Design flow for embedded FPGAs based on a flexible architecture template

B. Neumann, T. von Sydow, H. Blume, T. G. Noll
Chair of Electrical Engineering and Computer Systems
RWTH Aachen University
Schinkelstr. 2, 52062 Aachen, Germany
email: {neumann, sydow, blume, tgn}@eecs.rwth-aachen.de

Abstract

Modern digital signal processing applications have an increasing demand for computational power while needing to preserve low power dissipation and high flexibility. For many applications, the growth of algorithmic complexity is already faster than the growth of computational power provided by discrete general purpose processors [1]. A typical approach to address this problem is the combination of a processor core with dedicated accelerators. Since changes in standards or algorithms can change the demands on the accelerators, an attractive alternative to highly customised VLSI-macros is the use of reconfigurable embedded FPGAs (eFPGAs). First commercial products combining a general purpose processor core and an embedded FPGA recently emerged (e.g. Stretch S6000 [2], Menta eFPGA-augmented CPUs [3]). For many digital signal processing applications, a significantly improved efficiency in terms of power dissipation, throughput and chip area can be achieved by tailoring both the processor core and the reconfigurable accelerator to the given application domain [4].

In this work, a methodology to design highly customisable eFPGA-architectures starting from a high level description is presented. The design framework elaborated during this work enables a physically optimised VLSI-design of the specified eFPGA and aims to support simulation of the according eFPGA-macros both on a functional and netlist-level by providing an elementary configuration tool based on the same high level description as the eFPGA-architecture.

1. Introduction

FPGAs are widely used as an attractive compromise between highly efficient physically optimised VLSI-designs and software programmable processors. Due to their reconfigurability, FPGAs are highly flexible and allow for relatively short design cycles since no physical changes to the underlying hardware have to be made in case of a redesign. However, they offer lower physical implementation costs compared to software programmable processors, as the inherent parallelism of many algorithms can be exploited in contrast to sequential processor architectures.

As a result, commercial FPGA-architectures have been optimised to suit a wide variety of applications from network related and digital signal processing to the realisation of soft core processors. For an embedded FPGA used as configurable accelerator, however, the requirements concerning the provided resources are often well defined and much narrower than for discrete "general purpose" FPGAs. Hence, eFPGAs can be optimised for a certain set of applications and thus achieve higher efficiency in terms of power dissipation, area and speed. First investigations on a reconfigurable ASIP with a reconfigurable accelerator based on a parametrisable eFPGA-architecture have shown significant improvements in energy- and area-efficiency [5].

2. Parametrisable eFPGA-architecture

2.1. Overview

The eFPGA architecture presented here is based on a highly parametrisable architecture template targeting an arithmetic-oriented application domain. Figure 1 shows an overview of the complete architecture template with all relevant parameters. Some of them are described by a single value (e.g. the number of LEs in a row and column), while others require a more complex definition (e.g. the connectivity per switch point). In the following, the architectural components and the according parameters are discussed in detail.

