SIMULATION-BASED HW/SW CO-DEBUGGING FOR FIELD-PROGRAMMABLE SYSTEMS-ON-CHIP

Ruediger Willenberg

Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
email: willenbe@eecg.toronto.edu

Paul Chow

Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
email: pc@eecg.toronto.edu

ABSTRACT

We are presenting SimXMD (*Simulation-based eXperimental Microprocessor Debugger*), a tool that allows developers to debug microcontroller code and custom hardware simultaneously. SimXMD connects a GNU debugger instance to a full-system simulation of an embedded FPGA system. This enables free-roaming investigation of hardware-software interactions inside the system, including reverting back to an earlier point in simulation time. A custom memory logging mechanism enables access to variables in on-chip, off-chip and cached memory. SimXMD is *open source*, and its modular architecture facilitates extension to other embedded processors as well as different simulators and debuggers.

1. INTRODUCTION

Developers of reconfigurable embedded systems have a host of useful debugging tools at their convenience. To verify their RTL designs at the behavioral and gate levels, they can use digital system simulators [1][2]. FPGA vendors provide on-chip logic analyzer functionality [3][4] that enables the designer to trigger on and examine signals in a live FPGA system via a JTAG connection.

On the software side, to debug the C/C++ or assembler code running on an embedded processor core, designs can be downloaded into hardware and then stepped through from instruction to instruction or breakpoint to breakpoint with a debugging tool like GNU Debugger (GDB) [5] running on the host. The host connects to the on-chip system through JTAG, serial or network interfaces. If the code behaviour is not dependent on peripherals, a debugger can instead connect to an instruction set simulator and debug code on an emulated processor core. This process does not differ from debugging code for any non-FPGA embedded microprocessor system.

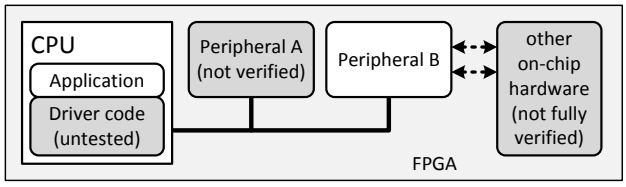

Fig. 1. System with hardware/driver combination to debug

The unique challenge of embedded systems in FPGAs can be identified in Figure 1: Designers can develop their own peripheral hardware, which commonly will require driver code to access it. Furthermore, established peripherals can communicate with other custom hardware implemented on the FPGA. In both cases, the task of verifying the hardware functionality involves writing software to interact with them. This software itself is prone to design errors. To debug a system with two untested interacting components, it is preferable if their interaction can be precisely traced.

Furthermore, to debug embedded code in the established way, an FPGA bitstream has to be generated after each hardware change, which is time-consuming. Systems that are functionally correct might also not yet meet timing, so that configuring the chip for debugging purposes is precluded.

In this paper, these challenges are met by presenting SimXMD (Simulation-based eXperimental Microprocessors Debugger), a tool that allows CPU code to be debugged in a full-system digital simulation. The designer can step through C or Assembler code with the software debugger and simultaneously observe cycle-by-cycle behaviour of any hardware signal in the simulator's wave window. Our contributions are:

- Introduction of a tool that allows easy correlation of software and hardware behaviour. Instead of a highly customized proof-of-principle, we focus on providing a system that is compatible with and can be productively used with established FPGA tools
- The capability to "go back in time" and resume debugging from a previously simulated time segment.
- Providing a software architecture that is easily extensible to other processor models and tools.

This paper is structured as follows: Section 2 explores re-

lated work. Section 3 illustrates how SimXMD operates from the user perspective, while Section 4 details how this functionality is achieved. The current project status is reported in Section 5, and Section 6 concludes.

2. RELATED WORK

Co-simulation of hardware and software is a well-established field that has produced extensive work since its inception [6]; much of the work has rightly centered on accelerating the simulation process [7]. Its focus is on verification of correct processor functionality for large amounts of code, e.g. complex operating systems and applications. Towards this goal, researchers have tried to minimize communication between simulated software and hardware with sophisticated notification and timing mechanisms[8][9] and by employing transaction-level[10] and bus functional[11] models; some approaches simulate software on higher abstractions than instruction-level[12][13]. Many of these models are very processor-centric and the simulated hardware, transactional or cycle-accurate, centers on providing a standard memory hierarchy, networking and communication peripherals.

