Grow_line	This procedure increases the length of the specified line by the indicated increment
Int_to_string	This procedure convert a string to an integer

C-Language Interface

Drive a simulation from a C program:

Most VHDL simulation packages allow the user to interface actual executable programs to the simulation. These programs, usually written in the C or C++ language, allow very complex behaviors to be implemented without the speed or space overhead associated with using VHDL models. Alternately, the whole system could be modeled in C. This could be faster than VHDL for large simulations, but would require a mixed environment including both VHDL and C development tools. This could be a good approach for an engineer who was familiar with C, especially for one who works in a company where systems are simulated in C before they are partitioned and handed off to circuit designers for implementation.

System Modeling in C

Saeed Karamooz of Ramix, the customer who developed the SRAM controller used in this presentation, also developed a unique method for modeling. It involved modeling the system environment in the C programming language and writing stimulus vectors to a text file. The vectors were then applied to the device under test in a straightforward fashion, and the response vectors stored in another file. These were analyzed by similar program, also written in C. One nice thing about this technique is that it is applicable to virtually any simulation environment or tool. The biggest problem is that it is not interactive. Handshaking is not possible, so it does not have all the power and flexibility that would be possible from writing VHDL models. If this technique were used in conjunction with the C language interface of a VHDL simulator, no flexibility would be lost.

Other Useful Features

Component Instantiation

Models are simply instantiated in a design as components. Signals are used to connect the models to the DUT, just as if they were actual components on a board.

Generics

Generics allow the passing of a numeric parameter, or instance specific information, into an entity. This allows for parameterized models, which are more flexible in their application or reuse. For example, an SRAM model could be developed that would have a variable size and width. Even delays could be passed into each instance of a component's instantiation.

Arrays and User Defined Types

Arrays are useful for many reasons. A simple use can be to sequence through a list of vectors or patterns for application to a circuit. One technique that is used in our example is to define an array type that is used to specify what kind of delay is to be used in the simulation: None, Minimum, Typical or Maximum.

Packages

Packages allow the key parameters of a test program to be defined in a common file. This is useful for any type of complex design, but they are especially useful for defining

Designing a Test Bench using Models

IEEE Notes on Behavioral Modeling

Reference: <http://standards.ieee.org/catalog/press/VHDLMODS/TUTORIAL/MOD3/NOTES/HTML/SLIDE2N.HTM>

These notes are excellent guidelines for writing behavioral models. Rather than try to rewrite them, I have cited the source, which you can study further on your own. The key points made are:

- Model a system at multiple levels of abstraction.
- Hide the structure
- Focus on behavior and functionality
- At top level of abstraction, ignore timing.
- Follow standard practices of software engineering
 - ☞ Simplifies maintenance and reuse
- Structure the design
 - ☞ Define each component to have strong cohesion
 - ☞ Define set of components to be loosely coupled
- Use top-down iterative refinement
- Use abstract data typing to hide and encapsulate data

Using VHDL, a system designer can model a circuit (i.e., a component or system) at multiple levels of abstraction. In prior lessons, we have concentrated on the basic elements and the structural forms of describing models in VHDL. In this module we concentrate on the behavioral view, that is, describing how the circuit is to perform. We hide the structure of the design when modeling a circuit behaviorally. Instead, we are vitally interested in the functionality of the circuit. At the highest levels of abstraction, we even ignore timing.

When modeling in VHDL it is important to follow standard practices of software engineering. Otherwise, the model will be hard to maintain, even by the person who wrote it. In addition, to aid the reuse of models, even "throw-away" models should be created with care, and with the thought that others may use it. Typical model design and coding practices include structuring the design, iteratively refining a high-level view of the model down to its final form, employing abstract data typing to hide and encapsulate data, and organizing the individual model components so that they are loosely coupled (small number of interface signals) and have strong cohesion (keep strongly related functions in the same architectural body).

Top Level Schematic

(Schematic Diagram Included in Presentation)

In the schematic, a design for a CPLD-based SRAM controller is linked to models for all the other major components in the design. This CPLD design belongs to the Ramix Corporation and was targeted for a Mach M4-192/96-7 part. This part was chosen because the SpeedLocking of the internal delays allowed the part to work at an effective internal frequency of 122 MHz. State changes were required on both edges of a 66 MHz clock, and the M4 part was the only one with the desired density that would work. Vantis' .25 micron M4A family, which is sampling as this paper is written, will push the usable frequency to around 180 MHz. The models are listed below with a short description.

i960 CPU model

Two architectures

The CPU model has two architectures. A main process embedded in the VHDL code drives the first itself. The test language-parsing engine drives the second. The special instructions that are recognized by the parser are listed below. The text-parsing engine may seem like an extra level of complexity, and for the first implementation it is. But it offers one great advantage in flexibility and ease of use. The first architecture requires that the VHDL file be recompiled and loaded every time a change to the test sequence is made. This violates our desire to decouple high level tasks from low-level tasks. The instructions are

simple and are modeled after the bus operations that are required. Modeling all the assembly language op codes would have been too complex and would have been less useful. In this model, only ten instructions were needed, and these include setting the address and data for a given operation.

