

Modeling of a Futurebus+ System Architecture Using VHDL

Bob Schetlick
Protocol, a Division of Zycad Corporation
100 Enterprise Drive, Suite 500
Rockaway, New Jersey 07866

Abstract

This paper describes how VHDL has been used to create a behavioral-level model for the analysis of Futurebus+ based system architectures. The goals of the effort are presented as a backdrop for the information that follows. An overview of the model design is presented, which includes several modeling and analysis methodology highlights. This is followed by the discussion of an example architectural analysis highlighting the use of pseudo-random distributions and Rate Monotonic Scheduling theory to determine backplane traffic message mixes. The results of the analysis including a Futurebus+ specification issue will be presented.

Goals of the Effort

The goals of the effort were to capture the full Futurebus+ behavior in an executable model, as defined in the IEEE 896.1 and 896.2 Standards, to provide the capability to test various multiple node Futurebus+ backplane topologies via simulation, and to define and utilize an extensible modeling philosophy, resulting in a model that may be reused later during a hardware design for implementation verification.

The primary goals of the effort was to develop a VHDL behavioral model that captures the requirements contained in the IEEE 896.1 and 896.2 Futurebus+ specifications in an un-ambiguous manner. This "specification model" would provide an executable version of the Futurebus+ protocol. Such a model may be used to learn the intricacies of the bus protocol by executing scenarios to demonstrate the proper behavior. This type of model would be useful to system architects, interface and board designers to test their designs with, and Futurebus+ novices as a learning mechanism or sounding board.

In addition to implementing all of the specification requirements, the model also needed to support the execution of Futurebus+ protocol sequences when configured in various backplane topologies. This includes not only the instantiation of multiple copies of the specification model to emulate backplane bus loading scenarios but also the instantiation of the specification model with Futurebus+ implementation models to create a multi-vendor backplane scenario. This feature becomes significant when architecture analysis simulations are to be performed.

In capturing the Futurebus+ requirements into a VHDL model, another goal was to define and utilize an extendible modeling philosophy that would allow the model to be used during the development of a "real-life" Futurebus+ interface in addition to its functioning as a specification model or an architectural analysis tool.

The final goal was to use the developed model to represent, with reasonably accuracy, an implementation-independent view of a Futurebus+ backplane and to measure such metrics as throughput and latency under various loading conditions. The level of detail within the Futurebus+ model lends an adequate level of accuracy to the simulation analysis.

The creation of the basic Futurebus+ functional model and the architectural analysis did not actually take place during a single, contiguous effort. The Futurebus+ model was the result of a VHDL specification modeling effort¹ whose goal was to capture the requirements contained in IEEE 896.1 and 896.2 into an un-ambiguous and executable form. This model was later updated and used in a follow-on effort to analyze backplane bus architectures². For the purposes of this paper, however, the technical approach will be described as if it occurred as a single effort.

Model Design Overview

The model design uses modular behavioral VHDL to facilitate development, to increase understandability, and to allow the model or portions of its code to be reused. Given the complexity of the model used in the backplane bus analysis, it will be described along its highest level module boundaries. These three parts are the node-level model, the specification model, and the system-level model.

Node-Level Model — Figure 1 shows the architecture of the Futurebus+ node model. A node, in Futurebus+ terminology, is a complete entity (no relation to VHDL entity) that resides on a backplane (i.e., a node is equivalent to a Futurebus+ board). This node-level model consists of three parts: the backend, a logic system translator, and the specification model.