In embedded FPGA systems, the focus lies on the opportunity to develop highly-customized peripherals, and debugging and verification tools should center on providing insight into those systems. SimXMD is targeted to hone in on very close interactions of embedded software and peripherals. Obviously a non-optimized, cycle-accurate simulation of a whole microprocessor is less time-efficient than some of the approaches referred to above; however, this is outweighed by the productivity that the tool delivers by supporting and extending existing infrastructure that FPGA designers are familiar with. Booting the Linux kernel on a MicroBlaze is outside the intended scope of SimXMD.

NIFD[14] is a debugging infrastructure that uses GDB and JTAG infrastructure to establish signal readback from the FPGA fabric. Signal state in the FPGA can be retrieved and displayed on the host and can be correlated with cross-platform GDB debugging of embedded processors on the same chip. Both debugging mechanisms can share the same JTAG connection. Crosthwaite et al.[15] are using the popular open source processor emulator QEMU to emulate code execution on a Microblaze. This processor model can interface to either C-language behavioral models of hardware components or simulator-based HDL models.

Benini et al.[16] use the GDB Remote Serial Interface in the reverse way than we do: They use a GDB client to control an instruction set simulator and retrieve information to synchronize it with a SystemC hardware simulation.

As mentioned in the introduction, FPGA vendors have established a host of debugging tools[1][2][3][4]. More recently, Xilinx has offered its ISim [17] simulator with hardware co-simulation capability. It can partition the simulation into a part being simulated conventionally in software and a part being run on an FPGA. Communication and synchronization between the two parts is achieved over a JTAG connection. ISim is very useful for co-simulating components with real-time constraints like an Ethernet core; however, debugging code running on an on-chip MicroBlaze while simulating the peripherals on the host is still not possible. ISim's functionality is therefore orthogonal to SimXMD's features.

3. SIMXMD OPERATION

3.1. Remote GDB debugging

The open-source GNU Debugger (GDB) software supports the debugging of many types of embedded processors via the *GDB Remote Serial Protocol* [18][19]. All communication between debugger and hardware is implemented through a simple protocol of request-and-reply strings that can be exchanged over TCP or UDP sockets, serial devices or POSIX pipes. Table 1 lists a few examples of common GDB remote commands.

Table 1. GDB remote command examples

Command	Meaning
?	Indicate reason why target halted
p20	Read register 0x20
g	Read the whole register set
s	Step forward by one assembler instruction
c	Continue until the next breakpoint
m65a,2	Read 2 memory bytes starting at address 0x65a
Z0,017c,4	Set instruction breakpoint at address 0x017c, type is 4-byte breakpoint

Figure 2 illustrates how GDB remote serial debugging works on a system built with Xilinx EDK: A GDB instance, optionally accessed through a Graphical User Interface, connects through a TCP socket to the Xilinx Microprocessor Debugger (XMD), which acts as a TCP server for the remote serial protocol. The vendor-specific XMD connects through the FPGA's JTAG interface to an on-chip MicroBlaze(uB) processor using the MicroBlaze Debug Module (MDM). The MDM can either use the processor's built-in hardware breakpoints for debugging, or function as a serial communication device for a MicroBlaze binary with debug code.

In contrast to Figure 2's customary XMD setup, Figure 3 depicts simulation-based debugging with SimXMD: GDB connects via TCP to SimXMD, which now acts as a TCP server in XMD's place.

