NIFD: Non-Intrusive FPGA Debugger
Debugging FPGA 'Threads' for Rapid HW/SW Systems Prototyping

Hari Angepat, Gage Eads, Christopher Craik and Derek Chiou
Department of Electrical and Computer Engineering
The University of Texas of Austin
{angepat,geads,craik,derek}@fast.ece.utexas.edu

***Abstract*—Debugging hardware has always been difficult when compared to debugging software, in large part due to a lack of convenient visibility. This paper describes the open NIFD framework that provides software-like debugging facilities to both pure FPGA and hybrid FPGA/software platforms, allowing a designer to treat the hardware logic like a specialized remote software debug target. NIFD provides features such as single stepping, breakpoints, and examination of the full hardware state from a standard debug console such as GDB. The framework leverages built-in readback support to enable *non-intrusive*, *transparent* debugging with full *observability* and *controllability*. This technique is not only useful for debugging, but can also be used in production environments for infrequent events such as the slow sampling of counters.**

I. Introduction

With the growing complexity of modern digital systems, debug, verification, and validation have become critical design concerns. This paper introduces a open framework, NIFD (Non-Intrusive FPGA Debugger), that enables software-like debugging of FPGA-based simulators used for modeling complex digital systems. This framework allows a designer to treat an FPGA emulation environment as a specialized software debug target without adding significant overhead in hardware complexity. NIFD provides the capability to debug FPGAs using variable inspection, breakpoints, and single stepping.

The debugging framework being proposed in this paper is being developed in conjunction with the FAST simulator [1]. FAST simulators are one of a new class of FPGA-based computer system simulators [1]–[4] that run multiple orders of magnitude faster than traditional software simulators, fast enough and complete enough to run real software while accurately predicting performance. A FAST simulator is partitioned into a SW functional model and a FPGA-based timing model, where the functional model executes *first*, provides a trace to the timing model that then corrects the functional model when necessary for accuracy. This simulation technique has enabled us to simulate complex out-of-order superscalar x86 targets, using a single FPGA, orders of magnitude faster than conventional cycle-accurate simulators. The NIFD framework was conceived, designed, and implemented to make debugging FAST simulators appear to be the same as debugging a purely software simulator, thus requiring the ability to debug FPGA-implemented components of the simulator precisely in the same way as debugging the software-implemented components of the simulator. In that spirit, we provide a GDB interface to interact with the FPGA. Though NIFD was originally developed to debug FAST, the NIFD framework is applicable to many FPGA or hybrid FPGA/software systems.

A. Challenges to Flexible FPGA Usage

Traditionally, hardware development requires a stricter testing strategy than software development due to the high cost and time of fixing hardware bugs. In ASIC development, two to three verification engineers per RTL designer is not uncommon. Standard verification methodologies include unit tests that test a single block and integration tests that test several blocks together.

One can quickly and cheaply try out a design on the FPGA at speeds equal to or approaching the final product. Accordingly, FPGA development is often allocated less time and fewer verification engineers than ASIC development. However, the basic verification problem has not fundamentally changed from an ASIC design. Complexity can still be significant, interactions with other chips and software are still present, and visibility is still poor when running on the FPGA itself.

Further complicating matters, FPGAs being used for exploratory work, such as in our architectural simulators, change frequently and extensively. Rapid change makes building static testbenches difficult as the behaviors and entire subsystems may be replaced frequently.

B. Contributions

This paper introduces NIFD, an open, minimally-intrusive, rapid turnaround, debugging framework for FPGAs. We demonstrate the feasibility of using software debugging abstractions for FPGAs in a working prototype of a human-in-the-loop debugger. We also show that such abstractions are key to providing a seamless debugging environment for hybrid software/FPGA platforms.

II. Prior Work: FPGA Debugging

There has been significant work in the area of FPGA debugging and co-simulation that we group into the following main categories: Soft-Logic Analyzers, Soft Scan-Chains, Soft Replication, Soft Processor Debug Monitor IP cores, and RTL Emulation Platforms.

A. Logic Analyzers

In-situ logic analyzers such as Xilinx Chipscope [5] or Altera SignalTap [6] enable the monitoring of selected user signals by inserting small circular history buffers that capture some

limited number of signal samples. A centralized trigger unit controls these buffers, allowing multiple logic expressions to be used to trigger a sample dump. Physical transport of sample data off-chip is typically accomplished over the JTAG port to a waveform viewer running on an attached computer. Users may select trigger conditions at run-time while signals are selected at synthesis-time. However, there is generally a limit to the number of monitored signals and changing the monitored set requires either re-synthesis or additional place-and-route runs which hinders effective turn-around time.