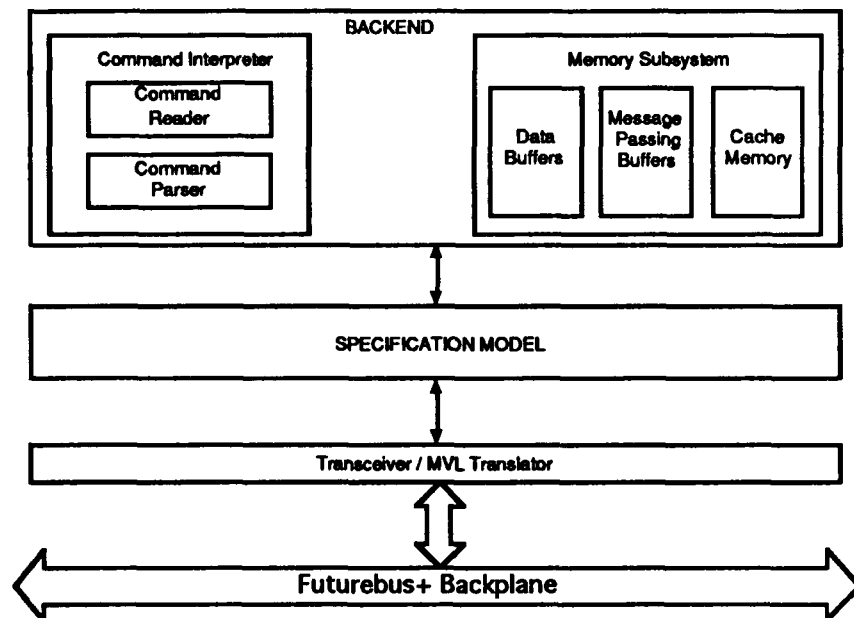


Figure 1. Node Model Architecture

The specification model is the part of the node model that behaviorally implements only the Futurebus+ requirements. Anything that is required for simulation control or integration that is not specified within the IEEE Futurebus+ standards is implemented outside of this specification model. The functionality implemented by the backend and the MVL translator fall into this category.

The backend provides the user interface mechanism and node-level functions that are implied or necessary for execution but are not actually specified by the Futurebus+ standards. Within this backend are two discrete functions: the command interpreter and the memory subsystem. The command interpreter is the user's interface to the specification model. This interface allows the user to specify, in Futurebus+ specific high-level commands, the behavior required of the model. The user may select any of the various optional behaviors defined in the standards to allow the creation of any scenario that can occur in a Futurebus+ system. Modifications were made to the command interpreter to specifically support the use of the model in a backplane bus analysis. These features are discussed in detail in the next section.

The MVL translator simply changes the boolean representations used within the specification model into the IEEE-1164 standard logic system (MVL9) to support the wired-OR nature of the multi-drop Futurebus+ backplane. This allows the model to be integrated in a simulated backplane with actual VHDL implementation models in any level of abstraction, as long as the IEEE-1164 standard is utilized.

Specification Model — As described above, the heart of the model used in the analysis effort was designed to be a specification model. This means that all of the requirements stated or implied by the associated specification are captured within the model. Specification models differ from design models in that they must remain as implementation independent as possible. This means that the kinds of design trade-offs made during a typical development that result in a specific hardware architecture or composition must be avoided whenever possible. In our Futurebus+ specification model, the only architecturally significant hierarchy included is defined or implied by the specification itself. For example, IEEE 896.1 divides the Central Arbitration, Distributed Arbitration, and Parallel Protocol descriptions into different chapters. Our model displays the identical functional decomposition.

The level of detail needed to implement the requirements in the specification tends to create a complex model that could be more than is necessary for architectural simulation and analysis. For our bus analysis effort, this model was used because it represented a complete implementation of the bus protocol and because it already existed.

As stated previously, the specification model hierarchy is based on the structure of the IEEE 896.1 specification. This allows for a relatively logical decomposition into manageable functional blocks. Figure 2 shows the specification model itself decomposed into four functional blocks: Arbitration Controller, Parallel Protocol Controller, System/Bus Management Controller, and the Control and Status Registers (CSRs).

The Central and Distributed Arbitration controllers implement the two different bus arbitration protocols defined by the Futurebus+ protocol. Each controller, when enabled, will accept a command to obtain mastership of the bus and perform the necessary protocol sequence to obtain mastership. The controllers, whether they are actively involved in the current arbitration or not, monitor their respective bus lines to follow protocol sequences. The two controllers are completely independent, since the Distributed Arbitration Controller may act as a secondary message bus even while central arbitration is being used in the system. The central arbiter is located in the specification model's testbench in order to keep the specification model itself philosophically pure. (The central arbiter functionality is not specified by the IEEE standards, but is only suggested.)

