Platform-based Testbench Generation

R. Henftling, A. Zinn, M. Bauer, W. Ecker, M. Zambaldi
Corporate Logic, Design Automation
Infineon Technologies AG, Munich, Germany

Abstract

This paper presents a new technology that accelerates system verification. In a real life example, we achieved a speed-up of a factor of about 5000. The key for this speed-up is a configurable, synthesizable testbench architecture, which can be completely mapped to emulators or FPGAs. Exploiting generic controllers and re-using protocol-specific stimuli generators combined with topology and micro-program generation is responsible for almost zero overhead compared to behavioral testbenches.

1. Introduction

The complexity of large designs described on system level and therefore the complexity of the testbenches used for verification have steadily increased over the last decades. Accordingly, the verification time for such a design has dramatically risen and has reached an unacceptable level. It consists of the time needed for building the testbench, the pure simulation time and the time of a turn-around cycle.

Corresponding to design, re-use is a key methodology to increase the complexity of testbenches [2], [8], [9], [10]. This methodology can obviously applied to standardized protocol generators[1], because many designs serves one or more of these protocols. The protocol generators give the possibility to abstract the input stimuli and improve the test pattern generation [11]. These generators are usually tool-specific or language-specific and contain abstract generic protocols, coding templates, and/or a set of common subprograms[5]. Unfortunately, there is no standard for these solutions.

An other method to reduce verification time is to shorten the execution time for simulation, for example by using emulators or FPGA-based prototypes. Mapping the design to an emulator is straightforward, because it is nothing else than a re-mapping of the design to another target technology. However, the testbench is mostly written in a behavioral way and thus it is not synthesizable (i.e., it cannot be mapped automatically to hardware). A time consuming re-modeling of the testbench in a synthesizable way is needed.

The starting point for our work was a behavioral testbench environment as presented in [1]. The test patterns are written in a very abstract way, like micro instructions. The advantage of this approach is that the tests can be implemented efficiently and almost error-free with an obvious test coverage. The protocol instructions are passed to protocol generation units (PGUs) that serves the interfaces to the device under test (DUT). This approach guaranties as much flexibility as possible with a maximum of re-use. The evaluation of the test patterns is written back in form of a report that allows a fast and easy analysis, whether the test pattern were correctly executed and whether the expected values were reached.

In the following, we tried to combine the flexibility of a behavioral testbench with the execution performance of a synthesizable one. To reduce verification time without a dramatically increase in modeling time, we developed a re-use-centric method, which shows similarities to platform-based design [6], [7]. The method uses programmable control units, well defined interfaces and protocol-specific, synthesizable testbench elements. The control units can be generally re-used, whereas the synthesizable protocol generators can only be protocol-specifically re-used.

As for behavioral testbenches, the code for protocol generators are written in a re-usable, but synthesizable way. The sequencing and synchronization is done by a general, but generic and programmable hardware unit that behaves like a micro sequencer [12].

In addition, the behavior of the synthesizable testbench is the same as its behavioral counterpart, especially regarding synchronization and report mechanism. The new approach is as hardware independent, despite of the memories. To verify the new approach we decided to use an emulator, but also an FPGA-based prototype is conceivable.

This paper is organized as follows: First, related work is referenced. Next, the semantic of the abstract model and the target architecture are presented. Afterwards, the fundamental points how to come from a behavioral to a synthesizable testbench are explained. Finally, application of the testbench, experimental results, and enhanced concepts are discussed.

[1] The naming strongly depends on the underlying language. So, names as virtual component, testbench element, protocol generator, or transactor are used.

1530-1591/03 $17.00 © 2003 IEEE

2. Related Work

As mentioned, the dramatically increasing chip and testbench complexity is responsible for permanently increasing simulation times required for verification. In addition, larger time windows of the simulated scenarios increase the simulation times further. Hardware accelerated simulation is seen as one key technology to counter this phenomenon.

Partially accelerated testbenches can be modeled via API for most of the emulators. When using this approach, you can avoid writing synthesizable testbenches, but unfortunately you can not get the hole performance of the accelerator. In 1999, we used an Ikos NSIM accelerator and achieved a speed-up of about a factor of ten against a plain simulation with a software simulator [3].

