AN EFFECTIVE AND FLEXIBLE APPROACH TO FUNCTIONAL VERIFICATION OF PROCESSOR FAMILIES

David Malandain, STMicroelectronics, Rousset, France, david.malandain@st.com
Pim Palmen, STMicroelectronics, Bristol, UK, pim.palmen@st.com
Matthew Taylor, STMicroelectronics, Bristol, UK, matthew.taylor@st.com
Merav Aharoni, IBM Research Labs, Haifa, Israel, merav@il.ibm.com
Yaron Arbetman, IBM Research Labs, Haifa, Israel, arbetman@il.ibm.com

Abstract

Functional verification is one of the most critical stages of microprocessor design. Its goal is to achieve the maximum level of confidence in the conformance of a processor design to its specification. A powerful methodology is necessary in order to cope with the major technical challenge which is posed by functional verification of a processor, and which stems from the vast state space that must be verified. This need becomes even more crucial when faced with the concurrent verification of several processor families. In this paper we describe a strategy for verification of several designs, which allows for maximum sharing of resources and knowledge among the verification projects, thus resulting in a significant increase in the efficiency of verification and in an associated reduction in the time required to verify a new design.

Introduction

The main goal of functional verification is to ensure that the design implementation matches the specified architectural behaviour. A powerful methodology is mandatory to be able to cope with the challenge imposed by the enormous state space of a processor. The challenge is increased further when the methodology needs to be able to cope with verification of several processor families, especially when they are based on completely different architectures. Today, functional verification is the main parameter in achieving time to market with fully functional silicon.

Common verification practices include both simulation and formal methods [10], but simulation and test generation are still the main vehicles for verifying the design. In this paper we will focus on simulation based verification.

The core of efficient simulation-based functional verification is a powerful test-generator [6]. Perhaps the most significant capability required of such a test-generator is the ability to combine randomness with directed generation. Ideally we would like to be able to create a specific scenario that is of interest (for example an instruction that results in overflow), while leaving part of the scenario up to randomness (for example what causes the overflow, what registers are used, etc.). We would like to be able to capture this scenario in a definition file, so that successive runs of the file will produce different test files, each one of which recreates the desired scenario.

When verifying several different processors, additional capabilities are required of the test-generator. The most basic requirement is that the test-generator must be generic, one that is flexible enough to accommodate the peculiarities of all the designs. Furthermore, the verification process involves building a reservoir of knowledge which can be shared among different designs.

The core of the simulation based methodology described in this paper is Genesys[8]. Genesys is a model-based, directed-random dynamic test-generator. It enables the user to ally randomness and control for generating a virtually infinite number of high-quality tests. It is a generic tool which has been employed in the verification of over 30 various designs.

STMicroelectronics Processor Families

STMicroelectronics is a semiconductor supplier with a product line ranging from small discrete devices to complex system on chip solutions. Within only a few years a number of processors have been functionally verified using the Genesys functional test generator inside ST: The 64 bit Chameleon (CM10) multimedia processor (a research project into building a complex system on a chip [1]), the ST100 DSP core family [2], the SH5 designed by SuperH (started as a joint development between ST and Hitachi [3]), the ST200 VLIW processor joint development between ST and HP [4] and the STM7 micro-controller.

Verification Plan

The starting point for functionally verifying a processor is a comprehensive top level verification plan. A typical plan for simulation based verification specifies the test code or stimuli that need to be run on the processor. There are three requirements that apply to the test code:

- It should cover all features of the Design Under Test (DUT)
- It should be easy to create.
- It should be easy to debug.
The first type of tests to be run on the DUT are Architecture Verification Plan (AVP) tests. These tests aim to verify all architectural features of the DUT. Each such test typically consists of a short stream of one particular instruction, focusing on the architectural corner cases of that instruction. The test specification is usually deterministic, which allows easy creation and debug. These tests should find relatively simple bugs on the individual instruction level.

