gNOSIS: Mining FPGAs for Verification

Mehrdad Majzoobi, Richard Neil Pittman, Alessandro Forin

Microsoft Research

August 2011

Technical Report

MSR-TR-2011-141

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

gNOSIS: Mining FPGAs for Verification

Mehrdad Majzoobi, Richard Neil Pittman, Alessandro Forin

Microsoft Research

Abstract

Hardware verification on FPGAs runs more than three orders of magnitude faster than software simulations, however with much lower visibility into the design under test. To expedite the task of debugging and specification verification, we propose a tool framework that automates many tedious aspects of the process. We provide tools to mine assertions either from simulation or hardware traces, to generate assertion checking engines implemented as efficient Verilog state machines, to rewrite the user's Verilog code inserting probes to the relevant signals, and to dynamically vary the operating clock frequency of the design under test. During implementation, we ensure that the layout of the original design is preserved as much as possible by automatically generating placement constraints, and thereby minimizing the uncertainty introduced by other on-chip debugging techniques.

This continues the work of the gNOSIS project to explore new techniques for FPGA verification and debugging by leveraging features of the FPGA chips and tools. This work draws upon previous works in specification mining and synthesis and combines it with new techniques for automated verification in hardware.

1 Introduction

It is critical to validate the correctness of any system prior to in-field utilization. Any errors during operation may lead to fatal or disastrous financial outcomes. For instance, in 1995, an error in the FDIV (floating-point division) instruction of an early Intel Pentium processor [5] resulted in a charge to Intel of approximately $500M. NASA lost a 125 million Mars orbiter in 1999 loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models.

Checking the correctness of the underlying system versus the intended functionality and the expected behavior is called verification. In many fields of engineering, the designed system has to be verified under various inputs and operating conditions. In integrated circuit and hardware design, the debugging and verification becomes even more challenging. Unlike software that runs on fixed processor architecture, every hardware design can require a different circuit structure and architecture, making it more difficult to provide a one-size fits-all solution. In addition, signals are often buried under the device package and devices typically have limited access I/O pins. As a result, the designer will be faced with limited observability and controllability of the system under test. Furthermore, hardware synthesis and mapping tools often require a much longer time to recompile the system as opposed to software design. It is desired to avoid recompiling the whole system as much as possible. Last but not least, the increasing complexity of systems is adding an extra level of effort in verification and debugging.

The state-of-the-art in hardware debugging and verification can be divided into three main categories. The first category is the simulation-based methods, which involves simulating the behavior of the hardware under design. The simulation requires writing test benches and input test vectors. Self-checking test benches and transaction based test benches are common ways of simulating the hardware. The second category is based on monitoring the design behavior during runtime by probing pertinent signals using either integrated or external logic analyzer. Thirdly, Formal mathematical methods are used to verify the system behavior against an abstract mathematical model of the system often described using FSMs, labeled transition systems, Petri nets, timed automata, hybrid automata, and/or process algebra.

In this paper, we introduce an assertion-based software-hardware hybrid method to verify the system functionality. The introduced methodology attempts to pin point bugs mainly introduced post synthesis and mapping. The framework learns and mines assertions from signal traces obtained by functional simulations. The mined assertions are then combined with user-applied assertions. These assertion which are specified in a standard linear temporal logic (LTL) language, Property Specification Language (PSL), are then synthesized into Verilog finite-state machines (FSMs). Next the original system is fed into a probe-insertion tool to automatically provide interconnection between the verification units (VUNITs) containing the assertions checking state machines and signals to be monitored. The debugging interface communicates with the VUNIT assertion-checking engine through an Ethernet link. The user can control which assertion to be checked and check which assertion has failed at what clock cycle. The tool flow is shown in Figure 1.

2 Background

The goal of the gNOSIS project is to advance the state of the art in FPGA hardware debugging and verification. The gNOSIS project investigates new techniques that leverage features of the FPGA and manufacturer's tools that most hardware developers are not aware or do not regularly use.

