By Donna Mitchell,
is vice president of strategic marketing,
SynaptiCAD Inc.

Manual and Automatic VHDL/Verilog Test Bench Coding Techniques

One of the most time consuming tasks for users of HDL languages is coding test benches to verify the operation of their design. In his book "Writing Testbenches," Janick Bergeron estimates that 70% of design time is spent verifying HDL code models and that the test bench makes up 80% of the total HDL code generated during product development. In this paper we propose the use of automatic code generation tools to reduce the time required to create and maintain test benches. In particular, we will discuss TestBencher Pro an automatic code generation tool for VHDL and Verilog test benches.

One of the most time consuming tasks for users of HDL languages is coding test benches to verify the operation of their design. In his book "Writing Testbenches," Janick Bergeron estimates that 70% of design time is spent verifying HDL code models and that the test bench makes up 80% of the total HDL code generated during product development. In this paper we propose the use of automatic code generation tools to reduce the time required to create and maintain test benches.

In particular, we will discuss TestBencher Pro an automatic code generation tool for VHDL and Verilog test benches. TestBencher Pro automates the most tedious aspects of test bench development, allowing you to focus on the design and operation of the test bench. This is accomplished by representing each bus transaction graphically and then automatically generating the code for each transaction. TestBencher makes use of the powerful features of the language that is being generated and the engineer does not have to hand-code each transaction. When hand coding, the designer would have to take the time to deal with the specifics of the design (port information, monitoring system response, etc) as well as common programming errors (race conditions, minor logic errors, and code design problems). This removes a considerable amount of time from the test bench design process because TestBencher manages the low-level details and automatically generates a valid test bench.

TEST BENCH TECHNIQUES

Designers usually have three choices when it comes to implementing a test bench model:

1. purchase or write a **complete functional model** that models the internal operation of the component being modeled.
2. write a **bus-functional model** (BFM) that describes the behavior of the part at the interface-level (bus transaction level) without modeling the internal operation of the part.
3. write **raw test vectors** that describe input to the MUT and expected outputs from the MUT as patterns of digital data.

When writing complete functional models for complex parts, users face two major problems: (1) information about internal component operation of complex parts is generally vendor proprietary and not available to the end user, and (2) complete functional models for complex parts are large and require too much time to code and debug. Even when you are able to purchase a functional model, the increase in simulation times incurred because of the large models are usually too much to justify the cost using the model. Raw test vectors are adequate for testing small systems, but as the test suite size increases it becomes more difficult to maintain and write the raw data. One of the reasons behind the development of HDL languages was to provide ability to write behavioral models that did not necessary represent synthesizable circuits.

Bus functional models, BFMs, overcome the difficulties of creating complete functional models because they can be created directly from the interface information contained in data sheets. In addition, BFMs are much smaller than complete models because they only emulate the interface rather than the entire part so the resulting models simulate much faster.

BFMs overcome the difficulties of creating and maintaining raw test vector data, because of the modular design techniques used to create the models. BFMs can verify correct functioning of complex behaviors required by the model under test, and react to output from the model under test by applying differing stimulus sets based on that output. These benefits have made bus-functional models the preferred choice for writing HDL test benches for testing complex systems (although test vector-based test benches are still suitable for testing designs with simple architectures).

DIFFICULTIES OF WRITING BUS FUNCTIONAL MODELS

Despite these benefits, manually coded bus functional models still suffer from one of the major disadvantages of manually coded test vectors: they are difficult to write and they can only be debugged at simulation time. Debugging bus-functional models is actually more difficult than debugging raw test vector test benches

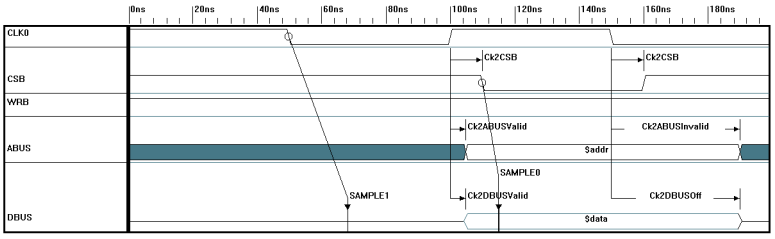

Figure 1. Timing diagram of Read Transaction.

