A New Approach to Bus Functional Models

ASIC design engineers have traditionally used bus functional models (BFM) to verify ASIC interaction with defined bus protocols and other off-the-shelf chips. The BFM came into existence when FPGAs and ASICs started to interface to other chips for which a complete, full-functional model was too complex or not available. Engineers realized that the behavior of devices was less important and what really counts is what happens on the pins of a chip. A bus functional model is a model that provides a task or procedural interface to specify certain bus operations for a defined bus protocol. For microprocessors these transactions usually take the form of read and write operations on the bus. Bus functional models are easy to use and provide good performance. Unfortunately, BFM accuracy is only as good as the engineer who wrote it and the data book used to write it. The challenge facing new projects is where to get bus models and how to guarantee they are accurate. New methods for model creation are needed that can provide all the benefits of the BFM without the accuracy problems and the increasing model creation times. This article gives a brief history of modeling issues and presents a new way to create bus models using silicon based modeling methods.

Modeling Choices

In the past, engineers have used three basic types of models to do ASIC and board simulation as shown in Figure 1. The first type of model is the bus functional model. The BFM is a model that can generate different bus transactions for a given device, not limited to, but usually a microprocessor. Hardware engineers like the BFM because it is easy to drive from a testbench. Unfortunately, the BFM is created by a modeling engineer from reading the data book. This process always leads to accuracy issues. The second type of model is the RTL, or full-functional model. Full functional models are usually derived from the RTL or even gate level description of the chip. Accuracy is excellent since it is derived from the code used to produce the silicon, but performance is extremely slow. Sometimes these models are encrypted to protect the source code from users. It is not uncommon for such a model to require hours of simulation just for the reset sequence of a complex device like a networking processor. The third type of model is the instruction set simulator. Software engineers use the ISS to start executing code early in the project, before any prototypes are available. The ISS offers good performance, but makes no attempt to be accurate. Besides accuracy and performance questions, all three of these software models suffer from limited availability. Both the BFM and the ISS must be created by a modeling

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Performance</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFM</td>
<td>✗</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>RTL</td>
<td>✓</td>
<td>✗</td>
<td>?</td>
</tr>
<tr>
<td>ISS</td>
<td>✗</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>DeskPOD™</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

FIGURE 1. Modeling Alternatives
engineer and typically take anywhere from four months to more than one year to create. A few years ago there were many small modeling companies who were creating bus models and selling them to a handful of users. This made for a very nice small business with reasonable model creation times and good return on investment. Over the last few years all of these companies were faced with increasing chip complexity, too much pressure to deliver bug free models, and eroding prices. As a result most of these companies have found other ways to make money. The RTL model also suffers from availability because there are very few semiconductor companies who are willing to give out the source code or some encrypted version of the source code to more than a few key customers because of legal and support issues. Only a few lead, tier-one customers have access to such a model.

To compound the problem EDA vendors have created HW/SW co-verification tools using these modeling methods, specifically the BFM and ISS. The tremendous time-to-market benefits of HW/SW co-verification can be lost if too much time is spent debugging model problems. This has lead to greater acceptance of co-verification by software teams and less by hardware teams. The message from co-verification users is clear, software engineers do not care about accuracy, only performance is important. Hardware engineers are just the opposite, accuracy is paramount, speed is secondary. Only co-verification solutions that provide both fast and accurate models stand to be accepted by both hardware and software engineers.

**New BFM Solutions**

In addition to the general problem of increased bus model creation time, designs now involve technical hurdles that make bus model creation nearly impossible. Many projects include multiple bus masters, split address and data bus tenures, complex bus arbitration methods, and new protocols for out-of-order transaction completion. On top of all of these issues comes cache coherency. A symmetric multiprocessor (SMP) design is a challenge once limited to large server and workstation design. This type of design is now finding its way into embedded systems on projects like high performance networking equipment. All of the details of a multi-master write back cache protocol is beyond the scope of this article, but suffice it to say maintaining cache coherency using complex bus protocols is no trivial task. DeskPOD is a silicon based modeling system from Simpod, Inc. that connects to a network and can be used to solve these bus model issues. It's primary use is as a hardware modeler that provides full-functional, cycle-accurate models. It is also provides fast, accurate models used for HW/SW co-verification. The remainder of this article describes how to hardware modeling technology can be used to create bus functional models.

