Accelerated Verification of Digital Devices Using VHDL

Sandi Habinc, Peter Sinander

European Space Agency, Electrical Engineering Department, Microelectronics Section
Postbus 299, NL-2200 AG Noordwijk, The Netherlands
Tel. +31 71 565 6565, Fax. +31 71 565 4295
sandi@ws.estec.esa.nl, psi@ws.estec.esa.nl

Abstract

As digital designs become more complex and increasingly include a processor on the chip, verification – and in particular the generation of testbenches – is becoming a bottleneck. This paper presents two aspects for improving the verification of microprocessors; program-less verification, and methods for handling large differences in abstraction level between a reference model and the actual design. Program-less verification is a type of pseudo random verification where the notion of a software program executing on the microprocessor has been abandoned. This removes some restrictions on the pseudo random data and instruction streams, and avoids limitations and bugs in software tools such as assemblers. A high level of abstraction is important when creating reference models, since it reduces the risk for introducing errors in the reference, as well as the effort for coding it. However, the testbenches must then be able to cope with behavioural differences due to phenomena such as prefetching, speculative execution etc.

1 Introduction

With growing complexities of digital devices, the generation of a complete verification suite requires an ever increasing portion of the development time and resources. Today, even though extensive verification has been employed, devices with incorrect functional behaviour are occasionally manufactured and marketed. Typically, the erroneous logical implementation has not been detected due to inappropriate verification criteria. An important contributing factor to such oversights is that those preparing the test suite inadvertently tend to base it on anticipated device applications and therefore misinterpret the specification in the same way as the designers. An example of this could be when an embedded microprocessor is verified by co-simulation with the target application software. Another contributing reason is that the verification bottleneck lays in the time spent on developing the test suite rather than simulating it (Ref. 1).

We have approached these problems by stimuli generation for microprocessors decoupled from the foreseen usage of the device. Our method relies on comparison of behaviour between different representations of the microprocessor. The notion of a valid software program is abandoned for the stimuli generation, effectively resulting in a program-less verification approach for microprocessors. Additionally, limitations associated with software developments tools are eliminated in the stimuli generation process. The ratio between simulation time and testbench coding time is increased, allowing a larger number of errors to be found per time unit spent on stimuli development. These characteristics yield in an acceleration of the overall verification process.

For complex designs it is necessary to obtain a correct reference model that is used throughout the development (Ref. 2). When describing a reference at a high abstraction level, the functionality is not obscured by implementation details which allows an easier understanding of the problem and reduces the possibility for introduction of errors. By applying the same stimuli to the test object as to the reference model, errors can be detected by comparing the obtained results. The testbench has to be able to adapt dynamically to the response from either model to tolerate correct but dissimilar behaviour resulting from the models being at different abstraction levels.

The stimuli is pseudo randomly generated and applied to the test object and the reference. Existing methods employ generators coupled with assemblers and other software development tools, producing pseudo random software programs to be executed as stimuli. Such stimuli have some disadvantages since they have to adhere to the limitations that are attributed to a valid and executable program. Instead of relying on existing pseudo random generators and software tools, we generate the stimuli during the actual simulation. A microprocessor is simply seen as a state machine being fed with instructions and data which are conceived on the fly, effectively resulting in a program-less verification approach. Any correlation between the specified function and its usage is thus avoided by completely abandoning the notion of a program.

To cope with different abstraction levels and to allow program-less verification, we have chosen to develop the complete testbench in VHDL. The testbench includes the reference model and the test object, and pseudo random generation of executed instruction and data sequences. Today, testbench development is more time consuming than the actual simulation. Methods increasing the simulation portion of the time spent on verification are therefore becoming attractive. The outlined approach offers an acceleration of the verification process by reducing the effort required for the definition and implementation of a stimuli for finding a given number of errors.

This paper starts with an overview of different functional verification approaches. It is followed by a presentation of our method and an in-depth description of how it was applied to a real project. Some relevant cases are then discussed. Finally, the conclusions from the experience are drawn.

2 Current verification approaches

2.1 Manual stimuli and inspection

In the past when designs were small, tests were conceived manually. For the stimuli, the simulator was commanded directly in front of the workstation, and the response in the form of waveforms was visually inspected on the screen.

