DISSERTATION

Design Methodology for Custom Reconfigurable Logic Architectures

Design of Application Domain Specific Mixed-Grained Ultra-Low-Power Reconfigurable Logic for Control and Computation in Wireless Sensor Networks

Submitted at the Faculty of Electrical Engineering, Vienna University of Technology, in partial fulfillment of the requirements for the degree of Doctor of Technical Sciences

under supervision of

Univ.Prof. Dr. habil. Christoph Grimm
Workgroup for Design of Cyber-Physical Systems
Technische Universität Kaiserslautern

and

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Institute number: 384
Institute of Computer Technology
Vienna University of Technology

by

Johann Glaser
Matr.Nr. 9825761
Ebenhochstraße 8/EG/10, 4020 Linz

September 2015 _____

Kurzfassung

Aktuelle Trends in Richtung allgegenwärtiger Elektronik und dem Internet of Things (IoT) verlangen eine geringe Leistungsaufnahme der Komponenten, zum Beispiel der Knoten eines Funksensornetzwerks. Ein Ansatz zur Reduktion des Energieverbrauchs ist die CPU durch autonome Module zu entlasten. Diese übernehmen einfache Aufgaben, z. B. periodische Sensormessungen. Dadurch kann die CPU länger in einem inaktiven Low-Power Modus verbleiben. Sie wird nur aktiviert, wenn komplexere Aufgaben ausgeführt werden müssen, z. B. um einen neuen Wert über das Funknetzwerk zu übertragen. Solche autonomen Module müssen rekonfigurierbar sein um unterschiedliche Ansprüche zu erfüllen, um an neue Umgebungen angepasst zu werden und um Fehler zu korrigieren. In dieser Dissertation wird eine neue Methodik zur Entwicklung solcher rekonfigurierbarer Module vorgestellt.

Im Gegensatz zur Entwicklung mit FPGAs, bei der Chips mit einer vordefinierten Architektur konfiguriert werden, beinhaltet die vorgestellte Methodik die Entwicklung der Halbleiterschaltung. Die rekonfigurierbaren Module müssen sowohl digitale Steuerungslogik als auch Datenverarbeitung unterstützten. Um die Chipfläche und Leistungsaufnahme zu verringern wird eine gemischt-granulare Logikarchitektur eingesetzt. Neben feingranularen Funktionseinheiten und Signalen beinhaltet diese auch grobgranulare Funktionseinheiten mit komplexerer Funktionalität, die Signalvektoren mit mehreren Bits verarbeiten. Das erfordert, dass heterogene, also mehrere verschiedene Arten von Funktionseinheiten integriert werden. Daraus folgt, dass jedes rekonfigurierbare Modul spezifisch für den gegebenen Anwendungsbereich entwickelt werden muss.

Aktuelle Entwicklungsmethoden für rekonfigurierbare Logikarchitekturen sind auf den Anwendungsbereich Datenverarbeitung limitiert und unterstützen keine digitale Steuerungslogik. Diese Ansätze verwenden entweder grobgranulare oder feingranulare, aber unterstützen keine gemischtgranularen Architekturen. Die Funktionseinheiten der rekonfigurierbaren Logik müssen entweder explizit instanziiert oder manuell zugeordnet werden. Aktuelle Architekturen für rekonfigurierbare Zustandsautomaten benötigen viel Chipfläche oder verursachen hohen Stromverbrauch.

Die Entwicklungsmethodik, die in dieser Dissertation vorgestellt wird, ist die erste, die beides, digitale Steuerungslogik als auch Datenverarbeitung, unterstützt. Sie ist universell und unabhängig vom Anwendungsbereich des rekonfigurierbaren Moduls. Die zugrundeliegende Architektur ist eine Sammlung von rekonfigurierbaren Funktionseinheiten, die mit einem rekonfigurierbaren Interconnect verbunden sind, und unterstützt ausdrücklich gemischt-granulare Logik.

Die Funktionalität eines rekonfigurierbaren Moduls wird mit einem Satz von Beispielapplikationen als VHDL oder Verilog Logikdesigns angegeben. Das ermöglicht die Definition von digitaler Steuerungslogik und Datenverarbeitung gemeinsam mit allen Signalen und Zyklus-genauem Timing. Von diesen Beispielapplikationen werden die Funktionseinheiten und das Interconnect in einem neuen semi-automatischen Verfahren optimiert. Das so entstandene rekonfigurierbare Modul kann jede Beispielapplikationen implementieren und stellt zusätzlich Flexibilität für neue Applikationen bereit. Um diese Flexibilität zu erhöhen können bei der Optimierung des Moduls zusätzliche Funktionseinheiten und Verbindungen hinzugefügt werden. Das rekonfigurierbare Modul wird für die Integration in einem Chip als IP Core erstellt.

Die Entwicklungsmethodik ist die erste, die die Verifikation der Beispielapplikationen, aller Zwischenschritte und des generierten rekonfigurierbaren Moduls einbezieht. Dazu werden Simulation und Logical Equivalence Checking verwendet, um die Übereinstimmung mit der Spezifikation sicherzustellen. Zusätzlich zu den Beispielapplikationen, die für die Entwicklung verwendet wurden, können mit dem rekonfigurierbaren Modul auch neue Applikationen implementiert werden.

Neben der Entwicklungsmethodik wird in dieser Dissertation eine neue rekonfigurierbare Architektur für Zustandsautomaten eingeführt, um digitale Steuerungslogik effizient zu implementieren.

Die Entwicklungsmethodik wurde als Entwicklungsumgebung implementiert, die speziell angefertigte, Open-Source und kommerzielle Programme integriert. Alle Tätigkeiten, die nicht notwendigerweise manuell sind, sind automatisiert, um den Entwickler zu entlasten und um hohe Produktivität sicherzustellen. Mit der Entwicklungsumgebung wurde ein exemplarischer Funksensornetzwerkknoten als SoC mit einem rekonfigurierbaren Modul entwickelt. Der WSN SoC wurde in einem 350 nm CMOS Prozess hergestellt. Er implementiert alle Beispielapplikationen und alle neuen Applikationen korrekt. Damit wurde die Tauglichkeit der Entwicklungsmethodik belegt. Das rekonfigurierbare Modul erzielt eine 180-fache Reduktion des Energieverbrauchs für Sensormessungen im Vergleich zur integrierten CPU. Die Chipfläche ist 2.2 mal größer als die gleichzeitige Integration aller Beispielapplikationen und neuen Applikationen, aber 4.0–4.3 mal kleiner als embedded FPGAs. Das rekonfigurierbare Modul benötigt 9.1–23.4 mal weniger Konfigurationsdaten als embedded FPGAs. Weiters stellt es genug Flexibilität zur Verfügung um verschiedene neue Applikationen zu implementieren und um Probleme von Beispielapplikationen zu korrigieren.

Abstract

In current trends towards ubiquitous computing and the Internet of Things (IoT), low power consumption is of increasing concern, for example in wireless sensor network (WSN) nodes. One approach to reduce power consumption is to off-load the CPU by autonomous modules. These relieve the CPU from simple tasks, e.g., performing periodic sensor measurements. The CPU in turn stays in an inactive low-power mode for extended periods. It is only activated if more complex tasks have to be accomplished, such as communicating a new value via the wireless network. Such autonomous CPU supplement modules must be reconfigurable to suffice different requirements, to adapt to new environments, and to fix design issues. In this thesis a new design methodology for the development of such reconfigurable CPU supplement modules is introduced.

Contrary to FPGA design, where chips with a predefined reconfigurable architecture are configured, the proposed methodology includes the development of the silicon circuitry itself. The reconfigurable modules have to support both, control-dominated tasks as well as data processing. To reduce silicon area and power consumption, the approach utilizes a mixed-granularity logic architecture. Besides fine-grained functional units and signals, this adds coarse-grained functional units with more complex functionality and operating on multi-bit vectors. This requires that heterogeneous, i.e., multiple different kinds of functional units, are integrated. This further requires, that each reconfigurable module is specifically developed for its given application domain. The design methodology proposed in this thesis addresses this task.

State of the art design methodologies for reconfigurable logic architectures are limited to application domains for data processing but do not support control-dominated tasks. These approaches either use coarse-grained or fine-grained architectures, but do not provide mixed granularity reconfigurable logic. The functional units of the reconfigurable logic either have to be instantiated explicitly or are mapped manually. State of the art architectures for reconfigurable finite state machines (FSMs) require large chip area or cause high power consumption.

The design methodology introduced in this thesis is the first which supports both, control-dominated and data processing tasks. It is universal and independent of the application domain of the reconfigurable modules. The underlying architecture is defined as a collection of reconfigurable functional units connected via a reconfigurable interconnect and specifically supports mixed granularity logic.

The functionality of reconfigurable modules is specified with a set of example applications as VHDL or Verilog logic designs. These enable the definition of control-dominated as well as data processing tasks including all signals and cycle accurate timing. From these the functional units and the interconnect are optimized in a novel semi-automatic procedure. The resulting reconfigurable module can implement any of the example applications and provides flexibility for future use cases. In order to further increase its flexibility, additional functional units and routing resources can be included during the optimization. The reconfigurable module is delivered as an IP core for the integration in a chip design.

The design methodology is the first to incorporate the verification of the example applications, of all intermediate steps, and of the generated reconfigurable module. Simulation and logical equivalence checking are used to ensure full compliance to the specification. Besides the example applications used in the development, new applications can be implemented with the reconfigurable module.

Additional to the design methodology, in this thesis a novel reconfigurable architecture for FSMs is introduced, to improve the support of control-dominated tasks.

The design methodology was implemented as an EDA design flow incorporating custom, open-source, and commercial tools. All tasks which are not essentially manual are automated to assist the designer and to achieve high productivity. The design flow was used to develop an exemplary WSN node SoC including a reconfigurable sensor interface module. The WSN SoC was produced in a 350 nm CMOS process. It correctly implements all example applications and new applications. This demonstrates the feasibility of the design methodology. The reconfigurable module shows a 180-fold reduction in energy consumption for sensor measurements, compared to the integrated CPU. Its chip area is 2.2 times larger than the parallel implementation of all example and new applications but 4.0–4.3 times smaller than embedded FPGA implementations. The reconfigurable module requires 9.1–23.4 times less configuration data than embedded FPGAs. Additionally it provides enough flexibility to implement diverse new applications and to fix design issues of example applications.

Acknowledgments

This PhD was a large project and could not have been performed successfully without the help of many supporters. I want to express my gratitude to you.

First I want to thank my supervisor Univ.Prof. Dr. habil. Christoph Grimm for his guidance, support, patience, for giving me the freedom to explore, and for the urge every once in a while to stop with the perfectionist improvements and to come to an end. Thank you also for putting me in charge of the "Invent a Chip" pupils competition, which helped the deepening of my understanding of the ASIC design flow.

