Interactive Models and Testbenches in VHDL

Himanshu Thaker and Janick Bergeron
hemi@bnr.ca janick@bnr.ca
Bell-Northern Research Ltd., Ottawa, Ontario, Canada

Abstract

A popular software property, that of interactiveness, is brought to the realm of hardware models. Through interactive testbenches a model developer can provide a more flexible environment in which to exercise his model and gain a quicker and better confidence in its correctness. Through interactive interfaces to the models themselves, users unfamiliar with the model or the modeling language used can run and control a simulation, monitoring the information that is of interest.

This approach to modelling and testing is illustrated by several actual examples as well as the tools facilitating the development of interactive models and testbenches are presented.

Section 1. Introduction

Historically, models have been considered as disposable implementation verification commodities with the sole purpose of verifying that a given implementation met a set of written specifications. As the complexity of those specifications has grown, it has become more and more difficult to completely and unambiguously express them in a written natural-language form. Models are now starting to appear as an integral part of the specification documents with natural-languages used as an extension of the model to express information which cannot be easily conveyed in the modelling language.

Writing *accurate* models of complex system components is not an easy task. Providing an infrastructure and a strategy for testing a model (i.e. a testbench) has been found to occupy 40% or more of the modeler's time and must not be neglected; after all, one can only be as confident in one's model as in one's testbench. Many tools and methodology have been designed to speed-up and facilitate the writing of the model itself[1][2]. Just recently have tools started to appear to help the modeler in what is becoming an ever increasing portion of his work[3]. Interactive testbenches and associated support tools are presented as a mechanism for providing versatile and easy-to-modify testing to models.

The effort required to write and test accurate specification models makes them very expensive and no longer easily disposable. Finding other applications for these models other than mere executable specification would be an inexpensive way of leveraging the investment in them. Unfortunately, models tend to be hard to transfer to other groups of users as they require in-depth knowledge in order to modify or adapt them to their new application. It becomes necessary not only to provide accurate model, but also to provide *usable* models. Interactive interfaces are presented as a powerful way of making models more usable.

Section 2. Interactive Testbenches

Many models written in VHDL are simulated in batch mode. That is, a model is written and compiled, and then test vectors are created and applied all at once to the model. The user examines the output, possibly modifies the vectors to help isolate any problems, and reapplies the vectors, until a bug is found and fixed. However, in models where errors can be quickly and early identified, this method of simulation is very inefficient. In these cases, it would be preferable to interact with the model, dynamically changing the "vectors" as the simulation proceeds.

An example of one such case is shown in Figure 1. This system employs a CSMA-CD (Carrier Sense, Multiple Access - Collision Detect) protocol to communicate among the components. One of the goals for the modelling exercise for this system is to observe the behavior of the *entire* system to various collision scenarios. This requires the ability to interactively inject faults into the transmission medium, set the hardware into known states, and observe the status of various signals and registers as the simulation proceeds.

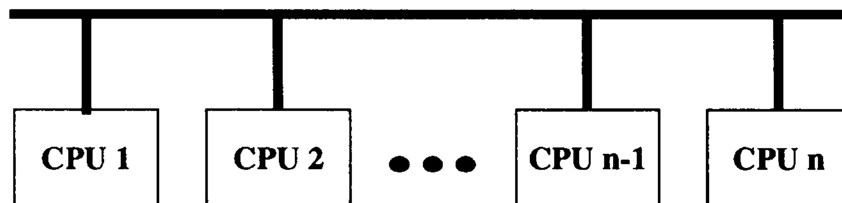

Figure 1 : System Overview

The model for the system is depicted in Figure 2. The system includes multiple instantiations of a CPU (except for their identification, all CPUs are identical), a model of the transmission medium, and a controlling process that also interfaces to the user. It is through the interface that the user can interactively control the simulation. For example, if a user wants to observe the method in which a CPU handles a message when its input buffer is full, he could converse with the model as shown in Figure 3.

