

Designing Procedural-Based Behavioral Bus Functional Models for High Performance Verification

Gregg D. Lahti
gregg.d.lahti@intel.com
Tim L. Wilson
tim.l.wilson@intel.com

Intel Corporation

ABSTRACT

Testing a design is half of the design cycle and one of the most important phases of the pre-silicon design phase to ensure that the product is bug free. Our method of setting up a test bench to drive stimulus incorporates the use of Bus Functional Models (BFMs) to drive stimulus and verify functional results from a Device Under Test (DUT). BFMs allow skeleton models of a device to be used; they do not necessarily carry the full functionality of a component but do imitate the "bus" or I/O signal characteristics.

Existing BFMs, such as the Synopsys PCI Source Models, use hard-coded test code or file-resident test code, which requires overhead for command parsing. Hard-coded or file-driven methods for testing have many shortcomings and limitations. Instead, a pure VHDL procedural-driven methodology eliminates these limitations and problems by embedding the test code, written in VHDL, into the test bench and utilizing the full power of the VHDL event-driven simulator.

Using this test and simulation methodology, we were able to model and simulate a complete chipset using a processor BFM, RAM BFMs, and modified Synopsys PCI BFMs all using VHDL procedural-based testing.

1.0 Introduction

Testing designs to ensure bug free operation is a complex and laborious task. It is estimated that half of a project development involves testing the design, which significantly relates to the required man-effort and time-to-market of the design. As the design complexity and transistor count increase, faster and more efficient methods to test the design are mandatory to ensure a bug free design. Utilizing Bus Functional Models enables less simulation overhead and can enable more complex functional test operations to be written.

The VHDL source code is included in Appendix 8.0 and can be found via the web on the SNUG website or at <http://www.primenet.com/~greggl/snug99.html>.

2.0 Test Bench Basics

The basics of a test bench can be broken down into three elements: a DUT, stimulus to the DUT, and response checking from the DUT as shown in figure 1.0.

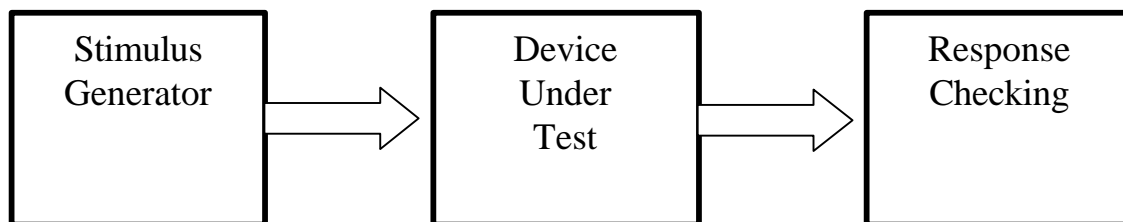


Figure 1.0 Basic Test Bench Block Diagram

2.1 The Device Under Test

The DUT gets stimulus and the response checking module checks the output of the DUT for correct operation. The stimulus driving the DUT can be generated in many different formats and from different sources, but it is primarily focused towards exercising the DUT to output a known response.

2.2 Stimulus

How stimulus gets applied to the DUT matters in terms of simulation performance, reusability, and debugging ability. Some methods can include:

- File-based vectors applied to the DUT ports in test-vector formats
- Use of Bus Functional Language (BFL) code to drive simulation models
- Basic assert or force statements in VHDL or simulator-specific commands

Methods of stimulus can directly affect simulation performance. As an example, heavy usage of file I/O can drastically slow simulator performance in comparison to memory-resident simulator code execution.

2.3 Response Checking

Somehow, the response of the DUT needs to be checked for validity. One method is to capture the pin events of the DUT into a file during simulation. Once the simulation has been verified that it is good, the trace file is saved as a golden trace file and compared to future simulation runs after the DUT has been modified. This method does work, but it requires human interaction to verify that the golden simulation is correct. This method also requires that a trace file be kept which may have many drawbacks and side-effects: trace file revision control, the size of the trace file may be very large and occupy huge amounts of disk space, and determining the difference between a new simulation trace file and the golden trace file can be a major problem if the response of the new simulation “slips” clock cycles and time events do not line up correctly. This method also doesn’t easily test potential timing or protocol violations of the DUT.

