

# A Programmable Bus Functional Model for Board-level VHDL Simulation

Patrick A. McCabe, Ronald W. Wilcox, Hugh J. Blair  
Honeywell Inc.  
Space Systems  
13350 U.S. Hwy. 19 North  
Clearwater, FL 34624

## Abstract

*The structure of a file-driven, programmable VHDL bus functional model (BFM) and its application for design verification within a simulation test bench is presented. The model emulates the behavior of a microprocessor bus interface to memory and I/O. An example implementation is shown for the Honeywell RH32 RISC microprocessor, although the generalized techniques can be applied to any microprocessor bus. The structure of the BFM is described in detail. A command language for the BFM is defined, including bus commands. The model provides the ability to perform board-level simulations with a high level of fidelity, without requiring a detailed model of the microprocessor being emulated. Additionally since the model is file driven, it supports stimulus file revisions and multiple versions without re-compilation of the VHDL testbench.*

## 1. Introduction

In order to verify an ASIC design, a system-level simulation should be performed. A bus functional model (BFM) can be used in the system simulation as a substitute for a detailed model of the processor in the system. In the case where the processor is a new development, the BFM can be developed early in the design cycle and used as the starting point in further system design. A BFM can also be created when a commercial processor is to be used in the system, and for which a VHDL model is unavailable. The BFM is inserted into a VHDL testbench in place of a detailed processor RTL model, for the purpose of debugging ASICs in a board-level simulation.

The BFM described in the paper provides a number of benefits. It can accurately model the processor-to-memory bus operations at a behavioral level. The flow of bus operations is controlled by an ASCII script file, which directs

the test sequence for the simulation. The BFM then produces the desired bus operations during simulation using the script file. Because the model uses a high level of abstraction, there is no learning curve associated with processor-specific ISA and development tool software. Use of a BFM provides faster simulation than using a detailed (RTL-level) processor model. If a problem is found in the script, a simple edit of the script file is all that is required - no recompilation is necessary. However, the BFM is not suitable for processor performance benchmarking or software debug purposes. The model also does not include the processor internal registers, ALU, pipelines, caching, instruction set architecture, or other internal structure of the microprocessor.

In addition to producing the signal waveforms corresponding to the desired bus operations, the model also provides the ability to automatically check data provided to the processor model via the bus interface, thus eliminating time-consuming results verification by inspection. Correct results indicate that the bus operation, and associated external devices (such as memory and I/O devices) participating in the operation, worked as expected, thus verifying the hardware design. Incorrect results are automatically detected, and error messages are reported as they occur via VHDL assertion statements to the simulation operator.

Commands in the script file are written in an interpreted control language created for the BFM. The commands may be of two types: configuration commands and bus functional commands. Configuration commands configure the BFM for a specific mode of operation, such as bus timeout period, and require no simulation time to execute. Bus commands define the main modes of operation that the bus can perform, such as read/write of memory or I/O.

The methodology for verifying correct simulated behavior of the BFM vs. the processor being modeled is also discussed. Additionally, issues related to using the BFM in place of a detailed processor RTL model are discussed.

## 2. Model Structure

The internal model structure of the BFM consists of a command parser, syntax checker for commands read from the script file, and an output stimulus state machine.

Commands are read from the script file via VHDL TEXTIO routines [1], and are loaded into a script array. At this point, syntax checking can be performed on the commands, and the user notified if an error is detected.

The output stimulus state machine creates the required signal waveforms for the bus operations specified in the script file. These are typically multi-cycle operations which emulate the behavior of the processor bus being modeled.

Figure 1 illustrates the generic structure of the bus functional model.

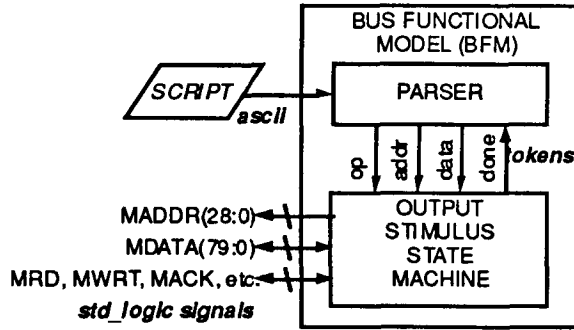


Figure 1: Bus Functional Model (BFM) Structure

### 2.1 Parser

The command parser's primary functions are to read commands from an ASCII text file and store them in a script array, and then to read this script array and pass signal values to the output stimulus state machine. The script array is an array of records which holds the sequence of commands and bus signal values specified by lines in the command file. The command parser consists of two loops: a script array writer and a script array reader.