The second type of tests that are run are Implementation Verification Plan (IVP) tests. These tests target all implementation specific features of the DUT such as pipelines, caches, etc. These tests are more complex to write and debug than the AVP tests, as they consist of specific interdependent streams of instructions. The IVP tests will uncover more complex bugs related to implementation specific features of the design.

The third type of tests are the directed Random Verification Plan tests (RVP). The RVP tests are aimed at finding bugs that have been missed by the AVP and IVP tests. They target areas that are bug sensitive, using experience from previous projects. The RVP is extended with more test specifications during the verification process using feedback from code coverage results and information about bugs that are found. Experience shows that bug fixes tend to introduce new bugs or expose existing bugs.

The fourth and final type of test code consists of legacy code and of application code such as operating systems etc., but creation of this code is outside the scope of this paper.

**Test Plan Implementation Requirements**

The three different test plans need different test implementation techniques, ranging from a completely deterministic approach for the AVP tests to a directed random approach for the RVP tests. Furthermore, we need to be able to determine the pass or fail status of each test. For AVP tests, a self checking technique where the result is known beforehand can be used but this is not possible for RVP tests that may contain complex sequences of floating-point instructions. For those tests, the DUT results need to be compared with results from a reference model to determine a pass or fail status.

Ideally, one tool should be used to implement all test plans to allow reuse of test code and to reduce overheads such as tool maintenance, etc.

**The Genesys Test Generation Tool**

Genesys is a directed-random test-program generator that was developed at the IBM Haifa Research Lab and is based on the Model Based Test Generator [5,6,7,8]. Genesys enables the creation of test programs ranging from completely deterministic to totally random. The system consists of five basic interacting components:

1. The generic, architecture oblivious test generator which is the engine of the system.
2. The knowledge base which holds a formal description of the targeted architecture and allows the incorporation of complex testing knowledge. This knowledge base includes heuristics which represent cumulative verification engineer expertise.
3. The instruction set simulator which serves as a reference platform.
4. The storage control unit (SCU) which contains the architecture specific information regarding the memory management.
5. The user interface which enables the Genesys user to control the test generator.

The genericity of its engine enables application to a wide range of different architectures, with minimum effort. Table 1 lists the various processor types verified with Genesys.

**Table 1. Designs Using Genesys for Functional Verification**

<table>
<thead>
<tr>
<th>Type</th>
<th>ST</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC</td>
<td>CM10</td>
<td>PowerPC (11 designs)</td>
</tr>
<tr>
<td></td>
<td>SH5</td>
<td>RS6000 (10 designs)</td>
</tr>
<tr>
<td>VLIW</td>
<td>ST100 (2 designs)</td>
<td>ST200</td>
</tr>
<tr>
<td>CISC</td>
<td>STM7</td>
<td>x86 (4 designs)</td>
</tr>
<tr>
<td>DSP</td>
<td>ST100 (2 designs)</td>
<td>3 designs</td>
</tr>
<tr>
<td>Embedded controller</td>
<td>STM7</td>
<td>5 designs</td>
</tr>
<tr>
<td>Cache controller</td>
<td></td>
<td>5 designs</td>
</tr>
</tbody>
</table>

The main output of the generator is a test file which consists of a sequence of instructions, starting from a given initial architectural state, and a section describing the architectural state at the end of the test which forms the expected results. These expected results are used for comparison against the results from running the test on the RTL to determine if a test passes or fails.

We will describe the Generation Engine and the Reference Simulator in more detail in the next two sections. In the next chapter, we will discuss the implementation of the Knowledge Base and SCU components for the ST designs.

**The Generation Engine**

Genesys is primarily an instruction driven test generator. This means that the whole test generation is based upon the selection of the instructions which will appear in the test. The generation scheme is such that Genesys first selects an instruction and only then attempts to fulfill the existing biasing...
requests (i.e., the generation of some event). The instruction selection is controlled by the following biasing directives:

- **Instruction selection:** There are several options for selecting instructions from those which are available: each one separately, as a set (either predefined in the database or built by the user), as a format or using a macro (which is a user defined sequence of instructions). Another option is enumeration over a set of instructions, which is a feature that is especially useful for coverage purposes.
- **Policy order:** The relative ordering of the instructions may be random, ordered or cyclic.
- **Control on the instructions which will appear in the not-taken leg of a branch instruction.**
- **Loops:** Specific control is provided in order to control the generation of loops.

