

A BFM Simulation Strategy for Verilog

Rodney Pesavento
Michael Privett

Intel, Corp
Chandler, Arizona/USA

rodney.pesavento@intel.com
michael.privett@intel.com

Abstract

Don't wait until the system level to check out your complex blocks. Start quality simulation at the unit level. One method is to emulate other blocks in the design using Bus Functional Models. This presentation covers a simple way to write BFMs in Verilog that improve cycles per second of your test bench and allow a high level way to develop good functional tests. Pure Verilog BFMs are especially good for behavioral code testing. As a bonus some tips on improving simulation speed are included.

1.0 Introduction

I haven't use Verilog in five years. Though I never considered myself an expert, I saw the beauty in Verilog's simplicity. It did everything I wanted it to do. The company I worked for at that time possessed the technology to write efficient, readable bus functional models (BFMs). But what I had learned from the experts, I have long forgotten. Since then, I have mostly used VHDL. VHDL provides constructs, such as packages and access types, that work well for test benches and BFMs. However, benchmarks show that Verilog simulates faster.

Currently, I'm in a group looking at the benefits of Verilog. In the past we have written all our models in VHDL, but we need the speed of Verilog. Our simulations show VCS to be 2x to 8x faster than the fastest VHDL simulator. Since we haven't been using Verilog or VCS consistently throughout the years, we don't have all the answers. However we have leveraged our VHDL experience and have rewritten most of our models over in Verilog and have come up with some interesting ideas.

In writing this paper we hope to start a trend in sharing ideas on test benching and BFMs. Sharing ideas in verification technology can only benefit the design community. Think of the money projects can save by not buying test bench development tools, but tapping into a good repository of information on the construction of good BFMs and test benches.

2.0 What Is a BFM

A BFM simply models the bus interface of some unit, regularly a microprocessor. It does not contain the RTL or gate level specifics of the unit's internal workings. The purpose of a BFM is to gain simulation speed, ease of use, and ease of creation.

2.1 Hard Model

Hard models are fully functional RTL or gate level models. They contain all the internal structures of the actual device. This causes hard models to be very inefficient for driving stimulus. If a hard model such as a processor was bolted into the system, tests would have to be written in assembly code. The verification environment would have to have an assembler for the ASM code and the processor would require binary files to load into a memory. In the system, the processor will fetch from memory, every instruction it executes. The instruction might be an indirect write to the unit under test. In such a case, the system would read from memory, to get the address, then write to the unit. A hard model would also have to simulate all the internal structures of the unit. The potential for an instruction causing address calculations, additions, jumps, etc., is high. A hard model has many internal structures that have to perform some function before it can create the bus cycle desired, making it a slow and inefficient testing tool.

2.2 BFM Model

A BFM is better than a hard model. Even if the hard model is very fast, hypothetically, as fast as the BFM, its throughput efficiency is still slower. Because the processor model still performs instruction fetches over 50% of the time, it's effective throughput is diminished. At best case it would really behave as if it were half the speed of a BFM. The BFM executes the write or read directly, without an instruction fetch. The test code is written directly in Verilog so a separate

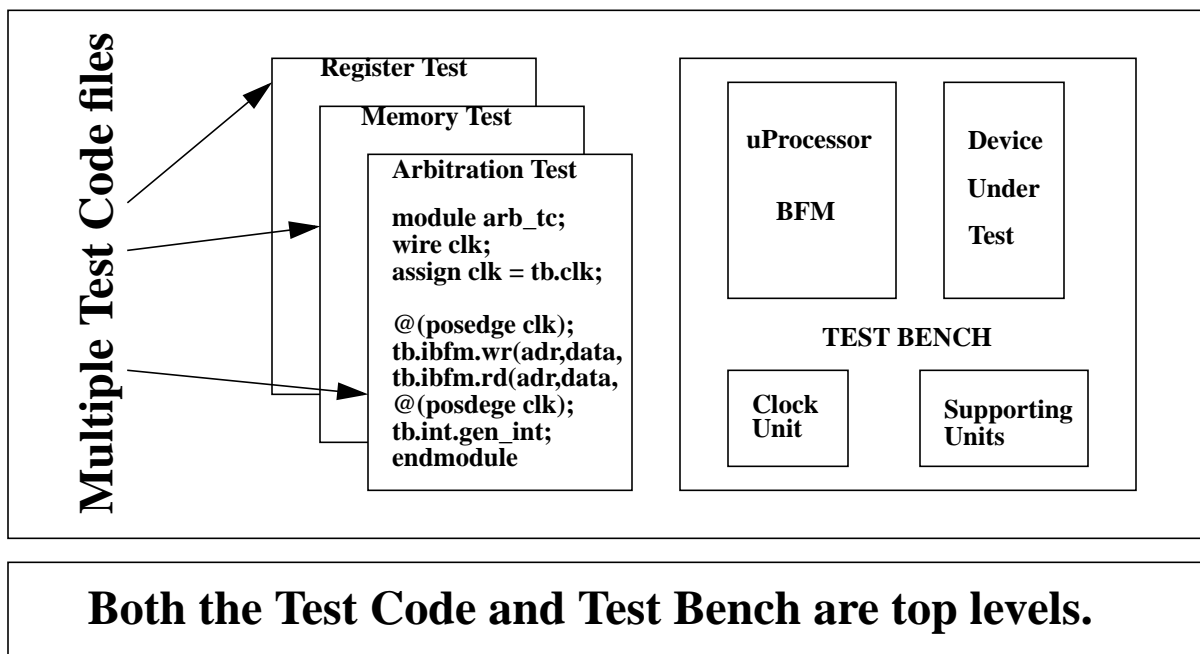
compiler is not needed.

A BFM has fewer constructs than a hard model, making fewer structures to simulate, which, in turn, makes it faster. The internal workings of a BFM generally consist of tasks that initiate or perform the bus cycles. Two types of BFM are discussed in this paper. The first uses tasks to setup a state machine to perform the actually bus cycles. Though more flexible, it is more complicated to design. The second one is best for unit level tests. It uses only tasks and is not meant for a bus where it is not the only master.

