A HW/SW Co-design Tool for Modern FPGA's with FPGA-Embedded Processors

Authors: Jason Scott, Ted Bapty, Sandeep Neema, Brandon Eames
Vanderbilt University
Andrew Vandivort, Sarir Khamsi, Troy Gangwer
Raytheon

Abstract

Field Programmable Gate-Arrays (FPGA) containing one or more embedded processor cores within the FPGA fabric (both hard & soft processor cores) are quickly becoming a mainstream architecture for high-performance signal processing. Design tools for these new architectures from FPGA vendors have steadily matured to provide a viable, if laborious, solution for designers. These tools merge software compilers, hardware compilers, and basic IP to support a manual SOC design process. Given the complexity of potential target systems, there is a need to design at a higher level.

The current FPGA-plus-embedded processor core combination provides great opportunity for high-performance systems. The strengths of both the FPGA fabric to implement highly optimized hardware processing functions and a processor core to provide overall control and flexible processing tasks not suited for hardware implementation. In designing these systems, it is critical that bottlenecks in the control flow do not impact the time-critical data flow of the application.

This paper describes the ongoing work in developing a high-level co-design design environment for the latest FPGA SOC architectures.

Introduction

Field Programmable Gate-Array (FPGA) architectures that support embedded processor cores within the FPGA fabric are becoming a mainstream architecture for configurable-logic, high-performance signal processing systems. Design tools for FPGA + FPGA-embedded processor architectures have improved over the past few years to make this architecture a more attractive option for embedded systems engineers. These design tools include software compilers, FPGA place & route tools, simulation tools, debug facilities, and other miscellaneous tools needed to support the design and implementation process for a full system-on-a-chip (SOC) solution. These tools, however, are simply a merging of standard hardware and software development tools. With growing size and complexity of potential system designs available to today's large-transistor count hardware platforms, there is a need to design at a higher level of abstraction. Presented in this paper is a high-level co-design environment for seamless integration of hardware & software components, named Adaptive Computing Systems Model-Integrated Design Environment (ACS-MIDE).

The ACS-MIDE co-design environment uses a Model-Integrated approach to managing the complexity inherent in designing resource-constrained, high performance embedded systems. The Model-Integrated Computing (MIC) approach uses a multi-aspect modeling paradigm to capture the information necessary to synthesize a system[1][2][3]. This includes computation algorithms, requirements, and available (hardware) resources.

These design aspects are defined in a graphical language customized for the domain of embedded computing systems.

The ACS-MIDE co-design tool provides an environment where the algorithm specification is separate from the implementation target hardware platform specification, allowing portability across platforms. The hardware platform specification is an annotated block diagram of the physical processing elements of the system and their interconnections. The algorithm specification is data flow diagrams annotated with execution specifications, or constraints, such as end-to-end latency. The combination of data flow and constraints fully specifies the behavior of the target system.

Algorithmic elements (blocks) of this dataflow model can be implemented as software components or hardware components. Similarly, processing components can often be implemented various ways for a single implementation target, with various time/space tradeoffs. Given a set of alternatives, the tradeoffs are between the algorithm and system constraints (which we term a *Design Space*). The co-design tool supports modeling these alternate implementations so that all tradeoffs can be evaluated simultaneously.

Once a design configuration is identified, the tool synthesizes the configurable hardware designs (top-level VHDL), real-time processing schedules for processors, interface code, and other configuration files. These are compiled with an dataflow kernel and support libraries to produce a full deployable design (FPGA configuration files and software executable code). The underlying mechanisms used to implement the design are critical in developing an efficient system. An efficient, small footprint kernel supports dataflow execution. Efficient, hardware supported, interfaces support low overhead, high bandwidth communication between hardware and software processing components.

In the past, we have used the ACS-MIDE tool for development of systems platforms based primarily on discrete DSP and FPGA devices. We have recently extended the ACS-MIDE toolset to support FPGAs with integrated processors as an implementation target. With the FPGA-embedded processor, communication between FPGA-implemented hardware components and software tasks executing on the embedded processor takes place via a microprocessor bus. The Xilinx Virtex-II Pro FPGA, for example, has tool and IP support for IBM's CoreConnect bus architecture. The CoreConnect bus architecture is well suited for efficient data transfer between on-chip hardware components and software tasks. We use the CoreConnect bus architecture to implement an efficient interface between software tasks executing on the embedded processor and hardware tasks executing within the FPGA fabric. This paper describes our initial prototype tool supporting FPGA/FPGA-embedded processor(s) SOC architectures.