The script array writer in the BFM is a loop which reads the entire text command file line-by-line, checks for syntax errors, performs type conversions on the input strings to enumerated or nine-state values, and stores these converted signal values in the script array. Strings in the command field are converted to an equivalent enumerated-type command. Reading, type conversion, and storage of the remaining fields of the command line is dependent upon the command type. Command formats for the BFM

are described in Section 3. Comment lines are also permitted in the script file

This script array writer loop completes while the BFM is held in reset. Reading the entire command file at the beginning of the simulation run is performed to complete error checking of the input text as quickly as possible. Since an error in reading this command may cause the simulation to be aborted or to perform an incorrect sequence of operations, early detection and reporting of syntax errors in the command script is important when running simulations which will not complete for hours. VHDL assertion statements are used to report syntax errors via the simulator's debug window.

The script array reader in the BFM reads back the commands and bus signal values from the stored records and assigns them to internal BFM signals. Commands are of two varieties: configuration commands and bus commands. Bus commands will be passed to the output stimulus state machine for execution during simulation. Configuration commands cause values to be

assigned to BFM signals only at the time they are read. Some of these signals are assigned to output ports which represent processor output discretely. Others are assigned to internal signals and are used by the output stimulus state machine in modifying modes of subsequent bus cycles, but do not themselves cause waveforms to be generated on the processor-to-memory bus (the RH32 MBUS).

Maximum flexibility in generating bus waveforms is achieved by reading script array records in a loop until a bus command is encountered. This enables BFM mode changes by way of configuration commands to be accomplished without requiring processor clock cycles. Consequently, any number of configuration commands may be read and executed in zero simulation time.

## 2.2 Output Stimulus State Machine

The output stimulus state machine (OSSM) executes the interpreted commands from the script array and generates output signal waveforms on the MBUS corresponding to the bus operations specified in the script file. The signal waveforms represent those generated by the RH32 processor RTL model on the MBUS when performing memory read and write cycles as well as memory-mapped I/O cycles. The MBUS consists of a 29-bit address with parity, two 32-bit bidirectional data busses with parity for memory-mapped I/O and EDAC for memory operations, plus several request/handshake control lines.

The OSSM begins execution of bus commands at the first processor clock after the BFM reset signal is de-asserted. The OSSM sequences through the following states while emulating a typical bus command: 1) issue bus request (assert true) and wait for grant, 2) issue memory request (enable bus drivers and assert true) and wait for memory acknowledge, 3) remove bus and memory requests (assert false), and 4) tri-state the bus (disable bus drivers). When execution of the bus command has been completed, the OSSM signals the parser process that it is ready to execute a new command.

Several other processes run concurrently with the OSSM to provide additional functions such as a four-port bus arbiter, a bus watchdog timer, a

bus idle-time counter, and a bus data comparator. The arbiter and the bus watchdog timer emulate the functionality of additional logic within the processor. The watchdog timer also assists in testing by reporting a timeout condition via the simulator's debug window. The bus idle-time (no operation) counter allows the script file writer to specify periods of bus inactivity, such as when the processor is fetching instructions from its internal cache rather than from main memory. The bus data comparator, when enabled, reads data (along with parity or EDAC bits) returned via the MBUS and compares it to an expected data value specified as part of the read command in the script file. If a mismatch is detected, it is reported via VHDL assertion statements, and the user has the option to halt the simulation using this assertion. This mechanism permits the BFM to automatically check results of the bus operation, thus greatly reducing the amount of time spent on the task of verifying results by visual inspection of the simulation display.

## 2.3 Support Package

The BFM also makes use of a package which provides definition of the enumerated command type, as well as several special-purpose functions for frequently used logic such as generation and checking of the bus parity and EDAC. It also includes several type conversion functions used by the command parser and the output stimulus state machine.

## 3. Control Language Definition

For a given simulation run, operation of the BFM is controlled by the sequence of commands specified in the script file. A number of representative commands are described below, but is not a complete listing of the commands currently implemented in the model.

### 3.1 Configuration Commands

Configuration commands are used to place values on signals which the user may wish to change from one simulation run to another, or to change during the course of a single simulation run. These commands make it possible to verify operation of a device being tested with the processor model in a variety of operational modes without having to edit and recompile a VHDL testbench. Some example configuration

commands are shown in Table 1.

Table 1: BFM Configuration Commands

COMMAND	FORMAT	DESCRIPTION
Set Read Modify Write	RMW	The RMW command causes the BFM to generate a read-modify-write signal on the MBUS, denoting an atomic read/write sequence for memory. Timing of the signal is determined by read and write commands which follow it in the script file.
Set and Reset Check Discrete	SCK RCK	These commands toggle a processor output discrete, which specifies operation from either RAM or startup ROM.
Arbiter Hold Mode	AHM	AHM places the model's bus arbiter into its "hold bus" mode. This affects the BFM's bus timing by reducing time delay between bus cycles/operations.
Wait For Resume	WFR	This command is useful in synchronizing BFM operations to external events. WFR directs the BFM to suspend execution of script commands until a signal is received via the "resume" port of the BFM. The resume signal may be used to represent, for example, an interrupt which signifies completion of a DMA data transfer by the UUT.