My thanks also go to my second supervisor Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch for his assistance and feedback. Thank you for asking the right questions at the right time, which provided valuable orientation and drive to complete this work.

I also want to thank my colleagues Sumit Adhikari, Florian Brame, Markus Damm, Klaus Gravogl, Jan Haase, Georg Möstl, and Florian Schupfer. Your support, inspiring discussions, help, and proof-reading made the work on the subject a pleasant and memorable experience and greatly improved the outcome.

My thanks also go to the students Armin Faltinger, Mario Faschang, Peter Hanger, Sebastian Plunger, and especially to Georg Blemenschitz, Martin Schmölzer, and Clifford Wolf for the implementation of designs and tools which greatly supported my work. Thank you for unhesitatingly including my suggestions and wishes, and for the numerous interesting discussions.

Further, I want to thank Infineon Technologies AG and ams AG for the manufacturing of the test chips. My thanks also go to the colleagues and design support staff for help with the tools, design elements, and design rules, and for helping me to find confidence in the designs before tape-out.

I also want to thank the Institute of Computer Technology (ICT) at the Vienna University of Technology (TU Wien) and the Research Institute for Integrated Circuits (RIIC) at the Johannes Kepler University Linz (JKU) for providing a productive and supportive working place. Special thanks go to my colleagues for nice chats during breaks, for joint meals, for helping hands, for collegiality and friendship, and for moral support when times got tough.

Lastly, I would like to thank my friends and my family for accompanying this period. Thank you for sharing laughs, for listening, and for being part of a supportive and reassuring network.

This work has been supported in part by the Sensor Network Optimization through Power Simulation (SNOPS) project, which was funded by the Austrian government via FIT-IT (grant number 815069/13511) within the European ITEA2 project GEODES (grant number 07013), and by the Austrian COMET K-project ECV under contract number 815105.

Johann Glaser
Vienna University of Technology
September 2015

Table of Contents

1

Introduction

Many areas of modern life are surrounded with embedded systems. The guiding paper [Wei91] coined the term "Ubiquitous Computing" back in 1991 for this trend. In the meantime, embedded systems became an important economic factor. [Jos14] estimated the worldwide market for embedded technology in 2013 with a total of $142.8 billion. For the next five years, he expected an annual growth of 5.4%, which leads to $152.4 billion for 2014 and $198.5 billion for 2019.

On one hand, embedded systems enable the operation and development of ever more complex and optimized systems, e.g., RFID in logistics, sensors and actors in building automation, communication systems in networks, etc. On the other hand, embedded systems themselves have become consumer products in a fast growing market, for example mobile phones, tablets, PDAs, paddles, function watches, sports devices, heart rate monitors, and GPS trackers. Constraints in size, energy efficiency and not at least cost, raise challenges to the development of embedded systems.

With the increasing popularity of embedded systems, the demands on usability are increasing too. One factor is the battery lifetime, which strongly depends on the power consumption of the device. To reduce the power consumption, techniques can be applied at all levels of the design process. At the physical level, techniques like voltage scaling or pass-transistor logic are applied. At the logic and architecture level the methods include gate sizing and clock gating. Even at the highest software and system level, approaches like compression of the code or energy aware task scheduling provide energy reduction [BMM01].

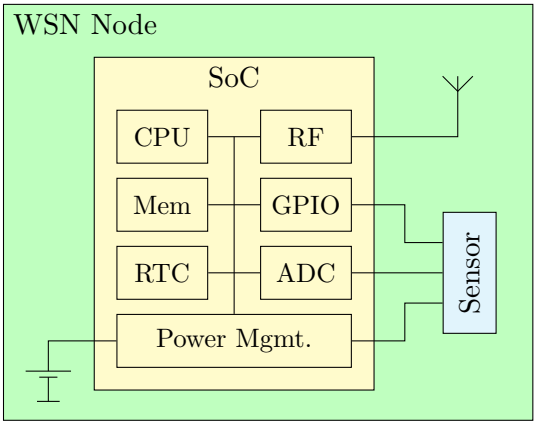
One special example of embedded systems with a particularly low energy consumption are the nodes of wireless sensor networks (WSNs) [Cal04, RSZ04]. A number of self-sufficient embedded systems with a low duty cycle of operation and a typical power consumption in the range of micro-Watts transmit data via radio communication. One approach to increase the energy efficiency of WSN nodes is to outsource routine tasks, which are usually performed by CPUs (central processing units), to autonomous CPU supplement modules with ultra-low-power reconfigurable logic. This enables the CPU to stay in inactive low-power modes for extended periods of time and thus reduces the total energy consumption.

While modern embedded microcontrollers have numerous peripherals to off-load the CPU, e.g., timers, serial interfaces, direct memory access (DMA), etc., these are manually developed and only provide limited functionality and reconfigurability. This work introduces a complete and verified methodology to develop powerful as well as flexible, yet ultra-low-power, reconfigurable CPU supplement modules.

1.1 Problem Definition

A typical WSN node comprises of a CPU (central processing unit) with memory and peripherals like a real time clock (RTC), general purpose input and output (GPIO) ports, analog to digital converters (ADCs), radio frequency (RF) transceivers, and power supply (see Fig. 1.1) [RSZ04, KJA+14]. The CPU is the main instance and controls all peripherals. It ensures the sequential execution of all tasks and reacts on external as well as internal events.

Figure 1.1: A typical WSN node consists of a microcontroller or system on chip (SoC), power supply, sensors and an antenna. The microcontroller usually contains a CPU, memory (Mem), RF interface, digital inputs and outputs (GPIO), an analog-to-digital converter (ADC), a timing unit (RTC) and power management [RSZ04, KJA+14].



The specific problem addressed in this thesis is illustrated with an exemplary sensor interface of a WSN node.

1.1.1 Example: Sensor Interface Task

One task of a WSN node is to do periodic measurements of a sensor and send a packet with the new value via the wireless interface. To save power, the node should only send a packet, if the value has changed compared to the previous value. Further, the application defines a change as a difference of more than a certain adjustable threshold.

The operation of the sensor interface is split into the following steps:

1. Switch on the sensor power supply
2. Wait until the sensor value has settled

3. Sample and convert the analog sensor voltage to a digital number
4. Switch off the sensor power
5. Compare the ADC value to the old value
6. If the values differ, send the new value via the RF interface
7. Wait until the start of the next period

This *sensor interface task* will be used as an example at several places throughout this thesis.

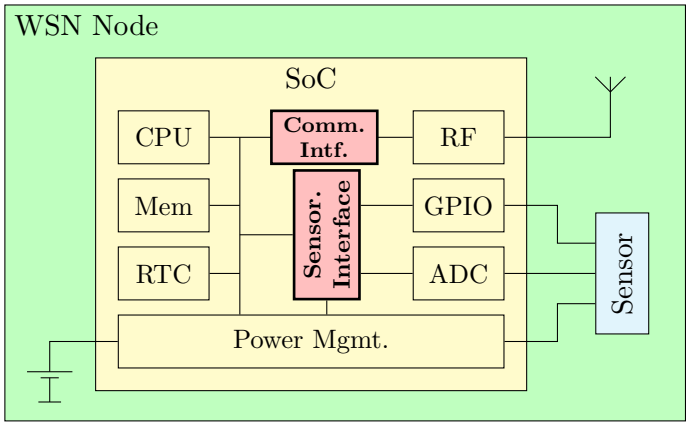
1.1.2 Power Reduction with Autonomous CPU Supplement Modules

For most embedded systems with a CPU it is set to an inactive low-power mode after all tasks are performed. The CPU is only reactivated by external and periodically timed events. The process of activation from the inactive low-power mode and vice versa itself consumes an amount of energy. This is wasted, because no tasks are performed in this time. Further, the mentioned simple tasks include waiting periods (e.g., for the response of a sensor or for the ADC to finish the conversion), which also imply wasted energy [GHDG09].

Once activated, the CPU performs simple tasks (e.g., measurement of sensor values) and tests (e.g., comparison with previous value) as well as more complex tasks (e.g., sending a network packet). Afterwards, the CPU is set to a low-power mode again.

However, many of these simple tasks can be performed autonomously by dedicated logic circuits, specifically added to a WSN system on chip (SoC) or application specific integrated circuit (ASIC) for that purpose. These CPU supplement modules off-load the CPU and only activate it for more complex tasks. Obviously the modules must consume considerably less energy than the CPU. Note that this approach is not limited to a single CPU supplement module. For each task a dedicated and optimized module can be included in the SoC. Figure 1.2 shows the exemplary WSN SoC extended by two CPU supplement modules, one for the sensor interface and one for the communication interface, e.g., handling media access control (MAC) and routing protocols.

Figure 1.2: WSN node with two CPU supplement modules, reproduced from [GHDG09, GHDG10, GHG10, GGHG11] with permission.

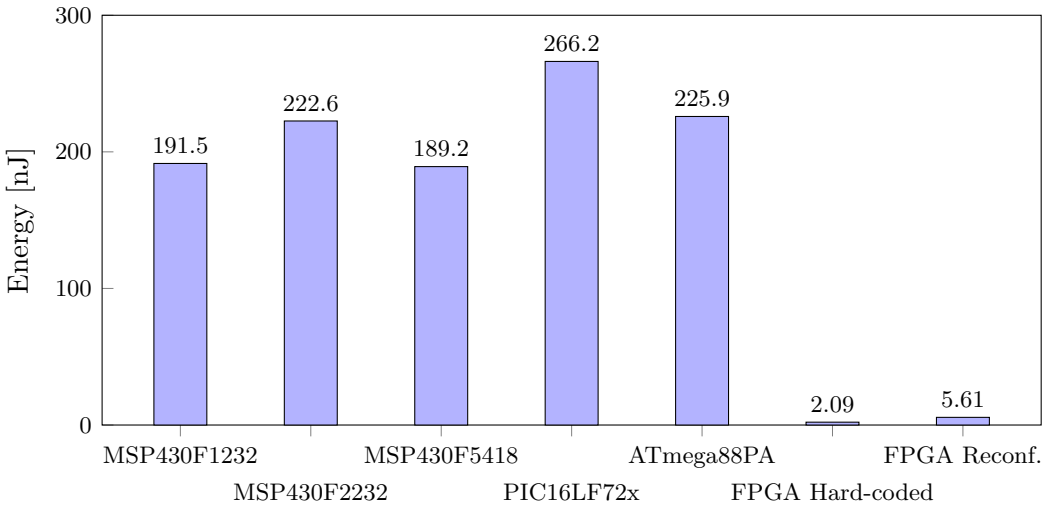


In the sense of hardware/software partitioning this approach means a shift to the hardware portion. The partitioning of the sensor interface task is done between the finding of a changed value (at the hardware side) and the assembly and transmission of the network packet (at the software side), i.e., in step 6 (see Sec. 1.1.1) The CPU supplement module will notify the CPU by an interrupt, if and only if the sensor value has changed by more than a certain threshold compared to the value when the previous notification was sent before.

