Debugging Framework for FPGA-based Soft Processors

David Sidler
Dept. of Computer Science, ETH Zürich
david.sidler@inf.ethz.ch

Ken Eguro
Microsoft Research, Redmond
eguro@microsoft.com

***Abstract*—Soft processors are one way to raise the computational abstraction of FPGAs while keeping the advantages of reconfigurable hardware, such as adaptability, deterministic performance and high performance/watt. Software developers can quickly build, test and deploy applications using familiar tools while still leveraging important optimizations such as application-specific custom instructions. However, they also present unique debugging problems. For example, the higher-level programming abstraction runs contradictory to classical low-level debugging tools like logic analyzers. In this work we present a debugging framework for FPGA-based soft processors that enables step-by-step debugging at the level of all soft processor instructions, time-travel debugging, post-mortem memory dumps, and performance metrics. By using knowledge about the soft processor's internals, our framework can capture execution traces up to 60x more space-efficient than traditional embedded logic analyzers.**

I. Introduction

Historically data centers have been using general-purpose processors, but specialized hardware, such as FPGAs, GPUs, and NPUs, has become increasingly popular thanks to their bandwidth, latency, and performance/watt advantages. At the same time, though, the nature of modern data center and cloud applications highlight resiliency and agility. This makes ease of code creation, code maintenance and interoperability between different platforms critical. This is an area in which traditional HDL-based FPGA implementation has been weak.

Raising the programming abstraction is a popular solution to improve the developer experience. One way to achieve this is through language-to-gate flows such as OpenCL [1], [2], HLS [3], [4], [5] and domain specific languages [6]. Most of these languages are compiled to RTL before entering the traditional FPGA tool suites. For developers only familiar with the high-level abstraction, debugging compiler-generated code, much less debugging on the hardware itself, is simply too difficult. Therefore, they rely on early high-level simulations to verify and debug their applications. This limits the type of issues that can be detected, for example finding and addressing problem with I/O peripherals, timing, or signal integrity generally requires debugging on a live-running circuit. Another way of raising the programming abstraction is through soft processors [7], [8] which retain many of the advantages over C-to-gate style tool-flows while avoiding their limitations. Not only provide soft processors a programming abstraction that is completely familiar to traditional software-only developers, but can also incorporate custom instructions for specific applications unlocking the low power and high performance that has become the hallmark of FPGA-based specialized hardware. Furthermore, soft processors have the ability to load and execute different programs with far less compilation effort and run-time disruption.

However, debugging code running on a soft processor can still be complicated. First, even debugging tools of commercial soft processors such as the Nios II have relative primitive capabilities - limited to simple step-by-step execution, the examination of memory locations and the register file. Second, highly optimized soft processors that incorporate many custom instructions present another challenge. These monolithic CISC-like operators must be instrumented with traditional embedded logic analyzers, e.g., Xilinx Chipscope [9] or Altera SignalTap [10]. Unfortunately, using these tools effectively requires time and a depth knowledge of the internals of the processor and overall hardware design techniques that are contrary to the original promise of ease-of-use.

In this paper we describe an easy-to-use yet sophisticated debugging framework for soft processors. It supports basic features, such as step-by-step execution and halting execution with breakpoints, along with more advanced features such as time-travel debugging and automatic performance metrics. Using this framework, we instrumented a CISC-like stack processor, however we believe the lessons learned from this work are general and can be applied to other soft processors. One specific takeaway from this work is that while indiscriminate instrumentation will guarantee that all relevant information necessary for debugging will be captured, it is exceedingly inefficient. This work shows that by using knowledge of how specific structures are used we can support sophisticated debugging capabilities with minimal additional resources.