978-3-9810801-3-1/DATE08 © 2008 EDAA

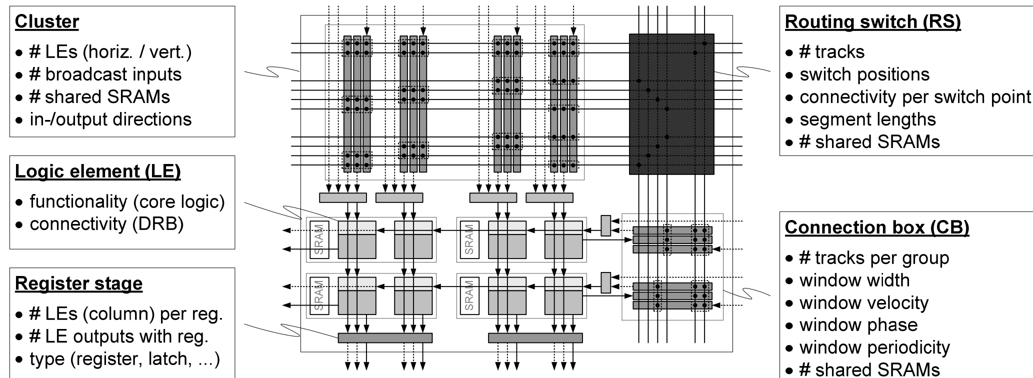

Figure 1. eFPGA architecture template

2.2. Cluster

A typical characteristic of arithmetic datapaths is the organisation in function slices and bit slices. A function slice represents one of many consecutively processed elementary functions (e.g. n-bit addition, n-bit XOR-operation etc.), while a bit slice represents all processing elements in the same column corresponding to the same bit value (e.g. bit 0 of two successive function slices). Figure 2 shows an according scheme of processing elements organised in function slices and bit slices.

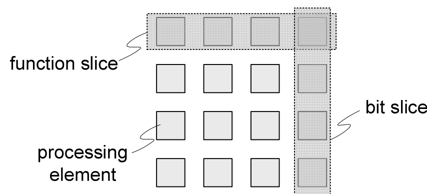

Figure 2. Typical arithmetic datapath scheme

Most communication between function slices and bit slices is local, i.e. only between direct neighbours. In addition, operands are typically fed to the datapath using a broadcast scheme.

The eFPGA-architecture reflects typical arithmetic datapath schemes by using two-dimensional clusters of logic elements with a distributed interconnect rather than one-dimensional clusters with a central connection box. The signals coming from the connection box are distributed to the logic elements in rows and columns according to the function slices and bit slices, such that all logic elements in a row or column share the same input signals using so-called broadcast lines. This reduces the number of signals that need to be provided by the connection box and hence reduces the significant overhead imposed by the configurable connection boxes. Figure 3 shows the organisation of logic elements in a cluster and the according global routing resources. The size of the cluster can be varied in the horizontal and vertical direction independently. Also, the number of broadcast lines per row and column can be changed in designs based on the template. Broadcast lines can be fed to the cluster from all four directions, and in the same way the outputs of the LEs at all four cluster borders can be fed to the connection box. The actual connectivity can be any set of the four possible data directions (north, east, south, west) for inputs and outputs independently. Between adjacent clusters, feedthrough stages are provided to use the broadcast lines of the neighbouring clusters as inputs for the current cluster, hence creating virtually larger clusters by cascading several of them.

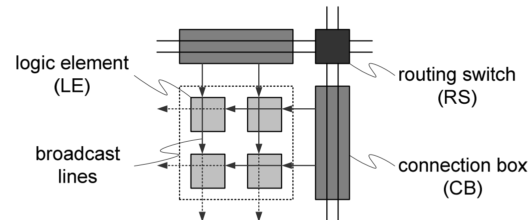

Figure 3. Cluster architecture

2.3. Logic elements

The local connectivity between the logic elements is provided by dedicated routing blocks (DRB) located in the logic element (see Figure 4). Each DRB is a set of multiplexers used to connect broadcast signals or local signals to the core logic of the logic element. The actual connectivity can be defined in the architecture template by stating all sources connected to the DRBs with their offset to the actual LE as illustrated in Figure 4. The functionality of the core logic itself is specified by a list of elementary boolean functions that the LE can process (e.g. full addition, gated full addition etc.). Registers can be inserted per logic element or with a reduced density, e.g. every second LE-row.

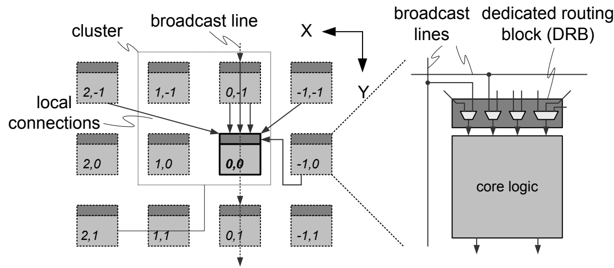

Figure 4. Local connectivity of logic element

2.4. Configuration memories

To reduce the overhead of the memory cells (typically SRAMs) used to store the configuration of the FPGAs logic and routing resources, the present architecture template allows for sharing the configuration bits and thus configuring several adjacent elements identically as illustrated in Figure 5. This scheme is applied to the logic elements as well as to the interconnect resources, where adjacent switch points or connection points can share a single SRAM block. The degree of SRAM sharing can be adjusted in the architecture template in reasonable limits.

