 By Subbu Meiyappan and James Steele, VLSI Technology

THE USE OF ADVANCED VHDL CONSTRUCTS CAN GREATLY ENHANCE MODELING EFFICIENCY. LEARN HOW TO EFFECTIVELY USE VHDL FOR DYNAMIC-MEMORY ALLOCATION, HIERARCHICAL TESTBENCHES, AND CREATING FOREIGN-LANGUAGE INTERFACES FOR BEHAVIORAL MODELING.

VHDL constructs and methodologies for advanced-design verification

THE NEED TO MODEL MEMORY is common in any verification environment. The simplest way to model memory is to employ static allocation techniques using array data types. For example, to model a width-times-depth-sized memory, you use a construct similar to the one shown in **Listing 1**.

For small memories, static-allocation techniques are the simplest to read, understand, and implement. The static-allocation method allocates the required memory before the program accesses memory—in other words, when the simulator starts up (at zero simulation time). This method has some inherent drawbacks: Every bit of allocated memory corresponds to several bits of physical memory; thus, simulator performance is limited for large memories.

Consider a processor with a 32-bit-wide address bus. Theoretically, the processor can access 4 Gbytes of memory. If you assume, while simulating the model of this processor, that every bit of allocated memory corresponds to 1 bit of physical memory, then a static-allocation scheme that allocates all of this memory requires about 4 Gbytes of RAM (In a real case, every bit of allocated memory corresponds to several times that amount.) If the required RAM is unavailable, the program translates the memory accesses into disk accesses, severely slowing a simulation run. On a statistical note, no single typical simulation run requires all 4 Gbytes of memory. Even if you allocate all 4 Gbytes, each simulation run typically hits 10% or less of the memory space, although these accesses can be random. In other words, it is typical for a simulation to toggle all the address bits but not access every byte of memory. Because static-allocation techniques for such deep memories greatly limit simulation performance, you should use methods to model memory efficiently such that the program allocates it in an "on-demand" basis using dynamic-allocation techniques. With dynamic allocation, the program allocates physical memory on demand on the host machine as the model simulates new transactions.

LISTING 1—SIMPLE MEMORY MODEL USING STATIC ALLOCATION

```
subtype mem_width_t is std_logic_vector(WIDTH-1 downto 0);
subtype mem_depth_t is integer range 0 to DEPTH-1;
type mem_t is array(mem_depth_t) of mem_width_t;
variable my_mem : mem_t;
```

The approach described in this article is one of the classic methods that developers have used for computational problems that use data structures with statically nondeterministic depths. You usually solve such problems by implementing linked lists that a program creates and builds dynamically on demand. On-demand creation of a linked list of data

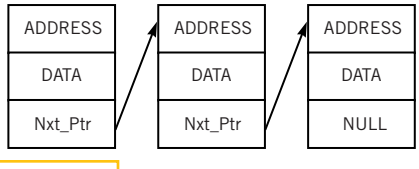

Figure 1 This flow chart demonstrates a linked-list organization for a memory model.

structures and allocation of new memory locations relies heavily on dynamic-memory allocation and pointer-manipulation techniques. VHDL, using *access* data types, provides a mechanism to allocate new memory locations on demand along with pointer arithmetic to manipulate the pointers. **References 1** and **2** detail dynamic-memory allocation using access data types and linked lists. Designers use an application of access data types to dynamically create linked lists for efficient memory modeling. Note that the access data types are nonsynthesizable VHDL constructs, so you can use them only for simulations.

Listings 2 and **3** illustrate the implementation of a complete VHDL package for sparsely allocated memories. This method of modeling memory is efficient in random-simulation environments in which the program randomly accesses memory. **Listing 2** is the VHDL description of the package that encompasses the primitives and prototypes for creating such a memory. **Listing 3** is the VHDL description of the package body that implements this memory. The example is split between two listings to simplify the underlying details of the descriptions. To understand these details, you should have a basic knowledge of VHDL; the article explains these details with references to line numbers in the code.

Addr0	Data0
Addr1	Data1
Addr2	Data2
.	.
.	.
.	.
AddrN	DataN

Figure 2 From a user's perspective, the flow chart in Figure 1 looks like a contiguously located RAM.

Figure 1 demonstrates how to organize the linked list for the memory. Every node in the list contains an address, a data item to store, and a pointer to the next node in the linked list. From a user's perspective, this memory looks like a contiguously located RAM (**Figure 2**).

Every time the program accesses a new address during write operations, it creates a new node with the address, data, and a pointer to the next node in the linked list. During read operations, the program traverses the list, trying to match the corresponding read address. If it finds an address match, the program returns the data in the node corresponding to that address. The program stores each node as a *record* data type that it creates dynamically on demand. The *Nxt_Ptr* must be of an *access* type the program uses to point to the next node. A first attempt to write the type declarations for this structure might look like the code in **Listing 4**.

Listing 4 shows that the definition of *mem_entry* uses the type *entry_ptr* as the data type of one of its elements but does not declare *entry_ptr* until after the definition of mem_entry. To solve this "chicken-or-the-egg" problem, **Listing 2** shows the incomplete-type declaration in line 10; lines 11 through 16 create some type definitions for the linked-list data structure. The *Mem_entry* record contains the location, the data, and the pointer to the next node in the list.

The idea behind linked lists is to keep track of the pointers. You need to create a pointer that keeps track of the top of the linked list, so that you can traverse the tree from that location. To do this task, you create *head_ptr* in lines 19 through 22 (**Figure 3**). *Head_ptr* always points to the topmost node in the list. The *Nxt_Ptr* in the last node list is a null pointer that indicates the end of the list.

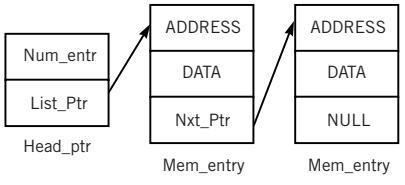

Figure 3 This two-node linked-data structure shows how you keep track of pointers in a linked list.