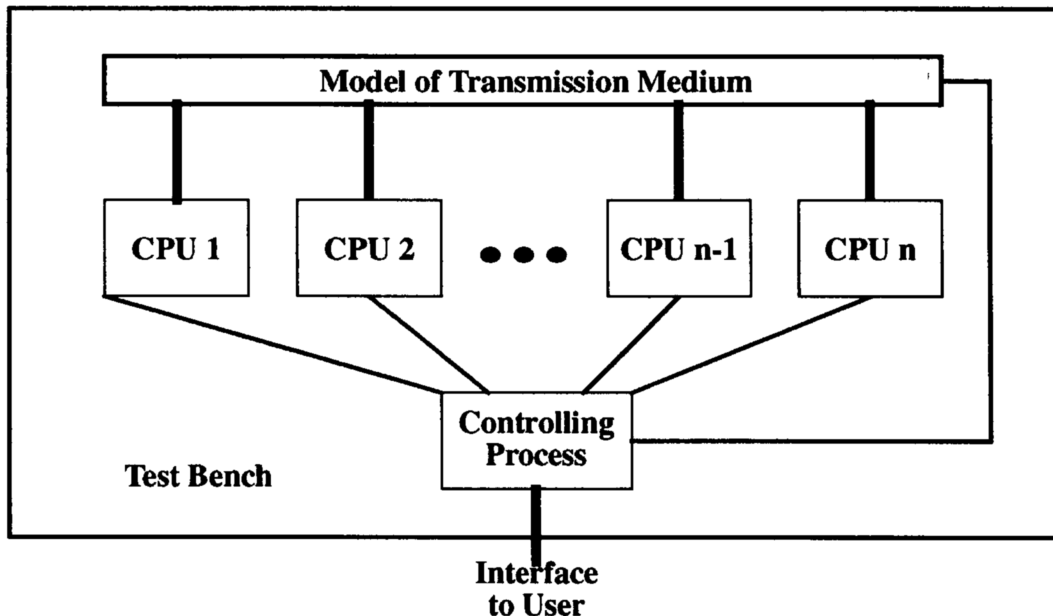

Figure 2 : Model of System

```
CMD> set bfull on cpu 3
CMD> send from 8 to 3 after 10 ms mesg 198
CMD> run 100 ms
...
CMD> status cpu 3
...
CMD> read cpu 3 buffer
```

Figure 3: Sample Dialog

In this dialog, the *buffer_full* flag is set on the third CPU, and then CPU #8 is set up to transmit to CPU #3 after a certain amount of time. The simulation is allowed to run for 100 milliseconds, and then the status of CPU #3 is observed. Finally, that CPU's buffer is read.

To increase controllability, the interface to the CPUs consists, in addition to the physical signals, of control signals that allow the user to read and write to registers and flags, to set up fault conditions and modify the level of debug information produced.

Integrated into the code of the CPU is the usage of a DEBUG package. This package allows the user to selectively display the level of detail that is printed to the output. This gives the user the ability to have verbose detail printed when testing crucial sections and have terse messages (or no messages) printed when simulating to observe the system over a long period of time.

With this model, it is possible to observe the behavior of the CPUs in a system environment that would be difficult, if not impossible to physically set up (e.g. collision scenarios where multiple CPUs start transmitting at the same time) and examine registers as the simulation proceeds to help isolate the problems. Because the model is interactive, the user can modify the state of any CPU and observe different parts of the system under varying degrees of details, *while* the model is running. This allows for rapid verification of the protocol.

This particular system has also been used as a learning tool, to enable a new member of the project to quickly familiarize himself with the protocol. Being able to simulate basic scenarios, and observe the effect at various places was a valuable exercise, as it allowed the individual to ask "what-if" type questions that could not have otherwise been answered with simple vector testing or a paper document.

Section 3. Interactive Models

In the same fashion, interactive testbenches are used to provide a flexible testing strategy, an interactive interface to the model itself can provide better controllability and observability than otherwise possible, while removing the user from the harsh details of the model itself.

Models of programmable devices are prime candidates for interactive capabilities. With the ever-increasing popularity of hardware/software co-simulation as a system-level verification strategy[5], models of the hardware which provide a programmer's view of the device, similar to logic analyzers or in-circuit emulators, are of great help in monitoring and debugging software running on a model of the hardware.

