Hardware/Software Co-Simulation in a VHDL-based Test Bench Approach

Matthias Bauer, Wolfgang Ecker
Siemens AG, Corporate Technology, ZT ME 5
D-81730 Munich
E-Mail: {Matthias.Bauer, Wolfgang.Ecker}@mchp.siemens.de

Abstract

Novel test bench techniques are required to cope with a functional test complexity which is predicted to grow much more strongly than design complexity. Our test bench approach attacks this complexity by using a strong hierarchical architecture, application domain-independent synchronization, reusable modules, and easy incremental extendability based on table-driven techniques. In addition, the integration of VHDL/C co-simulation under the control of the test bench makes it possible to use the hardware model for software testing and vice versa and thus enables extreme reductions in test bench coding. The efficiency of our test bench has already been demonstrated in several industrial projects, among them a four-ASIC ATM board with one embedded core and one external micro controller.

1 Introduction

Test bench complexity is going to be the dominating factor in future ASIC and digital system design. In this context, Nortel noted at VIUF'96 Spring conference in the panel "VHDL in Use: Experience from Telecom and Networking" that the size of test bench code increases much more than the size of RT code. A similar conclusion was reached at the last DAC'96 in the panel with the embedded tutorial "Verification of Electronic Systems". Here design complexity was related to Moore's Law, while test vector complexity was compared with Murphy's Law. Synopsys argued at the EURO-DAC'96 panel "What do Tool Vendors think" along the same lines and pointed out that design complexity increases 4x every year whereas functional test complexity goes up by 100x over the same period.

This dramatic increase in test bench complexity implies two major problems: Test generation and evaluation, and test execution.

The second problem is tackled by simulation speed improvement using e.g. cycle-based simulators, parallel simulation, simulation hardware acceleration, or emulation. Furthermore, test vector reduction is applied by using e.g. metrics to recognize how many tests are actually required. Formal methods could also help in functional tests. However, they are not able to deal with current levels of system complexity and can be applied to particular problems only.

To attack the first problem, we developed a hierarchical, flexible, and extendable test bench approach for regression tests with generators as well as analyzers running under control of a synchronizer. To further reduce test outlay we integrated VHDL/C co-simulation capabilities into our test bench. In this way we were able to use the unit under test as a hardware model for the software test. Moreover, software acts as a generator and analyzer for the unit under test. The key point is the selective activation of software units clustered in subroutines in conjunction with test streams under control of a test synchronizer.

The paper is organized as follows: First related work, especially test bench methods and hardware/software co-simulation capabilities are discussed. Then the basic principles of the test bench are presented, followed by a detailed description of the test bench approach and the VHDL/C co-simulation integration. Afterwards an application example is discussed. A look at future work concludes the paper.

2 Related Work

2.1 Test benches

A domain-independent method for pure HDL test benches based on building blocks was presented in [Sch95]. Applying stimuli from a file in a wave format [WAV] has the disadvantage that bit values only can be specified and applied to the unit under test. A domain-specific test bench for DSP models was shown in [Arm94].

The sequential nature of all of these approaches reduces their applicability. Systems requiring a concurrent and only partially synchronized data stream as ATM traffic cannot be tested suitably in this way.

2.2 Co-Simulation

Two major approaches exist for co-simulating hardware and software [Bar96]: Simulating the final machine code on a processor model or compiling the software for a computer and linking the executable to a bus functional model of the processor, which is simulated in conjunction with the hardware component.

The first approach distinguishes in the kind of processor model: Software model or hardware model. Different detailed software models are used ranking from an instruction set model to a cycle accurate model. Sometimes even propagation delay is considered at the interface. An accurate timing can be achieved at the expense of simulation effort. New approaches use the compilation of an intermediate code for simulation speed-up without reduction of accuracy [Ziv96].

The second approach differentiates in the used coupling method between software and simulator [Bec92, Rom96, Row94, Sch96, Sil95]. The benefits of this approach is the nearly real time execution of the software, however, under

"Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and /or a fee."

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

loss of timing accuracy. An approach to decrease this disadvantage by back-annotation of software runtime was presented in [Soi95].