because the reactivity of the bus-functional model makes it more difficult to check all the BFMs possible behaviors and isolate errors in the test bench from errors in the model under test. A typical BFM might be used to model a microprocessor's interactions with an IO device or memory subsystem. This type of BFM would need to generate control and data signals from the microprocessor for each type of microprocessor bus transaction (e.g. read cycle, write cycle, interrupt processing) with exact timing requirements. Verifying MUT output and reacting to MUT output also adds complexity to BFMs. For example, a microprocessor BFM might have to wait for a data valid signal from a memory subsystem before completing a read cycle.

Most BFMs are currently hand-coded in HDL from signal interface and timing information (timing diagrams) published by component vendors. This type of HDL code is difficult to create and maintain because it is difficult to visualize the waveforms generated by the HDL code and compare them to vendor timing diagrams. Recently, graphical specification tools such as TestBencher Pro by SynaptiCAD Inc have emerged to simplify this process. TestBencher Pro generates HDL test bench code from user-entered timing diagrams that describe a test bench's bus transactions.

DESIGN OF A BFM-BASED TEST BENCH

Several different styles can be used to code BFM-based test benches, but the initial design process is the same for each approach. A BFM only needs to model a small portion of the complete functionality of a chip since none of the internal operation is modeled and only some aspects of external operation typically need to be modeled. For example, a BFM for a microprocessor to test a memory interface design will only need to model the microprocessor's read/write bus transactions. Such a model will only need to generate and verify waveform transitions for the signals involved in those transactions. Therefore the first step in designing a BFM is to decide what transactions need to be modeled to test the MUT and to obtain descriptions of those transactions (usually provided by manufacturers in the form of timing diagrams). Figure 1 shows the timing diagram for the read transaction we will model in this article.

After selecting the transactions to model, the next step is to write HDL code for each transaction type. Transactions should be coded as reusable, parameterized pieces as test benches usually require multiple uses of a transaction with differing data values. For example, a test bench for a memory subsystem will typically read and write differing data bus values to differing address locations within the memory. The choice of HDL determines what types of reusable blocks are available for modeling transactions. For the rest of this article, we will discuss the writing of a BFM for VHDL, but many of the issues are the same for Verilog as well. A basic understanding of VHDL syntax is assumed for the rest of this article.

PROCEDURE-BASED AND COMPONENT-BASED TRANSACTION MODELING

In VHDL, either procedures or components should be used for representing bus transactions because these constructs are reusable and can contain sequential statements that allow for the passage of simulation time (VHDL functions can't be used for this reason). Procedures are easier to use than components because less coding is required and invoking the transaction is simpler. Procedure-based transactions can be invoked by a simple procedure call whereas extra control signals must be added to a component and triggered from the calling location to initiate a component's transaction. Care should be taking when creating a control signal scheme, because the concurrent nature of the code makes it easy to leave a race condition in your design.

Component-based transactions (CBTs), however, do offer two important advantages over procedure-based transactions (PBTs) that stem from the fact that procedures must be executed within the context of a single process: (1) CBTs can contain multiple internal processes, allowing the modeling of concurrently-executed sections within a transaction, and (2) CBTs can be triggered asynchronously from within an executing transaction without causing the executing transaction to suspend execution whereas execution of a PBT causes its triggering transaction to suspend until the PBT finishes executing. These features often make component-based transactions a more appropriate choice for representing complex parts which have concurrent execution requirements.

