gNOSIS: A Board-level Debugging and Verification Tool

Md. Ashfaquzzaman Khan
Boston University

Richard Neil Pittman
Microsoft Research

Alessandro Forin
Microsoft Research

July 2010

Technical Report
MSR-TR-2010-106

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

gNOSIS: A Board-level Debugging and Verification Tool

Md. Ashfaquzzaman Khan
Department of Electrical & Computer Engineering
Boston University
Boston, MA, USA
azkhan@bu.edu

Richard Neil Pittman, Alessandro Forin
Microsoft Research
Redmond, WA, USA
{pittman, sandrof}@microsoft.com

***Abstract*— It is notoriously hard to verify and debug the final, board-level implementation of FPGA designs. The task involves manual intervention and creativity, unpredictable time costs, and it is further complicated by side-effects of the monitoring circuits inserted into the Design Under Test (DUT). In this paper, we introduce gNOSIS, an automated tool for board-level debugging and verification of FPGA designs. gNOSIS uses the Capture/Readback features of the FPGA to checkpoint the entire state of the circuit with little or no modification to the DUT. The tool then correlates the design registers provided in the netlist with their state in the FPGA configuration memory, and with the expected state. If the states match, execution proceeds by restoring the state of the FPGA and continuing execution for a set number of cycles. When an error is encountered, the time and location of the error is reported and the last good checkpoint is used for further debugging. gNOSIS eliminates the manual labor and long wait times required by currently available tools (e.g. Chipscope). It provides much greater visibility at a lower cost. More importantly, it provides the required infrastructure for fully automated debugging using more intelligent offline tools.**

Keywords-FPGA; debugging; verification; board; circuit; netlist; state; checkpoint; automation; simulation; design; DUT;

I. INTRODUCTION

The process of verifying and debugging the final, board level implementation of FPGA designs has changed little since the advent of the first FPGAs. Most of the research has focused on the earlier steps, in pursuit of a correct-by-construction ideal tool flow [15]. Commercially available tools such as Xilinx Chipscope and Altera SignalTap [8, 9] today provide very limited visibility inside the circuit, and yet require hours of manual labor and wait time. Designers use their intuition to select the interesting signals to monitor, and the tool inserts a logic analyzer (ILA) and a communication circuit into the design. At runtime, the designer can select sophisticated trigger conditions for tracing, but based only on the selected signals. The captured trace data is saved in Block RAMs and the user can study it extensively using offline tools [10].

The core problem with these tools is that the designer intuition as to what signals are most relevant can be quite wrong. Indeed, if we knew *where* to look we would already have a pretty good idea of what the error is, which is the goal, not the assumption here. This circularity in the approach leads to a time-consuming, unbounded cycle in the debugging process. Since the tools provide no information about which signals to select for probing, the designers must rely exclusively on their experience and good fortune. Further complications are due to the additional components added by these tools into the DUT. These components vary with design and choice of signals and they cause variant changes in timing/placing/routing and make it difficult, sometimes impossible, to find bugs that are placing/routing dependent.

Previous work on this debugging problem can be divided into two categories: Scan-chain based and Readback based. The former inserts additional circuit elements, either by hand or by some tool, to implement a scan-chain inside the user design [1, 2, 6]. Although this facilitates viewing the entire state of the circuit, the area penalty and change in timing behavior is significant.

The latter uses the Readback feature provided by some FPGA vendors [3, 4, 7]. This feature allows reading back the entire configuration bitstream of a design out of the FPGA [11]. We follow this latter approach in our own work. By capturing the state values before reading back, it is possible to retrieve the entire state of the circuit at a given time. The BYU work uses this feature, but requires the designer to use JHDL [3].

Here we introduce gNOSIS, an automated board-level debugging/verification tool that allows the designers to see the entire state of the circuit and provides precise information about the timing and location of bugs. gNOSIS correlates the design registers provided in the netlist to their locations in the configuration memory and the simulated instance. It then uses the Capture/Readback features of the FPGA to retrieve the entire state of the circuit after a given number of cycles. The retrieved values are then used to verify the behavior of the circuit, currently by comparing against simulation. A complete match is saved as a checkpoint, allowing verification to continue. Any mismatch is reported along with its precise location and time, thus providing the designer with a great entry point into the board-level debugging problem. The verification by gNOSIS is automatic and the user does not have to worry about which signals to probe. The user does not have to stay in front of the computer, gNOSIS runs unsupervised and for as long as necessary to find the error(s). We are not aware of any other tool that provides this much visibility to the developer without severely impacting the DUT.