II. Soft Processor

The soft processor we instrumented with our debugging framework implements a *Trusted Machine* (TM) as used in Cipherbase[11]. Cipherbase is a databse system that stores and processes strongly encrypted data. The FPGA-based *Trusted Machine* implements a stack machine that evaluates expressions from SQL queries. Despite using our debugging framework with this specific soft processor, we consider the

978-1-5090-5602-6/16/$31.00 © 2016 IEEE

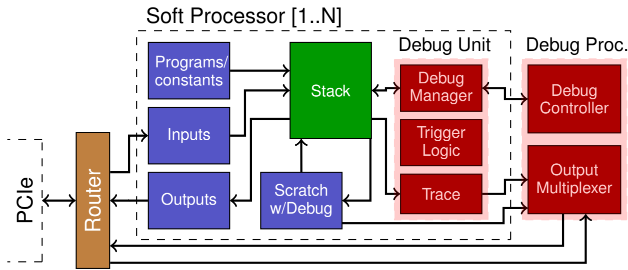

Fig. 1: Instrumentation of the soft processor with the *Debug Unit* and the *Debug Processor*

Trace

IC	PC	Instruction
0	0	LoadInput
1	1	LoadInput
2	2	Comp
3	3	Out

Trace Operands

IC	Cmd	Mem.	Type	Off.	Len.
0	PUSH	INPUT	VAR	0	6
1	PUSH	INPUT	VAR	1	5
2	POP	INPUT	VAR	0	5
2	POP	INPUT	VAR	1	6
2	PUSH	SCRT.	FIX32	0	4
3	POP	SCRT.	FIX32	0	4

Fig. 2: Trace for example program

method presented in this paper to apply to a wider range of soft processors, and not only to our specific design.

Since the internal processing abstraction in Microsoft's SQL server is a stack machine, this was used for the TM in Cipherbase. The stack machine, see Fig. 1, has multiple memory banks: the program memory which holds multiple programs, the input memory which holds the input data, the output memory where the results are written to, and a stack which holds operands and intermediate values. This processor includes an enhancement beyond the traditional stack machine architecture, a scratch memory. This scratch memory is used to optimize execution and eliminate repeated computation. All intermediate results are appended to the scratch memory and can be referenced later, thereby avoiding their recomputation.

The stack-based nature of the TM offers both opportunities and challenges for efficient remote debugging. For example, as will be discussed in more detail later, while there is potentially a large amount of intermediate state contained in the stack, if the debugging framework is aware of and can leverage the fact that the stack memory is indeed used as a stack, we can record and transmit incremental changes efficiently. Another aspect we consider is the different access modes of the various memories. For example, during program execution the program and input memory are read-only, the output memory is write-only and the scratch memory can be read from any location, but new results written to scratch are only appended.

III. Hardware Debugger

A. Instrumentation of the Soft Processor

The soft processor was instrumented with debug and trace logic as shown in Fig. 1. The debug logic consists of the *Debug Units* which is inserted into each soft processor and a single *Debug Processor* located externally. The *Debug Processor* is connected to its own PCIe channel and receives commands from the host which it forwards to the corresponding *Debug Unit*. The content from different memories can be send through the *Output Multiplexer* to the host. Separating the debug logic into *Debug Processor* and *Debug Unit* minimizes the amount of logic inserted into the soft processor to not alter its execution model and allows debugging of multiple soft processors simultaneously through a single *Debug Processor*.

The *Debug Unit* can execute the following commands halting, resuming, and single-stepping program execution. It consists of three parts, the *Trigger logic*, the *Debug Manager*, and the *Trace* module. The *Trigger logic* is responsible for keeping track of the breakpoints set by the developer and halt program execution when a breakpoint is reached. This is implemented through a bitvector, where each bit represents a line of code in the program. This bitvector is compared to the current program counter. The *Debug Manager* handles the incoming commands, resumes and halts execution of the current program or initiates a memory read-out which forwards the data to the *Output Multiplexer*. The *Trace* module contains internally two trace buffers implement in BRAM. The first one stores the instructions and the second one the operands.

