An Integrated Debugging Environment for Reprogrammble Hardware Systems

Kevin Camera
kcamera@eecs.berkeley.edu

Hayden Kwok-Hay So
skhay@eecs.berkeley.edu

Robert W. Brodersen
rb@eecs.berkeley.edu

Berkeley Wireless Research Center
University of California, Berkeley
2108 Allston Way, Suite 200
Berkeley, CA 94704

ABSTRACT

Reprogrammable hardware systems are traditionally very difficult to debug due to their high level of parallelism. In our solution to this problem, features are inserted into the user's design which allow the system to be monitored and updated at runtime. An assortment of logic is added before synthesis to allow variable buffering, assertion checking, and automatic breakpointing. Low-level clock control and access to off-chip storage is managed by a custom hardware operating system. Through the addition of these features, a system can be debugged directly on the hardware, bypassing simulation and reducing iterations through the design flow.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]; B.5 [Register-Transfer Level Implementation]; B.6.3 [Design Aids]: Simulation, Verification

General Terms: Design, Verification

Keywords: design, simulation, verification

1. INTRODUCTION

Field-programmable gate arrays (FPGAs) are integrated circuits that allow arbitrary computing elements to be mapped onto a reconfigurable 2D array of combinational logic blocks (CLBs). To implement an algorithm on an FPGA, the input description (usually some form of hardware description language, or HDL) is *synthesized* into its basic logical operations, mapped into equivalent CLB functions, and finally placed and routed spatially on the FPGA fabric. The final physical implementation, represented as a configuration bitfile, is then loaded onto the FPGA via a configuration bus and the device is ready to serve its purpose.

Related research has shown that FPGA-based computing platforms, such as the Berkeley Emulation Engine (BEE)[1] and BEE2[2], can achieve much higher levels of performance than processor-based platforms. This is largely due to the efficient direct mapping of algorithms onto the array, rather than executing a purely sequential series of instructions. This performance gap will continue to become larger, as FPGA capacity scales in two dimensions with Moore's Law, and processor performance scales only as a function of the maximum core clock frequency.[1]

While FPGAs have shown a desirable benefit in performance, they have been hindered by very difficult programming and debugging methodologies. Although less complicated than a full-custom chip design, the FPGA design process still resembles a traditional hardware design flow, as mentioned above. Typically, the system under development is described in an HDL like VHDL or Verilog, or possibly in a specialized higher-level language. At this point, the designer can simulate the system in software to prove correctness. However, simulation of a very large, parallel hardware design is extremely slow on even the fastest workstations, typically at least 10^6 times slower than the actual hardware. This trend is also getting worse as FPGA capacity continues to scale up, causing software emulation to demand even more memory and processing power.

One natural observation is that it would be faster to move directly onto the hardware platform to execute, rather than emulate, the design in progress. There are two prohibitive bottlenecks to this approach. The first is the time required for the place and route (PAR) stage. PAR can take anywhere from minutes to hours for a single run targeting a modern FPGA. The process is also entirely "flat", meaning even a single design change requires PAR to rerun for the whole chip. The second bottleneck is the complexity of observing the hardware itself. Inspecting the running hardware requires that any signals of interest are designed in advance to be accessible on external pins, requires a sophisticated piece of equipment (i.e. logic analyzer) to capture the signals, and requires the designer to manually interpret the values and timing of all the captured waveforms.

The focus of this work is to bring powerful, high-level debugging controls directly onto the FPGA platform, allowing rapid and early design verification and exploration without software simulation. While significant work has been done in the areas of logic verification and distributed debugging,

[1]This assumes an inherent limitation in the amount of instruction-level parallelism that can be exploited, which is a more advanced topic of computer architecture.

AADEBUG'05, September 19–21, 2005, Monterey, California, USA.

our work differs in several ways. Static logical error diagnosis, such as in [3], only deals with errors compared to a functional specification and does not provide any dynamic design exploration. Advanced models, such as [5], have also been developed to automatically trace errors in HDL/RTL level designs. While this greatly improves the efficiency of RTL debugging, it does not eliminate the increasing complexity and performance limitations of software-based simulation. Large-scale hardware designs can also exhibit similarities to distributed computation on parallel machines, the latter of which is reviewed in [4]. However, our target platform and design flow feature a fully synchronous, direct-mapped hardware architecture which does not suffer from the limitations of distributed computation such as the lack of a global clock and a common memory space.

