# Partially Reconfigurable Cores for Xilinx Virtex

Matthias Dyer, Christian Plessl, and Marco Platzner

Computer Engineering and Networks Lab,
Swiss Federal Institute of Technology, ETH Zurich, Switzerland
(dyer|plessl|platzner)@tik.ee.ethz.ch

**Abstract.** Recent generations of high-density and high-speed FPGAs provide a sufficient capacity for implementing complete configurable systems on a chip (CSoCs). *Hybrid CPUs* that combine standard CPU cores with reconfigurable coprocessors are an important subclass of CSoCs. With *partially reconfigurable* FPGAs, coprocessors can be loaded on demand while the CPU remains running. However, the lack of high-level design tools for partial reconfiguration makes practical implementations a challenging task.
In this paper, we introduce a design flow to implement hybrid processors on Xilinx Virtex. The design flow is based on two techniques, *virtual sockets* and *feed-through components*, and can efficiently generate partial configurations from industry-quality cores. We discuss the design flow and present a fully operational audio streaming prototype to demonstrate its feasibility.

## 1 Introduction

In the last years, many custom computing machines have been presented that achieve high performance by coupling general-purpose CPUs with field-programmable logic. These computers map the runtime-intensive parts of algorithms to application-specific coprocessors implemented in reconfigurable hardware. Many of the early custom computing machines attached one or several FPGAs to a processor via the memory bus or a coprocessor interface. In such a coupling, the processor architecture largely dictates the interface between CPU and coprocessor which often results in speedups that are far from the optimum. Usually, these interfaces lack bandwidth and flexibility, i.e., they cannot be adapted to match the coprocessor's data and control flows.

*Hybrid CPUs* integrate CPU cores and reconfigurable logic in form of a Configurable System on a Chip (CSoC) and thus potentially overcome the interface limitations. Hybrid CPUs come in two flavors: hybrids using *hard* CPU cores and hybrids based on *soft* CPU cores. Hard CPU cores, e.g., PowerPC core in Virtex-II Pro [15] or ARM7 core in Triscend A7[13], are built-in dedicated CPUs, whereas soft CPU cores, e.g., LEON [5] or OpenRISC 1200 [10] are fully synthesizable blocks of intellectual property (IP). Soft CPU cores are of particular interest for building custom computing machines as they not only allow the modification of the interface between CPU and coprocessors, but also facilitate adaptions to the CPU core itself. Soft cores enable the rapid exploration of different couplings between CPU and coprocessors. The interface can be tailored specifically to the applications need, ranging from memory mapped interfaces over FIFOs to more sophisticated structures such as memory prefetch buffers and DMA controllers.

Published in: *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*
 DOI: 10.1007/3-540-46117-5

Recent generations of high-density and high-speed FPGAs, e.g., the Xilinx Virtex series, provide a sufficient capacity for implementing or prototyping complete configurable systems on a chip. Virtex devices are *partially* reconfigurable, i.e., the reconfiguration affects only a part of the FPGA while other parts remains working. Although many authors proposed and described partially reconfigurable systems, the reduction to practice for Virtex devices lags behind. The major hurdle is the lack of high-level design tools that support partial reconfiguration. The Xilinx Java class-library JBits [6] is a prominent example for a design tool at a somewhat lower level. JBits allows to create circuits basically by structural description and to manipulate and generate full and partial configurations. Currently, JBits does not offer synthesis, optimization, or timing analysis capabilities. This restricts the usability of JBits for designing complex hybrid processors. Developers of high-quality cores will resort to hardware description languages and proven synthesis and FPGA design implementation tools.

To facilitate the development of partially reconfigurable coprocessors, we introduce a new design flow that combines industry-quality design tools with JBits' ability of creating partial configurations. The main contributions of this work are:

- a design flow to implement hybrid processors on Virtex FPGAs
- a technique to create partial configurations from optimized cores
- an audio streaming prototype to prove the feasibility of the design flow

Section 2 of this paper reviews different design flows and related work for generating partial reconfigurable cores. Section 3 presents detail of our design flow; Section 4 discusses a hybrid CPU prototype implementation. Finally, Section 5 concludes the paper.

