An Ethernet-Accessible Control Infrastructure For Rapid FPGA Development

Andrew Heckerling, Thomas Anderson, Huy Nguyen, Greg Price, Sara Siegal, John Thomas
MIT Lincoln Laboratory, Lexington, MA 02420
{heckerl, toma, hnguyen, gprice, ssiegal, jthomas}@ll.mit.edu

Introduction[1]

Field Programmable Gate Array (FPGA) technology is widely used in many application areas such as intensive numerical processing, sophisticated control, and high-speed IO interfacing. One key factor for success in such vast diverse application space is the flexibility to customize (configure) the on-chip logic fabric and embedded specialized silicon macros to produce designs that are streamlined and highly integrated. This "flexibility", on the other hand, also means a minimally-supported development environment, which makes application development and debugging more difficult. An analogy is a PC board with no operating system or BIOS. The lack of a minimum standardized control infrastructure on the FPGA is also a huge barrier for meaningful interaction with the outside world. As a result, input-output and control structures for FPGAs are frequently created from scratch for each project.

This paper addresses the above limitations with a computer-accessible control structure on the FPGA. This small-footprint control infrastructure provides the developer with a foundation and scaffolding to quickly build up an application. Through Ethernet, external software can observe and control internal states of the application function core being developed. The infrastructure is essentially a container for housing the application-specific intellectual property cores (IPs).

The container has enough functionality to serve as a computer-FPGA control interface for a real-time FPGA-based processor system. Its Gigabit Ethernet interface and remote-DMA capability also make it a reasonable platform for hosting medium-grade computer–FPGA co-processing.

Application IP Container

The FPGA IP container, highlighted in **Figure 1**, is accessible through software calls from the computer. The software library allows the application software on the computer to request reads and writes to the FPGA address space. The library handles (behind the scenes) the details of formatting one or more request GigE packets and interpreting the returned results, abstracting the process to simple remote-DMA calls.

[1] **This work is sponsored by the United States Air Force, under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Air Force.**

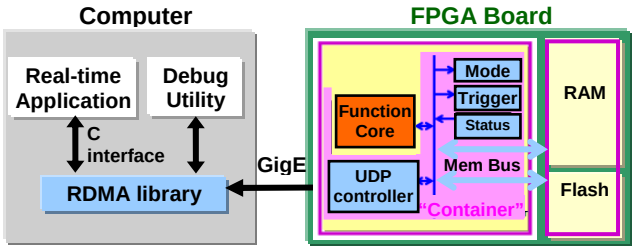

Figure 1: Container provides an open control interface from a computer to the application-specific function core(s)

The container structure consists of four major FPGA components. The first major component is a UDP controller, which implements the UDP protocol and decodes packets into DMA commands. The second is the DMA controller, which executes DMA transactions on a Wishbone bus. The third is the Wishbone bus itself, which is a simple interconnect between the DMA controller and a variety of registers and peripherals. Finally, there are a number of Wishbone peripherals that are useful for FPGA development.

UDP Controller

The UDP controller receives packets from an Ethernet MAC and decodes properly addressed and formatted UDP packets into commands for the DMA controller. Once the command has been executed by the DMA controller, the resulting status and data responses are re-packaged into UDP messages and reported back to the network address that made the request.

UDP was chosen as a transport-layer protocol for efficiency reasons and because its stateless nature made it more suitable for implementation in digital logic than a more complicated protocol like TCP. The command-response protocol implemented on top of UDP was designed for simple translation into commands for the DMA controller.

DMA Controller

The DMA controller receives commands to read or write a block of address space and translates them into the required master read or write cycles on the Wishbone bus. The status of the completed read or write transaction and, in the case of a read, the resulting data is reported to the upstream controller that sent the command. In our design, this is the UDP controller, but the DMA controller is not UDP-specific. Transaction size can range from a single four-byte word to 8 KB of data, and the DMA controller supports both constant-address and ascending-address reads and writes.

WISHBONE Bus

The WISHBONE SoC Interconnect Architecture is an open source, public domain bus architecture for System-on-Chip architectures.[2] The WISHBONE specification, interconnect IP, and a variety of peripherals are freely available from the OpenCores web site.