This technique can be compared to printf-style for software applications. The user must decide what to 'print/select' at 'compile/synthesize' time. If they want to add a new 'variable/signal', it requires a new 'compile/synthesis'. For small software projects the turn-around time can a few seconds of human-in-the-loop time. For hardware projects, however, the place and route time can be on the order of hours. In addition, such techniques have significant performance overheads as the number of 'prints/signals' increases.

B. Soft Scan-Chains

Design-level scan-chain insertion techniques [7]–[9] use netlist rewriting to provide transparent debugging of arbitrary FPGA logic by inserting additional logic in-front of all state elements. By connecting these elements to form a design-level scan-chain, it is possible to examine the full state of the design at run-time. Physical transport of the chain data off-chip is typically provided over JTAG. While this technique provides full visibility, it consumes significant FPGA resources, up to 100% increase, often making it impossible to use when resources and/or timing is a concern.

Soft scan-chains can be compared to application tracing or binary instrumentation for software. The user provides a 'binary/netlist' to a tool that inserts hooks to 'tracing functions/scan-chain'. This allows the user to have full visibility into their design but with significant resource and performance costs.

A related technique, soft replication, creates two instances of a user design to minimize the intrusiveness of the scan-chain insertion on the running design. One instance of the design operates normally, while the other has a design-level scan chain and lags behind the normal design with its clock controlled by a software debug agent. The design-level scan chain on the lagging design allows full visibility and control but at a very high resource cost.

Soft replication is somewhat akin to the strategy of forking processes periodically for checkpointing, fault tolerance and debugging purposes in software development. The user provides a 'binary/netlist' to a tool that instantiates duplicate copies of the design. For hardware designs, this still requires soft scan chain insertion and duplication of logic.

C. Soft Processor Debug IP Cores

Debug cores are the closest in spirit to our approach. In particular, for soft microprocessor cores, debug cores, such as the Microblaze Debug Monitor (MDM) [5], can provide a remote GDB target that can be controlled from software. The debug core directly controls soft processor logic enabling single-step execution, reading architectural registers and setting program-counter based breakpoints. However, such cores typically require a stable set of signals to monitor and restrict the number of signals that can be monitored (typically register-files and/or bus accessible memory locations). Thus such debug cores are typically provided with soft processor cores, but not usually for more ad-hoc blocks (e.g. DMA controllers, Network Routers, etc). Our approach attempts to fill in the feature gap left between stable design-level debug cores and ad hoc debugging.

D. RTL Emulation Platforms

RTL emulators such as Palladium [10] are yet another approach to debugging FPGAs. While they provide RTL-level signal visibility, they require custom toolchains and platforms and are typically not suitable as a generic ad-hoc debugging framework for FPGAs.

III. NIFD Debug Framework

NIFD is designed as an open framework using a series of modular HW and SW components. Software components, written in C and Python, are provided to parse netlists, readback FPGA frames, and implement the GDB user interface. Hardware components, written in Verilog, are provided to communicate over JTAG, set breakpoints, and control clocking a user design.

The key technique used to enable non-intrusive sampling of FPGA state is the ability to readback hardware state over an existing configuration cable in an actively running FPGA. We currently support the Xilinx logic family due to availability, however, other FPGAs with similar readback support could also be supported.

A. Tool Flow

FPGA debugging using NIFD is very similar to standard software debugging. Software is compiled normally with a typical compiler flag to embed symbol and source line-numbers into the generated binary. The standard GDB debugger can use these symbols to setup breakpoints and provide useful stack backtraces.

In our framework, hardware debugging symbols are generated in much the same way as in software. Standard FPGA implementation tools are used to synthesize and implement a user design into a deployable bitstream (in our current implementation this is a Xilinx ISE xst/map/par toolflow). Similar to a software 'debug' flag, we require that synthesis preserves the module level hierarchy. For the Xilinx ISE toolflow, this requires setting a project-wide "keep hierarchy" flag when synthesizing the design. While this does reduce room for synthesis optimization somewhat as modules cannot be collapsed, we have found it to be lightweight enough to keep enabled for the majority of development time.

Our framework constructs a symbolic map that relates state elements in a user design to the spatial location of that symbol

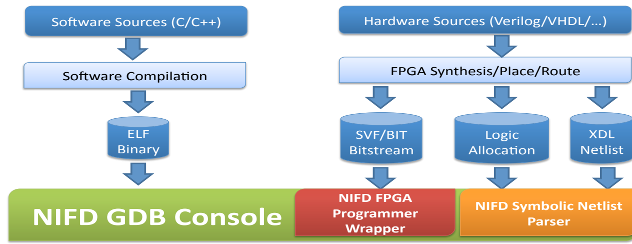

Fig. 1. Tool Flow

on the FPGA. This information comes from both a logical allocation file and an ASCII representation of the post-place-route netlist. This basic toolflow is depicted in Figure 1

B. SW Core Library

The SW components of the framework consist of a jtag cable layer, a core readback layer and a symbolic readback/netlist parser layer. At the lowest layer, we communicate with the FPGA fabric via standard JTAG cables by wrapping an existing open-source JTAG cable library, urjtag [11], which enables support for a variety of cables. The core readback layer handles the low-level details of invoking a frame readback operation against the FPGA configuration logic. This includes setting up a readback transaction, extracting the useful dynamic state from the static configuration data in the readback reply, as well as handling specifics of frame layout differences between various FPGA families.