3.0 Test Bench Overview

Verilog supports multiple top levels. Generally only two are needed, one for the test bench and one for the test code. The test bench contains one or more of the following types of units: a clock driver, a BFM, and a unit under test as shown in Figure 1. By having a test bench that uses a separate test code file, multiple independent test code files can be created to organize testing. Only one is used to drive the test bench for a given simulation, which makes for improved compile times.

Figure 1. Test Bench Block Diagram



3.1 Organization

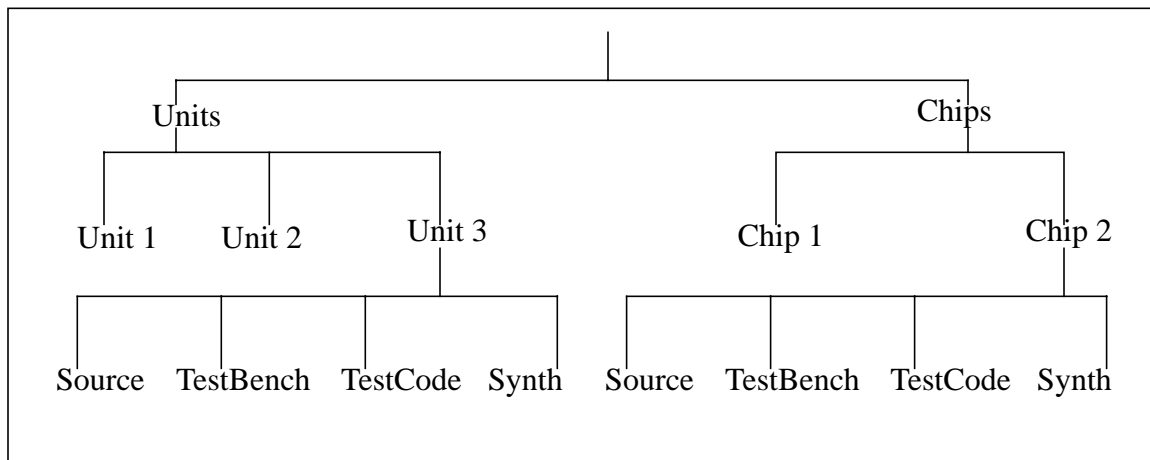
Put every file in its place. Having a usable directory structure provides benefits for both coding and testing. Use a structure that makes it easy to add externally created source code, models and tests. This usually requires a methodology that is structured enough for scripts to work, but flexible enough so they don't break when a non-conforming model is added.

It is all too easy to have one directory that contains all the files, but, that doesn't support multiple engineers working on the unit, some for design and some for testing. Even if your using CVS, a

revision control program, updates and sharing get tricky. On the other extreme, it is also not efficient to have a directory for every type of file. Such a structure is very cumbersome and yields few, if any, benefits.

The structures that have been most efficient are ones in which the unit level directory hierarchy mirrors the system level hierarchy. Such a system is usually created by making good compromises between the necessities of both the unit and system levels. Figure 2 shows a structure that has been proven to be easy to use for both designing and testing at the two levels. Not shown in the figure is the models directory, which usually does not conform to the standard because models are often legacy, purchased, or a single file. Models often have their own directory structure anyway.

Figure 2. Design Directory Structure



3.2 Scripts

When creating a test bench for a unit it is a good idea to use a small set of automated scripts to do the work. As always, first automate the tedious tasks like typing in the port declaration for an instanced block. Use a naming convention that allows the ports to connect by name, without re-naming, by using the same signal at each level of hierarchy on each block to which it is connected. Create a script that grabs the module declaration and converts it to an instance declaration. Next, add in some reporting. List each signal and how many connections it has. When reviewing the reports make sure every signal has more than one connection.

Another good script to have handy is a tag file creator for the vi editor. A tag file allows the designer to move through the design file hierarchy inside the vi editor. Creating a tag file is simple to do using Perl. Write a script that searches all Verilog files for the following key words: module, function, and task while excluding their “end” counterparts. Save those lines in an array that contain the key word, the file name including path, and the line number. It isn’t even necessary to strip out the unwanted parts, except for the newline marker. Finally, use the Perl sort command to sort the array and then print it. Table 1 shows a snip-it of code for creating a tags file.

Table 1: Verilog Tag File Creator

```
#!/usr/bin/perl
require "find.pl";

@FileList = (); @TagList = (); # create two arrays, one for the file list and one for the tag list

$pwd = `pwd`;
chop($pwd);
&find("$pwd"); # find all the verilog files under the pwd hierarchy see sub wanted

foreach $FileTag (@FileList) {
  open(FILE, "$FileTag") || die "Can't open file $FileTag: $!\n";
  @KeyList = (); # initialize an array for the key word list

  while (<FILE>) {
    if (!(/^[\s]*//) && !(/^\s*.*$/ && (^*$$/ || !(\s*/))) &&
        !(/^[s]*endmodule/ || /^[s]*endfunction/ || /^[s]*endtask/) &&
        ((/^[s]*module\s[\s]*\S[\S]*[s]*/) ||
         (/^[s]*function\s[\s]*.*/) ||
         (/^[s]*task\s[\s]*\S[\S]*/))) {

      @KeyList = (@KeyList, $_); # put lines that contain the key work in the key word list
    } # end if
  } # end while
  close(FILE);

  foreach $KeyLine (@KeyList) {
    chop($KeyLine);
    $_ = $KeyLine;
    if (/^[s]*module\s[\s]*\S[\S]*[s]*[(\s]*/) {
      $KeyWord = $1;
      $TagLine = $KeyWord . ' ' . $FileTag . ' ' . "/" . $KeyLine . "\n";
      @TagList = ( @TagList, $TagLine );
    }
    # create elsif for task, function, and vectored function using the above "if" as a template
    ...
  } # end if
} # end inner foreach
} # end outer foreach

print sort( @TagList ); # print the tags to stdout in alphabetic order
exit;
## end of main program

# sub programs
sub wanted {
  # add any file that has a .v extention but not if it exists in a directory called "gates"
  if (/^.*\s\.v$/ || (/^gates$/ && ($prune=1))) {
    @FileList = ( @FileList, $name);
  }
}
## end of file
```