FPGA vendors have also started to implement features to aid in debugging, such as Xilinx ChipScope™[9]. However, their solution requires the memory cores to be inserted at design time, and cannot be changed without rerunning the lengthy implementation tool flow. One alternate debugging solution developed recently is the UNSHADES system[7], which uses a small, auxiliary debug controller to start and stop the system clock and read or write all register contents through the configuration port. This solution has very low overhead and is complementary to any existing design. However, it is also dependent on specific Xilinx configuration features, and requires an external control FPGA tied to a host workstation to provide the actual debugging interface. This research is more focused on a general-purpose, integrated debugging methodology which utilizes the hardware platform to debug itself, and is applicable to any FPGA vendor or even alternative programmable architectures.

The methodology chosen here is based on inserting hardware elements into the design which provide user-driven debugging support. A similar use of logic insertion on FPGAs can be found in [8], although their focus was on the debugging of embedded processor code rather than direct-mapped hardware. Because logic is being injected into the user's design, our approach is not intended to preserve the timing of the system. In fact, the critical timing paths in the design will surely be lengthened as a result of both logic insertion and generally longer wire delays caused by additional crowding on the device. Therefore, a final timing-driven pass through the tool flow will still be necessary once functional and algorithmic correctness is proven in order to achieve maximum performance. Recall that since the hardware has 10^6 times better performance than simulation, there is plenty of headroom to partially degrade the speed of the hardware and still come out well ahead.

The organization of this paper is as follows. Section 2 describes the hardware operating system, being developed in-house, which provides several valuable runtime services. Section 3 and its subsections describe the supported debugging features and their practical implementation. Section 4 concludes the paper with the current state of this research and plans for the future.

2. PLATFORM AND OS SUPPORT

We are developing our own hardware operating system, the Berkeley Operating system for ReProgrammable Hardware (BORPH)[6], designed to improve the efficiency of programming and verifying large-scale reconfigurable hardware platforms. It allows a hardware designer to deploy hardware applications in a software-like runtime environment. FPGA designs are abstracted as user processes, thereby allowing flexible access to a variety of services such as network interfaces and filesystems. BORPH is being designed on BEE2[2], but its principles apply to any platform built from an array of reprogrammable devices.

The current implementation of BORPH divides the tasks of the OS into two categories. Tasks that are not timing critical (network access, mass storage, etc.) are handled by the heavyweight main kernel (`mk`), while timing critical system calls and cycle-accurate process management are handled by distributed, lightweight microkernels (`uk`) on each FPGA device.

Each user design is physically encapsulated by a copy of `uk`, making it a relocatable object. Arbitrary I/O redirection is handled by setting `uk` parameters at process load time. Furthermore, `uk` controls the clock input of a user process, which enables the kernel to stop and resume a process as needed during runtime. Since all the OS services related to debugging communicate only with the local `uk` identified with the current process, all references to kernel or OS interfaces in later sections refer to `uk`.

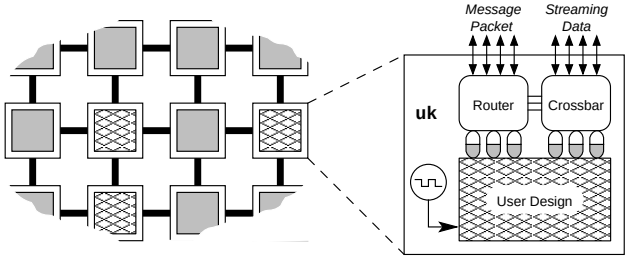

Figure 1: Architecture of a user process

General filesystem support is provided to all hardware processes. A process communicates with the encapsulated `uk` by passing packetized messages over a number of parallel lightweight interfaces. Copies of `uk` (i.e. separate process instances) residing in different parts of the system can then communicate with `mk` and each other, forming a scalable message passing network that routes data to the appropriate file location. In all cases, if a file operation is not ready, the corresponding process will be blocked automatically by `uk`. File operations are exposed to the user through a synchronous, zero-latency library component at design time, abstracting away all the details of the filesystem implementation. The availability of a filesystem is a major benefit to the user for data collection during design verification. In addition, files are utilized by the debugging infrastructure itself as an expansive storage device for buffering data values (as described starting in Sect. 3.2).

