SimXMD: Integrated debugging of C code and hardware components

Ruediger Willenberg, Paul Chow
Electrical and Computer Engineering, University of Toronto
Toronto, Ontario, Canada
willenbe@eecg.toronto.edu
pc@eecg.toronto.edu

(Demonstration Paper)

***Abstract*—In our demonstration, we present SimXMD, a tool that enables developers to debug microcontroller code and custom hardware simultaneously. SimXMD (*Simulated eXperimental Microprocessor Debugger*). SimXMD connects a GNU Debugger instance to a ModelSim instance simulating an embedded FPGA system with a Xilinx Microblaze processor. We will demonstrate debugging a multiprocessor FPGA system where the processor cores are connected through custom-designed network hardware. SimXMD is Open Source, and its modular architecture facilitates extending it to other embedded processors as well as different simulators or debuggers.**

I. Introduction and related work

Developers of reconfigurable embedded systems have a host of useful debugging tools at their convenience. To verify their RTL designs on 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 or 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.

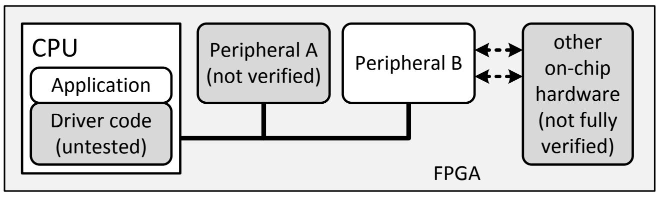
CPU
Application
Driver code (untested)
Peripheral A (not verified)
Peripheral B
other on-chip hardware (not fully verified)
FPGA

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 peripherals, which usually will need driver code to access them. 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. Also, systems that are functionally correct might not yet meet timing, so that downloading to the chip for debugging is precluded.

SimXMD meets these challenges by allowing the 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.

Until EDK Version 10, Xilinx offered the now-deprecated Virtual Platform Generator [6], which simulated older versions of the MicroBlaze processor as well as its buses and standard peripherals, though not the custom peripherals that are an essential advantage of FPGA systems.

More recently, Xilinx has offered its ISim [7] simulator with hardware co-simulation capability. It can partition the simulation into a part being simulated conventionally in software, and one 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.

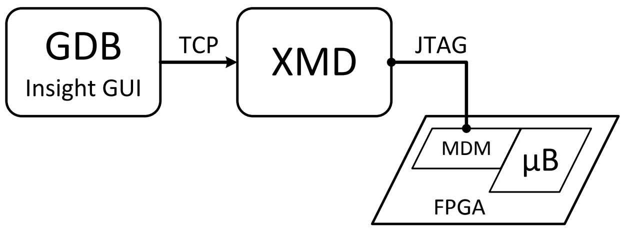

Fig. 2. Regular GDB debugging in Xilinx system

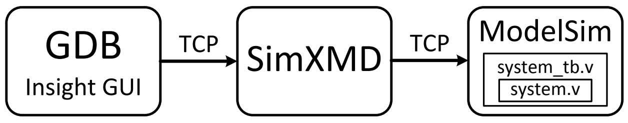

Fig. 3. Debugging with SimXMD

II. Operating concepts

A. Basics of remote GDB

The open-source GNU Debugger (GDB) software supports the debugging of many types of embedded processors via the *GDB Remote Serial Protocol* [8][9]. 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.

Figure 2 illustrates how GDB remote serial debugging works on a system built with Xilinx EDK: A GDB instance with the Insight Graphical User Interface [10] 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.

B. SimXMD system setup

In contrast to Figure 2, which shows the customary setup using XMD, Figure 3 depicts a SimXMD configuration: 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 compiled the testbench and system files and started the system simulation. For the purpose of simulation, Xilinx EDK generates a Verilog testbench file that instantiates the whole FPGA system and generates external stimuli like the clock and reset signals. SimXMD modifies this testbench to include further signals and processes that help with the MicroBlaze debugging. 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 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 command-line, for example to run ModelSim up to a certain point in time without debugger control.

SimXMD's task is to parse the requests that GDB sends and translate them into ModelSim instructions:

- Register read requests are converted to commands that query the simulation model's processor registers.
- Breakpoints are managed as a set of address registers in the testbench; their values are compared every cycle with the processor's program counter. If a breakpoint is reached, the simulation is interrupted.
- Memory reads need to be translated from the requested global memory address to the instances and addresses of memory blocks that compose the complete processor memory.