LISTING 2–VHDL PACKAGE CONTAINING MEMORY PROTOTYPES AND PRIMITIVES

```
1.  ------------------------------------------------------------
2.  Memory Allocation Package - mem_pkg
3.  ------------------------------------------------------------

4.  library ieee;
5.  use ieee.std_logic_1164.all;

6.  package mem_pkg is

7.  --********************
8.  --** Declare types **
9.  --********************
10. type mem_entry ;
11. type entry_ptr is access mem_entry;

12. type mem_entry is record
13.   address          : integer;
14.   data             : std_logic_vector(15 downto 0);
15.   nxt              : entry_ptr;
16. end record;

17. -- To help in finding an address, we use the record
18. -- "head" to point to the first element of the list.
19. type head is record
20.   num_entries      : integer;
21.   list_ptr         : entry_ptr;
22. end record;

23. type head_ptr is access head;

24. procedure wr_data (
25.   constant location: in integer;
26.   constant data     : in std_logic_vector;
27.   variable first    : inout head_ptr
28. ) ;

29. procedure rd_data (
30.   constant location  : in integer;
31.   variable data      : out std_logic_vector ;
32.   variable allocated : out boolean;
33.   variable first     : inout head_ptr
34. ) ;

35. end; -- mem_pkg
```

The use of the *num_entries* in the head pointer is to provide hooks for creating a searchable linked list using an algorithm, such as binary sort. The binary-sort algorithm can benefit from knowing the number of entries in a list. You arrange the linked list as a linear-data structure, in which the physical values of the addresses in the linked list are in ascending order. For example, if you allocate three locations in memory—0x0011, 0x0F00, and 0xFFFF—the first address in the list is 0x0011, and the last address in the list is 0xFFFF. Throughout this article, memory is organized to hold 16 bits of data for each address.

Listing 3 defines the *mem_pkg* package. The body of the package contains the implementations for the *wr_data* and *rd_data* procedures. The *wr_data* procedure accepts the address (location), the data to be stored in that address, and the pointer to the head of the linked list. Five scenarios exist in which a write to memory can occur. Assume that a memory write to the following locations in the following order takes place:

LISTING 3–VHDL PACKAGE DESCRIPTION IMPLEMENTING MEMORY OF LISTING 2

```
1.  package body mem_pkg is
2.  procedure wr_data (
3.    constant location: in integer;
4.    constant data    : in std_logic_vector;
5.    variable first   : inout head_ptr
6.  ) is
7.    variable temp_ptr    : entry_ptr;
8.    variable new_ptr     : entry_ptr;
9.    variable prev_ptr : entry_ptr;
10.   variable done        : boolean := false;
11. begin
12.    -- set done to true when allocation occurs
13. done := false;
14.   -- first access to memory
15. if first.num_entries = 0 then
16.     first.list_ptr := new mem_entry;
17.     first.num_entries := 1;
18.     first.list_ptr.address := location;
19.     first.list_ptr.data := data;
20.     first.list_ptr.nxt := null;
21.     done := true;
22. -- address is lowest value so far in allocation to put
23. -- at head of list
24. elsif location < first.list_ptr.address then
25.     new_ptr := new mem_entry;
26.     new_ptr.data := data;
27.     new_ptr.nxt := first.list_ptr;
28.     new_ptr.address := location;
29.     first.list_ptr := new_ptr;
30.     first.num_entries := first.num_entries + 1;
31.     done := true;
32.   else   -- location must be >=first.list_prt.address
33.     temp_ptr := first.list_ptr;
34.     while temp_ptr /= null and not done loop
35.       if temp_ptr.address = location then
36.           -- address already allocated
37.         temp_ptr.data:= data;
38.         done := true;
39.       elsif temp_ptr.address > location then
40.         new_ptr := new mem_entry;
41.         new_ptr.address := location;
42.         new_ptr.data := data;
43.         new_ptr.nxt := temp_ptr;
44.         -- break pointer chain and insert new_ptr
45.         prev_ptr.nxt := new_ptr;
46.         first.num_entries := first.num_entries + 1;
47.         done := true;
48.       else
49.         prev_ptr := temp_ptr;
50.         temp_ptr := temp_ptr.nxt;
51.         end if;
52.     end loop;
53.  -- address must be greater than address of
54.  -- last pointer in chain
55.  if not done then
56.    new_ptr := new mem_entry;
57.    new_ptr.address := location;
58.    new_ptr.data := data;
59.    add new_ptr to end of chain
60.    new_ptr.nxt := null;
61.    prev_ptr.nxt := new_ptr;
62.    first.num_entries := first.num_entries + 1;
63.    done := true;
64.  end if;
65. end if;
66. wait for 0 ns;
67. end wr_data;
68. procedure rd_data (
69.    constant location: in integer;
70.    variable data    : out std_logic_vector;
71.    variable allocated : out boolean;
72.    variable first   : inout head_ptr
73. ) is
74.  variable temp_ptr    : entry_ptr;
75.  variable is_allocated : boolean;
76. begin
77. -- set allocated to true when read hits already
78. -- allocated spot
79.  is_allocated := false;
80.   if ( first.list_ptr /= null and first.num_entries /= 0
          and location >= first.list_ptr.address ) then
81.     temp_ptr := first.list_ptr;
82.      while (temp_ptr /= null and not is_allocated and
               location >= temp_ptr.address) loop
83.        if temp_ptr.address = location then
84.         -- address has been allocated
85.         data := temp_ptr.data;
86.         is_allocated := true;
87.       else
88.         temp_ptr := temp_ptr.nxt;
89.       end if;
90.     end loop;
91.   end if;
92.  if not is_allocated then
93.   data := (data'range => 'U');
94.  end if;
95.  allocated := is_allocated;
96.  wait for 0 ns;
97.  end rd_data;
98.  end;
```

LISTING 4–DEFINING MEM_ENTRYWITH AN UNDECLARED DATA TYPE ENTRY_PTR

```
type mem_entry is record
    address         : integer;
    data            : std_logic_vector(15 downto 0);
    nxt             : entry_ptr;
end record;
type entry_ptr is access mem_entry;
```

Address	Data
0x0F00	0xAAAA
0x0011	0xBBBB
0x0F00	0xCCCC
0x0200	0xDDDD
0x1FFF	0xEEEE

The first time the routine writes into memory, the program sets the pointer such that the program allocates only one memory location. The *nxt_pointer* points to a null location. The program allocates address 0x0F00 with data 0xAAAA. The code in lines 15 through 21 illustrates this situation. The next situation involves a memory location that the program accesses such that the address of the requested memory is less than the value of the address in the first node of the linked list. In this case, the program inserts this new requested location at the top of the list. For example, the address 0x0011 is less than the address 0x0F00, so the program inserts it at the topmost node in the list. The code in lines 24 through 31 illustrates this scenario. To write to a previously allocated memory location, the program overwrites previously written memory. A write to location 0x0F00 overwrites the data 0xAAAA with data 0xCCCC without any further allocation. The code in lines 35 through 38 performs this operation. A fourth scenario is a write-to-memory location whose physical memory address is between the address in the element in the top of the list and the address in the last element in the list. For example, if the program has allocated addresses 0x0011 and 0x0F00 and receives a request for an allocation to address 0x0200, the program should insert this node between the 0x0011 and 0x0F00 elements in the list (lines 39 through 51). The last case is a write to a memory location whose physical address is greater than the address in the last element of the list. In this case, the program appends the newly allocated location to the list, and it becomes the last element of the list. If the program allocates addresses 0x0011, 0x0200, and 0x0F00 and you request a write to address 0x1FFF, the node corresponding to the address 0x01FFF is appended to the list as the last element and its Nxt_Ptr is set to null (lines 55 through 62).