One common disadvantage of all these approaches is that software runs completely independently of the test case, similar to a reactive environment model. Thus the separate activation of software units for testing and debugging cannot be performed. In addition, detailed and test scenario-dependent stimuli activation by software execution is impossible.

3 Our Approach

The basic idea of our test bench is a hierarchical structure of all our tests, which is reflected in the structure of the related VHDL code. In detail, the interfaces of the unit under test are classified. For each classification a set of operations is defined which applies a sequence of stimuli to the unit under test. These operations again may be clustered in several levels. Finally a set of high-level operations which we call *commands* is defined. The structure of this hierarchy is directly reflected in its implementation. VHDL design units, which we call in this case *applications*, relate to an interface classification, and procedures to operations or commands.

The tests are mostly parameterized either statically by using generics and files, or dynamically by parameterizing an operation. Combinations of both techniques are also supported e.g. by a file name associated as a parameter with an operation. The file identified by the parameter contains detailed information about a sequence of values or operations.

A completely application-independent *master synchronizer* controls and synchronizes the execution of operations of different applications. It reads a *control file* which contains the top-level test in ASCII form. Fast turnaround and VHDL-independent test specification, which is especially appreciated by software designers, is achieved in this way.

Besides that, this approach shows a high re-use potential due to the fact that our test bench can easily be extended by instantiating a new application and broadcasting the new application to the master synchronizer via tables. Furthermore, the possibility of multiple usage of applications in one test bench increases the re-use potential of this approach.

Our approach also tackles the problem of test evaluation. Hierarchical and parameterizable analyzers can be included in the test bench and be synchronized by the master synchronizer as well. In addition to detailed error reports written by each application, the error status is collected by the master synchronizer and logged globally. A final error summary allows for easy go/no go analysis.

VHDL/C co-simulation capabilities are integrated as a mixture of analyzers and generators. The test generation with regard to hardware is established by a bus functional model of the simulated micro processor supporting basic read and write operations. These operations can be activated by the master synchronizer directly, by additional linkable VHDL code called V-ware, or by a C-code tool independently connected to the VHDL environment via pipes and TCP/IP. Finally, the master synchronizer can activate selected procedures of both, V-ware and C-software.

4 Implementation Details

4.1 Test bench structure

The overall test bench structure is shown in Figure 1. It is fully implemented in VHDL using 15K LoC and composed of the *master synchronizer* and *applications*. The master synchronizer reads a control file and writes a report file via VHDL text I/O. The application stimulates the unit under test and gets the responses.

4.2 Master synchronizer

The master synchronizer, the heart of the test bench, consists of an initialization routine, an interpreter loop, and command queues.

The initialization routine of the master synchronizer builds up a linked structure over all applications and their commands, which are used in the control file. This action is performed on a table-driven basis to allow for easy extension. Overthat, the use of generic parameters for the table allows to pre-compile the master synchronizer. Based on this information, the control file is subjected to a syntax check. Finally the interpreter loop is started.

In the control file, user defined commands are allowed which will be executed by the applications and built-in commands. User-defined commands for an application are defined via tables in a particular package which is only visible for the master synchronizer and the application. Built-in commands are hard-coded in the interpreter. They consist of constant definitions, an include mechanism, loops, conditional statements, report comments, and synchronization commands.

Each executed user-defined command is reported to a file with start time, end time, and status, which is returned by the application. If the status is an error, a string with a description is also passed. At the bottom of the report file the number of errors are indicated.

All user-defined commands between two synchronization commands are executed concurrently, if the commands are intended to be executed by different applications. If more than one command for one application occurs in the command file between two synchronization operations, then these commands are executed sequentially.

To examine this behavior, the interpreter loop of the master synchronizer sends all user-defined commands to the command queues. For each application one command queue for sending the operation and one for receiving the status of each operation is instantiated. The communication between the master synchronizer and the command queues is a dynamic integer communication (see 4.3) with 1xN topology.