On the other end of the chain, ModelSim has initiated a system simulation based on compiled HDL code. After starting the simulation in ModelSim, a *tcl* script sets up a simple TCP server running in the background. SimXMD connects to this server as a client. The TCP server receives

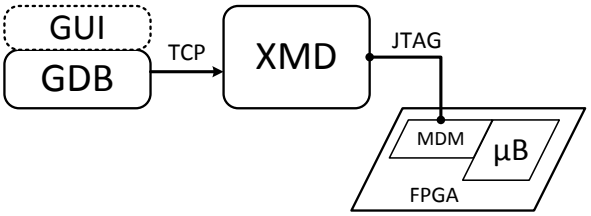

Fig. 2. Regular GDB debugging in Xilinx system

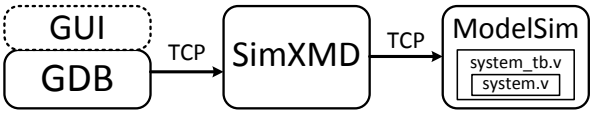

Fig. 3. Debugging with SimXMD

commands for the ModelSim command-line from SimXMD and returns the output of the initiated commands. Since the TCP server is a background process, commands can still be entered by hand into the ModelSim command-line, for example to run up to a certain simulation time without any debugger control.

SimXMD's main task is to parse the requests that GDB sends and translate them into ModelSim operations. To report the state of processor, memory and variables back to GDB, SimXMD monitors the simulated MicroBlaze's *Trace Port*, which tracks all cycle-to-cycle processor operations.

The user experience from the GDB perspective is identical to the earlier case: The user can step through the C or assembler source, set and remove breakpoints and watch local and global variables. But in addition to that, the user can simultaneously observe the hardware state changing in ModelSim's waveform window.

Debugging with SimXMD entails the following sequence of steps:

1. Simulation model generation by EDK
2. Modification of simulation and scripts by SimXMD
3. Compilation and start of simulation by ModelSim
4. Connection of a debugger to SimXMD
5. SimXMD translates between debugger and simulator

3.2. SimXMD debugging modes

SimXMD can switch between two different modes of operation, *Run mode* and *Replay mode*. Figure 4 illustrates this with a simplified representation of the simulator's waveform window and GDB's source code window. SimXMD starts in *Run mode* (A): The reported processor state is based on the most recent time simulated, which will be indicated by a special position-locked simulation cursor (see vertical line). To continue program execution to the next breakpoint, the simulation itself is run until the breakpoint condition is reached. GDB will then request updated information for all processor state and monitored variables. The information reported by SimXMD will again be based on the most recent simulation time.

The user can now pick any time inside the already simulated timeframe by marking it with a regular simulation cursor (dotted line) and directing SimXMD to resume debugging at this time(B). GDB is at this moment idle and waiting for user input; therefore, it still shows the most recent processor state from Run mode. By asking GDB to *step* or *continue*, the user initiates execution from the chosen time.

When the next break condition is reached in the simulated data, SimXMD marks the time with the locked cursor and GDB updates all displayed state according to that time; the system has transitioned into *Replay Mode*(C). The user can continue debugging in the pre-simulated data, with the locked cursor being moved forward in time on each step.

SimXMD can transition back into *Run Mode*(D) for two reasons: The user can deliberately choose to switch back to Run mode, and therefore to continue debugging and simulating from the most recent simulation time forward. Secondly, the user can keep debugging in Replay mode up to the most recent simulated time. If SimXMD can't find another break condition in the pre-simulated timeframe, it will automatically transition to Run mode and continue simulation until a break condition happens.

Replay mode capability offers two important advantages:

1. In *Run mode*, it is easy to go past a critical point where things are starting to go wrong in hardware. At the point where this hardware behaviour is fully identifiable, the debugger has already passed the critical code segment. Replay mode enables the designer to easily jump back in time and re-examine the code; because SimXMD logs all memory modifications (see Section 4.2), the designer can even decide at any point to examine additional variables or memory locations.
2. Especially for larger systems, *Replay mode* is significantly faster than *Run mode* because SimXMD only needs to search for logged trace port events instead of simulating the whole system step by step. This means that a system can be pre-simulated for a longer stretch of time without the designer's involvement, and the designer can then examine interesting segments in *Replay mode*.