A better method to check response is to have the test bench automatically check the DUT response during the test. A self-checking test bench should understand the protocol or timing of the DUT, check for violations that may arise, and be easily written such that all response is accounted for in some manner. Self-checking test benches require human interaction during the development and testing phase, but almost no interaction to verify future simulations after initial verification. Items such as cycle slips (i.e. bus operations that take one or two clocks longer to complete but still are legal within the protocol) can be accounted for and be allowed in the response-checking algorithm.

Using a self-checking test bench can aid in testing from the RTL to gate-level tests by utilizing the same timing and protocol checks for either simulation. This method eliminates separate test benches for gate-level simulation and keeps the same environment for all tests, cutting down on the test bench development time and effort.

3.0 Modeling Devices as Bus Functional Models

Stimulus and event-checking modules can be modeled in various levels of abstraction, from very basic stimulus events to complex, high-level models. As an example, a PC chipset design may use a processor model as a stimulus module in system-level testing. The processor model could include the full representation of a processor down to the RTL or gate level. However, most companies other than Intel won’t have a full processor model of a due to cost, design availability, or intellectual property concerns. Testing a system utilizing a four million gate model isn’t performance effective either.

An alternative to using an exact processor model is to model the bus timing and protocol without implementing the overhead of the internal states. Using a Bus Functional Model (BFM) enables the engineer to assert stimulus just like a processor but without the extra overhead of the internal

workings of a CPU which isn't relevant to the rest of the system. For example in testing a Pentium II™ chipset, using the internal L1 cache or ALU of a processor doesn't directly cause processor or chipset bus operations. However, operations like cache fills, code fetches, or memory writes would cause cycles between the processor bus and the chipset. Using a BFM allows the engineer to execute stimulus that directly affects the DUT without the overhead of a full model. A BFM can easily model external bus cycles to occur without setting up internal states, such as the steps required to cause a cache line fill or write.

3.1 Basic BFM Example

A very basic test bench using a BFM could look like the example source code in Appendix 8.1. In this example, the basic VHDL test bench can utilize a BFM with pre-determined stimulus built into the BFM. The amount of overhead for this BFM is minimal.

This method is useful, but not practical because the "instructions" which drive the BFM are hard-coded into the model which limits the flexibility for coding and development and the coding level of these instructions is very low. The engineer has to manually edit the test code to change the stimulus timing. Also, if the engineer wishes to segment the test code into functional tests or if multiple engineers need to develop code, this method quickly runs out of steam.

3.2 Driving the BFM From External Files

Another method to driving the BFM is the use of a file that contains Bus Functional Language (BFL) code. This code is organized as a model-specific format that the BFM reads and does operations from, much like low-level assembly code that uses an operand type and data to accomplish an instruction. The code generally resides in a text file and is read into the BFM during simulation time all at once or at a line per instruction depending upon how the model is written.

3.2.1 BFL Code

One of the best advantages of using BFL code is that commands are created and utilized for specific BFM operation and the format of the commands can be anything the engineer desires as long as the VHDL constructs can parse the text file. As an example, the PCI Source Models from Synopsys utilize BFL code to drive the BFM stimulus. The BFL code is structured into specific bus commands with user-parameters for function control. For example, in figure 2.0 has the BFL command for a PCI memory read:

```
print_msg("PM: memory read to PT to verify initialization ");
read_cycle(mem_read, "00000000", 1,"0","86551643", 0, false);
read_cycle(mem_read, "00000004", 1,"0","539700f7", 0, false);
read_cycle(mem_read, "00000008", 1,"0","00017300", 0, false);
```

Figure 2.0 BFL Code Example

The PCI BFM reads in the text file containing instructions, and parses out the user-defined cycle type, address, data, cycle length fields, and other user definitions to drive the PCI bus.

However, BFL code has some serious disadvantages that should be considered:

- File I/O is slow
- Requires separate language parser to do complex operations (i.e. loops, math ops)
- Complex operations not possible (i.e. read, modify, write)
- Synchronization of multiple BFM's is complex and cumbersome
- Sampling and driving signals in the test bench is difficult

One of the main drawbacks of using BFL code is that the BFL code must be “knowledgeable” of data operations: i.e. one cannot do a memory read, modify the read data in some form, and write the data back to a memory location. Since BFL code is parsed, all commands must have predicted outcome or the BFL command structure must understand the use of memory elements or variables to do simple tasks like loops which makes BFL code and the code parsing section of the BFM very complex.

