Hardware/Software Infrastructure for ASIC Commissioning and Rapid System Prototyping

Peter Reichel Jens Döge

Fraunhofer Institute for Integrated Circuits IIS
Design Automation Division EAS
Zeunerstr. 38, 01069 Dresden, Germany
E-mail: {peter.reichel,jens.doege}@eas.iis.fraunhofer.de

***Abstract*—FPGAs are a key enabling technology for rapid and efficient system prototyping and initial commissioning of newly developed integrated circuits. One major aspect is the setup and control of interface components between devices under test (DUT) and the FPGA infrastructure. So, as to maintain high flexibility in conjunction with the ability to deal with changes of requirements and use cases, as well as unforeseen or faulty behavior of the DUT, we propose a novel reconfigurable hardware/software infrastructure. IP blocks, such as register files or interface components to external hardware are attached as leafs to a tree-like communication system optimized for alterations. It is designed as an *Embedded Linux* compatible CPU subsystem to be accessed from user space via a uniform and portable kernel driver. Thus, it implements transparent access to custom functionality from user applications without specific knowledge concerning the hardware/software coupling.**

***Index Terms*—rapid system prototyping, hardware/software co-design, Embedded Linux, image sensor characterization, ASIC commissioning, FPGA**

I. Introduction

The design of complex, mixed-signal integrated circuits, such as image sensors, often requires the implementation and evaluation of prototypes of crucial subcomponents prior to system integration. All these blocks need to be tested and characterized regarding their electrical or optical properties [1]. The commissioning of prototypes is usually a very challenging task, because both the analog and digital interfaces have to be considered. With the use of programmable logic for the configuration, digital stimulation and readout of results from DUTs and peripherals, a high flexibility can be achieved. Standardized interfaces and custom IP for FPGAs help minimizing the effort necessary for system setup. In conjunction with image sensor characterization, it is important to test the DUT under real operating conditions. Integrating the imager into an experimental, stand-alone camera system based on the setup during commissioning is necessary. The control of application-specific hardware blocks and the description of the test scenarios is done in software, as far as practicable. Because of its availability for many soft- and hard-CPUs in FPGAs, *Embedded Linux* is a very popular operating system. For every setup, a tailored environment consisting of hard- and software parts is necessary. Because of potentially unpredictable behavior or malfunctions of the DUT, the behavior, the structure, as well as the hardware/software (HW/SW) interfaces are likely to change several times during commissioning. The decision to perform a given measurement task in software or, maybe also partially, in hardware is being revisited more than once. Consequently, this also has a significant impact on software development, since hardware and software parts have to fit together.

In order to simplify and speed up the whole process, a new hardware/software infrastructure is proposed. It is optimized for easy design alterations and the possibility to achieve them without the need for in-depth knowledge of FPGA-based hard- and software development. Basic components like register files and I/O blocks are provided as IP that can be used whenever accessibility by software is needed. A tree-based communication structure connected to the embedded CPU, and a general-purpose kernel driver for *Embedded Linux* [2] provides access from user space to attached devices. Asynchronous events generated by the hardware components can be passed to the application. Because of the frequent use of SystemC models for the DUT, the whole infrastructure has also been modeled using this system modeling language, allowing preparation of commissioning and the necessary development of test software before physical hardware is available. A user space library acts as abstraction layer and provides high level interfaces, which allow the test software to operate on the model the same way as on real hardware.

II. Related work

In modern Systems-on-Chip (SoC), the links for the interconnection of CPU, memory and peripherals play a central role. An overview of different buses used today is given in [3] and [4]. Most SoC bus systems, such as the AMBA family or CoreConnect, provide different options for high-speed data transfer on the one hand, and the link of simple components on the other. This is done by the possibility to choose an appropriate interface complexity, e.g. a simplified bus protocol or less signal lines. Each endpoint is assigned to an individual range in address space, allowing certain access operations [5]. The selection of components can be done by a central arbitration device (e.g. Processor Local Bus, PLB [6]) or by some kind of distributed resolution. In the latter case, hierarchically arranged arbiters can be used for splitting the address space into smaller slices and keeping only one

component. In both cases, adding a single device to the design could result in the need to re-partition the whole address space. To overcome this, it is possible to utilize a fully distributed resolution with one arbiter within each endpoint [7].