The *rd_data* procedure is much simpler because it returns only the data that the program has already written in allocated locations. The *rd_data* procedure accepts the address (location) of the data that the program needs to read, a storage variable for the data (data) the program will return, a flag (allocated) to indicate whether the program allocated the requested address by a previous write, and the pointer to the head of the linked list. **Listing 3** shows the implementation of the *rd_data* procedure.

The *rd_data* procedure walks through the allocated pointers (until it reaches a null pointer). The procedure compares the address of the location with each address in the linked list. If the program finds a match, the program returns the data corresponding to that address (**Listing 3**, lines 80 through 90). A null pointer indicates the end of the list. If the program has not allocated the location,it returns an "unknown" value (**std_logic U**) (lines 92 through 94). In line 88, a copy of *first.list_ptr*, starting from the

current head_ptr, allows you to walk through to the end of the list until you hit a null pointer. If you do not copy the first.list_ptr and make the assignment in line 82 *first.list_ptr = first.list_ptr.nxt*, you are modifying the head_ptr.

Listings 2 and **3** illustrate the use of advanced VHDL constructs to allocate memory on the fly. **Listing 5** illustrates the use of such a package for an application that requires a large amount of memory. This testbench verifies and illustrates the operation of *mem_pkg* by allocating memory corresponding to random locations in physical memory and reading them back. The procedure *display_read_data* displays two parameters, *loc* and *data*, to stdout according to the following rule: If any values other than 0 or 1 in *dat* correspond to unknown values, the program prints them as binary values; otherwise, it prints them as hex values. This rule is necessary because the *hwrite* function in line 31 defines *std_logic_misc* and converts *std_logic_vector* to *bit_vector*, losing all the Multi-Valued Logic (MVL9) information. Other efficient procedures that are internal to VLSI Technology print dat without performing such checks. For the sake of completeness, **Listing 5** shows the procedure *display_read_data*. **Listing 5** also shows the test VLSI uses to verify the memory. Some procedure calls in **Listing 5** illustrate the use of the variables and constants in the calls. Lines 75 through 76 show a common way to stop the simulator after the test is complete. Compiling the previous model puts the lowest memory access at location 0 and the last memory access at location 0x7FFFFFFF (=2 Gbytes). Static alloca-

LISTING 5–MEMORY-PACKAGE DESCRIPTION FOR A LARGE-MEMORY APPLICATION

```
1.  -- mem_tb.vhd
2.  library ieee,mem_lib;
3.  use ieee.std_logic_1164.all;
4.  use ieee.std_logic_textio.all;
5.  use ieee.std_logic_unsigned.all;
6.  use std.textio.all;
7.  use mem_lib.mem_pkg.all;

8.  entity mem_tb is
9.  end mem_tb;

10. architecture mem_tb_hdl of mem_tb is
11.   procedure display_read_data(
12.     loc : integer;
13.     dat : std_logic_vector) is
14.   variable l                : line;
15.   variable U_found : boolean := false;
16.   begin
17.     write(l, string'("Read location "));
18.     write(l, loc);
19.     write(l, string'(" with "));
20.     U_found := false;
21.     for i in dat'range loop
22.       if (dat(i) /= '0' and dat(i) /= '1')  then
23.         write(l,string'("0b"));
24.         write(l,dat);
25.         U_found := true;
26.       exit;
27.      end if;
28.    end loop;  -- i
29.    if not U_found then
30.      write(l,string'("0x"));
31.      hwrite(l, dat);  -- defined in std_logic_textio
32.    end if;
33.    writeline(output, l);
34.    deallocate(l);
35.   end display_read_data;
36. begin
37.  process
38.   variable allocated  : boolean;  -- not used but needed
39.   variable rdata    : std_logic_vector(15 downto 0);
40.   variable location   : integer;
41.   variable mem_head   : head_ptr; -- beware about
    initializing an array of these
42.   begin
43. -- This has to be done to initialize the head  pointer!
44.   mem_head := new head'(0,null);
45. -- A series of writes to locations in memory
46.   location := 53;
47.   wr_data(location, To_StdLogicVector(X"AAAA"), mem_head);
48.   location := 10077845;
49.   wr_data(location, To_StdLogicVector(X"BBBB"), mem_head);
50.   location := 9876;
51.   wr_data(location, To_StdLogicVector(X"CCCC"), mem_head);
52.   location := 0;
53.   wr_data(location, To_StdLogicVector(X"0000"), mem_head);
54.   location :=CONV_INTEGER(To_StdLogicVector(X"7FFFFFFF"));
55.   wr_data(location, To_StdLogicVector(X"FFFF"), mem_head);
56. -- A series of read backs
57.   location := 53;
58.   rd_data(location, rdata, allocated, mem_head);
59.   display_read_data(location, rdata);
60.   location := 10077845;
61.   rd_data(location,rdata, allocated, mem_head);
62.   display_read_data(location, rdata);
63.   location := 9876;
64.   rd_data(9876, rdata, allocated, mem_head);
65.   display_read_data(location, rdata);
66.   rd_data(0,rdata, allocated, mem_head);
67.   display_read_data(0, rdata);
68.   location :=CONV_INTEGER(To_StdLogicVector(X"7FFFFFFF"));
69.   rd_data(location, rdata, allocated, mem_head);
70.   display_read_data(location, rdata);
71. -- read back to an unallocated location
72.   location := 1000;
73.   rd_data(location, rdata, allocated, mem_head);
74.   display_read_data(location, rdata);
75.   assert false report "End of access type memory test"
76.     severity failure;
77.   wait;
78. end process;
79. end mem_tb_hdl;
```

Circle 3 or visit www.ednmag.com/infoaccess.asp

tion needs 2 Gbytes of storage, which causes a significant slowing of simulation. The dynamic allocation in the example requires you to allocate only two locations on the fly. **References 2** and **3** show other similar methods to do this task. Of all the methods so far, the one described in this article is the simplest to understand and implement and the most efficient method to use. You can also extend these methods of creating linked data structures to models that require such data structures. For example, Universal Serial Bus (USB) and Firewire require hardware to set up such data structures. A parallel software approach for such complicated data structures is useful for verifying the hardware counterparts of these structures.