**New Technology**

Many engineers are familiar with hardware modeling techniques and products. The concept of using a chip as a simulation model has been around for many years. The basic concept is to communicate the device inputs from a logic simulator, drive the chip's pins, sample the outputs, and return the outputs to the simulator. This technology provides a way to get simulation models for those chips that have no good model. This basic hardware modeling tool can also be used as a bus functional model. Two technologies and some extra software can transform a full-functional hardware model into a bus functional model.

**Disengage Mode**

One of the great benefits of a bus model is it's performance. Since it only runs the specified bus transactions, all of the overhead of a full-functional hardware model is not there. Just to initialize a microprocessor cache, memory management unit (MMU), translation lookaside buffer (TLB), and setup a stack can take thousands of instruction fetches. One way to alleviate this problem is to change the synchronization between the hardware model and the simulator. If the hardware model is allowed to decouple from the simulator it is no longer bound by the simulation speed.
DeskPOD operates using two primary synchronization modes, **engage mode** and **disengage mode**. When operating in engage mode the hardware model is run in lock-step with the logic simulator. This is the most accurate mode because device pin values are exchanged every cycle. In engage mode all bus transactions are seen in the simulator making it a lower performance mode. Disengage mode allows the hardware model to execute cycles without interaction with the logic simulator. Disengage mode is a higher performance mode because model speed is not limited by the logic simulation performance and there is no network traffic being sent to and from the simulator. By allowing the hardware model to dynamically switch between the two synchronization modes only those bus transactions that are “interesting” are seen on the waveform and the “uninteresting” transactions are hidden from the simulation and run at very high speeds relative to the logic simulator. What is interesting or uninteresting can be decided by the user and defined by sets of pin values (conditions) to change synchronization modes. At the start of a project probably everything is interesting, but later and idle bus or fetching instructions from ROM is not so interesting. Disengage mode is the first technology used to transform a hardware model into a bus functional model.

**Local Memory Models**

The disengage mode by itself is helpful but most devices cannot run too many cycles without new inputs, even with the large, multi-level internal caches that exist on today’s chips. For example, to disengage on the uninteresting fetches from the ROM, the data for these ROM accesses must be kept local to the hardware model so it does not have to access the logic simulator to get it. By providing local memory models and a bus/memory controller these memory accesses can be serviced inside the hardware model, in the disengage mode. The combination of disengage mode and local memory models provide the technology to run software on a microprocessor without accessing the logic simulator except when specified.

**A Little More Software**

To finish off the bus model implementation using a hardware model a little more software is needed. First, a task interface to the bus model is needed so that it looks just like any other bus model. Some bus models use verilog tasks, other models use PLI tasks. Since this BFM is not a verilog model, a PLI task is a more appropriate choice.

Once the task interface is in place, a way to run the specified bus transactions on the microprocessor bus is needed. To accomplish this a small C program is run on the target microprocessor that will initialize the processor and process commands coming from the simulator. A loop can interpret commands coming from the testbench, carry them out on the bus, and provide the required results. The result is the diagram in Figure 2. The testbench is written in verilog, just like with software bus models. To create a bus transaction a set of PLI tasks is available that covers all of the bus protocol. The PLI program will send the parameters of the request directly over the network into a local memory. This is done using the standard C API for the hardware model. The small C program running on the processor will read and interpret the request from the local memory and execute the operation on the bus.
All of this processing is done in the disengage mode and none of the code fetches or accesses to the local memory are seen on the simulation waveform. When the bus transaction specified by the user actually appears on the device pins the synchronization mode is then switched to engage mode and this transaction will appear on the waveform. Immediately following the specified transaction, the hardware model will again switch to disengage mode and process the results of the transaction. The last remaining issue is how to synchronize the testbench. A few simple verilog registers are used to specify when the model is ready for commands, when a command completes, and when new memory results are ready.