2.1 State of the Art in Industry

Both major manufacturers of FPGAs provide on-chip infrastructure for debugging and verification on the hardware. These are Chipscope for Xilnix [7] and SignalTap for Altera [1]. Both these tools insert probes to signals into the design under test and capture their states using a sampling clock and storing them a buffer. When the limit of the buffer is reached or after a configurable number of samples have been taken, the contents of the buffer is transmitted to a host PC and displayed in logic analyzer software GUI to be analyzed by the testing engineer. These tools have some advantages and disadvantages.

Both these tools allow hardware engineers to debug their designs on the device in a way similar to the bench logic analyzer. Hardware engineers can select signals to monitor and view cycle by cycle traces of these signals in their design. Tools are provided to insert the infrastructure for this trace capture post synthesis at the netlists level. This relieves the task of modifying the design source for debugging and limits the chances of the hardware designer introducing new bugs in the process. The debugging infrastructure communicates to the host over boundary scan using the same JTAG cable used for configuring the device.

The logic for capturing, storing and sending the traces back the host are implemented using the

FPGA resources including logic, routing, and memory. This strategy consequently limits the length and number of the traces that can be captured and analyzed. The Xilinx Chipscope debugger limits the number of signals per logic analyzer to 256. This is adequate for most design needs. However more complex designs with wide data widths can quickly exceed this limit. The additional logic added to the design also changes how the design is implemented on the device. The debugging infrastructure increases the fanout of probed signals. In addition, it requires the design logic to be placed and routed differently. These changes to the design implementation can potentially alter the behavior being observed making replication of error conditions under observation more challenging.

Since the design under test is augmented with the debugging infrastructure post-synthesis and pre-implementation, the design implementation must be repeated to add the debugging infrastructure or to change it. These implementation steps on average take 10 min to several hours depending on the complexity of the design. This leads to a much longer debugging cycle as opposed to software debugging. In many cases it can require multiple debug cycles selecting different signals to trace and re-implement the design in order to observe the incorrect behavior responsible for the bug. For this reason, it can take several hours to diagnose and correct a single bug.

2.2 gNOSIS Project

The gNOSIS project explores new techniques for FPGA verification and debugging by leveraging features of the FPGA chips and tools. Using these manufacturer features the gNOSIS tools mine information about the behavior of a design from simulation and hardware. The gNOSIS tools are designed to increase the faculty available to hardware developers while minimizing the impact and potentially changing the behavior being observed.

The first gNOSIS tools attempted to address the issues of the existing manufacturer tools related to selection or number of signals that can be observed without re-implementing the design. These tools were able to observe the entire state of the design running on a FPGA using the CAPTURE and READBACK features of the FPGAs. The design tools provided the locations of the state of the registers stored in the FPGA configuration after a CAPTURE is asserted. The configuration memory was read using the Internal Configuration Access Port (ICAP). Only the logic required to assert the CAPTURE and read back the configuration memory are added. There were no probes inserted into the design itself. Thus, entire design visibility is achieved with minimal impact to the design under test that will not change as the traces of interest change. However, the bandwidth out of the ICAP severely limits the sampling rate of the traces of the design under test. A cycle by cycle observation was not possible.

This next generation of tools approaches the problem from a different direction by performing self-checking on the device using model data extracted from simulation traces. The tool flow summarized in Figure 1, is based on assertions mined from simulation traces synthesized into hardware cores.

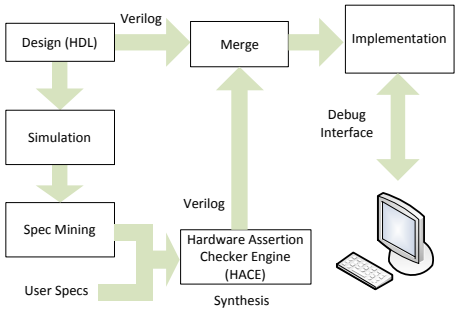

Figure 1. gNOSIS Assertion Based Tool Flow

3 Static Mining from Traces

3.1 Assertion Based Verification (ABV)

An assertion is a statement about a specific functional characteristic or property that is expected to hold for a design. Assertion-based verification increases observability in simulation while providing targets for formal verification, which increases controllability. Furthermore, assertions facilitate design reuse through self-checking code.