Another OS service critical to debugging is a global cycle-accurate timer that is synchronized across the entire array of FPGAs in the system. The user-driven interface of the debugger (see Sect. 3.3) uses system calls to the timer to control cycle-by-cycle execution of a process.

3. DEBUGGING INFRASTRUCTURE

Section 1 stated the huge gap in execution time between software simulation of a hardware design and running on the hardware itself. While this may seem fairly obvious, the reason designers spend so much time emulating their design in

a software simulator is that generating an FPGA bitstream is a very time-intensive process (mostly due to PAR), and accessing data on-chip requires complex test machinery *and* requires that all signals of interest be exposed on external pins at design time. Both these problems must be overcome to make debugging directly on the hardware a reality.

The main philosophy behind the debugging infrastructure is to exploit the vast spatial resources of the hardware platform to directly monitor and manipulate the design in its native architecture. The "ease of debugging" on a platform is a very subjective metric that is difficult to quantify. However, a valid solution to the problems described above should meet the following criteria: the **design elements are robustly parameterized** to allow exploration without re-implementing the hardware; all **design variables are readily accessible** for reading and writing; and finally, the user should have **complete control over process execution**. The combination of mutable variables and soft configuration of functional units allows for design exploration directly on the platform with few if any iterations back through PAR, while variable access and deterministic process control bring the convenience of sequential observation to the massively parallel hardware domain.

The practical implementation of this debugging framework has three parts, each related to a distinct phase of the design creation process.

- A library which includes abstractions of debugging features and highly parameterized functional units
- A *stitcher*, an automated tool which inserts debugging logic underneath the user's design
- A runtime interface which provides control over process execution and visual feedback of design operation

Each of these phases are covered in the following subsections. Note that our current in-house design environment is based around Simulink, a graphical dataflow language which is part of the Matlab suite by The Mathworks. Designs are specified using the Xilinx System Generator blockset, and are compiled directly into a mixture of synthesizable VHDL and pre-optimized netlist components. However, all the concepts presented here could be applied to a purely HDL-based flow, or in even more powerful ways as additional semantics of a higher-level language. The framework presented here is not specific to any single type of design flow.

3.1 Library Components

The initial interaction between the designer and the debugging infrastructure occurs during design entry. In order to facilitate the later stages of the debugging process, it is necessary for the designer to provide hints to the system of what aspects of the design may need to be explored at runtime. To make this process as simple as possible, it is best to provide a set of library components which abstract away the details of the underlying implementation. Note that even the use of the term 'library' is somewhat abstract, as some of the components described below could also be implemented as built-in features of the design environment or description language itself.

The first component included in the design library is a means of tagging and identifying variables. In software design, a variable precisely represents a data type and its location in memory, both of which can be referenced fairly easily by a debugger. However, in hardware design, a 'variable' can either be a stored (i.e. registered) value or an arbitrary name for the intermediate results of an operation (i.e. wires between sets of logic gates). In the latter case, these wires can end up being lost altogether in the synthesis and mapping phases of hardware generation due to logic optimization. Therefore, it is necessary to *tag* signals in the hardware design by placing a variable block on any signals that the designer would like to track at runtime. The variable block serves three purposes: it defines a name for the specified variable; it defines a set of parameters for the variable which will assist the debugging tools with how to store and route the variable's data; and finally, it provides a placeholder for the stitcher to insert the necessary logic for data access and breakpointing at runtime. Each instance of a variable does not automatically allocate a large number of hardware resources. Rather, it will change the amount and type of storage it consumes based on whether or not the user is currently observing it (controlled by the runtime interface described in Sect. 3.3) and what the expected rate of change is (which is hinted by the designer in the parameters of the variable block). Therefore, variables should be defined as often as needed to make them available at runtime without recompiling the design. Variable storage is handled by a combination of direct buffering and offloading to the OS, as mentioned in Sect. 3.2.