4. IMPLEMENTATION

4.1. System and testbench modifications

For the purpose of simulation, Xilinx Embedded Development assembles a complete source code hierarchy that represents the system inside the FPGA. Furthermore, it gener-

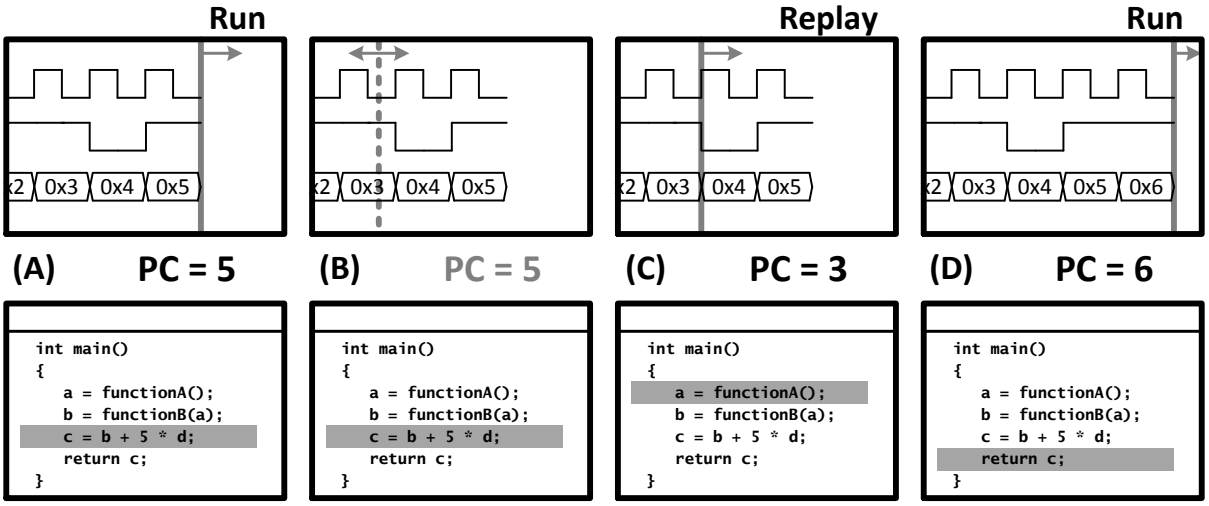

Fig. 4. SimXMD debugging modes (above: ModelSim waveform window - below: GDB window)

ates a Verilog testbench file that instantiates the whole system and generates external stimuli like the clock and reset signals. Lastly, it provides *tcl* scripts that automate compilation, simulation and wave window setup.

SimXMD uses the Microblaze's built-in trace port to monitor the register state, the current program counter and the nature of memory accesses. To facilitate debugging functionality, it modifies some of the files generated by EDK and adds others:

- System definition *(system.v)*: To be accessible from the testbench[1], the trace port signals of the MicroBlaze instance need to be connected to wires. Our modification inserts wires and connects them to the instance's trace port.
- Testbench *(system_tb.v)* Since the internal MicroBlaze signals are not accessible, the register file is not directly available. Instead, the testbench manages its own copy of the register file, which is updated according to the trace port register access signals. Furthermore, instruction memory breakpoints are inserted by comparing the breakpoint registers with the trace port program counter. In *Run mode*, an activated breakpoint will interrupt the simulation run by calling the Verilog *$stop* task.
  The testbench modifications are actually written into a separate file, and then included into the file with a single line `` `include `` instruction. This keeps the testbench simple and does not conflict with other testbench modifications like input stimuli or assertion testing on outputs.
- Simulator scripts *(*.do)*: SimXMD adds several *tcl* scripts that execute steps to bring the simulator into a state in which it is ready to work with SimXMD and GDB, namely: Compile the simulation model and testbench, set an environment variable to link a dynamic library into the simulation, start the simulation, setup the waveform window, run the simulation up to the start of *main()* and open a TCP server port to receive SimXMD commands. Another very convenient script function is the addition of SimXMD-specific menu options to the waveform window. Because of this, SimXMD can operate invisibly in the background and does not clutter the already contested screen real estate.

[1] In the currently used Verilog and VHDL dialects it is not possible to access signals in VHDL components lower in the hierarchy from the Verilog top-level or testbench. Since most Xilinx EDK components are VHDL-based, only the information in the system top-level file, which can be generated in Verilog, is accessible for the testbench.