The improved visibility of gNOSIS also opens the doors to smarter debugging techniques. For example, we can insert assertion checking circuits in the design [17] and when they

trigger, let gNOSIS retrieve/analyze data to find the reason. Although the current version of gNOSIS uses simulation results for verification, it can use any kind of checks/analysis in software, since it has access to the entire state of the circuit. We can also use gNOSIS to run a design to a certain point in the hardware, then checkpoint and resume execution in simulation. This is especially helpful when the simulation time required to reach our point of interest is prohibitively long.

In contrast to the previous works, gNOSIS does not require any change in the design flow and does not change the DUT significantly. In the following sections we will describe how our tool uses the input/output of existing tools (Xilinx ISE [9], Mentor Graphics Modelsim [12]) to realize such automatic board-level verification with little or no change to the DUT.

II. GNOSIS OVERVIEW

The goal of gNOSIS is to provide hardware developers with a more complete way to debug and verify their board-level FPGA designs. More specifically, (1) the designers should be able to maintain their design flow and languages; (2) changes required to the DUT should be minimal or none; (3) the tool should tell the designers precisely where the error is; (4) unlike with Chipscope, the process should be automatic and designers should not have to stay in front of the computer during the entire verification process; (5) the tool should be extensible and allow for additional automation.

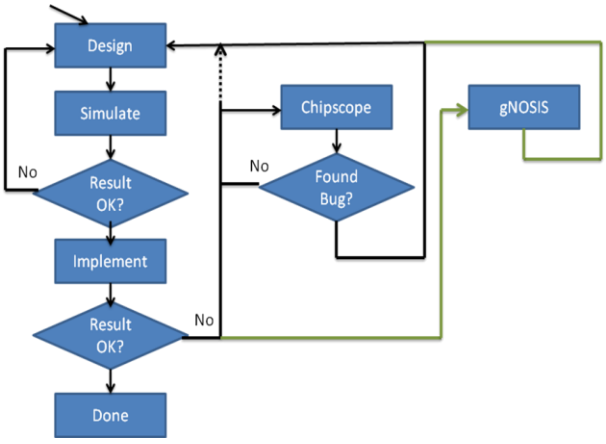

Figure 1. Design flow using gNOSIS.

The initial implementation of gNOSIS targets the Xilinx FPGA and Development Environment (ISE), Modelsim Simulator and Verilog HDL. As show in Figure 1, it only adds an optional path in the existing design flow. From the user's perspective, the work flow is as follows.

The user first designs in Verilog and inserts the (fixed) gNOSIS probe. Then the user generates the configuration data (bitstream) in a regular fashion, making sure that the following files are also generated: <design_name.ngc>, <design_name>.ll, and a post-synthesis simulation file. This can be done with a few additional mouse clicks. The .ngc file contains the netlist information for the synthesized design. The .ll file contains the location of the state variables in the bitstream [11]. The user then creates a testbench and performs simulation and dumps the simulation result in .vcd (Value Change Dump) file. The gNOSIS-specific part of the flow is as follows.

The user first feeds the bitstream, the .ll and the .ngc files to the gNOSIS preprocessor, which extracts the netlist information and maps it to the location information found in .ll file. The output of this step is a plain text file that contains FlipFlop (FF) names, their location in the bitstream and their initial values. In an ideal case, the .ll file should be sufficient, but we need to verify some information (see later).

The preprocessor also generates a help file that contains the signal names that must be dumped from simulation to compare with the board level implementation. The user may optionally choose to use this help file to reduce the size of the .vcd file from simulation.