The second type of component needed in the design library is a means for defining assertions. Some typical examples of assertions in software would be to check for division by zero, ensure pointers are valid, or ensure that loop or array indices are in-bounds. Assertions have typically never existed in the hardware domain – the design had to be proven 100% correct during simulation. Similar principles to software can be applied to assertions in hardware: check the range of arithmetic operands, check for overflow on operations that don't allow it, check for state machine deadlocks or invalid states. In the context of the design library, the assertion block is a special component which has one boolean input which, when true, will instantly halt the design and wait for user interaction. Halting process execution is handled by trapping to the OS, as mentioned in Sect. 3.2.

The third type of library component represents services provided by the OS. In the software domain, one of the most die-hard manual debugging methods is piping runtime data into a file or directly to the console. In this debugging environment, the hardware OS still provides the same features. Data can be sent to a file, which could reside either in memory or on an attached disk, or to one of the special-purpose files like an interactive console. These features allow the user to perform manual debugging if necessary, and more importantly, provide a high-level service for data collection to simplify algorithm exploration and tuning.

A final characteristic that should be provided by the library is an extremely parameterized set of functional units. By having highly configurable sets of operators and using variables to define the control inputs to these operators, the hardware system can be reprogrammed on the fly to emulate a large number of different algorithms, microarchitectures, or numerical precisions. The principle for constructing the functional units should be to inherently provide many physical alternatives within a single library block, at the cost of spatial hardware resources. During the functional verifica-

tion phase, it should be sufficient to expend larger amounts of the hardware to verify the correctness of the system incrementally. This is especially true for FPGA arrays, the type of platform to which this environment is most applicable, where designs are often highly modular and can be debugged piecemeal.

3.2 Stitching

The stitcher is responsible for inserting logic to support debugging operations and for creating interfaces to the local OS. Rather than have to modify the sophisticated and proprietary FPGA tool flow, the stitcher works by traversing the hierarchy and augmenting the design before synthesis, such that the standard tools are unaffected. The stitcher also creates a database of names and internal IDs to match debugging components and their hardware elements in the final system. There are three different modules used by the stitcher: *variable control units*, *assertion control units*, and a *debug controller*.

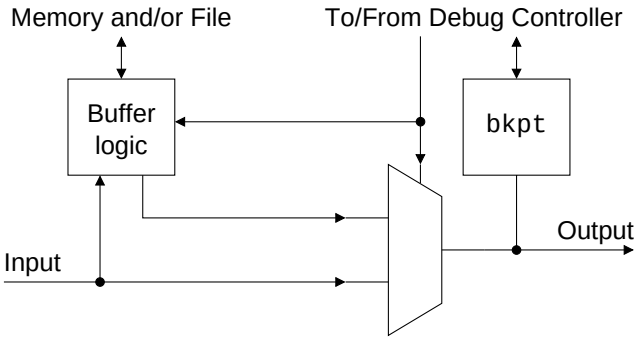

Figure 2: Block diagram of a variable control unit

Each variable placeholder in the original design is replaced with one variable control unit (VCU) in the actual hardware. A simple diagram of a VCU is shown in Fig. 2. The variable is implemented such that the normal values during operation are automatically copied into a circular buffer, which could either be stored in on-chip memory or relatively fast off-chip RAM. Currently, the initial size of the buffer is specified manually by the designer in the variable parameters, but in the future the VCU may automatically control the buffer size depending on the variable class[2] and current memory utilization. The size of the buffer is also adjusted at runtime when the user chooses a variable for close observation. In order to allow more variable history to be stored than attached memory might allow, the buffers are also streamed to one monolithic file through the OS. This method is chosen to minimize the kernel communication bottleneck, prevent the need for the OS to manage a separate file for each variable, and to optimize the bandwidth used for sending data to remote devices.