LISTING 6–USING A SIGNAL RAM_DATA TO CREATE A MEMORY ELEMENT

```
type MEM is array(DEPTH-1 downto 0) of\
Std_Logic_vector(WIDTH-1 downto 0);
signal ram_data   : MEM;
```

The previous example shows other features that improve memory modeling. You can use a binary search in place of a linear search. **Listing 5**, during both reads and writes, uses a linear-search method that starts at *head_ptr* and searches through the list until it hits a null pointer. This method can be inefficient if the program has allocated a large number of locations, and accesses are usually toward the end of the list. To speed the operation, software engineers have developed other search methods for linked lists, some of which the material in **Reference 4** details . One method is a binary-search method that searches only half the list at a time. This method uses the *num_entries* field in the *head_ptr* to divide the list into two parts. In **Listing 5**, the program does not use the allocated flag to make any decisions. You can use this flag to return a known data value or the address of the unallocated location itself, or you can sometimes allocate a new location. Deleting an allocated address is another way to improve program efficiency. Although not common in a real system test, it is not unusual in a corner-case test to delete an allocated address to verify that the test did not "memorize" the data and to verify that a read to nonexistent memories is incon-

Circle 4 or visit www.ednmag.com/infoaccess.asp

sistent. One last technique is to dump the entire memory. After you do a test, you can do a memory dump of all the allocated addresses and data. You can accomplish the dump by walking through the list and using the *display_read_data* procedure to print location and data for verification by another program or script.

SHARED VARIABLES MODEL MEMORIES

Designers usually use signals to communicate between parts of a design, but signals require more simulation time and system resources than VHDL variables require. VHDL 93 introduced shared variables to replace signals in certain situations. A few possible scenarios exist in which you can use shared variables to model efficient testbenches through an illustrative example.

You can declare variables in processes, subprograms, or both, and the scope of the variables lies within the process or subprogram in which you've declared them. More often than not, situations exist in which another process would like to monitor the variable for its value. Because you can't do this monitoring with variables, you must use signals. The overhead in using signals to communicate between processes is enormous. Signals require more physical memory on the host machine than variables. As the number of signals grows, simulator speed and simulation efficiency decrease. For example, consider a simple case of a dual-port, synchronously driven asynchronous RAM. This memory has additional requirements: You initialize the memory during reset such that all memory locations have the value loaded in data_in during the reset time. In a synchronously driven asynchronous RAM, the read port is asynchronous to the write port, although both the read and write ports are synchronous to their respective clock domains. Traditional design approaches use signals to create a memory element that is accessible from different clock domains (**Listing 6**).

This ram_data signal is DEPTH× WIDTH bits and occupies a lot of physical memory on the host system. You can reduce the physical-memory requirement by 50% or more by using the shared variables in the VHDL 1993 standard (**Reference 5**). Shared variables are a subclass of the variable class of objects. **Listing 7** shows how you can use shared variables for the application de-

LISTING 7–USING SHARED VARIABLES FOR CREATING THE MEMORY OF LISTING 6

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.  use IEEE.STD_LOGIC_UNSIGNED.ALL;

4.  entity sv_mem  is
5.    generic ( DEPTH : integer := 8;
6.              WIDTH : integer := 8);
7.    port (
8.      reset_n       : in std_logic;
9.    -- Signals Synchrnous to Write clock domain
10.     wr_clk        : in std_logic;
11.     wr_n          : in std_logic;
12.     wr_addr       : in std_logic_vector(2 downto 0);
13.     data_in       : in std_logic_vector(WIDTH-1 downto 0);
14.   -- Signals Synchrnous to Read clock domain
15.     rd_clk        : in std_logic;
16.     rd_n          : in std_logic;
17.     rd_addr       : in std_logic_vector(2 downto 0);
18.     data_out      : out std_logic_vector(WIDTH-1 downto 0)
19.  );
20.  end sv_mem ;

21.  architecture behav of sv_mem is
22.  type MEM is array(DEPTH-1 downto 0) of
     Std_Logic_vector(WIDTH-1 downto 0);
23.  shared variable ram_data    : MEM;
24.  begin  -- behav

25. ------------------------------------------------------
26. Reset Process: Initialize Memory
27. ------------------------------------------------------
28. resetP : process (reset_n)
29.  begin  -- process resetP
30.   initP : for i in 0 to DEPTH-1 loop
31.     ram_data(i) := data_in;
32.   end loop initP;
33.  end process resetP;
34. ------------------------------------------------------
35. -- Write Process: When wr_n is asserted with an address,
    write to memory
36. ------------------------------------------------------
37. writeP : process (wr_clk )
38.  begin  -- process writeP
39.   if wr_clk'event and wr_clk = '1' then
40.     if (wr_n = '0') then
41.        ram_data(conv_integer(wr_addr)) := data_in;
42.     end if;
43.   end if;
44.  end process writeP;
45. ------------------------------------------------------
46. -- Read Process : when rd_n is asserted with an address,
    read data from memory
47. ------------------------------------------------------
48. readP : process (rd_clk )
49.  begin
50.   if rd_clk'event and rd_clk = '1' then
51.     if (rd_n = '0') then
52.       data_out <= ram_data(conv_integer(rd_addr));
53.     end if;
54.   end if;
55.  end process readP;
45. end behav;
```

Circle 5 or visit www.ednmag.com/infoaccess.asp

scribed in **Listing 6**. In **Listing 7**, lines 1 through 20 declare the entity. The **reset_n signal** is the asynchronous reset to the memory. The **wr_n**, **wr_addr**, and **data_in** signals are synchronous to the **wr_clk** clock domain. The **rd_n**, **rd_adr**, and **data_out** signals are synchronous to the **rd_clk** clock domain. Line 22 defines a type to model a 2-D memory element; the rows point to each word, and the columns point to each bit in a word. Line 23 declares a shared variable of type **MEM** defined in line 22. Note that you declare the shared variable in the same declarative section that you would declare a signal. You can declare a shared variable in the declarative parts of an entity, architecture body, package, and package body of VHDL. Lines 25 through 33 define a process that executes during the assertion of **reset_n** to initialize all the rows of the memory with the value loaded in **data_in**. This process is one of two that writes to the shared variable **ram_data**. Lines 34 through 44 describe another process that writes to the shared variable upon the assertion of **wr_n**. If **wr_n** is active with **wr_clk** running during reset, you have unpredictable data that the program stores in the **ram_data**. Lines 45 through 56 describe the read process that reads data from this shared variable **ram_data** when **rd_n** is active. Again, the output **data_out** is nondeterministic on the assertion of **rd_n** and **wr_n** for the same addresses in **rd_addr** and **wr_addr**.

Use shared variables with caution. Nondeterministic outputs result when two processes try to write to the shared variable concurrently (**resetP** and **writeP**) or when the program concurrently does read and write accesses to the same location. It is best to guarantee by

LISTING 8–TOP-LEVEL CONFIGURATION OF AN SDRAM TESTBENCH