Figure 2 contains the VHDL source for a component-based model of the read cycle transaction (tbread) in Figure 1. Several of the signals in the entity declaration are lower-cased to indicate that they are either sequencing control signals (trigger and status) or data parameter signals (addr and data) for the bus transaction and do NOT directly connect up to the MUT (note: this is just a coding convention, VHDL compilers are not case-sensitive). Sequencing control signals are necessary to initiate the component's transaction and

are also useful during debug for determining the current execution status of a test bench. Figure 2 demonstrates one approach to sequencing control; other realizations are possible. Sequencing signals are not needed for procedure-based transactions since procedure-based transactions can be called directly. Data parameter signals are used to set the parameters of the transaction that vary across transaction invocations. Passing of data parameters in procedure-based transactions is usually performed using variables rather than signals, but variables are not allowed in component declarations so signals must be used to pass the parameters.

PROCESSES WITHIN THE READ CYCLE TRANSACTION

The architecture body of the read transaction contains two processes: the update_status process and the main process. The update_status process controls execution of the transaction based on the trigger signals, and updates the status signal to set the current execution state of the transaction cycle. Two trigger signals are used to avoid contention problems that result when multiple drivers attempt to drive a VHDL signal. Signal trigger is driven by external components to invoke the transaction and signal int_trigger is an internal trigger driven by the transaction component to modify its own transaction state (e.g. to abort or complete the transaction). Whenever either of these signals changes, the update_status process executes, and updates the status signal to reflect the new value of the trigger signal that changed. Valid execution states are WAITING to execute, executing ONCE, execution DONE, and LOOPING (repeatedly executing).

The main process contains the code that presents test vectors to the MUT, verifies output from the MUT, logs information about test bench execution, and reacts to output from the MUT. Since the main process has no sensitivity list it begins execution as soon the simulation is started. It sets its internal trigger signal to WAITING, tristates any signals which it should only drive while executing (more on this later), and then suspends execution until its status signal is triggered externally. When the status flag is set to ONCE or

```
 1  library ieee,std;
 2  use ieee.std_logic_1164.all;
 3  use work.TBDefinitions.all;
 4
 5  entity tbread is
 6    port(
 7         trigger : in TStatus;
 8         status : inout TStatus;
 9         CSB : out std_logic;
10         WRB : out std_logic;
11         ABUS : out std_logic_vector(15 downto 0);
12         addr : in std_logic_vector(15 downto 0);
13         DBUS : inout std_logic_vector(15 downto 0);
14         data : in std_logic_vector(15 downto 0)
15        );
16  end tbread;
17
18  architecture Diagram of tbread is
19    use std.textio.all;
20    use ieee.std_logic_textio.all;
21    use ieee.std_logic_arith.all;
22
23    signal int_trigger : TStatus;
24
25  begin
26
27    update_status : process(trigger,int_trigger)
28    begin
29      if (trigger'event) then
30        if (trigger /= WAITING) then
31          status <= trigger;
32        end if;
33      end if;
34      if (int_trigger'event) then
35        if (int_trigger /= WAITING) then
36          status <= int_trigger;
37        end if;
38      end if;
39    end process update_status;
40
41    main : process --main process for individual diagram
42      variable tbSampledValue : line;
43      variable SAMPLE0 : boolean;
44      variable value_SAMPLE0 : std_logic_vector(15 downto 0);
45      variable SAMPLE1 : boolean;
46      variable value_SAMPLE1 : std_logic_vector(15 downto 0);
47    begin
48      mainloop :
49      loop
50        int_trigger <= WAITING;
51        DBUS <= "ZZZZZZZZZZZZZZZZ";
52        wait until (status = ONCE) or (status = LOOPING);
53          CSB <= '1';
54          WRB <= '1';
55          ABUS <= "XXXXXXXXXXXXXXXX";
56          DBUS <= "ZZZZZZZZZZZZZZZZ";
57        wait for 68.096 ns;
58          value_SAMPLE1 := DBUS;
59          deallocate(tbSampledValue);
60          write(tbSampledValue,value_SAMPLE1);
61          SAMPLE1 := not (DBUS = "ZZZZZZZZZZZZZZZZ");
62          if (SAMPLE1) then
63            tbLog("On DBUS, expected ""ZZZZZZZZZZZZZZZZ"":
              detected " & tbSampledValue.all, WARNING);
64            assert FALSE
65            report "On DBUS, expected ""ZZZZZZZZZZZZZZZZ"":
              detected " & tbSampledValue.all
66              severity WARNING;
67          end if;
68        wait for 36.904 ns;
69          ABUS <= addr;
70          DBUS <= "ZZZZZZZZZZZZZZZZ";
71        wait for 5 ns;
72          CSB <= '0';
73        wait for 5 ns;
74          value_SAMPLE0 := DBUS;
75          deallocate(tbSampledValue);
76          write(tbSampledValue,value_SAMPLE0);
77          SAMPLE0 := not (DBUS = data);
78          if (SAMPLE0) then
79            tbLog("On DBUS, expected data: detected " &
              tbSampledValue.all, WARNING);
80            assert FALSE
81            report "On DBUS, expected data: detected " &
              tbSampledValue.all
82            severity WARNING;
83          end if;
84        wait for 45 ns;
85          CSB <= '1';
86        wait for 30 ns;
87          ABUS <= "XXXXXXXXXXXXXXXX";
88          DBUS <= "ZZZZZZZZZZZZZZZZ";
89        wait for 50 ns;
90        if (status = ONCE) then
91          int_trigger <= DONE;
92          wait until (status = DONE);
93        end if;
94      end loop;
95    end process main;
96
97  end Diagram;
```