An average speed-up of about a factor of 100 is feasible by using modern emulation technology [4]. For certain applications a higher speed-up is achievable. However, the speed-up is generally limited by the not accelerated part and the communication overhead as said by Amdahl's law [13].

In order to avoid this bottleneck, a synthesizable protocol generator must be written by hand or an existing data stream generator (e.g., from the companies Agilent or Heynen) must be used. The drawback of this solution is its limited application domain and limited debugging capabilities.

Our testbench environment reaches from a behavioral to a fully synthesizable abstraction level. It is possible to run the testbench with the same test patterns on a fast HDL simulator (e.g. for debugging), partially on an accelerator, or full on a hardware emulator. An automatic generation of most parts of the environment guaranties the consistence between the different variations.

3. Our Approach

After the description of the semantic, the realization of the target architecture is presented. On the base of this architecture we describe the testbench synthesis, which takes into consideration the instruction encoding.

3.1. Semantic of Abstract Description

In our behavioral testbench, the testcases are executed according to a static set of parallel threads, each consisting of a sequence of protocol operations, sequential control operations, object write operations, or synchronization operations. Data exchange between protocol threads as well as within one threads is done via global objects. The semantic of reading and writing the objects is deterministic because objects are updated at synchronization operations[2] only.

The number of threads is equivalent to the number of interfaces[3]. The pins and so the number of interfaces to the chip are assumed to be exclusive resources, so that one thread per interface is sufficient.

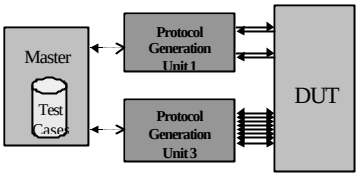

Figure 1: Implementation of the abstract description

The behavioral architecture for the execution of the testcases is implemented according to the master-worker pattern as shown in Figure 1. The workers are responsible for generating the protocols applied to the DUT and are consequently called PGU. The master reads the testcase description and executes statement by statement. The following cases can be distinguished:

1. If the statement requests the execution of a protocol operation, then the operation is not immediately executed. Instead, the request for the execution is stored in a queue. There is one queue for each worker.
2. If a synchronization statement is executed, then all operations stored in the queues are executed by sending a request for execution to the different PGUs in parallel, but only one request per PGU at a time. If one PGU has finished execution, then the request is acknowledged. If another protocol operation is in the queue (i.e., is planned for execution), then the next request is sent to the PGU. This continues until all queues are empty (i.e., all planned protocol operations are executed).
3. In all other cases (e.g., a control statement or an object assignment) the statement is executed immediately.

3.2. Target Architecture

To gain full performance we chose a decentralized approach as shown in Figure 2. The central controller of the behavioral testbench is split into one controller per testbench element (TBE). The controller is written in a re-usable and synthesizable way. Further, it will be called micro sequencer and is part of each TBE. A TBE is the unit, which interprets the testcase description and generates the protocol-specific signals to the DUT. Each TBE serves one protocol interface of the DUT. Besides this micro sequencer, a protocol generator and two memory blocks are instantiated inside a TBE.

[2] This is similar to the VHDL execution semantic, where signals are updated at the next wait statement earliest.

[3] An interface is a set of pins from the DUT which pertain to a specific protocol.

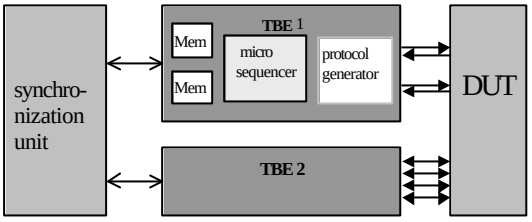

Figure 2: Testbench target architecture

The testcase description is distributed to the appropriated TBE, and stored inside a memory block called instruction memory unit. This sequence of instructions is interpreted as a micro program.

The return values of the protocol generators are the content of the other memory block. These are status values, start and end cycle of the instructions, and maybe return values from the DUT.

Each TBE has the same micro sequencer for reading the next instruction and an instruction decoder. All TBEs have to be synchronized by a synchronization unit. They are connected by a logical conjunction. This conjunction shows the micro sequencer, whether all other protocol generator units are waiting or if there is at least one protocol generator unit that is still operating.