3.3 Coding Tips

For simple, high speed test benches, the clock generator will dictate the maximum speed at which it can run. It's not obvious but two different clock generators can differ greatly in performance. The standard one that everybody uses is the obvious one (See Table 2). The performance does not

Table 2: Clock Generator Techniques

Standard Clock 1	Standard Clock 2	Clock Trick
<pre>always begin : clk1_blk clk1 = 0; #10 clk1 = 1; #10; end // clk1_blk</pre>	<pre>initial clk1 = 0; always #10 clk1 = ~clk;</pre>	<pre>always begin : clk1_blk clk = 0; forever #10 clk = ~clk; end // clk1_blk</pre>

vary if using an initial statement and an always. Nor does it matter if the signal is directly assigned to a value or if the signal is assigned to the inverse of itself. However, adding one little trick of using a forever inside an always improves the speed of the clock generator by 25%. It is also important to not that a blocking assignment is almost twice as fast as the non-blocking. Now, this is just the intrinsic speed of the clock block. But, it does effect the overall simulation speed, considering that the clock block toggles twice per cycle.

Two clocks do not slow the simulation down by twice as much. So using two clocks close in period but offset from each other only slows the simulation down by 40%, even though there are twice as many events. However, the impact can be minimized. This is the part that is counter intuitive. When more than one clock is needed, create a clock module that is parameterized. A simulation with ten clocks using the periods: 30, 50, 70, 110, 130, 170, 230, 310, and 370, showed the difference between ten instanced clocks and ten clocks created using ten always blocks. The instanced clocks ran about 60% faster than the ten clocks modeled using ten always statements.

Table 3 shows raw numbers from the simulations. All simulations were run on a Sun Ultra 60 using VCS 4.0.2. The unix command "time" was used to report user time. Since there were no other jobs running, user time accounted for about 99% of total time. Also, all simulation results were done using batch mode. Interactive simulations were run to verify that all the clocks were toggling correctly.

Table 3: Clock Sim Numbers For Non-Blocking Assignment

Description	CLK Cycles	Sim Time	cycles/sec
Single Clock using either standard model.	100M	367 sec	272K
Single Clock using forever loop	100M	286 sec	349K
Two Clocks using standard model, with the same period but offset. Cycles refers to accumulative total of all clocks	200M	657 sec	304K

Table 3: Clock Sim Numbers For Non-Blocking Assignment

Description	CLK Cycles	Sim Time	cycles/sec
Two Clock using forever loop, with the same period but offset. Cycles refers to accumulative total of all clocks.	200M	462 sec	433K
Multiple Clocks using multiple always statements (in-lined). Ten clocks were created using 10 always statements. Each always used the forever loop trick. Cycles refer to the accumulative total of all clock cycles.	~104M	439 sec	237K
Multiple Clocks using a parameterized module. A clock module was created and instanced 10 times using parameters to change the period. Cycles refer to the accumulative total of all clock cycles.	~104M	304 sec	342K

3.4 Timescale

For most simulations, basically anything without back-annotated gates, it is very important to have the time unit and time precision be the same, and no more precise than needed. For all the simulation listed above, the timescale was defined as 1ns/1ns. It could have been defined as 10ns/10ns, but then the delay numbers would have to be changed to reflect a 10ns unit time. Interestingly, the simulation only runs slightly faster, but a 10ns resolution would not be usable for most clock periods or external models with delays. Table 4 shows raw simulation run times contrasting the different performance of timescale values. Simulation time was two billion ns.

Table 4: Comparison of Sim Times for Timescale Values for Blocking Assignment

Test	CLK Cycles	1ns/1ns	1ns/100ps	1ns/1ps
Standard Single Clock	100M	201 sec	441 sec	457 sec
Standard Dual Clock	200M	355 sec	1184 sec	988 sec
Improved Single Clock	100M	138 sec	388 sec	391 sec
Improved Dual Clock	200M	187 sec	1100 sec	852 sec
Multiple Clocks In-lined	104M	201 sec	693 sec	825 sec
Multiple Clocks Instanced	104M	142 sec	383 sec	361 sec

Comparing the values in Table 4, column “1ns/100ps” to “1ns/1ps”, it appears that using a resolution greater than the base of precision actually works to slow the simulation speed. But further testing showed that the slow down only occurred when the unit time and precision time differed in the base time units used, (i.e. ns vs. ps). A simulation using a timescale of “10ns/10ns” completed in the same time as one with a timescale of “1ns/1ns”.

4.0 Adding Timing to Verilog Models

The Verilog language makes it very easy to add timing and timing checks to a model. The `specify` block allows the designer to add path delays to a model or a design for simulation. A `specify` block accomplishes this by adding timing between an input source and an output destination. The path cannot start at an internal point or end at an internal point. Also the output must be connected to a signal defined as a wire, not a reg construct.

Though in-line delays produce slightly faster simulation times, they lack the flexibility that the `specify` block provides. `Specify` blocks are convenient because they keep all the model delays grouped together. In-line delays are more tedious to change because they are scattered throughout the code. That is not to say, “don’t use in-lines,” but, use them sparingly and when necessary. For instance, a good place to use an in-line delay is for the clock generator as shown in Table 2. An in-line delay would also be used if it were necessary to have a delay on an internal module path.

4.1 Adding Delays

The `specify` block allows for a very flexible application of the delay data. It supports the standard `min:typ:max` delay structure. But, if only one delay is specified it is used for each delay condition. It supports different timing for each of the six possible output transitions, (0->1, 0->Z, 1->0, 1->Z, Z->0, Z->1), which allows separate timing for turn off times for a 3-state output. The six X transitions can be explicitly specified or inferred from the standard ones. It also handles applying delays bit-by-bit from one vector to another, or an all-bits to all-bits. The later is useful for a decoder. Path delays can also be dependent on the state of a control signal. The signal controlling the delay must be an input or an inout or a compile time constant. Other features exist, these are just some of its more useful capabilities.

Table 5 shows a `specify` block for an address select decoder. Each input of the decoder affects each output, so the full module path construct, `*>`, is used.