The second step is to feed the bitstream file, post-synthesis simulation file, simulation result (.vcd file) and output of the pre-processor to the main program of the tool. The tool uses these inputs to repeatedly execute the design on the FPGA board and compare the result against the data from the .vcd file. If the current simulation is not long enough gNOSIS automatically performs a new simulation step. A successful match is saved as a checkpoint and execution proceeds. A mismatch is reported with its precise time (number of cycle) and place of occurrence. Once a mismatch is found, the most recent checkpoint is used to launch a Modelsim simulation instance for further debugging.

In addition to the debugging flow mentioned above, we can also use the tool to run a design to a certain point in the hardware then checkpoint and resume execution in simulation. This allows us to perform additional verification in simulation by reducing the time required to get to events of interests. This was especially useful for identifying a design error that only manifested itself after running the NetBSD operating system in multi-user mode on top of the eMIPS processor.

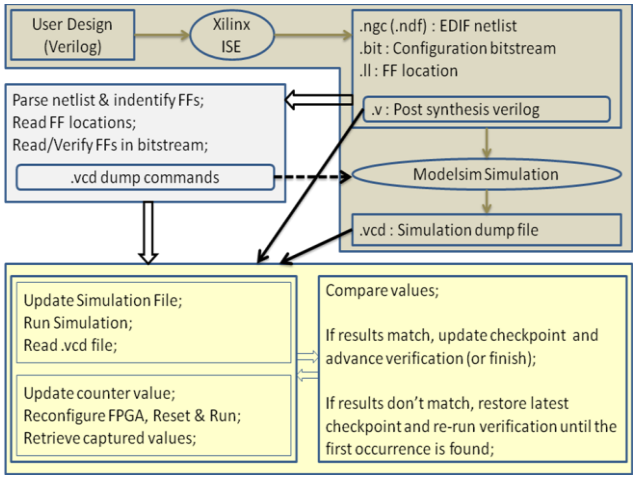

Figure 2. Overview of how gNOSIS works.

III. HARDWARE BLOCKS

The hardware portion gNOSIS consists of two parts, the Server/Monitor and the Probe inserted into the DUT. The Server/Monitor would ideally be implemented in a separate FPGA so that the DUT placing & routing is minimally impacted by the insertion of the probe. The probe itself only implements a counter, ICAP (internal configuration access port), capture and startup primitives. However, currently we have implemented both modules in a single FPGA (Xilinx Virtex 5: xc5vlx110t-2ff1136) to allow us to focus on the hardware/software interface and tool flow.

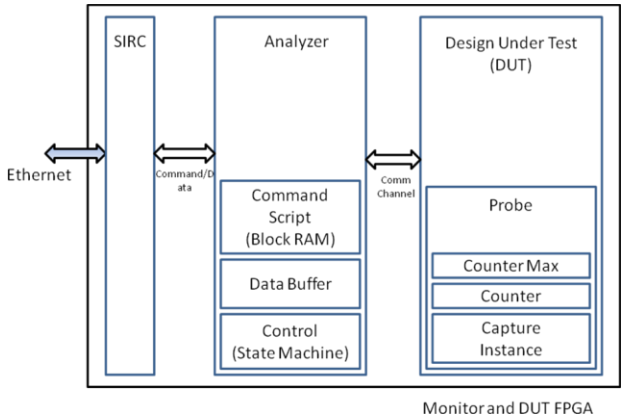

Figure 3. Current implementation for probing by gNOSIS.

Our implementation of the Server/Monitor circuit has two parts: (1) SIRC, the module that supports communication with PC through Ethernet [14], and (2) Analyzer, the module that implements the control for the Readback and allows the user to actually communicate with the DUT FPGA through the probe. The Analyzer supports writing batch commands from PC. The Analyzer stores command templates or sequences of configuration commands to perform actions such as read and write to configuration memory with fields such as starting address and word count that can be set by the control logic in order to execute a specified action. The Analyzer sends these commands over a communication channel to the Probe in the DUT.

The Probe facilitates capturing and reading back the state of the FPGA by instantiating a CAPTURE_VIRTEX5 block and an ICAP_VIRTEX5 block [11], since Xilinx tools don't provide this feature anymore [16]. There is an up-counter in the DUT clock domain which keeps track of the number of cycle that the DUT has run after RESET. When its value matches another set of registers, Counter_Max, the state of the FPGA is captured using the CAPTURE_VIRTEX5. By changing the value of Counter_Max, we can control when the capture occurs. Once the FPGA state is captured, it can be read back to PC through the ICAP_VIRTEX5 block.