Figure 5 outlines how and where an interactive monitor can be included in a model of an instruction-set processor. Initial conditions are set to guarantee that the monitor will be invoked as soon as possible, thus giving immediate control of the model to the user. It is very easy to control, from within the *MONITOR* procedure, all variables and all signals with drivers in the process without changes to the original model by using side-effects. Purists may want to pass all controllable signals and variables as arguments to the procedure but at the expense of reduced readability and run-time efficiency. Typical com-

```
process
    ...      --  Declarations visible in the monitor
    variable StepLeft: INTEGER = 0;              -- Initial break
    variable BreakPoint: BooleanArray(AddressRange) =
        (others => False);
    procedure Monitor is
    begin
        ...      -- Interactive Interpreter
    end;
begin
    wait for 0 ns;         -- All processes must initialize
    loop
        ...      -- Fetch and decode next instruction
        if StepLeft = 0 or BreakPoint(PC) then Monitor; end if;
        ...      -- Execute the instruction
        PC := PC + 1;
        StepLeft := StepLeft - 1;
    end loop;
end;
```

Figure 4: Interative Monitor in Model of Instruction-Set Processor

mands offered in such a monitor enable the user to set breakpoints, step through the code, display the content of the registers, force values into registers, run to next breakpoint or quit altogether. Figure 6 shows a sample from a possible interaction with a monitor imbedded in a model of the 56156[5].

```
Stopped at 0x0F4E: CLR A; X0->B
CMD> step 1
Stopped at 0x0F4F: ADD A,X0; Y0->X0
CMD> registers
A = 0000000000; B= 0000007EED; X = F54A7EED; Y= 0000FF00;
M 0:0000 1:0000 2:0000 3:0000
N 0:0000 1:0000 2:0000 3:AAAA
R 0:0000 1:FFFF 2:0000 3:AAAA
CMD> exit
```

Figure 5: Interacting with a monitor in a model of the 56156

Early uses of interactive monitors in models of instruction-set processors have identified the need for an instruction disassembler. Rather than provide a separate disassembling function which would duplicate most of the instruction decoder, the image of the instruction to be executed is built in parallel with the translation of the instruction from bit patterns to symbolic opcodes. Figure 7 shows a portion of the instruction decoder/disassembler for a model of the 56156.

If a simple command-line interface is not sufficient, the interactive interpreter may be used to communicate with a graphical front-end. Instead of reading from and writing to the terminal, I/O goes through the host system's interprocess communication system to a more elaborate user front-end. Figure 8 shows how a monitor for an instruction-set processor has been interfaced with Bell-Northern Research's in-house framework providing a multiple-window, mouse-driven model monitor[4].

Section 4. Command Parser Generator

Creating the interactive testbench required writing a parser to interpret user commands. To assist us in this task, an automatic parser generator was written. This tool (called

```
if Match(Instruction, "0100|||DATA ALU") then
     -- Decode ALU Operation
     if Match(Instruction(7 downto 0), "0000F001") then
          INSTR.OP := CLR; Write(INSTR.IMAGE, "CLR ");
          if Instruction(3) = '1' then
               INSTR.D.MODE := BB; Write(INSTR.IMAGE, "B");
          else
               INSTR.D.MODE := AA; Write(INSTR.IMAGE, "A");
          end if;
     else ...
     end if;
     -- Decode the parallel move
     case Instruction(11 downto 8) is
     when "0000" =>     INSTR.MV1.S.MODE := X0;
                        Write(INSTR.IMAGE, "; X0->");
                        if INSTR.D.MODE = AA then
                             INSTR.MV1.D.MODE := BB;
                             Write(INSTR.IMAGE, "B");
                        else
                             INSTR.MV1.D.MODE := AA;
                             Write(INSTR.IMAGE, "A");
                        end if;
     when ...
     end case;
else ...
end if;
```