### 3.2 Bus Functional Commands

Bus commands are those commands which cause the BFM to generate signal waveforms in accordance with the timing diagrams for the

processor-to-memory bus of the modeled processor. For purposes of command definition, the two 32-bit data busses of the MBUS are referred to as the lower (or even) bus and the upper (or odd) bus. Some example bus commands are shown in Table 2.

Table 2: BFM Bus Functional Commands

COMMAND	FORMAT	DESCRIPTION
Read Memory, Both	RMB address odd_data even_data	RMB causes the model to generate a read memory bus cycle to the specified address and compare the data actually read from the odd and even data busses to fields "odd_data" and "even_data".
Write Memory, Upper	WMU address odd_data	WMU causes the model to generate a write memory bus cycle to the specified address and drive the data specified in the "odd_data" field onto the odd data bus.
Write Memory, Lower	WML address even_data	WML causes the model to generate a write memory bus cycle to the specified address and drive the data specified in the "even_data" field onto the even data bus.
Programmed Input, Lower	PIL address even_data	PIL causes the model to generate a programmed input cycle in I/O address space and compare the data actually read from the even data bus to field "even_data".
Programmed Output, Lower	POL address even_data	POL causes the model to generate a programmed output cycle in I/O address space and drive the data pattern specified in the "even_data" field onto the even bus.

### 4. Example Implementation: RH32 RISC microprocessor

The Honeywell RH32 processor is a 3-chip type radiation-hardened 32-bit RISC processor with floating-point coprocessor and up to 8 Kbytes

each of instruction and data caches, and is capable of operating at a 25-MHz clock rate [2]. The RH32 was developed under contract from the Rome Laboratory, Rome, NY<sup>1</sup>. RH32 is intended for spaceborne processing applications, including mission data processing and spacecraft control processing. The RH32 CPU is a reduced

instruction set integer processor with a 5-stage pipeline, and is capable of executing most instructions in one clock cycle, therefore, the RH32 has a maximum performance of 25 MIPS at a 25-MHz clock rate. The floating-point processor (FPP) is an IEEE-754 compliant coprocessor, and handles all floating point operations. The 8-Kbyte cache memory consists of a cache controller and on-chip data and tag rams, and operates as a write-through cache. The same cache chip type is used to implement the separate instruction and data caches. Up to four caches of each type may be used in an expanded cache system.

The RH32 caches interface to the rest of the system (memory and I/O) by way of a memory bus (MBUS), consisting of 64-bits (two 32-bit words) of data plus 14-bits of error detect detection and correction (EDAC), a 29-bit double-word oriented address bus, plus miscellaneous control signals. Both instruction and data caches are supported, and multiple cache chips of each type may be used in an expanded cache system. Basic MBUS operations originated by the caches consist of 32- or 64-bit memory operations with EDAC, and 32- or 64-bit I/O operations with parity. The RH32 was synthesized from several dozen VHDL RTL models.

A VHDL-based bus functional model (BFM) was created which emulates the MBUS operations of the RH32 processor. The BFM was used in

board-level system simulations to verify ASIC interfaces to the MBUS, ASIC internal functionality, and system-level interactions among multiple ASIC's. [3]

Since the RH32 MBUS is the primary interface to the system, the BFM development concentrated on the emulation of MBUS operations. Internal implementation details of the RH32 chipset that do not affect the MBUS behavior were not incorporated into the BFM.

The primary MBUS interface logic is located in the RH32 cache chip, and consists of a four-port arbiter, MBUS address/data/control signals, a watchdog timer, and parity/EDAC generation and checking logic. The BFM instantiates the same four-port arbiter as used in the RH32 VHDL RTL, but the remaining functions were implemented using behavioral VHDL.

An example of how the BFM is instantiated in the system simulation is shown in Figure 2. Here, the BFM is used to drive data and control signals into the ASIC under test. Different script files are created and run to test different functional features of the ASIC. Additionally, the BFM can be used to set up conditions within the ASIC which cause it to initiate interactions with other ASIC RTL models or behavioral models within the testbench.