Beyond the instruction selection, Genesys offers control over the resources such as registers and memory locations which will participate as input and output of the instructions. It is important to note that control is given on two distinct scopes: *global* and *specific* (per instruction). In all the biasing directives varying degrees of randomness are allowed. For example, in address selection, the level of biasing may vary from high level specification such as stipulating the degree of alignment, boundary and crossing relative to any standard storage size (i.e. word or page). At the other extreme of low level biasing one may control specific bits of the address. The list below summarises the different controls provided:

- **Resource initialization:** The user may initialize any resource defined by the architecture (registers, memory, initial address). During the test generation resource reloading is also supported[11].
- **Resource selection policy:** Addresses and registers can be selected according to multiple policies, differing mainly in their relative reuse of the resources.
- **Field biasing:** The instruction fields - their content and occurrence - are under complete user control.
- **Address biasing:** The user has comprehensive control over the addresses which will be accessed during the test. This includes both the effective addresses and physical memory. Genesys supports address translation, allowing the user extended control on generation of all possible events which may occur during translation. This includes events such as aliasing and all types of exceptions. The provided support is of course design dependent, as address translation mechanisms may differ greatly among architectures.
- **Cache usage biasing:** Genesys provides biasing directives toward all the main cache events (Hit, Miss, Replace, Transfer-in, Transfer-out) using a simple cache model, where addresses reach the cache in the order they appear in the instruction stream. Multiple caches both for data and instructions and including hierarchical relationships between them, are supported. They are defined through the knowledge base along with their particular parameters.
- **Symbols:** Symbols can be specified instead of any resource, providing a means for conveniently creating dependencies between resources, without having to specify their exact identity.
- **Testing knowledge (TK):** Generation and validation functions can be incrementally added into the database to bias the generation of data for any resource. The TK is written in C and can make use of a large library of predefined functions for the generation of interesting values and random events. Generation functions are designed to generate specific corner cases, whose probability of occurring would otherwise be negligible. Validation functions do not generate data but verify that either the data or the instruction meets certain requirements. This is useful for avoiding generating undefined behaviour and helps control the generation, for example not allowing exceptions to be generated. TK can be reused for different test generation systems.

The user has complete control over the probabilities of achieving the events mentioned above: always, never and any level in between, providing generation of tests ranging from fully deterministic to completely random.

**The Reference Simulator**

An architecture specific reference simulator is connected to Genesys so that after an instruction is generated, it is executed by the simulator. This allows the generator to take the current state of the simulator into account for the generation of the next instruction. In case the simulator state after an executed instruction is undesirable, then the last instruction is rejected and the state of the simulator is reverted to the state it was in prior to the rejected instruction. These mechanisms allow complex dependencies in a test, such as conditional branching depending on the result of a floating-point operation, while making sure that the test does not cause architecturally undefined behaviour. This means that Genesys will not create illegal tests.

The reference simulator also allows Genesys to include the reference architectural state at the end of the test, which is used to check against the results of the test case running on the DUT.

**Applying Genesys to ST Architectures**

Key to using Genesys in ST is its efficiency in finding bugs while reducing verification time scales and the ability to quickly deploy the tool to new projects. This is possible due to the flexible nature of the knowledge base and the way this knowledge base is implemented by ST.
Building the Knowledge Base

The knowledge base holds a description of the targeted architecture and consists of three parts: the modelling of the architectural resources, the instruction set which includes the encoding and resources used by the instructions, and the testing knowledge which contains information about the corner cases in the architecture. The knowledge base is traditionally modelled by hand using a GUI, but ST has automated this process. The automatic knowledge base population system takes a simple description of the architecture instructions, and converts this into the database which is read by Genesys, by use of a series of scripts. The description of the instructions is generally available as it forms part of the simulator implementation code.