The Probe receives commands from Analyzer through the communication channel. Those commands for the configuration logic and memory are passed to the ICAP while other bits in the command stream control the reset and clock state of the DUT. The user can, through the software interface, update Counter_Max, read back the captured configuration data, reset DUT and so on.

Figure 3 shows the block diagram of our implementation. For clarity, the internal connections are not shown. The ICAP_VIRTEX5 instance is also not shown. In this implementation, the communication channel is realized as a pair of FIFOs designed to isolate the logic and clocking of the Server/Monitor and the DUT/Probe. In future implementations the Server/Monitor and the DUT/Probe would be in separate FPGAs and the channel would be a physical channel between them, as shown in Figure 4.

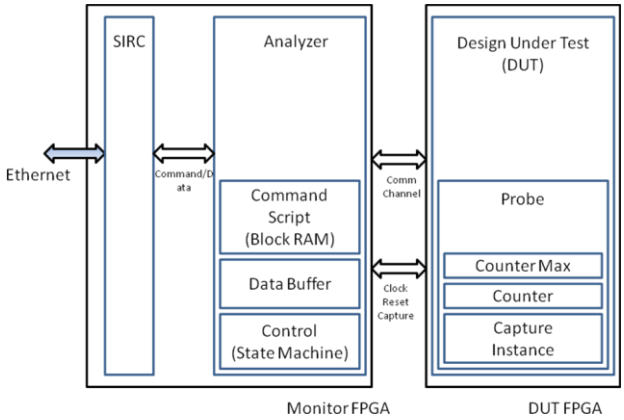

Figure 4. Ideal setup for probing by gNOSIS.

If we can send RESET/Capture signal to DUT, synchronize user clock and read configuration data back using ICAP, then the DUT can be entirely separated from gNOSIS by moving the larger components of the Probe to the Server/Monitor FPGA. The most significant of these is the large up-counter on the DUT clock that is used to control the timing of capture. In any case, the DUT FPGA will still have to accommodate CAPTURE_VIRTEX5 and ICAP_VIRTEX5 blocks. If the external pins are accessible, the ICAP could be replaced by the SELECTMAP interface [11], but the Capture has no such external interface except through a command to the SELECTMAP which is imprecise. The advantage of our design is that, unlike Chipscope, where configuration changes every time your signals of interest or trigger conditions change, the change required in the DUT will be uniform no matter what your design is or which signals you want to monitor.

To read back captured data from a Xilinx FPGA, a series of commands need to be sent to the ICAP. These commands include variables such as the start address, word count etc. The Analyzer implements this feature in the form of command-script. The user is allowed to write batch commands, through SIRC, to the Analyzer which saves those commands in Block RAM. The user then sends execute command, which executes the command in Block RAM. Value of the variables in the command scripts are inserted from registers, which also can be updated by the user at any point. This flexible construction enables an easy and compact implementation of the Readback feature.

IV. GNOSIS PRE-PROCESSING

gNOSIS pre-processing converts <design_name.ngc> to <design_name.ndf> using Xilinx utility 'ngc2edif'. The result is a netlist level description of the design in EDIF 2 0 0 format [13], which is then parsed. Next, the <design_name.ll> file is parsed to find FF location in the bitstream.

In an ideal case the parsing the .ll file should be sufficient. It is supposed to contain enough information to identify the FFs of the design and their positions in the bitstream. However, we have found that in some cases some flip-flops in the netlist are missing. If they are not present, the user needs to be aware of its impact on verification and vendors need to be informed of this as well.

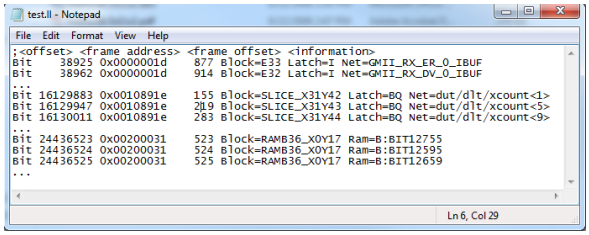

Figure 5. Example of a .ll file.

