The Bidirectional Wire Model with Delay

Michael D. McKinney
VHDL Design Services
2113 Normandy Drive
Irving, TX 75060
E-mail : 74037.3515@compuserve.com

Abstract

Building system simulations in VHDL can sometimes be an intensive and complicated task. Usually a major part of that task involves modeling bidirectional networks such as backplanes, motherboards and even connections between ASIC's on a printed circuit card. Many of these networks involve a delay of some kind. This paper presents a concept for a bidirectional wire model which can contain a delay specification.

Introduction

The need for a bidirectional model in VHDL first became apparent while working on the implementation of the EIA-567 standard early in 1992. The idea was to create a shell around a particular core element of a design which would contain all constraint testing required for the design as well as insertion delay and options for several propagation delays per pin. For purely input and output pins, the implementation was trivial : simply assignment statements with delay. But for bidirectional pins the implementation became quite tricky.

It became clear that a model of a bidirectional network or wire which included delay would have several applications outside of the EIA-567 system. For example, modeling of backplane or motherboard wires which typically have high loading and delay parameters; actual modeling of capacitive delay loading on circuit board traces connecting ASIC's or other IC's; a method for providing representative but customizable pin-to-pin delays for components or ASIC's during system verification, even though the environment may still be using "pure" (zero delay) VHDL modules; and a way of keeping track of which drivers on a particular bidirectional network are currently driving without having to delve deeper into the ASIC designs themselves.

One article was found in current literature which addressed this problem, and its focus was on modeling transmission line parameters such as voltage, current and capacitance, and had logical transmission and delay as secondary concerns. The article served to clarify and bound the problem area.

Development

What was needed was a model of a wire which would conform to these three basic specifications as well as possible:

I. The model would behave logically exactly like a wire.

II. Delay in both directions could be specified, not necessarily the same value. This make the model easier to use.

III. Because of its possible instantiation many times in a single design, the model should place as little burden as possible on the simulator environment.

Item I above was further subdivided into the following set of rules:

Under Asynchronous Conditions:

1. **Both ports should resolve to the same value;**
2. **Both ports should resolve to the correct value.**

These rules deal with how the outputs of the wire should behave in a simulation, and are not rules about how the internal algorithm should perform. Other rules will be stated, but any developed model must perform in a simulation with these two basic ones.

Terms

There are two terms which must be fully appreciated in order to understand the discussion which follows: EVENT and TRANSACTION.

An EVENT is an activity on a port which changes the logical value of that port. A TRANSACTION is an activity on a port which does not change the logical value of that port.

A basic rule is that EVENTs are also TRANSACTIONs, but TRANSACTIONs are not not always EVENTs.

First Cut

An early attempt at creating a wire model was as simplistic one. The signal flow was simple also:

When an EVENT occurs on one port, drive its value to the other port (Figure 1).

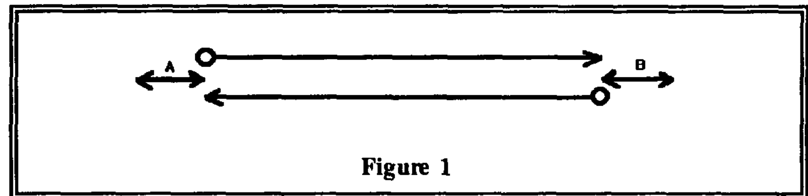

Figure 1

While at first this seems adequate (two straight-forward concurrent signal assignment statements), this structure has a fatal flaw. A component or process (in this case an implied process) introduces a new driver onto the networks of port A and port B, where a simple signal connection does not. If at any time both of the ports drive active (strong) values, the wire component will lock up into that state. Since this activity occurs very often in a typical simulation, results are very likely fatal. The basic reason behind this effect is that the model can only sense the resolved values on its ports, and the model itself is providing one of those drivers. No other network driver can overdrive the strong wire model values except to force an 'X' or 'U'. Therefore, while the model technically passes rule 1, it fails for rule 2.