In assertion-based verification, RTL assertions are used to capture design intent in a verifiable form as the design is created. Assertions enhance observability coverage, making it easier to spot the source of an error. Debug time is greatly reduced in this way. Assertions improve controllability coverage with formal verification. When verifying assertions, formal verification algorithms explore the equivalent of billions of input patterns without requiring test vector creation.

Assertion-based verification is a multi-faceted approach to verifying a collection of partial specifications more efficiently. Assertions enable capabilities for both simulation and formal verification and, in the case of open standard assertions, a common method of capturing intimate design knowledge for both.

These assertion based methods can be implemented in the hardware device to allow for the design to perform self-checking at operational speeds in a semi-autonomous manner only sending back to the host incidence of assertion violations over a simpler and lower bandwidth interface. However, in order to use assertions to perform this self-checking, it is necessary to have assertions to test against the design under test. Unfortunately, most developers do not elect to write these assertion models when they design their modules.

3.2 Specification Mining

In principal, the assertions and specifications have to be written by the designer. However, typically because of tediousness of the task, and the fact the set of provided assertions may not provide a complete coverage, the process of property specification is often not properly done if it is done at all. In some cases the effort to write complete specifications is equivalent to the effort of implementation of the design. So why not implement the design and extract the specification as you verify the design. This is called specification mining.

The specification mining refers to the process of machine learning the specifications from circuit execution traces. In this approach, the specifications are generated automatically instead of being written by the designer. Mining specifications while debugging assumes there are likely bugs present in the design. Assertions mined from a design containing bugs will yield incorrect assertions. During the specification mining execution patterns that repeat with higher frequency in the signals traces represent the correct functionality. In other words, it is assumed that the abnormal behavior is represented by the outliers that don't appear as frequently. Thus bugs can be identified by these outliers and corrected.

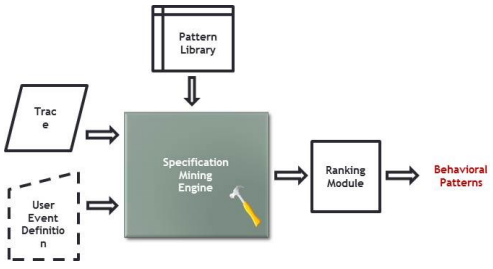

Figure 2. Specification Mining Flow

During specification mining, the system is simulated given a set of input vectors. The signal traces along with a set of patterns to look for are then fed to a mining machine, as shown in Figure 2. The patterns are in form of Linear Temporal Logic (LTL). The specification mining engine searches for occurrence of the specified patterns. The matching patterns are then combined into more concise properties by merging and chaining the logic expressions. In addition, the matched patterns are ranked based on the frequency they occur in the traces. The LTL expressions are described using Property Specification Language (PSL). The details of spec mining system are given in [6].

As development of the hardware system continues, the designers can test these assertions against each iteration of the design to ensure that design continues to meet the mined specifications. However, running traces against assertions in software is computationally intensive and time consuming. It is possible to quickly and more efficiently verify a subset of assertions at operational speeds on the hardware device.

4 Dynamic Mining from Probes

4.1 Specification Synthesis

After a hardware designer has performed specification mining on a design under test, there are a set of assertions expressed in Linear Temporal Logic (LTL) associated with the design. These can be analyzed against simulation and hardware traces in software tools but this requires a significant amount of time and computational resource. This is especially true when compared to the run time of the implemented hardware system. Testing these assertions against the hardware system at operational speeds would be several orders of magnitude faster and with higher fidelity. However, while software tools can analyze traces and evaluate assertions in their LTL state, to perform run-time verification during system execution, the extracted properties and the ones specified by the user must be synthesized into a hardware core or verification unit (VUNIT).