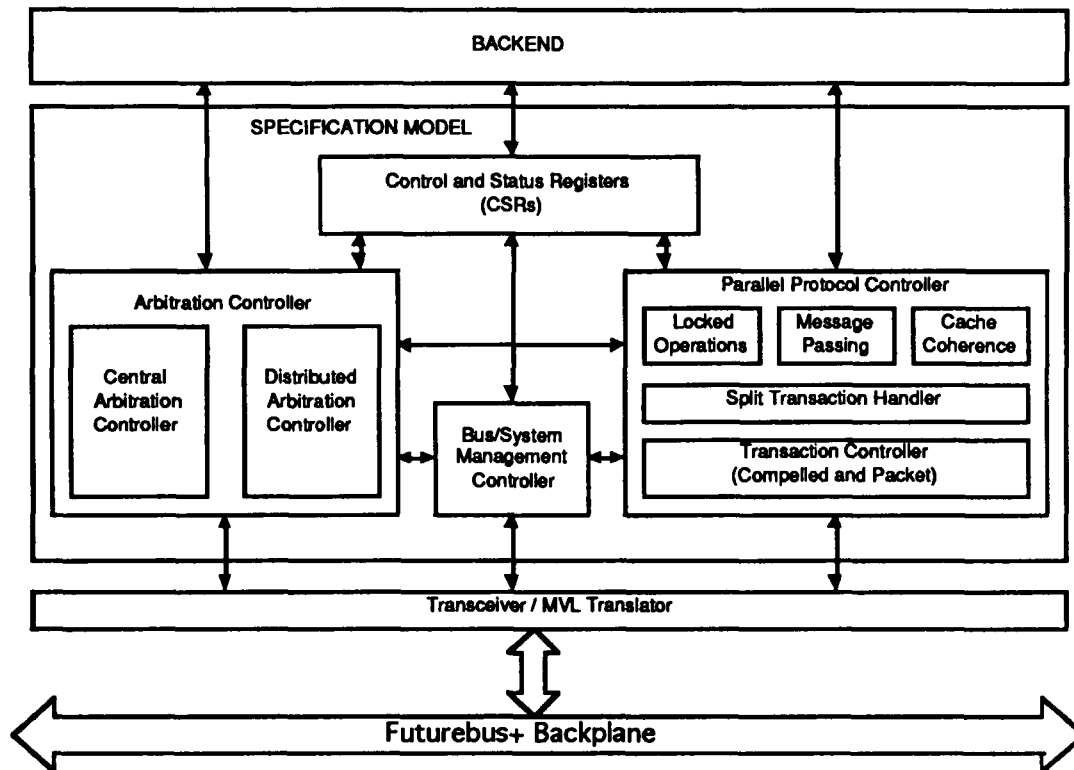


Figure 2. Specification Model Architecture

The Parallel Protocol Controller handles most of the other bus protocols, including basic transactions, split transactions, locked transactions/operations, message passing, cache coherency, and packet mode. It monitors the bus lines, makes transitions within its internal state machines, and when necessary, drives the information and synchronization lines. In addition to driving and following the Futurebus+ defined protocol, the controller's state machines also detect non-compliant behavior and signals this condition to the Backend so it can notify the user.

The Protocol Controller, in addition to driving and monitoring the Futurebus+ signal lines, also communicates with the other functional blocks to receive information such as its having been granted mastership (Central and Distributed Arbitration Controllers), the occurrence of a reset event (System Management) and a change in a programmable bus parameter (Control and Status Registers).

The System Management function controls such actions as power-on sequencing, system reset, bus initialization, and live insertion and withdrawal. The Control and Status Registers (CSRs) function implements a set of IEEE-standard registers used for communication between nodes within a multi-processor system.

The functionality of the Futurebus+ was implemented in a modular fashion for several reasons. It allowed the model to be incrementally developed and tested over time such that a model that implemented some subset of functionality would be available for other purposes. Second it allows interfaces of varying complexity to be represented through minor changes in this single model. This is particularly important for Futurebus+ where the standards define functional subsets called Profiles, which represent classes of interoperable interfaces targeted at specific applications (e.g. Profile B specifies a subset of functionality required for Futurebus+ I/O applications). The third reason for

modularity in the model design is to allow code reuse between the specification model itself and its associated VHDL testbench. The testbench was used during development to stimulate and record the specification model's behavior. As such, it needed to implement all of the specification model's functionality and incorporate the Central Arbiter function and the control of live insertion and withdrawal of modules (nodes).

System-Level Model — A system-level model was formed by instantiating multiple copies of the node-level model and interconnecting them by (simulated) Futurebus+ backplane signals. Figure 3 shows the architecture of the topology tested in the backplane bus analysis. This figure identifies the node in slot 0 to be a Futurebus+ testbench. This was the specification model's testbench, originally developed to validate the behavior of the specification model. The testbench was used as one of the nodes in the system model because several of the extra functions implemented in the testbench, such as the central arbiter and the power distribution (for live insertion and withdrawal), are required in the system-level model.