BFL code communication across multiple BFM's is another problem. For example, three PCI master BFM's were placed in a test bench. Each BFM would require a separate BFL test code file to drive them. How does each model communicate or synchronize command operations with each other? In the Synopsys PCI Source Model version, a special backplane bus of signals connects all the BFM's together and special BFL instructions allow the use of these signals to communicate between BFM's. This method is limited, since it is based upon specific operations or boolean conditions of the pre-determined backplane signals and doesn't allow much flexibility in the BFL test code.

Another drawback is that BFL code must be read into the simulator via file I/O, which is generally slower and has tendencies to slow simulation if the file needs to be read in a line-at-a-time for execution. Utilizing network file systems with BFL code can further reduce simulation performance. If file I/O is a must, the preference would be to have BFL code read in its completion at the beginning of the simulation to reduce the performance impact.

Another method for using BFL language is compile the BFL code into object code using a separate parser, compiler, or interpreter that the VHDL BFM can understand. This involves using a separate application written specifically for this task. This method is useful if the language is very complex, but it does involve more overhead: the compiler must be custom written, intermediate output files must be kept around causing more disk and control overhead, and the parser/compiler must have intelligence to deal with complex algorithmic operations.

The drawbacks to BFL code are serious enough from a high-level, coding point of view to warrant a different look at how simulation code could be written utilizing VHDL-based commands.

4.0 BFM High-Level Testing Methodology

4.1 Testing with BFMs

Using a pure VHDL procedural-driven methodology eliminates BFL code hazards by embedding the test code, written in VHDL, into the test bench and utilizing the full power of the VHDL event-driven simulator. This method of testing has the following benefits:

- Power of VHDL language for test code creation
- No external parsers required
- Compiled in native language, faster runtime
- Easier method of synchronization of multiple BFMs in test bench
- Faster simulations due to less overhead
- Reuse of test bench, test code
- Eliminates need for expensive hardware acceleration

The main components of testing with BFMs in VHDL can be broken out into four components:

- Use of BFMs within the test bench
- Use of globally accessible packages containing BFM procedure calls
- Using separate test code entity/architectures that drive and automatically check stimulus
- Use of configurations to assemble different abstractions of the test bench components

The following sections will deal with the example code in Appendix 8.2. The representation of how the items fit together can be expressed in the figure below.

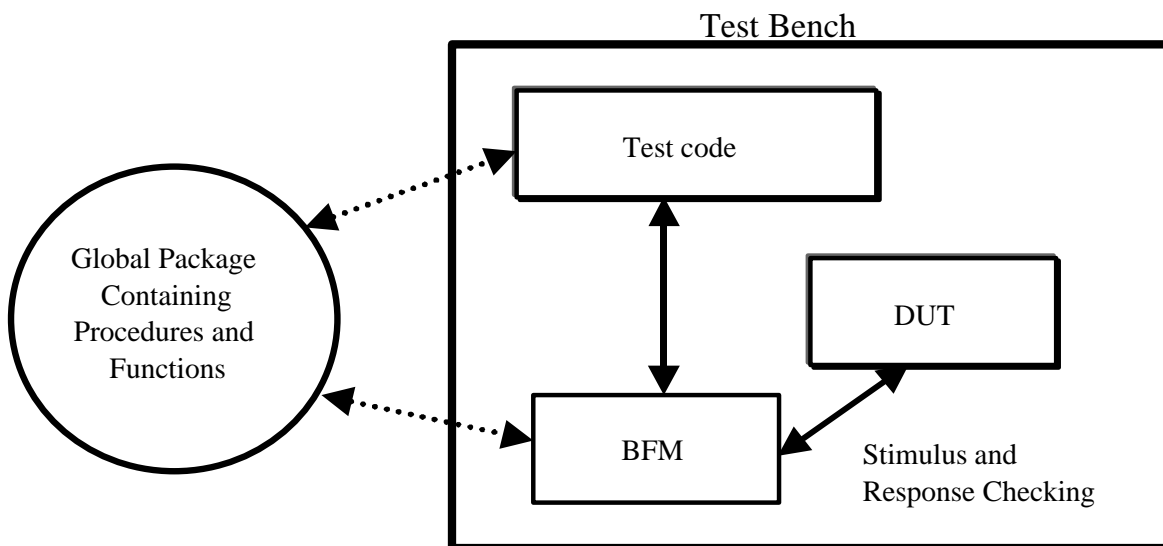


Figure 3.0 Test Bench Components

4.2 Global Packages

Modeling in VHDL has advantages due to the strongly typed language format, but VHDL also has very strict scope rules as compared to Verilog. The use of packages, types, and records give the engineer flexibility to create BFM's at a higher level of abstraction similar to C code that a programmer would write. Using high level abstraction enables thorough architecture testing by giving the engineer freedom to code in complex operations from a programmer or system point of view and yet still retain the low-level primitive operations of bus cycles, event comparisons, and timing checks from the BFM to the DUT.