In addition, we also wanted to verify the frame addresses of the FFs in configuration data, since we will use them while reading back configuration data from the Probe using a limited size buffer.

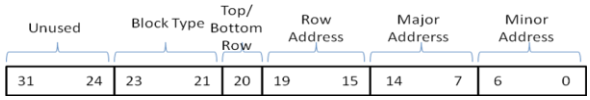

Figure 6. Cofigurtion Frame Address of Xilinx Virtex 5.

As shown in Figure 6, the frame address of a Xilinx device depends on that particular device and hence is not continuous [11]. While configuring a device using iMPACT, the frame address is set to 0 and all data is sent to the device at once. Frame address is increased inside the device automatically to make sure data goes to the right place. Although there's no documentation on how the address is incremented, by generating and studying the configuration data in debug mode, it is possible to retrieve that information [5]. That's how we determined the sequence of increase of frame address for our target device.

To be specific, in debug mode, data of every frame in the bitstream is preceded by its frame address. This can be found by searching for 0x30010001 commands in the bitstream. This command is followed by the frame address. For our target device, this was ultimately converted into an array of structures consisting of the discrete increment of addresses. Below is the example of first few entries of that array. The frame address starts from 0 and increases by 1 till 53. The next address is 128. Then it again increases by 1 till 163. The frame address after 163 is 256.

```
incrementalAdr incrAdrOriginal[] = {
    {53, 128}, // (for XC5VLX110t)
    {163, 256},
    {291, 384},
    ….
```

After verifying the FF existence and correct position in the bitstream, the list is sorted according to the ascending order of their absolute position in the bitstream. This makes it faster to read data from the FPGA later. The sorted and verified FF info is then printed for use in the main verification process.

In addition, a help file containing the essential signal names is generated so that the user may use it to reduce the size of the .vcd dump file from simulation.

V. VERIFICATION BY GNOSIS

The main program of gNOSIS is where actual verification takes place. It first reads the sorted FF info and maps them to the symbols used in the .vcd file. Signals in the .vcd file are given a small symbol of one or more characters to reduce the size of the file. gNOSIS then creates an instance of SIRC to start communication with the FPGA. It then runs the main verification loop of Figure 7.

```
While (curr_num_cycle <= MAX_NUM_CYCLE){
    Update timer info with curr_num_cycle (number of cycle after RESET, after which states will be captured);
    Reconfigure FPGA using iMPACT;
    DUT Reset/Run;

    Update simulation file and run simulation;
    Update values of FF from .vcd file;

    Wait long enough for capture to occur in FPGA;

    Read back configuration data from FPGA;
    Update values of FF from Readback data;

    Compare FF values from simulation and FPGA;
    if (PASSED){
        Update checkpoint and continue verification;
        Increase CUR_INTERVAL (optional);
        curr_num_cycle += CUR_INTERVAL;
    }
    Else{
        If curr_num_cycle was already 1 clock away from the latest checkpoint, then we have found the exact timing of error. Print all error information, invoke Modelsim and Exit;
        Otherwise, Rollback to the most recent checkpoint state and continue verification, but this time with finer interval (e.g. CUR_INTERVAL /= 2;)
        curr_num_cycle = latest checkpoint time + CUR_INTERVAL;
    }
}//end while
```

Figure 7. Main verification loop of gNOSIS.

A. Updating Timer & Capturing Data

The timer value (Counter_Max) is updated in the bitstream and loaded into the FPGA using iMPACT. A Reset signal, which also sets the counter in the DUT clock domain to zero, is then sent to the DUT. Once the reset is de-asserted, the DUT execution begins. When the counter reaches the timer value, capture signal is asserted and current state of the FPGA is saved in the configuration data. It is essential to make sure that the captured data is not overwritten (e.g. due to the counter completing another timer cycle) before it is read by the PC. This is done by using the "ONE_HOT" feature of the CAPTURE_VIRTEX5 and by using a guard register, which goes from '0' to '1' when the capture signal is asserted, and remains '1' until the DUT is reset again.