4.2. Memory access logging and retrieval

Accessing the state of memories in the system is not straightforward for a host of reasons:

- All the processor state including the registers and the program counter is read from the MicroBlaze's trace port. However, because the processor is pipelined and different actions are taken in different stages, the trace port signals are delayed a few cycles to be able to indicate all the processors actions for a single instruction in the same cycle. As a consequence, memory writes that happen logically after the currently traced instruction can have already executed, and a corresponding variable could be shown with a wrong value given the current breakpoint.
- A related issue arises out of the fact that contrary to a real software breakpoint exception, the pipeline is not

flushed and memory operations are not finished before polling memory values - since the simulator stops in *Run mode*, memory accesses are interrupted together with the processor.
- Accessing data in a simulated external memory is hard, since the organization of data in the external memory simulation models might be vendor-dependent, and might even change as a result of different memory controller settings.
- To cover all data storage accurately, cache behaviour would have to be tracked and simulated caches be readable. While the trace port reports cache operations, the structure of the cache could change between different microarchitecture versions, making it complex to keep our access patterns correct and up-to-date for different versions of the MicroBlaze.
- Lastly, contrary to signals, ModelSim by default only stores the last valid state of instantiated memories. Our *Replay mode* would therefore not be able to retrieve earlier memory state from these models.

For all these reasons, SimXMD keeps track of memory changes independently instead of relying on ModelSim. Figure 5 illustrates the mechanism. On every write access indicated by the MicroBlaze trace port, our modified Verilog testbench calls a task named *$memlog*. This task has been implemented in C code and compiled into a shared library that can be called by ModelSim through the *Verilog Programming Interface* (VPI).

The *$memlog* task accepts the relevant write access information of time, address, data and byte enables and stores it in a tree-based dynamic data structure that allows efficient retrieval of memory state by SimXMD. *$memlog* allocates a shared memory section that SimXMD can also access, therefore allowing efficient log management without any communication overhead. Race conditions cannot happen because SimXMD only retrieves memory state when ModelSim is interrupted.

$memlog stores any MicroBlaze write accesses happening after the start of the simulation. If memory at the requested address has never been modified by the requested time, SimXMD instead checks for the initialization state with which the simulation had been started. This is usually true for the processor's instruction memory, but also for unmodified global variables.

$memlog's limitations pertain mostly to shared access of memory with other processors or peripherals (also see Section 5.3) .

4.3. Software Architecture

4.3.1. Language and component framework

SimXMD has been written in C++ using the Qt Application Framework [20], which is available for open source projects

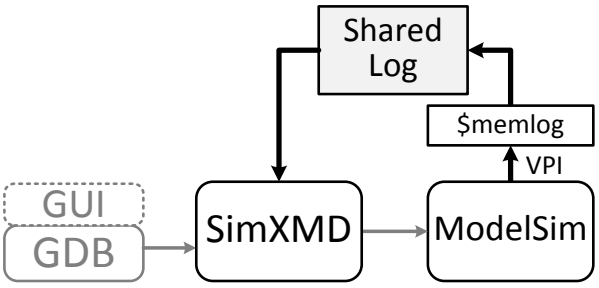

Fig. 5. Memory access logging and retrieval between ModelSim and SimXMD

under the *Lesser GNU Public License* (LGPL) [21]. Qt provides portable Graphical User Interface (GUI) components as well as portable components for other operating system functionality like sockets. All relevant functionality is provided in the default command-line mode. However, a GUI can be enabled by command-line option; it offers a convenient alternative to specify certain settings and splits up the log output into three separate windows for debugger, processor and simulator.

4.3.2. Modularity

SimXMD makes use of polymorphism in C++ to provide for extensibility. Figure 6 shows a simplified version of SimXMD's class hierarchy. The debugger interface, processor and simulator interface are modeled in the abstract base classes *debug_base*, *core_base* and *sim_base*. They provide generic functionality to communicate between each other that is independent of what specific processor, debugger and simulator are used. In our current configuration, these interfaces are inherited and implemented by *core_Microblaze*, *debug_GDB* and *sim_Modelsim* respectively. The application's main object *SimXMD* holds pointers of the three base class types. However, they actually each point to an instance of an inherited class. In the current implementation, only one child class each exists, so that only a MicroBlaze processor can be simulated with ModelSim and debugged with GDB. Support for a new processor, for example the Altera Nios II, can now be added to the tool with limited effort by implementing a new class inheriting from *core_base*, without communication to the debugger or simulator changing. This class would need to handle the different structures of the Nios II simulation model and the Altera SoPC Builder projects. The same inheritance-based process works for other debuggers and other simulators, as long as they support a basic set of functionality that is essential for SimXMD operation. Note that a number of non-GDB debuggers support GDB's remote serial protocol; in these cases, no other debugger class is required.