The protocol generator can be protocol-specifically re-used only; all other parts can be generally re-used. The only adaptation of the memory blocks and the micro sequencer, which depends on the instructions, can be done by generics which adjust the width of signals.

Recapitulating, the synchronization unit, the memories, and the micro sequencer are re-usable, synthesizable, and re-configurable. The protocol generator is protocol-specific.

3.3. Testbench Synthesis

When we talk about testbench synthesis, we have in mind the way how to come from our behavioral testbench and abstract testcases to the target architecture and the definite instruction memory contents. At the moment, most parts of the target architecture have to be assembled by hand. In principle, the architecture could be automatically derived from the corresponding behavioral testbench. It must be pointed out that almost every component of the testbench is a general one that can be re-used and adapted to the current circumstances by generics. Only the protocol generator is interface specific, but it can taken from a re-use library or has to be written in synthesizable HDL. In addition, some VHDL packages are generated that contain the instruction information as well as the generic values and ease the adaptation of the TBE. On the other hand, the memory contents are completely generated.

The process how to gain the generated parts is illustrated in Figure 3. Basically, it is split into two phases. In the first phase the master of the behavioral testbench generates some intermediate files that contain the necessary information in an abstract format. During the second phase a post-processor written in Perl generates the final instruction encoding information and the final memory contents based on these intermediate files.

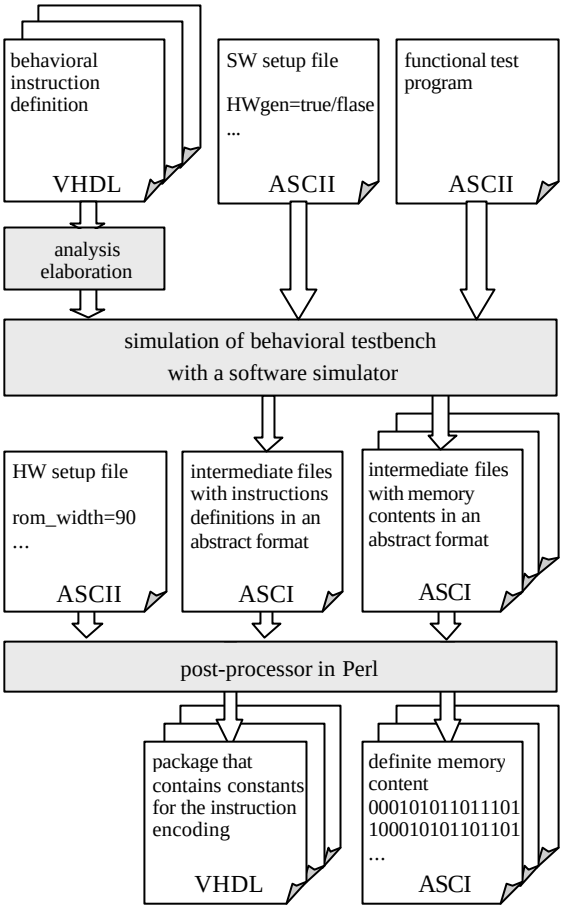

Figure 3: Data flow of testbench synthesis

The first phase of the testbench synthesis is a modified simulation of our behavioral testbench. Accordingly, the master firstly evaluates the setup file[4] that contains the information which testcase is currently to be simulated. It starts interpreting the functional testcases. However, instead of executing protocol instruction and synchronization operations, it generates the intermediate code files. There is one file that contains all defined instructions with all their parameters of all used TBEs. In addition, there is one intermediate file for every TBE that contains the micro program that has to be executed by this element. The synchronization operations are generated into all files.