One method of creating high-level test constructs is to use functions and procedures in VHDL to represent the bus cycle operations. These function and procedure definitions reside in a package, allowing global accessibility by the BFM and test code with VHDL use clauses. The procedures are styled from a programmer view of bus cycles, such as memory or I/O cycles. For example, Intel386™ microprocessor bus operations can be reduced to eleven types of bus cycles: memory read, memory write, I/O read, I/O write, two sizes of each afore-mentioned cycles (8 or 16-bit operations), no-op, interrupt, and halt/shutdown. The test code procedures can be broken down to a procedure for each function and executed by a process in a separate entity/architecture.

A connection between the procedures and the BFM stimulus state machine exists through the use of signals and record types to allow the BFM to get “commands” and provide feedback to the originating test code segment, as shown in figure 4.0.

```
type cycle_t is (idle, write, read);

type instruction is record
  bus_cycle   : cycle_t;           -- cycle type
  addr       : std_logic_vector(31 downto 0); -- address
  data       : std_logic_vector(31 downto 0); -- write/read value
  rdata      : std_logic_vector(31 downto 0); -- actual read data
  request    : std_logic;         -- request handshake
  busy       : std_logic;         -- busy handshake
end record;

-- declare signal, initialize to values. The rdata and busy signals
-- are set to Z's since the BFM will assign drivers to them
signal cmd   : instruction := (idle,
                              (others => '0'),
                              (others => '0'),
                              (others => 'Z'),
                              '0',
                              'Z');
```

Figure 4.0 Type and Record Definitions

In this example, the record contains information fields that the BFM uses to drive stimulus and pass back to the process that contains the executed procedure:

- The cycle type
- Address of the cycle
- Write data if it's a write operation or expected read data if the cycle is a read operation
- Actual read data the BFM received
- Handshaking strobe signals used by the main procedure and the BFM for communication.

Note that when using VHDL procedure calls, the record signal must be included within the procedure call itself since each procedure is reading and modifying this record. The handshaking signals used must also be visible in the package definition for the BFM and the main procedure call visibility scope.

```
swr("00000000", "deadbeef", cmd); -- write cycle
idle(cmd); -- idle
srd("00000000", "deadbeef", cmd); -- read cycle
```

Figure 5.0 Example BFM Procedure Call

In our 8.2.2 `ibc_bfm_pkg.vhd` example, each major bus operation procedure executes a main procedure called `ibc_cyc()`. This procedure deals with BFM handshaking and “queuing” of BFM commands by the use of the `request` and `busy` handshaking signals and acts as the main communication channel between the test code and the BFM.

4.2.1 Utilizing BFMs in the Test Bench

An entity/architecture of our BFM is instantiated within our test bench. The architecture of the BFM contains bus control, timing information, and a state machine that executes the bus operations. The state machine uses the handshaking signals to determine start and asserts busy during operation. The state machine also provides the actual read data values back into the record.

4.2.2 Test Code Basics

The test bench needs some connection to the test code. In the test bench example `ibc_tb.vhd`, in Appendix 8.2.7, a single line which instantiates the test code can be found on line 45. This empty-port instantiation allows the test code entity/architecture to be connected into the test bench. There are no ports on the test code, as all test bench signals and procedures are referenced via global packages that contain the signals and procedures assignments.

The test code still needs to be defined. In the `ibc_tst1.vhd` example in Appendix 8.2.8, a process called `test` containing the procedure calls is placed within the test code architecture. This process is effectively the stimulus driver; it utilizes the BFM procedure calls from the global package to drive the BFM stimulus. Placing the test code in a process allows the use of the

VHDL language to create loops using user-defined signals, variables, and types as the engineer sees fit. Complex algorithms can be created with the BFM procedure calls, such as writing 32-bit words to every eight-byte address memory elements in a given address range:

```
tv := to_stdlogicvector(X"a5a5e3e3");
for i in 16#00000100# to 16#00000140# loop
  if (i mod 8 = 0) then      -- every other dword
    wr(conv_std_logic_vector(i,32),tv,cmd); -- write value to DUT
    tv := tv xor tv0;      -- just for kicks, xor the next data
  end if;
end loop;
```