C. System and testbench modifications

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 support the current functionality, the following changes are being made to the system top-level file as well as to the testbench:

- *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.
- *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.
- *system_tb.v:* Instruction memory breakpoints are inserted by comparing the breakpoint registers with the trace port program counter and, on identity, calling the Verilog *$stop* task to interrupt simulation.

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 eases other common testbench modifications, e.g. the inclusion of board components external to the FPGA, like DRAM.

D. Sequence of operation

To debug with SimXMD, the following steps have to be taken:

1) Generate a simulation model with EDK and start ModelSim

[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.

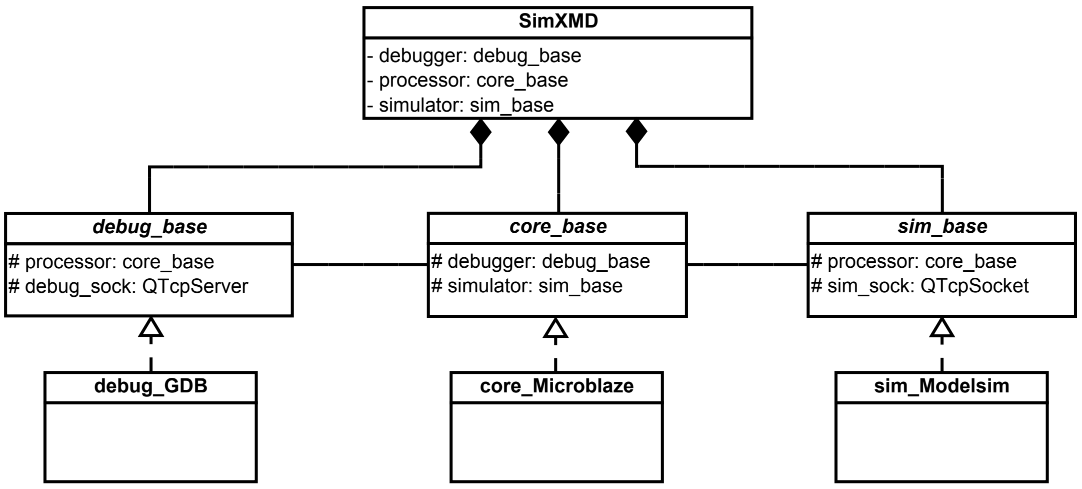

Fig. 4. SimXMD class hierarchy

2) Start SimXMD (optionally supply project path information on the command-line)
3) If not done with command-line arguments, choose an EDK project path (and, in multi-core systems, the core to be debugged) through the GUI
4) SimXMD will parse the project files for relevant information like the structure of memory in the simulation model and the binary to be debugged
5) SimXMD will modify the system and testbench files with the additions described in Section II-C
6) Compile the simulation model, load the simulation and enable the wave display
7) Start the tcl script to run the ModelSim TCP server
8) Connect SimXMD to the ModelSim server
9) SimXMD runs the simulation until either the Microblaze starts execution (Address 0) or up to the beginning of *main()*
10) Start GDB and connect to SimXMD's server port; GDB can optionally be started from SimXMD. GDB will request some initial information like the program counter state; SimXMD will request this information from ModelSim and reply to GDB; GDB will indicate the current position in the source code (First assembler instruction in the C runtime or the first line of *main()*)
11) Continue debugging as you see fit

III. Software Architecture

A. Graphical User Interface

SimXMD has been written in C++ under use of the Qt Application Framework [11], which is available for open source projects under the *Lesser GNU Public License* (LGPL) [12]. Qt provides portable Graphical User Interface (GUI) components as well as portable components for other operating system functionality like sockets. While a GUI is not strictly necessary for SimXMD functionality at this point (all relevant operating parameters can be provided through command-line arguments), a GUI offers additional flexibility, e.g. the option of changing settings without re-starting SimXMD. Currently, the SimXMD GUI also allows a closer look at its operation through three log windows that report GDB and ModelSim communication and processor-specific information.