State of the art universal buses only define the physical connection and the protocol for data transfer. When accessing a component attached to a bus system from software, some kind of abstraction layer is needed. In Linux, a kernel driver [8], [9] is used, which hides the access to physical memory from the user. Writing such a driver is an extremely complex and time consuming task, requiring deep knowledge of kernel details. During commissioning of newly developed ICs with changed requirements or unforeseen or faulty behavior of the DUT, any relevant change made to the interface hardware has to be applied to the driver. Because driver development is unreasonable for a hardware engineer, direct low-level access from user space is desirable. More than that, recovering a user space application after a crash is much easier than a kernel driver, which, depending on the design of the operating system, may require a system reboot. One way to achieve this is the use of memory mapping (`mmap()`) on `/dev/mem`, which bypasses kernel security measures. Alternatively, the UIO Framework [10] performs a mapping of only selected components or address ranges into user space and allows the implementation of device drivers as regular applications. Each device is represented by a corresponding device file accessed by `mmap()`. UIO also supports the handling of asynchronous events by blocking a `read()` on the device file until an interrupt occurs. Changes made to the address layout of the underlying hardware are announced to UIO via its configuration.

The Reconfigurable System-on-Chip (RSoC) [11] framework for hardware accelerators utilizes DMA data transfer for the communication between user components and the systems memory. This is done by the *RSoC Bridge*, which links the FPGA accelerator to the rest of the system. Depending on the configuration of the bridge, the *RSoC Driver* is initialized automatically and can be configured from user space. For data exchange, the *RSoC Driver* provides one device file per accelerator. The RIFFA framework [12], [13] has been designed for FPGA-based accelerators placed inside a host PC and connected via PCIe. It provides basic IP blocks for linking proprietary components with each other and the host interface. All the components are connected using Processor Local Bus (PLB) [6]. There are different bindings for programming languages to ease the software development. SIRC [14] is another implementation of a platform for low level hardware access from application software running on Microsoft Windows. In contrast to the systems described above, in SIRC an Ethernet connection is used for that purpose. The MECA system [15] is a platform consisting of a custom board and a software environment based on *Embedded Linux*. An FPGA, located close to a dedicated single-chip computer on the same board, provides several digital I/Os. The purpose is to provide a platform for educational and hardware experiments. Components on FPGA can be accessed from user space applications. FrontPanel™ [16], another platform designed for fast prototyping, allows access to user defined HDL modules inside an FPGA, by a PC connected via USB or PCIe. It provides a graphical user interface (GUI) as well as an SDK for custom PC-based applications. One drawback of this solution is that the PC is essential for operation so it can only be used in laboratory setups, not as a stand-alone device.

Component	Description
`register`	Single N bit value available to ambient logic.
`reg_file`	Collection of 2^M HW registers with configurable widths.
`reg_connector`	Decoder and data composition for 2^M user defined registers with arbitrary width.
`bram`	Hardware-attached two-port block RAM with configurable width and depth .
`i2c_bridge`, `spi_bridge`	Bridge to external serial devices.
`info`	Special module for information on bus configuration at runtime. Also build data such as a time-stamp or SVN[1] revision number is made available.

Table I
OVERVIEW OF BASIC BLOCKS.

III. HARDWARE PLATFORM

One key component of digital and mixed-signal SoCs is the HW/SW-interface for the configuration and parametrization of integrated functional blocks with register files or memories to hold the setup data. From a software perspective, they represent memory areas, while for hardware access, a low-level interface is included. With the implemented scanning access changes made to the HW/SW interface are visible from software with almost no additional effort.

One important part of our interconnect infrastructure is a well-tested and reusable implementation of components for the encapsulation of the bus connection and the address space integration. It is a very effective building block to reduce the implementation effort for parts not being in the primary research focus. In table I a collection of basic blocks used as basis for software accessible modules is given. The interface to the communication infrastructure is built around the generic endpoint `base_slave` and always remains the same. In figure 1, the channels `to_slave` and `to_master` represent this link to the communication infrastructure. All variables within `<<...>>` are generics to be assigned during instantiation. Each slave possesses a unique identifier (COMP_ID), a type designator (COMP_TYPE), and a unique mapping to a range in address space, with the corresponding size determined by ADDR_WIDTH. Support for asynchronous events can be added for each component and is controlled by HAS_IRQ. The maximum count of addressable components as well as their maximum address width are globally defined for every design.