The VUNIT performs assertion checking on the design at speed with the system clock. The process of synthesizing the properties involves a transformation from LTL into a finite state automata/machine (FSM), and each atomic expression is written in combinational logic. The FSM and the combinational logic are both written in Verilog HDL. Efficient synthesis of the LTL properties is tackled in [2,3].

The synthesis procedure is based upon the idea of rewriting the properties given an input value. After rewriting the properties under a large enough number scenarios and inputs, a transition chain as shown in Figure 3 can be constructed:

Figure 3. Transition Chain generated from LTL assertion.

The transition chain represents the captured behavior of the design under tests as a series of states that can be translated into a FSM. This can make for a very large state space and can result in large hardware blocks. For very large chains the same state may appear multiple times. By combining these into the state, the chain can be wrapped into a more compact FSM, such as that represented by Figure 4.

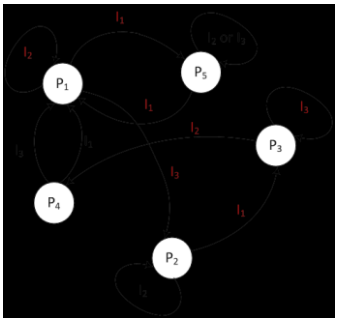

Figure 4. FSM of LTL assertion after state optimization.

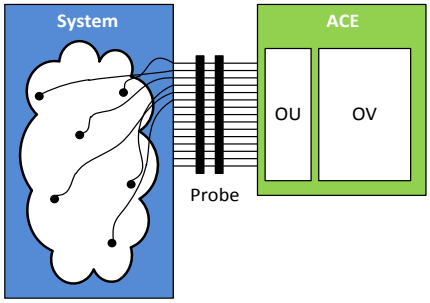

Figure 5. Probing signals within the design hierarchy.

For each assertion, an observation unit (OU) and verification unit (VU) are generated in Verilog HDL. The OU combines the traces of the probed signals and tests for the occurrence of transition events. The VU contains the FSM that encodes the behavior represented by the assertion from which it was generated. The VU transitions through the states of the asserted behavior. If the OU identifies an event that is not allowed by the assertion in the VU, a violation is triggered. If not the VU continues to transition normally.

4.2 Automated Probe Insertion

The assertions being used in this self-checking verification system are based on the interaction of signals in the design captured from simulation traces. The specification mining tools have access to all the signals in these traces in a dump file regardless of their location in the design. However, connecting to all these signals to probe them and connect them to their assertion checking cores is not as simple. Some signals that are ports of the top level modules are easy to access but others may be buried deeper in the design hierarchy.

Current industry tools for hardware debugging attach probes to the signals post synthesis at the netlists level. This prevents the hardware designer from having to alter his code but the synthesis process performs some optimizations that result in signal combination, signal loss and some name changing. This makes probing some signals difficult. Since signal name resolution from synthesis is beyond the scope of this experiment, it was decided to insert assertion checking cores and probe connections in modified HDL source.

However, previously it was expressed that manually modifying code to add debugging infrastructure was tedious and prone to error. For this reason, we introduce a tool for parsing the system Verilog code and prepare it for hardware assertion checking using the assertion based VUNIT cores. The tool written in Python parses the Verilog design source searching for signals in the hierarchy from a list provided by the hardware designer. The tool then rewrites the Verilog sources and automatically inserts wire and extends module ports to route these signals to the top level model, represented by Figure 6. This way they can be accessed through the interface bus and connected to the OU of the VUNIT.

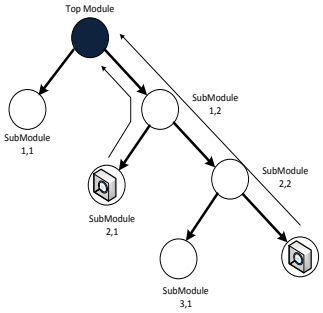

Figure 6. Passing internal signals to top level for probing.