Figure 2. Read Cycle Transaction Component.

LOOPING, the component begins executing the bus transaction; applying test vectors and checking MUT output.

METHODS FOR CODING TEST VECTORS WITHIN THE TRANSACTION MODEL

Test vectors are typically coded using one of three methods: (1) a single signal assignment statement for each signal that queues up all the transitions for that signal during the bus transaction (see figure 7), (2) sets of signal assignment statements separated by wait statements every time a signal transitions (method used in figure 2), and (3) reading of the test vector data and/or wait times from an external file. Single assignment statements are acceptable for use in simple test-vector based test benches because they form a compact representation of the test vectors, but they are problematic for BFM-based test benches because it is difficult to determine the cause of signal transitions during single-step debug of the test bench and because the predefined delivery of all the transitions at once makes it difficult to change the state and time of transitions on a signal in response to output from the MUT. Therefore either method 2 or 3 should be used for BFM-based test benches. File-based test vectors offer the advantage of being able to change the test vectors during debug without recompiling the test bench, but they do require a little more coding overhead to read the external file.

VERIFYING AND REACTING TO OUTPUT FROM THE MODEL UNDER TEST

The blue signal sections in Figure 2 represent output values from the MUT. The "Sample" points on the diagram indicate checkpoints in the bus transaction at which the MUT output will be tested. The code on lines 61-67 of Figure 2 checks DBUS output from the MUT to verify that the lines are initially tri-stated and logs an error message and asserts a warning if the DBUS lines are not tri-stated. The code on lines 78-83 also checks DBUS output, but this time against a parameterized data value rather than one hard-coded into the transaction model.

One important aspect of this transaction model is that the checking code is contained in the same process as the test vectors. The advantage of this approach is that application of test vectors and the checking for responses to those test vectors is automatically kept synchronized.