CPU Control Instructions

CPU Model Control Instructions	
Address	Specifies address to be used in subsequent read or write operations
Data	Specifies data to be used in subsequent read or write operations
Write single	Performs single write cycle
Enable data checking	Enables checking of data read back during read operations
Read single	Performs single read cycle
Number of burst cycles	Specifies the number of burst cycles to run during subsequent burst reads or writes
Wait cycle	Specifies which cycle to cause a CPU wait cycle during burst reads or writes. Specifying "0" disables CPU wait states.
Number of wait cycles	Specifies the number of wait cycles to insert during subsequent burst reads or writes (if wait cycles are enabled)
Write burst	Perform burst write cycles
Read burst	Perform burst read cycles

With these ten instructions, all the various modes of reading and writing the SRAM can be checked. Simple combinations of commands are used to implement all the required test scenarios. This model can be used in any system that requires this functionality on an i960 CPU.

SRAM model

For simplicity, the SRAM model is built from two simple 16 deep arrays of data, one at the lower address space of the model, the other is at the top end. The second array is implemented to allow RAM access to roll over from one page to another. The model responds to the standard SRAM control signals, write enable, output enable, chip select, address and data. Data is stored into the array when a legal write cycle is executed and is retrieved when a legal read cycle is executed. In addition to storing and retrieving data, the model also checks the SRAM timing requirements, if enabled. The timing parameters can be selected to be none, minimum, typical or maximum. The minimum access time doesn't really matter, so it was set to zero delay. When the timing checks are done in the model, where it makes the most sense, the model can be used in any system and will automatically function properly. The error checking code doesn't have to be re-implemented in each new application. Thus ease of use, and reuse, is enhanced.

Other Models Required

Chip Select Decode Model

This is a simple combinatorial decoder function with no parameter checking. The function and the propagation delay timing of the actual circuit are duplicated. The delay can be selected as none, minimum, typical or maximum.

Latched Transceiver Model

This is a simple sequential model with limited parameter checking for setup and hold. The function and the propagation delay timing of the actual circuit are duplicated. The delay can be selected as none, minimum, typical or maximum.

Test Language Parsing Engine

The test language parser is a powerful tool for extending the capability of a VHDL simulator which allows (with the use of functional models) to conceptualize the simulation at a high level of abstraction which increases productivity and efficiency. The bus cycles of the i960 CPU will form the basis for the custom test language. If useful, instructions will also be added to read the contents of the SRAM model directly, to allow the verification of the write functions independently of the read functions.

Test Scenarios

The simple function of this circuit requires very simple test cases. Each of the bus cycle types for the i960 model will be exercised by writing to the SRAM and reading it back.

- Single Write then Read
 - Bank 0 or 1 Start
- Burst Write then Read
 - Bank 0 or 1 Start
 - With/without CPU wait states
 - With/without SRAM page break
 - Overlap CPU wait states with page break

Third Party Model Sources

All standard bus models are available (PCI, SCSI, VME, ISA, etc.) as well as more specific models that are often supplied by the vendors of IP cores. The Logic Modeling division of Synopsys offers a wide variety of bus interface models that aid in verifying compliance and interoperability of standard bus designs. Many other third party vendors supply a wide variety of cores as well. An impressive list of non-commercial model sources are listed on the Web at: <http://www.vhdl.org/comp.lang.vhdl/FAQ1.html#3.2>

Evaluating the Benefits

Example Waveform Display

(Example Waveform Display from the Model Technology Simulator Included in Presentation)

High Level Simulation Run

The following is a sample listing from a simulation run from the modeling test bench. This can be examined in a few seconds to see if there were any errors. While it may take more work to get the first simulation running, you can see that each time a simulation is run, the time savings is huge.

```
Testing Single Write, Even Start
Testing Single Read, Even Start
  **Error in I960RP: Data Error,
  ** Address: 000000, Data was: uuuuuuuu sb: 55555555
Testing Single Write, Odd Start
Testing Single Read, Odd Start
Testing Burst Write, Even Start, No Page Break
Testing Burst Read, Even Start, No Page Break
Testing Burst Write, Even Start, With Page Break
Testing Burst Read, Even Start, With Page Break
Testing Burst Write, Even Start, With Page Break on last cycle
Testing Burst Read, Even Start, With Page Break on last cycle
Testing Burst Write, Odd Start, No Page Break
Testing Burst Read, Odd Start, No Page Break
Testing Burst Write, Odd Start, With Page Break
Testing Burst Read, Odd Start, With Page Break
Testing Burst Write, Even Start, No Page Break, single wait cycle
Testing Burst Read, Even Start, No Page Break, single wait cycle
Testing Burst Write, Odd Start, No Page Break, single wait cycle
```

```
Testing Burst Read, Odd Start, No Page Break, single wait cycle
Testing Burst Write, Odd Start, With Page Break, single wait cycle
**Error in SRAM512: Address Changed during Write,
** Address: 000004, Data: aaaaaaaa
Testing Burst Read, Odd Start, With Page Break, single wait cycle
Testing Burst Write, Odd Start, No Page Break, longer wait cycle
Testing Burst Read, Odd Start, No Page Break, longer wait cycle
Testing Burst Write, Odd Start, With Page Break, longer wait cycle
Testing Burst Read, Odd Start, With Page Break, longer wait cycle

Test Complete
There were 2 errors during this test
```