4.3 Placement Enforcement

Using the VUNIT, it is possible to verify the behavior of the design under test at operating speed with minimal human interaction. However this still suffers from the same problems the manufacture debugging tools do as a result of inserting probes and additional logic into the design. This is undesirable because in some cases such as timing, placement and routing it can change the behavior of the design in subtle ways. To address this, when the design is re-implemented with the VUNITs and probes inserted, it will be constrained to be implemented as close to the original design implementation as possible.

To achieve this, we developed a script written in Python that extracts the original placement of the design from Xilinx Description Language (XDL) file generated during the implementation of the system-on-chip. XDL is a proprietary Xilinx low level netlist format. There is no official documentation for XDL but it is fairly easy to understand its syntax by just looking at a sample XDL file. Also the XDL file contains comments that provide syntactical information as well. The extracted placement is written as constraints in a user constraint file (UCF). This UCF is provided as input to the tool flow when the design is re-implemented with the VUNIT and probes. Using these constraints the placement is enforced to follow the same behavior as in the original design.

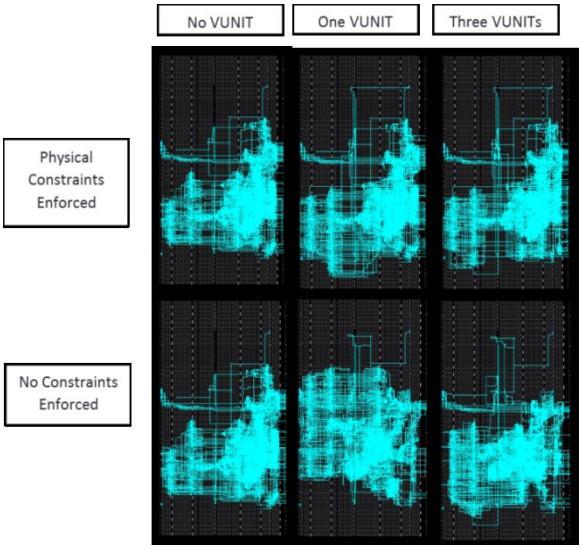

Figure 7. Placement of SIRC with different quantities of assertion cores (VUNIT).

Figure 7 shows the placement of SIRC (Simple Interface for Reconfigurable Computing) circuit before and after enforcing physical constraints with zero, one, and three inserted VUNITS. As it is shown in Figure 7, the original placement is retained when the constraints are enforced on the top rows. However, the placements on the bottom row, where no constraints are enforced, follow a drastically different pattern.

Table 1. Delay along Critical Path

	0 vunit	1 vunit	3 vunits
PCF enforced	8.160ns/ 122.549MHz	8.664ns / 115.420MHz	8.242ns / 121.33MHz
No PCF enforced		7.852ns / 127.356MHz	7.985ns / 125.235Mhz

The delay of the critical path for each corresponding implementation of Figure 7 is shown in Table 1. As expected, the maximum

clock frequency is slightly reduced when constraints are applied.