Next Steps

Following on the heels of the first trial came an attempt at breaking the looping action of the previous structure. The signal flow for the model is now:

When an EVENT occurs on one port, drive its value to the other port, unless it is the result of a previous EVENT on the other port (Figure 2).

The model is now structured into a VHDL process because of the required evaluations, and introduces the concept of recognizing whether an EVENT on a port was caused by internal or external activity. At the time of an EVENT on a port, the algorithm signals that the other side should ignore its next EVENT. While this structure is more precise, it still contains the error shown in the previous example. However, the error now shows up as a function of the depth of the networks connected to each port. In addition, a more subtle error was introduced in that the sequential process statements produced a priority for port A. This ultimately produced unacceptable behavior.

The purpose of this excersize is, as exactly as possible, to model the logical and delay characteristics of a wire. The wire model therefore may be placed on any node of an arbitrarily deep heirarchical design. The depth of the network on either side of the wire cannot be known by the model itself.

With the current structure, when an external change occurs on a port during the delay time from the opposite port, OR when a change occurs externally exactly when the internal change does, the logic of the structure breaks down and the component will again tend to latch into an active or strong state. Therefore, this structure will also fail for rule 2 and may occasionally fail for rule 1 as well.

A New Rule

It began to be clear now that neither port could arbitrarily transport its value to the opposite side without some further code evaluations, and an in-depth understanding of the behavior of a wire produced this third basic rule:

3. The port of the wire must pass an "unresolved" network value rather than the "resolved" value, to the opposite port.

"Unresolved" for this rule means the value of the external network connected to the port without that port's contribution to the network. Therefore a method had to be found to "unresolve" these networks for use inside the model.

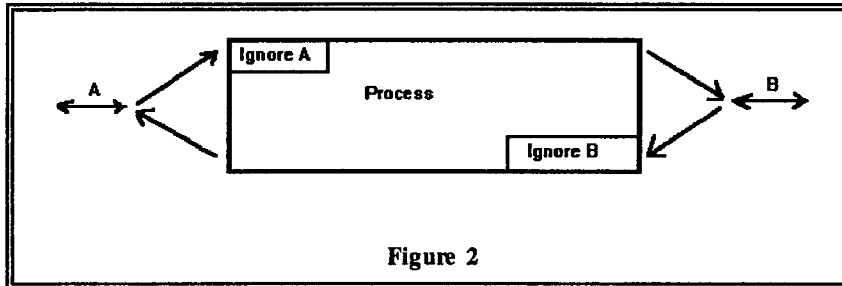

Figure 2

If a set of logic states is being used which includes the 'Z' state, and assuming the presence of a resolution function, there are two methods for disconnecting signals from networks. First, by the use of signals of kind register, a part of the VHDL language itself, and second, by using the 'Z' state, which represents an off, high impedance, or no-current state. Signals of kind register can be set to "null", removing them from consideration from the resolution function, but only from within VHDL blocks, or with disconnect statements. This method was rejected as too complex, and held out the possibility of a greater load on the simulator environment. The 'Z' state method was therefore chosen.

Try Again

The next model had this structure:

When an event occurs on either port, 1) turn off both ports, evaluate both "unresolved" values, 2) re-drive ports with new values as necessary (Figure 3).

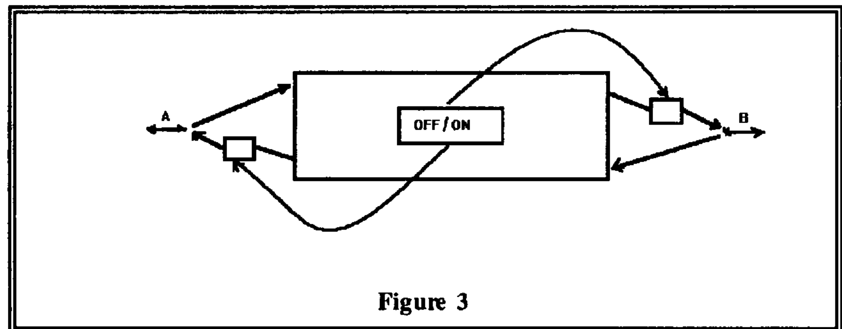

Figure 3

This time a single process was used and used WAIT statements. The structure appeared to solve the major problems up to this point: first, the priority problem was eliminated because the single process evaluated both ports at the same time, and second, the model was shown to produce both the same and correct values on its ports during basic testing.