Table 5: Specify Block Example

```
module address_decode(addr,sel);
  input [2:0] addr; // 3 bit address slice from address[20:18]
  output [7:0] sel; // 8 bit select for each of the 8 units
always begin
  addr = 0;
  addr[sel]=1;
end
specify
  specparam rise_delay = 1:2:3; // specify min:typ:max rise time delay
  specparam fall_delay = 1:2:3; // specify min:typ:max fall time delay
  (addr *> sel) = (rise_delay, fall_delay); // use *> so each addr bit affects each sel bit
endspecify
endmodule
```

4.2 Adding Checks

Verilog supplies eight system tasks to perform timing checks. They are listed in Table 6. Generally, models only need the \$setuphold task. The separate versions provides a mechanism to selectively turn off the setup or the hold checks for a max delay or a min delay simulation. Use the other tasks to check constraints when testing a gate level device that has back-annotated timing. For instance, the \$period task can check the output ports of a gated clock to verify that the period has not diminished because of the gate.

Table 6: Verilog System Task for Timing Checks

\$setup	data event must occur at least a minimum time before reference event
\$hold	data event must occur at least a minimum time after reference event
\$width	an edge to edge reference event has at least a minimum width
\$period	a same edge to same edge reference event has at least a minimum width
\$skew	data event must occur within a maximum time after the reference event
\$recovery	data event must occur at least a minimum time after an edge reference event
\$setuphold	combines the \$setup and \$hold system tasks into one.
\$nochange	check that the data event does not occur between the start and end offsets from the reference event

5.0 A State Machine Based BFM

The system level test bench usually needs to be robust in order to achieve the quality of verification required by today's highly integrated designs. This often means that more than one bus exists on the chips interface. Since these buses generally are not synchronous to each other, models must execute independent from each other. A state machine (SM) based BFM is a good way to apply dynamic stimulus to the device under test.

5.1 Implementation Issues

By definition, the state machine based BFM uses a state machine to generate the bus cycles. For a processor, the SM would generate memory read, memory write, I/O read, and I/O write cycles. These type of cycles are usually easy to represent in a state machine. State machines are also good at handling bursting and early termination. It could also be setup to handle special cycles like interrupt acknowledge or shutdown.

The BFM can have multiple state machines. A second SM could be designed to handle interrupts. If an interrupt occurs during an active bus cycle, the second SM could capture it and start the decode of calling the interrupt service routine (ISR). However, the BFM does not actually contain the code for the ISR. This is because the ISR is written by the user of the BFM to accomplish whatever is needed for that particular test environment. To support this, the BFM would have predefined a module to contain the task for the ISR. That file would exist in the users test code area and would be compiled into the test bench for the simulation.

The test code issues cycles to the BFM via self checking tasks. Each task correlates to a particular bus cycle, like a memory read. A read task sets up the address and data expected for the read as well as the wait states that the read should take. If the data expected is different from the actual, a message is displayed to report the error. Likewise, if the wait state value is not met, a message is once again displayed.

Special tasks can be created to handle errors. Certain errors may be minor, only requiring reporting. The user may determine that data compare errors are not catastrophic. However, a wait state error may indicate that the device under test is no longer working properly. In such a case, the user has set up the message task to display the error and then call the \$finish system command to stop the simulation.

The state machine is kept in sync with the test code by using a busy flag. When the SM is in its Idle state the busy flag is false, but when it is in an active state the busy flag is true. The flag prevents a second task request from being granted if one is already in service. This is important as the task exits after it sees the busy signal asserted. Doing so allows the test code to call a task from another BFM controlling another bus. Having this level of control allows for a test to set up two different buses attempting access to a shared resource to test arbitration or collision.

5.2 BFM Performance

The BFM example shown in Figure 3 of Appendix A, shows a simple state machine BFM with timing. This BFM has tasks to initiate 8-bit memory reads and write, and 8-bit I/O reads and writes. Three other tasks are included to perform an idle cycle, request a system reset, and an internal task to check read data.

The supporting test bench file is shown in Figure 4. It connects the BFM to a clock generator and a device under test, simply called “chip”. The chip, described in Figure 5, contains a simple bus controller, see Figure 6, that drives the internal RAM memory model from Figure 7. In order to show the speed of the BFM, this test bench is designed to represent a minimum load. Table 7 show performance numbers for the state machine BFM. The test code is shared between the two BFMs and is in Figure 10, it performs a series of writes and reads from the memory.

Table 7: State Machine BFM Performance Numbers

Special Condition	cycles	time	cyc/sec
no delays, no timing checks	6.3M	50 sec	130K
delays w/o timing checks	6.3M	90 sec	70K
delays with timing checks every clock	6.3M	93 sec	67.7K
delays with timing checks qualified by read strobe	6.3M	93 sec	67.7K

As shown above, checking every clock, as opposed to checking only under a strobe condition, does not change the simulation time. This indicates that timing checks should be coded the way they are needed, not because of a simulation speed issue.

6.0 A Task Based BFM

Use task based BFM for unit testing. Units are often less complex than the whole system, and hence, do not need a robust test bench. A simple BFM can facilitate the early testing of a complex block especially if the unit has a simple interface or just one bus interface. The task based BFM is extremely efficient if the device under test performs many calculations but uses relatively few bus access cycles to keep it going. The reason for this is, the BFM is not looping through an idle state every clock cycle. The BFM does not toggle any signals when a task is not active. Nor, does it make any decisions based on input when a task is not active.

6.1 Implementation Issues

The BFM uses tasks to drive the signals of the bus. Each task is responsible for a type of bus cycle, for example, an 8-bit read. The task then drives the signals during the correct clock period to execute read cycle. The task would also be responsible for comparing the data expected to the actual data for that 8-bit read cycle.

As with the state machine BFM, the test code calls the task by specifying the Verilog hierarchy that contains the test. So, one top level module, the test bench, contains the BFM and the device under test, and another top level module, the test code, contains the tasks to execute. Unlike the state machine BFM, the task BFM does not return to the test code before the cycle is finished. That is because the task creates the wave form and does not just kick off a state machine to generate the cycle. This causes a hassle for units needing two independent BFM's.