If the user chooses to override or rewind a variable's value (as described in Sect. 3.3), the variable data is read from the local buffer or a file instead of using the computed results. This allows an arbitrary system state to be restored, such as when investigating a failure condition. The VCU also contains a programmable comparator which can be used at runtime to support data-dependent breakpoints. When instructed by the user, the debug controller will enable the comparison and set its threshold, and the process will be halted (in a manner similar to an assertion failure, described below) when the comparison becomes true. The other control points of the VCU are also driven by the debug controller, and respond directly to user input at runtime.

It should be mentioned here that state machines in the design are treated as a special case. State machines are a critical source of information in most hardware designs. They often define the control flow and mode of operation of the system, and can potentially be common sources of design errors. The path through a state machine alone can indicate the overall behavior of the system before a failure occurred. For these reasons, all state machines are automatically assigned variables for the purpose of debugging, where the current state is the value of the variable.

Similar to variables, each instance of an assertion block in the original design will be replaced by an assertion control unit (ACU) in the hardware. In the current architecture, with the assertion condition defined externally by the designer, the ACU simply waits for a logical high value on its input. Once this happens, the debug controller will signal a trap on the kernel bus, which will immediately stop the main design clock. With the system halted, control is effectively passed to the user via the remote interface, and the debug controller remains in a lock-step state until the user returns the system to normal operation. In order to preserve the current state of the system at the time of the failure, it is necessary to trigger the kernel trap in the same cycle as the assertion. Clearly this can have a significant impact on the operating frequency of the design. However, as mentioned in Sect. 1, this environment is focused mainly on replacing simulation in the early design phases. As such, it is acceptable to take a penalty in clock frequency and still be far ahead of the 10^6 performance gap versus simulation. The only data sent to the ACU is an enable signal, which is assigned by the debug controller to indicate whether or not the user has chosen to ignore the given assertion. In the future, situations for which automatic assertions should be inserted (such as for common design bugs), as well as alternate methods for tolerating latency in the assertion trap (such as compensating for the delay by adjusting the variable buffers), will be investigated.

Finally, the single debug controller (DC) is a manager for the system under test, and serves as an arbiter between the OS, the VCUs and ACUs, and the remote user interface. It is the DC that regulates the design clock through the use of OS timers. In the nominal state, the design clock keeps running until the DC initiates a trap, such as for an assertion failure. At this point, the DC can single-step the design or run bursts of cycles as instructed by the user. The DC also handles user access to variables. Upon request, the DC can switch a variable between the running and recall modes, or override the variable buffers altogether to restore a user-defined state. While the DC is a simple state machine arbiter and should not consume a large number of resources, it will scale weakly in size with the number of VCUs and ACUs.

[2]For example, a `stream` class is expected to be changing on every cycle as inputs are continually processed, while a `param` class would change very rarely based on the mode of operation.

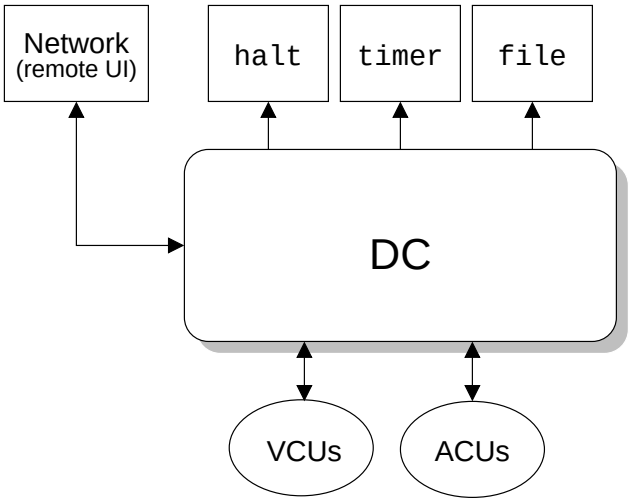

Figure 3: Conceptual diagram of debug controller functions, with kernel-managed services at the top and the user design at the bottom

3.3 Runtime Interface

On the client (user) side, the runtime interface to the system appears as a general-purpose shell with a variety of commands to load, stop, and monitor a process. In the current incarnation of this environment, the user interface is implemented entirely in Matlab to provide a very high level of integration with the original design model. The server (hardware) side of the interface is implemented within the DC logic and operates through a network interface managed by the OS. The DC parses commands coming from the network and translates them into OS system calls and control signals sent to the VCUs/ACUs.