If a synchronization operation is read by the interpreter, then the *synchro* signal is set to start. Each command queue now sends all collected user defined commands to its application. A new command is sent only if the last command was acknowledged by the application. The command queue sets itself a ready signal if all commands have been sent to the applications and all acknowledgment messages have been received.

If all commands have been processed, which is shown by the setting of the synchro signals of all queues, the master synchronizer requests all acknowledgment information from the command queues and writes it to the report file. The interpreter loop continues afterwards until a new synchronization or the end command has been reached.

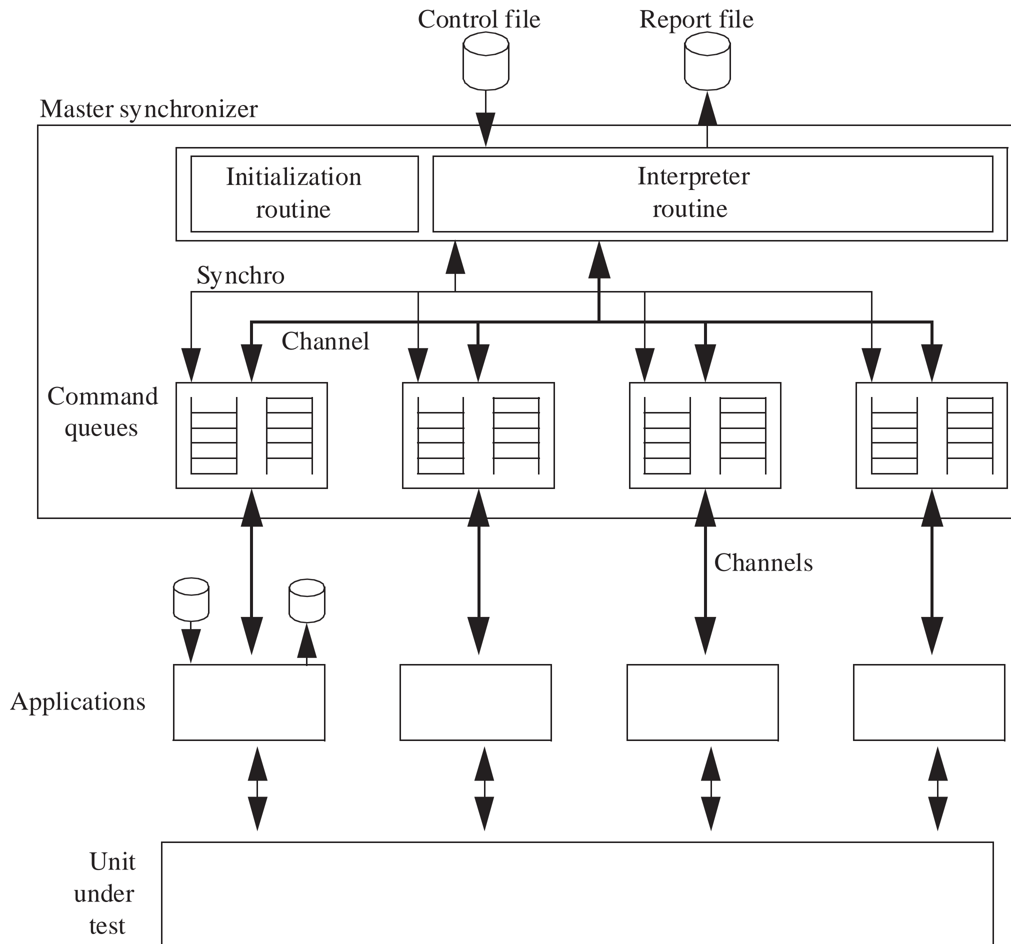

Figure 1: Test bench structure

4.3 Communication and synchronization

The communication between the master synchronizer and the applications is established by a dynamic integer handshake protocol, which is a delta cycle-based transmission of variable-length integer vectors [BaEc93]. A second communication layer for transmission of other VHDL types, such as strings or enumeration types, is based on this integer communication. It is required due to the lack of polymorphism or at least variant records in VHDL. A third layer is responsible for operation and parameter transmission. All subroutines implementing the second layer are collected in one package. But subroutines of the third layer are assembled in packages specific to each application. Each message requesting the execution of a command must be acknowledged to guarantee the causality of the individual test actions.