The way around this limitation is to use the Verilog commands “fork” and “join”, which cause the statements between them to be executed concurrently. Table 8 shows an example of a fork-join construct used to drive two different buses in a test bench. Such a test might cause a collision for the read, hence testing an arbitration unit. Though this technique can be tricky in getting the timing just right, it is much easier than having an always loops for the processor bus and another for the PCI bus. Hand shaking for flow control creates more complexity in the two always loops approach.

Table 8: Fork-Join Example BFM Example

```
always begin : tc1
  fork
    tb.pci_bfm.rst(3);
    tb.uproc_bfm.rst(5);
  join
  ...
  fork
    tb.pci_bfm.rd(addr,data,ws3);
    tb.uproc_bfm.rd8(addr,data,ws5);
  join
  ...
```

6.2 BFM Performance

The task based BFM, as shown in Figure 8, is designed to directly stimulate the memory model. Its test bench, see Figure 9, consists of the BFM, a clock generator, and the RAM, which was reused from the chip module. Though the real reason to use a task based BFM is that it does not consume simulation time when it is not applying a cycle to the bus, it is hard to show its performance with such a test case. So, the test shows the performance of the BFM when running cycles.

Table 9: Task BFM Performance

Special Condition	cycles	time	cyc/sec
no delays, no timing checks	6.3M	37 sec	170K
delays w/o timing checks	6.3M	72 sec	87K
delays with timing checks every clock	6.3M	83 sec	76K
delays with timing checks qualified by read strobe	6.3M	83 sec	76K

The biggest benefit of the task BFM is that when it is not running bus cycles it doesn't take up much simulation time.

7.0 Conclusions and Recommendations

This paper has described many useful mechanism for designing and using BFMs. They are simple to implement and easy to understand. However, these techniques should not be taken as the best way to accomplish a goal. There is always a better way, we just have to find it. But, this is a good starting point.

Where do we go from here? There are many model requirements that are not present in these ideas. It would be nice to have a repository that stored examples such as these. Some ideas to be shared would be good examples of an efficient ROM, interrupt control, connect-disconnect bus, cache in a BFM, etc. As a design community we can either band together and share ideas or pay for consulting or special point tools that obfuscate the testing environment.

By using and expanding the two BFM techniques the ASIC design cycle can be shortened. Verification of units using the task based BFM allows for fast simulations. If the task based BFM and the state machine BFM use the same task names for calling identical cycles, most of the unit level tests can be converted to system level tests. This methodology allows for the easy transformation of unit level test to system level regressions. Having a good set of tests ready to go when the system is brought up saves engineering time.

8.0 Appendix A

This appendix contains the code examples for the state machine and task BFMs. It also contains the supporting files such as test benches. Refer to the sections on the BFMs for further description.

Figure 3. State Machine Based BFM Code Example

```

\timescale 1ns/1ns
\define IDLE 0
\define RD_RDY 1
\define WR_RDY 2

module ebfm(clk, rdy_n, rst_n, addr, ads_n,
            bfm_rst_req, mio_n, rd_n, wr_n, data);

input    clk;    // mpu clock
input    rdy_n;  // data ready
input    rst_n;  // bus reset
output [15:0] addr; // address bus
output   ads_n;  // address/data stobe
output   bfm_rst_req; // reset request
output   mio_n;  // memory/io
output   rd_n;   // read strobe
output   wr_n;   // write strobe
inout [ 7:0] data; // bidirectional data bus

reg [15:0] addr;
reg   ads_n;
reg   bfm_rst_req;
reg   mio_n;
reg   rd_n;
reg   wr_n;

tri [7:0] data;

reg [3:0] state; // bus cycle state
reg [3:0] lws;  // local wait state
reg [7:0] ldata; // local data
reg [15:0] laddr; // local address

reg busy; // bus cycle busy signal
reg idle_req; // request for idle cycle
reg iord_req; // request for io read cycle
reg iowr_req; // request for io write cycle
reg mrd_req; // request memory read cycle
reg mwr_req; // request memory write cycle

assign data = (wr_n) ? 8'hz : ldata;

initial begin : init_bfm_blk

```

```

state = 0;
busy = 0;
bfm_rst_req = 0;
mrd_req = 0;
mwr_req = 0;
idle_req = 0;
iord_req = 0;
iowr_req = 0;
end

always @(posedge clk or negedge rst_n)
begin: bus_ctl_blk
if (!rst_n) begin
lws <= 0;
state <= `IDLE;
addr <= 0;
ads_n <= 1;
mio_n <= 1;
rd_n <= 1;
wr_n <= 1;
busy <= 0;
end
else begin
case (state)
`IDLE : begin
case (1'b1)
mrd_req : begin
lws <= 0;
state <= `RD_RDY;
addr <= laddr;
ads_n <= 0;
mio_n <= 1;
rd_n <= 0;
wr_n <= 1;
busy <= 1;
end
mwr_req : begin
state <= `WR_RDY;
addr <= laddr;
ads_n <= 0;
mio_n <= 1;
rd_n <= 1;
wr_n <= 0;
busy <= 1;
end
end
iord_req : begin

```

```

state <= `RD_RDY;
addr <= laddr;
ads_n <= 0;
mio_n <= 1;
rd_n <= 0;
wr_n <= 1;
busy <= 1;
end
iowr_req : begin
state <= `WR_RDY;
addr <= laddr;
ads_n <= 0;
mio_n <= 1;
rd_n <= 1;
wr_n <= 0;
busy <= 1;
end
idle_req : begin
state <= `IDLE;
addr <= 0;
ads_n <= 1;
mio_n <= 1;
rd_n <= 1;
wr_n <= 1;
busy <= 1;
end
default : begin
state <= `IDLE;
addr <= 0;
ads_n <= 1;
mio_n <= 1;
rd_n <= 1;
wr_n <= 1;
busy <= 0;
end
endcase
end
`RD_RDY : begin
lws = lws + 1;
if (rdy_n) begin
state <= `RD_RDY;
addr <= laddr;
ads_n <= 1;
mio_n <= mio_n;
rd_n <= rd_n;
wr_n <= wr_n;