By reusing a formal description of the architecture that is used to build the reference simulators, ST’s automated population flow is capable of modelling between 70% and 90% of the knowledge base. For each processor, there remain relatively few features that must be modelled by hand.

Testing Knowledge

ST has accumulated thousands of lines of testing knowledge for various types of instructions of the various processors, including a large floating point library originally developed by IBM. Most of the testing knowledge is reusable between test generators even for quite different architectures. The cumulative testing knowledge becomes more advanced and mature for every new architecture, so that less time is spent writing new testing knowledge for each successive architecture.

Modelling of Advanced Architectural Mechanisms

Genesys contains functionality to test mechanisms which are standard in most architectures, such as branch prediction, caches, and pipelining. However each processor family also has a set of unique architectural and micro-architectural mechanisms. These features need to be modelled in the knowledge base. Genesys allows such mechanisms to be modelled in the testing knowledge, or in external libraries. One such library is the SCU (Storage Control Unit) library which models the MMU (Memory Management Unit) of the DUT.

Under the control of Genesys, the SCU selects values for the registers which configure the MMU of the processor. Based on the biasing that the user has selected, the SCU performs the correct initialisations aimed so that memory accesses made in the test stress the design, for example by hitting memory boundaries or by causing read/write/execute protection exceptions. Although the SH5 and ST200 have very different types of memory management units, they can all be modelled in the SCU.

An increasingly common feature of modern processors is bundling instructions into a very long instruction word (VLIW), as used in the ST100 and ST200 processors. The constraints that arise in a VLIW processor due to the architecturally visible latencies can be implemented in Genesys using validation functions.

Many processors support more than one mode of operation, each mode having a different instruction length, for example a 16-bit encoding for compact code and a 32-bit encoding for a powerful instruction set. Being able to switch between the instruction sets is often an error prone area and is thus a very important feature to target with test generation. Both the SH5 and ST100 processors have multiple instruction sets. The STM7 processor uses a CISC type variable length instruction set.

Processor architectures evolve over time, both during the design implementation and also for new versions of a processor in a family. The flexibility and reusability resulting from the use of the knowledge base allow quick adjustment of the testing to the evolving design.

Strategy and Efficiency of Test Implementation

A good test implementation strategy is necessary to implement the verification plans within the projects’ time scales while ensuring the test cases are efficient in finding bugs.

Test Generation Strategy

Implementing the AVP and IVP verification plans by hand-writing assembler test programs is very time consuming, as every variation of a test needs to be hand coded. It is more efficient to generate the tests with a test generator such as Genesys, introducing a highly controlled degree of randomness. It allows the tests to focus on a mechanism or corner case while randomising other behaviour. With this methodology, writing one test definition for each corner case is sufficient to generate a set of tests. This set will implement the same part of the plan as all the particular handwritten equivalents.

Take for example an instruction pipeline forwarding mechanism where forwarding is possible to more than one stage of the pipeline. When handwriting assembler tests, a test for forwarding to every possible pipeline stage needs to be written. When using focused random generation, a random set of instructions can be specified while enabling the register reuse option. If enough tests are generated from this single test definition, then all possible forwarding mechanisms will be tested.

Efficiency in Finding Bugs

Apart from a gain in test plan implementation time, the contribution of the randomness in focused random tests often triggers a bug that would not have been found by the deterministic methodology, as the randomisation often triggers cases that are on the boundary of the test specification or just outside it. In the example of the previous paragraph, the test plan normally specifies a specific stream of instructions which test the forwarding mechanism. Experience shows that
forwarding mechanisms often work for some instructions but not for all.

The focused random methodology is very likely to find problems like this, as random combinations of instructions are used for testing the forwarding mechanism. Handwritten assembler tests, on the other hand, only tend to test the mechanism with the specific stream of instructions specified by the test plan and may miss the combination of instructions that triggers a bug.