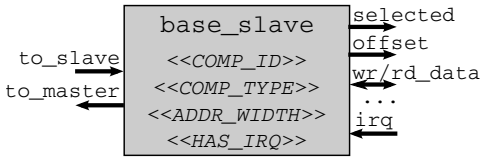

Figure 1. Generic `base_slave` used within all basic blocks.

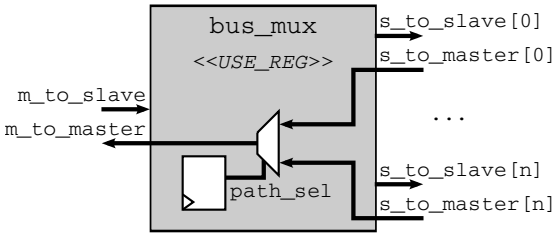

Figure 2. Simplified diagram of the bus multiplexer module `bus_mux`.

A typical RTL design consists of a hierarchy of modules [17] and anywhere within such a hierarchy, software accessibility may be needed. The proposed communication infrastructure follows a tree-based approach, spanned over the design. Every component around the basic slave module `base_slave` acts as a leaf within that structure, while a bus multiplexer module `bus_mux` (fig. 2) is used for the respective branch. The left side of this `bus_mux` can be connected the same way as regular slaves, allowing the extension of communication infrastructure where needed. On the right side, multiple ports are available. For the signal propagation from the root to the leafs, equivalent to a broadcast within the whole tree, the outputs `s_to_slave[...]` are forwarding information from the input port `m_to_slave`. The other direction is equipped with a multiplexer for the selection of a single path to the master.

To achieve a high flexibility in the extension of the structure at any point in hierarchy, the decoding of the unique identifiers is performed locally within leafs. During the selection phase,

[1]Apache Subversion (SVN), a revision control system mainly used for software development.

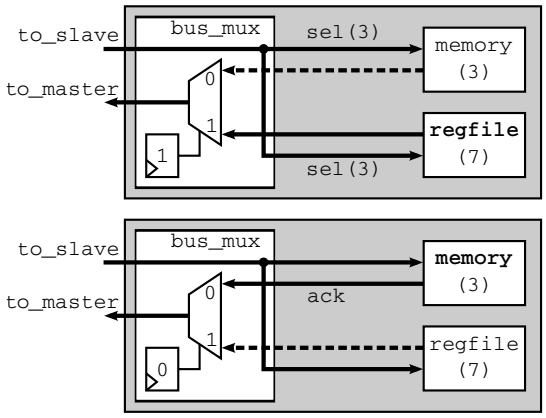

Figure 3. Demonstration of the selection phase.

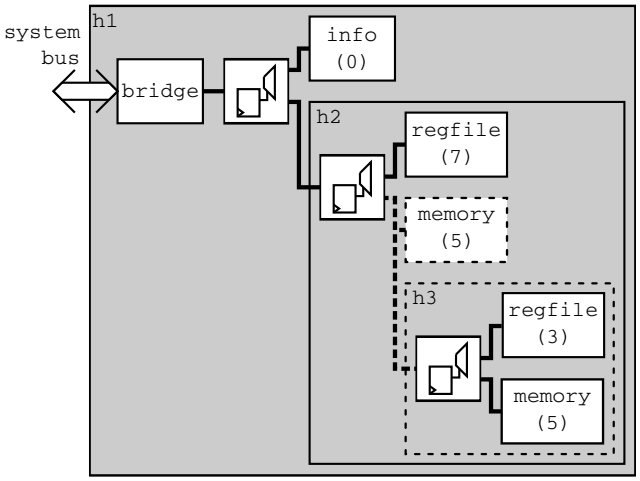

Figure 4. Example of a hierarchical design with spanned communication infrastructure.

the identifier is being forwarded until the requested slave is reached. An example for an endpoint indexed by identifier '3' is shown in the upper half of figure 3. On the path back to the root, every `bus_mux` is configured accordingly (lower half of figure 3). During the acknowledgement phase, the newly selected instance transmits its type designator and address width as payload. Both values are available within the `info` component which is also used for scanning the tree. Note that the access to the selected endpoint is exclusive and any changes need an additional selection phase.