DRIVING SIGNALS WITH PARAMETERIZED DATA VALUES

Line 69 demonstrates the use of a parameterized data value to drive a signal to the MUT. In this particular case, the code allows the re-use of the transaction with differing address values. This address value will be specified in an external component in the test bench prior to the triggering of the transaction (see the section on transaction trigger procedures below for more details).

```
 1  library ieee,std;
 2  use ieee.std_logic_1164.all;
 3  use work.TBDefinitions.all;
 4
 5  entity TestBenchSequencer is
 6    port(
 7      apply_tbwrite_1_addr : out std_logic_vector(15 downto 0);
 8      apply_tbwrite_1_data : out std_logic_vector(15 downto 0);
 9      trigger_tbwrite_1 : out TStatus;
10      status_tbwrite_1 : in TStatus;
11      apply_tbread_1_addr : out std_logic_vector(15 downto 0);
12      apply_tbread_1_data : out std_logic_vector(15 downto 0);
13      trigger_tbread_1 : out TStatus;
14      status_tbread_1 : in TStatus
15      );
16  end TestBenchSequencer;
17
18  architecture tbMain of TestBenchSequencer is
19    use std.textio.all;
20    use ieee.std_logic_textio.all;
21    use ieee.std_logic_arith.all;
22  begin
23    process          --top level testbench process
24
25      procedure call_tbwrite_1(in runMode  : TStatus,
26                               in waitMode : TWaitMode)
27      begin
28        trigger_tbwrite_1 <= runMode;
29        wait until (status_tbwrite_1 = runMode);
30        trigger_tbwrite_1 <= WAITING;
31        if (waitMode = WAIT) then
32          wait until (status_tbwrite_1 = DONE);
33        end if;
34      end;
35
36      procedure call_tbread_1(in runMode  : TStatus,
37                              in waitMode : TWaitMode)
38      begin
39        trigger_tbread_1 <= runMode;
40        wait until (status_tbread_1 = runMode);
41        trigger_tbread_1 <= WAITING;
42        if (waitMode = WAIT) then
43          wait until (status_tbread_1 = DONE);
44        end if;
45      end;
46
47    begin
48          apply_tbwrite_1_addr <= "1111000000000000";
49          apply_tbwrite_1_data <= "1010101011101110";
50          call_tbwrite_1(ONCE,WAIT);
51
52          apply_tbread_1_addr <= "1111000000000000";
53          apply_tbread_1_data <= "1010101011101110";
54          call_tbread_1(ONCE,DONE);
55
56          apply_tbread_1_addr <= "1111000000000000";
57          apply_tbread_1_data <= "1110111011101110";
58          call_tbwrite_1(ONCE,WAIT);
59      wait;
60    end process;
61
62  end tbMain;
```

Figure 3. Test Bench Sequencer Component.