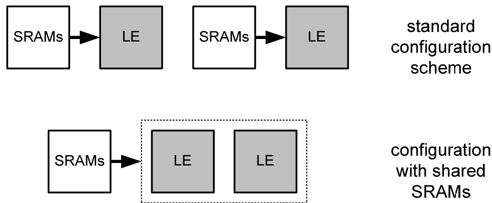

Figure 5. Shared SRAMs for logic elements

2.5. Routing switch

The routing switch of an FPGA is a set of switch points that are located at crossing points of horizontal and vertical routing tracks. The number of switch points available as well as their connectivity determines the flexibility of the complete routing switch. It was shown that it is not necessary to provide a fully populated routing switch to achieve a good amount of flexibility [6]. The architecture template presented here is very flexible concerning the definition of available routing resources. The number of routing tracks in horizontal and vertical direction can be chosen independently. Each switch point is defined by its position in the matrix of crossing lines as well as the connectivity inside the switch point. Different switch points can have different flexibility. In addition, the segmentation of the interconnect can be adjusted by assigning each routing track a certain segment length (corresponding to the number of routing switches that are bypassed before the line connects to the next routing switch). Figure 6 shows the architecture and the main parameters of the routing switch used in the architecture template.

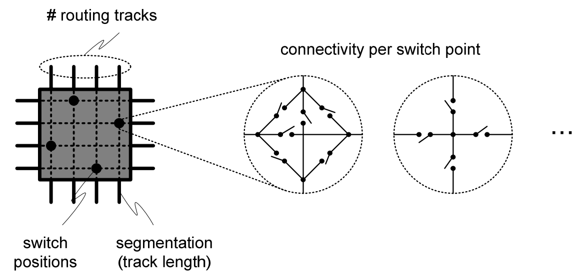

Figure 6. Parametrisable routing switch

2.6. Connection box

Finally, the architecture template offers a highly flexible description of the connection box similar to the routing switch definition. Three types of routing channels are supported: fully connected, periodic connectivity and unconnected. Fully connected tracks offer full population of the connection box, i.e. each track can connect to each according broadcast line of a cluster. However, they have the highest implementation costs. Unconnected tracks can be implemented for fast signal routing, as the capacitive load of these wires can be kept very low. Periodic tracks use a special connection type best suited for arithmetic datapaths, where signals on a bus are typically ordered by the weight of their bits. Accordingly, periodic routing channels have a window of connection points that "slides" across the tracks with a given "velocity". The connection box defined in the architecture template can be composed of any mix of tracks with different channel widths and sliding window specifications. The architecture of the connection box and the according parameters are illustrated in Figure 7

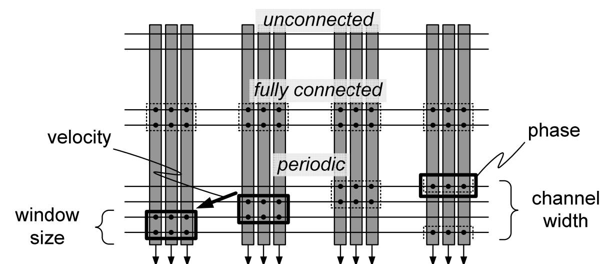

Figure 7. Parametrisable connection box

3. Design flow

As many of the architectural features in the presented eFPGA are unique and not common in standard FPGAs, there is currently no tool support available. Most research conducted in the field of (e)FPGA-architectures is based on the VPR design flow [7] which can only be used to

model standard island style FPGA-architectures with LUT-based logic elements and a small choice of routing switch architectures. Hence, an important goal of this work is the creation of a self-contained design methodology to design application domain specific eFPGAs and the according basic tool support. Figure 8 shows the overall design flow applied here.