The first thing a user must do is load the process onto the hardware. The user's design is actually a form of soft core wrapped inside a kernel interface, as described in Sect. 2. Standard input and output is determined by the loading arguments, allowing a process to communicate through the console, attached physical interfaces on the platform, or virtually through existing files or pipes. How an FPGA is programmed and how process data is actually routed around the platform is a fairly complicated task performed by the OS, and beyond the scope of this paper. At load time, the user also has the option to start the process with a normally running clock, or in a halted state to manually control the clock from the beginning. Once the process is running normally, the user is free to monitor file contents and observe any output on the console.

One of five events can cause the system to stop running and prompt the user: an expiring timer, an assertion failure, a breakpoint trigger, manual user interruption, or any general exception that could be detected by the OS (i.e. resource exhaustion, critical temperature, hardware failure, etc.). Once the process is halted, the user has a set of commands to investigate the system, update variable contents, and proceed with execution. Some of these commands are listed in Table 1.

Table 1: Some examples of runtime shell commands

`load`	Load a process onto a free FPGA
`halt`	Stop a specified process as soon as possible
`runfor`	Run the process for one or more clock cycles
`cont`	Run the process until the next exception
`break`	View, enable, or change a breakpoint
`view`	View a variable's value or history
`set`	Override a variable's value or source
`rewind`	Rewind the system state by n clock cycles

4. CONCLUSION

By integrating all the features described above, it is possible to exploit the vast resources of the hardware platform to assist in both functional verification of the design and algorithmic tuning. Design exploration performed on the hardware runs several orders of magnitude faster than any software simulation, and can also be exploited to avoid iterations through PAR.

Currently, many OS services and individual components have been implemented. Successive work will focus on completing the remaining services and coding the stitcher and runtime interface. Future plans include an analysis of system effectiveness under resource constraints and studies on automatic determination of variable storage and automatic detection of common hardware design bugs.

5. ACKNOWLEDGMENTS

This work was funded in part by C2S2, the MARCO Focus Center for Circuit and System Solutions, under MARCO contract 2003-CT-888.

6. REFERENCES

[1] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, R. W. Brodersen, and B. Nikolić. Rapid design and analysis of communication systems using the BEE hardware emulation environment. In *Proc. IEEE Rapid System Prototyping Workshop*, June 2003.

[2] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers*, 22(2):114–125, Mar./Apr. 2005.

[3] P.-Y. Chung, Y.-M. Wang, and I. N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on VLSI Systems*, 2(3):320–332, Sept. 1994.

[4] V. K. Garg. Observation and control for debugging distributed computations. In *Proc. Third International Workshop on Automatic Debugging (AADEBUG)*, volume 2 of *Linköping Electronic Articles in Computer and Information Science*, pages 1–12, May 1997.

[5] B. Peischl and F. Wotawa. Modeling state in software debugging of VHDL-RTL designs – A model-based diagnosis approach. In *Proc. Fifth International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 197–210, Sept. 2003.

[6] H. K.-H. So. BORPH: An OS for reprogrammable hardware. http://bwrc.eecs.berkeley.edu/People/Grad_Students/skhay/BORPH.

[7] J. Tombs, M. A. Aguirre Echanóve, F. Muñoz, V. Baena, A. Torralba, A. Fernandez-León, and F. Tortosa. The implementation of a FPGA hardware debugger system with minimal system overhead. In *Proc. Field Programmable Logic and its Applications (FPL)*, volume 3203 of *Lecture Notes in Computer Science*, pages 1062–1066, Aug. 2004.

[8] M. G. Valderas, E. de la Torre, F. Ariza, and T. Riesgo. Hardware and software debugging of FPGA based microprocessor systems through debug logic insertion. In *Proc. Field Programmable Logic and its Applications (FPL)*, volume 3203 of *Lecture Notes in Computer Science*, pages 1057–1061, Aug. 2004.

[9] Xilinx, Inc. *ChipScope Pro Software and Cores User Guide*, Feb. 2005. http://www.xilinx.com/ise/optional_prod/cspro.htm.