4.4 Applications

Applications are the direct interface to the unit under test. Examples of applications are a generator, an analyzer, or a mixture of both. For the implementation two application types are imaginable: *Single-sensibility functionality* and *multiple-sensibility functionality*.

Single-sensibility functionality means that the application process is blocked until a command is received. This command is then executed. Afterwards the acknowledgment is sent back to the master synchronizer. Alternatively the acknowledgment can be given if the command is not finished or only partially executed. Finally the process is blocked again. Multiple-sensibility functionality allows also sensibility with regard to other signals, i.e. its communication is non-blocking. If a message is present on the channel, the same behavior as in the case of single-sensibility functionality is performed. In the other case, if present, the functionality for the event on one of the other signals is executed.

The integrity of master synchronizer and applications is established by using one single table for the definition of the commands. The parsing, packing, and unpacking of commands and parameters is performed based on this table.

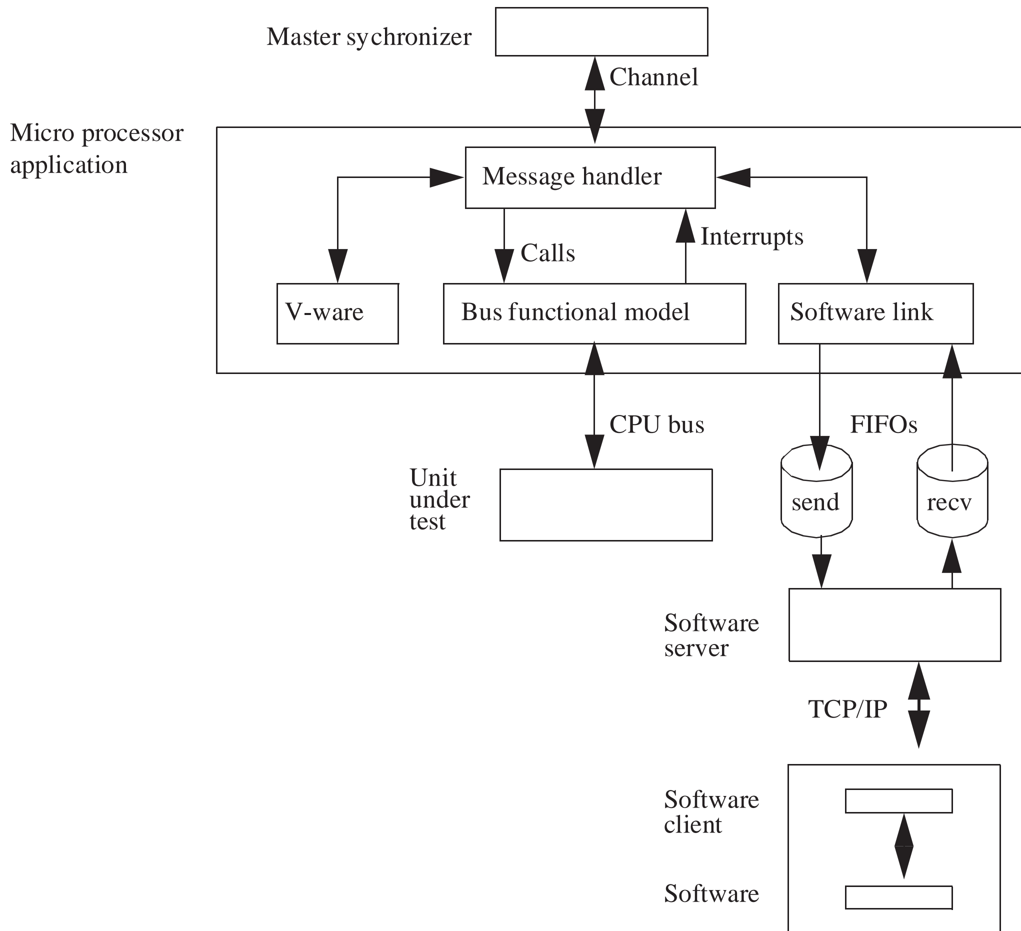

Figure 2: Hardware/software co-simulation environment

5 Co-Simulation

As already mentioned, hardware/software co-simulation is motivated by the fact that hardware can be used to test and debug software routines or the entire software. Conversely, the software can be used to test the hardware. The result of this knowledge is an enormous dispense. The validation of the software can be started after completion of the behavioral model of the unit under test. During the development of the RT model of the unit under test, the software can use this model, and during the design of the software the hardware can use the software for testing.