As indicated in Figure 3, the generation of the intermediate file also depends on the instruction definition of the behavioral PGUs. These instruction definitions are placed in VHDL

[4] Generally, the setup file contains information that must be know right at the beginning of the simulation. In addition, it specifies, if a traditional behavioral simulation has to be done or if the necessary files for the synthesizable testbench should be generated.

packages and are stored in a sophisticated table structure. The tables contain the information which instructions are defined for a specific PGU, how many arguments and return arguments of an instruction are expected, and of which type the arguments and return arguments are.
During the second phase, first, the Perl post-processor reads a special setup file. It contains the information that is important for the generation of the final memory contents. Among other things, the width of the memory units and the format for the instruction encoding (see Section 4) is defined. Second, the post-processor reads the intermediate file with the instruction definitions and generates one VHDL package for every used TBE. These packages include all information for the generic parameters and preserves consistency between the instruction files, the memory contents, and the PGUs. Third, the post-processor reads the instruction files and generates based on the information of the setup file the final memory contents.

3.4. Instruction Encoding

As mentioned, the micro sequencer is responsible for reading one word from the instruction memory unit, for interpreting the read value, and for passing the extracted instruction and parameters to the PGU. Thus, it is an essential condition that the format of the instructions is well defined in order to execute them correctly.
The format of an instruction strongly depends on several factors. The most important one is the word width of the instruction memory unit. Number and length of the parameters of an instruction also have a wide impact on its format. Moreover, the fact whether the length of a parameter is static or may change dynamically must be considered. Finally, the number of different instructions defined for this TBE also influences the format. As a result of these facts the format may vary from instruction to instruction.
The basic instruction format is characterized in Figure 4. A complete word in the instruction memory unit is divided into two parts. One of these parts takes the instruction, while the other contains the parameters required for this instruction. The width of the instruction field can be derived from the number of defined instructions for this TBE. The rest of the memory word is considered to be the parameters.

n-1 ... number of bit 0

Parameters	instruction

Figure 4: Basic instruction format

There are two principle ways in which the parameters are assembled in the parameter field: serial and parallel. In this context, serial means that all parameters are stored one after the other. In contrast, parallel means that there are columns in the memory unit and one column is used to store one parameter.
The serial format is illustrated in Figure 5. As mentioned, the parameters are stored one after the other. Accordingly, two different alternatives may occur. First, the width of all parameters and the width of the instruction together are smaller than the word width of the memory unit. In this case the rest of the memory word is unused and is filled up to the instruction length with zeros (see Figure 5, instr1). Second, the width of all parameters and the width of the instruction together are bigger than the word width of the memory unit. In this case the command is split into two or more memory words as shown (see Figure 5, instr2).

n-1 ... number of bit 0

00000	param2	param1	instr1
param2	param1		instr2
000	param4	param3	instr2

Figure 5: Serial instruction format

The parallel format is pictured in Figure 6. Every parameter has its own column in which it is stored. If one parameter is bigger than the width of its column the whole command has to be split over two or more memory words. The part of a column that is not used by a parameter is filled up to the column width with zeros. Generally, the parallel format is only used for very long parameters and if the parameters are required inside the PGU at the same time.

n-1 ... number of bit 0

param2		param1		instr1
00000000	param2	00	param1	instr1

Figure 6: Parallel instruction format

In addition, parameters with arbitrary data length have to be discussed. In this case, the parameter in the memory unit is split into a pair of values. The first value specifies the length of the second value, while the second value is the parameter itself. In this case the length field must of course have a defined size. Consequently, we come to a format that is shown in Figure 7.

n-1 ... number of bit 0

param1	length	instr1
00000000	param1	instr1

Figure 7: Parameter of arbitrary length

The coding of the instructions and the parameter length are generated from the existing behavioral instruction specification. Therefore, it is necessary to supplement the instruction definition. Further information about the instruction format is given in a setup file. Here the data width of the memory unit is specified. In

addition, the user can determine, whether the parameters of an instruction are assembled in serial or parallel format. Finally, the width of the length field for parameters of arbitrary data length must be defined.

To guarantee full compatibility between the memory content and the protocol generator, a TBE-specific VHDL package is generated. This package contains constants that represent all important information of the instruction encoding. Accordingly, the micro sequencer unexceptionally uses these constants to access the instruction memory unit. Instructions which regard the micro sequencer (e.g. synchronization commands) are defined in an own sequencer VHDL package for re-use reasons.

Instructions with *tristate* or *don't care* values in the parameters have to be replaced. Each value-bit '0', '1', 'Z', and/or 'D' is represented by two bits which means that two data signals must be used when mapped to a hardware platform. The second signal is like an enable signal. Because many emulation systems do not support *tristate*, the tristate-pads of the DUT are replaced by a logic to evaluate the 2bit-coded values.