Furthermore, interaction with SimXMD might be necessary for envisioned future functionality. An example would be the capability to go back to an earlier point in time in the simulation where a hardware misbehaviour has been pinpointed, and show the related position in the C code. This function would need to be requested by the user through the GUI; SimXMD would then read the current time cursor position from ModelSim, and then request model state information for that time, so that the next GDB requests can be answered referring to that point in time

B. Modularity

SimXMD makes use of polymorphism in C++ to provide for extensibility. Figure 4 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 debugger, processor and simulator are used. In our current configuration, these interfaces are inherited and implemented by *debug_GDB*, *core_Microblaze* and *sim_Modelsim* respectively. The application's main GUI object *SimXMD* holds pointers of the three base class types. However, they actually each point to an object of an inherited class, as defined by command-line options or even changed later during operation. In our 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

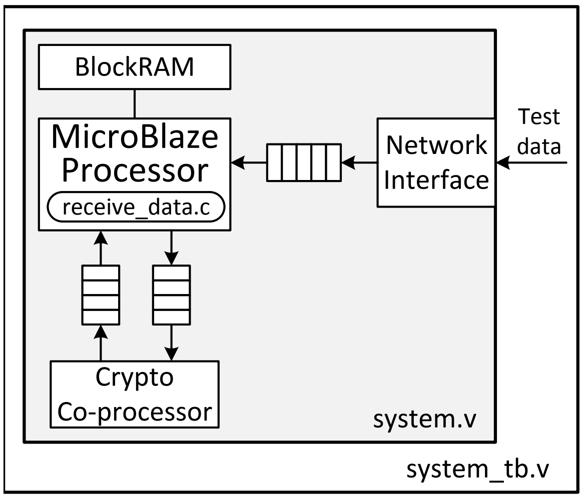

Fig. 5. Processor system for debug demonstration: MicroBlaze with on-chip memory, network peripheral and cryptographic co-processor

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.

C. Platform Independence

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 we have not evaluated this yet, we expect SimXMD to work without problems with the Windows versions of Xilinx EDK and ModelSim.

Finally, the simultaneous debugging of several processor instances on the same chip could be very useful for parallel systems. SimXMD should support this at some point.

IV. Availability & Cooperation

The most current 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 [13], essentially allowing free use and modification with attribution in non-commercial and commercial projects.

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*.

We also hope for vendor support, at this point especially from Xilinx, to enhance compatibility with their intellectual property and tools.

V. Demonstration Overview

Figure 5 shows the simple processor system that we are using for our SimXMD demonstration: A MicroBlaze processor is connected via 32-Bit FIFOs (*Fast Simplex Links*) to two peripherals, a network device and a cryptographic co-processor. The network peripheral sends received network packets with encrypted data to the processor. The software running on the CPU strips the header off the packet, sends the data to the co-processor for decryption and reads the decrypted data back.

For the demonstration, a testbench provides the simulated system with input packets. The system shows two issues:

1) The decrypted packet data starts with a wrong data word and misses the last data word.
2) After resetting the system once, the co-processor produces completely wrong data.

With SimXMD, we can now debug these problems by following the code execution and the peripheral operation at the same time.

VI. Conclusion

We demonstrated SimXMD, a tool to debug C and assembler code on processors during full-system FPGA simulation. SimXMD currently provides for all essential features to debug on a Xilinx MicroBlaze processor with GDB and ModelSim. More debugging features, as well as extension to other processors and software tools, is planned as future work.

Acknowledgment

We would like to extend our gratitude to Xilinx, CMC Microsystems, Embedded Systems Canada (emSYSCAN) and NSERC for supporting our research.

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] Xilinx Virtual Platform Generator. [Online]. Available: http://www.xilinx.com/itp/xilinx10/help/platform_studio /ps_c_dbg_debugging_sw_vp.htm
[7] Xilinx ISE Simulator. [Online]. Available: http://www.xilinx.com/tools/isim.htm
[8] GDB User Guide: Remote Serial Protocol. [Online]. Available: http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html
[9] (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
[10] Insight GDB GUI. [Online]. Available: http://sources.redhat.com/insight/
[11] Qt Framework. [Online]. Available: http://qt-project.org/
[12] GNU Lesser General Public License. [Online]. Available: http://www.gnu.org/licenses/lgpl.html
[13] Apache License, Version 2.0. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.html