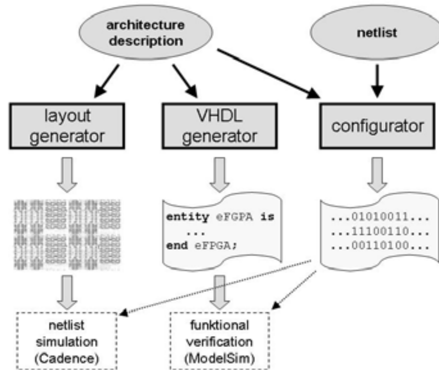

Figure 8. Design flow

The architecture template described above was formulated as a high level description using MATLAB. Based on this architecture description, three main steps are supported by the design flow. First, a layout generator creates a VLSI-layout of the specified eFPGA based on a small set of handcrafted, physically optimised basic cells such as the multiplexers for the DRB or the switch points of the routing switch. Several studies concerning this part of the design flow have been published before [8][9].

While the first automatically generated eFPGA-layouts still needed to be configured manually (i.e. each SRAM cell had to be configured with the proper value), an automatic bitstream generator supporting the complete architecture template is currently under construction. The configuration bitstream is used to conduct netlist simulations based on layout-extracted netlists. To verify the functionality of the eFPGA-macro, a VHDL-generator creates a functional description that can be simulated using common simulation tools like ModelSim. The output of the configurator is based on a netlist that describes the signal flow graph mapped to the eFPGA. Currently, the netlist description is still complex, as each logic element and each routing resource has to be described here. However, for arithmetic datapaths this netlist is highly regular which reduces the effort to generate it manually. Currently, a very time-consuming and error-prone work is the generation of the configuration bits and the routing of signals between the logic elements. The placement process is less complex due to the regularity of the examples considered here (i.e. arithmetic datapaths). Consequently, a future step will be the implementation of a routing tool supporting the parametrisable architecture.

Similar approaches to automatic (e)FPGA design have been proposed e.g. with GILES [10] or PYTHAGOR [11]. However, those design flows have significant constraints regarding the FPGA-architectures (e.g. only island style FPGAs are supported) and the physical implementation style. As an example, using standard cell implementations (e.g. proposed by PYTHAGOR) leads to unfavourable physical implementation costs concerning area, performance and power dissipation. The first results presented here are based on a completely functional co-design of a routing switch layout, the according VHDL-model and the configuration bitstream based on a given netlist.

3.1. Layout generator

The layout generator is based on a prior work on automated VLSI-design of regular datapaths [12]. This so-called datapath generator uses a textual description of a signal flow graph (SFG) and a small set of hand designed layout cells to generate a layout. Since the textual SFG description can be parameterised, the datapath generator allows for a very flexible implementation process, e.g. when parameters like word lengths are changed in the SFG. Starting from the MATLAB-based high level description of the eFPGA architecture a datapath generator suited SFG description is automatically generated. Due to the highly modular design style, the eFPGA-macro can be ported to different CMOS-technologies with small effort, since only few hand designed layout cells are required.

After the layout is generated, standard netlist extraction and simulation tools can be used to characterise the eFPGA macro in terms of area, timing and power dissipation.

3.2. VHDL-generator

Based on the architecture description, a VHDL-model of the eFPGA is created automatically. It incorporates the functional description of the basic configurable elements like routing switch points or logic elements and combines them according to the architectural parameters. The VHDL-model of the eFPGA is used to verify the functionality defined by the netlist using existing simulation tools. It is also useful to test the eFPGA macro created by the layout generator for correctness by co-simulation of the layout-extracted netlist and the functional VHDL-model.

3.3. Configurator

To enable simulations of the eFPGA-macro (on functional as well as on netlist level), all configuration

bits have to be set properly. Due to the very large number of configuration bits, it is necessary to have an automated way of creating the bitstream from the mapped netlist. Existing bitstream generators like DAGGER [13] lack the support for highly parametrisable (e)FPGA architectures as the one described here.

The configurator elaborated as part of the design methodology presented here creates configuration bitstreams based on the netlist and the architecture specifications. It also uses the information from the layout generator to determine the actual position of all blocks to be configured in the macro. The configuration bitstream is composed of elementary configuration table entries that must be provided for the basic eFPGA elements like routing switch points or logic elements. The elementary tables can be created with small effort, as only few bits are required to configure these basic elements. The bitstream is then concatenated according to the position of the elements in the overall macro.