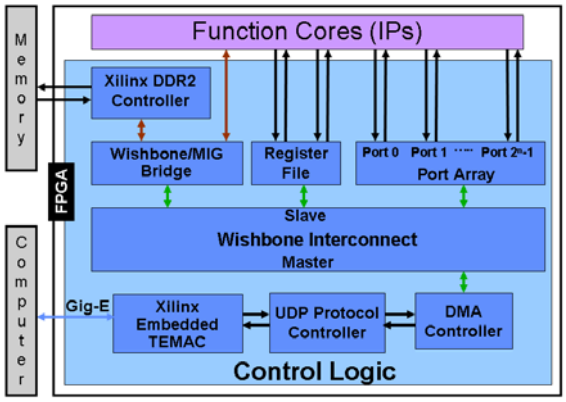

Figure 2: FPGA Control Infrastructure

Our design uses a 32-bit implementation of the WISHBONE bus to tie together the DMA controller and a variety of peripherals, as shown in **Figure 2**. Each peripheral uses a portion of the WISHBONE bus address space. An "interconnect" module is responsible for partially decoding the address from every WISHBONE cycle and directing the transaction to the proper peripheral. Our "interconnect" module is derived from one produced by the WISHBONE Builder[3], but contains modifications to improve flexibility in attachment to peripherals and robustness in the case of peripheral failure.

Peripherals

A number of WISHBONE peripherals have been created in order to provide various memory-mapped functions. Some of these peripherals are strictly on the FPGA. For example, we have created a register file which can be written and read via the WISHBONE bus and provides general purpose control and status to other portions of the design. A "port array" peripheral provides memory mapped FIFO ports, allowing a block of data to be streamed out of a FIFO interface, into processing logic, and back into another FIFO interface to be later read back by the DMA controller. This has been very useful for testing streaming-type processing logic, which often expects data to arrive in a FIFO-like interface.

A dual-port memory controller bridge has also been constructed to interface between the WISHBONE bus and a DDR2 controller created by the Xilinx Memory Interface Generator (MIG). In addition to connecting the WISHBONE bus to the DDR2 controller, this bridge contains a second "passthrough" port, which allows data-processing logic to interface to the bridge as it would have interfaced to the memory controller. This design allows high-speed processing logic to share the memory with the lower-speed control and debugging logic. In addition to DDR2 memory, FPGA block-RAM can also be connected to the Wishbone bus with minimal interface logic.

[2] http://www.opencores.org/projects.cgi/web/wishbone/wishbone
[3] http://www.opencores.org/projects.cgi/web/wb_builder/overview

Results

Our controller infrastructure was implemented and tested on Virtex 5 110LXT and 50SXT FPGAs. The software library has been tested under Windows XP/Cygwin and VxWorks.

On the Virtex 5-50SXT, the following resources were used:

LUTs	FFs	BRAM Kbytes	Clock rate
5941/32640 (18%)	10118/32640 (20%)	112.5 / 594 (19%)	125 MHz

The communication rate with the computer reached 13 MB/s, the highest rate supported by our minimally-optimized software library. It is estimated that the FPGA infrastructure can exceed this limit by a wide margin and reach Gigabit Ethernet speeds or higher.

This control infrastructure was used as a computer-FPGA interface for a FPGA-based processor in MicroTCA platform, as shown in **Figure 3**. The MicroTCA environment has a GigE hub connecting to all payload slots in the system via backplane high-speed connection.

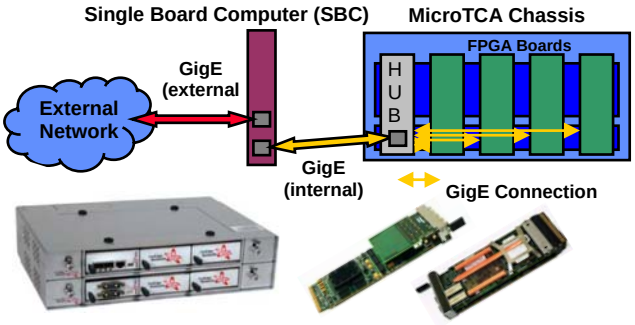

Figure 3: MicroTCA System

Future Work

Possible future work includes extending the container framework to a multi-chip environment as shown below in **Figure 4**.

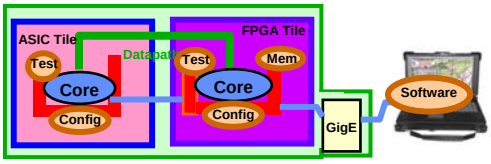

Figure 4: Multi-Chip Extension

The container can also be further developed to support integration of FPGAs as co-processors in the PVTOL[4] framework.

[4] H. Kim, N. Bliss, R. Haney, J. Kepner, S. Mohindra, S. Sacco, G. Schrader, E. Rutledge. "PVTOL: A High-Level Signal Processing Library for Multicore Processors." *High Performance Embedded Computing Workshop 2007.* Lexington, MA. September 2007.