In Figure 4 the example for a design with three layers of hierarchy denoted with `h1` to `h3` is given. In every layer, there are components accessible from the outside with instances of basic blocks as specified in table I and one instance of `bus_mux` at each level. A special `bridge` is used to tie the custom design below `h1` to the system bus of the SoC. It represents the master within the communication infrastructure and can be considered as the root of the tree. For each leaf, the corresponding identifier is written in brackets.

First assume, that instance `h3` is not contained in the design and the memory within `h2` is linked instead. However, due to changed requirements, it is decided to insert a third level of hierarchy to encapsulate and extend the memory by extra logic and some additional registers. Within the inserted level `h3`, another `bus_mux` instance is now prepended to the memory and a register file. There is neither a need to reassign any identifier, nor is there any change necessary within `bus_mux` configuration. The assignment of unique identifiers is done manually and the order of their distribution across the design is irrelevant.

Another very important feature of the proposed design infrastructure is the interrupt propagation setup. It has to stay in operation at any time, even independently on the selected slave. Because of the limited number of endpoints, it is possible to utilize a bit-vector with a unique source identifier. Each instance of `base_slave` assigns only the bit marked by its identifier and all the other bits are set to zero. Within

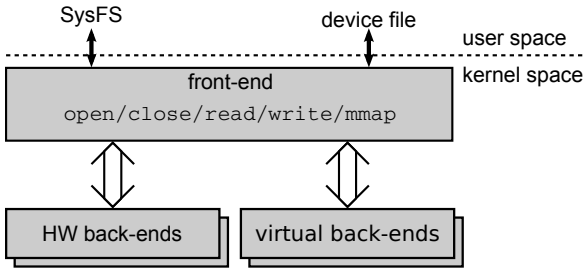

Figure 5. Decomposition of the driver in frontend and multiple backends.

`bus_mux`, the interrupt vectors `s_to_master[i].irq` are joined to `m_to_master.irq` using a bitwise OR-operation across all channels. This will effectively result in a direct connection from endpoint to the master after synthesis.

IV. SystemC Model

In order to support digital and mixed signal modeling of commissioning setups and to allow starting software development without real hardware, the interconnect infrastructure proposed in section III has been implemented in SystemC [18], [19], using Transaction Level Modeling (TLM) [20]. There is no need for a separate `bus_mux`, because the TLM channels are routed through the design hierarchy. During elaboration, a consistency check for the uniqueness of slave identifiers is performed. Instead of a bridge for linkage to the system bus, the communication is tunneled through a TCP/IP socket by a simple text-based protocol. The SystemC model acts as the server and opens a port for incoming connections. Any remote access arrives completely asynchronously and has no reference to simulation time. This is achieved using the `async_request_update()` mechanism introduced in SystemC 2.3 [21].

V. Software Architecture

The software part consists of a lightweight Linux kernel driver providing direct hardware access from user space and an object-oriented abstraction library used for application development.

A. Kernel driver

The kernel driver is inspired by the UIO framework [10], enhanced by multiple access to a collection of devices mapped into a continuous address space. For easier maintenance, the driver is split (fig. 5) into a frontend, which provides the interface to user space, and a backend for the communication with the hardware. This allows the definition of virtual backends, to be used for debugging and verification. Each backend provides information and access methods for all its components. A virtual backend mimics the existence of slaves by defining regions of appropriate size within main memory.

For the ease of change, it is important to allow automatic setup depending on the hardware configuration. Only very few additional information are necessary for this process. The address range is determined by a device tree [22] entry as

```
hwbus0: hwbus@60000000 {
  compatible = "eascbus";
  reg = < 0x60000000 0x4 >;
  interrupt-parent = <&axi_intc_0>;
  interrupts = < 4 4 >;
  skipscan = "no";
  identifier = "hwbus0";
};
```

Listing 1. Device tree for a single HW backend.