```

```

busy <= 1;
end
else begin
state <= `IDLE;
addr <= 0;
ads_n <= 1;
mio_n <= 1;
rd_n <= 1;
wr_n <= 1;
busy <= 0;
if (mrd_req || iord_req)
check_data;
end
end
`WR_RDY : begin
lws = lws + 1;
if (rdy_n) begin
state <= `WR_RDY;
addr <= laddr;
ads_n <= 1;
mio_n <= mio_n;
rd_n <= rd_n;
wr_n <= wr_n;
busy <= 1;
end
else begin
state <= `IDLE;
addr <= 0;
ads_n <= 1;
mio_n <= 1;
rd_n <= 1;
wr_n <= 1;
busy <= 0;
end
end
endcase
end
end // bus_sm_blk

////////////////////////////////////
task check_data;
integer i;
integer rd_error_v;
begin
for (i=0;i<=7;i=i+1) begin
if ((ldata[i] !== 1'bx) &&

```

```

        (ldata[i] !== data[i]))
        rd_error_v = 1;
    end // for i
    if (rd_error_v)
        $display("Error: Expected %x, but Actual
%x\n",ldata,data);
    end
endtask
////////////////////////////////////////////////////////////////
task bfm_rst;
begin
    bfm_rst_req <= 1;
    @(posedge clk);
    bfm_rst_req <= 0;
end
endtask
////////////////////////////////////////////////////////////////
task idle;
    input [31:0] cyc_cnt;

    integer i;
begin

    for (i=0;i<=cyc_cnt;i=i+1) begin
        // if busy, wait for not busy
        if (busy) @(negedge busy);

        idle_req <= 1; // gen idle request

        ldata <= 0; // setup local data
        laddr <= 0;
        lws <= 0;

        // wait for cycle to start, then exit
        if (!busy) @(posedge busy);

        idle_req <= 0;
    end
end
endtask
////////////////////////////////////////////////////////////////
task memrd8;
    input [15:0] taddr; // task address
    input [ 7:0] tdata; // task expected data
    input [ 3:0] tws; // wait states

```

```

begin

    // if busy, wait for not busy
    if (busy) @(negedge busy);

    mrd_req <= 1; // gen read request

    ldata <= tdata; // setup local data
    laddr <= taddr;
    lws <= tws;

    // wait for cycle to start, then exit
    if (!busy) @(posedge busy);

    mrd_req <= 0;

end
endtask // memrd8
////////////////////////////////////////////////////////////////
task memwr8;
    input [15:0] taddr; // task address
    input [ 7:0] tdata; // task write data
    input [ 3:0] tws; // wait states

begin

    // if busy, wait for not busy
    if (busy) @(negedge busy);

    mwr_req <= 1; // gen write request

    ldata <= tdata; // setup local data
    laddr <= taddr;
    lws <= tws;

    // wait for cycle to start, then exit
    if (!busy) @(posedge busy);

    mwr_req <= 0;

end
endtask // memwr8

endmodule
// end of file

```

Figure 4. SM BFM Test Bench

```

`timescale 1ns/1ns

module ebfm_tb;

    reg    clk;
    reg    rst_n;
    wire [15:0] addr;
    tri [ 7:0] data;

    ebfm eb (
        .clk(clk), .rdy_n(rdy_n), .rst_n(rst_n),
        .addr(addr), .ads_n(ads_n),
        .bfm_rst_req(bfm_rst_req), .mio_n(mio_n),
        .rd_n(rd_n), .wr_n(wr_n), .data(data));

    chip c1 (
        .addr(addr), .ads_n(ads_n), .clk(clk),
        .rst_n(rst_n), .mio_n(mio_n), .rd_n(rd_n),
        .wr_n(wr_n), .rdy_n(rdy_n), .data(data));

    always begin : clk_blk
        clk = 0;
        forever begin
            #10 clk = ~clk;
        end
    end

    always begin : btn_rst_blk
        // come out of reset automatically
        rst_n <= 0;
        @(posedge clk);
        @(posedge clk) rst_n = 1;

        // but allow for the tests to generate a reset
        forever begin
            @(posedge bfm_rst_req);
            rst_n = 0;
            @(negedge bfm_rst_req);
            @(posedge clk);
            @(posedge clk) rst_n = 1;
        end
    end
endmodule
// end of file

```

Figure 5. Chip Top Level

```

`timescale 1ns/1ns

module chip(addr, ads_n, clk, rst_n, mio_n,
            rd_n, wr_n, rdy_n, data);

    input [15:0] addr;
    input  ads_n;
    input  clk;
    input  rst_n;
    input  mio_n;
    input  rd_n;
    input  wr_n;
    output  rdy_n;
    inout [ 7:0] data;

    wire [11:0] iaddr;

    busctl b1 (
        // inputs
        .addr(addr), .ads_n(ads_n),
        .clk(clk), .rd_n(rd_n),
        .rst_n(rst_n), .mio_n(mio_n),
        .wr_n(wr_n),
        // outputs
        .ird_n(ird_n), .iwr_n(iwr_n),
        .rdy_n(rdy_n), .sela(sela),
        .seld(seld), .selm(selm),
        .selr(selr)
    );

    mem1024x8 m1 (
        // inputs
        .addr(addr[9:0]),
        .clk(clk),
        .rd_n(ird_n),
        .selm(selm),
        .wr_n(iwr_n),
        // inout
        .data(data)
    );
endmodule
// end of file