In [GHDG09] the potential for energy reduction by using autonomous CPU supplement modules was examined. Figure 1.3 shows the energy consumption of one sensor measurement, i.e., one iteration of the sensor interface task as described in the previous section, performed by five different ultra-low-power MCUs (microcontroller units) and two Xilinx Virtex 4 FPGA (field programmable gate array) implementations. The sensor interface task was implemented in the C programming language for the MCUs and in VHDL (VHSIC hardware description language) for the hard-coded FPGA implementation. For the reconfigurable FPGA implementation, the FSM (finite state machine) of the hard-coded implementation was replaced by a reconfigurable implementation with a RAM (random access memory) (cf. Sec. 2.6). A more detailed discussion of these results is given in Sec. 2.1.1.

The hard-coded FPGA implementation (2.09 nJ) requires approximately 90.5 times less energy than the lowest power microcontroller MSP430F5418 (189.2 nJ). This shows the large potential for energy reduction possible using autonomous CPU supplement modules.

Figure 1.3: Energy consumption of one sensor measurement performed by ultra-low-power MCUs and FPGA implementations [GHDG09].



1.1.3 Reconfigurable CPU Supplement Modules

However, when replacing firmware by hardware, the flexibility after production of the chip must be preserved. The implementation as firmware executed by the CPU provides maximum flexibility of the WSN node, because the firmware can be easily replaced in the flash memory. The CPU supplement modules must provide comparable flexibility too. However, since the application domain is known at design time, the flexibility can be reduced, which enables even more power reduction.

Using reconfigurable CPU supplement modules allows to utilize the same WSN SoC in multiple different application scenarios. The SoC can be produced in higher quantities and sold for a wider market range, effectively distributing the non-recurring engineering (NRE) costs [Har01a]. Production defects or faults during life-time in certain reconfigurable cells can be bypassed and thus increase yield and provide fault tolerance. Reconfigurability also allows to fix bugs without requiring a redesign of the SoC [GCS+06].

Further, the external sensors used by the WSN node can be replaced by newer or different products, e.g., with lower energy consumption, with higher resolution, or if the original product is discontinued [GHDG09]. A reconfigurable communication interface, for example, allows to follow protocol updates and implement newer, more advanced protocols [GCS+06].

To implement such reconfigurable modules, first a suitable reconfigurable target architecture must be defined. Secondly, the methodology for the development of the reconfigurable modules must be defined.

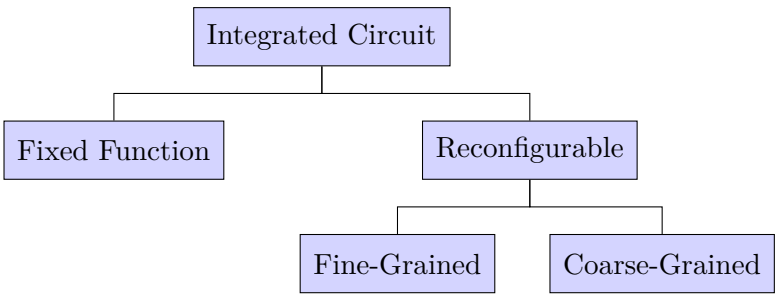
1.1.4 Reconfigurable Target Architecture

The *reconfigurable target architecture* has to be suitable for control logic (e.g., FSMs, timers) as well as arithmetic functions (e.g., calculations, comparisons). Generally, commercial (re-)configurable logic chips are available since 1970 starting with programmable logic arrays (PLAs) up to modern complex programmable logic devices (CPLDs) and FPGAs [BR96, Tua01]. Besides that, a large number of research projects introduced reconfigurable architectures optimized to a range of criteria, especially flexibility, area, speed, and power consumption [GCS+06, Rab97, SVKS01].

The proposed reconfigurable CPU supplement modules must be *included* in the custom SoC or ASIC for a tight coupling and to avoid delays and voltage matching problems, which would degrade power and area efficiency. Several commercial suppliers offer embedded FPGA (eFPGA) intellectual property (IP) cores [GZ97], which can be customized to provide exactly the required functionality and flexibility. For a detailed review see Sec. 2.3.2.

One characteristic of (re-)configurable logic circuits is the logic granularity, which specifies the size of the fundamental blocks and the amount of data handled by each block [CH02]. It ranges from fine-grained circuits like FPGAs, where logic functions for each single bit are defined, up to architectures with coarse-grained blocks like arithmetic and logical units (ALUs) or even processors (see Fig. 1.4). Coarse-grained logic is better suited for computational tasks. It requires less configuration data and less routing switches. This leads to lower area requirements and lower power consumption as well as faster loading of the configuration [HHHN00c].

Figure 1.4: Chips are either fixed function or reconfigurable, where the latter are sub-divided by the granularity of the reconfigurable logic circuits.



As stated above, the functionality of the proposed reconfigurable CPU supplement modules covers fine grained control logic as well as coarse-grained arithmetic functions. For best energy efficiency, the granularity of the underlying reconfigurable architecture should be matched to the granularity of the planned functionality. Therefore a multi-granular architecture provides the best solution [Rab97]. Additionally, since most control functions are performed as FSMs, an implementation of a single reconfigurable FSM functional unit further improves energy efficiency.

For fine-grained reconfigurable logic, a large number of simple and identical functional units can be deployed, e.g., look-up tables (LUTs) and D-type flip-flops (D-FFs) in FPGAs. In contrast, coarse-grained reconfigurable logic either needs very complex and powerful functional units, or a *heterogeneous* set of different functional units. The former alternative requires more power and area, because only a single functionality of every functional unit can be used at a time and therefore a large portion will be unused.

Desirable high flexibility defeats energy efficiency [WZG+01], therefore the heterogeneous set of such coarse-grained functional unit should be *tailored to the application domain* of the CPU supplement module. This means that instead of a unique and universal coarse-grained heterogeneous reconfigurable module template, the set of functional units for, e.g., a sensor interface module, should be optimized to this application domain, and will most likely be different from the optimized set for, e.g., a communication interface.

Therefore, for best energy efficiency, the proposed approach should use multi-granular, heterogeneous and application domain specific reconfigurable modules.

1.1.5 Design Methodology

At this point, the characteristics of the underlying architecture for the proposed reconfigurable modules are determined. As outlined above, the second step is the definition of the *methodology* for the development of such reconfigurable modules. For commercial reconfigurable chips and embedded FPGAs, the hardware structures are already defined or are created on customer request by the supplier. The developer only designs his actual application and the accompanying tools derive the configuration data for the hardware structures to implement this application.

In contrast, in this thesis a dedicated *design methodology* to develop the underlying hardware structures is required. The *starting point* of the design methodology is the demand for a reconfigurable module, given by a product development or research interest, and the specification of the required functionality. The *result* and the deliverable of the design methodology is an IP core [GZ97] of the reconfigurable module, which can be integrated in an SoC (see Fig. 1.5). Additionally, the result includes a mechanism to generate the configuration data, which implements the actual application, for this specific reconfigurable module instance.

To locate the proposed design methodology and to analyze prior research on design and deployment of application domain specific as well as universal reconfigurable logic, the following two design phases are defined (see Fig. 1.6):

Pre-Silicon Design Phase: Before production of a semiconductor, in the pre-silicon design phase, the reconfigurable logic circuit itself is developed.

Post-Silicon Design Phase: After the production of the semiconductor, in the post-silicon design phase, an actual application is developed and implemented by appropriately configuring the reconfigurable logic circuit.

Figure 1.5: Starting with a demand for a reconfigurable module, the result of the proposed design methodology is an IP core of the reconfigurable module and the configuration data to implement the actual applications.

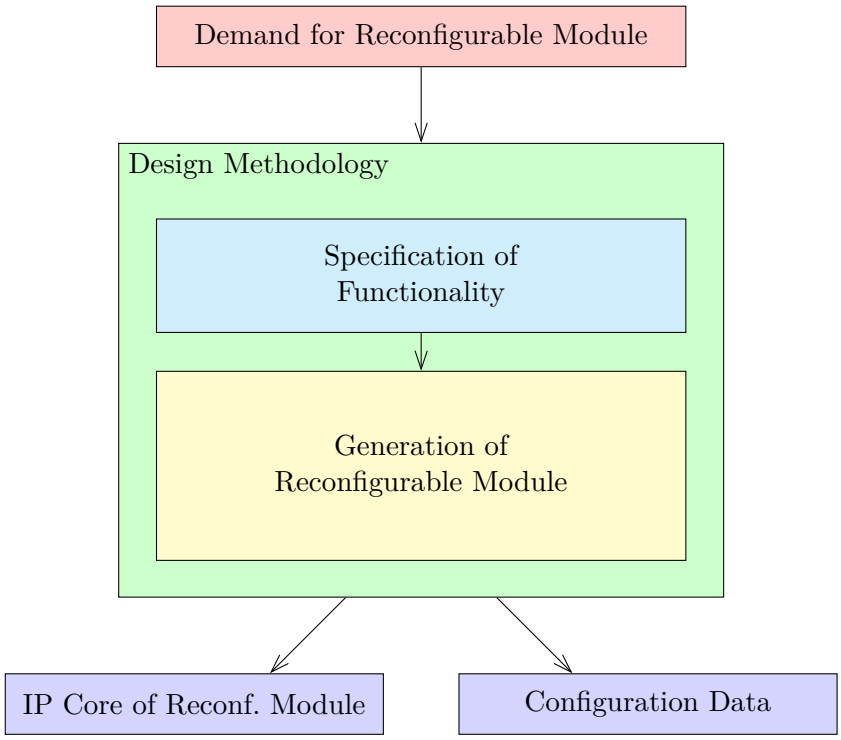
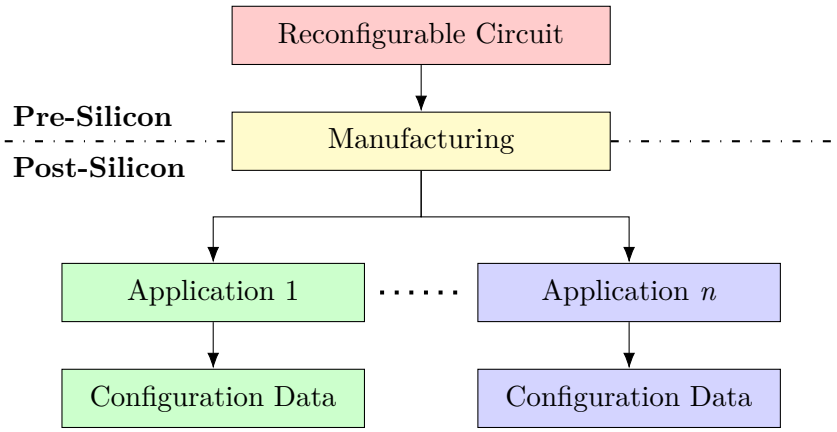


Figure 1.6: Design flow diagram for reconfigurable logic, reproduced with modifications from [GHG10, GGHG11] with permission. In the *pre-silicon design phase* a reconfigurable circuit is developed and more or less tailored to an application domain. After manufacturing, in the *post-silicon design phase*, the actual application is developed. The reconfigurable logic is configured to implement this application.