```
1.  ------------------------------------------------------
2.  Top Level Configuration
3.  ------------------------------------------------------

4.  library ieee,models_lib,test_lib;

5.  configuration dram2_cfg of tb is
6.     for tb       -- open up test bench architecture
7.        for all: sdram_mod
8.          use configuration models_lib.sdram11x8_9_cfg;
9.        end for;
10.       for all: tst_code
11.         use entity test_lib.dram2_tst(dram2_tst);
12.       end for;
13. many other configuration specifications
14. for other components in the tb architecture.
15. .
16. .
17. .
18.   end for;
19. end dram2_cfg;
```

design that no two processes will access the same shared variable during the same simulation cycle. One way to fulfill this requirement is to use a semaphore approach. **Reference 3** describes a method that implements a semaphore approach using requests, grants, and a central resource that monitors the access to the shared variable.

Introducing shared variables opens many applications for efficient modeling. Designers often use shared variables in applications in which the status of a responder may depend on the last operation performed by one of several devices plugged into a bus. You don't need a resolution function, because only one device at a time can perform an operation. Use shared variables in this case to help assign status to the responder. A real-life example of such a model is a USB hub that contains the information about the latest USB DEV attached to it. Another example is a multichannel asynchronous-transfer-mode switch. Many telecommunications models can efficiently use the behavior of shared variables.

CONFIGURATIONS AND DESIGN HIERARCHY

You can use VHDL's configuration constructs to eliminate the need for multiple unique testbenches in environments in which many people are simulating a design. You use a common configuration template at the top level of the testbench that calls out other configurations for each component in the template. These second and succeeding configuration levels determine the structure and behavior of the components in the system. Generally, most of these second-level components are the same for each test. The only second-level (or subsequent) components that differ from the standard component are those that accommodate the function you are testing. This methodology greatly eases maintaining multiple testbenches. It also facilitates random simulations, because the creation of "variant" testbenches is already built into the architecture.

LISTING 9–SECOND-TIER CONFIGURATION OF AN SDRAM TESTBENCH

```
20. library ieee;  use ieee.std_logic_1164.all;
21. library models_lib;use models_lib.mem_comps_pkg.all;

22. entity sdram_mod is
23.  port (
24.   sdram ports . . .
25. )
26. end sdram_mod;

27. architecture sdram11x8 of sdram_mod is
28. begin
29.  d : sdram
30.   generic map (
31.     row_adr_size => 11,
32.     col_adr_size => 8
33.     other related generics)
34.   port map (
35.     sdram ports
36.  );
37. end sdram11x8;

38. Other Architectures…

39. library models_lib;
40. configuration sdram11x8_9_cfg of sdram_mod is
41.   for sdram11x8
42.     for all: sdram
43.       use entity models_lib.sdram(sdram_9_hdl);
44.     end for;
45.   end for;
46. end sdram11x8_9_cfg;

47. library models_lib;
48. configuration sdram11x8_10_cfg of sdram_mod is
49.   for sdram11x8
50.     for all: sdram
51.       use entity models_lib.sdram(sdram_10_hdl);
52.     end for;
53.   end for;

54. end sdram11x8_10_cfg;

55. Other Configurations
```

To illustrate this testbench methodology, consider a scenario in which you need to test a synchronous-DRAM (SDRAM) controller. In this SDRAM, we tested a large system controller of which the SDRAM controller was a small part. During simulation tests, you must use many types of SDRAMs, varying in both size and performance. VHDL provides many ways to organize a hierarchy to support these tests. Because several people may be simulating the design, you must provide one basic configuration as an initial starting point, with each person generating his own individual test variations. Some problems, however, exist with team VHDL-design projects.

One problem is that unnecessary code duplication may occur when several people generate the same variations in architectures. Each person may not know about other people's work, because each may locally maintain the code or place it in files that they predominantly control. Another problem arises when you need to pass large numbers of generics to the model. Identical generics may exist in many of the architectures at a single level, increasing the chance for modification errors. A third problem occurs when a large number of port and generic maps clutter the reference files. **Figure 4** shows a hierarchy you can use to minimize these problems.

The hierarchy consists of a top-level configuration, second-tier configurations with an entity and architectures, a third-tier entity with architectures and a fourth-tier basic functional entity. **Listing 8** represents the top-level configuration in the testbench. Line 6 in **Listing 8** contains the architecture, which the example does not show. The architecture contains a chip containing the SDRAM controller, an SDRAM module contain-

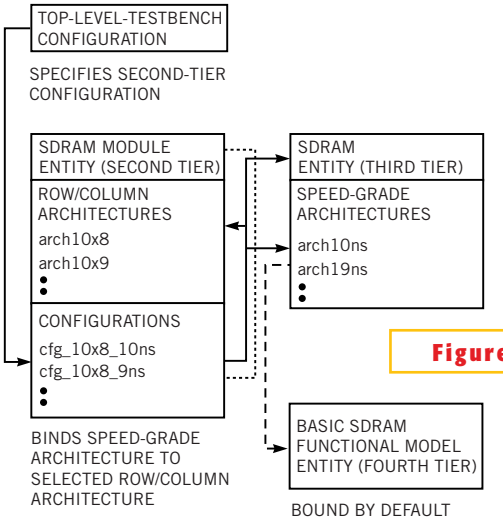

Figure 4

Using a VHDL design hierarchy, such as the one shown here, you can minimize the problems you might encounter when multiple designers work on design testbenches.

LISTING 10—THIRD-TIER ARCHITECTURE OF AN SDRAM TESTBENCH