Figure 6.0 Complex Loop

Or a classic read-modify-write cycle as shown in figure 7.0 that reads a value from memory, modifies the read data, and writes out the newly modified piece of data back to memory:

```
-- first read the memory
srd("00000020", "babeface", cmd);
idle(cmd);
-- now get the actual data read
get_rdata(tv, cmd);
tv(15 downto 0) := tv(15 downto 0) - 16#0ac1#;
tv(31 downto 16) := tv(31 downto 16) + 16#1fbe#;
-- write it back
av := to_stdlogicvector(X"00000020");
wr(av, tv, cmd);
-- read it for verification
rd(av, tv, cmd);
```

Figure 7.0 Read-Modify-Write Cycle

An interesting side effect of placing these in a process is that a single-point of BFM instruction execution is formed, allowing multiple BFMs to be driven from one or different processes. Using any number of processes to control multiple BFM execution can eliminate the need for side-band signals between BFMs and allow greater test code control.

4.2.3 Configurations and Test Bench Building

Using configurations allows the multiple test code elements to be switched into the test bench. Separating code into functional pieces at a high level of abstraction can also enable software development and design reuse.

In the configuration example in section 8.2.9, the `ibc_tst1_cfg` configuration binds the `ibc_tst1` test code, the `ibc_bfm` BFM, and `ibc_dut` DUT entity/architecture pairs instantiated in the test bench architecture. Using this method a configuration that binds an individual piece of test code can be written to allow multiple test code architectures to be bound to the same test bench architecture.

Using configurations can be complex depending upon the test bench composition and the levels that require binding. However, configuration usage lends easy hookup to other abstractions of the models. For example, the DUT in the 8.2.9 example is bound to a configuration using a behavioral architecture. Once the DUT is synthesized into a gate-level representation, another configuration could be written to bind the gate-level architecture to the instantiated entity of the test bench without changing the test bench or test code, achieving gate-level simulation with timing. The use of configurations allows multiple abstractions to be connected without changing the overall connection in the test bench.

5.0 High Level Testing Paradigms and Caveats

5.1 BFM's with Timing

In the previous examples, no timing information was modeled in the BFM's. Timing on the stimulus to the DUT as well as timing checks on the receiving end from the DUT can be modeled in many different ways within the BFM. One method is to use “after” delays on signal assignments, such as the `ibc_bfm.vhd` file in Appendix 8.2.4. The BFM will assert the `ibc_rd_n` signal after a delay value specified in the generic port. Timing checks can be done using the event-based operations in VHDL as shown in the `check_block` section of the BFM example in 8.2.4. This is more complicated, since most timing events must be checked from a setup and hold of a reference edge but easily accomplished using the `'event` attribute and doing a little math from the last reference edge and the simulation time. In the `ibc_tst1_cfg.vhd` example in 8.2.10, a configuration is shown to set timing parameters and timing checks via generic settings through the configuration.

5.2 Multiple BFM's, Multiple Processes, Single Package

It is possible for multiple processes to drive multiple BFM's using one set of procedure calls. At first glance, this seems like an easy problem- just instance multiple BFM's, use the same package, and everything should work. Unfortunately, it's not that simple for the following VHDL reasons:

- Synchronization of the main procedure call to the BFM needs to understand which BFM to communicate with
- Multiple drivers on a signal of type record are not allowed in VHDL

One way to accomplish this is to array the command record so that each BFM uses a unique record. By doing this, the test code can distinguish which BFM should get the command and each BFM has a separated area from the rest. This also allows multiple processes to drive the global signal of type array of type record. An example is shown in figure 8.0.

```
type cycle_t is (idle, write, read);

type cmd_rec is record
  bus_cycle      : cycle_t;
  addr           : std_logic_vector(31 downto 0);
  data           : std_logic_vector(31 downto 0);
  rdata         : std_logic_vector(31 downto 0);
```

```

    request      : std_logic;
    busy         : std_logic;
end record;

constant MAX_BFM_C : integer := 2;
type instruction is array (0 to MAX_BFM_C) of cmd_rec;

signal cmd : instruction := ( -- record # 0
    (idle,
     (others => '0'),
     (others => '0'),
     (others => 'Z'),
     '0',
     'Z'),
    (idle, -- record # 1
     (others => '0'),
     (others => '0'),
     (others => 'Z'),
     '0',
     'Z'),
    (idle, -- record # 2
     (others => '0'),
     (others => '0'),
     (others => 'Z'),
     '0',
     'Z')
);