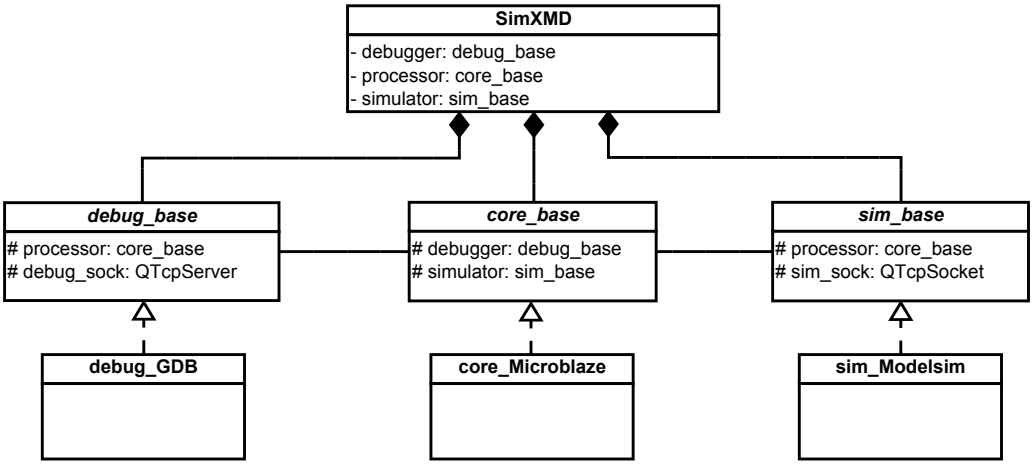

Fig. 6. SimXMD class hierarchy

4.3.3. Portability

A positive side-effect of using the Qt framework for designing the tool is operating system independence. An application that handles all of its user interface, file accesses and communication channels through Qt classes can be compiled without changes for 32- and 64-bit versions of Linux, Windows and MacOS. While this has not been evaluated yet, SimXMD is expected to work without major modifications with the Windows versions of Xilinx EDK (13.1 and higher) and ModelSim.

5. PROJECT STATUS

The development and evaluation of SimXMD up to this point have been carried out with the following software and components:

- OpenSUSE Linux 12.4 64-bit
- Xilinx Embedded Development Kit 14.2, including Xilinx SDK and GDB 7.3.5 for MicroBlaze
- Xilinx MicroBlaze processor version 8.40a
- Mentor Graphics ModelSim SE 10.1c
- DDD 3.3.12, KDbg 2.5.2, Nemiver 0.9.4

5.1. GDB support

SimXMD currently supports the following GDB requests:

- **?** - Indicate reason for target halt; currently this is always S05, the POSIX code for a trap condition
- **p,g** - Read one specific or all processor registers
- **s** - Step one assembler instructions
- **c** - Continue to next breakpoint
- **Z0** - Set instruction breakpoint
- **z0** - Remove instruction breakpoint
- **m** - Read from memory
- **H** - Choose which thread subsequent requests apply to; this is acknowledged without action because MicroBlaze does not support multithreading
- **D** - Disconnect

For all other requests, SimXMD answers with a legal empty string (plus parity), therefore cleanly indicating that this feature is not implemented. With the current feature set, GDB can connect to the simulated processor, inquire about all essential state, execute code line by line or up to breakpoints, and indicate the values of variables; most unsupported functionality is related to multithreading.