Depending on the response, subsequent stimuli were conceived in an interactive manner. For example, if an error was detected, its effect on the verification could be reduced by avoiding to exercise that part of the design until the error had been corrected.

While this approach is manageable for small designs, its drawbacks include:

- risk for confusing the actual behaviour as seen on the screen with what has been specified, especially when under pressure to complete the work;
- limited length of the verification since requiring the person to be present at the workstation while simulating;
- unsuitable for repeatable regression testing since interactive, and furthermore the quality of repeated simulations varies with the day-to-day performance of the person running it.

2.2 Extension for regression verification

As an extension to the above approach to handle regression testing, simulator commands were recorded in a file during the first simulation, and then played back for repeat simulations. Similarly, the response from the design was also stored in a file, for example as periodically sampled signal values. The manually inspected response from the first simulation was stored as a reference file, and files resulting from subsequent simulations were then compared to this reference using some sort of comparison utility.

This extension is suitable for verifying that the design behaves exactly as during the reference simulation, for example to verify that it has not changed due to a modification in the design. However, due to the rigid criteria to exactly correspond to the reference simulation, it is not well suited to handled minor changes in the behaviour that are still within the specification. An example of such a case could be a non-critical signal being active one clock cycle later than in the reference simulation. Even if this would be acceptable from a specification point of view, it would be signalled as an error by the verification, since it is not sufficiently intelligent to know whether or not such a change represents an error. Since the stimuli is just a playback without any feedback from the design, changes could get the stimuli completely out of synchronisation with the actual state of the design.

2.3 Automated testbenches

With the introduction of hardware description languages, the designer was not only provided with a powerful tool for describing the design itself, but also with a tool highly suitable for verification. The concept of testbenches was introduced, comprising stimuli generators and checkers for functional correctness. The correctness checkers can be devised in ways allowing different, although correct, responses from different implementations or representations of a design under test. With checkers that abstract from implementation details and concentrate on functionality, verification at a higher abstraction level is made possible.

An example of how verification is performed at a higher abstraction level is when a checker comprises a bit-serial receiver that outputs the received data for further processing. A higher level checker can then analyse the received data without involving any implementation details of the communication protocol in the process. The checker is thus not bound by cycle or waveform-true comparison and can therefore accept low-level deviations. When cycle-true comparison is desired, as for regressive verification, the testbench can be complemented with signature generators and checkers based on multiple input serial registers.

In this approach the objective is to verify each specified function of the design in a systematic manner. However, complicated interaction that might exist between different functions can be overlooked during the verification. The effort to find obscure errors is often a laborious task since the creation of explicit stimuli exercising all aspects of the design require careful analysis of the functionality and the implementation itself.

2.4 Pseudo random stimuli

To improve the chances to detect obscure errors in complex designs, pseudo random stimuli generation is often introduced to complement the systematic verification. Since it is nearly impossible to determine the correct response from a pseudo random stimuli, comparison with a correct reference model is often used (Ref. 2). One important criterion is that the test object and the reference have been developed independently in isolation to eliminate unintentional correlation leading to systematic errors.

The same stimuli in the form of a program is executed on both the test object and the reference. The logical significance or meaning of the test object output, being address and data for a microprocessor, is arbitrary and is therefore not considered in the verification. The test object output is therefore only compared with the output of the reference. Any deviations encountered must consequently be examined, which normally requires that the behaviour of both objects is analysed since the cause of the problem could originate in the reference. As long as the behavioural differences are minute, the reference can be represented in a variety of styles, ranging from instruction simulators and behavioural models to real hardware.

For microprocessors, a common way of producing pseudo random stimuli is to compile a test program using a test generator, typically written in the C language (Ref. 2, Ref. 3). The generation process normally requires the use of an assembler or other software developments tools, which introduces limitations on the produced test program. Since the test program is generated prior the actual execution of the instruction sequence, there are no possibilities for adapting the stimuli to the response obtained during the simulation. This limits the allowable behavioural differences between the test object and its reference. It effectively renders the method unsuitable for developments with large differences in abstraction level between the reference and the test object.

The usage of a software program as executable stimuli inadvertently introduces correlations between processors and their foreseen applications. Executable programs are bound to the following characteristics, limiting the efficiency of current pseudo random verification approaches:

- deterministic result and purpose of any subprogram;
- correlation between processor state and memory:
- an instruction fetch to an address expects a certain instruction to be returned;
- a data fetch to an address expects that previously written data are returned;

- exceptions and subroutines need to return to their origin;
- limitations in program flow, avoiding backward jumps resulting in infinite loops etc. (Ref. 4);
- deterministic termination of the program.