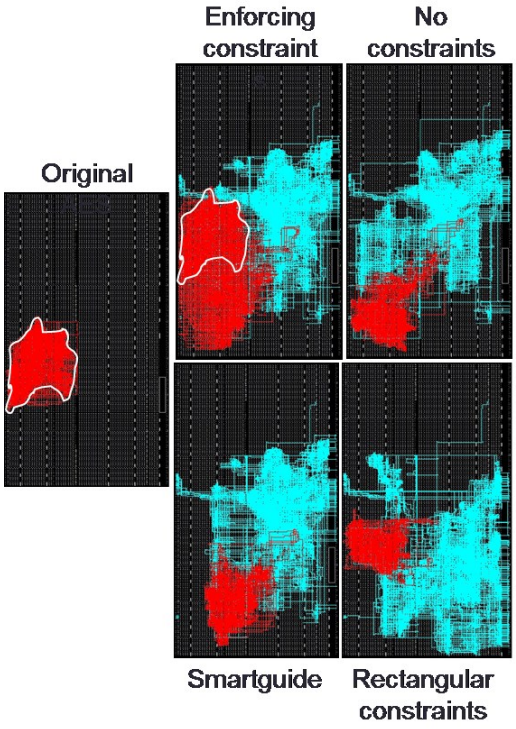

Figure 8. AES placement Strategies

The result of the same study on an AES core is shown in Figure 8. The leftmost plot shows the placement of a stand-alone AES core. The placement of the same AES module is shown when attached to the debugging SIRC interface along with 10 VUNITS. We study four different scenarios. In this first case, the placement constraints are extracted from the original run (module) and enforced on the second run (module+debug). Since the names of some components and nets change in the second implementation - due to randomized arbitrary naming done by the tool – some of the constraints are not properly enforced. As a result, these components don't follow the expected placement. An alternative solution is instead enforcing component wise placement, we extract the box coordinates that encompass the module and enforce the box area instead. In addition, Xilinx has its own "smartguide" proprietary algorithm to guide the placement. However, as it appears the algorithm is not highly effective here.

In addition, often it is needed to perform verification on individual module separately when dealing with system-on-chips. In this case as well, the designer would like to have the least possible impact on the module under test when verifying its functionality. Again, one of the important properties that need to be preserved for the module is its placement in the main system. Therefore, using this same setup it is possible to enforce its placement and routing of this module as if it was deployed in the actual system. Debugging infrastructure, including test stimulus and communication to host may be placed around the module under test.

5 Summary

This paper has presented a prototype framework for an automated self-checking hardware debugging infrastructure and tools for applying it to a design under test. A diagram of the flow is presented in Figure 9.

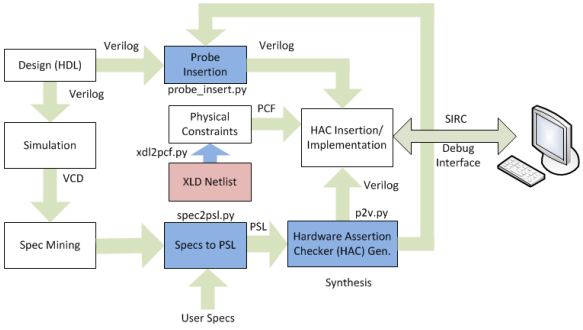

Figure 9. Detailed gNOSIS assertion based tool flow.

The hardware designer begins development of the design under test in a simulated

environment. As development progresses the hardware designer submits traces of the design to a specification mining tool to extract the behavior of the design. Infrequent events in the specifications are flagged as possible bugs and further investigated. When design and verification under simulation is complete, the design is implemented in hardware. At this point, the design should work correctly as it did in the simulator but as is often the case, there is some error causing behavior in the hardware implementation that did not appear in the simulation.

In this case, the placement of the original implementation is extracted and placement constraints are generated from it. A subset of the constraints used to verify the design in simulation is synthesized into VUNITs. The design Verilog sources are augmented by the probing tool to pass all the signals needed for the assertion checkers to the top level where they are connected. The design is re-implemented with the VUNITs using the placement constraints to enforce the original implementation the manifested the incorrect behavior.

The SIRC interface was extended in order to allow the hardware designer to interface to the assertion checking cores implemented in the design under test. From SIRC, the hardware engineer could run his application in addition to activate and deactivate VUNITs and check for violations.

6 Software Controlled Clocking

When working to extract that last level of performance from a design it would desirable to be able to dial up and down clock frequency on the fly and without having to resynthesize and recompile the design. This could further used to perform timing tests and pinpoint timing violations. For this reason we propose using the dynamic reconfiguration port available on the Xilinx PLLs and modify the PLL attributes through the debugging interface. The Xilinx PLL is presented in Figure 10.