Figure 6: Decoding and Dissassembling 56156 Instructions

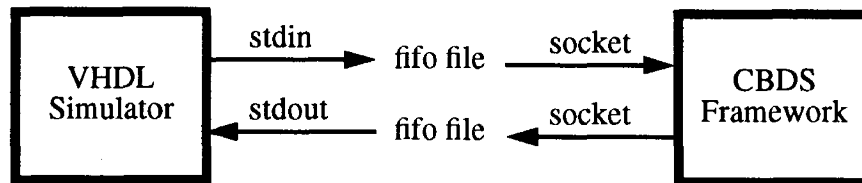

Figure 7: Graphical Interface to Interactive Model

VCMG for VHDL Command Module Generator) is similar in functionality to the UNIX *lex* and *yacc* tools in that it takes input describing the formal syntax and generates a parser from it. Like *lex* and *yacc*, VCMG allows you to quickly specify the syntax; however, it is up to the developer to create the semantics. Input is parsed and separated into tokens, and inserted into a record. Unlike *yacc*, where actions are interspersed in the parsing code, the developer must create the code to examine this record and take appropriate action outside of the generated parser.

The input for VCMG is a grammar, in formal BNF-like constructs, of the language the generated VHDL must parse. Vcmg has two distinct sections, separated by '%%'. The first section is used to specify the grammar itself, and the second section is used to specify bounds checking for the integer variables.

The grammar section is where the designer specifies the actual parsing sequence. The syntax is:

command param1 param2... paramN;

where param is defined to be

token or [param] or {param} or param | param

A token (at the leaf level of the recursive decomposition) can be a string literal, an integer token or a string token. An integer token is identified by the prefix "I." and a string token by "A.". Square braces indicate an optional parameter, curly braces indicate a group of mandatory parameter and the pipe symbol indicates a choice of parameters. A parameter without braces is also deemed mandatory.

The last section is where integer variables have their constraints identified. The syntax is:

I.variable range MIN to MAX;

where MIN and MAX can be integer values, or the key words "high" or "low".

Shown in Figure 9 are examples of a VCMG grammar specification. In the first example, VHDL code will be generated to parse for the keyword "run", followed by an integer, followed by an optional modifying token indicating the units of time that the integer represents. The second example shows a more complex grammar specification.

```
run {I.time} [ns | us | ms];modem
%%
I.time range 0 to high;
```

(a)

```
send [from] I.cpu [to I.cpu2] [[after] I.time] [ns | us | ms];
generr I.cpu {off | {on {single [after] I.time [ns | us | ms]}
                    | {random [one_in] I.prob} } };
run {I.time} [ns | us | ms];
%%
I.time range 0 to high;
I.prob range 1 to high;
I.cpu range 0 to 8;
I.cpu2 range 0 to 8;
```

(b)

Figure 8 : Example vcmg specification.

VCMG generates a package called "commands". The command package prompts the user to enter text until a valid (according to the syntax) sequence is entered. Figure 10 shows an example of a dialog with a command parser generated from the first example of Figure 9 .

```
CMD>run 100 ns          -- valid command
CMD>run 200             -- valid command
CMD>run -3 ms           -- invalid command
Token '-3' is not in range.
  Valid range is 0 to 2147483647
Invalid Command!
```

Figure 9 : Sample Dialog using code from Figure 9(a).

This package can then be interfaced to the testbench, as depicted in Figure 11 . The command package contains an enumerated type called OPCODES that enumerates all the given commands. It also defines a record type that forms the basis of communication between the calling routine and the *Get_Command* procedure. Finally, the prototype for this procedure is defined.