Previous work has been performed to improve the efficiency of pseudo random verification. Instead of applying fully random data values to an arithmetic unit, weighting of the parameters can be performed to cover corner cases and improve the test coverage (Ref. 5). To reduce the stimuli length, intelligent instruction selection can be performed taking into account the predicted internal state of the test object (Ref. 6). To improve the quality of the generated stimuli, sequences can be generated that explicitly exercise different functional units simultaneously to increase the possibility to catch errors related to complex function interaction (Ref. 4).

Little effort has however been made to remove the limitations inherent to software tools and the notion of software programs, an area that we will explore further in the following section. To achieve optimal verification, it is important that all the above approaches are being addressed.

3 Accelerated verification approach

We add a new aspect to the generation of the pseudo random stimuli than what has been previously addressed. Stimuli are generated on the fly without accepting the limitations of executable programs, also taking into account any deviations between test object and reference responses. All the necessary steps can be performed directly in VHDL without the need for external software development tools. It is basically a hardware designers approach to software-free microprocessor verification.

3.1 Program-less pseudo random stimuli

There is no resemblance of a software program in the testbench, only a sequence of instructions to be executed. The testbench does not implement a traditional memory in which a program is stored. When the processor reads instructions or data, it is provided a pseudo random value not related to the issued address. Two instruction fetches to the same address can thus result in different instructions. A read operation will not return the same data as accessed by previous read or write operations. If an instruction was read from an address at one time, the test sequence might very well result in a data fetch from the same address later on, effectively eliminating the notion of data and program address space.

The deterministic flow of a program is abolished, allowing any sequences of jump and call instructions to be realised. If a jump is made to the same address as the current program counter value, the subsequent instruction fetch to the same address will generally result in the return of a different instruction. There is therefore no need for analysing jump sequences to prevent that infinite loops occur, since this risk is eliminated trough the merge of the code generation and execution.

A subroutine call or exception handling routine does not have to return to the original program flow as is in real executable software, since there is no program flow what so ever in our approach. As a result, the significance of data such as operational parameters is ignored, allowing e.g. a division by zero without disturbing the simulation.

3.2 Coping with different abstraction levels

As said in the introduction, for complex designs it is necessary to obtain a correct reference model that will be used throughout the development process. By describing the reference at a high abstraction level, the functionality is not obscured by implementation details which reduces the possibility for introduction of errors. However, the difference in abstraction levels will most likely result in behavioural differences between the two models.

For example, if the test object reads a group of operands in a different order than the reference, the computed result will still be the same. The same applies to prefetched instructions that are not executed as well as speculatively executed instructions which are discarded. An example of the former is when one of the models makes a prefetch of an instruction superseding a branch instruction which is not executed because the branch is taken, whereas the other model does not make the prefetch at all; the overall program flow still remains the same.

To cope with the behavioural differences between the two models, the testbench has to analyse actively in real-time the response from the test object and modify the stimuli as necessary. By applying certain tolerance levels for such deviations instead of immediately reporting an error and terminating the simulation, much of the associated analysis work can be automated.

3.3 Verification acceleration

The engineering task is then to develop a stimuli generating infrastructure that spawns the test suite and handles different behavioural responses from the two models. Arbitrarily long simulations can then be performed without human interventions. As previously mentioned, the simulation time is not the limiting factor, testbench development is the bottleneck. We move the test case generation to the actual simulation phase, allowing more test cases to be simulated with less development effort and therefore accelerating the verification process.

Although the method might seem to be simple, applying it to an actual real-life project is not always straightforward as will be shown in the following section.

4 Verification of a Mil-Std-1750 microprocessor

The objective was to independent verify the correctness of the instruction set implementation of a microprocessor model, ignoring all signal properties such as timing and waveforms.

The object to be verified was a VHDL model, intended for board level simulation (Ref. 7), of an existing 16-bit microprocessor according to the MIL-STD-1750 instruction set standard. The model was described at a high abstraction level, although encompassing architectural bit-level details. The model was capable of executing approximately one thousand instructions per second when run on a SparcStation 10.