## 2 Design Flows for Partial Virtex Configurations

A Virtex configuration bitstream contains an integer number of so-called *frames*. Frames are the basic units of reconfiguration and determine the settings of all FPGA resources in the vertical dimension. If the reconfigured frame contains only stateless elements, i.e., no flip-flops or LUTs used as RAM, there will be no transient effects. Elements with state, however, are overwritten with their initial values. A procedure to avoid this is to stop the clock and read back the current state values prior to reconfiguration. The retreived state data must be included in the reconfiguration bitstream. After reconfiguration, the clock is enabled again. Although Xilinx Virtex FPGAs support partial reconfiguration, an integrated design flow that allows to develop complex partially reconfigurable systems is missing so far. Currently, developers have the choice between two basic design tool flows: direct bitstream manipulation based on standard FPGA synthesis and design implementation tools, and bitstream generation and manipulation with JBits.

### 2.1 Direct Bitstream Manipulation

Standard design implementation tools for FPGAs generate full configuration bitstreams. The structure of Virtex bitstreams is partly open to the public, which allows to directly manipulate such bitstreams manually or based on scripts and programs.

A rather simple manipulation is to change the contents of LUTs and BlockRAMs [14] in a bitstream. Such a technique has been used, for example, to customize logic functions at download time for instance-specific SAT solvers [11]. By extracting the relevant frames from a full configuration, partial bitstreams are generated. In combination with LUT modifications, runtime customization of FPGA cores becomes feasible. This technique could be used, for example, to dynamically change coefficients of a digital filter. In principle, this technique can also be used to generate partial configurations for reconfigurable coprocessors. Figure 1 shows two full configurations, each containing a static core and a dynamic core. The set of frames containing the dynamic core can be extracted directly from the full bitstream to form a partial configuration. At runtime, the partial configuration is loaded on demand.

The main advantage of direct bitstream manipulation is that it bases on full configuration bitstreams that can come from arbitrary standard FPGA synthesis and design implementation tools. These tools are laid out to optimize circuit qualities, such as speed and area. Further, when the partial configurations modify only the contents of LUTs and BlockRAMs, the direct bitstream manipulation is quite simple and efficiently implemented.

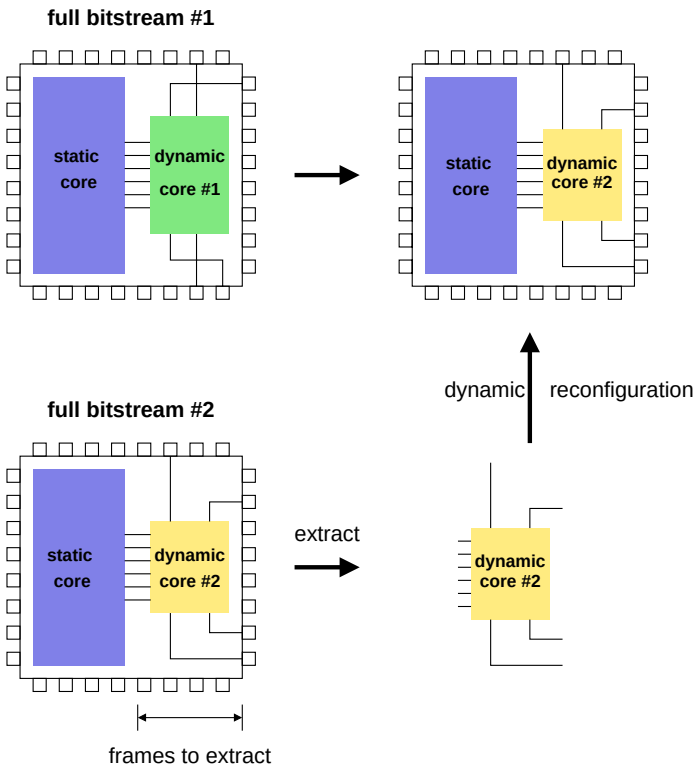


**Fig. 1.** Extracting a partial bitstream from a full bitstream.

Direct bitstream manipulation shows two limitations. First, for more complex designs the low-level manipulation of bits in a monolithic bitstream becomes extremely tedious. Second, as the routing cannot be changed, different coprocessor cores must occupy exactly the same subarea of the FPGA, and the interface between the CPU and coprocessors must be bound to a fixed location. Moreover, it must be ensured that the routes for the CPU core do not run through the partially reconfigured area and vice versa. Because current design implementation tools do not allow to pose location constraints on routing resources, constraining the routing becomes more or less a trial-and-error process, involving manual intervention (see Section 3). Nonetheless, direct bitstream manipulation has been used for generating partial reconfigurations in [8] [7]. A comprehensive study of the technique can be found in [7].