In the *pre-silicon design phase* the desired application scenario of the reconfigurable circuit has to be analyzed. From this the requirements for logic blocks and connectivity are derived and used to develop the deployed functional units and routing resources.

In commercial reconfigurable architectures like FPGAs the pre-silicon design phase is performed by the chip vendor. The internal structures, gates, connections, and switches are designed to meet

the requirements of as many applications as possible. On the other hand, the usage of FPGAs covers the *post-silicon design phase*. The designer utilizes the chips to implement his application.

The distinction between pre- and post-silicon design phase is applicable for universal reconfigurable architectures as well as for application domain specific architectures. In both phases an appropriate (explicit or implicit) design methodology and according tools are employed. Specifically for application domain specific reconfigurable architectures, the pre-silicon design methodology is often also oriented towards that application domain.

In contrast, the design methodology proposed in this thesis must support the development of reconfigurable modules for a wide range of application domains, for example sensor interfaces, communication interfaces, signal processing, serial protocol handling, etc. Therefore the design methodology must be *independent* of the application domain of the designed reconfigurable circuit.

1.1.6 Field of Application

This work assumes the development of an SoC or ASIC with the inclusion of one or more reconfigurable CPU supplement modules. The modules are limited to digital logic and do not include analog circuits. Techniques like dynamic reconfiguration or configuration scheduling are not considered [SVKS01]. The application scenario of the CPU supplement modules is primarily low-power design and does not cover, for example, high-performance reconfigurable computing [CH02]. Further, there is only small interaction between the CPU supplement module and the CPU, because the module is added to perform autonomous tasks. This also means that the reconfigurable architecture is not intended as a tightly coupled co-processor or as reconfigurable instruction set architecture (ISA) extension [TCW+05].

Low-power applications in WSN nodes are the origin and motivation of the proposed design methodology and the sensor interface task as described in Sec. 1.1.1 is used throughout this thesis as an example. However, the proposed design methodology for the design of reconfigurable CPU supplement modules is neither limited to this example nor to WSN applications. It can be employed to develop reconfigurable modules for a wide range of application scenarios, e.g., WSN communication interfaces, signal processing, serial protocol handling, etc.

1.1.7 Requirements

From the aforementioned application scenario and use cases, the following requirements for the design methodology are derived.

- The design methodology must be independent of the application domain of the reconfigurable CPU supplement module, i.e., the design methodology must not be specialized for a certain application domain.
- The defining property of reconfigurable circuits is their flexibility to implement various applications. For this work, an appropriate way to specify this flexibility is required. Existing fine-grained homogeneous circuits like FPGAs are specified in terms of available resources (e.g., number of LUTs, pins, memory). In this thesis, the reconfigurable CPU supplement modules should be optimized to a given application domain, therefore the design methodology must provide a way of specification in terms of possibilities and accomplishable functionality.

- One requirement for the proposed reconfigurable CPU supplement modules is to provide flexibility beyond the required functionality and perform new functionality not anticipated during development. Therefore, the design methodology must allow to include that additional flexibility, and further a way for its specification.
- For best energy efficiency, the design methodology must support a multi-granular reconfigurable architecture.
- In chip design automation, one main principle is the verification of the results to eliminate expensive revisions of the developed chips. The proposed methodology must provide means to perform verification and to ensure correct results, i.e., that the generated reconfigurable CPU supplement module fully complies to and fulfills the specified functionality.
- To enable high productivity, the design methodology must be implemented as a fully automated design flow which only requires the designer's manual work to enter information which specifies aspects of the designed reconfigurable module or where the specific human intelligence and experience are necessary.
- Further, the design flow must offer user-friendly interfaces. It should especially be easy to learn and not require special skills.
- The design methodology must have short iteration times between manual interactions, i.e., the automated procedures must not lead to long lasting interruptions of the designer's working process.
- The methodology and the resulting reconfigurable modules must be independent of the semiconductor process.
- The methodology must be compatible to commercial ASIC tools as well as custom in-house design flows.

1.1.8 Scientific Basis

Several similar approaches as proposed in this thesis have been investigated. Some devices of the commercial Gecko MCUs by Silicon Laboratories Inc. include a "Low-Energy Sensor Interface" (LESENSE) peripheral [Lar11, Ene14], optimized for capacitive and inductive touch sensors. The peripheral autonomously performs sensor excitation and measurement to off-load the CPU. This module is hard-wired and only provides limited configurability. However, it shows that the industry considers CPU supplement modules to handle external sensors an important approach. The XiSystem SoC [LCB+06] includes two different CPU supplement modules, one with coarse-grained logic for pipelined data processing and one fine-grained FPGA for control-dominated applications. It does not allow to interact between data processing and bit-level functions. Both CPU supplement modules were developed manually to provide universal functionality and the user only handles the post-silicon design phase.

For the Nymble System [HLOK13], the user marks regions of algorithms in his C application code. These are automatically extracted and converted to Verilog code, which implements a hardware accelerator for the marked region. This approach is intended for high-performance data processing. It generates a non-reconfigurable CPU supplement module and does not allow to specify

cycle accurate timing. [Hen99] introduced a methodology for the optimization of the energy consumption of a whole SoC with a CPU, caches and accelerators. Although this approach focuses on energy consumption, it is also limited to data processing and non-reconfigurable accelerators.

An automated methodology for the creation of application domain specific reconfigurable logic was published by [CH08]. From a set of applications, the required resources are determined. Then a template for the reconfigurable architecture is optimized using an iterative process. Each application is mapped and the quality of the mapping is evaluated and used for improvements of the architecture. However, this approach is limited to coarse-grained data processing applications. [KMM14] use a generic fine-grained FPGA fabric as template to map a set of applications. The place and route algorithms seek to match instances of the applications and map these to the same functional units. After an iterative optimization, all unused resources are removed and the remaining reconfigurable resources are replaced by simplified logic to switch between the applications. Therefore this approach does not provide flexibility for new applications.

The examples show that previous work concentrates on data processing or does not provide enough flexibility. To the best of the authors knowledge, no design methodology which optimizes reconfigurable modules to a given application domain, and which support control-dominated tasks *and* data processing for ultra-low-power applications have been shown.

1.1.9 Specific Problem Definition

To reduce the energy consumption of WSN nodes, tasks performed by the CPU are reassigned to autonomous CPU supplement modules. These modules have to be reconfigurable to be used in different application scenarios. The specific problem addressed by this thesis is the appropriate methodology for the development of such reconfigurable CPU supplement modules.

1.2 Hypotheses

Based on the existing research and specific problem definition for the proposed design methodology, the following qualitative and quantitative hypotheses are postulated.

Hypothesis 1: Feasibility of Design Methodology

The proposed design methodology leads to a working reconfigurable CPU supplement module which provides the full specified functionality plus additional flexibility to implement new functionality. This hypothesis is evaluated by developing a reconfigurable module for a WSN node sensor interface with the proposed design methodology using a set of target applications. The resulting IP core is integrated in an SoC and manufactured. The hypothesis is provisionally accepted, if the reconfigurable CPU supplement module IP core in the SoC correctly performs the functionality of the target applications, and if it correctly performs new and different applications.

Hypothesis 2: Energy Reduction

The proposed design methodology leads to a reconfigurable CPU supplement module, which energy consumption is lower than the energy consumption of a CPU, both performing the same task and implemented in the same semiconductor technology. To evaluate this hypothesis, the sensor interface task is once performed by the reconfigurable module, and once by the CPU, both in the manufactured SoC. The hypothesis is provisionally accepted, if the energy consumption of the reconfigurable module is lower than the energy consumption of the CPU.

Hypothesis 3: Area Reduction

The proposed design methodology leads to a reconfigurable CPU supplement module, which chip area is smaller than the chip area of the parallel non-reconfigurable implementation using a traditional ASIC design flow. Further, the proposed design methodology leads to a reconfigurable module, which chip area is smaller than the chip area of an (embedded) FPGA with the full functionality. To evaluate the first sub-hypothesis, all target applications used to specify the functionality of the reconfigurable modules are implemented concurrently, together with the required multiplexers, in the same process technology as the SoC. The hypothesis is provisionally accepted, if the reconfigurable CPU supplement module requires less chip area than the implementation of all target applications in parallel. To evaluate the second sub-hypothesis, each target application is individually implemented using an (embedded) FPGA. The hypothesis is provisionally accepted, if the reconfigurable module requires less chip area than the chip area of an (embedded) FPGA, which provides exactly the amount of resources as required by the target application with the highest resource utilization.

Hypothesis 4: Configuration Data Reduction

The proposed design methodology leads to a reconfigurable CPU supplement module, which requires less configuration data than an (embedded) FPGA with the same functionality. To evaluate this hypothesis, each target application is individually implemented using an (embedded) FPGA. The hypothesis is provisionally accepted, if the reconfigurable module requires less configuration data than an (embedded) FPGA, which provides exactly the amount of resources as required by the target application with the highest resource utilization.

1.3 Goals and Tasks

The task addressed in this thesis is the development of a design methodology for reconfigurable CPU supplement modules. All existing approaches are targeted to computing and data processing and do not provide means for control-dominated tasks. The main challenge of this work is to extract and use applicable concepts from existing approaches and to introduce new concepts for control logic, and finally to integrate these to a homogeneous methodology.

The scientific method for the development of the proposed design methodology and for the validation of the hypotheses is divided in the following tasks:

1. Scientific development of the proposed *design methodology* for reconfigurable modules, based on prior research, and evaluation and comparison to alternative solutions.

2. Implementation of the design methodology as an automated *design flow*.
3. Utilization of the design flow to develop a reconfigurable CPU supplement module, integration into an SoC, and production of the SoC.
4. Test, characterization, and measurement of the SoC.
5. Evaluation of the results and validation of the hypotheses.

1.4 Contributions

This thesis extends the state-of-the-art by the following *core contributions*:

- **Design Methodology.** The main contribution of this thesis is a design methodology to develop reconfigurable CPU supplement modules which are optimized for a given application domain (Ch. 3). To the best of the authors knowledge, the introduced design methodology is the first which is universal and independent of the application domain, especially by the concurrent support of both, control-dominated tasks and data processing.

  The design methodology was used to implement a reconfigurable module for the sensor interface task in a *WSN SoC*. This application domain includes control-dominated tasks as well as data processing. The experimental results show that the design methodology leads to a functioning reconfigurable module which correctly implements different variants of the sensor interface task (Sec. 5.1). Additionally, enough flexibility is provided to implement new functionality (Sec. 5.6.3). The reconfigurable module requires less power compared to a CPU to perform the sensor interface task (Sec. 5.3). It requires less chip area (Sec. 5.4) and less configuration data (Sec. 5.5) than an embedded FPGA.