To fully support testing and debugging of both hardware and software, we defined a processor application for our test bench environment as shown in Figure 2. This application consists of a message handler, a bus functional model, V-ware and a software link.

5.1 Message Handling

The central feature of the processor application is a *message manager*, which receives messages from either the master synchronizer or the software. This message is decoded and dependent on the execution on certain actions:

- If the message directly requests a micro processor command, then the corresponding procedure of the bus functional model is called and an acknowledgment is sent back to the source of the message, either the master synchronizer or the software.
- If the message requests software actions, then the message is passed on to the software via a software link.
- If the message is an acknowledgment of a software command, then the acknowledgment is forwarded to the master synchronizer.

5.2 Bus Functional Model

Speaking generally, any kind of micro processor can be used in a system. For our co-simulation purpose a *bus functional model* is sufficient.

Commands for the bus functional model are called from the master synchronizer. These commands are defined in a table as for all applications. The basic commands are the *read compare* and *write* commands. For defining complex composite commands the *nop* operation is also necessary. Complex composite commands are for example read, read in sequence, write in sequence or polling commands. The

implementation is performed in an additional package. Only basic commands directly reflect the CPU bus. Other operations are mapped on these basic commands.

The micro processor application also supports the option of executing macros. Macros are a predefined number of commands which are implemented in VHDL. If the micro processor component receives such a macro command from the master synchronizer, then this command is forwarded to a V-ware component. The V-ware component decides which macro procedure has to be executed. The macro definition must also be implemented in the command definition package. For all commands that can be used in the macro a VHDL-calling procedure is included in the command definition package, so that each command which can be invoked from the control file can also be invoked within VHDL.

Interrupt and DMA handling is also included in the co-simulation application. For this feature primarily the V-ware component is used. If an interrupt occurs, an exception handling procedure is called, which sends an exception request to the V-ware component. In the V-ware the interrupt and DMA routine can be easily adapted to the necessary requirements. For both interrupt and DMA handling a counter is available to indicate the number of exceptions detected. These counters can be modified in the interrupt or DMA routine. To read the value of the counter a command is defined in the command definition package which can be used only in the control file.

5.3 Software Handling

To communicate from VHDL with software two approaches are possible: first *the VHDL-tool-dependent approach* based on a non-portable VHDL-C interface and second a *tool-independent text I/O approach*.

The VHDL-tool-dependent approach has the disadvantage that for each VHDL simulation tool a complete new test bench would be necessary, because each vendor has a different VHDL-C interface for its tool. This is completely at variance with our goal of reducing code outlay for test benches by re-use.

The text I/O approach doesn't have this disadvantage. For this communication method between VHDL and C all data is transmitted in ASCII notation. Two FIFOs (named pipes) are necessary to send and receive data from C to VHDL and vice versa. To receive data in VHDL, a file is opened and not closed until simulation is completed. Furthermore, this file is only read if it is not empty. For sending data from VHDL to C, a second file is opened and closed for each operation, because this is the only method in VHDL to flush a buffer.

The software should be executed on the same host as the VHDL tool or on any remote host. Thus, to build up a communication channel between the software and the hardware an interface module for the software was designed called software client. Its function is to communicate via UNIX sockets with the test bench and the processor model and to take activation requests for subprograms from the hardware or test bench controller and to pass read and write operation requests from the software on to the bus functional model.

Direct communication with the hardware, however, is not possible, because VHDL is not able to communicate via UNIX sockets. So as an intermediate device a software server is defined, which controls the FIFO communication to the VHDL-tool and the UNIX-socket communication with the software client.