The board level model had already been extensively verified as part of the development. The foundry production test had been applied to the model, including an instruction test verifying the standard. In addition, specific tests for each instruction implemented had been devised, including testing of boundary conditions etc. The timing and bus cycle behaviour had also been verified. Finally, the model had successfully executed simplified application software in a board level system.

The reference model had been converted to VHDL from the design data base used during the development of the existing hardware. The model was slow, only executing a few instructions per second when run on a SparcStation 10. Several iterations of the hardware had already been used for some years.

It should be noted that in this project the reference model was much slower than the test object, since the task was to verify a fast board level model. This does however neither affect the approach nor the conclusions from this exercise, since in an active device development the roles would simply be reversed for the two models.

4.1 The verification approach

The verification was performed without acquiring any implementation knowledge of the test object or the reference model. It was assumed to be sufficient to understand the instruction set architecture, treating the two models as pure black boxes. In this way, any correlation between the implementation and the verification was eliminated.

The testbench was implemented exclusively in VHDL, using no external software tools such as compilers, assemblers or linkers. Both models were simulated simultaneously on the same simulator. This was necessary in order to handle differences in timings, number of clock cycles per instruction, bus cycle behaviour, and in instruction and data flows.

We initially took a broadside approach to pseudo randomised stimuli generation, with no or only a few limitations during instruction and data flow generation. To allow long sequences of legal (and illegal) instructions the testbench had to be able to accept out of order instruction and data fetches, handle allowable mismatches between the two models, etc. The objective was not to halt the simulation immediately, but to try to recover and continue if the deviations upon detection of a discrepancy were found acceptable. The goal was to accelerate the verification process by reducing the time spent on the testbench development and instead spend the available time on simulation and error analysis.

4.2 Instruction set data base

To be able to handle behavioural deviations between the two models, a detailed instruction set data base was described in a VHDL package. The data base was used during generation of the instruction and data sequences, as well as during error reporting that will be discussed later.

One of the primary functions of the data base was to allow the code generator to know what kind of information the microprocessor would expect during a bus access. For example, it was possible to identify whether to provide an opcode or a parameter to the microprocessor during an instruction fetch. This was used for several different purposes, one being the possibility to exclude arbitrary instructions from the verification if necessary.

The following list shows some of the information that was stored in the data base for each instruction:

- mnemonic, description and usage;
- opcode, number of operand accesses and data types;
- address mode and register usage;
- potential exceptions.

4.3 Instruction filtering

The instruction set data base allowed selection of subsets of the available instructions. It was possible to exclude an instruction from the test set if it was found working incorrectly. This allowed further verification runs without generating more errors of the same type. The verification could thus continue while the model was being corrected. The instruction in question could then be enabled again for further verification. This made it possible to target specific instruction types, e.g. jump, etc., or to select only instructions with previously erroneous implementations.

4.4 Instruction sequence generation

The generation of the instruction sequence was performed in two steps. Firstly, a partial sequence was generated for the reference model which was executed an arbitrary number of instructions ahead of the test object. The generated instruction and data flow was stored together with associated address information in a first-in-first-out type of virtual buffer. The test object was then fed with the instructions and data from the buffer as they were accessed. The test object was not bound to execute the buffered instructions in the same order as they were stored. This was done to allow mismatches between the test object and reference behaviour which will be described further on.

The testbench decoded the bus cycles as generated by the reference model, to know what kind of information was expected. When a compound instruction was fetched, the whole instruction with all associated parameters was generated and remembered. As the reference fetched the remaining parts of the compound instruction, the already generated information was dispatched. The instruction set data base was instrumental in the generation of the reference instruction and data sequences. The selection of what instruction to generate was done pseudo randomly, but could be controlled by the aforementioned filtering capability.

The testbench also decoded the bus cycle accesses made by the test object. If an instruction was requested, the testbench would first see if the provided address was available in the buffered instruction sequence and would in such a case dispatch it. If the requested address was not first in the queue, the preceding instructions would be reported and removed from the buffer, allowing instructions to be dropped.

If the requested address was not in the buffer, a non-operation instruction would be issued, for example to allow the test object to have a different prefetch implementation than the reference. This also allowed the test object to generate an exception not previously done by the reference, without immediately upsetting the simulation.