The symbolic readback/netlist parser decodes the resource allocation tables and netlist of the user design to create a hardware symbol table. We implemented parsers for the Xilinx logic allocation file (.ll) and the ASCII netlist format (.xdl). The files are parsed first to extract a set of (symbol name, frame addr, frame offset) tuples. A second pass recombines the individual bits into user symbols using a set of heuristics about naming conventions, name mangling and bit slice packing.

C. HW IP Core Library

The HW components of the framework consist of a fail-safe communication link over JTAG, as well as a breakpoint controller. The JTAGlink module provides a simple 8-bit fifo abstraction that can be connected to software via a pseudo-serial protocol. This enables communication with the FPGA even if the machine around the FPGA has itself crash (as can happen with hybrid CPU/FPGA platforms). The framework also provides a lightweight breakpoint controller that provides a set of trigger ports and registers to maintain trigger conditions. The breakpoint controller enables the user to setup a limit value for a synthesis-time signal and halt the user design when the condition is reached. The non-intrusive sampling of FPGA state can then be used to examine the FPGA in an ad-hoc fashion.

D. User Interface

At the user interface level, we implement a GDB 7.x+ plug-in using Python that exposes GDB-style operators for the FPGA fabric. This includes tab-completion for symbolic names, as well as wrappers for programming the FPGA directly from the debug console. We implemented breakpoint commands by adding support for communicating with our breakpoint controller using the JTAGlink as described above. By sending packets over this link, we support the GDB abstraction of setting, listing, enabling, and disabling breakpoints for the set of signals attached at synthesis time to the controller. Co-debug with software is provided transparently with GDB 7.1+ support for switching debug views between multiple software processes and the hardware 'process'.

E. Challenges and Limitations

While implementing debug on-top of existing hardware readback logic is attractive, it does have some limitations. First, certain elements such as RAMs cannot be read back online due to limited port resources. We use a clock-gating approach to avoid actively using the RAMs while a readback is in progress. Second, as we use a slow-speed configuration link for readback, our off-chip bandwidth for sampling is limited. While alternative high-speed links are possible, for small numbers of samples, the current implementation is sufficient for transparent user-interaction. Finally, as readback forms the basis of the framework, we are restricted in the platforms we can support. We currently only support Xilinx parts although the framework is applicable to any fabric that supports readback in some fashion.

IV. NIFD Example Session

We present an example session of a user with a Xilinx Virtex-II Pro XUPV2 development board connected to a host computer via a USB JTAG programming cable. The NIFD gdb console is presented to the user which allows for programming the FPGA, setting breakpoints, and sampling FPGA state of the JTAG link.

In Fig 2(i) the user first loads the bitstream and synthesis generated placement and allocation files to create the NIFD symbol table. The board communication channel is then setup, allowing the FPGA to be programmed.

The user may then use the 'nifd-print' command to print the state of a variable in the active design (Fig 2(ii)). The command provides tab-completion to navigate through the design hierarchy.

Along with simply printing variables as the 'thread' executes, NIFD allows setting breakpoints to halt the design when a given value is matched. In the example, the user uses the 'nifd-breakpoint' command to set a breakpoint against the 'cnt' variable with a value of 5. (Fig 2(iii)). Once the breakpoint triggers, additional variables may be monitored. The breakpoint can also be disabled by signalling to the breakpoint controller (Fig 2(iv)).

By providing this familiar software debug session for hardware (that can be viewed as user 'threads' executing on the FPGA), NIFD lowers the bar to debugging FPGAs. By hiding programming detail, allowing variables of interest to be selected after place-and-route, and controlling breakpoints

```
            Welcome to NIFD!
(gdb) nifd-set-bitstream          fpgatop.bit
(gdb) nifd-set-bitsyms            fpgatop.ll
(gdb) nifd-set-bitxdl             fpgatop.xdl
(gdb) nifd-set-fpgalink-program   usb
(gdb) nifd-set-fpgalink-readback  jtag
(gdb) nifd-fpga-program
```

(i)

```
(gdb) nifd-print <TAB>
    ....
    cnt
    fifo/rdptr
    ....
(gdb) nifd-print cnt
    cnt has value 0
```

(ii)

```
(gdb) nifd-breakpoint cnt 5
    cnt bkpt=5 at indx=0
(gdb) nifd-breakpoint-list
    index  status     value
    -----  ------     -----
      0      ON         0
(gdb) nifd-fpga-resume
```

(iii)

```
(gdb) nifd-print cnt
    cnt has value 5
(gdb) nifd-breakpoint-list
    index  status     value
    -----  ------     -----
      0    TRIGGERED    5
(gdb) nifd-breakpoint 0 off
```

(iv)

Fig. 2. Example User Session

while the FPGA is active executing a 'thread' maintains the illusion that the FPGA is simply a custom processor executing a program.