ENDING THE BUS TRANSACTION

Lines 90-93 check the status signal to see if the transaction is to be executed ONCE or repeatedly (LOOPING). If the transaction is to be run once, the status is set to DONE so that during the next iteration of main-loop, the transaction will stall on the wait statement that checks status. Otherwise, status will stay set to LOOPING and the transaction will re-execute (the statement that sets int_trigger to WAITING on line 50 will not affect the status flag because the update_status routine doesn't update the status flag when a trigger signal is set to WAITING).

```
 1  entity testbench is
 2  end testbench;
 3
 4  library ieee,std;
 5  architecture TB of testbench is
 6  use std.textio.all;
 7  use ieee.std_logic_1164.all;
 8  use work.TBDefinitions.all;
 9  use work.all;
10
11  signal apply_tbwrite_1_addr : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
12  signal apply_tbwrite_1_addr_0 : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
13  signal apply_tbwrite_1_data : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
14  signal apply_tbwrite_1_data_0 : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
15  signal trigger_tbwrite_1 : TStatus;
16  signal trigger_tbwrite_1_0 : TStatus;
17  signal status_tbwrite_1 : TStatus;
18  signal apply_tbread_1_addr : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
19  signal apply_tbread_1_addr_0 : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
20  signal apply_tbread_1_data : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
21  signal apply_tbread_1_data_0 : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
22  signal trigger_tbread_1 : TStatus;
23  signal trigger_tbread_1_0 : TStatus;
24  signal status_tbread_1 : TStatus;
25  signal CSB : std_logic := 'Z';
26  signal WRB : std_logic := 'Z';
27  signal ABUS : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
28  signal CSB_0 : std_logic := 'Z';
29  signal WRB_0 : std_logic := 'Z';
30  signal ABUS_0 : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
31  signal DBUS : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
32  signal CSB_1 : std_logic := 'Z';
33  signal WRB_1 : std_logic := 'Z';
34  signal ABUS_1 : std_logic_vector(15 downto 0):
      = "ZZZZZZZZZZZZZZZZ";
35
36  --Component configurations
37
38    component TestBenchSequencer
39      port(
40          apply_tbwrite_1_addr : out std_logic_vector(15 downto 0);
41          apply_tbwrite_1_data : out std_logic_vector(15 downto 0);
42          trigger_tbwrite_1 : out TStatus;
43          status_tbwrite_1 : in TStatus;
44          apply_tbread_1_addr : out std_logic_vector(15 downto 0);
45          apply_tbread_1_data : out std_logic_vector(15 downto 0);
46          trigger_tbread_1 : out TStatus;
47          status_tbread_1 : in TStatus
48           );
49    end component;
50  --  for all: TestBenchSequencer
51  --          use entity TestBenchSequencer(Diagram);
52
53    component tbsram
54      port(
55          CSB : in std_logic;
56          WRB : in std_logic;
57          ABUS : in std_logic_vector(15 downto 0);
58          DATABUS : inout std_logic_vector(7 downto 0)
59           );
60    end component;
61  --  for all: tbsram
62  --          use entity tbsram(Diagram);
63
64    component tbwrite
65      port(
66          trigger : in TStatus;
67          status : inout TStatus;
68          CSB : out std_logic;
69          WRB : out std_logic;
70          ABUS : out std_logic_vector(15 downto 0);
71          addr : in std_logic_vector(15 downto 0);
72          DBUS : inout std_logic_vector(15 downto 0);
73          data : in std_logic_vector(15 downto 0)
74           );
75    end component;
76  --  for all: tbwrite
77  --          use entity tbwrite(Diagram);
78
79    component tbread
80      port(
81          trigger : in TStatus;
82          status : inout TStatus;
83          CSB : out std_logic;
84          WRB : out std_logic;
85          ABUS : out std_logic_vector(15 downto 0);
86          addr : in std_logic_vector(15 downto 0);
87          DBUS : inout std_logic_vector(15 downto 0);
88          data : in std_logic_vector(15 downto 0);
89           );
90    end component;
91  --  for all: tbread
92  --          use entity tbread(Diagram);
93
94  begin
95
96    sequencer : TestBenchSequencer
97          port map(
98          apply_tbwrite_1_addr_0,
99          apply_tbwrite_1_data_0,
100         trigger_tbwrite_1_0,
101         status_tbwrite_1,
102         apply_tbread_1_addr_0,
103         apply_tbread_1_data_0,
104         trigger_tbread_1_0,
105         status_tbread_1);
106   mutMSB : tbsram
107         port map(
108         CSB,
109         WRB,
110         ABUS,
111         DBUS(15 downto 8));
112   mutLSB : tbsram
113         port map(
114         CSB,
115         WRB,
116         ABUS,
117         DBUS(7 downto 0));
118   tbwrite_1 : tbwrite
119         port map(
120         trigger_tbwrite_1,
121         status_tbwrite_1,
122         CSB_0,
123         WRB_0,
124         ABUS_0,
125         apply_tbwrite_1_addr,
126         DBUS,
127         apply_tbwrite_1_data);
128   tbread_1 : tbread
129         port map(
130         trigger_tbread_1,
131         status_tbread_1,
132         CSB_1,
133         WRB_1,
134         ABUS_1,
135         apply_tbread_1_addr,
136         DBUS,
137         apply_tbread_1_data);
138
139 --Update shared outputs from tributary outputs
140   process(
141           apply_tbwrite_1_addr_0,
142           apply_tbwrite_1_data_0,
143           trigger_tbwrite_1_0,
144           apply_tbread_1_addr_0,
145           apply_tbread_1_data_0,
146           trigger_tbread_1_0,
147           CSB_0,
148           WRB_0,
149           ABUS_0,
150           CSB_1,
151           WRB_1,
152           ABUS_1,
```

```
153          )
154    begin
155      if (apply_tbwrite_1_addr_0'event) then
156        apply_tbwrite_1_addr <= apply_tbwrite_1_addr_0;
157      end if;
158      if (apply_tbwrite_1_data_0'event) then
159        apply_tbwrite_1_data <= apply_tbwrite_1_data_0;
160      end if;
161      if (trigger_tbwrite_1_0'event) then
162        trigger_tbwrite_1 <= trigger_tbwrite_1_0;
163      end if;
164      if (apply_tbread_1_addr_0'event) then
165        apply_tbread_1_addr <= apply_tbread_1_addr_0;
166      end if;
167      if (apply_tbread_1_data_0'event) then
168        apply_tbread_1_data <= apply_tbread_1_data_0;
169      end if;
170      if (trigger_tbread_1_0'event) then
171        trigger_tbread_1 <= trigger_tbread_1_0;
172      end if;
173      if (CSB_0'event) then
174        CSB <= CSB_0;
175      end if;
176      if (WRB_0'event) then
177        WRB <= WRB_0;
178      end if;
179      if (ABUS_0'event) then
180        ABUS <= ABUS_0;
181      end if;
182      if (CSB_1'event) then
183        CSB <= CSB_1;
184      end if;
185      if (WRB_1'event) then
186        WRB <= WRB_1;
187      end if;
188      if (ABUS_1'event) then
189        ABUS <= ABUS_1;
190      end if;
191    end process;
192
193  end TB;          -- end testbench
```