Design Representation

The ACS-MIDE co-design tool uses a model-integrated approach, where the algorithm description (model) is decoupled from the hardware resource description (model). In this section, we describe the modeling environment. The section is divided into the following major categories: Algorithm Models, Resource Models, and Constraint Specification.

Algorithm Model

The algorithm model aspect describes the processing algorithm structure. Algorithms are specified in a data-flow, block diagram (a specification very familiar to DSP engineers). To manage system complexity, hierarchy is used to structure algorithm definition. This logical composition of systems using component subsystems has proven effective design structuring for very large, complex systems. The algorithm is modeled as a dataflow structure with the following classes of objects: *compounds*, *primitives*, and *alternatives*. A *primitive* is a basic element representing the lowest level of processing. A primitive maps directly to a processing task that will be implemented as either a hardware function or a software function. Primitive objects are annotated with attributes. These attributes capture measured performance, resource (memory, FPGA logic elements) requirements, and other user-defined properties.

A *compound* is an aggregation object that may contain primitives, other compounds, and/or alternatives. These components are connected within the compound to define the information dataflow. Compounds provide the hierarchy in the structural description that is necessary for managing the complexity of large designs.

A design *alternative* object allows the specification of multiple design alternatives for a given task. These design alternatives can be either compounds or primitives, allowing hierarchies of design alternatives. With alternatives, the algorithm models can describe a huge number of potential design implementations. The set of potential designs is termed a “Design Space”. Modeling a design space yields greater opportunity for design optimization and portability across changing requirements and hardware platforms.

In signal processing, tasks can often be accomplished in multiple ways (e.g. in the spatial or the spectral domain). Both approaches can achieve the same basic results but with vastly different algorithm designs. Other algorithm characteristics can vary as well, such as latency and/or accuracy. In the spatial domain a filtering function can be achieved by performing a standard mathematical convolution. In the frequency domain, the function uses the FFT, followed by a multiplication with the spectral representation of the filter, followed by an inverse FFT (see Figure 1). The spectral method is more efficient as the filter order increases, resulting in a faster, smaller system. On the other hand, since the FFT is a block-based computation, the latency is at least a block-length.

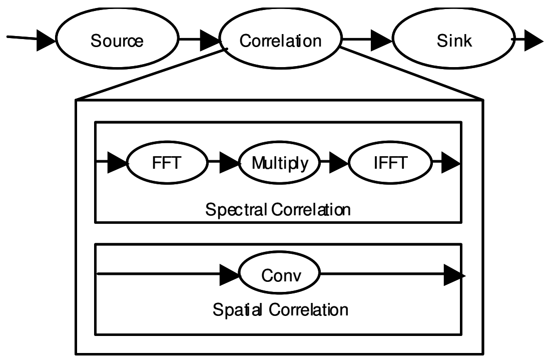

Figure 1: Algorithm Alternative – Specifies Design Flexibility

Each of these alternative methods has different performance attributes and different hardware requirements. The selection of the best alternative depends not only on the hardware that is available, but also on whether the hardware is to be time-shared, and what hardware has been allocated to support the processing algorithms that are required for operations in different modes.

For the high-level designer, algorithm alternatives allow a virtual separation of algorithm from implementation. Typical algorithm design requires the engineer/physicist to consider the hardware details of the underlying architecture to achieve an efficient implementation. The ultimate effect is that the resulting algorithm reflects the hardware structure. This practice leads to highly non-portable, technology-specific designs. System upgrades to use more current technology require a bottom-to-top redesign. Algorithm alternatives promise to separate the algorithm from the architecture, to postpone the implementation decisions to a much later step in the design process. This approach should greatly simplify technology migration efforts. The selection of the desired implementation technology is determined in the synthesis process, driven by power consumption, throughput, latency, specific part availability, and other architectural interactions.

Resource Model

The resource model aspect defines the target hardware platform. The target hardware platform is modeled in terms of hardware processing components and the connections among them. The hardware processing components may be of type: processor elements (such as DSPs or general-purpose RISC/CISC processors), programmable logic components (such as FPGAs), or dedicated hardware ASIC components for fixed functions (such as FFT computation). Data Sources and Data Sinks capture the specifics of hardware I/O. The resource model defines processing elements of the systems and the physical interconnection between these components. The connections capture the "as-built" topology of the physical implementation.