SimXMD supports the Eclipse-based debugger frontend that is part of Xilinx Software Development Kit, and in principle any other debugger frontend using the GDB remote protocol. The *mb-gdb* command-line as well as the popular *DDD*, *KDbg* and *Nemiver* GUI frontends can be started automatically through menu buttons that SimXMD integrates into the ModelSim wave window; this unfortunately does not work for the Xilinx SDK debugger because of limitations in Eclipse's command-line interface.

5.2. Performance

To judge SimXMD's impact on both simulation and debug performance, we ran three experiments. For all of them, we used a very small MicroBlaze system (default MicroBlaze core at 100MHz, 64KB of BlockRAM, no external RAM, AXI bus, one GPIO device) implemented for a Xilinx Spartan 6 FPGA on a Digilent Atlys board. The BlockRAM was initialized with an application writing up to 32KBytes of data byte by byte into a contiguous block of on-chip RAM.

5.2.1. Simulation time

We first measured the time that Modelsim needs on the test system to simulate writing different-sized blocks of data,

Table 2. Simulation time in seconds

Write size	w/o SimXMD	w/ SimXMD
1KB	6.9	7.3
2KB	13.8	14.5
4KB	27.3	29.0
8KB	54.9	57.7
16KB	109.0	117.1
32KB	218.9	231.7

Table 3. Memory allocation for *$memlog*

Write size	Log memory
1KB	40KB
2KB	56KB
4KB	88KB
8KB	152KB
16KB	280KB
32KB	536KB

once with an unmodified simulation model as generated by Xilinx EDK, and once with a system with SimXMD modifications. Table 2 shows the results. On average, the modifications add 6.0% of simulation time. We assume that most of this time is spent in the VPI extension for memory logging, while the testbench modifications only add a very small amount of Verilog to process in relation to the rest of the simulation model. It is worth noting that simulating any more complex system would make the time spent on SimXMD additions relatively smaller.

5.2.2. Memory usage

Table 3 lists the memory allocated by the VPI *$memlog* module for different amounts of written data. It grows linearly after an initial allocation of about 24KB; note that because the basic logging unit is a processor word, writing the same overall amount of memory with 32-bit words would have only required about a quarter of the indicated size. Obviously a pattern of completely random accesses through the whole address space would inherently result in a much larger allocation, but that would not reflect realistic application behaviour. Overall, we believe that the memory allocated by *$memlog* is small in comparison to ModelSim's memory use to log waveforms.

5.2.3. Responsiveness to debug code stepping

To compare performance between in-system debugging and SimXMD debugging, we wrote a GDB script that executes 50 *step* requests cycling through the C code line by line. We measured the execution time of this script for the real system as well as SimXMD in *Run* and *Replay* modes; Table 4 lists the results, averaged for a single step instruction. Because of the delays incurred by JTAG communication, SimXMD actually performs faster than hardware under these specific circumstances. Obviously the real system would be orders of magnitude faster than SimXMD when waiting on a rare breakpoint condition. However, going through code step by step is a common debugging task and therefore relevant to productivity. Simulating more complex systems would also slow down SimXMD *Run* mode, while *Replay* mode would not be significantly impacted.

Table 4. Average time for a single code line step

Hardware w/ JTAG	1.350 s
SimXMD Run mode	0.850 s
SimXMD Replay mode	0.313 s

5.3. Limitations

Debugging with SimXMD has a few limitations; some of these are intrinsic to the way that we are using simulation, while others will hopefully be removed in future work:

- Trace port delay: All of an instructions' actions are reported by the trace port after the instruction has completed and left the pipeline. Consequently, some of these actions are visible in the periphery a few cycles before the current debug time. For most actions this delay is between 2 and 4 cycles, but if the processor is stalled for a bus access the gap can be significantly larger. If the user keep this in mind, there are no practical disadvantages to it.
- Since SimXMD only observes simulated system behaviour, it can inherently not modify register, variable and memory contents while debugging. Corresponding GDB remote requests are denied as unsupported.
- As SimXMD only logs memory changes performed by the processor, volatile memory locations can not be guaranteed to be shown correctly. If other processors or components share direct memory access with the debugged processor, changes by these actors would currently not be observed, and a satisying and universal solution to this problem is hard to conceive.
- Since SimXMD relies on the MicroBlaze's externally available trace port information, no special register except the MSR is accuratedly reported in GDB.