  The design methodology was introduced in section 3 of the book chapter [GW14]. In [GHG10] and [GGHG11] the foundations and concepts for the required tools were published (cf. "Early Approach" in Sec. 3.1.5). The full details and the evaluation are published with this thesis.

- **Transition-Based Reconfigurable FSM (TR-FSM).** The second core contribution is a novel architecture for reconfigurable FSMs (Sec. 3.7).

  The TR-FSM was included in the reconfigurable module of the WSN SoC as well as in an earlier manually developed reconfigurable module test chip (Sec. 5.2). The TR-FSM architecture reduces the power consumption, chip area, configuration data, and propagation delay time compared to other reconfigurable architectures for FSMs (Sec. 5.8), and allows straight-forward mapping algorithms [GGHG11].

  The TR-FSM architecture was published in the conference paper [GDHG10]. The straight-forward algorithm to map FSMs to the TR-FSM architecture was published in [GGHG11]. The journal paper [GDHG11] extends [GDHG10] with more details and an evaluation of the TR-FSM.

Besides these core contributions, this thesis provides the following *supplemental contributions*:

- **Design Flow.** The design methodology was implemented as a fully automated design flow. This implements the scientific concept as a practical integrated development environment (Ch. 4) to enable the user-friendly utilization of the design methodology.

  The design flow was used to implement the reconfigurable module of the WSN SoC. The resulting test chip validates the feasibility of the design flow (Sec. 5.1). The design flow is fully automated to provide high productivity (Sec. 5.6.2).

  The design flow is published with this thesis.

- **Low-Power Technique.** One dedicated application scenario of reconfigurable modules developed with the design methodology and the design flow is the use as a novel low-power technique for embedded systems at the architectural level (cf. Sec. 1.1.2). Together with existing low-power techniques, it provides further reduction of the power consumption. This is beneficial in always-on applications like WSNs.

  The WSN SoC utilizes this low-power technique with the included reconfigurable module. This enables the reduction of the energy consumption per sensor measurement by a factor of nearly 180 (Sec. 5.3).

  This low-power technique was proposed in the conference paper [GHDG09] and validated in the journal paper [GGHG11] and in this thesis.

During the development of the design methodology, scientific publications on individual aspects were contributed:

- The approach to reduce the energy consumption with a reconfigurable CPU supplement module was evaluated in the conference paper [GHDG09] (cf. Secs. 1.1.2 and 5.3.4).

  [GHDG09] **Johann Glaser**, Jan Haase, Markus Damm, and Christoph Grimm. Investigating Power-Reduction for a Reconfigurable Sensor Interface. In *Proceedings of Austrochip 2009*, Graz, Austria, 7. October 2009.

- In the conference paper [GDHG10] the TR-FSM as a reconfigurable architecture for FSMs was introduced (cf. Sec. 3.7). The formal description of FSMs, the name, and the analysis were contributed by Markus Damm, who also attended the conference.

  [GDHG10] **Johann Glaser**, Markus Damm, Jan Haase, and Christoph Grimm. A Dedicated Reconfigurable Architecture for Finite State Machines. In *Reconfigurable Computing: Architectures, Tools and Applications, 6th International Symposium, ARC 2010*, volume LNCS 5992 of *Lecture Notes on Computer Science*, pages 122–133, Bangkok, Thailand, March 2010. Springer Berlin Heidelberg.

- The initial architecture for reconfigurable CPU supplement modules was proposed in the conference paper [GHDG10]. This also used the TR-FSM as a building block for control-dominated tasks.

  [GHDG10] **Johann Glaser**, Jan Haase, Markus Damm, and Chritoph Grimm. A Novel Reconfigurable Architecture for Wireless Sensor Networks. In *Tagungsband zur Informationstagung Mikroelektronik 10*, pages 284–288, Vienna, Austria, 7.–8. April 2010. OVE.

- The foundations of the design methodology for reconfigurable modules and the concept for the required tools were published in the conference paper [GHG10] (cf. “Early Approach” in Sec. 3.1.5).

[GHG10] **Johann Glaser**, Jan Haase, and Christoph Grimm. Designing a Reconfigurable Architecture for Ultra-Low Power Wireless Sensors. In Zabih Ghassemlooy and Wai Pang Ng, editors, *Proceedings of the Seventh IEEE, IET International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 343–347, Northumbria University, Newcastle upon Tyne, United Kingdom, 21.–23. July 2010.

- With this early approach, a reconfigurable module was manually developed and produced as a test chip. Additionally, the TR-FSM was extended, implemented as a VHDL module, and integrated in the reconfigurable module (cf. Sec. 5.2). The journal paper [GDHG11] extends [GDHG10] with more details on the TR-FSM and its characterization according to chip area, delay, and power consumption.

  [GDHG11] **Johann Glaser**, Markus Damm, Jan Haase, and Christoph Grimm. TR-FSM: Transition-based Reconfigurable Finite State Machine. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(3):23:1–23:14, August 2011.

- More details on the design methodology, the tools, and on the manually developed reconfigurable module were published in the journal paper [GGHG11], which is an extension of [GHG10]. The FPGA design used in the evaluation platform to stimulate the test chip was contributed by Klaus Gravogl.

  [GGHG11] **Johann Glaser**, Klaus Gravogl, Jan Haase, and Christoph Grimm. A Reconfigurable Architecture for Ultra-Low Power Wireless Sensors. *The Mediterranean Journal of Electronics and Communications (MEDJEC)*, 7(3):255–266, 2011.

- In the conference paper [WGS+12] an algorithm for the "Merge" step was published. Clifford Wolf developed and implemented the optimization algorithm and the tool InterSynth for the generation of the interconnect. The author of this thesis motivated this tool and contributed the introduction, reconfigurable hardware structures, related work for the interconnect, and the design flow integration, and attended the conference.

  [WGS+12] Clifford Wolf, **Johann Glaser**, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-Driven Interconnect Synthesis for Heterogeneous Coarse-Grain Reconfigurable Logic. In *Forum on Specification and Design Languages (FDL)*, pages 194–201, Vienna, Austria, 18.–20. September 2012.

- Clifford Wolf also implemented the free and open-source Verilog synthesis tool Yosys, which was published in the conference paper [WG13]. The contribution of the author of this thesis comprises the introduction, the phrasing, and the demonstration at the conference.

  [WG13] Clifford Wolf and **Johann Glaser**. Yosys – A Free Verilog Synthesis Suite. In *Austrochip Workshop on Microelectronics*, pages 47–52, Linz, Austria, October 2013.

- With the book chapter [GW14] the conference paper [WGS+12] was extended with the introduction of the automated design methodology. More details on InterSynth and Yosys were contributed by Clifford Wolf.

  [GW14] **Johann Glaser** and Clifford Wolf. Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In Jan Haase, editor, *Models, Methods, and Tools for Complex Chip Design*, volume 265 of *Lecture Notes in Electrical Engineering*, pages 201–221. Springer International Publishing, 2014.

Additional to the scientific publications, the author of this thesis supervised two master theses:

- While the energy consumption of the MCUs in [GHDG09] were determined analytically, Georg Blemenschitz measured the power consumption of MCUs and the manually designed reconfigurable module (cf. Sec. 5.3.4) [Ble15]. He used a more complex task and contributed the firmware of the MCUs, the automation of the measurement, the analysis of the results (cf. Sec. 5.3.2), and numerous extensions to the evaluation platform.

  [Ble15] Georg Blemenschitz. Evaluierung der Reduktion der Leistungsaufnahme durch eine rekonfigurierbare Architektur. Master's thesis, Technische Universität Wien, Institut für Computertechnik, 2015. (in preparation).

- Martin Schmölzer evaluated the semi-automated design methodology using only the tools Yosys, TrfsmGen, and InterSynth, before the design flow was implemented [Sch14]. He additionally contributed an extension to the TR-FSM.

  [Sch14] Martin Schmölzer. Design of a Flexible Data Path for Heterogeneous Coarse-Grain Reconfigurable Logic Circuits. Diploma thesis, Vienna University of Technology, 2014.

Further, the author of this thesis supervised five student projects which developed the serial bus masters (SPI by Georg Blemenschitz, I^2C by Mario Faschang, 1-Wire by Peter Hanger, PWM, SENT, and SPC by Sebastian Plunger) and the UART module (Armin Faltinger), which were used in the manually developed reconfigurable module and the WSN SoC (cf. Sec. 5.2).

1.5 Challenges

To reach the goals as defined above, a number of challenging tasks have to be accomplished.

- To specify the functionality of a reconfigurable module, the challenge is to find a suitable format of specification.
- The scientifically developed design methodology has to be implemented as an actual design flow. This has to handle a large amount of information in numerous different kinds of representation. The challenge is, that these representations must be suitable for human developers as well as automated programs.
- To automate the generation of the reconfigurable module, the challenging task to find the proper algorithms and the proper electronic design automation tools has to be accomplished.
- To provide a homogeneous design flow, the system integration task of combining a number of different tools from different vendors, handling multiple different file formats, has to be performed.
- The requirement for high productivity and a simple to learn design flow poses the challenging task to concurrently fulfill the contradicting claim for a powerful tool which allows customization to all of the designer's requirements.
- The work on the implementation of the design flow requires a broad range of knowledge and skills, especially from the fields of computer science, computer engineering, and electrical engineering.

1.6 Organization of Thesis

The remainder of this thesis is structured as follows: In the next chapter 2 the state-of-the-art is evaluated and the open topics are defined. Building on this basis, in chapter 3 the new approach for the design methodology is developed. Following, in chapter 4 this approach is realized as a complete design flow for reconfigurable modules. This design flow was used to develop an SoC to demonstrate the methodology and to evaluate the aforementioned hypotheses and requirements (chapter 5). Finally, in chapter 6 the work and the results are summarized, the scientific contribution is stated, including a discussion of its implications, potential and limitations, and an outlook for future work is given.

2

State of the Art

As outlined in chapter 1, the motivation for this work is to reduce the power consumption of embedded systems. This is achieved by supplementing the CPU with dedicated and reconfigurable hardware modules. These take over a subset of the tasks from the CPU, which in turn can stay for longer periods in an inactive low-power mode. To locate this approach within the large field of low-power techniques for embedded systems, in Sec. 2.1 an overview of state-of-the-art techniques at all levels of design abstraction is given.

To keep the power consumption of these reconfigurable CPU supplement modules as low as possible, they are optimized for a given application domain. The development of application domain specific reconfigurable CPU supplement modules involves two distinct areas:

1. application specific dedicated hardware to assist the CPU, and
2. reconfigurable logic.

A large amount of research and development has been conducted in both areas individually, but also in combination. Relevant contributions will be reviewed in this chapter with a special focus on CPU supplement modules, low-power approaches, coarse- and multi-granular logic and applications in WSNs.