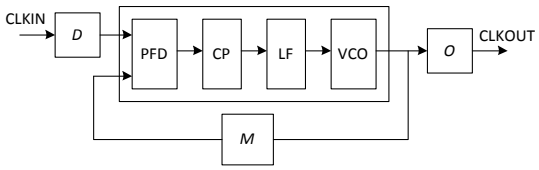

Figure 10. Virtex PLL Structure.

However, the PLL attributes D (divide), M (multiply), O (divide) cannot take any values. Their values are constrained by maximum and minimum phase/frequency detector input frequency as well as VCO output frequency. We have developed a tool that given a target clock frequency it searches for optimal PLL setting that achieves the closet output frequency. For instance, the plot in Figure 11 shows 11,000 unique frequencies that can be synthesized with one single PLL sorted in ascending order.

Using this tool to search for the optimum parameters for achieving a given frequency the hardware design could use debugging facilities provided in the system to adjust the clock frequency to the desired level. This functionality would further augment the amount and type of information available to hardware designers about their designs as they work to debug them and push for greater performance. At this time the hardware facility for this feature are still under development.

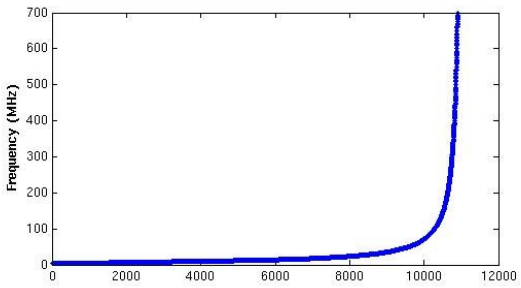

Figure 11. Possible Synthesizable Frequencies with a single PLL.

7 Future Work

The flowing list highlights possible revision, expansion, improvement of each subsystem and codes written:

- The conversion from specification mine tool output to Linear Temporal Logic (LTL) constructs is not completely in place. The main reason is that the output of the specification mine has changed in the latest version, which requires another parsing to be done.
- The automated connection between VUs and the design after probe insertion has still not been finalized. Ideally the names of the signals to be probed have to extract from LTL constructs, observation units (OU) have to be generated that connect the VUs to the design probes.
- Placement extraction tool can be improved by making it output geometrical boundaries for sub-module (to avoid the problem of component name change during Mapping).
- The placement tool currently only supports Virtex 5 families and it can be expanded to support Virtex 6 devices.
- Dynamic reconfiguration of the PLL attributes is still on the to-do list, however the algorithm that searches for the optimal PLL setting is fully developed and working.

8 References

1. Altera, Inc. "Design Debugging Using the SignalTap II Logic Analyzer." 2012. http://www.altera.com/literature/hb/qts/qts_qii53009.pdf
2. Grigore Rosu, Klaus Havelund. "Rewriting-Based Techniques for Runtime Verification." Journal of Automated Software Engineering. Vol. 12 (2). pp 151-197. 2005.
3. Hong Lu, Alessandro Forin. "Automatic Processor Customization for Zero-Overhead Online Software Verification." IEEE Transactions on VLSI. November 2008.
4. Md. Ashfaquzzman Khan, Richard Neil Pittman, Alessandro Forin. "gNOSIS: A Board-Level Debugging and Verification Tool." Proceedings of the IEEE Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 43-48. 2010.
5. V.R. Pratt. Anatomy of the Pentium Bug., TAPSOFT, V. 915. pp 97-107. 1995
6. Wenchao Li, Alessandro Forin, Sanjit A. Seshia. "Scalable Specification Mining for Verification and Diagnosis." Proceedings of the Design Automation Conference (DAC). pp. 755-760. June 2010.
7. Xilinx, Inc. "Chipscope Pro and the Serial I/O Toolkit." 2012. http://www.xilinx.com/tools/cspro.htm