Constraint Specification

System constraint specifications have four categories of design constraints: (a) composability constraints, (b) resource constraints, and (c) performance constraints to establishing linkages between properties in different modeling categories. Composability constraints are logic expressions that restrict the composition of alternative processing blocks (e.g. FFT-HW can be used with IFFT-HW). Resource constraints are logic expressions describing the selection of processing blocks based on resource limitations. Performance constraints are integer constraint expressions limiting the end-to-end latency, power and space. These constraints allow the designer to control the potential design space for the analysis/synthesis process.

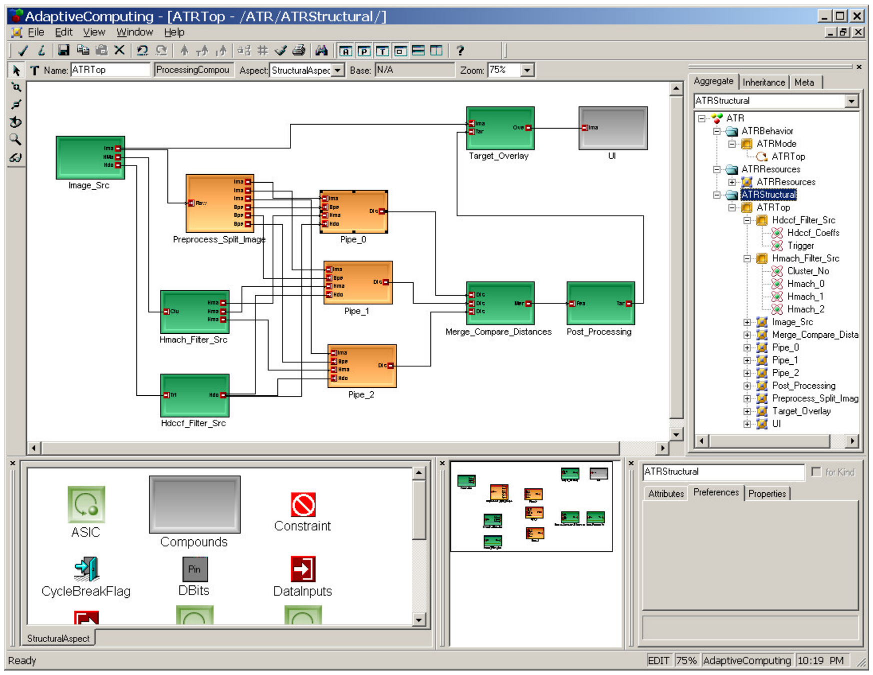

Figure 2: Algorithm Dataflow Model (GME Screenshot)

Design Space Exploration

The end product of the design modeling process described above is a *design space* consisting of requirements, potential implementations, and resource sets. The task of the designer is to select the appropriate implementation and resource assignments for the design. For example, an FPGA implementation of an FIR filter can be created as a set of multipliers operating in a parallel-pipelined fashion for maximum throughput (with maximum resource usage). An alternate implementation, minimizing resource usage, can be implemented with a signal multiplier that is time-multiplexed to produce its output. The best implementation depends on system constraints: real-time processing constraints and hardware platform constraints such as size, weight, power constraints.

Given the flexibility in defining design alternatives, this space can be extremely large (moderately sized design examples have defined a space of 1024^{th}). It is unreasonable to assume that a designer can handle such a large design space without sufficient tools. The set of design solutions must be evaluated to a design (or *configuration*) that best satisfies a number of design criteria. There are inherently a large number of conflicting design criteria in highly-constrained embedded systems. The analysis tools must allow efficient exploration, navigation, and pruning of this space to select feasible hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility. Given the size of the design space, and the complexity of the analysis, a powerful analytical method is required. The tool finds the set of viable designs (not necessarily optimal, rather a set that satisfies constraints). Typical design spaces range from 1000's to 10^{20} design options. The co-design tool permits simultaneous evaluation of all options.