```

Figure 8.0 Multiple BFM Record and Signal Declaration

There are a couple of caveats to using this approach. Each BFM instantiated within the test bench must have some unique identifier. This can be resolved by adding a generic port of type integer and assigning the generic a unique value in the configuration. The second problem exists with multiple processes driving multiple BFMs: each process can drive one or more BFMs but two or more processes cannot drive the same BFM. This relates to the multiple drivers (in this case processes) on a signal of type record again. For example, process A drives BFM number 0 and BFM number 1. Process B cannot drive BFMs 0 or 1, but can drive BFM number 2. This rule also applies to the request and busy signals of the record. This relationship is shown below:

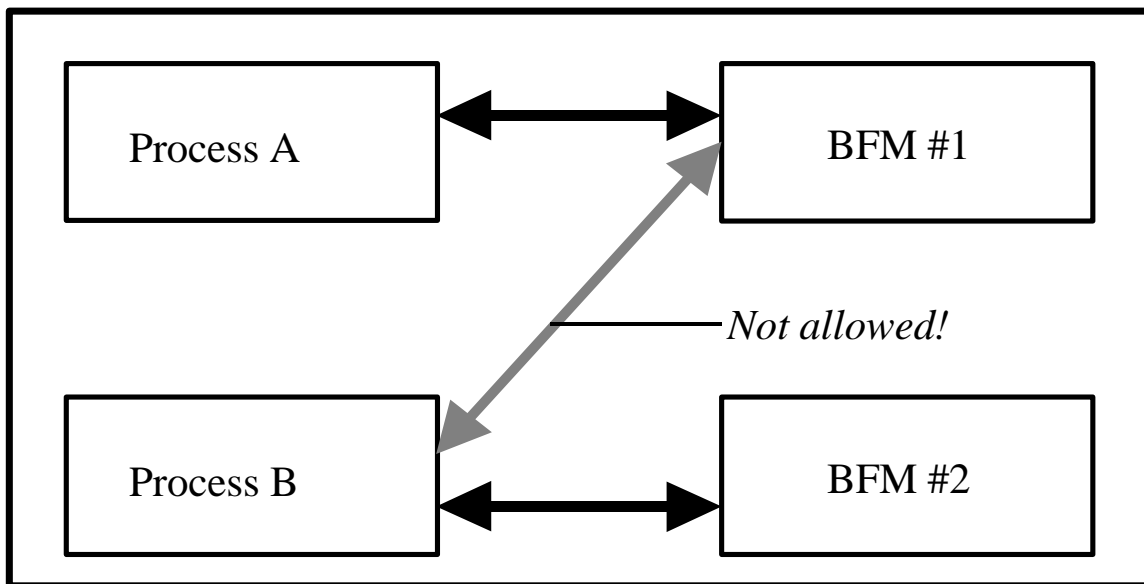


Figure 9.0 BFM Driving Relationship

For a full example, see Appendix 8.3.5, which uses two instances of the master BFM, `mbc_bfm.vhd`, instantiated in a test bench with a basic DUT and test code. The examples of 8.3 show how multiple BFMs can be created, connected, and driven utilizing a procedural-based test bench.

The advantages of multiple processes are very advantageous when doing multi-process type operations, i.e. multi-masters on a bus. Using a separate process to control each BFM with user-defined signals in the architecture to communicate between the processes offers the engineer greater flexibility in writing tests than straight BFL code with a limited communication channel between the BFMs. The test code processes utilize the VHDL language for looping, variables, and other elements.

5.3 BFMs for Performance

The method of modeling devices in VHDL can drastically affect simulation performance in terms of cycles-per-second (CPS). Generally, the lower the level of HDL code, the lower the CPS of the simulation. As an example, code at the gate-instantiated level runs the slowest, as more information is presented to the simulator on a gate-by-gate basis. The more port maps or instantiated components, the more work the simulator must do to resolve all signals in delta time segments and advance to the next time unit which slows simulator performance. In contrary, the higher the level of HDL coding, the faster the simulation will run due to less overhead and more containment of code within components or processes.

There are specific style methods when coding VHDL for speed that will improve performance. And these guidelines are general and may not be met due to model restrictions.