Figure 4. Top-level test bench structural component.

TRANSACTION TRIGGER PROCEDURES AND THE SEQUENCER COMPONENT

Figure 3 contains the source code for the sequencer component that describes the order in which bus transactions will be executed. A bus transaction is initiated by calling a transaction trigger procedure (e.g. call_tbread_1). The runMode parameter of the trigger procedure specifies whether to execute the transaction once or continously. The waitMode parameter specifies whether or not the sequencer should wait until the end of the bus transaction before starting a new transaction. The ability of the sequencer component to continue executing after triggering a bus transaction is one of the advantages of component-based transaction models. If a call was made to procedure-based transaction, the sequencer component could only resume execution when the transaction procedure was completed.

TRIGGERING A NEW TRANSACTION FROM WITHIN A TRANSACTION MODEL

All the bus transactions in the test bench in this article are invoked from the sequencer component, but in more complex test benches one transaction may initiate a new transaction by making a call to the new transaction's trigger procedure. For example, if a serious error occurred during a transaction, the transaction could respond by performing a system reset transaction.

CONSTRUCTING A TOP-LEVEL STRUCTURAL MODEL TO CONNECT THE BFMS TO THE MUT

The final part of the BFM test bench is the top-level structural model (Figure 4) that instantiates the transaction components and model under test. It also describes the interconnection of the transaction models and model under test. In this test bench a sequencer component, 2 SRAM components (MUTs), and a read and write transaction component are instantiated. If the test bench required simultaneous execution of two bus transactions of the same type (e.g. test bench modeled two microprocessors performing simultaneous reads to different memory banks), then two components would need to be instantiated for that transaction type.

```
 1  library ieee,std;
 2  use std.textio.all;
 3  use ieee.std_logic_1164.all;
 4
 5  package TBdefinitions is
 6    type TStatus is (WAITING,ONCE,DONE,LOOPING);
 7    procedure tbLog(
 8                  constant str   : in string;
 9                  constant level : in severity_level := WARNING
10                  );
11    function tbSeverityToString(
12                            constant level : in severity_level
13                            ) return string;
14  end TBdefinitions;
15
16  package body TBdefinitions is
17
18    file LOGFILE : text is out "tbtut.log";
19
20    procedure tbLog(
21                  constant str   : in string;
22                  constant level : in severity_level := WARNING
23                  ) is
24      variable logline : line;
25    begin
26      write(logline, string'("TestBencher: "));
27      write(logline, tbSeverityToString(level));
28      write(logline, string'(" At "));
29      write(logline, NOW);
30      write(logline, string'(": "));
31      write(logline, str);
32      writeline(LOGFILE, logline);
33    end tbLog;
34
35    function tbSeverityToString(
36                            constant level : in severity_level
37                            ) return string is
38    begin
39      case level is
40        when NOTE =>
41          return string'("NOTE");
42        when WARNING =>
43          return string'("WARNING");
44        when ERROR =>
45          return string'("ERROR");
46        when FAILURE =>
47          return string'("FAILURE");
48      end case;
49    end tbSeverityToString;
50  end TBdefinitions;
```