The approach we have taken is to use symbolic methods based on Ordered Binary Decision Diagrams to represent, navigate and prune the design space. In a symbolic representation, sets/spaces are represented as a Boolean expression over the members of the set. The members of the set are encoded as binary variables under a binary encoding scheme. The principal benefit of the approach is that it does not require enumeration of the set/space to perform operations. Ordered Binary Decision Diagrams [4][5] are a canonical representation of logic functions, representing Boolean functions as directed acyclic graph in a memory-efficient format. The operations over the Boolean functions are implemented as graph algorithms rendering "manipulation" of the space fast and efficiently.

With this symbolic formalism, the application of logical constraints is relatively straightforward. The user-defined logical constraints can be represented as a Boolean expression over the components of the design space. Constraint application is then just conjunction of the constraint Boolean expression with the Boolean expression that represents the design space. The resultant Boolean expression represents the "constrained" design space. Application of the integer arithmetic constraints such as timing and power constraints is not so straightforward (see [6] for details). However the basic approach remains the same.

The constraints "prune" the design space due to the requirements specified in the constraint. These constraints can be iteratively applied to the design space, with the goal of reducing the "1024^{th}" to a more manageable set containing 10's to 100's of design alternatives. We have implemented the approach described above in a design space management tool that allows solving these constraints in an iterative manner. The design engineers can apply the constraints and visualize the sensitivity of the design space to the constraint. If the constraint is extremely tight it can be released and other constraints can be applied instead. Finally when the design engineer is satisfied with the remaining design choices after constraining the design space he can move to the next step of design synthesis/simulation.

<u>Execution Environment</u>

The execution environment must support implementation platforms with the following attributes:

• Heterogeneity: Optimizing the architecture for performance, size, and power requires that the most appropriate implementation techniques be used. Implementations may require software (implemented on RISC, microcontroller, and DSP processors), configurable hardware in FPGA fabric, and/or a mix of ASIC components.

- Low Overhead/High Performance: the execution environment must minimize overhead, since overhead results in extra hardware requirements.
- Hard Real-Time: The target systems have significant real-time constraints.

This list points out some of the design complexities. Working alone, the execution environment cannot solve these problems. The overall system design approach must span from the top-level algorithm system requirement & resource specifications down to the hardware/software implementations. The Execution Environment forms the infrastructure onto which these designs are projected. The Execution Environment must be designed with an interface suitable for synthesis from a MIC-Generator approach. The concepts, properties and interfaces of the execution environment must be compatible with the design representation and synthesis approach. Capabilities and interfaces should be tuned to simplify the generator. This requirement demands a simple, uniform interface with a well-defined, consistent set of semantics that apply throughout the system. Since the system includes software, hardware, and interactions between parallel modules, a common structure must map to a wide range of components.

The semantics of the execution environment implements a large-grain dataflow architecture. At this level, there is no implied implementation of the dataflow tasks. A dataflow task executes when his specified input data is available and, upon completion of execution, produces output data. The execution semantics of the dataflow tasks are maintained at a higher level. The semantics of the communication links between the dataflow tasks are asynchronous queues. When the generic large-grain dataflow graphs are implemented, they must be mapped down to a physical implementation. The implementation takes the form of either software or hardware. Software tasks execute on a DSP or general-purpose processor. Hardware tasks are implemented in configurable hardware (FPGAs), ASIC implementations, or combinations of both. Both task implementations logically equivalent, representing processing functions on data.

The Execution Environment spans software and hardware. The software environment consists of a simple, portable real-time kernel with a run-time configurable process schedules, communication schedule, and memory management [7]. Communications interfaces are supported within the kernel, making cross-processor connections invisible. Memory management is integrated with the communication subsystem, providing low-overhead messaging between processing tasks.

The hardware Execution Environment supports the same operational semantics. The implementation, however, is much different. The Virtual Hardware Kernel exists as a concept used in the system synthesis. The ACS-MIDE tool synthesizes a set of top-level VHDL descriptions for each FPGA component in the target platform. Hardware tasks are specified as direct implementations from the component library. Communications interfaces are selected from a library of interface types, based on the requirements of the tasks on either end. The communication infrastructure works in cooperation with the software communications, performing the signal buffering, and the necessary off-chip interfaces and data converters to present data in the format required by the destination processing element. As the system is expanded to support a wider-variety of target hardware, the set of interface types will grow in capability.