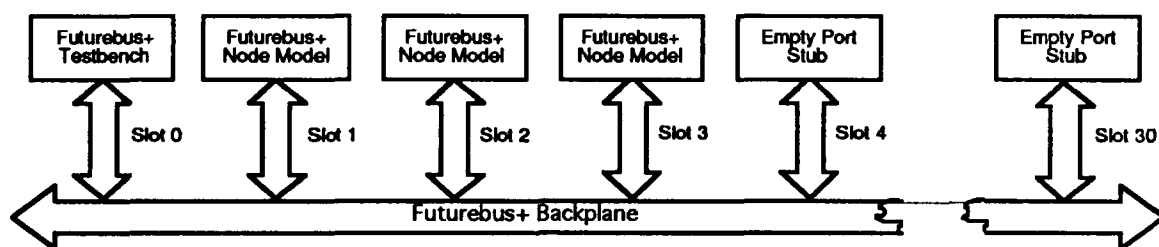


Figure 3. System-Level Model Architecture

Three copies of the Futurebus+ Node model were instantiated with the testbench model to form a 4-node Futurebus+ system model. The figure also shows empty port stubs located in each of the unused slots. This was done so that a full 30-slot Futurebus+ backplane could be developed once, with any number of nodes instantiated for a given scenario. To change the topology, the stubs are simply replaced with an actual node model.

Having discussed the basic design of the Futurebus+ model itself, we can now concentrate on the specific features added to the model for the purpose of using the model in a backplane bus analysis.

User Interface and Analysis Support Functions

As described in the previous section, the backend of the node model provides a high-level command user interface. These commands allow the user to define the behavior to be exhibited by the model for given situations. The formal language defined for the model is sufficiently powerful to allow any possible Futurebus+ protocol sequence to be executed in the simulation. An example of the commands accepted by the model is provided in Figure 4.

In the first part of this example, the model, acting as master, acquires the bus using the Central Arbitration protocol and requests that a Read-Unlocked transaction be sent to bus address 0000H. The read data is to be stored in buffer 01000H with the maximum length of the buffer being eight bytes. After the transaction completes, the model gives up bus mastership (ends tenure). The second part of the example commands the model to respond as busy to the next transaction in which it is addressed.

```

-- Master requests Read transaction

    REQ_CENT_ARB using RQ0

        REQ_TRANS READ_UNLOCKED to bus_addr 16#0000_0000#
        BUFF_PTR 16#0000_1000# BUFF_LEN 8 bytes

    END_TENURE

-- Slave Responds Busy

    TRANS_RESP BUSY

```

Figure 4. User Interface Command Example

In addition to these basic functions of the backend, the logic necessary to automatically generate message traffic was added. This logic consists of a set of pre-defined message types, each with specific characteristics and a mechanism to select messages to be sent based on a pseudo-random distribution function. This type of design allows the randomness associated with multi-processor synchronization and real-time environment changes to be represented in the simulation scenarios.

The message mix was implemented by a CASE statement in which each entry represented a different message type. The CASE entries defined the various parameters for that message, including word count, message period/deadline, and priority. The message selection mechanism used a function written in VHDL to pseudo-randomly generate an integer number. This integer was then used as an index into the CASE statement to select the particular message type to be sent. In this way, the overall message traffic generated by a Futurebus+ node will conform to the desired message mix.

Results analysis logic was also added to the Command Interpreter in each model that performs two functions. It automatically captures the data required to calculate throughput and latency and also performs those calculations. The latency and throughput calculations are on a node basis (i.e. the average latency for a given node to send its message). The bus utilization is calculated manually based on the results of all of the Futurebus+ node models in the system. All results are stored in a text file for analysis after the simulations are complete.

Example Architecture Analysis

The architecture analysis consisted of three parts: the definition of the system architecture configuration(s), the development of the message mix (or mixes) to be used in the simulation scenarios, and the analysis of the simulation results.

System Configuration —The original approach consisted of emulating two typical multi-processor architectures. The first type was to be a loosely coupled scheme utilizing a message passing mechanism for inter-processor communication. By definition in a loosely coupled system, activity on a single process resides within a single processor, resulting in minimal bus traffic. This implies that a relatively large number of processor boards may be located on a single backplane without being

restricted by bus bandwidth limitations. The second type was a tightly coupled architecture using a shared memory approach to inter-processor communication. Tightly coupled systems are more apt to distribute a single process across multiple processors, thereby creating high levels of bus traffic. For this reason, tightly coupled systems tend to have relatively small numbers of processors on a single backplane due to bus bandwidth limitations.