- Use as few processes as possible
- Limit component instantiation to a minimum
- Avoid using “after” delays
- Use variables within processes whenever possible
- Use built-in IEEE packages for conversions, types, functions, and procedures

5.3.1 Use as Few Processes as Possible

Processes take up simulation schedule resources and must get evaluated whenever the sensitivity list is modified or if a wait condition has been met, depending upon the coding style of the process. Signals in the sensitivity list must be evaluated every time there is a change in status or resolution. It is much better to code only a few processes rather than have many processes due to scheduling overhead.

5.3.2 Limit Component Instantiation to a Minimum

Component instantiation also causes schedule overhead similar to processes. The simulator must evaluate the connecting signals, schedule the pending events, and resolve the state. Limiting components reduce simulator overhead.

5.3.3 Avoid Using “after” Delays

Using “after” time assignments adds additional schedule elements within the simulator and any resolution function overhead required to resolve the signal state. The fewer the time events scheduled, the faster the simulation.

5.3.4 Use Variables in Processes Whenever Possible

Variables take approximately one-seventh the amount of overhead as signals. Since variables are contained within a process and not visible external to any other processes, no overhead for the resolution function is required. Variables also take up less memory in a system simulation and are easier for the simulator to work with.

The use of variables is the most beneficial when large arrays are coded. In the `ibc_dut.vhd` example, the simulator memory resident size was 280K larger using an array of signals than an array of variables for the 32 lines x 32 bits defined as `mem_t` type, which resulted in about a 256 byte per bit of signal in memory increase. Variables also save time in the simulator since they never need to be scheduled or resolved.

5.3.5 Use `std_logic` or IEEE Built-in Functions and Procedures Whenever Possible

Avoid using a custom resolution function or type for signals. Most simulators have the IEEE libraries coded as part of the simulator for faster performance: less overhead of compiled VHDL code parsing and the basic math and type resolution functions can be performed much faster within the simulator architecture. Most simulators take advantage of the `std_logic` type and the associated resolution table as a built-in native function. Creating a new bus type with a unique resolution function adds to the amount of work the simulator must do to schedule, resolve, and advance the simulation time. Creating a new procedure or function that may duplicate an IEEE-defined, built-in one just creates more overhead for the simulator to analyze, elaborate, and then schedule. When possible, utilize the IEEE package to the fullest for conversion functions, such as the function to convert an integer value into `std_logic` type, `conv_std_logic_vector()`. There are many IEEE builtin conversion functions that run much faster than a user-coded ones.

For a performance example, a 40K-gate model was coded for a project at Intel that originally used a custom resolution function and type for interconnecting buses within the model. This model was later converted completely to IEEE `std_logic` that resulted in a factor of over 4X for simulation cycles per second performance. The built-in IEEE packages were much faster in simulator performance and much more portable between engineers and projects with the use of `std_logic`.

5.4 Modifying the Synopsys PCI Source Model

The Synopsys PCI source model utilizes BFL code from a file or built-in procedure calls within the model. The models are very useful, but are limited in terms of simulation effectiveness with the BFL methodology. One way to fix this is to add in the hooks to allow execution via procedures from a global package. This modified PCI source model was used successfully at Intel to test and validate a PCI-based design.

5.4.1 Code Notes

Since the PCI models are licensed by Synopsys as source code, the new code additions and code locations needed to make the models procedure-based will only be given. A diff file (generated by `diff -e`) that can be applied to the PCI master revision 6.0 source code using the `patch` program is provided in section 8.4. The majority of the edits to the model were done in the user-defined section of the `pcimaster.vhd` file. A new package called `pcimaster_pkg.vhd` was created to contain the global signals and procedure definitions.

This modified BFM was used to validate a 32-bit PCI based design, so the 64-bit PCI function hooks are not present. Also, not all functions were completely implemented since the design project didn't require the entire PCI model function complement. However, most functions, structure and idea are included to give the user ample example to create any other needed procedures such as 64-bit memory reads and writes.

5.4.2 The `pci_pkg.vhd` File

This file contains all the procedures to drive the PCI BFM model. The main procedures call the `pci_mcyrc()` main procedure to communicate the test code process to the PCI master BFM. The command record contains the data fields required for the PCI BFM to execute the bus cycles. The command record, `mpci_rec_t`, is typed into an array and set to a signal, `mpci_rec`. This global signal is of type array so that each BFM numbered 0 through 5 can access each individually assigned record.

Each PCI command procedure must understand the master number of the BFM that is assigned the job, and the global signal, `mpci_rec`, which is passed into the procedure as an `inout`. For example, the PCI idle command uses the master number, number of cycles, and the global signal as the passing parameters:

```
pci_idle(0,5,mpci_rec)
```

In the example, master number 0 gets 5 PCI idle cycles. The procedures are setup in this manner to facilitate multiple PCI BFMs in the test bench and easier command execution. See appendix 8.4.1 for a full listing.