### 2.2 Bitstream Generation and Manipulation with JBits

The Xilinx JBits SDK [6] provides access to most of the Virtex resources through a Java class library. JBits starts operating on an initial configuration. All Virtex resources can be instantiated and configured. At any time, JBits can save changes to the design as partial bitstreams. Up to now, JBits supports structural circuit design only, but it enables hierarchical designs by grouping subcircuits into modules or cores. JBits also includes an automatic router, which can dynamically route and unroute connections. In the design flow described in [12], JBits manipulates designs, given as EDIF netlists, that have been fully mapped and placed by synthesis tools. JBits adds the routing and generates configuration bitstreams. JBits development is still in flux. For example, currently the abilities of the autorouter are still limited and some Virtex resources cannot be manipulated, e.g., long lines.

JBits offers two advantages over direct bitstream manipulation. First, JBits introduces a higher level of abstraction as it operates on CLBs, routes, etc., rather than on raw bits in a bitstream. This feature also opens up future dynamic applications, where tasks can be relocated and connections can be re-routed online. Second, JBits includes versatile functions for full and partial bitstream manipulation.

The limitation of JBits stems from the fact that it does not yet provide support for combinatorial and sequential synthesis, timing-driven placement and advanced routing. Currently, a pure JBits design flow seems only applicable for smaller or data flow oriented applications. Developers of more complex designs, such a coprocessors, prefer synthesis from RTL specifications in hardware description languages and corresponding optimization tools.

### 2.3 Combining High-Quality Synthesis with JBits

To get the best features from both worlds, standard design flows and JBits can be combined. We experimented with two different ways to combine complex, high-quality cores with JBits. Both design flows synthesize the static and dynamic cores with standard synthesis and design implementation tools, and generate the partial configuration bitstreams with JBits. The first approach uses JBits to merge the cores but does not dynamically re-route the connections. The locations for the reconfigurable cores and the interface between static and reconfigured cores are bound to fixed locations. The second approach places and routes the cores dynamically. In this paper, we focus on the first approach. The details of this technique are described in the next section.

## 3 Merging Cores with JBits

This section presents our design tool flow that allows to generate an initial full configuration and a number of subsequent partial configurations. We discuss the details of the tool flow on the example of a soft CPU core plus one coprocessor as initial configuration, and coprocessor cores as partial configurations. The tool flow relies on two techniques: the *virtual socket*, a fixed-location interface between the CPU and the coprocessor cores, and *feed-through components* to constrain routing.

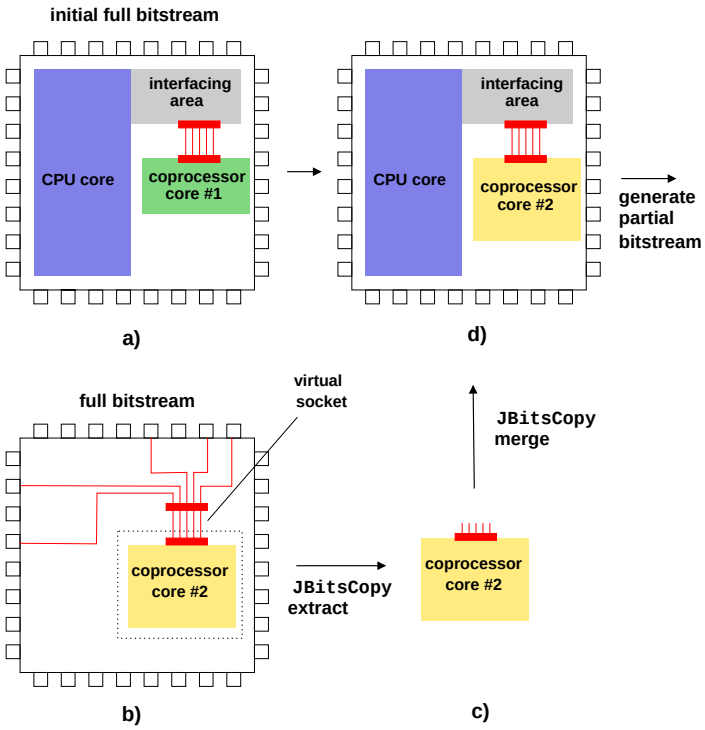


**Fig. 2.** Generation of initial full and partial bitstreams.

### 3.1 Overall Tool Flow

The overall FPGA area is divided into two non-overlapping parts, one part for the CPU core and the other one for the reconfigurable coprocessors. The generation of the initial full configuration involves following steps (shown in Figure 2a)):

1. Create an initial design which consists of the CPU core and a coprocessor component. Any synthesis tool or core generator can be used to derive high-quality designs.
2. Insert the virtual socket, a predefined interface component. Connect the CPU and coprocessor via the virtual socket.