B. Running Modified Simulation and Retrieving State

The post-synthesis simulation file generated by ISE contains netlist level information of the DUT. By manipulating the type of FF used in the design, we can control how the circuit behaves when RESET is applied. For example: to reset a FF to 1, we change it to FDS; to reset a FF to 0, we change it to FDR. This is how gNOSIS updates the simulation file for any given point of time and performs simulation. We assume that RESET is performed only once in the beginning of the execution.

The simulation is run using the command line options of Modelsim and through some simple .do files which are executable scripts for the Modelsim simulator. The results are dumped in a .vcd file which is parsed to retrieve simulated values.

C. Reading Back Captured Data and Retrieving State

After updating/performing simulation, gNOSIS waits long enough for the capture to occur in FPGA. The exact wait time is calculated from timer and DUT frequency. The configuration data are then read back to PC through SIRC, Probe and ICAP. Every call to Probe is made with a frame address and number of frames to read consecutively from that address. Since the increase of frame address is not continuous all along, an array of addresses is maintained to handle the discrete increases. To increase the bandwidth of Readback, as many as 40 consecutive frames are read at a time when there are enough consecutive frames. This is also limited by the data buffer size in the Probe. Otherwise the number is limited to the maximum number of consecutive frames available from the current frame address.

The Readback data is saved in a binary file which is then read to retrieve the state of the FFs in the design. Not every word of the binary file is read. Instead, only the places of interest are directly read by manipulating the file pointer.

D. Comparing Simulation and Readback Values

After updating the data from simulation and board-level execution, the expected values from simulation are compared against observed values from board-level execution. If all the values match, we assume that the design has worked correctly up to current time. This state is then saved as checkpoint and the next timer value is decided. The increase in timer value can be of any precision, e.g. by 1 cycle, by a fixed number of cycles or by any other incremental value decided by the user to reduce verification time.

If there is a mismatch between expected and observed data, the state is restored to the latest checkpoint. If the checkpoint interval was only 1 cycle we have found the exact time when the error occurs. At this point gNOSIS reports all mismatches in a text file and invokes Modelsim simulation from the point of the most recent checkpoint.

E. Data Structure

Data structure to store FF data is a singly linked list of structure that basically contains FF name, net name, simulation value, board-level value, checkpoint value, position in bitstream and symbol in .vcd file. Checkpoint time and checkpoint file pointer position (for the .vcd file) is stored separately.

VI. Test Case and Performance

We have verified the functional correctness of gNOSIS using a simple counter circuit. The counter was simulated in Modelsim and executed on the FPGA in parallel. Values at different time points were compared by gNOSIS to verify the correct behavior. An error was introduced artificially by pressing a switch to disable the counter, to see if gNOSIS can catch it. In all cases, errors were successfully reported with the precise number of cycle and name and values of the incorrect registers.

There are three areas where performance can be compared with Chipscope or other approaches: (1) Area/Resource overhead, (2) Preparation time and (3) Usability.

The area/resource overhead is the increase in resource utilization compared to the original design. We found that with gNOSIS the overhead is small; it is constant for any design and for any number of signals of interest. The gNOSIS overhead is shown in Table III. In contrast, Table I and Table II show that the area overhead of Chipscope is large and depends on the design and signals of interest. Overhead data for Chipscope was collected using gNOSIS itself and an embedded processor design (eMIPS) which uses 18,268 slice registers and 21,237 slice LUTs of a Xilinx xc5vlx110t, at 26% and 30% utilization respectively. The area overhead for scan-chain based solutions also varies with design and can go up to 100% [1, 2, 6].

TABLE I. Overhead in Chipscope (eMIPS Example)

Signal Count (ILA x Signal)	Area		Time (Min)
	Registers (%)	*LUTs (%)*	
8 (1 x 8)	230 (1.3)	254 (1.2)	55
32 (1 x 32)	311 (1.7)	301 (1.4)	62
128 (1 x 128)	612 (3.4)	450 (2.1)	56
256 (2 x 128)	1203 (6.6)	846 (4.0)	59

TABLE II. OVERHEAD IN CHIPSCOPE (GNOSIS EXAMPLE)