4. Design example

4.1. Routings Switch

As a first step in verifying the proposed design methodology, a routing switch generator comprising all elements of the design flow was implemented based on the architecture described in paragraph 2.5. As an example, a routing switch with 32 tracks both in horizontal and vertical direction was specified in the according architecture description. Switch points with different flexibility are provided as exemplary basic components for the routing switch. As the MATLAB-based description is on an abstract level, the architecture specification for the complete routing switch can be created within few minutes. Figure 9 illustrates the syntax describing the architecture.

Three global parameters describe the channel widths (*rs.trk_h, rs_trk_v*) and the use of configuration sharing (*shd_srams.rs*). Each switch point is defined by a set of potential signal routes according to their input and output directions (north, east, south, west). From the flexibility required by each switch point, the routing switch generator extracts a set of basic layout cells that need to be designed for the VLSI-implementation. The design of the according switch point macros and the SRAM-cell required for the configuration storage takes some hours for a skilled designer.

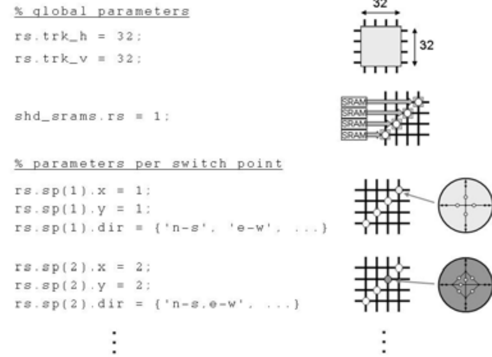

Figure 9 Code segment and acc. architecture

The layout generator automatically determines the optimum placement of the basic cells for a given aspect ratio of the routing switch. It also calculates the optimum aspect ratios of the configuration blocks that are required per set of switch points sharing the same configuration. The automatic layout generation of the complete routing switch by the layout generator takes less than 20 minutes on a Sun UltraSPARC III (1GHz, 4GB RAM). Consequently, redesigns of the routing switch e.g. to analyse the influence of different parameters on the area, timing and power dissipation can be conducted very quickly.

A simple netlist describing the connections to be provided by the routing switch was used to automatically generate the configuration bitstream with more than 200 bits and the testbench for ModelSim (functional simulation) and Cadence Spectre (netlist simulation). Figure 10 shows an exemplary implementation of the specified routing switch in a 90nm CMOS-technology.

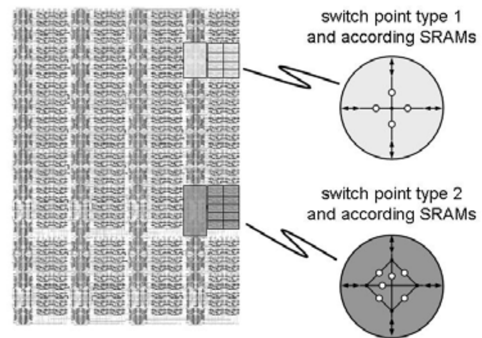

Figure 10 VLSI-layout of a routing switch

4.2. Exemplary mapping of an FIR-filter

To verify the efficiency of the overall eFPGA-architecture, an exemplary datapath was mapped to a complete eFPGA by hand. The routing switch and the according configuration bits were generated automatically as described above. The other components (logic elements etc.) were designed with the according basic cell layouts and configured by hand. Using this semi-automatic

approach, it was possible to map a 4-tap FIR-filter to the eFPGA with reasonable effort. To rank the efficiency of the architecture, the datapath was also mapped to a commercial FPGA (Altera Cyclone). Figure 11 shows the according results in the design space in terms of mW/MOPS and $MOPS/mm^2$ based on a diagram taken from [8]. The grey areas represent the regions in the design space for different hardware architectures that were determined based on actual implementations. The efficiency of the optimised eFPGA-architecture is about an order of magnitude better compared to the commercial FPGA. In contrast to the work described in [8], where only basic functions like a multiplication could be mapped to the eFPGA, the mapping of a more complex example was enabled by the automated design flow described here.