```

Figure 6. Bus Control for Chip

```
timescale 1ns/1ns

define IDLE 0
define WS0 1
define WS1 2
define WS2 3
define WS3 4

module busctl (addr, ads_n, clk, rd_n, rst_n,
              mio_n, wr_n, ird_n, iwr_n, rdy_n,
              sela, seld, selm, selr);

input [15:0] addr;
input  ads_n;
input  clk;
input  rd_n;
input  rst_n;
input  mio_n;
input  wr_n;
output ird_n;
output iwr_n;
output rdy_n;
output sela;
output seld;
output selm;
output selr;

reg iwr_n;
reg rdy_n;
reg sela;
reg seld;
reg selm;
reg selr;

reg local_hit;
reg [2:0] state;

reg ws0;
reg ws1;
reg ws2;
reg ws3;

assign ird_n = rd_n;
```

```
always @(addr or mio_n) begin :
res_decode_blk
sela = 0;
seld = 0;
selm = 0;
selr = 0;
ws0 = 0;
ws1 = 0;
ws2 = 0;
ws3 = 0;

case ( {addr[15:10],mio_n} )
7'b000000_1 : begin
selm = 1;
ws0 = 1;
end
7'b000001_1 : begin
selr = 1;
ws1 = 1;
end
7'b000000_0 : begin
sela = 1;
ws3 = 1;
end
7'b000001_0 : begin
seld = 1;
ws2 = 1;
end
endcase

local_hit = sela || seld || selm || selr;
end // res_decode_blk

always @(posedge clk or negedge rst_n) begin
: bus_sm_blk
if (!rst_n) begin
rdy_n <= 1;
state <= 0;
iwr_n <= 1;
end
else begin
case (state)
`IDLE : begin
if (!ads_n && local_hit) begin
state <= `WS0;
iwr_n <= ~ws0 | wr_n;
```

```

        rdy_n <= ~ws0;
    end
    else begin
        state <= `IDLE;
        iwr_n <= 1;
        rdy_n <= 1;
    end
end
end
`WS0 : begin
    iwr_n <= ~ws1 | wr_n;
    rdy_n <= ~ws1;
    if (ws0)
        state <= `IDLE;
    else
        state <= `WS1;
    end
end
`WS1 : begin
    iwr_n <= ~ws2 | wr_n;
    rdy_n <= ~ws2;
    if (ws1)
        state <= `IDLE;
    else
        state <= `WS2;
    end
end
`WS2 : begin
    iwr_n <= ~ws3 | wr_n;
    rdy_n <= ~ws3;
    if (ws2)
        state <= `IDLE;
    else
        state <= `WS3;
    end
end
`WS3 : begin
    iwr_n <= 1;
    rdy_n <= 1;
    state <= `IDLE;
end
endcase
end
end // bus_sm_blk

endmodule

// end of file

```

Figure 7. 1024 Word by 8-bit Memory

```

`timescale 1ns/1ns

module mem1024x8
(addr,clk,rd_n,selm,wr_n,data);

    input [9:0] addr;
    input  clk;
    input  rd_n;
    input  selm;
    input  wr_n;

    inout [7:0] data;

    tri [7:0] data;
    reg [7:0] mem [0:1023];

    // data is latched in on the rising edge of wr_n
    always @(posedge clk) begin : memwr_blk
        if (selm & ~wr_n)
            mem[addr] = data;
    end // mem_blk

    assign data = (selm & ~rd_n) ? mem[addr] :
    8'hz;

endmodule
// end of file

```

Figure 8. "Task Based BFM Code Example

```

`timescale 1ns/1ns
`define MAX_TIMEOUT_C 10

module ibfm( clk, addr, rd_n, rst_n, sela,
            seld, selm, selr, wr_n, data);

input      clk;          // mpu clock
output [12:0] addr;     // internal address bus
output     rd_n;        // read strobe
output     rst_n;       // internal bus reset
output     sela;       // select for unit a
output     seld;       // select for unit d
output     selm;       // select for memory
output     selr;       // select for rom
output     wr_n;       // write strobe
inout [ 7:0] data;     // bidirectional data bus

reg [12:0] addr_i;     // internal address bus
reg      rd_n_i;      // read strobe
reg      wr_n_i;      // write strobe
reg      rst_n;       // internal bus reset
reg      sela_i;     // select for unit a
reg      seld_i;     // select for unit d
reg      selm_i;     // select for memory
reg      selr_i;     // select for rom

reg [ 7:0] dataout;   // driver for data out
tri [7:0] data;      // 3-state net for data

//Intermediate signals are created to avoid
//a error from the specify block, which
//requires the destination to be of type net.
wire rd_n = rd_n_i;
wire wr_n = wr_n_i;
wire [12:0] addr = addr_i;
wire sela = sela_i;
wire seld = seld_i;
wire selm = selm_i;
wire selr = selr_i;

// top level signal dec section
reg busy;

assign data = ~wr_n_i ? dataout : 8'hz;

```

```

initial begin : bfm_init
    busy = 0;
    forever begin : service_rst
        @(negedge rst_n)
            addr_i = 0;
            rd_n_i = 1;
            sela_i = 0;
            seld_i = 0;
            selm_i = 0;
            selr_i = 0;
            wr_n_i = 1;
            dataout = 8'hz;
        end
    end
    ///////////////////////////////////////////////////////////////////
task bfm_rst;
begin
    while (busy) @(posedge clk);
    busy <= 1;

    rst_n <= 0;
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);

    rst_n <= 1;
    busy <= 0;
end
endtask
///////////////////////////////////////////////////////////////////
task memrd8;
input [14:0] laddr; // local address
input [ 7:0] ldata; // local expected data
input [ 3:0] ws;    // wait states

integer timeout_cnt_v;
integer rd_error_v;
integer i;
begin
    rd_error_v = 0;
    timeout_cnt_v = 0;

    while (busy) @(posedge clk);
    busy <= 1;
end

```

```

{selr_i,selm_i,seld_i,sela_i} =
    sel_decode(laddr[14:13],1);
addr_i <= laddr[12:0];

@(posedge clk);
rd_n_i <= 0;
for (i=1;i<=ws;i=i+1)
    @(posedge clk);

@(posedge clk);
for (i=0;i<=7;i=i+1) begin
    if ((ldata[i] != 1'bx) &&
        (ldata[i] != data[i]))
        rd_error_v = 1;
end // for i
if (rd_error_v)
    $display("Error: Expected %x, but Actual
%x\n",ldata,data);

sig_clean_up;
busy <= 0;

end
endtask // memrd8
////////////////////////////////////
task memwr8;
input [14:0] laddr; // local address
input [ 7:0] ldata; // local write data
input [ 3:0] ws; // wait states

integer timeout_cnt_v;
integer rd_error_v;
integer i;
begin
    timeout_cnt_v = 0;

    while (busy) @(posedge clk);
    busy <= 1;

    {selr_i,selm_i,seld_i,sela_i} =
        sel_decode(laddr[14:13],1);
    addr_i <= laddr[12:0];

    for (i=1;i<=ws;i=i+1)
        @(posedge clk);
    wr_n_i <= 0;