Signal Count (ILA x Signal)	Area		Time(mm:ss)
	FFs (%)	*LUTs (%)*	
8 (1 x 8)	230 (7)	250 (6)	10:03
16 (1 x 16)	260 (8)	267 (8)	9:20
32 (1 x 32)	311 (10)	296 (7)	9:07
64 (1 x 64)	410 (13)	347 (8)	9:48
128 (1 x 128)	605 (19)	444 (10)	9:45
256 (2 x 128)	1182 (37)	394 (9)	12:26

TABLE III. OVERHEAD IN GNOSIS

Signal Count	Area (ICAP+capture+64 bit counter)		Time (Min)
	Registers	*LUTs*	
Any	201	160	0

In terms of preparation time, Chipscope must go through the implementation phase which includes translation, mapping, PAR, and through the bitstream generation phase every time a setting is changed. For the eMIPS design case shown in Table I, it takes about one hour to get a new bitstream generated on a 3GHz PC, Intel Core 2 duo. This does not include the time required to select the signals manually or to reconfigure the FPGA. In gNOSIS, this time is reduced to nearly zero. The user will still have to copy a few files and run gNOSIS!

In terms of usability, Chipscope has a superior interface with a nice GUI. But the user still needs to decide which signals she/he wants to monitor and under what conditions. This trial and error continues until the bug is found, demanding large amounts of manual labor and time. In contrast, gNOSIS tells the exact time and location of the bug, enabling the designer to focus on the details of the bug once it is found. And since the process is automated, the user can run gNOSIS and be engaged in other work while gNOSIS continues verification. Additionally, multiple instances of gNOSIS could run in parallel, all unsupervised, verifying different portions of a large project.

The runtime of a single iteration for the gNOSIS verification loop is shown in Table IV. The Dependency column indicates if the time is device dependent (e.g. the size of bitstream file), or design dependent (e.g. number of FFs in the design).

TABLE IV. RUNTIME PER ITERATION OF GNOSIS

Action	Time (sec)	Dependency
Update Timer	0.083	Device
Reconfigrure using iMPACT	16.152	Device
Simulation Update + Run [Execution on FPGA in parallel]	0.031+ 1.688	Design
Parse .vcd File	0.245	Design
Readback from FPGA (3MB)	0.233	Device
Parse Readback Data	0.001	Design
Compare	0.595	Design
Total	18.448	

VII. CURRENT LIMITATIONS OF GNOSIS

The current version of gNOSIS works only for the cases where the errors are identically reproducible for every run of the design. It also compares FFs only; Block RAMs and I/Os are not yet incorporated. Another assumption is that 'Reset' is applied only once, in the beginning of the execution.

Currently gNOSIS only works for Verilog designs targeting Xilinx Devices. The supported simulator is Modelsim only. The design must be synthesized using Xilinx ISE with 'keep hierarchy' option set to 'No', because otherwise <design_name.ll> file may have duplicate names for wires in different instances of similar modules. The configuration data must be generated without CRC data or encryption.

VIII. CONCLUSION

We have reported the initial implementation of gNOSIS, a board-level debugging/verification tool for Xilinx FPGAs. gNOSIS uses the Capture/Readback features of the FPGA to retrieve the state of the entire FPGA at a given time and verifies it against simulation data. Using gNOSIS is pleasant because it never requires repeated changes in the design or in the design flow. In addition to improving the efficiency of board-level debugging, gNOSIS also opens the door for exploring smarter debugging techniques. For example, we can insert assertion check circuits in the design and on failure, trigger gNOSIS to retrieve/analyze data to find the reason. We can also use gNOSIS to run a design to a certain point in the hardware then checkpoint and resume execution in simulation. This allows us to perform additional verification in simulation by reducing the simulation time required to get to the events of interests. .

Using the netlist connectivity information along with the information that gNOSIS provides, it is also possible to provide the user with the name of signals that may have caused the error.

Immediate future works include incorporating Block RAMs and I/O in the verification process and eliminating iMPACT from the main verification loop. By supporting user defined triggers while maintaining complete visibility of the circuit, we can find the bugs that are not identically reproducible. Implementing the write configuration feature in the monitor circuit will also speed up the verification process and allow checking what-if cases, such as loading an intermediate state in the FPGA and see how it works.