3. Run FPGA back-end tools with constraints on the locations of the CPU, the coprocessor, and the virtual socket. This generates the initial full bitstream.

The tool flow respects the characteristics of the Virtex frame-based configuration mechanism. The part of the CPU core that is located in the same (vertical) frames as the coprocessor area, denoted as *interfacing area* in Figure 2, contains only stateless resources. A partial bitstream for a coprocessor is generated as follows:

1. Create the coprocessor design with any front-end tool.
2. Connect the coprocessor to the predefined virtual socket. The CPU-bound signals of the virtual socket are connected to unused I/O pins to prevent the optimizer from removing the socket component.
3. Run FPGA back-end tools with constraints on the locations of the coprocessor and the virtual socket. This generates a full bitstream (shown in Figure 2b).
4. Use the tool *JBitsCopy* to i) extract the coprocessor from the full bitstream (shown in Figure 2c)), and ii) merge the coprocessor with the initial full bitstream (shown in Figure 2d)). By this, the initial coprocessor area is overwritten with the new coprocessor. The new coprocessor fits seamless into the initial design, provided the location constraints for the coprocessors and the virtual socket have been respected. JBits is then used to generate a partial configuration bitstream that reflects the coprocessor area.

### 3.2 Virtual Socket

The virtual socket is a component that provides fixed locations for a set of pre-defined signals. All signals from the CPU or the I/O pins to the coprocessor and vice versa are routed through this interface. The only exceptions are the global nets for clock and set/reset signals. Because the interface is static, new coprocessors can be developed without having access to the CPU design.

### 3.3 Constraining the Design

The Xilinx back-end tools allow to put placement constraints on a design's logic resources, but not on the routing. On the other hand we have to make sure that routing from one core, CPU or coprocessor, does not cross the area occupied by the other core. Such routes could potentially be destroyed when a different core is reconfigured. The same holds for connections between the CPU and coprocessor cores and any specific resources, such as the BlockRAM, external memory, or I/O devices. We use floorplanning by location constraints to separate the CPU and coprocessor areas. Routing is constrained by following techniques:

– *Local routing:* We define connections between logic blocks that belong to the same core as local routing. Cores are constrained to rectangular areas on the FPGA. Depending on the size of the bounding-box and the core's requirements, some routes can cross the rectangle. Obviously, the rectangle can be enlarged to include the routing as well. Increasing the rectangle's dimensions by only a few CLBs proved to work well since the delay-based router prefers short local routes. Our experiments show that separating the initial areas for CPU and coprocessor cores by a

*safety border* of 4-6 CLBs successfully prevents the unintentional overlapping of routes from the two cores. This was also confirmed in [7].

– *Disturbing lines:* External devices, such as memories, are connected to the CPU core via I/O pins. We denote such non-local routes that cross the coprocessor area as disturbing lines, see Figure 3a). To control non-local routing, we introduce *feed-through* components. A feed-through component is a CLB with its LUT programmed to the identity function. Being a logic resource, a feed-through component can be constrained to any location on the FPGA. Disturbing lines are routed through one or more feed-through components that are placed such that the disturbing lines keep out of the coprocessor area, shown in Figure 3b). Placing the right number of feed-through components at the right locations is a manual process. However, this is needed only once for the initial full configuration.

– *Virtual Sockets:* The virtual socket is also built from feed-through components to allow the placement of the interface. The routing of signals around the CLBs that form the virtual socket is critical, as the virtual socket defines a border between static and reconfigured parts. Using a single feed-through CLB does not guarantee a sufficient safety margin. Therefore we have designed a special hard macro, the *2-CLB feed-through* component, using the Xilinx FPGA editor. The macro consists of two feed-through CLBs connected by a local, straight route. Figure 4 displays the interfacing area with the 2-CLB feed-through macros connecting to the coprocessor core.