B. Debugger Interface

The interface exposed to the host reflects functionality common in software debuggers. To interact with the hardware debugger the host sends messages over PCIe to the FPGA. A message contains a command, an optional argument, and the index of the soft processor targeted. The commands are:

- **ENABLE_DEBUG**, enables the hardware debugger.
- **HALT_EXEC**, halts the program execution at the end of the current instruction.
- **CONTINUE_EXEC**`(BOOL singleStep)`, resumes execution when program was halted before. In case `singleStep` is TRUE the execution is resumed only for a single instruction.
- **SET_BREAKPOINT`(UINT loc)`**, sets a breakpoint for the instruction at position `loc` in the program.
- **UNSET_BREAKPOINT`(UINT loc)`**, removes the breakpoint at position `loc` in the program.
- **GET_DUMP`(ENUM mem)`**, dumps the content of a memory specified by `mem` to the debug output. This is used to dump the scratch or trace memory.

IV. Trace Reconstruction

To illustrate the trace generation and reconstruction we introduce a short example program. The program loads two strings, `alpha` and `beta`, from the input memory and pushes them onto the stack. Then these two strings are popped from the stack and compared to each other. The resulting value `-1` is pushed onto the stack. In the last instruction this value is then popped from the stack and written to the output memory.

The trace generated by this example program is stored in the two trace memories, *Trace* memory and *Trace Operands* memory, as shown in Fig. 2. For each instruction an entry is added to the *Trace* memory containing the instruction counter (IC), program counter (PC) and the instruction name. Since

an instruction can have multiple operands, for instance the `Comp` instruction in our example program, the trace data about the operands is stored in a separate memory. This separation leads to a memory conscious implementation which makes no assumptions about the number of operands an instruction can have. The *Trace Operands* memory stores for each operand, the instruction counter, the memory operation and location, and the data type. Entries in the two memories can be linked through the IC. The memory descriptors in the linked entries can be resolved through the memory readouts leading to a complete human-readable trace. Since, the constants, input, scratch, and output memory are not modified during program execution, it is sufficient to only store memory descriptors.

V. Debugger Capabilities

The interface to the *Hardware Debugger* provides all functionalities required for software-like step-by-step debugging. Further the execution trace can be reconstructed by reading out the memories of the stack machine.

Step-by-Step Execution: Using our debugging framework the programmer can set breakpoints to halt the program at the instructions of interest. She can also single-step through the program or halt it at the next instruction.

Time-Travel Debugging: By reconstructing the complete trace, the programmer can track all intermediate steps and results since the start of the program. In Cipherbase, there are known upper limits on the number of executed instructions. Thus, we can provision the trace buffers in hardware to have enough space to hold the complete trace. But in any case, it is possible to go back in time for a few hundred instructions tracking the events leading to the current state of the program.

Post-mortem Debugging: To our knowledge this functionality was not yet explored in FPGA-based designs. Post-mortem debugging allows the retrieval of valuable information after the program crashed, similar to memory or core dumps in software environments. Since the framework continuously collects data about the execution of the current program, in case of a program crash a memory read-out can be triggered by the software debug thread. The programmer can then time-travel through the reconstructed trace which lead to the crash. This functionality can be very useful in production deployment or FPGA-based distributed systems, e.g. [7], [12].

Performance Counters: The *Trace* module keeps for each instruction track how long its execution took (in cycles). This is valuable feedback to programmers who are not familiar with the internals of the stack machine.

VI. Evaluation

For our evaluation, we focus on two metrics, the amount of memory required to store a program trace and the overhead in resource usage due to the debugger and trace logic. The evaluation was done on an Altera Stratix V D5 FPGA.

A. Reduction in Trace Size

In TABLE I we compare trace size capturing three different programs using either our framework or Altera SignalTap. The first program is the example program used in section IV, the other two programs are not discussed in detail since they are used in Cipherbase. However for each program the number of instructions, operands, and execution cycles is listed. Using these numbers the size of the trace for the two different approaches is calculated. Unlike, SignalTap which stores the value of every interesting signal in each cycle, our framework only stores data when a new instruction or operand is loaded. As a result, SignalTap requires 26-64 times more memory in comparson to our approach when capturing the same information about the program execution. This means our framework can capture an up to 64 times longer program trace than SignalTap.