It is important to note that there are two levels of abstraction for being application (domain) specific or application (domain) independent: First, the hardware architecture of a (reconfigurable) CPU supplement module can be independent or specific to an application (domain). Secondly, the methodology to design that architecture can depend on the application (domain) or be universal and independent of the application (domain). Further, a reconfigurable CPU supplement module can be specific for an application domain, while a hard-wired non-reconfigurable CPU supplement module can only be specific to a single application, hence the parentheses in the previous sentences.

To evaluate and compare the reviewed approaches, a classification scheme for common properties is derived in Sec. 2.2 and will be used in the following sections.

As first aspect of the proposed approach, in Sec. 2.3 reconfigurable CPU supplement modules are reviewed, which were manually developed for a broad range of applications. These provide general insight on reconfigurable logic and CPU supplement modules.

In this thesis, the CPU supplement modules are tailored to a specific application domain for even lower power consumption. Additionally, the design of these should have a high degree of automation. The automated design of application specific but non-reconfigurable dedicated logic is investigated in the area of hardware/software co-design, which will be reviewed in Sec. 2.4.

The application domain specific CPU supplement modules developed in the field of hardware/-software co-design implement a fixed functionality. In Sec. 2.5, research on the fusion of the two topics of application domain specific CPU supplement modules and reconfigurable logic are reviewed in particular detail. This is followed by a classification of the reviewed approaches.

In Sec. 2.6 reconfigurable architectures for FSMs are discussed, which are an important element in logic design. The chapter closes with conclusions on the reviewed works and their assessment as a basis for this thesis (Sec. 2.7).

2.1 Low-Power Embedded Systems

In nearly all embedded systems, the reduction of the power consumption is of major concern. The reasons for low-power design can be categorized in three groups:

1. The main goal of *high-performance (embedded) systems* like desktop computers, servers, and communication equipment is performance. These systems employ high-performance semiconductors as CPUs and large FPGAs in which high power density and therefore heat dissipation occurs. The dissipated power has to be conducted from the gates to the case and further to the heatsink to limit the die temperature. Additionally, the high supply current must be conducted to every point on the chip with adequate stability and signal integrity [RP96]. [FH05] coined the term “power wall” for the limit of power P and clock period t: $P \cdot t^3 = \text{const}$. This means that twice the clock frequency leads to eight times the power dissipation. Low-power techniques are utilized to address these problems.

2. Users demand for high performance of (rechargeable) *battery powered embedded systems* like smart phones, portable media players, and tablets [Mac04, RP96]. For example, the smart phone Samsung Galaxy S5 contains a quad core processor with a clock frequency of up to 2.5 GHz.[1] The rationale of low-power techniques for such devices is to extend the battery life.

3. Embedded systems supplied from disposable (non-rechargeable) batteries (primary cells) and/or energy harvesting have to be *extreme low power systems*. These systems stay in an inactive sleep or standby state most of the time. [Ozk12] further divides this group: Firstly, devices with batteries which last for weeks to months. These devices only have little processing tasks, for example remote controls, portable test equipment, wireless keyboards, and blood-glucose meters. Secondly, systems which are "barely alive" and stay without activity for very long periods. The battery lasts for months to years, for example remote monitoring, utility metering, data loggers, and WSN nodes.

The scope of this thesis is the third case, i.e., systems which have batteries with limited capacity but should last for months to years and therefore stay in an inactive mode for a large fraction of time.

In the next section, sources of power consumption are discussed. For these, a large amount of low-power techniques has been developed which will be summarized in Sec. 2.1.2.

2.1.1 Power Consumption of Embedded Systems

The main power dissipating components of embedded system are semiconductors. The power consumption of CMOS (complementary metal-oxide-semiconductor) chips [RCN03, Tew10, BMM01] is the sum

$$P = P_{\text{leakage}} + P_{\text{switching}} + P_{\text{short-circuit}}. \tag{2.1}$$

$P_{\textbf{leakage}}$ results from non-ideal properties of insulation material. With successive process shrink this component is increasing. Additionally transistors in the Off-state show residual currents.

$P_{\textbf{switching}} = \frac{CV^2f}{2}$ reflects the charging and discharging of gate and wire capacitance, with C being the capacitance, V the supply voltage and f the switching frequency. More precisely, each net has an individual switching frequency and capacitance, so the equation is a simplified depiction.

$P_{\textbf{short-circuit}}$ accounts for the short time during switching, when both, the top PMOS and bottom NMOS transistors are conducting and thus shorting the supply rails. This term again is proportional to the switching frequency as well as the slew rate of the signals, i.e., higher slew-rate reduces the time and therefore energy of short-circuit conduction.

The leakage power is also referred to as static power, whereas the switching power and the short-circuit power are conjointly referred to as dynamic power.

The main type of embedded systems related to this thesis are WSNs. One typical task of WSN nodes is to periodically perform sensor measurements as described in Sec. 1.1.1. In [GHDG09]

[1] http://www.samsung.com/us/mobile/cell-phones/SM-S902LZKATFN#key-specs [2015-08-20]

the power consumption of a firmware implementation executed by a CPU was compared to the power consumption of an implementation using dedicated reconfigurable hardware.

For the firmware implementation five ultra-low-power MCUs were investigated. The energy consumption per sensor measurement cycle is reproduced in the first five bars of Fig. 2.1 and in the first five columns of Tab. 2.1.[2] The values were determined by implementing the sensor interface task as a C function. This was compiled and the resulting assembler instructions were evaluated to calculate the runtime (yellow) at an operating frequency of 4 MHz. Interrupt overhead (wake-up from low-power mode, interrupt latency, context save, context restore, and return from interrupt) was calculated from datasheet specifications. The PIC16(L)F72x does not have individual interrupt vectors, therefore additional overhead to determine the source of the interrupt was considered. Waiting times for the duration of sensor settle (red, 10 µs) and the ADC conversion (violet, all MSP430: 4.5 µs, PIC16LF72x: 10 µs, ATmega88PA: 25 µs) were assumed constant. The runtimes were multiplied by the supply voltage of 3 V and the current according to the datasheets. To allow a comparison to the FPGA implementations, the current consumption values only include the CPU and memories of the MCUs but exclude the peripherals, especially the ADC. Therefore, all individual values in Fig. 2.1 and Tab. 2.1 comprise the energy consumption to perform one sensor measurement exclusively of the CPU and memories. For example, the waiting times for the sensor settle and ADC conversion do not include the energy consumed by the sensor or the ADC.

For the implementation using dedicated reconfigurable hardware, a Xilinx Virtex 4 FX XC4VFX20-FF672 FPGA on the ML405 evaluation platform [Xil08] was used. The power supply circuitry includes shunt resistors. The voltage drop was measured to determine the current consumption of the 1.2 V core supply voltage of the FPGA. For the evaluation of the energy per measurement, the difference between the active and inactive design was used. As mentioned in Sec. 1.1.2, the hard-coded FPGA implementation was developed in VHDL. For the reconfigurable FPGA implementation the sensor interface task was realized with a reconfigurable FSM implemented using a BlockRAM (cf. Sec. 2.6). Note that in both cases the integrated PowerPC CPU was not used.

The detailed energy consumption values show the total energy split in the individual causes. The largest contributions are the actual sensor interface function, waiting for the sensor to settle, and waiting for the ADC conversion. This shows that a large part of the energy is consumed for waiting. On the other hand, the the FPGA implementations directly implement the functionality without interrupt overhead and do not cause additional energy consumption during the waiting periods. These differences and the lower energy to perform the actual function enable the reduction of the total energy by a factor of approximately 90.5 compared to the lowest power MCU.

To reduce the total power consumption of semiconductors, primarily the dynamic power has to be reduced [DM95, Mac04]. As shown above, the dynamic power depends on the supply voltage and the frequency. [GHDG09] investigated the potential for energy reduction at decreased frequencies (see Fig. 2.2, please note the irregular frequency values). The results show that a reduction of the clock frequency below 1.0 MHz does not have a marked impact. The reason is that the actual sensor interface function (yellow) requires a proportionally longer execution time, which results in a constant energy consumption. Above 1.0 MHz, the energy per measurement increases because the waiting periods have a constant time and the power consumption linearly depends on the frequency, hence the energy increases linearly too.

[2] The sum values were already shown in Fig. 1.3.

Figure 2.1: Energy consumption of one sensor measurement performed by different architectures, reproduced with the original raw values from [GHDG09].

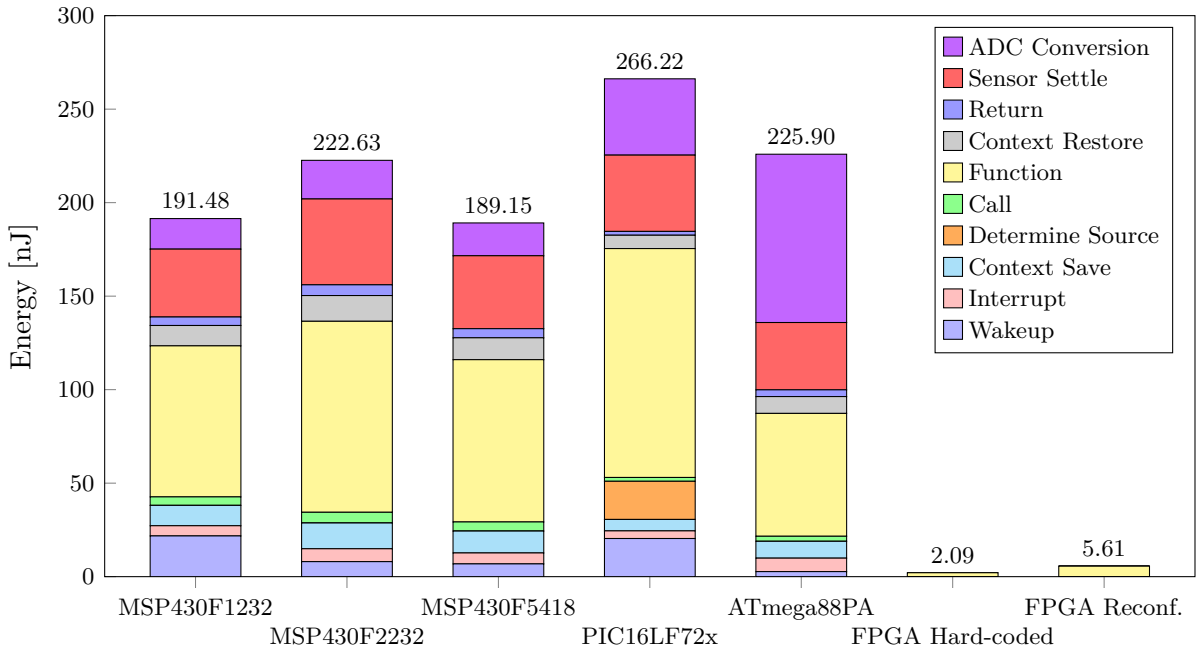
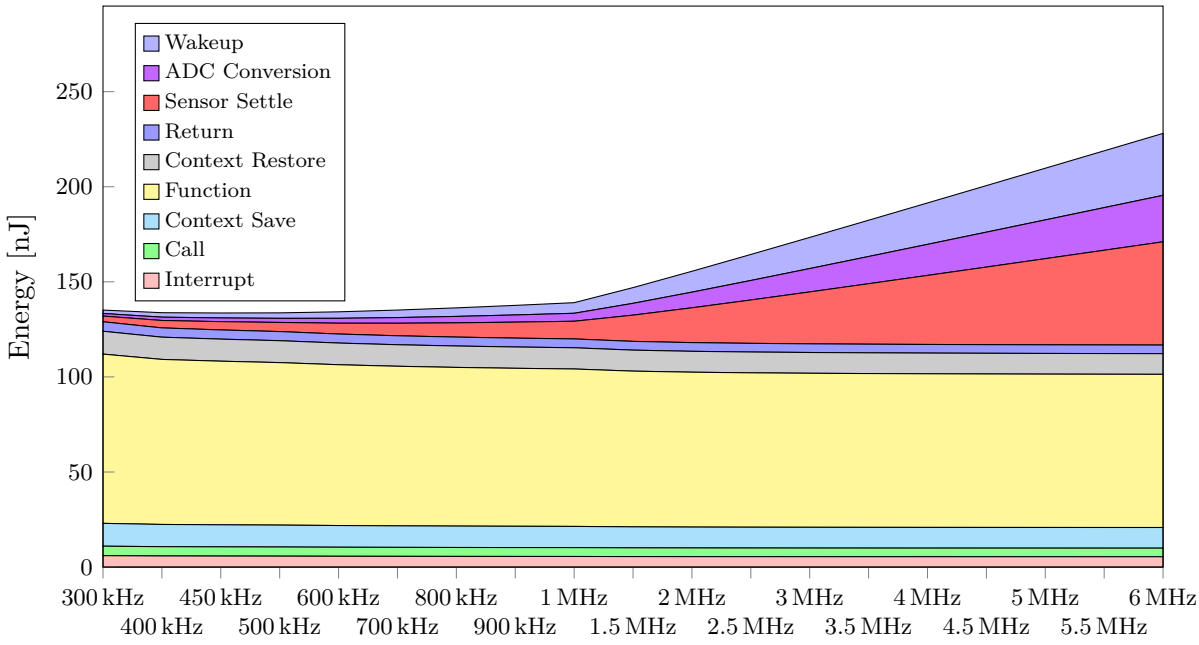