Figure 5. Test bench types and helper routines.

```
1  --- Queues up three signal transitions in
2  --- one VHDL statement (not recommended).
3    DBUS <= transport "ZZZZZZZZ",
4                "11011011" after 110 ns,
5                "ZZZZZZZZ" after 210 ns;
```

Figure 6. Example of a single assignment statement

AVOIDING TOP-LEVEL SIGNAL CONTENTION USING RESOLVED SIGNALS OR SHARED SIGNALS

Multiple transaction models often need to drive a common set of signals in the top level test bench model. For example, in this testbench, the read and write transactions both need to drive the address lines to the MUT. Each transaction component contains its own signal drivers, so some means must be provided in the test bench for avoiding contention between the drivers.

One way to avoid signal contention is to use resolved signals. The declaration of a resolved signal includes a reference to VHDL procedure that checks the state values of all drivers on the signal and determines what the resulting value of the signal should be. The IEEE standard signal type std_logic is an example of a commonly used resolved signal type. When std_logic signals are used to resolve signal contention, the transaction models in the test bench should be coded to tri-state their output signals at the end of the transaction.

The primary problem with using resolved signals to resolve signal contention in a BFM test bench is the need to tri-state signals at the end of each transaction. It is often desirable to allow signals to remain driven until another transaction needs to drive the signals (e.g. clock signals shouldn't be tri-stated). One way to allow this is to avoid signal contention using "shared" signals. In a "shared" signal approach, each transaction drives its own copy (e.g. ABUS_1, ABUS_2) of the shared signal (e.g. ABUS) which connects up to the MUT. A top-level process examines each copy of the signal (referred to as tributaries) and sets the value of the shared signal to the value of the last tributary signal that has updated. Lines 141-193 of Figure 4 contain a process that updates several shared signals.

CONCLUSION

BFM test benches provide several benefits over traditional test vector based test benches; the most important being automated verification of MUT functionality and the ability to react to output from the model under test. The disadvantage to the use of bus functional models is that they are more complex and take longer to code and debug than test vectors. There are a number of design decisions to be made when writing BFM-based test benches that result in tradeoffs between test bench complexity, performance, and flexibility.

The recent emergence of graphical tools for describing BFM-based test benches simplifies creation of test benches that are nearly as efficient as hand-coded test benches and contain fewer errors because of the graphical feedback they provide. As an example, the test bench code discussed in this article consists of excerpts from a test bench generated by TestBencher Pro from graphically-entered timing diagrams and a transaction sequence script file. This software and a tutorial are included on the CD enclosed with this magazine.

Although HDL test benches are written by virtually every HDL user, there is not a lot of practical information about writing test benches in the existing HDL literature. This article only covers a few issues related to test bench coding. Below are some references to other aspects of VHDL test bench coding ■

Donna Mitchell is vice president of strategic marketing at SynaptiCAD Inc. She received her BS and MS degrees in electrical engineering from Virginia Tech. Mitchell is one of the two founders of SynaptiCAD Inc. She co-developed SynaptiCAD's first product, a timing diagram editor called Timing Diagrammer. Mitchell's industry experience includes 14 years of hardware and software development. Prior to her work at SynaptiCAD, she designed board-level mixed- signal systems at Analog Devices and Burr Brown.

REFERENCES

- Writing Testbenches, Janick Bergeron
- VHDL Coding Guide and Methodologies, Ben Cohen, 1995. Chapter 10 discusses test benches.
- VHDL Made Easy, David Pellerin, 1996. Chapter and appendix on writing test benches.
- SynaptiCAD maintains info about test bench coding automation tools at http://www.syncad.com.