```
56. library ieee,models_lib;
57. use ieee.std_logic_1164.all;
58. use models_lib.sdram_fm_comps_pkg.all;

59. entity sdram is
60.   generic (
61.     row_adr_size : integer := 11;
62.     col_adr_size : integer := 8;
63.     trc          : time := 120 ns;
64.     tras         : time := 70 ns
65.     many! other generics
66.    );
67.   port       (
68.    sdram ports
69.   );
70. end;

71. -- This architecture specifies a 9 ns sdram.
72. architecture sdram_9_hdl of sdram is
73.  begin
74.   sd9 : sdram_fm
75.    generic map (
76.     trc          => 90 ns,
77.     tras         => 54 ns,
78.     row_adr_size => row_adr_size,
79.     col_adr_size => col_adr_size
80.     Many! other generics
81.    )
82.    port map (
83.     sdram ports
84.    );
85. end sdram_9_hdl;


86. -- This architecture specifies a 10 ns sdram.
87. architecture sdram_10_hdl of sdram is
88.  begin
89.   sd10 : sdram_fm
90.     generic map (
91.      trc          => 100 ns,
92.      tras         => 60 ns,
93.      row_adr_size => row_adr_size,
94.      col_adr_size => col_adr_size
95.      Many! other generics
96.     )
97.     port map (
98.      sdram ports
99.     );
100. end sdram_10_hdl;
```

ing the SDRAM, a test stimulus, and other components you need for testing. Line 7 is the second-tier configuration specification that references an SDRAM configuration containing the necessary binding information to completely specify the SDRAM characteristics. You can use this top-level configuration as a template for other tests requiring different components simply by modifying the configuration specifications and other parameters. This modification capability lets you copy and modify the template for each new test. Using a template, all the people simulating the design have common code styles and a common way of customizing the testbench to suit their needs. Because this configuration conveys the characteristics of the overall testbench, reference as few generics as practical to keep the file clean and easy to read. **Listing 9** shows the second-tier SDRAM configuration from **Figure 4** along with the accompanying entity and architecture that it configures. This configuration shows which entity or architecture pair you should use for the *sdram_mod* instances in Line 7 of **Listing 8**. Line 23 also binds the component called SDRAM in Line 29 to its own entity or architecture pair. The component name need not match the entity name to which it is bound. Many more architectures may exist for the entity *sdram_mod*, along with other configurations; each may specify different sizes, arrangements, and SDRAM characteristics. Lines 31 and 32 show one such specification. Keeping all of the architectures and configurations in one file or location helps reduce unnecessary duplication; you don't need local or isolated code blocks.

The basic SDRAM functional model in the fourth tier requires externally supplied speed information in addition to size and function information. You also pass this information to the model as generics. Because so many timing-related generics exist for a given speed, passing all of them through each instance of the SDRAM component (line 29) makes this file unreasonably large and difficult to use. Another level of hierarchy accommodates this problem. You don't assign any speed generics at the second tier of the hierarchy, keeping the architectures in **Listing 9** relatively small and specialized to indicate SDRAM size and function (line 31).

The third-tier architectures contain components with timing information assigned to the generics. The generic values for size and function pass from the second-tier architectural components. You develop the third-tier architectures (**Listing 10**, lines 76 through 77 and 91 through 92) for the timing and speed information. Then, the second-tier configurations bind the third-tier timing entity and architectures to the components in the second-tier size and function architectures. The binding of the fourth-tier entity (and architecture) to the third-tier components occurs by default. **Listing 11** shows the fourth-tier entity. By using the flexible-binding capabilities of VHDL configurations and specializing the architectures this way, you can easily organize the code, making it easy to read, understand, modify, and enhance. When you configure all of the top-level components this way, it is easy to add variations to the testbench by introducing, for example, a new architecture at one level, along with an accompanying second-tier configuration that you can then reference from the top-level configuration. Note that if you can modify the basic functional models, you may not need additional hierarchies. Instead, you can design the basic model to automatically set up many of the otherwise generic parameters, hard-coding them into the model with selections that you can do with a single generic.

FOREIGN-LANGUAGE INTERFACE

VHDL has powerful language constructs that you can use to model the behavior or structure of any hardware device. In the past, when VHDL and other hardware-modeling languages were unpopular, you wrote models in general-purpose-programming languages, such as C. In addition, you have advanced C libraries available to perform complex

LISTING 11—FOURTH-TIER ENTITY FOR AN SDRAM TESTBENCH

```
101. library ieee;
102. use ieee.std_logic_1164.all;

103. entity sdram_fm is
104.    generic (
105.       row_adr_size : integer := 11;
106.       col_adr_size : integer := 8;
107.       trc          : time := 120 ns;
108.       tras         : time := 70 ns
109.       );
110.    port   (
111.       -- sdram ports
112.       );
113. end sdram_fm;

114. architecture sdram_fm_hdl of sdram_fm is
115. begin
116. .
117. .
118. .
119. end sdram_fm_hdl;
```

computations on the fly, which is difficult to do in VHDL. In such instances, VHDL provides a way to use foreign elements to help modeling with its foreign-language interface. To use the foreign-language interface, you must adopt the following procedures. This section shows the use of the foreign-language interface through an example that is specific to the ModelSim simulator from Model Technology (www.model.com). First, create the C model, including simulator-specific calls. Next, compile the model and create a shared object. (Note that this shared object may not be portable from system to system, for example, from a machine running Solaris to one running HPUX.) Finally, create an entity and architecture with the foreign attribute to bind the model. You can substantiate this entity in any other design, as required.

LISTING 12–VHDL D FLIP-FLOP MODEL WITH A C INTERFACE

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;

3.  entity dflop is
4.   port (
5.     reset : in std_logic;
6.     d : in std_logic;
7.     clk : in std_logic;
8.     q : out std_logic
9.    );
10. end dflop;

11. architecture arch of dflop is
12. attribute foreign : string;
13. attribute foreign of arch : architecture is
    "reg_init dflop.so";
14.  begin
15.  end;
```