Table 2.1: Energy consumption in nJ of one sensor measurement performed by different architectures, original raw values from [GHDG09].

	MSP430 F1232	MSP430 F2232	MSP430 F5418	PIC16 LF72x	ATmega 88PA	FPGA Hard-c.	FPGA Reconf.
Wakeup	21.78	8.03	6.83	20.40	2.70	–	–
Interrupt	5.45	6.89	5.85	4.08	7.20	–	–
Context Save	10.89	13.77	11.70	6.12	9.00	–	–
Determine Source	–	–	–	20.40	–	–	–
Call	4.54	5.74	4.88	2.04	2.70	–	–
Function	80.77	102.13	86.78	122.40	65.70	2.09	5.61
Context Restore	10.89	13.77	11.70	7.14	9.00	–	–
Return	4.54	5.74	4.88	2.04	3.60	–	–
Sensor Settle	36.30	45.90	39.00	40.80	36.00	–	–
ADC Conversion	16.34	20.67	17.55	40.80	90.00	–	–
Sum	191.48	222.63	189.15	266.22	225.90	2.09	5.61

Figure 2.2: Frequency dependence of the energy consumption of one sensor measurement performed with the MSP430F1232 (please note the irregular frequency values), reproduced with the original raw values from [GHDG09].



2.1.2 Low-Power Techniques

In the previous section, the total energy consumption of a semiconductor was reduced by decreasing the operating frequency. This technique is only beneficial to a certain degree. Besides that, a large number of other power reduction techniques for semiconductors and for whole embedded systems were proposed. The interested reader is directed to the reviews, surveys, editorial books, and overview papers [Tew10, Mac04, GM02, BMM01, MPS98, RPL96, DM95, CSB92] to find more information on low-power techniques. In this section, a coarse overview of low-power techniques mentioned in these works is given. Additional references for specific topics are stated below.

Low-power techniques are applied at all different levels of abstraction of embedded systems. The above mentioned works use similar but slightly different categorizations for these techniques. For this summary a unified categorization is derived from these works together with [Nie98, Mar03, Hea03].

System Level. The system level defines a whole system or application consisting of multiple interacting embedded systems and the environment, e.g., a WSN. The main task of a WSN is to retrieve and communicate environmental information and (if actors are involved) perform actions depending on that information. [Sar12] links energy and information and formulates ten information-based principles for ultra-low-power design. One important principle in terms of WSN is to balance the cost of computation and communication, because computation generally reduces the amount of information to be transmitted. This is addressed by the topic of data fusion (also called sensor fusion and information fusion) [LWj11].

Network Level. In WSNs, the RF communication is a major contribution to energy consumption [Roe04, Mah04]. Therefore a large number of protocols optimized for low power consumption,

but also for low latency, high throughput, high precision synchronization, etc., were developed. At the MAC protocol layer the access to the shared medium (the RF channel) is organized. The different protocols optimize the timing, synchronization and duration of listen and transmission periods as well as the avoidance or handling of collisions to reduce the total energy consumption of a single node or of the whole network [GL00, HXS+13].

For example, the CSMA-MPS (carrier sense multiple access with minimum preamble sampling) protocol [MB04] stores information of the time window the communication partner will listen for communication attempts (plus a correction factor for the drift of the crystals). Further, it requires a radio transceiver with high bitrate and fast turnaround times between transmit and receive operation. This is used to quickly alternate sending the data and listening for the acknowledgment of the receiver. This avoids the need for a network wide synchronization or a global beacon signal. While other protocols require a wake-up packet which is then followed by a data packet, CSMA-MPS merges these tasks into one packet and thus further reduces the energy consumption.

At the routing layer the communication across nodes with no direct connection is managed. This also includes dynamic routes which, e.g., depend on the current battery level [EAA13]. Instead of choosing the route which consumes the lowest total power (e.g., by passing the least number of intermediate nodes), [CT00] found that it is beneficial to route the data along nodes with the highest battery reserves, e.g., [MMR06].

Operating System Level. In larger embedded systems like smart phones and laptops, but also in WSN nodes, an operating system is employed, such as Contiki [DGV04] or TinyOS [LMP+05]. The operating system is responsible to manage the resources shared among the tasks. The CPU execution time is also a shared resource and is managed by the task scheduler. For power reduction, energy aware task schedulers consider the availability and power consumption of resources to determine the start time of the tasks.

Dynamic power management switches between different power states of the system (e.g., active, idle, standby) to activate the required resources. For example, ACPI (Advanced Configuration and Power Interface) [Uni14] defines one active (S0) and five sleep states (S1–S5). Generally, dynamic power management utilizes facilities provided at other levels like adaptive voltage and frequency scaling implemented at the technology level. Operating system level energy optimization is more efficient when utilizing the knowledge of the individual tasks' demand for performance.

Algorithm and Software Level. Energy optimization at the algorithm and software level comprises techniques and knowledge applied during software design. This includes selecting algorithms with lower power requirements, usually connected to reduced complexity. Several algorithms can utilize parallel processing offered by the hardware. Additionally, the application itself can perform power management using more complex information than is available to the operating system

Compiler Level. Many techniques can be applied directly by the compiler. Since the execution and storage of software requires energy, optimizing the software for size also optimizes for speed and power consumption. Energy-aware compilers utilize the knowledge of the underlying CPU architecture (e.g., register set, pipelines, instruction set architecture (ISA), caches). Highly optimizing compilers employ knowledge on the different power consumption of the individual instructions and register allocation. Generally, utilizing the principle of locality (e.g., store data in registers instead of memory) improves the runtime and the power consumption. Additionally

CPU-independent code transformations intended to reduce the runtime (e.g., common subexpression, loop unrolling, see [BFS04]) improve the power consumption.

Architecture Level. Low-power techniques at the architecture level optimize the architecture of the hardware for energy reduction. A whole WSN node usually consists of a small PCB (printed circuit board) with a number of components, of which some might have different supply voltage requirements. This wastes energy in voltage regulators and voltage level translators. To avoid this voltage matching problem, all components should be integrated in a single SoC [MGH05].

One information-based principle of [Sar12] at the architecture level is the most energy-efficient trade-off between analog and digital processing. Analog processing consumes more energy the more precision is required. On the other hand, with less analog processing, faster and thus more power consuming ADCs and digital circuits are required.

Another power reduction technique at the architecture level is to find an optimum trade-off in hardware/software partitioning using hardware/software co-design (cf. Sec. 2.4). This means, that software tasks are moved to hardware accelerators to offload the CPU. The reconfigurable CPU supplement modules proposed in this thesis are an example for this approach.

Another approach of a low-power technique at the architecture level is to provide parallel or pipelined processing for applications and algorithms which benefit from these. Several CPU designs specially optimized for low power consumption were proposed. These employ application specific or general low-power instruction set architectures and register sets. Ultra-low-power MCUs provide an active state and a number of low-power states. For example, the Microchip PIC24 microcontrollers with the nanoWatt XLP technology and the TI MSP430 microcontrollers have one active and four to five low-power modes [Tew10].

Although larger FPGAs require low-power techniques to avoid problems with heat dissipation and supply current limitations, several low-power FPGAs offer multiple power modes with full and reduced functionality. FPGAs with non-volatile configuration storage avoid the power consumption for configuration at startup [Tew10, BAPZ09a, BAPZ09b]. Another example is the functional partitioning of large state-machines with datapath (FSM+D [GR94]) into smaller FSMs and datapath segments to reduce the power consumption.

Logic Level. Logic synthesis tools perform optimization and transformation of combinational logic and map the logic to the gate library. The combinational logic optimization includes “don’t care” optimization, path balancing to reduce glitches, and factorization.

For FSMs, power-aware encoding of the state vector was proposed (cf. Sec. 2.6). For sequential circuits, flip-flop retiming can reduce the total (spurious) switching activity and thus the power consumption. A very common and widely used low-power technique at logic level is clock gating, i.e., disabling the clock signal for a subset of flip-flops. Automatic clock-gating can be performed for vectored registers, but more power reduction can be achieved by manual clock gating of larger functional units utilizing higher-level information. [RN00] proposed “Control Generated Clocks” instead of gated clocks for an FSM+D setup. They use control signals from the FSM for the datapath as clock signal, which simplifies timing analysis and reduces the power consumption by up to 73%. It further reduces the gate count.