However, there were now three new issues: first, the structure has to evaluate a lot of code, since all the possible pairs of values must be looked at individually; second, the process was triggered by EVENTs and took a minimum of 6 delta-times to complete; and last, the EVENT produced by the re-drive on the ports was implicitly ignored.

Deeper testing of the model, however, caused it to fail. It was found that triggering on EVENTs was not sufficient, because when a network changed it value from a stronger to a weaker value only a TRANSACTION was produced. It was also found that a "one process" structure did not lend itself to inclusion in the EIA-567 method, and a switch to a "two concurrent process" structure was necessary. Lastly, it was found that, although the model was getting closer to actual wire behavior, under special conditions the model would now fail rule 1. It took some time to nail down the cause of the problem: external EVENTs being missed during the last scheduled delta-time.

Final Model

Finally, it was decided that the model must not arbitrarily skip or ignore EVENTs during the final delta-time, but had to try to distinquish between internal and external activity, both of which will cause an EVENT or TRANSACTION on the port. Since EVENTs and TRANSACTIONs were being sensed before the model's contribution to the network was removed, a simple evaluation of the network state was found to be not sufficient.

The final structure was developed such that the code segment was smaller, and the model could always reproduce the correctly resolved values in a timely manner. The time to completion was reduced to 3 delta-times. This is the final structure:

In identical concurrent structures, 1) turn off the port, evaluate result and drive opposite port as required, re-drive the local port, and 2) start over immediately for external EVENTs, but start over in the next delta for internal EVENTs (Figure 4).

Thorough individual testing of the model was not able to "break" it, and including the model into real designs and testbenches proved it to be entirely reliable.

References

IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual, (ANSI).

IEEE Std 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability, (Std_Logic_1164), (ANSI).

Harr, Stanculescu, Eds, Applications of VHDL to Circuit Design, Kluwer Academic Publishers, Boston, 1991.

Berge, et. al., VHDL Designers Reference, Kluwer Academic Publishers, Boston, 1992.

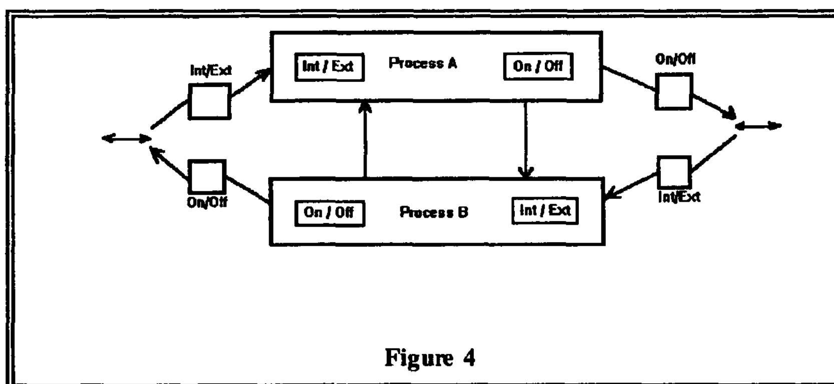

Figure 4

Conclusions

The finished model has been rigorously tested and has been shown to pass all rules while having minimum impact on the simulator environment where it is used. Both of the delay times across the wire can be controlled easily and need not be the same value. The model takes a minimum of 3 delta-times to complete its activity, but under special circumstances will wait in order to complete correctly after all other network resolutions are finished. Its activity is entirely synchronous with respect to the design or testbench, and therefore does not have processes continually active, which may impact the simulator environment.

Applications may include modeling backplane systems, modeling wire loading delays between devices on circuit boards and monitoring which IC device drivers are active at certain points in the simulation.