Our foreign-language interface uses a simple C model of a D flip-flop, the VHDL interface shown in **Listing 12**, and the C-code in **Listing 13**. The D-type flip-flop model is positive-edge-triggered and asynchronously reset. It is useful to examine the C code before discussing the architecture of the entity *dflop*. This C model mimics the behavior of a D-type flip-flop. The function calls that begin with *mti_* are ModelSim-specific. At a minimum, the C model needs the initialization function and a function modeling the required behavior. A little knowledge of C is useful for understanding the code in **Listing 13**. You can find detailed explanations on ModelSim-specific function calls in the ModelSim user manual. You compile this C file using any ANSI-standard-compliant C-compiler link to create a shared-object file. For the D flip-flop example, compile the C model for a Solaris platform using the following commands: cc -c -DI$MODELTECH/include dflop.c and ld -G -o dflop.so dflop.o The architecture *arch* of entity *dflop* binds the compiled C code via the VHDL foreign attribute. The VHDL statement: attribute foreign of arch : architecture is "reg_init dflop.so"; indicates that you need a foreign architecture in the file *dflop.so*, which has an initialization function *reg_init*, for *arch*. Note that the initialization function is key to all C models. This function typically allocates memory to hold variables for the instance, registers a call-back function to free the memory upon simulator start-up, saves the handles to the signals in the port list, creates drivers on ports that it will drive, creates one or more processes (a C function that you can call when a signal changes), and sensitizes each process to a list of signals (**Reference 6**). You can pass generics to C models. Some C models are simulator-specific; you should refer to the simulator's user manual for further information.

RECORDS IN PROCEDURES

VHDL procedures are constructs that VHDL provides to hide unnecessary complexity and detail from the main body of code. In doing so, these procedures often become extremely complex or have to handle large numbers of signals. Passing information to the procedure through individual elements quickly becomes impractical if many signals are involved. As an example, consider a procedure call to initiate a PCI master to begin a transaction. The procedure body takes input information and toggles outputs in a specific sequence. The procedure then passes these outputs outside itself. When the PCI-master functional model, located elsewhere in the design, sees the outputs, the model initiates the PCI transaction.

Listing 14 defines a package with the signals that you typically need to communicate with and control a PCI master. In this simple case, the program individually defines each signal. **Listing 15** shows a process for initiating a PCI memory write followed by a PCI memory read. The procedures take the *cyc*, *addr*, and *tc* information and then drive or examine the remaining PCI-master-specific sig-

LISTING 13–C MODEL OF A D FLIP-FLOP

```
#include <stdio.h>
#include "mti.h"

/* positions for the enum std_ulogic */
/* assumes values on the ports use std_logic encoding! */
#define STD_U 0
#define STD_X 1
#define STD_0 2
#define STD_1 3
#define STD_Z 4
#define STD_W 5
#define STD_L 6
#define STD_H 7
#define STD_D 8

typedef struct {
    signalID reset;
    signalID d;
    signalID clk;
    driverID out1;
    long old_clk;
} inst_rec;

/* module initialization function */
reg_init(region, param, generics, ports)
regionID region;
char *param;
interface_list *generics;
interface_list *ports;
{
    inst_rec *ip;
    signalID outp;
    processID proc;
    extern free();

    /* allocate space for user data */
    ip = (inst_rec *)malloc(sizeof(inst_rec));

    /* "restart" will call "free(ip)" if used */
    mti_AddRestartCB(free, ip);

    /* get the input ports */
    ip->reset = mti_FindPort(ports, "reset");
    ip->d = mti_FindPort(ports, "d");
    ip->clk = mti_FindPort(ports, "clk");

    /* get the output port and create a driver on it */
    outp = mti_FindPort(ports, "q");
    ip->out1 = mti_CreateDriver(outp);

    /* make process and sensitize the signals we care about */
    proc = mti_CreateProcess("FFPR", eval_reg, ip);
    mti_Sensitize(proc, ip->reset, MTI_EVENT);
    mti_Sensitize(proc, ip->clk, MTI_EVENT);
}

/* function to evaluate a simple dff */
void eval_reg(ip)
inst_rec *ip;
{
    long new_reset;
    long new_clk;
    long new_val;
    int do_sched = 0;

    /* get new reset & clock values */
    new_reset = mti_GetSignalValue(ip->reset);
    new_clk = mti_GetSignalValue(ip->clk);

    /* if reset is set, schedule out a '0' value */
    if (new_reset == STD_1) {
        new_val = STD_0;
        do_sched = 1;
    }
    /* if clock changed, and is a rising edge use "d" value */
    else if (new_clk != ip->old_clk && new_clk == STD_1) {
        new_val = mti_GetSignalValue(ip->d);
        do_sched = 1;
    }
    /* update old clock for next evaluation */
    ip->old_clk = new_clk;
    /* schedule (if needed) the new value */
    if (do_sched)
        mti_ScheduleDriver(ip->out1, new_val, 0,
MTI_INERTIAL);
}
```

Define MVL9

Define Inputs, Outputs, Signals

Setup function

Flip Flop Model

LISTING 14—VHDL PACKAGE WITH THE SIGNALS TO CONTROL A PCI-MASTER MODEL

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;

3.  package pci_pkg is
4.  signal cyc     : std_logic_vector(4 downto 0)
                          := (others => 'Z');
5.  signal addr    : std_logic_vector(31 downto 0)
                          := (others => 'Z');
6.  signal tc      : std_logic_vector(15 downto 0)
                          := (others => 'Z');
7.  signal byten   : std_logic_vector(3 downto 0)
                          := (others => 'Z');
8.  signal cdelay  : std_logic_vector(15 downto 0)
                          := (others => 'Z');
9.  signal data    : std_logic_vector(31 downto 0)
                          := (others => 'Z');
10. signal delay   : std_logic_vector(15 downto 0)
                          := (others => 'Z');
11. signal step    : std_logic_vector(15 downto 0)
                          := (others => 'Z');
12. signal same    : std_logic := 'Z';
13. signal lock    : std_logic := 'Z';
14. signal back    : std_logic := 'Z';
15. signal mode    : std_logic_vector(1 downto 0)
                         := (others => 'Z');
16. signal cmd_mode: std_logic_vector(15 downto 0)
                          := (others => 'Z');
17. signal start   : std_logic := 'Z';
18. signal rdata   : std_logic_vector(31 downto 0)
                          := (others => 'Z');