4. Application and Results

The usability of our approach has already been proven both in a pilot project as well as in a real world application. During the pilot project, the DUT was a simple multiplication unit. For computing the result, it needs a certain number of clock cycles depending on the applied stimuli. The synthesizable testbench consisted of three TBEs: two stimuli generators and one analyzer. After the successful evaluation in the pilot project, we continued with a real world example. Here, the DUT was a hard disc controller with several interfaces (e.g. micro controller interface, JTAG interface, pin-manipulation interface). This application of our approach showed very promising results as well.

Both the pilot project and the real world example were executed on the *Celaro* emulator from Mentor Graphics. In the first case, the testbench and the DUT ran with full speed of the *Celaro* which is about 2 MHz. In the latter one, the testbench with the DUT ran with at least 300 kHz in full probing mode[5]. Table 1 shows the reachable execution frequencies of all used TBEs separately, the complete testbench (with the four testbench elements), the DUT component itself, and the testbench in combination with the DUT. In addition, the sizes of all parts are displayed. The table clearly shows that the design - not the testbench environment - is the execution time limiting factor. If it is possible to improve the execution speed, especially of the design (e.g., by using FPGA prototypes), then the simulation speed can be increased further on.

Used components	Number of gates[6]	Frequency on *Celaro*
JTAG TBE	12,000	1,200 kHz
Micro TBE	7,500	700 kHz
Reset TBE	4,300	1,400 kHz
Shell TBE	101,000	700 kHz
Complete testbench	130,000	660 kHz
Disc controller	530,000	395 kHz
Disc controller with testbench	660,000	395 kHz

Table 1: Frequency and size of all used blocks

Our new approach reduces simulation time dramatically. This can be illustrated by comparing the execution performance of traditional simulation on a software simulator and the execution performance using our new approach. With traditional simulation, frequencies of about 10 Hz up to 100 Hz are reachable. Our behavioral testbench runs with 78 Hz[7]. This frequency depends very much on the executed testcase: An event-driven simulator is very fast when executing only a few events. In contrast, our real world example could be executed with almost 400 kHz. Therefore, with our new approach a speed-up of about a factor of 5000 can be achieved against plain simulation. In other words, a verification time of one week using traditional simulation can be reduced to 120 seconds.

As already mentioned, the memory contents for the synthesizable testbench are automatically generated from its behavioral counterpart. Hereby, the generation is done in two phases. The time needed to generate the intermediate files based on a varying number of protocol instructions is shown in Table 2. The second phase takes only about two seconds and is relatively independent from the number of protocol instructions.

Number of instructions[8]	Time in seconds
81	4.9
145	7.4
874	30.0

Table 2: Runtime to generate intermediate files

5. Enhanced Concepts

The presented concept is not restricted to one thread per PGU. Descriptions which allow mixing of protocol operations inside one thread and a dynamically changing number of threads can also be mapped to the target architecture with

[5] Full probing mode means that all signals in the design are recorded and can be shown in the waveform viewer.

[6] Number of gate equivalents without memories

[7] The frequency is calculated by dividing the executed clock cycles by the absolute simulation time.

[8] Equivalent to number of lines in the instruction file

one micro sequencer per protocol unit. Two extensions must be made:

1. Each access to the protocol unit remains an exclusive operation (i.e., if two threads request the execution of a protocol from one protocol unit) then they must be synchronized implicitly.
2. The synchronization concept must be enhanced. It is no longer sufficient to have only one signal from each TBE to the central synchronization unit to indicate that a synchronization point has been reached. This is necessary because the synchronization points are no longer definite.

The presented concept also works fine if pins are no longer exclusive resources of one protocol. This is the case if the DUT possesses multiplexed pins or protocol processors. We insert in this case a switching unit called testbench shell between the PGU and the DUT (see Figure 8). We keep one controller and one sequential execution thread per protocol that is to be generated. However, we switch the connection by configuring the switching unit similarly to requesting execution of protocols from a protocol generator. Alternatively, each PGU could be extended by an active signal. This active signal is then used in the testbench shell to configure the switch.