V. Comparisons to Alternative Approaches

Previous techniques [7]–[9], [12]–[18] have limitations such as (i) imposing significant requirements on the design flow including explicitly marking signals to be used for debugging, (ii) requiring a full resynthesis when adding additional debug signals and (iii) requiring FPGA resources for the tracing structures that sometimes results in a 100% overhead.

Our framework is designed to provide a familiar software-style interface to the FPGA fabric itself, while minimizing hardware constraints. In particular we have attempted to leverage the built-in readback support as much as possible to provide no-overhead, transparent debugging. We also believe our framework is the first published work to treat hardware as an "FPGA-thread", reducing the debugging burden. This also allows a designer to have a completely unified view of the running system, extending to both hardware and software components of the system under test.

VI. Current Status and Future Work

Our current prototype is built using a standalone Xilinx XUPV2 board. The readback link uses a Xilinx USB programming cable that operates rates up to 500 kB/s. As static configuration overhead incurs a penalty of 25x, we have an effective sampling bandwidth of 20kB/s. This is sufficient for matching user interaction rates with a debug console.

While NIFD provides a comprehensive software-style view of an active FPGA program, it would be useful to provide concurrent views of both software and hardware in a more unified fashion. As NIFD currently exposes a FPGA program as a separate "software process", we can monitor both hardware and software from the same console. Unifying breakpoints and single-stepping to allow cross triggering breakpoints across multiple such processes is the subject of future work.

Additionally, we are adding support for a flexible board platform configuration interface that enables an end user to quick add support for their specific FPGA platform board. Once completed, we believe the introduction of the open NIFD framework can significantly lower the development bar typically found in FPGA/hardware development and have a significant impact on developer productivity.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0541416 and 0615352. We thank Xilinx for software and hardware donations, and the anonymous reviewers for their feedback.

References

[1] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proc. of IEEE Symp. of Microelectronics (MICRO)*, Dec. 2007.

[2] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs," in *Proc. of IEEE Symp. on Field Programmable Gate Arrays (FPGA)*, Feb. 2008.

[3] D. Patterson, Arvind, K. Asanović, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, , M. Oskin, J. Rabaey, and J. Wawrzynek, "RAMP: Research Accelerator for Multiple Processors," in *Proc. of Hot Chips 18*, Aug. 2006.

[4] N. Dave, M. Pellauer, Arvind, and J. Emer, "Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA," in *Proc. of the Workshop on Architecture Research using FPGA Platforms (WARFP)*, Feb. 2006.

[5] Xilinx, "Xilinx." [Online]. Available: www.xilinx.com

[6] Altera, "Signaltap." [Online]. Available: www.altera.com

[7] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting bitstreams for debugging FPGA circuits," in *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

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

[9] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification," in *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[10] Cadence, "Cadence incisive palladium." [Online]. Available: www.cadence.com

[11] UrJTAG, "Universal JTAG." [Online]. Available: urjtag.org

[12] K. Camera, H. K.-H. So, and R. W. Brodersen, "An integrated debugging environment for reprogrammble hardware systems," in *Proc. of ACM Symp. on Automated Analysis-Driven Debugging (ADDEBUG)*, 2005.

[13] D. Castells-Rufas and J. Carrabina, "Jumble: A Hardware-in-the-Loop Simulation System for JHDL," in *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2007.

[14] K. S. Hemmert, "Source Level Debugging of Circuits Synthesized from High Level Language Descriptions," Ph.D. dissertation, Brigham Young University, 2004.

[15] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura, "A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication," in *Design Automation Conference (DAC)*, 2004.

[16] E. Roesler and B. Nelson, "Debug methods for hybrid CPU/FPGA systems," in *Proc. of IEEE Conf. on Field-Programmable Technology (FPT)*, Dec. 2002.

[17] E. de la Torre, M. Garcia, T. Riesgo, Y. Torroja, and J. Uceda, "Nonintrusive debugging using the JTAG interface of FPGA-based prototypes," in *Proc. of IEEE Symp. on Industrial Electronics*, 2002.

[18] SGI, "Reconfigurable application-specific computing user's guide," 2005.