Comparison of Results

Checking Waveform vs. Checking Listing

Even a simple waveform display, like the one shown in the presentation, has many points that need to be checked. This becomes tedious page after page and is nearly impossible to catch all the errors. If this needs to be done after each of several changes, it is very unreliable. Many errors could go undetected. Checking the listing from the behavioral model test bench is easy. The last line tells if there were any errors. If there were not, the job is done. If there were, a quick inspection reveals the location, along with a clue as to the location and the cause. All checks are done automatically each time the simulation is run. Any errors that are missed can be easily corrected by updating the models, and the error should be caught in subsequent runs.

Completeness

Higher level of abstraction

Testing designs at such a high level of abstraction has many benefits. First, it frees the engineer of dealing with the minute details of signal timing and sequencing while designing the simulation flow and the necessary test cases. These details are handled at the level of the model building. This focuses attention on insuring that all pertinent test scenarios are covered

Components allow self-checking test benches

Using a self-checking test bench makes running simulations and validating results much easier because the models can do the checking automatically. This eliminates the need to scrutinize hundreds or thousands of lines of vectors or waveforms each time a simulation is run.

Modeling favors test benches which conform to actual usage

Since the structure and function of the models in the test bench parallels the actual hardware design, it is easier to insure that the tests cover all the operational modes of interest.

Ease of use

Models can be generated by a separate group, and can be developed without regard to the final circuit configuration. This is an advantage because jobs can be done in parallel. Even without division of labor, the automatic encapsulation, which results from a proper model design, limits the scope of the job to the specific parameters of that model. The design of the model never needs to be repeated but it can be used over and over. The overall design of the test bench is also fairly straightforward since it mirrors the actual system design. The control mechanism can be derived from the actual system, as in the case of the CPU model of our example. But the greatest benefit comes from being able to manipulate the system from a higher level of abstraction than would otherwise be possible. This makes it very easy to design the simulation and validate its function.

Flexibility

Flexibility, which is a big factor in ease of use, is an immediate benefit of using functional models for simulation. Models can be exchanged for other models if the design changes. Models are automatically decoupled, so changes to one model do not affect other models. And new test scenarios can be added or inserted easily because the tests were specified at a high level of abstraction.

Reusability

While reusability may not be as important a consideration as some of the others when initially choosing a methodology, it should not be overlooked. Design reuse has become one of the major methods proposed to close the design productivity gap. Any survey of current literature will show a plethora of articles dealing with design reuse, intellectual property cores and the virtual socket interface (VSI). A company can develop intellectual property as a pre-validated library of behavioral models for commonly used components. These components are general purpose and can be reused just as the same hardware components can be used in multiple designs. The ultimate in reuse flexibility could result from generating behavioral models in an object-oriented language like C++. Then, in addition to all the other benefits of behavioral models, these objects could inherit characteristics according to the rules of classes and inheritance in the chosen language.

Summary

Building functional models to create a "virtual system" for simulating a programmable logic design allows the engineer to be more efficient. Allowing focus on higher levels issue, this techniques also yields much more flexible designs that can be easily validated, modified and reused. The features of VHDL support these needs with a robust set of modeling and programming features.

References

Printed References

- 1076-1993 IEEE Standard VHDL Language Reference Manual, IEEE Press, 1993
- VHDL Made Easy!, David Pellerin and Douglas Taylor, Prentice Hall, 1997
 - ☞ Great overview with healthy treatment of VHDL test benches
 - ☞ Includes code for parsing engine on CDROM
- The VHDL Reference Guide, Eamonn Quigley, Esperan Ltd. 1995
 - ☞ Ultra handy pocket guide

Web References

- comp.lang.vhdl FAQ list
 - ☞ <http://www.vhdl.org/comp.lang.vhdl/FAQ1.html>
 - ☞ Includes all kinds of useful information
- IEEE VHDL Interactive Tutorial Demo
 - ☞ <http://standards.ieee.org/catalog/press/VHDLMODS/TUTORIAL/HTML/HOMEPEG.HTM>
- Example code for the models and test benches in this project
 - ☞ http://www.latticesemi.com/lit_dl/

Trademarks

Copyright © 1999 Lattice Semiconductor Corporation. All rights reserved.

Vantis, the Vantis logo and combinations thereof, are trademarks, and MACH is a registered trademark of Lattice Semiconductor Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.