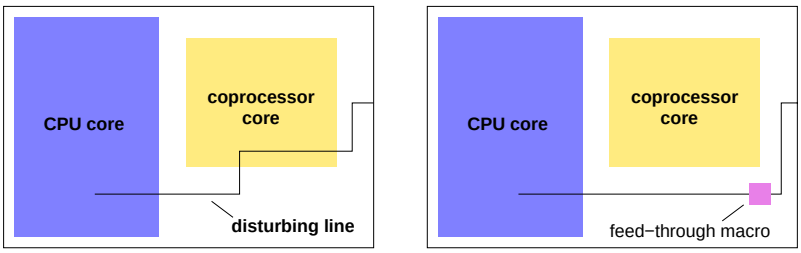


**Fig. 3.** Insertion of feed-through macros to avoid disturbing lines.

### 3.4 JBitsCopy

We have developed a JBits-based tool called *JBitsCopy*, that extracts fully synthesized circuits out of configuration bitstreams. JBitsCopy grabs all the FPGA resources, logic and routing, within a definable bounding box of an existing configuration and inserts it into another configuration. Additionally, JBitsCopy can relocate the grabbed design to different vertical and horizontal positions. The functionality of JBitsCopy is similar to the *JBitsRipper* tool described in [9], which targets Xilinx XC4000 series. JBitsRipper

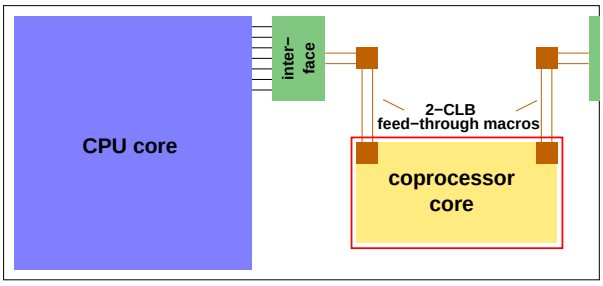


**Fig. 4.** The virtual socket uses 2-CLB feed-through macros to couple static and dynamic parts of the FPGA area.

allows to grab the configuration within a bounding-box and creates a Java program that reproduces this configuration.

## 4 Prototype implementation: Audio streaming

As case study and proof of concept for the proposed design flow, we have implemented a complete and fully operational audio decoding application [3]. The prototype consists of a minimal embedded computer based on a general-purpose CPU core, memories, network interface, and several coprocessors for hardware-accelerated playback of audio streams.

The CPU receives UDP packets containing encoded audio data from a network via an Ethernet interface. The CPU unpacks the audio data and sends it to the audio coprocessor's input FIFO via the virtual socket. The coprocessor decodes the audio stream and sends the raw audio data to the on-board digital-to-analog converter. Different formats for the encoding of audio data require different coprocessors. Depending on the audio format currently used, the audio decoders are dynamically configured into the prototype. The technical details of the prototype are:

– *Prototyping Platform*
  The prototype has been implemented on a XESS XCV800 board which consists of a Xilinx Virtex XCV800-4 FPGA and a multitude of I/O interfaces. A block diagram of the prototyping board is given in Figure 5.

The cores were designed in VHDL, synthesized and implemented using Synopsys FPGA Express 3.6.0 and Xilinx Foundation 4.1i tools, respectively.

– *CPU core*
  The soft CPU core is the SPARC V8 compatible 32bit LEON CPU [5]. The CPU was configured with 2kB separated data- and instruction-cache (implemented in internal BlockRAM), a 256 byte internal boot-ROM (implemented in internal distributed RAM) and an external 32bit memory interface. The CPU core requires 3865 Virtex slices which amounts to 41% of the XCV800's logic resources, and 14 BlockRAMs which equals 50% of the memory resources. Without any optimization, the CPU runs at 25 MHz. Applications for the LEON core run on top of the

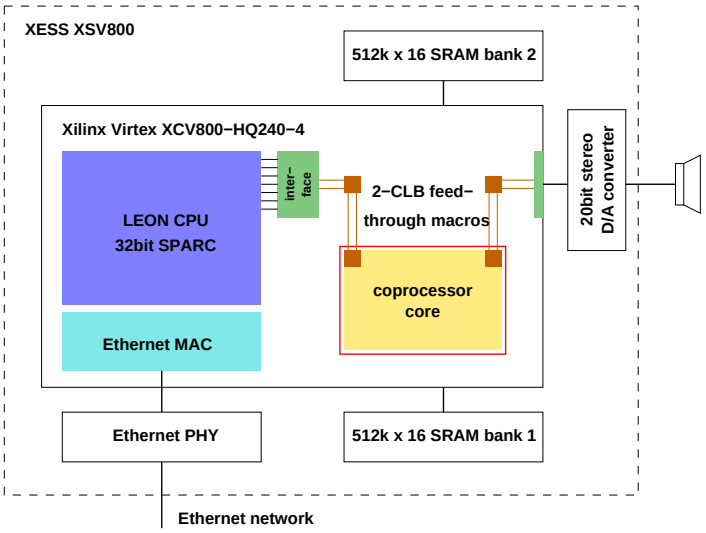


**Fig. 5.** Block diagram of the audio streaming prototype.

RTEMS [2] real-time operating system and are compiled using the GNU C based LECCS cross-compiler kit [4].