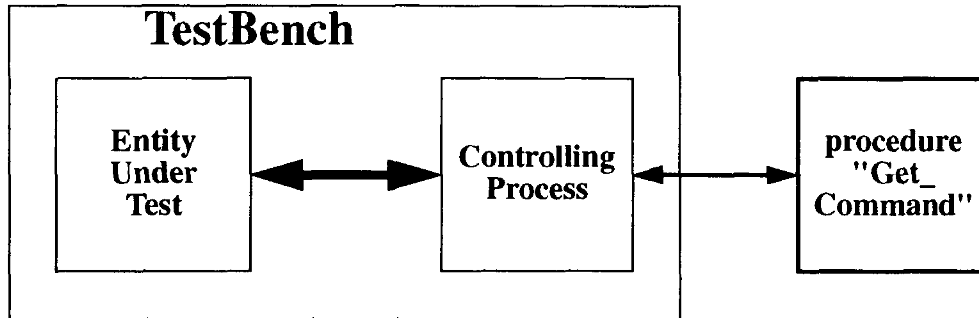

Figure 10 : Interfacing the TestBench

Every field in the record type has a default value that is assigned to it when the procedure GET_COMMAND is called. When this procedure parses the run time input, it fills in the appropriate fields as required. Thus, if a field is not filled in, it will remain at its default value, and then can be tested for by the controlling process that called the procedure. The body of the generated package consists of the code for the procedure *Get_Command.* It is in this procedure that parsing is accomplished.

After the command module has been generated, it is interfaced to the testbench by creating a process in the top level architecture that contains a looping structure, with the call to *Get_Command* as the first statement. A prototype of the controlling process is shown in Figure 12. For every command, one can readily determine the parameters that were entered by examining the various fields of the record returned by *Get_Command.*

```
process
    variable CMD : COMMAND_DESC;
begin
    CMD_LOOP : loop
        GET_COMMAND(CMD);
        if (CMD.OPCODE = O_command1) then
            -- code for what to do if we get command1
        elsif (CMD.OPCODE = O_command2) then
            -- code for what to do if we get command2
        elsif ...
        end if;
        wait for 0 ns;  -- to allow for possible delta cycles
    end loop;
end process;
```

Figure 11 : Controlling Process Prototype

The controlling process is responsible for driving the necessary signals that interface to the entity under test. For example, a command could forecast an event to occur on a signal after a specified amount of time by a simple signal assignment. Additionally, this process is responsible for advancing time, as directed by the user, which can be simply accomplished by having it wait for the specified amount of time.

VCMG has been successfully used on a number of models, and has reduced the time required to create the parser from over a week to a few minutes. The parser for the system in Figure 2 required 18 lines of formal syntax to produce over 920 lines of VHDL code.

Section 5. Conclusions

Several interactive testbenches and models have been written to help test models of actual chips and facilitate the transfer of the models to the design groups. They have

proven to be a powerful approach to a flexible testing methodology and easy-to-use models.

Interactive VHDL models can be thought of as interpretive environments, bringing with it all the well known advantages that interpretive systems have over compiled systems, such as allowing for quick verification and enabling the user to pose "what-if" scenarios. This, coupled with the DEBUG package, allows one to test their model with different levels of detail as required.

The automatic parser generator has been a great benefit to the interactive environments that the authors have written, reducing the time required to generate the parser from over a week to minutes. This parser essentially gives the user advanced text I/O capabilities in VHDL that VHDL lacks, in comparison to standard programming languages.

It is unfortunate that the VHDL environment offered by some vendors map the predefined file object *TextIO.Input* and *TextIO.Output* to pre-defined files rather than the process' standard input and output streams. Although it makes for a more consistent implementation of the tool itself, the richness and flexibility of interactive models can only be accessed through acrobatics of the very forgiving underlying operating system.

Section 6. References

[1] T. Naegele, "*No Need to Write: i-Logix Lets Designers 'Draw' VHDL*", Editors Choice, Military and Aerospace Electronics, March 1991.

[2] L. Lundberg, "*Generating VHDL for Simulation and Synthesis from a High-Level DSP Design Tool*", VHDL for Simulation, Synthesis and Formal Proofs of Hardware, pp. 149-161, Dordrecht, The Netherlands : Kluwer Academic Publishers, 1992.

[3] K. Khordoc, M. Dufresne, E. Cerny, P. Babkine, A. Silburt, "*Integrating Behavior and Timing in Executable Specifications*", to appear in CHDL-93.

[4] S. Sutarwala, P. Paulin and Y. Kumar, "*Insulin : An Instruction Set Simulation Environment*", to appear in CHDL-93.

[5] W. Loucks, B. Doray and D. Agnew, "*Experiences in Real-Time Hardware-Software Co-Simulation*", to appear in VIUF-93.

[6] Motorola, "*56156 DataBook*".