19. end;
```

nals so that the PCI master in another part of the design can begin the transaction on the PCI bus. For the most part, the process does not need to directly reference any of the signals going to the PCI master; they are all driven or examined solely in the procedure the program calls. With an increasing number of PCI reads and writes, the code becomes more difficult to read. This situation worsens as you expand as the functionality of the read and write procedures. The easiest way to avoid this problem is to define a record with elements corresponding to each of the required PCI-master signals and then create a signal of that record type. **Listing 16** is a package that defines such a signal. The record statements in lines 4 through 20 define a template; line 21 shows the actual signal. You need not specify the subelements of the signal when passing the whole signal to the package.

Once you declare the PCI read and write procedures, you can write them (**Listing 17**). You must pass four arguments during the procedure call (lines 20 and 21). You sometimes can't modify a model (in this case, the PCI model); in such a case, you should put the signals coming from the model intoa record. In this situation, map the individual signals to the record using concurrent signal assignments at a place at which both the composite and scalar signals are visible (in the appropriate direction). Be careful with records having components driving data both in and out. The signal of the record type must be a resolved signal. Drive the record elements that you shouldn't drive in the calling process to an appropriate value commensurate with the element's type, so you do not affect the signal value (Z in std_logic, for example). By using records in this way, you can use more highly complicated procedures without being burdened by lots of unnecessary code. □

References

1. Ashenen, PJ, *The Designers Guide to VHDL*, Morgan Kaufman Publishers

continued on pg 86

LISTING 15—VHDL PROCESS FOR INITIATING A PCI MEMORY WRITE FOLLOWED BY A MEMORY READ

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.  library models_lib;
4.  use models_lib.pci_pkg.all;
5.  library test_lib;
6.  use test_lib.test_pkg.all;

7.  entity wndw_tst is
8.  end wndw_tst;

9.  architecture
10. .
11. .
12. .
13. begin
14. process
15. variable dat : std_logic_vector(31 downto 0) :=
16.                htov("0000FFF1",32);
17. variable pci_adr : std_logic_vector(31 downto 0) :=
18.                htov("CCAA0000",32);
19. variable wid : std_logic_vector(31 downto 0) :=
20.                (others => 'X');
21. begin
22. .
23. .
24. .
25. pci_mwr(pci_adr, dat, 1, cyc, addr, tc,
26.         byten, back, cdelay, data, delay,
27.         step, same, lock, mode, cmd_mode,
28.         start, rdata);
29. pci_mrd(pci_adr, wid, 1, cyc, addr, tc,
30.         byten, back, cdelay, data, delay,
31.         step, same, lock, mode, cmd_mode,
32.         start, rdata);
33. .
34. .
35. .
36. end process;
37. end;
```

LISTING 16—VHDL PACKAGE PCI-MASTER SIGNALS

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;

3.  package pci_pkg is

4.  type mpci_rec_t is record
5.     cyc      : std_logic_vector(4 downto 0);
6.     addr     : std_logic_vector(31 downto 0);
7.     tc       : std_logic_vector(15 downto 0);
8.     byten    : std_logic_vector(3 downto 0);
9.     cdelay   : std_logic_vector(15 downto 0);
10.    data     : std_logic_vector(31 downto 0);
11.    delay    : std_logic_vector(15 downto 0);
12.    step     : std_logic_vector(15 downto 0);
13.    same     : std_logic;
14.    lock     : std_logic;
15.    back     : std_logic;
16.    mode     : std_logic_vector(1 downto 0);
17.    cmd_mode : std_logic_vector(15 downto 0);
18.    start    : std_logic;
19.    rdata    : std_logic_vector(31 downto 0);
20. end record;

21. signal mpci_rec : mpci_rec_t :=
22.    ((OTHERS => 'Z'),
23.    (OTHERS => 'Z'),    -- addr
24.    (OTHERS => 'Z'),    -- tc
25.    (OTHERS => 'Z'),    -- byten
26.    (others => 'Z'),    -- cdelay
27.    (others => 'Z'),    -- data
28.    (others => 'Z'),    -- delay
29.    (others => 'Z'),    -- step
30.    'Z', 'Z', 'Z',      -- same.lock,back
31.    (OTHERS => 'Z'),    -- mode
32.    (others => 'Z'),    -- cmd_mode
33.    'Z',                -- start
34.    (others => 'Z'));   -- rdata

35. end;
```

Inc, San Francisco, CA, 1996.

2. Bilik, S, "Modeling Sparsely Utilized Memories in VHDL," VIUF Fall Conference, 1996.

3. Cohen, B, *VHDL Answers to Frequently Asked Questions*, Second Edition, Kluwer Academic Publishers, Boston, MA, 1997.

4. Headington, Mark, and David Riley, *Data Abstraction and Structures Using C++*, DC Heath and Co, 1994.

5. *IEEE Standard VHDL Language Reference Manual 1076-1993*, IEEE Press, New York, NY, 1994.

6. *ModelSim EE/PLUS Reference Manual, Version 5.1*, Model Technology Inc, Beaverton, OR, 1998.

LISTING 17–PCI READ AND WRITE PROCEDURES

```
1.  library ieee;         use ieee.std_logic_1164.all;
2.  library models_lib;   use models_lib.pci_pkg.all;
3.  library test_lib;     use test_lib.test_pkg.all;

4.  entity wndw_tst is
5.  end wndw_tst;

6.  architecture
7.  .
8.  .
9.  .
10. begin
11. process
12.     variable dat : std_logic_vector(31 downto 0)
13.                  := htov("0000FFF1",32);
14.     variable pci_adr : std_logic_vector(31 downto 0)
15.                  := htov("CCAA0000",32);
16.     variable wid : std_logic_vector(31 downto 0)
17.                  := (others => 'X');
18. begin

19.     -- pci_mwr(address, data, count, pci_record);
20.     pci_mwr(pci_adr,dat,1,mpci_rec);
21.     pci_mrd(pci_adr,wid,1,mpci_rec);

22. end process;

23. .
24. .
25. .
26. end;
```

AUTHORS' BIOGRAPHIES

Subbu Meiyappan is a senior design engineer at VLSI Technology, a subsidiary of Philips Semiconductors. He has worked for the company for nearly three years, designing, developing, synthesizing, simulating, and validating high-performance intellectual-property blocks for PCI, ARM-ASB-based devices, and high-performance ASICs. He has a BE from Annamalai University (Annamalai Nagar, India) and an MS from Tennessee Technological University (Cookeville, TN). His interests include computer architecture, design automation, volleyball, and travel. You can reach him at Subbu.Meiyappan@vlsi.com.

James Steele is a staff design engineer at VLSI Technology, where he as worked for 12 years. He designs ICs and has developed wireless applications and PC/notebook chip sets. He has a BSEE from Arizona State University (Tempe, AZ). You can reach him at James.Steele@vlsi.com.

Circle 7 or visit www.ednmag.com/infoaccess.asp