5.4. Availability & Cooperation

The most recent version of SimXMD is available as open source at *http://www.eecg.toronto.edu/~willenbe/simxmd*

SimXMD is published under the Apache License 2.0 [22], essentially allowing free use and modification with attribution in non-commercial and commercial projects. While SimXMD should still be considered in beta status, we feel it can by now be productively used to investigate real-life designs, and we encourage users to give it a try.

We further invite interested parties to join us in the development and extension of SimXMD. We especially envision adding support for other FPGA processors and simulators. Please contact us under *willenbe@eecg.toronto.edu*.

6. CONCLUSION

This paper introduced SimXMD, a tool to debug soft processor code together with hardware components. SimXMD currently provides all essential features to debug code on a Xilinx MicroBlaze processor with GDB and ModelSim without significantly impacting simulator or debugger performance. More debugging features and extension to other processors and tools are planned as future work.

Acknowledgment

We thank Xilinx, CMC Microsystems, Embedded Systems Canada (emSYSCAN) and NSERC for supporting our research.

7. REFERENCES

[1] Mentor Graphics ModelSim. [Online]. Available: http://www.model.com/

[2] Cadence incisive enterprise simulator. [Online]. Available: http://www.cadence.com/products/sd/enterprise_simulator /pages/default.aspx

[3] Cadence Incisive Enterprise Simulator. [Online]. Available: http://www.xilinx.com/tools/cspro.htm

[4] Altera SignalTap II. [Online]. Available: http://www.altera.com/products/software/quartus-ii/subscription-edition/verification-board-level/swf-ver-bdlevel.html

[5] GDB. the GNU Project Debugger. [Online]. Available: http://sources.redhat.com/gdb/

[6] J. A. Rowson, "Hardware/software co-simulation," in *Proceedings of the 31st annual Design Automation Conference*, ser. DAC '94. New York, NY, USA: ACM, 1994, pp. 439–440. [Online]. Available: http://doi.acm.org/10.1145/196244.196458

[7] V. Živojnovic and H. Meyr, "Compiled hw/sw co-simulation," in *Proceedings of the 33rd annual Design Automation Conference*, ser. DAC '96. New York, NY, USA: ACM, 1996, pp. 690–695. [Online]. Available: http://doi.acm.org/10.1145/240518.240649

[8] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr, "Hysim: A fast simulation framework for embedded software development," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, 2007, pp. 75–80.

[9] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, 2008, pp. 290–295.

[10] S. Swan, "Systemc transaction level models and rtl verification," in *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006, pp. 90–92.

[11] Xilinx AXI Bus Functional Model. [Online]. Available: http://www.xilinx.com/support/index.html/content/xilinx/en /supportNav/ip_documentation/interconnect_infrastructure /axi_bus_functional_model.html

[12] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[13] R. Bedichek, "Some efficient architecture simulation techniques," in *Winter 1990 USENIX Conference*, 1990, pp. 53–63.

[14] H. Angepat, G. Eads, C. Craik, and D. Chiou, "Nifd: Non-intrusive fpga debugger – debugging fpga 'threads' for rapid hw/sw systems prototyping," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 356–359.

[15] P. Crosthwaite, J. Williams, and P. Sutton, "A unified emulation/simulation environment for reconfigurable system-on-chip development," in *Field-Programmable Technology (FPT), 2011 International Conference on*, 2011, pp. 1–8.

[16] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, "Systemc cosimulation and emulation of multiprocessor soc designs," *Computer*, vol. 36, no. 4, pp. 53–59, 2003.

[17] Xilinx ISE Simulator. [Online]. Available: http://www.xilinx.com/tools/isim.htm

[18] GDB User Guide: Remote Serial Protocol. [Online]. Available: http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html

[19] (2008) EMBECOSM Howto: GDB Remote Serial Protocol - writing a RSP server, Application Note 4, Issue 2. [Online]. Available: http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html

[20] Qt Framework. [Online]. Available: http://qt-project.org/

[21] GNU Lesser General Public License. [Online]. Available: http://www.gnu.org/licenses/lgpl.html

[22] Apache License, Version 2.0. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.html