– *Coprocessor cores*
We have implemented two audio decoding coprocessors, a PCM and an Intel/DVI compliant ADPCM decoder. The ADPCM core runs at 50 MHz and uses 430 Virtex slices, or 4.5% of the XCV800 resources; the PCM decoder fits into 35 slices, or 0.4% of the resources.

We envision the prototype application as a typical scenario for future embedded networked systems that load hardware functions on demand. The current limitations of our prototype are that the reconfiguration has to be initiated by the user and that the partial configurations are loaded onto the FPGA from a host computer via a configuration port. We are working to extend the prototype to perform fully autonomous reconfiguration. There, the configuration bitstreams are stored locally in the prototype and CPU software recognizes the audio format used and initiates reconfiguration. A straight-forward implementation uses a small controller in the system to control reconfiguration. However, recent advances in Virtex technology, i.e., internal configuration access port in Virtex-II, seem to make a reconfiguration controlled by the FPGA itself feasible. Our goal is to construct such an embedded networked reconfigurable system prototype.

## 5 Conclusions and future work

In this paper, we have discussed a design flow to implement hybrid CPUs and demonstrated it on the working example of one CPU and two coprocessor cores. The design flow is, however, no restricted to one single static and one single reconfigurable part.

Extensions to several partial reconfigurable subareas are straight-forward. In such a system, interfaces can be established by on-chip busses to which the reconfigurable modules connect [1].

We plan to extend our work in the following directions:

- Construction of a prototype with fully autonomous reconfiguration.
- Development of hybrid CPUs employing multiple reconfigurable coprocessors connected via on-chip busses.

## 6 Acknowledgements

We would like to thank Marco Wirz for his work on the prototype, including the design of the network subsystem, and Herbert Walder for the JBitsCopy tool and for sharing his JBits experience.

## References

1. G. Brebner and O. Diessel. Chip-Based Reconfigurable Task Management. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pages 182–191, 2001.
2. O. Corp. RTEMS Homepage. http://www.rtems.com.
3. M. Dyer and M. Wirz. Reconfigurable System on FPGA. Master's thesis, Computer Engineering and Networks Lab, ETH Zurich, March 2002.
4. J. Gaisler. LECCS: LEON/ERC32 Cross Compilation System. http://www.gaisler.com/leccs.html.
5. J. Gaisler. *The LEON Processor User's Manual*. Gaisler Research, version 2.3.7 edition, August 2001.
6. S. A. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based Interface for Reconfigurable Computing. In *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 2000.
7. A. Haase. Untersuchungen zur dynamischen Rekonfigurierbarkeit von FPGA. Master's thesis, TU Chemnitz-Zwickau, Germany, September 2001. (in German).
8. E. L. Horta and J. W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical report, Department of Computer Science, Applied Research Lab, Washington University, Saint Louis, July 2001.
9. P. James-Roxby, E. Cerro-Prada, and S. Charlwood. A Core-based Design Method for Reconfigurable Computing Applications. In *Proceedings of the IEE Colloquium on Reconfigurable Systems*, Glasgow, March 1999. IEE Informatics.
10. D. Lampret. *OpenRISC 1200 IP Core specification*. www.opencores.org, 2001.
11. P. Leong, C. Sham, W. Wong, W. Yuen, and M. Leong. A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm. *IEEE Transactions on VLSI Systems*, 9(1):197–201, February 2001.
12. S. Singh and P. James-Roxby. Lava and JBits: From HDL to Bitstream in Seconds. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2001.
13. Triscend Corp. *Triscend A7 Datasheet*, 2001.
14. Xilinx Inc. *Xilinx Application Note XAPP151: Virtex Series Configuration Architecture User Guide*, v1.5 edition, 9 2000.
15. Xilinx, Inc. *Virtex-II Pro Platform FPGA Handbook*, January 2002.