In addition, the execution environment is designed with an interface suitable for synthesis from a MIC-Generator approach. The properties of the execution environment are tuned to simplify the generator. This demands a simple, uniform interface with a well-defined, consistent set of semantics that apply throughout the system.

Virtex II-Pro Extensions to Execution Environment

As described above, for FPGA implementations the ACS-MIDE system generation automatically selects the best-fit interface component to interface FPGA processing tasks to the outside world (from the FPGA's perspective). Connections are made between hardware tasks, inserting proper conversion components as necessary.

The latest FPGA + FPGA-embedded processor core architecture provides a unique platform for software to work closely (high bandwidth connection) with programmable hardware. Our initial experiments for porting our design tools over to these new platforms have been with Xilinx's Virtex-II Pro FPGAs. The Virtex-II Pro platform is unique in that the FPGAs contain one or more (up to four) IBM PowerPC 405 "hard cores" embedded within the FPGA fabric. Xilinx has an extensive toolset (Embedded Development Kit – EDK) for hardware/software co-development on this platform. The EDK toolset can manage and synthesize a complete microprocessor/programmable hardware system on a FPGA.

The Virtex-II Pro PowerPC system infrastructure is based on IBM's CoreConnect bus technology [8]. The CoreConnect architecture implemented is a two-tier structure composed of IBM's CoreConnect Processor Local Bus (PLB) and On-chip Peripheral Bus (OPB). The PLB bus is the higher performance of the two and is directly connected to the PowerPC 405 core(s). It is used for interfacing to high-performance system devices such as FPGA internal RAM, external SRAM, DDR SDRAM, etc. The OPB bus, on the other hand, is not intended to connect directly to the processor, but rather function independently at a separate level of hierarchy from the PLB bus. A bridge unit (PLB-to-OPB) makes the OPB bus accessible from bus master components on the PLB bus (typically the processor). It is also possible for OPB bus master components to access components on the PLB bus via an OPB-to-PLB bridge component. More details on the Xilinx/CoreConnect architecture can be found in [9].

To support this new architecture, we need to interface our ACS hardware task components (in the FPGA fabric) to the software task components (running on the PowerPC(s)) in an efficient manner. It is important the processor (PowerPC) have fast (DMA/interrupt driven) access to the hardware components. Also, it is important that the processor have fast, low-latency access to its various memory banks via its local PLB bus. For these reasons, interfacing via the OPB bus was the logical choice for the Virtex-

II PRO architecture. An OPB slave peripheral was created to provide an interface between the OPB bus and the ACS hardware task protocol.

The typical ACS-MIDE hardware task interface, for example, implements a unidirectional data flow via a simple, low-overhead handshaking protocol supporting a data path with a parameterized width. The OPB_ACS_IO interface peripheral provides a link from the OPB bus to hardware task via the various hardware task interfaces supported by ACS-MIDE. This architecture is shown in Figure 3. The OPB_ACS_IO peripheral uses the IPIF interface layer described in [10] to provide easy access to the OPB bus signals. The peripheral exposes a bus-interface where unidirectional hardware interface components (that meet an existing ACS-MIDE supported interface standard) can attach (ACS_IN, ACS_OUT as shown in the figure). At this point, we now have connectivity from the OPB bus to our hardware tasks. This includes processing tasks, communication tasks such as additional buffering, as well as custom I/O device support tasks.