TABLE I: Reduction in Trace Size

	#Instr.	#Oper.	Exec. cycles	Our Trace [bit]	SignalTap [bit]	**Ratio**
Ex. Prog.	16	24	182	1,736	45,384	**1:16**
Prog II	24	30	596	2,298	147,808	**1:64**
Prog III	67	1,408	1,739	7,652	431,272	**1:56**

TABLE II: Resource Overhead

	ALMs			M20Ks		
#Soft Proc.	1	2	4	1	2	4
Soft Proc. only	10,552	20,400	40,941	119	234	464
Instr. Soft Proc.	11,089	21,896	45,259	128	248	488
Overhead [%]	5.1	7.3	10.5	7.6	6.0	5.2

B. Resource Usage

TABLE II shows the resource overhead for different deployments using 1, 2 or 4 soft processors. The overhead in ALMs is in the range of 5-10%, mostly due to increased connectivity and arbitration between the *Debug Processor* and the different *Debug Units*. The overhead in additional BRAMs used to store the trace is slightly lower at 5-7.5 %. This means that for a less than 10% increase in resources the circuit can be instrumented with our *Hardware Debugger*. We consider that our debug and trace logic is also very valuable in production deployments, however it can be removed to avoid the resource overhead.

VII. Related Work

Prior Work has investigated ways to improve the debugging experience for hardware circuits, thereby increasing productivity. These improvements can be categorized into four areas: 1) accelerating the debugging cycle, 2) improving visibility of signals in a circuit under test, 3) enhancing controllability during debugging (e.g., software-like step-by-step execution), and 4) debugging circuits implemented in a higher-level language.

A. Acceleration of the Debug Cycle

[13] showed a method to shorten the debugging cycle by instrumenting a bitstream with debugging hardware, thereby modifying the trigger and capture signals without a complete recompilation. Similarly, [14] use incremental synthesis techniques to insert the debug logic after place and route into the circuit. Since our framework debugs at the level of

soft processor instructions without modifying the underlying hardware, the debug cycle is in the range of seconds.

B. Visibility and Controllability

Visibility and controllability are often provided together, since the latter is often a requirement for the former. *Readback* a feature available in some older FPGA devices [15] and scan-chains[16], [17], [18] have the ability to read out the entire state of an FPGA circuit. Scan-chains consist of scan-registers which are linked together into a chain. However, scan-chains introduce a significant overhead in resource usage, a 84% increase was shown by [16]. Further, scan-registers can alter the timing behavior of a design. Both methods have to halt the circuit to extract a consistent state, this is commonly implement by disconnecting the clock signal from the main logic. However, halting the circuit makes it impossible to test the circuit at normal execution speed. Rendering these methods unsuitable for debugging interaction with I/O peripherals. Given the slow readout process, debugging at a cycle-granularity is not time-efficient and it might vanish any speed gain of hardware debugging over simulation. [19] addressed this by introducing watch-points to halt the circuit in predefined potentially interesting states. Our debugging framework provides visibility by collecting trace data for each executed instruction and controllability through contextual breakpoints in the high-level program or single-stepped execution, giving the developer the choice between high-scale and detailed debugging.

C. Debugging for High-level Languages

One major challenge when debugging high-level languages which compile to RTL, is to determine a meaningful mapping between the high-level code and the physical hardware implementation. The compiler can apply many transformations and optimizations to the programmers' code. Many approaches[20], [21], [22] which correlate RTL to high-level code use the resulting mappings to automatically detect discrepancies between the high-level code and the compiler-generated hardware implementation. This functionality can detect bugs in the compiler and synthesis tool, but not at the application level. [21] provides this functionality as part of a framework which also introduces variable inspection and watch-points for preselected variables. Since our framework is debugging at the program-level, a mapping between program instructions and RTL is not required.