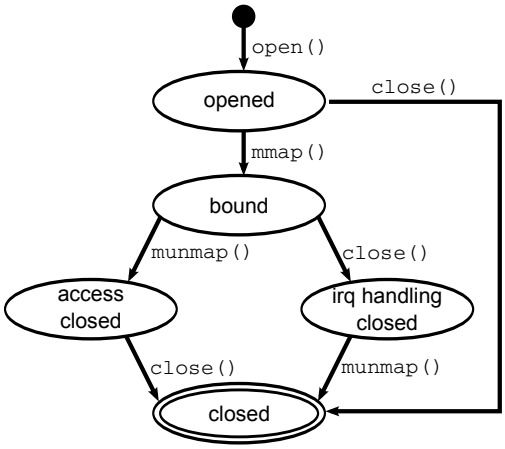

Figure 6. State transition diagram for application access to component.

depicted in listing 1, which is evaluated at driver probing phase. For each matching entry within device tree, one hardware backend is instantiated. It fetches the maximum count of slaves and the address range from the `info` component. During the following scan, all possible identifiers are examined by issuing appropriate reads. The corresponding result status, stored automatically within `info` component, supplies the driver with information about the endpoint for the given identifier. After the setup has been finished, all available slaves are registered in the backend.

For each backend, the frontend generates a corresponding device file in `/dev`, which is used for providing the common character interface [9]. In contrast to UIO, only one device file is created per master. Within SysFS [23], available information are supplied to user space. When the device file is being opened by an application, a corresponding driver instance will automatically be created by the kernel. However, this instance is not bound to any slave (fig. 6) and requires a `mmap()` call to establish the binding. To select a specific component, the offset used by `mmap()` points to the corresponding region in the address space of the SoC. Note that it is possible to open the device file by multiple simultaneous applications, but each individual endpoint can only be bound to a single process at once. Read and write is carried out using mapped memory access. For the closing of the connection to a slave, memory-mapping is terminated and the file-descriptor is deleted as depicted in figure 6. If an application terminates, this is done automatically by the operating system.

Asynchronous events generated by any endpoint are passed

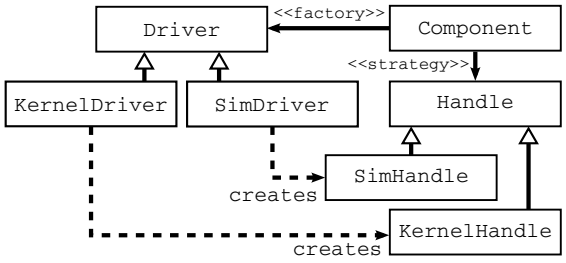

Figure 7. Simplified UML class diagram of user space library.

	Operation	Throughput accessing reg file in MiB/s	
		Spartan-6 @100 MHz (MicroBlaze)	Zynq-7020 @667 MHz (ARM)
1	single read	1.4	14.2
2	single write	1.5	16.9
3	block read	19.5	18.2
4	block write	25.8	23.8
5	ping pong read	1.3	12.1
6	ping pong write	1.4	15.7

Table II
RESULTS OF THROUGHPUT MEASUREMENT.

to the kernel via the SoC's interrupt mechanism. Each backend registers an appropriate interrupt handler. When an application calls `read()` on an already bound driver instance, the system call is blocked until the corresponding slave issues an interrupt. In contrast to UIO, the triggering of asynchronous events can be controlled for each device separately using the `write()` system call on a bound driver instance.

B. Library

To simplify access from custom applications, an object oriented library acts as abstraction layer. In figure 7, a simplified UML diagram is depicted. The library can be used in combination with a SystemC model using the infrastructure given in section IV. However, to access any component, a `Driver` object is needed. While the `KernelDriver` allows direct hardware access, the `SimDriver` sets up up a TCP/IP socket to a remote host running the SystemC model. This approach makes it possible to run the same application close to the hardware within an SoC or to control a SystemC model.

The `KernelDriver` object fetches all available information provided by SysFS. For each instantiated component, the device is opened separately. The *Abstract Factory* `Driver` is passed to any new instance of class `Component`, which allows the driver to create the appropriate *Strategy* of type `Handle`. The class `Component` provides different access methods for read and write, as well as the possibility to register a callback-function for asynchronous events. For each registered callback a separate thread is used, waiting for return from blocked read system call.