This approach would be implemented by using two different model configurations with two message mixes. The tightly coupled system would be represented by a 4 node system configuration with a message mix of many short messages. To emulate the efficient communication afforded by a shared memory architecture, a direct memory-mapped addressing scheme would be used to pass data between the bus and the model's back-end memory. The loosely coupled system would be represented by a 16-node system configuration with a message mix of fewer medium-to-long-length messages. The Futurebus+ Message Passing protocol would be used to emulate the high flexibility/high overhead typically associated with loosely coupled inter-processor communication.

After getting started, the decision was made to back off from our two-architecture/two-mix approach for performance reasons. Due to the complexity and size of the Futurebus+ specification model (40K+ lines of code each), the simulation run-time was unreasonably long (over 24 hours). Since the effort was predominantly intended to try out and prove a technology, the re-direction did not have severe impact.

The fall-back approach consisted of a single 4-node backplane configuration with two message mixes. The mixes were consistent with the tightly coupled (short messages, direct mapped) and the loosely coupled (medium/long messages, message passing protocol) described above. The architecture used is the one shown in Figure 3.

Message Mixes —As described earlier, the model is capable of automatically generating bus traffic based on pre-defined message mixes. The message mixes needed to be representative of an actual backplane bus application within a real-time system. To meet this objective, the mixes used in the analysis were derived from message characteristic data occurring on an actual parallel inter-module backplane bus within an existing avionics electronics rack. Surprisingly, this represented a bus loading of well below 10% of the available bus bandwidth. To form a scenario that applied a greater stress to the bus, a message mix was extrapolated, based on the message length and the relative message frequency obtained from the existing system.

The scheduling of messages on a bus usually happens in one of two ways. The first is a time-slice mechanism in which, at the time a system is designed, the designer decides how the available bus bandwidth will be allocated among competing processes. This method implies that the designer has a good understanding of the requirements placed on the backplane bus at the time it is designed. This method has the benefit of allowing the most efficient use of available bandwidth; however, the results are directly related to how well the designer anticipates the requirements on the bus.

A more flexible approach is to allow each process operating within the system to request use of the backplane bus asynchronously as required. The benefit of this approach is that the designer does not have to completely anticipate the requirements that will be placed on the bus as it is being designed, resulting in a more flexible system design. The drawback, however, is that a real-time system typically has message deadlines that must be met for the system to continue operating properly. Since this asynchronous mechanism, by definition, is not deterministic, it is difficult to guarantee message deadlines.

The Rate Monotonic Scheduling (RMS) algorithm addresses this issue for a subset of all types of message traffic. This algorithm relates a message's period or deadline to its priority. The theory is that short deadline messages have higher priority and are less

likely to be stalled waiting for other messages. RMS theory defines a criterion that, when met, guarantees messages will arrive on time. The subset of messages for which this algorithm works is periodic messages.

In our bus analysis, a variation on the Rate Monotonic Scheduling (RMS) algorithm was used to apply messages to the bus in a manner consistent with a real-time system application. The criterion defined by RMS theory was applied, but, due to the pseudo-random message generation mechanism in the model, messages were not periodic. Simulation results demonstrated that the deadlines were met for non-periodic messages using RMS theory as if the mix was generated using periodic messages.

The two message mixes used in our bus analysis are shown graphically in Figures 5 and 6. In each figure, the x-axis represents the message length (in 32-bit words) and the y-axis is the relative probability of that message type occurring. Figure 5 depicts the tightly-coupled mix as heavily loaded toward small messages, as would occur in a tightly coupled multi-processor system. Figure 6 depicts the loosely-coupled mix as loaded more with medium to long messages, as would occur in a loosely coupled multi-processor system.