The communication between test bench, software and micro processor is established as follows: If a software command, predefined in the command definition package, occurs in the control file, it is passed via VHDL communication channels to the micro processor application, converted to ASCII notation, and written to the send FIFO pipe. The send FIFO pipe is read from the software server and the data is moved from this pipe to the TCP/IP link. In this step the data is also passed as characters. The received data is analyzed by the software client which calls the corresponding software routines, among them a routine including the entire software. Thus, we can run the software as a pure environment model but also activate parts of it.

Each software command creates a sequence of micro processor commands resulting from I/O operations in the co-simulation. Each micro processor command is transmitted via TCP/IP and FIFO pipe back to the co-simulation application and then to the micro processor model. Here it is decoded, and the related VHDL procedure is called in order to execute one or more of the basic read, write or nop operations. After finalization an acknowledgment is passed back to the software via the send FIFO and socket. Finally an acknowledgment is sent from the software via the processor model to the master synchronizer.

Exception handling in the software is also possible. Instead of sending the interrupt request to the V-ware as predefined, the request can be sent user-defined to the software client, which calls C routines. It interacts with the micro processor model in the sane way as that of any other C procedure.

To sum up, the basic idea is that the VHDL test bench is master for the co-simulation and the software runs under the control of the VHDL simulator. The software is executed in zero delay, because the simulation tool is waiting for a micro processor command. Only the micro processor commands consume simulation time. Important is the conservation of the causality.

6 Usage in Designs

The test bench was, as already mentioned, used in several ASIC projects. One of them, a board design out of the ATM domain consists of four ASICs including one embedded core and one external micro controller. It shall now be used to show the application of our concept:

Each of the ASICs was first modeled as a behavior model and tested separately using the test bench approach described above. Applications were a clock generator, value sequencers adjusting the ASIC state, RAMs, the VHDL/C co-simulation unit and communication traffic generators as well as analyzers. ASIC-specific software was developed and tested using the test bench.

Afterwards the ASICs were virtually integrated in a board and included in a test bench as shown in Figure 3. Board-specific software executing the ASIC-specific software was developed and tested using this model.

In this way, the correctness of the specification could be validated at an early stage. First pieces of software were developed and tested. Test effort was reduced by the fact that no test frame for software test was required and huge sections of software could be used to test hardware.

Subsequently the ASICs were modeled at RT level and synthesized. Simultaneously software development continued and test cases were extended. The RT models were afterwards integrated in the test frame and are currently being tested.