A similar approach was followed for data fetches, but in this case buffered data were not immediately dropped to allow out of order operand fetches. Special provisions were implemented for atomic operations etc.

All the aforementioned mismatches between the two models were tolerated up to a user defined acceptance level, before the affected test sequence was terminated and the failure was reported.

4.5 Data generation

Although pseudo random generation of operands and other data should had been straight forward, some instruction set particularities complicated the task. To obtain deterministic results for some floating point operations, the operands, in both the memory and internal registers, had to be normalised. If this was not performed, the outcome from the computation would had been unspecified and different behaviours could be expected for the two models. Some of the data structures required alignment to even addresses when stored in memory and registers for correct operation, which also had to be handled explicitly in the testbench.

Purely random verification of a floating point unit is unlikely to find all errors. For example, there is only a small chance that the mantissas of two random operands will overlap and result in an actual calculation (Ref. 5). To overcome this limitation and to excite corner cases, we generated weighted pseudo random operands in the testbench, such as small or large negative and positive numbers, and floating point zero.

4.6 Observability and reporting

To improve the possibility for detecting errors, the visibility of the internal state of the microprocessor was increased by periodic readout of all internal registers. The instruction set architecture conveniently allowed all internal register to be written to memory by issuing a single instruction. The purpose was to allow the testbench to catch errors located in flags etc., that might not propagate to the memory before being overwritten. Note that no changes in the black box models were required.

An important trade-off to be made is how many random instructions should be executed before the internal registers should be read out. By allowing a large number of instructions between register dumps, errors might be masked by subsequent instruction execution. By only executing a few instructions between register dumps, errors due to interactions between instructions might be masked and the efficiency decreases due to overhead. The testbench implementation allowed an arbitrary number of instructions to be executed between internal register dumps.

To assist debugging, the testbench generated a disassembled listing of the executed instruction sequence. Not all code was reported, only the last relevant part before a major deviation occurred between the test object and the reference. The disassembly was also performed in VHDL, using the instruction set data base. The listing further indicated what instructions or operands had been dropped or added to make up for the differences in model behaviour, what data deviations had occurred and where exception handling had taken place in the form of a call to a interrupt routine.

A simple profile of the executed instructions was also created during the simulation, indicating how many times each instruction had been executed. More advance profiling tools have been reported (Ref. 2, Ref. 4),

4.7 Results

The verification resulted not only in the discovery of a large number of deviations between the board level model and the reference, but also revealed a few errors in existing hardware. Needless to say, the functional errors were corrected in the next revision of the microprocessor.

There were also several non-errors that were reported, requiring some effort to analyse and occasionally to adapt the testbench to cope with legal deviations between the models. After a couple of iterations, the testbench was capable of handling all acceptable deviations and long simulation runs could then have taken place. This was however not possible since new errors were regularly discovered which prevented any long instructions sequences, except for when all erroneous instructions were disabled.

The method will generally report many errors for each incorrect instruction implementation, which will require extensive analysis before resolving the cause behind each report. The approach is consequently suited for demonstrating that there are no or few errors left in a design at the end of the development.

Considering that the board level model had already been thoroughly verified, the selection of the approach was right at that time. If knowing that so many errors would be discovered, perhaps a simpler approach could had been taken to eliminate the bulk of the problems before employing the presented approach to find the remaining errors at a lower expense than what had been the case.

5 Related verification cases

The program-less approach is not limited to VHDL, but can be applied to any hardware description language as long as it is possible to generate the stimuli on the fly, while taking into account deviating responses from the test object and the reference. The following two examples will illustrate how the approach has been used, or how it sometimes has not been possible to use due to surrounding constraints.

5.1 Microcontroller

To illustrate some limitations with off-line generation of pseudo random programs, we have include an example on how the discovery of an error caused lengthy analysis work due to development tool limitations. A microcontroller was developed based on a synthesisable VHDL core from an intellectual property provider.

The test object was first subjected to all verification suits that were provided with the core, followed by pseudo random testing. The reference was represented as a commercial hardware emulator that was part of the software development environment. The stimuli was generated with a custom made C program and was output as an executable binary. In total, three errors were discovered by this additional verification.