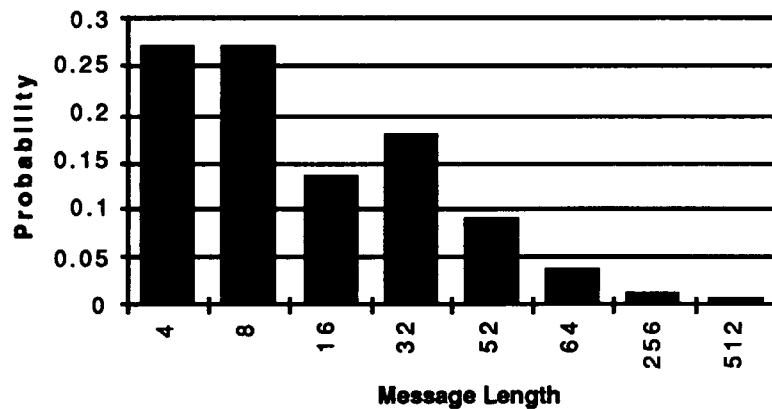


Figure 5. Tightly-Coupled Message Mix Distribution

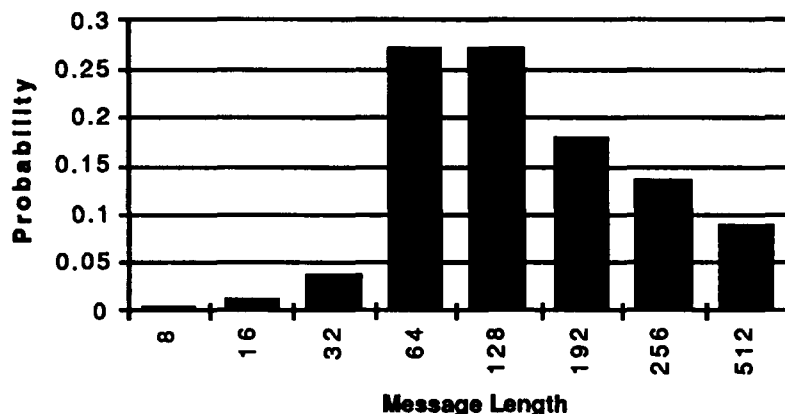


Figure 6. Loosely Coupled Message Mix Distribution

Having defined the two message mixes, the simulation testing and analysis could begin. The testing consisted of executing each mix on the developed Futurebus+ system model. The messages in the loosely coupled mix were sent using the Futurebus+ message passing mechanism for inter-processor communication. The messages in the tightly coupled mix were sent using the basic Futurebus+ transaction protocol to emulate a shared memory approach to inter-processor communication.

Results Analysis — The results that were obtained from the bus analysis effort fell into two categories. The first is the raw data for bus utilization—throughput and latency. The results here not only confirmed our presumption that message passing added considerable overhead to communication but also provided hard data on the magnitude of the overhead. This data could be used in deciding a direction for a proposed system architecture. The primary result was that VHDL was successfully used in performing the architectural level simulation analysis.

The second result that came out of the effort was rather unexpected. We found that in order to implement a real-time scheduling algorithm, the underlying backplane bus needed to have certain capabilities. Two of the most important of these is that the protocol must support a fair mechanism of bus mastership arbitration based on variable priorities and the support of message/transaction pre-emption. Pre-emption is important since a priority-based scheme hinges on the assumption that the pending message with the highest priority will be sent immediately. During any period that a high priority message has to wait for a lower priority message, a condition called priority inversion exists. The most effective way to reduce priority inversion is to suspend the lower priority message to let the higher priority messages get through.

The problem that we encountered is that the model used in the analysis was based on a Futurebus+ specification model that only implemented the requirements extracted from the specification. Since the Futurebus+ specification included a descriptive reference to pre-emption but did not include associated requirements, the capabilities were not present in the model. Having discovered the problem, the issue was brought to the attention of the IEEE group responsible for the development and maintenance of the Futurebus+ standards, which promptly took action to rectify the problem. This highlighted a very important deficiency in the Futurebus+ protocol that would impact real-time system developers, making the bus analysis effort all the more valuable an undertaking.

Conclusions

The importance of this effort is twofold. First, the development of detailed models of bus and communications protocols is not only possible but highly valuable. This can be seen from the fact that the developed specification model has been used in several capacities and is still proving its worth by identifying issues with the specifications to which they were designed.

Second, we have used VHDL as a means of performing architectural modeling on a complex backplane-bus-based system. This model provided a useful mechanism for trying anticipated message mix scenarios on various system configurations. The lessons learned included a clearer understanding of the tradeoff between the level of modeling detail and the time required to meet the intended goals. Even though our model may have been more detailed than was required, the results obtained were beneficial.

All in all, we believe that architectural modeling is a welcome addition to a VHDL-based top-down design process. By extending the top-down paradigm a level higher, the benefits of trying out an architecture before a design is in concrete is worth the extra effort required early in the process.

-
- 1 **Futurebus+ VHDL Simulation Development Program, NWC AR/DRRB,
Contract No. N60530-91-C-0345.**
 - 2 **JIAD Avionics Backplane Bus Assessment Program, Wright Labs,
Contract No. F33657-91-C-0058**