5.4.3 Summary of Edits to the `pcimaster.vhd` File

- Add generic to `pcimaster` port to enable BFL or procedure-based execution
- Add generic to `pcimaster` port for master selection number
- Add variables to the main process
- Add the handshaking, command parser, and execution code to the user-defined area

Save the contents of 8.4.2 in a file and use the `patch` program to apply the changes to the source code model.

5.4.4 Summary of Edits to the `pcimaster_fm.vhd` and `pcimaster_timing.vhd` Files

The changes to these files are mainly for the library naming convention. Apply the patch files using the `patch` program to the source code files for model modification.

6.0 Conclusion

Faster and more complex simulation operations can be achieved using procedural-based BFM's and VHDL coding techniques. The use of procedures in a global package to drive BFM's is much more flexible and faster than standard issue BFL code. Writing a VHDL test bench at a higher-level of abstraction enables more complex testing operations and better simulation performance which leads to more robust designs with fewer bugs at first silicon.

7.0 References

IEEE 1076-93 Standard VHDL Language Reference Manual, 1994 Institute of Electrical and Electronics Engineers, Inc, ISBN 1-55937-376-4

Synopsys PCI Source Models revision 6.0, Synopsys Inc.

Patch program, 1998 Larry Wall & 1997 Free Software Foundation, Inc, GNU copyleft agreement, <http://www.gnu.org>

8.0 Source Code Appendix

<http://www.primenet.com/~greggl/snug99.html>

8.1 Source for Basic BFM Test Bench

8.1.1 The generic_tb.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/generic_tb.vhd

8.1.2 The generic_tb_cfg.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/generic_tb_cfg.vhd

8.2 Source for Procedure-Based BFM Test Bench

http://www.primenet.com/~greggl/snug99/proc_bfm

8.2.1 The test_pkg.vhd File

http://www.primenet.com/~greggl/snug99/test_pkg/test_pkg.vhd

8.2.2 The ibc_bfm_pkg.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_bfm_pkg.vhd

8.2.3 The ibc_tb_sig_pkg.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_tb_sig_pkg.vhd

8.2.4 The ibc_bfm.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_bfm.vhd

8.2.5 The ibc_dut.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_dut.vhd

8.2.6 The testcode.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/testcode.vhd

8.2.7 The ibc_tb.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_tb.vhd

8.2.8 The ibc_tst1.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_tst1.vhd

8.2.9 The ibc_tst1_cfg.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_tst1_cfg.vhd

8.2.10 The ibc_tst2_cfg.vhd File

http://www.primenet.com/~greggl/snug99/proc_bfm/ibc_tst2_cfg.vhd

8.3 Source for Multi-BFM Test Bench

http://www.primenet.com/~greggl/snug99/multi_bfm/

The Multi-BFM source code references the `test_pkg.vhd` file from the 8.2.1 example.

8.3.1 The `mbc_bfm_pkg.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_bfm_pkg.vhd

8.3.2 The `mbc_bfm.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_bfm.vhd

8.3.3 The `mbc_tb_sig_pkg.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_tb_sig_pkg.vhd

8.3.4 The `mbc_dut.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_dut.vhd

8.3.5 The `mbc_tb.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_tb.vhd

8.3.6 The `mbc_tst1.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_tst1.vhd

8.3.7 The `mbc_tst1_cfg.vhd` File

http://www.primenet.com/~greggl/snug99/multi_bfm/mbc_tst1_cfg.vhd

8.4 Source for Synopsys PCI Source Model Multi-BFM Procedure Execution Edits

http://www.primenet.com/~greggl/snug99/pci_bfm_patch

8.4.1 The `pcimaster_pkg.vhd` File

http://www.primenet.com/~greggl/snug99/pci_bfm_patch/pcimaster_pkg.vhd

8.4.2 The `pcimaster_timing.pch` File

http://www.primenet.com/~greggl/snug99/pci_bfm_patch/pcimaster_timing.pch

8.4.3 The `pcimaster_fm.pch` File

http://www.primenet.com/~greggl/snug99/pci_bfm_patch/pcimaster_fm.pch

8.4.4 The `pcimaster.pch` File

http://www.primenet.com/~greggl/snug99/pci_bfm_patch/pcimaster.pch