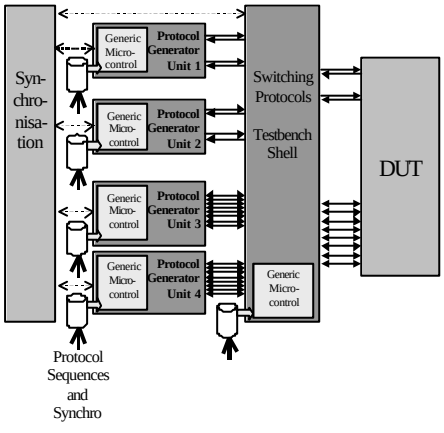

Figure 8 Switching protocols

The testbench shell concept can also be applied if the description style of the testcases would allow several threads accessing a PGU (as described above). The implicit synchronization concept of exclusive protocol operations must then be extended from one PGU to a set of PGUs sharing the same physical signal.

6. Summary

Our new testbench approach was applied on a real world design of a hard disc controller. The evaluation showed that the time required for the execution of the test scenarios can be reduced up to an average factor of 5000.

We presented a methodology that combines the benefit of abstract testcases with the execution performance of a synthesizable testbench. Our approach includes a seamless flow to generate most parts of the synthesizable testbench from its behavioral counterpart.

The key attributes of our approach are separation of protocols, higher level communication/synchronization mechanism, accelerated verification, and re-use.

Our further work will allow the use of SystemVerilog for testcase descriptions. In addition, we will realize our approach on an FPGA- based prototyping board in order to speed up the simulation up to 50 MHz. Another interesting point is the random generation of test patterns in hardware.

Acknowledgement

The verification of this approach was done in cooperation with Mentor Graphics. We thank the whole disk drive team from Infineon for giving the design to us and for supporting us in understanding it. Also we thank Mr. Dirk Hansen from Mentor Graphics for fruitful discussions, for setting up several testcases on Mentor Graphics' *Celaro* emulator, and for his help during the generation of the results for this paper.

7. Bibliography

[1] Bauer, M. and W. Ecker: "Hardware/Software Co-Simulation in a VHDL-Based Testbench Approach", DAC, 1997.

[2] York, G., Mueller-Thuns, R., Patel, J., and D. Beatty: "An integrated Environment for HDL Verification". International Verilog HDL Conference, 1995.

[3] Bauer, M., Ecker, W., Henftling, R., and Zinn, A.: "A Method for Accelerating Test Environments", EUROMICRO, Milano, 1999.

[4] Eric Melancon: "A C/C++ Testbench for Acceleration", Cadence white paper, Talk Verification Newsletter, August 2002, Volume 7, Issue 2

[5] EVCs from Verisity: http://www.verisity.com/products/evc.html

[6] Gary Smith: "Analysis of platform based Desing",http://www.eedesign.com/story/OEG2000906S0074

[7] Alberto Sangiovanni-Vincentelli: "Defining Platform based Design", http://www.eedesign.com/features/exclusive/OEG20020204S0062

[8] Davik Dempster, Michael Stuart: "Verification Methodology Manual – Techniques for Verifying HDL Designs". Teamwork International and TransEDA Limited, Second Edition June 2001, ISBN 0-9538-4821-3

[9] Marc Erickson: "Verification Crisis: Braccio tackles reusable verification", EETimes 18 June 2001, http://www.mint-tech.com/marc_erickson.htm

[10] Schütz, M.: "How to Efficiently Build VHDL Testbenches". European Design Automation Conference, S. 554-559, Brighton, England, September 1995

[11] Bauer, M., Ecker W.: "A VHDL-Based Hierarchical, Highly Flexible, and Extendable Testbench Approach". IEEE International High Level Design Validation and Test Workshop, Oakland, USA, November 1996

[12] Donnamaie E. White.: "Bit-Slice Design: Controllers and ALUs", Garland STPM Press, available at http://www10.dacafe.com/book/parse_book.php?article=BITSLICE/index.html

[13] David A. Patterson and John L. Hennessy: "Computer Organization & Design – The Hardware/Software Interface"; Morgan Kaufmann Publishers, Inc, S. 75