VI. IMPLEMENTATION AND APPLICATIONS

The proposed infrastructure has been implemented in VHDL [17] and is independent of the FPGA used. Currently, it is tied to the SoC's AXI-interface via corresponding `axi_bridge`. The data-width is set to 32 bit. For simplicity, burst-mode a transfer has not been realized yet. For timing improvement, registers can be placed within the path from slaves to the master. During elaboration, each instance of `base_slave` appends its hierarchical instance name followed by the identifier and the size of its memory area to a report file. This file is checked for duplicated identifiers. By default, the maximum number of slaves is limited to 32 and the size of the assigned address space to 16 kiB, but both values can be easily increased within the global configuration package.

The kernel driver has been implemented in C and supports one virtual as well as several hardware backends. The code is practically independent of the processor architecture used for the execution, and the access library for custom applications is implemented in C++.

The HW/SW design infrastructure has so far been used in several projects. As application examples, a camera system for remote machine monitoring [24] and a custom prototyping platform for tests and characterization of several prototypical and commercial image sensors [25] have been implemented. The framework itself was initially developed for a Xilinx Spartan-6 FPGA [26] with a MicroBlaze soft-core CPU [27] and peripherals necessary to run *Embedded Linux*. Components for acquisition control, memory management and I/O configuration, as well as the setup of the image sensor itself, utilize software access by the proposed communication system. Based on the library described in section V-B, the firmware to control the camera is a regular Linux application. With the development of SystemC models for the image sensor and the camera system, it was possible to develop the firmware without the need of final hardware. This allows the development of hard- and software in parallel.

Due to its system architecture, the proposed HW/SW infrastructure for ASIC commissioning and rapid system prototyping could easily be ported to Zynq [28], the next generation of Xilinx FPGA SoCs. The compilation of the kernel driver and the C++ based user space library for the embedded ARM CPU was sufficient to prepare the setup for the execution of existing applications.

To determine the total performance including hard- and software throughput, several measurements have been carried out. For example, a 16 KiB memory block was instantiated and connected to the communication infrastructure, as depicted in figure 4. The experimental setup consists of two FPGA systems, one based on Spartan-6 and the other one on Zynq-7020, with RTL running at 100 MHz. Table II illustrates three different access modes: single register access (rows 1 and 2), block access with 16 KiB at once (rows 3 and 4) and ping pong access (alternating accesses to two different endpoints, each implying an additional selection phase).

In the MicroBlaze soft-CPU, each access has a latency of 19 (read) or 15 (write) cycles, which results in a theoretical maximum throughput of 20.1 MiB/s for read and 25.4 MiB/s

for write access at 100 MHz. Due to synchronization issues, on Zynq there is one additional cycle necessary for both read and write, resulting in a maximum throughput of 19.1 MiB/s for read and 23.8 MiB/s for write respectively. The difference to the data given in the table is due to additional overhead from function calls and data movement in user space software. While the theoretical throughput is nearly reached for large blocks, the overhead has a large impact for single value access. Especially for the relatively slow MicroBlaze soft-CPU, this effect is dominating. Additionally, the peak interrupt rate handled by application software was measured as 2 kHz for the MicroBlaze soft-CPU and 15 kHz for the embedded ARM within Zynq device.

VII. Conclusion

The hardware of the proposed hardware/software infrastructure for ASIC commissioning and rapid system prototyping consists of several IP blocks, and can be used as a foundation for components to be accessed and controlled by software. Organized in a tree-based structure, extensions can be added as leaves without changes to the remaining hierarchy. For the hardware access from user space applications running on *Embedded Linux*, a newly developed kernel driver as well as a C++-library have been introduced. Implementation details are hidden from the user, which eases the development of Linux based firmware without detailed knowledge of kernel internals. Moreover, the kernel driver has been validated and can be used as is, even if the RTL of custom hardware is being modified for different test cases. The integration of SystemC models into the design process allows software development without the target hardware being available.

The implemented `axi_bridge` is fully functional, but the throughput for consecutive access can be increased by implementing a burst mode operation.

Currently a Python [29] binding is being developed to broaden the application options.

Acknowledgment

The authors would like to thank Lucas Stach for his great work in implementing the kernel driver.