One error in the core was located in a jump instruction, resulting in an incorrect program counter update. Since the stimuli was executed as a program, the jump introduced an infinite loop that never terminated. The simulation run for several hours before it was decided to abort it. Since the program did not terminate, there was no indication of what had gone wrong. To trace the problem, the stimuli was disassembled and run with a software debugger coupled with the VHDL simulator until the incorrect jump was identified. What further complicated the analysis process was that the available assembler was not able to generate the jump instruction under the same conditions as during the pseudo random test. A diagnostics program had to be assembled by hand before the cause of the error could be confirmed.

This example illustrates that pseudo random generators which rely on software tools are subject to a number of limitations. The problems with locating the error would have not occurred if the instructions would have been generated on the fly as in the program-less approach, since an infinite loop would not have been created. The execution would have terminated as soon as the number of deviating addresses issued during instruction fetching would have exceeded an arbitrary tolerance level.

5.2 Digital signal processor

A simplified version of the approach was applied during the validation of a digital signal processor. The device in question had been transferred from a commercial technology to a radiation-tolerant process. The random stimuli were applied to the first prototype devices and to the already existing commercial counterpart. The objective was to prove that the two devices were identical, rather than that the new device fulfilled all user requirements. In this setup, the applicability of the method was limited since program-less verification could not be applied without redesigning the hardware. This was however compensated by the speed of the execution, which allowed pseudo random instruction suites with lengths widely superseding what is feasible to simulate.

The only discrepancy discovered between the reference and the test object was related to one single arithmetic operation, where one bit in a status register was flagged one clock period too late. It turned out that the reference was of an older revision of the processor than what had been used for developing the radiation-tolerant device. The error was thus located in the reference and not in the test object. In fact, the only documented difference between the two revisions was the anomaly in said arithmetic operation. This single anomaly caused the test program to report many errors, which indicates that the effectiveness of the verification approach was high.

6 Conclusions

A program-less approach to microprocessor verification using pseudo random stimuli generation eliminates many of the limitations associated with verification through program execution and with related software development tools.

As illustrated in the examples, the method will generally report many errors for each incorrect instruction implementation, which will require extensive analysis before resolving the cause behind each report. The method is therefore most suited for verifying that there are no errors in a design, rather than as a means for identifying errors during the device development.

For complex designs it is necessary to obtain a correct reference model that can be used throughout the development process. By describing the reference at a high abstraction level, the functionality is not obscured by implementation details which reduces the possibility for introduction of errors. The difference in abstraction levels will however most likely result in behavioural differences.

Although the method tolerates behavioural differences between the test object and the reference, analysis of errors discovered by pseudo random stimuli can be very time consuming. A design should therefore be properly verified throughout the development phase, since errors detected late in the verification process are costly to handle (Ref. 8). For a few of the discussed examples it was not possible to perform program-less stimuli generation that adapted to the response from the execution. The main reason for this was that the reference or test object was represented as existing hardware.

The need for a fast reference model when developing devices in excess of half a million gates has been claimed in several reports (Ref. 2, Ref. 5). In most cases the reference model is developed in the C language to offer high simulation performance as compared to VHDL. Our and others' experiences are that the simulation time requires a lesser part of the verification phase than what is needed for developing and debugging of testbenches (Ref. 1). The conclusion is that the speed of the reference model is not that important when the test object is modelled at a low abstraction level and will thus in any case be responsible for most of the simulation time.

References

1 *Functional Verification of Large ASICs*, A. Evans et al., Proceedings of the 35th DAC, 1998

2 *I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor*, M. Kantrowitz et al., Proceedings of the 33rd DAC, 1996

3 *Hierarchical Random Simulation Approach for the Verification of S/390 CMOS Multiprocessors*, J. Walter et al., Proceedings of the 34th DAC, 1997

4 *Code Generation and Analysis for the Functional Verification of Microprocessors*, A. Hosseini et al., Proceedings of the 33rd DAC, 1996

5 *Verifying the PA-8000's FPU*, D. Smentek et al., Integrated System Design Magazine, March 1997

6 *Functional Verification Methodology of Chameleon Processor*, F. Casaubieilh et al., Proceedings of the 33rd DAC, 1996

7 *Using VHDL for Board Level Simulation*, S. Habinc, P. Sinander, IEEE Design and Test of Computers, Autumn 1996

8 *A C-Based RTL Design Verification Methodology for Complex Microprocessors*, J. Yim et al., Proceedings of the 34th DAC, 1997