ACKNOWLEDGMENT

We thank Ken Eguro for the SIRC API which we used as the hardware/software interface for gNOSIS.

REFERENCES

[1] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: concepts, overhead analysis, and implementation," Proc. International Symposium on Field Programmable Gate Arrays 2007 (FPGA 2007), pp. 188-196.

[2] A. Tiwari and K. A. Tomko, "Scan-chain based watch-points for efficient run-time debugging and verification of FPGA designs," Proc. Asia and South Pacific Design Automation Conference 2003 (ASP-DAC 2003), pp. 705-711.

[3] B. L. Hutchings and B. E. Nelson, "Unifying Simulation and Execution in a Design Environment for FPGA Systems," VLSI Systems, vol. 9, Feb 2001, pp. 201-205.

[4] B. L. Hutchings et. al., "A CAD Suite for High-Performance FPGA Design," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines 1999 (FCCM 1999).

[5] J. B. Note and E. Rannaud, "From the bitstream to the netlist," Proc. International Symposium on Field Programmable Gate Arrays 2008 (FPGA 2008).

[6] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification," Proc. Field-Programmable Logic and Applications 2001 (FPL 2001), pp. 483-492.

[7] W. J. Landaker, M. J. Wirthlin, and B. L. Hutchings, "Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System," Proc. Field-Programmable Logic and Applications 2002 (FPL 2002), pp. 75-84.

[8] Altera webpage: http://www.altera.com.

[9] Xilinx webpage: http://www.xilinx.com.

[10] http://www.xilinx.com/products/software/chipscope/chipscope_ila_tut.pdf.

[11] http://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[12] http://model.com/.

[13] Crawford John D. ,"An Electronic Design Inter-change Format,", Format", Proc. ACM/IEEE Design Automation Conference 1984 (DAC 1984), pp. 683- 685

[14] Ken Eguro," SIRC: An Extensible Reconfigurable Computing," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines 2010 (FCCM 2010).

[15] http://www.mathworks.com/fpga-design/.

[16] http://www.xilinx.com/products/jbits/.

[17] http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_verifier_ds.pdf.

APPENDIX

In addition to the works/features reported in the main body of the report, we have also implemented 'restore' feature in the FPGA. By using this feature we can restore the FPGA to any previously captured or reconfigured state, bypassing the reset every time. However, we have not yet implemented any use-case example of this feature. The main hurdle is the use of same FPGA for Monitor/Server and DUT. Restoring the DUT also restores the Monitor/Server, which we don't want to happen.

Another feature that is implemented but not tested thoroughly yet is the capability of using 'reset' to capture data again and again. The purpose is to eliminate iMPACT, the most time consuming portion right now, from the main verification loop. We have implemented a sub-command that will set the timer value (Counter_Max) without having to re-configure. However, we haven't yet been able to make CAPTURE_VIRTEX5 capture multiple times (without reconfiguring the FPGA) with "ONE_HOT" set to "true". It does capture multiple times when "ONE_HOT" is set to "false" (we assert the enable signal for one cycle only every time), but the capture timing sometimes seems to be off by one cycle. This results in inconsistency with simulation data and hence hasn't been incorporated in the final version.

While determining the frame address of the FPGA using debug-mode bitstream, some of the frame addresses were missing! So, we have assumed a continuous increase for those addresses. So far, that hasn't caused any trouble. We have run as large design as eMIPS through the pre-processor of gNOSIS, and it ran without any trouble.

The .ll file sometimes doesn't have information about some of the registers that are present in the EDIF netlist. The exact reason is not known yet. The preprocessor generates message if such cases exist.

The rule checker for the parser of EDIF was generated using BISON in cygwin.

How to run gNOSIS:

Step1: Generate bitstream and configure FPGA. Be sure to have .ll and post-synthesis verilog file generated too. The design should be synthesized with 'keep hierarchy' set to 'No'. The bitstream should not be in debug mode and should not have CRC or encryption data.

Step2: Copy .bit, .ngc, .ll to the pre-processor folder.

Step3: Run pre-processor with the design name as the argument.

Step4: Copy the resulting output file 'output_FF.txt' to the folder of the main verification program.

Step5: Set simulation files and run the main verification program.