**Test Generation Feedback Path**

Even focused random test generation can be ineffective if not controlled properly. The AVP, IVP and RVP together should cover all functionality in the design. However, a feedback path is required to understand the quality of the plans and how well they have been implemented. An example of a feedback tool is code coverage. Code coverage measures the amount of RTL code that is being exercised by the tests, without understanding the functionality of the code. Coverage of the functionality of the design can be measured by assertions and functional coverage tools [9].

### Results

One of the aims of the verification plans is to ensure the maturity of a design as soon as possible. The maturity is generally measured by the percentage of failing test cases. To reduce this percentage, bugs should be fixed as quickly as possible. Based on our experience, an indication of the number of instructions used to implement the various test plans, and the proportion of bugs found by each plan can be seen in Table 2.

**Table 2. Efficiency of Tests**

<table>
<thead>
<tr>
<th></th>
<th>AVP</th>
<th>IVP</th>
<th>RVP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of instructions</td>
<td>0.5-2M</td>
<td>5-10M</td>
<td>200-500M</td>
</tr>
<tr>
<td>Percentage of bugs found</td>
<td>25%</td>
<td>60%</td>
<td>15%</td>
</tr>
</tbody>
</table>

If the verification process consisted of only RVP testing, then the same bugs would most probably have been found, but the complexity of the RVP tests is so much higher that the time taken to debug them would be much greater. The power of the RVP tests lies in the obscurity of the bugs that they find. These more obscure bugs are missed by AVP and IVP tests.

**SH5 Results**

Figure 1 presents the distribution of bugs found during the final phase of the SH5 core verification.

**ST100 Results**

In the verification process of the ST100 DSP core family around 80% of the bugs were found using test generation. Genesys has been used for all test implementations including AVP and IVP tests. As an example of the distribution of bug discovery using other methods, we bring the ST122 core. In the process of its verification, 13% of the bugs were discovered while running hand-written tests, 4% by use of property checking, and the remaining 3% were found while running code on the emulator and while activating the software tool-chain.

**Formal vs. Simulation Based Verification**

Different verification methodologies are complementary rather than competitive. The type of bugs they tend to find are completely different. Formal verification can only be used effectively at block level, since it is limited by circuit complexity, whereas simulation based verification has no such limitation. Formal verification tends to find bugs that are very unlikely to be discovered (in the test bench set-up). Simulation based verification tends to find bugs in areas that are too complex or not obvious for formal tools to hit.
Summary

The effective and flexible approach to functional verification described in this paper has been used by STMicroelectronics to verify a number of diverse processors within a couple of years. The methodologies and ideas described in this work are platform independent in the sense that they can be applied to a variety of verification environments. The test generator which was chosen is Genesys. In addition to supplying strong generation capabilities, Genesys is generic and provides a high level modelling language. The knowledge base that it uses includes both a formal description of the targeted architecture, and testing knowledge.

The effectiveness of this methodology is measured in terms of the proportion of bugs found using different techniques and in terms of a reduction in the verification time. We have observed that the introduction of the methodology described above, combined with the use of Genesys, has resulted in a significant increase in the number of bugs detected by use of automatic test generation, as opposed to other verification methods, including handwritten tests and formal methods.

To achieve short system bring up times and to reduce the verification plan implementation time, an automatic knowledge base implementation flow has been constructed. Almost all the knowledge base population is performed directly from a formal, high-level description of the architectural resources and instruction set. In addition, generic testing knowledge has been developed, so that only a few architecture-specific features require manual implementation. This flow has drastically cut the verification time, enabling a small team of only four members to support all the projects specified above simultaneously.

We have shown how Genesys can quickly be adapted for different architectures, and how reusing testing knowledge and knowledge base modelling automation allow the verification team to meet tight project time scales. Furthermore, with the right test implementation strategy, the test implementation time can be reduced while the quality of the tests can be improved compared to manual test writing. Finally we’ve shown some real life results. For the final stage of the SH5 project and the two ST100 designs, most bugs were found with Genesys.

References

2. S. Palmieri, “Doubling Up. Innovative Processor Core for Embedded Applications Combines DSP and Micro-Processor Features (ST100),” New Electronics, 6/13/00, p.29-30