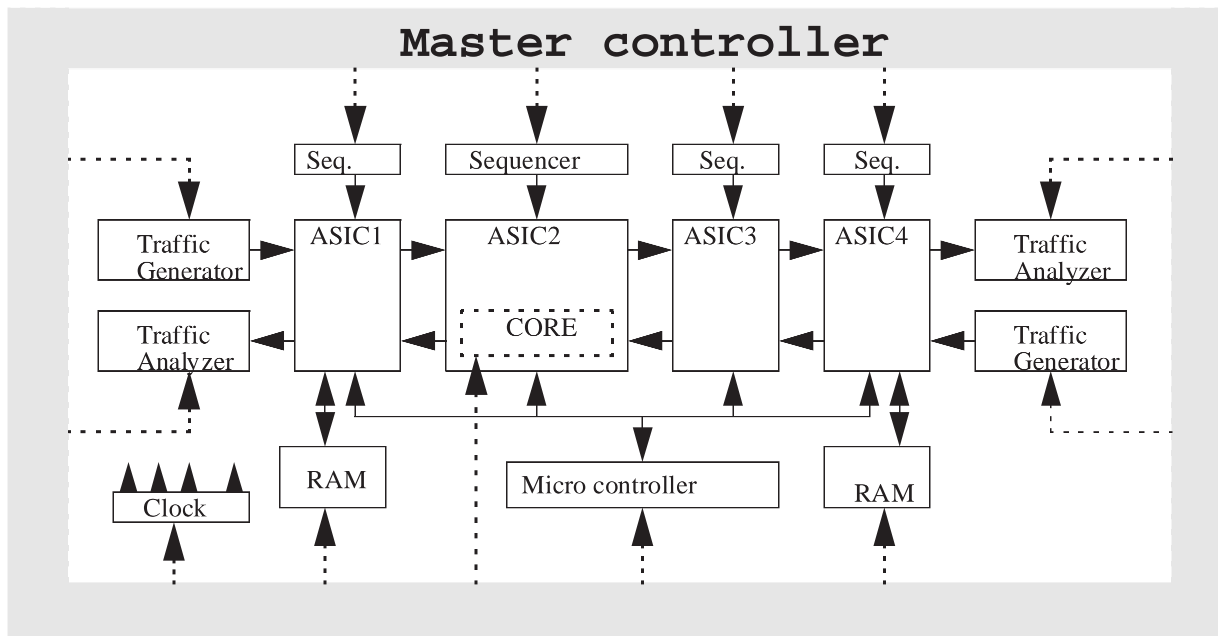

Figure 3: A test bench example

We expect additional benefits from our approach due to the fact that low software layers and bring-up software were tested before being run on the first hardware prototype.

7 Conclusion and Outlook

We presented a highly flexible test bench approach with VHDL/C co-simulation capabilities. Its main goal is the reduction of coding effort for test bench development by re-using and multiply using the code. This was achieved by developing an application-independent and parameterizable master synchronizer, a set of parameterized applications, and, most important, the inclusion of a VHDL/C interface under control of the test bench.

Future work will focus on simplifying applications improving timing accuracy and simulation speed up. We plan to integrate software runtime information into our interface in order to increase accuracy. In addition, partial modeling of the applications as RT models to allow for faster simulation is under discussion.

8 Bibliography

[Arm94] J.R. Armstrong, G. Frank, S. Hrishikesh, P. Gowri-sankaran, Z. Xu, "Test Bench development for RASSP DSP Models". First Annual RASSP Conference, Washington DC, August 15-18, 1994.

[BaEc93] M. Bauer, W. Ecker, "Communication Mechanisms for Specification and Design of Hardware Starting at System Level". VHDL-Forum for CAD in EUROPE, Innsbruck, Austria, March 14-17, 1993.

[Bar96] J.K. Bartolomew, G.J. Buza, "Connecting Hardware and Software Design Environments: Realities in Bridging the Gap". VIUF'96, Santa Clara, USA, February 28 - March 2, 1996.

[Bec92] D. Becker, R.K. Singh, S.G. Tell, "An Engineering Environment for Hardware/Software Co-Simulation". DAC'92, Anaheim, USA, June 8-12, 1992.

[Rom96] K.V. Rompaey, D. Verkest, I. Bolsens, H. De Man, "CoWare - A design environment for heterogeneous hardware/software systems". EURO-DAC'96, Geneva, Switzerland, September 16-20, 1996.

[Row94] J.A. Rowson, "Hardware/Software Co-Simulation". DAC'94, San Diego, USA, June 6-10, 1994.

[Sch95] M. Schütz, "How to Efficiently Build VHDL Testbenches". EURO-DAC'95, Brighton, Great Britain, September 18-22, 1995.

[Sch96] B. Schnaider, E. Yogev, "Software Development in a Hardware Simulation Environment". DAC'96, Las Vegas, USA, June 3-7, 1996.

[Sil95] A. Silburt, I. Perryman, J. Bergeron, S. Nichols, M. Duresne, G. Ward, "Accelerating Concurrent Hardware Design with Behavioral Modelling and System Simulation". DAC'95, San Francisco, USA, June 12-16, 1995.

[Soi95] J.P.Soininen, T. Huttunen, K. Tiensyrjä, H. Heusala, "Cosimulation of Real-Time Control Systems". EURO-DAC'95, Brighton, Great Britain, September 18-22, 1995.

[WAV] "WAVES Test Bench Utilities Packages for IEEE STD 1164".

[Ziv96] V. Zivojnovic, H. Meyr, "Compiled HW/SW Co-Simulation", DAC'96, Las Vegas, USA, June 3-7, 1996.