**Example Bus Model**

Next, an example of how this technology works is provided. In this example the bus protocol for the Motorola MPC7400 series microprocessors is used. Some highlights of this device include:

- Separate on chip instruction and data caches
- Level 2 cache interface with external data SRAM
  - Hardware-enforced cache coherency (up to 5 state MERSI protocol)
- Separate address and data busses, each with its own arbitration and control
  - Options for pipelined transaction operation, split transaction operation, and enveloped transaction operation
- Data streaming for reads and writes
- Support for full out-of-order bus transactions
- Up to 7 outstanding bus transactions, six pending plus one data tenure in progress
- Data interventions for multiprocessor systems

Listing 1 illustrates a simple memory write/read verilog testbench on the for the bus model. It looks and feels just like a software bus model. It uses the PLI tasks to specify bus commands and registers such as `read_ready` to provide testbench synchronization. There are many other features and issues related to verifying a real design on the MPX bus. Beyond the basic read and write transactions there are many different transaction types as well as address only transactions. Also, things like setting the state of a cache line on processor A to a given state and then running a bus transaction on processor B and watching the bus interaction. All of these can be accomplished using hardware models with the technology and software described here. This

![Verilog Testbench](image)

**FIGURE 2. BFM Architecture**
basic introduction to the technology makes it clear that hardware models will play an increasingly important role in the future of ASIC and system design.

**Summary**

The choice of what kinds of models to use for design verification depends on the skill set and experience of the engineers involved. Many engineers prefer to write the tests in C and run them on the target microprocessor. This results in a set of system diagnostics that can then be run on the final product in the lab. Other engineers who are not familiar with C or assembly language and do not want to understand how to initialize a microprocessor tend to use bus functional models to verify system operation. Ease of use and good performance make bus models attractive for testbench development for ASIC and board simulation. As bus protocols get more and more complex new techniques for bus model creation and validation are required. One solution to this problem is to use a hardware modeler to provide the functionality and features of a bus model. Using a hardware model as a bus model provides the best accuracy, takes the least time to develop, and has many other uses including full-functional models and HW/SW co-verification.

**Listing 1:**

```cpp
/*
 * Simple memory test using word writes and reads.
 * Uses a data equal to address pattern and verifies the
 * read data matches what was written.
 */
task sp_ppc_mem_test_word;
   input [0:31] address;
   input [0:7]  length;
   reg [0:4]  tt;
   reg [0:3]  tsiz;
   reg [0:63] data;
   integer i;
   reg [0:31] ad;
   reg [0:31] result_data;
begin
   tt = `SN_WWF; // Write with Flush
   tsiz = 4'h4;  // 4 bytes
   status = 1'b0;
   ad = address;

   /*
    * Write data = address pattern for specified number of words.
    */
   for (i = 0; i < length; i = i + 1) begin
      data[0:31] = ad;
      $spbfm_ppc_write(ad,tt,tsiz,data,status);
      ad = ad + 4;
   end

   tt = `SN_SBR; // Single beat read
   tsiz = 4'h4;
```
tbst = 1'b1; // non-burst
status = 1'b0; // assume fail
ad = address;

/*
* Submit read commands to read back data.
*/
for (i = 0; i < length; i = i + 1) begin
  $spbfm_ppc_read(`SP_PATH,ad,tt,tsiz,tbst,status);
ad = ad + 4;
end

status = 1'b0;
result_data = 32'h0;
ad = address;

/*
* Retrieve read data.
*/
for (i = 0; i < length; i = i + 1) begin
  @(posedge system.cpu.read.ready);
  $spbfm_ppc_read_result(`SP_PATH,ad,result_data,status);
  $display("Result data %0h",result_data);
  if (result_data != ad) begin
    $display("Data compare error. exp %0h read %0h",ad,
      result_data[0:31]);
    $finish;
  end
  ad = ad + 4;
end

$display("Word test passed\n");
end
detask