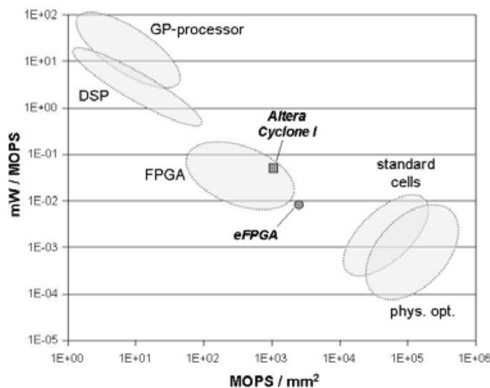

Figure 11 FIR-filter on eFPGA: design space

5. Conclusion

The design methodology presented in this paper is an important step for the evaluation of embedded FPGAs that are optimised for a certain application domain. By using a common, highly flexible architecture template, the eFPGA-architecture can be tailored to a given application domain systematically. The self-contained design methodology presented here enables the VLSI-design as well as basic tools for verification and simulation. Hence, the complexity of mapping exemplary datapaths to the eFPGA is reduced significantly compared to previous work. Using the simulation results based on actual VLSI-layouts of the eFPGA, a high-level model of the architecture is currently evolving that allows for a systematic analysis of the dependencies between eFPGA-architecture, mapped datapaths and the according efficiency. The exemplary result for an FIR-filter demonstrates the optimisation potential of FPGA-architectures when tailored for a given application domain.

6. Acknowledgements

This work is funded by the German Research Foundation (DFG) as part of the DFG Priority Program 1148 (Reconfigurable Computing Systems).

References

[1] J. Hausner: “Integrated Circuits for Next Generation Wireless Systems”, Proceedings of the European Solid-State Circuits Conference (ESSCIRC) 2001, pp. 26-29

[2] Stretch S6000 (website), http://www.stretchinc.com

[3] MENTA eFPGA-augmented RISC CPUs (website), http://www.menta.fr/efpga_cpu.html

[4] A. Ye and J. Rose, “Using Bus-Based Connections to Improve Field-Programmable Gate Array Density Implementing Datapath Circuits”, in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, No. 5, pp. 462-473, May 2006.

[5] T. von Sydow, M. Korb, B. Neumann, H. Blume and T. G. Noll, “Modelling and Quantitative Analysis of Coupling Mechanisms of Programmable Processor Cores and Arithmetic Oriented eFPGA-macros”, in Proc. Reconfigurable Computing and FPGA's 2006 (ReConFig '06), pp. 252-261, 2006.

[6] G. Lemieux and D. Lewis, “Design of Interconnection Networks for Programmable Logic” Kluwer Academic Publishers, 2004.

[7] V. Betz, J. Rose and A. Marquardt, “Architecture and CAD for Deep-Submicron FPGAs” in Kluwer International Series in Engineering and Computer Science, 1999.

[8] T. von Sydow, B. Neumann, H. Blume and T. G. Noll, “Quantitative Analysis of embedded FPGA Architectures for Arithmetic”, in Proc. Application Specific Systems, Architectures and Processors Conference 2006 (ASAP ’06), pp. 125-131, 2006.

[9] B. Neumann, T. von Sydow, H. Blume and T. G. Noll, “Design and quantitative analysis of parametrisable eFPGA-architectures for arithmetic“ in Advances in Radio Science, Vol. 4, pp. 251-259, 2006.

[10] I. Kuon, A. Egier and J. Rose, “Design, Layout and Verification of an FPGA using Automated Tools”, in Proc. 2005 ACM/SIGDA 13th international symposium on Field programmable gate arrays, pp. 215–226, 2005.

[11] A. Danilin, M. Bennebroek and S. Sawitzki, “A novel toolset for the development of FPGA-like reconfigurable logic”, in Proc. FPL 2005, pp. 640-643, 2005.

[12] O. Weiss, M. Gansen and T. G. Noll, “A flexible Datapath Generator for Physical Oriented Design” in Proc. European Solid-State Circuits Conference 2001 (ESSCIRC ‘01), pp. 408-411, 2001.

[13] K. Siozios et. al. “DAGGER: A Novel Generic Methodology for FPGA Bitstream Generation and its Software Tool Implementation”, in Proc. Parallel and Distributed Processing Symposium 2005 (IPDPS ‘05), p. 165b, 2005.