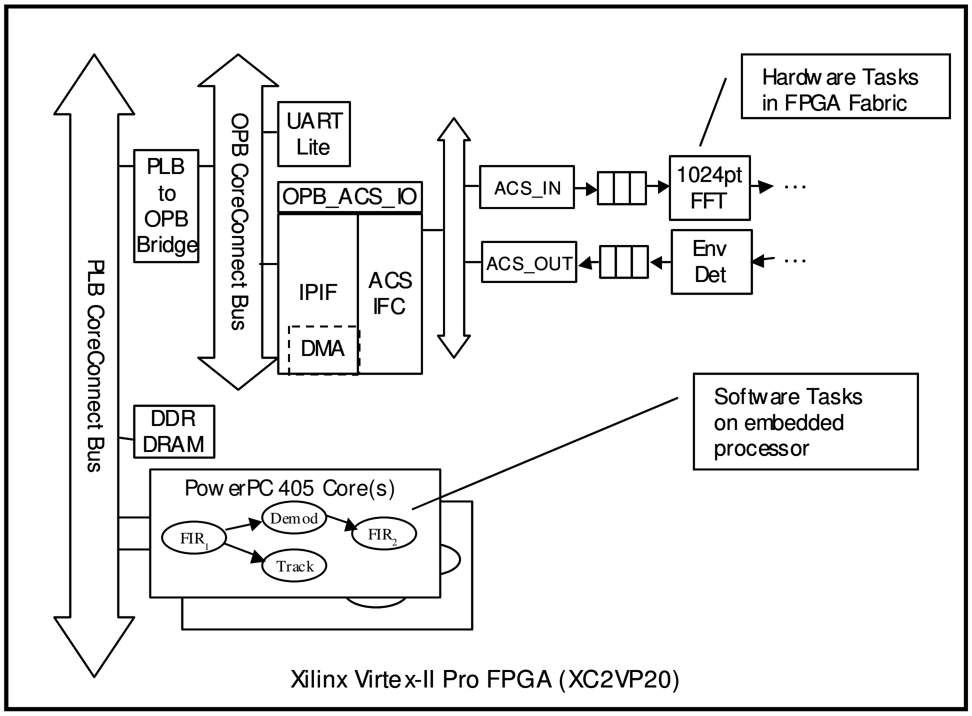

Figure 3: OPB interface peripheral

Currently under development is the inclusion of a DMA engine to handle data transfer to/from the OPB_ACS_IO peripheral. The use of DMA is essential not only to the implementation of high-throughput communication across the hardware/software boundary, but low-overhead communication on the software side so as few processor cycles as possible are expended on communication overhead.

Note that although we have not implemented a MicroBlaze design, our tool should be able to extend this platform with little or no changes due to the use of Xilinx's implementation of the CoreConnect bus architecture.

Conclusions

Real-time embedded system development and implementation are complex tasks. This is especially true for systems constructed using the latest generation of very powerful and flexible FPGAs that contain embedded processor cores. These systems are likely multi-processor, multi-FPGA systems interacting with several input/output devices and executing hundreds of hardware & software tasks. Design automation tools are necessary for automation of tedious and error-prone tasks such as managing the task allocation and communication of such a complex system. ACS-MIDE co-design tool has been developed to deal with these challenges.

In an ongoing work, the ACS-MIDE co-design environment has been extended to support the latest FPGA SOC architectures, specifically Xilinx's Virtex-II Pro architecture. We are currently in the process of improving the performance of the various interface components recently developed to support this architecture. The performance of these interfaces is critical to achieving the difficult goal of synthesizing a system with performance comparable to a hand-coded design. We are also considering support for preemptive kernels such as VxWorks on the embedded PowerPC core.

References

[1] Karsai G., Sztipanovits J., Ledeczi A., Bapty T.: Model-Integrated Development of Embedded Software, Proceedings of the IEEE, Vol. 91, Number 1, pp. 145-164, January, 2003.
[2] Evans D., Morris D., "Applying Modeling to Embedded Computer Systems Design", in "Codesign – Computer-Aided Software/Hardware Engineering", edited by: Rozenblit J. and Buchenrieder K., IEEE Press, 1994.
[3] Abbott B., Bapty T., Biegl C., Karsai G., Sztipanovits J.: Model-Based Approach for Software Synthesis, IEEE Software, pp. 42-53, May, 1993.
[4] Bryant, R.E., "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, June 1992
[5] Bryant, R.E., "Graph-based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, C35(8), 1986
[6] Neema S., Sztipanovits J., Karsai G., Butts, K.: Constraint-Based Design Space Exploration and Model Synthesis, Lecture Notes in Computer Science, 2003.
[7] Bapty T., Abbott B.: "Portable Kernel for High-Level Synthesis of Complex DSPSystems", Proceedings of the the International Conference on Signal Processing Applications and Technology, Boston, MA, May, 1995
[8] IBM, "The CoreConnect ^{TM} Bus Architecture", available at: http://www-306.ibm.com/chips/products/coreconnect/.
[9] Xilinx, "CoreConnect – A Simplistic Overview," 2003, available at: http://www.xilinx.com/esp/optical/collateral/CoreConnect.pdf.
[10] Xilinx, OPB IPIF Architecture, DS414, available at: http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb_ipif.pdf.