```

```

dataout <= ldata;

@(posedge clk);
sig_clean_up;
busy <= 0;

end
endtask // memwr8
////////////////////////////////////
task iord8;
input [14:0] laddr; // local address
input [ 7:0] ldata; // local expected data
input [ 3:0] ws; // wait states

integer timeout_cnt_v;
integer rd_error_v;
integer i;
begin
    timeout_cnt_v = 0;

    while (busy) @(posedge clk);
    busy <= 1;

    {selr_i,selm_i,seld_i,sela_i} =
        sel_decode(laddr[14:13],0);
    addr_i <= laddr[12:0];

    @(posedge clk);
    rd_n_i <= 0;
    for (i=0;i<=ws;i=i+1)
        @(posedge clk);

    @(posedge clk);
    for (i=0;i<=7;i=i+1) begin
        if ((ldata[i] != 1'bx) &&
            (ldata[i] != data[i]))
            rd_error_v = 1;
        end // for i
        if (rd_error_v)
            $display("Error: Expected %x, but Actual
%x\n",ldata,data);

        sig_clean_up;
        busy <= 0;

    end
end

```

```

endtask // memrd

task idle;
  input [31:0] cyc_cnt;

  integer i;
begin
  addr_i <= 16'h0;
  rd_n_i <= 1'b1;
  sela_i <= 1'b0;
  seld_i <= 1'b0;
  selm_i <= 1'b0;
  selr_i <= 1'b0;
  wr_n_i <= 1'b1;
  dataout <= 8'hz;
  for(i=0;i<=cyc_cnt;i=i+1)
    @(posedge clk);
end
endtask // idle

task sig_clean_up;
begin
  addr_i <= 16'h0;
  rd_n_i <= 1'b1;
  sela_i <= 1'b0;
  seld_i <= 1'b0;
  selm_i <= 1'b0;
  selr_i <= 1'b0;
  wr_n_i <= 1'b1;
  dataout <= 8'hz;
end
endtask // sig_clean_up

function [3:0] sel_decode;
  input [1:0] laddr;
  input      lmio_n;

  reg lselr, lselm, lseld, lsela;
begin
  lsela = (laddr==2'b00 && !lmio_n) ? 1 : 0;
  lseld = (laddr==2'b01 && !lmio_n) ? 1 : 0;
  lselm = (!laddr[1] && lmio_n) ? 1 : 0;
  lselr = (laddr[1] && lmio_n) ? 1 : 0;

  sel_decode = {lselr,lselm,lseld,lsela};
end

```

```

endfunction

// Setup the BFM timing parameters
specify
  // Specify the delays for the outputs
  specparam rd_rise   = 1:2:3;
  specparam rd_fall   = 1:2:3;
  specparam wr_rise   = 1:2:3;
  specparam wr_fall   = 1:2:3;
  specparam addr_rise = 3:4:5;
  specparam addr_fall = 3:4:5;
  specparam data_rise = 6:7:8;
  specparam data_fall = 6:7:8;
  specparam data_float = 6:7:8;
  specparam sela_rise = 5:6:7;
  specparam sela_fall = 5:6:7;
  specparam seld_rise = 5:6:7;
  specparam seld_fall = 5:6:7;
  specparam selm_rise = 5:6:7;
  specparam selm_fall = 5:6:7;
  specparam selr_rise = 5:6:7;
  specparam selr_fall = 5:6:7;

  (clk => rd_n) = (rd_rise, rd_fall);
  (clk => wr_n) = (wr_rise, wr_fall);
  (clk => addr) = (addr_rise, addr_fall);
  (clk => sela) = (sela_rise, sela_fall);
  (clk => seld) = (seld_rise, seld_fall);
  (clk => selm) = (selm_rise, selm_fall);
  (clk => selr) = (selr_rise, selr_fall);
  (clk => data) =
    (data_rise, data_fall, data_float);

  // Specify the timing requirements
  specparam data_setup = 2:3:4;
  specparam data_hold  = 0:1:2;

  // Check that the setup and hold requirements
  for the BFM are not violated.

  $setup(data, posedge clk, data_setup);
  $hold(posedge clk, data, data_hold);

endspecify
endmodule
// end of file

```

Figure 9. Task BFM Test Bench

```

`timescale 1ns/1ns

module ibfm_tb;

    reg    clk;
    wire [12:0] addr;
    tri [ 7:0] data;

    ibfm ib (
        // inputs
        .clk(clk),
        .rst_n(rst_n),
        // outputs
        .addr(addr),
        .rd_n(rd_n),
        .sela(sela),
        .seld(seld),
        .selm(selm),
        .selr(selr),
        .wr_n(wr_n),
        // inout
        .data(data)
    );

    mem1024x8 m1 (
        // inputs
        .addr(addr[9:0]),
        .clk(clk),
        .rd_n(rd_n),
        .selm(selm),
        .wr_n(wr_n),
        // inout
        .data(data)
    );

    always begin : clk_blk
        clk = 0;
        forever begin
            #10 clk = ~clk;
        end
    end

endmodule
// end of file

```

Figure 10. Test Code

```

`ifdef IBFM
`define BFM ibfm_tb.ib
`else
`define BFM ebfm_tb.eb
`endif

`define MEMLOW 15'h0000
`define ROMLOW 15'h8000
`define WS0 0
`define WS1 1
`define WS2 2
`define WS3 3

module bfm_tst1;

integer i,j,k;

wire clk;
assign clk = `BFM.clk;

initial begin : tst_blk

    `BFM.bfm_rst;

    for (j=0;j<=1023;j=j+1) begin
        for (i=0;i<=1023;i=i+1) begin

`BFM.memwr8(`MEMLOW+i,~i+j,`WS1);
            end
            `BFM.idle(5);
            @(posedge clk);
            for (i=1023;i>=0;i=i-1) begin
                `BFM.memrd8(`MEMLOW+i,~i+j,`WS0);
            end
        end // for j

    $finish;
end

endmodule
// end of file

```