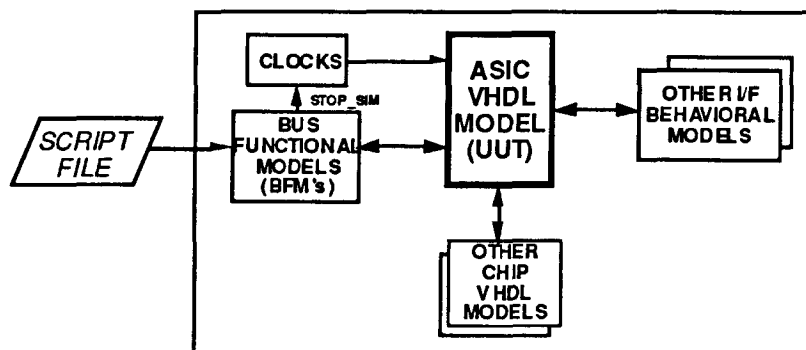


Figure 2: BFM in Context of a System Simulation

## 5. Comparison with RTL Model

Several characteristics were used to evaluate the effectiveness of the BFM approach to modeling compared to the RTL model. These

included wall time versus simulation time, the total number of simulator events for a given run, functional equivalence (fidelity) between the two models, and stimulus control and flexibility.

An improvement (decrease) in simulation time

is highly desired. The speed of the simulation relates directly to the amount and degree of simulation performed in a design cycle, which is typically limited due to schedule considerations such as time to market. The BFM and the RTL model it emulates also must exhibit equivalent behavior at the bus interface in order for testing with the BFM to be meaningful. The easy creation of stimulus files used to direct the simulation, and simplified creation of test scenarios, increase user productivity.

To verify that the BFM accurately modeled the RH32 MBUS interface, manual inspection of the timing waveforms was first performed. Later, a semi-automated procedure was developed in which signal data (address, data, and control lines) were captured from the simulation testbench using the RH32 RTL model running real software, and then converted into BFM commands by a shell script. The captured data files were read and reformatted into the BFM command syntax by decoding the text file fields corresponding to the control lines. The resulting file contained the commands that execute the exact sequence of bus operations through the BFM model. The converted BFM commands were then executed by the BFM, and the results compared with the RTL model simulation run.

The BFM, including its support package, consists of 2,500 lines of VHDL code, compared to 75,000 lines of VHDL for the RTL description of the RH32 chips. Clearly, the BFM source code is easier to maintain and debug than the VHDL RTL models. Additionally, the small size of the BFM allowed new features to be incorporated quickly at customer request. These features included the ability to force parity and EDAC errors on the bus interface in order to test the response of an ASIC design to error conditions.

## 6. Conclusions

There are several significant advantages to utilizing a BFM model over a complete RTL model.

If a VHDL RTL model is not available, then a BFM model can be easily created to model the required interface. If the VHDL RTL is available, the simulation time can be significantly reduced by using the BFM.

The BFM does not require processor-specific software development tools to develop simulation test vectors. The BFM command language is not overly expansive as it only pertains to bus transfer operations, and is easily understood by the user. The flexibility of the BFM command language allows a great deal of freedom in building bus operation scenarios that would be difficult, if not impossible, to create with software and the VHDL RTL model, thereby enabling more complete testing of the ASIC model.

The BFM also supports continual development of systems and fault simulation over the life cycle of the product. Lastly, the user has the ability to create multiple test scenarios without recompiling VHDL.

These features of the BFM increase productivity more than enough to justify the development time of a bus functional model for emulation of highly complex chips or systems for which the designer is building an interface. Use of BFM models following the structure described in this paper have been developed at Honeywell for several years, and have proven their usefulness on many different programs. Development and use of BFM's as an aid to system-level simulation have become an integral part of our ASIC design process.

## 7. Future Work

The programmable nature of the BFM is one of its best features. The authors are in the process of augmenting this feature with looping constructs in order to avoid long stretches of in-line code in the script file, which would be used to emulate repetitive bus operations.

Looping constructs being developed include FOR and WHILE loops, and a GOTO, which would require the addition of an optional label field in the script file.

In addition, an arithmetic/logical expression evaluation routine is being developed such that conditional branches could be added to the BFM command language. Conditional branches would allow a jump or loop-exit based on the result of an arithmetic or logical operation performed on data returned to the BFM from the ASIC under test. For example, a branch may be executed only if the returned data is equal to a specified value.

Such constructs when added to the BFM would allow more intricate test scenarios to be developed in order to enhance the testing performed in the testbench, while still allowing the script file to remain short, succinct and easy to read.

## 8. Footnotes

- 1 Contract No. F30602-88-C-0060

## 9. References

- [1] IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronic Engineers, Inc., New York, 1988.
- [2] Honeywell Inc., RH32 Processor User's Manual, Clearwater, FL, 1993
- [3] P. A. McCabe, "VHDL-based System Simulation and Performance Measurement", *VHDL International Users Forum*, Spring 1994 Conference, pp. 48-57, 1994.