Similar to embedded logic analyzers, HLS debuggers have limited capacity to capture signals. HLS tools must preselect signals, possibly using user-defined watch-points, to obtain the most relevant information for the application developer. [23] presented a technique to maximize the amount of information that can be collected in a fixed size trace buffer by applying knowledge of the circuit and its signals to compress the trace without losing valuable information. Our framework collects trace data at the granularity of program instructions, thereby benefiting from a natural compression by only storing information valuable and understandable to the developer.

VIII. Conclusion

We presented a debugging framework for soft processors addressing the following areas of FPGA debugging: Quick debugging-cycle, visibility and controllability of the circuit, and debugging at a high abstraction level. In addition to common features such as stepwise execution, the framework introduces time-travel debugging and post-mortem trace dumps. These two features are especially useful in production deployment, where rare bugs occur occasionally and the developer is only informed after the fact. On top of functional debugging, the performance counters in the trace allow for basic performance debugging. Despite those sophisticated features, the introduced resource overhead is minimal. This is achieved by using knowledge about the internal structure of the soft processor. We demonstrated the applicability of the framework on a CISC-like stack processor, however we believe that the lessons learned in this work also apply to other soft processors.

References

[1] Altera, "Programming FPGAs with OpenCL," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

[2] Xilinx, "Xilinx SDAccel," http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-wp.pdf.

[3] A. Canis, J. Choi, M. Aldham *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *FPGA'11*.

[4] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis," http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug902-vivado-high-level-synthesis.pdf.

[5] R. Nikhil, "Bluespec system verilog: efficient, correct RTL from high level specifications," in *MEMOCODE'04*.

[6] J. Bachrach, H. Vo, B. Richards *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC'12*.

[7] A. Putnam, A. Caulfield, E. Chung *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA'14*.

[8] K. Ovtcharov, O. Ruwase, J.-Y. Kim *et al.*, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, 2015.

[9] Xilinx, "ChipScope Pro 10 Software and Cores User Guide," http://www.xilinx.com/ise/verification/chipscope_pro_sw_cores_10_1_ug029.pdf.

[10] Altera, "Quartus II handbook volume 3: Verification," vol. 3, 2015.

[11] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann *et al.*, "Orthogonal security with cipherbase," in *CIDR'13*.

[12] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *NSDI'16*.

[13] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting bitstreams for debugging FPGA circuits," in *FCCM'01*.

[14] E. Hung and S. J. E. Wilton, "Incremental trace-buffer insertion for FPGA debug," *VLSI*, vol. 22, no. 4, pp. 850–863, April 2014.

[15] Xilinx, "Virtex FPGA series configuration and readback," *Application Note XAPP 138*, 2005.

[16] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, "Using design-level scan to improve FPGA design observability and controllability for functional verification," in *FPL'01*.

[17] M. Renovell, P. Faure, J. Portal, J. Figueras, and Y. Zorian, "IS-FPGA : a new symmetric FPGA architecture with implicit scan," in *ITC'01*.

[18] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *FPGA'07*.

[19] A. Tiwari and K. A. Tomko, "Scan-chain based watch-points for efficient run-time debugging and verification of FPGA designs," in *ASP-DAC'03*.

[20] P. Fezzardi, M. Castellana, and F. Ferrandi, "Trace-based automated logical debugging for high-level synthesis generated circuits," in *ICCD'15*.

[21] N. Calagar, S. Brown, and J. Anderson, "Source-level debugging for FPGA high-level synthesis," in *FPL'14*.

[22] L. Yang, S. Gurumani, D. Chen *et al.*, "AutoSLIDE: Automatic source-level instrumentation and debugging for HLS," in *FCCM'16*.

[23] J. Goeders and S. Wilton, "Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs," in *FCCM'15*.