The authors gratefully acknowledge the ENLIGHT project (http://www.enlight-project.eu/), funded by ENIAC and the German Federal Ministry of Education and Research, for the financial support.

References

[1] B. Blanco-Filgueira, P. López, J. Döge, and J. B. Roldán, "Evidence of the lateral collection significance in small CMOS photodiodes," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 2012.

[2] C. Hallinan, *Embedded Linux primer: a practical, real-world approach*. Pearson Education India, 2007.

[3] M. Mitić and M. Stojčev, "An overview of on-chip buses," *Facta universitatis-series: Electronics and Energetics*, vol. 19, no. 3, pp. 405–428, 2006.

[4] J. L. Ayala, *Communication Architectures for Systems-on-chip*, ser. Embedded Systems, J. L. Ayala, Ed. CRC Press, Inc., 2011, vol. 1. [Online]. Available: http://www.crcpress.com/product/isbn/9781439841709

[5] P. R. Schaumont, *A practical introduction to hardware/software codesign*. Springer, 2012.

[6] International Business Machines Corporation, *Processor Local Bus Architecture Specification Version 6 (PLB6)*. IBM, November 2012.

[7] J. Gaisler, S. Habic, and E. Catovic, "Grlib ip library user's manual," *Aeroflex Gaisler*, 2010.

[8] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*. O'Reilly Media, Inc., 2005.

[9] J. Quade and E.-K. Kunst, *Linux-Treiber Entwickeln: Eine systematische Einführung in die Gerätetreiber-und Kernelprogrammierung*. Dpunkt. verlag, 2011.

[10] H. J. Koch and M. S. Tsirkin. (2009, 7) The userspace I/O howto. Linux Kernel Documentation. [Online]. Available: http://www.kernel.org/doc/htmldocs/uio-howto

[11] J. Viktorin, P. Korcek, V. Kosar, and J. Korenek, "Framework for fast prototyping of applications running on reconfigurable system on chip," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, Oct 2013, pp. 347–348.

[12] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A reusable integration framework for FPGA accelerators," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 216–219.

[13] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–8.

[14] K. Eguro, "SIRC: An extensible reconfigurable computing communication API," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 135–138.

[15] S. Silverstein, J. Rosenqvist, and C. Bohm, "A simple linux-based platform for rapid prototyping of experimental control systems," in *Real Time Conference, 2005. 14th IEEE-NPSS*, June 2005, pp. 3 pp.–.

[16] Opal Kelly Inc., *FrontPanel – A new way to control and observe FPGA designs through virtual instruments on your PC.*, May 2014. [Online]. Available: http://assets00.opalkelly.com/library/FrontPanel-UM.pdf

[17] P. P. Chu, *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.

[18] T. G. S. Liao, G. Martin, S. Swan, and T. Grötker, *System design with SystemC*. Springer, 2002.

[19] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*, 2nd ed. Springer, 2010.

[20] F. Ghenassia *et al.*, *Transaction-level modeling with SystemC*. Springer, 2005.

[21] IEEE Standards Association and others, "IEEE standard for standard SystemC language reference manual," *IEEE Computer Society*, vol. IEEE1666-2011, 2012.

[22] G. Likely and J. Boyer, "A symphony of flavours: Using the device tree to describe embedded hardware," in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 27–37.

[23] P. Mochel, "The sysfs filesystem," in *Linux Symposium*, 2005, p. 313.

[24] J. Döge and E. Schäfer, "Fernmonitoring komplexer Fertigungsprozesse mit Hochgeschwindigkeits-Bildsensoren," in *MikroSystemTechnik Kongress 2013*, 2013, pp. 219–222.

[25] Aptina Imaging. (23.01.2013) 1/2-Inch Megapixel CMOS Digital Image Sensor MT9M001C12STM (Monochrome). [Online]. Available: http://www.aptina.com

[26] Xilinx Inc., "Spartan-6 FPGA family overview," 25.10.2011. [Online]. Available: DS160

[27] ——, "Microblaze processor reference guide," 24.04.2012. [Online]. Available: UG081

[28] ——, "Zynq-7000 all programmable SoC technical reference manual," 11.02.2014. [Online]. Available: UG585

[29] M. Pilgrim and S. Willison, *Dive Into Python 3*. Springer, 2009.