Circuit Level. Power optimization techniques at the circuit level summarize approaches involving transistors, gates, logic family, signal paths, and routing. This includes the gate library, e.g., by providing complex gates like AND-OR-invert which can be realized with a low transistor count. The power is further optimized by utilizing gates with different output strength (transistor

sizing). Weak drivers are chosen by the synthesis and the place and route tools for short nets with a low fan-out while strong drivers are used to drive high fan-out nets. Besides reducing the signal delay, this further reduces the signal transition times, which have a large impact on the short-circuit power (see Sec. 2.1.1).

Most CMOS semiconductors are built using static logic, i.e., all signals are driven by output drivers either high or low. Another technology called dynamic logic utilizes the inherent capacitance of wires and gates to store signal values. In this technology, combinational gates also have a clock input which periodically updates the output value. Due to leakage, the clock frequency has a lower limit. Dynamic logic does not lead to glitches, because each gate only has one transition per clock cycle. Further, it does not have short-circuit currents during signal transitions and has smaller capacitances to charge and discharge. On the other hand, even combinational gates with constant values require switching activity, but in total, it has potential for lower power consumption than static logic.

Another alternative to conventional CMOS logic is complementary pass-gate logic (CPL), which implements logic cells with pass-transistors. This simplifies the logic cells and reduces the transistor count. On the other hand, each pass-transistor causes a voltage drop, which reduces the drive current and the speed. The voltage drop is a special concern at low supply voltage in low-power designs. Depending on the actual application, pass-transistor logic can provide power reduction compared to conventional CMOS circuits.

Technology Level. Power optimizations at technology level affect the properties of transistors and the electrical circuits on the chip. This is performed in the process development at the semiconductor manufacturing fab. The designer or layouter only selects the technology or employs the optimized devices. As shown in Sec. 2.1.1 the dynamic power quadratically depends on the supply voltage. Unfortunately, the propagation delay time of logic gates increases when the supply voltage is reduced. Therefore various approaches were applied to use multiple supply voltages (higher voltage for timing critical paths and lower voltage for relaxed paths to save energy), but this requires level-shifters and separate power distribution networks.

Using a variable supply voltage also requires to inversely variate the clock frequency (dynamic voltage and frequency scaling, DVFS) using a closed-loop control circuit. This allows the dynamic adaption of the trade-off between circuit performance and energy efficiency to the demands of the application. The static leakage current of metal-oxide-semiconductor (MOS) transistors depends on their threshold voltage V_{T}, i.e., a lower V_{T} (and therefore higher difference to the supply voltage) provides higher speed but also higher leakage. Many fabs offer distinct sets of logic cells with higher and with lower V_{T} for relaxed and for timing critical applications, respectively. In simple applications, the whole circuit is designed with one of these sets. For demanding applications, synthesis tools can optimize the circuit by choosing the cells with higher and lower threshold voltage for slower or timing critical paths, respectively. Another technique at the technology level is termed variable threshold voltage. A negative bias voltage is applied to the substrate of inactive modules. This locally increases the threshold voltage and therefore reduces the leakage current.

At the beginning of this section a classification of the reasons to reduce the power consumption of embedded systems and integrated circuits was introduced. This was followed by an overview of the sources of power consumption. Finally an overview of widely used low-power techniques, categorized according to their design abstraction level, was given. This shall provide a framing and

motivation for the approach proposed in this thesis, i.e., to supplement the CPU with autonomous reconfigurable modules.

However, the goal of this thesis is the scientific development a *methodology for the design* of such autonomous reconfigurable CPU supplement modules. Therefore, in the next section, the properties of reconfigurable logic and CPU supplement modules are summarized and a scheme for the classification is developed. This is used in the following sections for the review of similar approaches and methodologies, which constitute the scientific basis for this thesis.

2.2 Properties and Classification of Reconfigurable Logic

When the CPU of embedded systems is supplemented with dedicated logic, two different paradigms of functionality are involved. The CPU executes programs as a serialized stream of instructions. In contrast, the functionality of a logic circuit is fully parallel. To emphasize this distinction, the following terms are used in this thesis. The flexibility of CPUs is denoted by "programmable", while the flexibility of logic is termed "configurable". Further, the specification of the functionality of CPUs is called a "program", "software", or "firmware",[3] while the specification of the functionality of configurable logic is termed "configuration" or "configware" [HM99].

In the past decades, a large amount of research on CPU supplement modules and reconfigurable logic was conducted. The main goal of the work on supplementing the CPU with dedicated hardware was to accelerate computational tasks. The interested reader is directed to the following books and survey papers: For a historical retrospect see [Est02]. A special focus on low-power signal processing can be found in [Rab97]. [Har01a, Har01b] review a large number of architectures for reconfigurable computing including programming and workflows and group them by the interconnect topology. A detailed overview of fine-grained and coarse-grained architectures as well as topics on the development and run-time reconfiguration are discussed in [CH02] and [Com03].

[GCS+06] provides a comprehensive overview of past works and architectures covering topics on heterogeneous and non-square reconfigurable hardware, power-aware tools, and verification. In the book [HD08] a large number of works for reconfigurable computing using fine-grained FPGAs are discussed. The first two chapters of the book [VS07] provide a survey of fine-grained and coarse-grained computing. These are followed by a large number of detailed case studies. The second chapter of [TSV07] provides a thorough overview of coarse-grained architectures and discusses general issues and the design, computation, academic and commercial platforms, and design automation software. [PTD13] reviews the research on reconfigurable computing. They provide an overview of architectures and technologies, languages and compute models, tools, run-time reconfiguration, and applications.

Note that many authors use the term "application domain" for a wide field, e.g., digital signal processing for any kind of signals and algorithms. In this thesis, the term is used for much smaller fields. For example, a reconfigurable CPU supplement module could be developed for the application domain "sensor interface". The specification also includes the sensor types and the post-processing of the values. Therefore, here the term "*wide* application domain" is used to

[3] A program and software are executed using an operating system. Firmware is directly executed on a processor or microcontroller, i.e., without an operating system.

denote the former meaning and "*narrow* application domain" or just "application domain" for the latter meaning.

For the analysis and comparison of previous research, in the following sections the relevant properties and characteristics of the investigated approaches have to be defined. These properties allow to categorize the reviewed approaches in a classification scheme. All works reviewed in the next sections are summarized in Tab. 2.2 on p. 48 using this scheme. In the rest of this section, this classification scheme is described. It integrates and extends the categorization schemes used in the above cited works and of [SVKS01] and [Gre02].

- As defined in Sec. 1.1.5, the development and deployment of reconfigurable circuits comprises the pre-silicon design phase and the post-silicon design phase. The works reviewed in the following sections can be categorized by the *covered design phases*.

  Note that some of the reviewed approaches generate application domain specific but non-reconfigurable logic circuits. To enable the categorization of these works, the definition of pre- and post-silicon design phases is generalized, so that the design of non-reconfigurable logic circuits is considered as pre-silicon design phase.

- The second property of the reviewed works is their *flexibility* in the post-silicon design phase:
  - hard-wired, non-reconfigurable circuits,
  - multi-mode circuits, which only support a defined set of switchable modes,
  - application domain specific reconfigurable logic which supports even new but similar applications, and
  - full flexibility, which is only limited by the available resources.

- As already mentioned in Sec. 1.1.4, the *granularity* of reconfigurable logic ranges from fine-grained to coarse-grained and also includes the combination of both as multi-granular circuits.

- The *topology* of the interconnect of reconfigurable logic is defined with three properties:
  - It can either be regular or non-regular.
  - Further, the topology can be flat or hierarchical. For hierarchical topologies, at each level a sub-topology with different characteristics is possible.
  - The third property describes the possible connections [WGS+12, Har01a], for example:
    - connections from all outputs to all inputs using multiplexers (MUXes), but this requires a large circuit overhead,
    - connections from a subset of outputs to a subset of inputs using MUXes,
    - alternating layers of functional units and interconnecting MUXes,
    - 1D chain of functional units,
    - 2D mesh interconnect like used in most commercial FPGAs,
    - multistage interconnect networks (MINs),
    - bus topology, as used in many microprocessors and microcontrollers,
    - crossbar topology, and
    - tree based interconnect topology.

- The reconfigurable logic can be optimized towards data processing or towards bit-level, control-dominated and event-based *wide application domain*.

- The main goal of the *optimization* can be performance, low power consumption, chip area, or flexibility. Also special extensions to improve the synthesis, the mapping of logic, or place and route algorithms are goals for optimization.

- Another property is the time and the frequency of *configuration*:
  - Although not considered reconfigurable logic, hard-wired logic circuits are "configured at design time" [SVKS01].
  - The configuration can be stored in one-time-programmable memory (e.g., with fuses) or in non-volatile memory (e.g., with EEPROM, electrically erasable PROM, cells [BAPZ09a]).
  - The term "configurable logic" is used for circuits, which are configured only once (e.g., upon start-up) or very infrequently.
  - In contrast, the term "*re*configurable logic" is used for circuits, which are reconfigured on a regular basis, e.g., to adapt a network protocol handler to changing protocols.
  - With dynamic reconfiguration, the configuration is changed frequently during the operation to reuse functional units.

- For reconfigurable as well as hard-wired CPU supplement modules, the *coupling* with the CPU influences latency and bandwidth [TCW+05].
  - External stand-alone processing units are placed in a separate chip.
  - Attached processing units and peripherals like GPIOs or timers are directly connected to the on-chip bus.
  - A co-processor is directly connected to the CPU.
  - Supplement modules can also be integrated into the CPU, e.g., to implement custom ISA extensions.
  - Finally, the CPU itself can be embedded in reconfigurable logic, which is in fact a reversed scenario.

- Related to the coupling, the *autonomy* of a CPU supplement modules ranges from supplement modules which are controlled by the CPU and only perform actions initiated by the CPU to supplement modules which themselves control the CPU (e.g., issuing interrupts).

- The abstraction level of the *design entry* for (reconfigurable) logic can range from
  - graphical architecture generator,
  - domain-specific languages,
  - programming languages (e.g., used in high-level synthesis),
  - hardware description languages like VHDL and Verilog, to
  - logic netlists.

Note that some of the above properties are pure categories, but others allow to order and sort the "values". With this set of properties the approaches investigated in the following sections will be categorized.

In the next sections, previous research on application domain specific and on reconfigurable CPU supplement modules is reviewed. This includes works dating back to the origins of these fields which introduced major contributions also relevant for this thesis. Especially the research reviewed in Sec. 2.5 partly shows a large overlap with this thesis. The relevant contributions but also the open topics are discussed in detail.

2.3 Application Domain Independent Reconfigurable CPU Supplement Modules

As stated above, the approach proposed in this thesis merges two distinct areas of research: 1) application specific dedicated hardware to assist the CPU, and 2) reconfigurable logic. In this section, approaches which cover both areas are reviewed, but contrary to this thesis these reconfigurable logic modules were manually developed and only slightly tailored to a wide application domain. Therefore these architectures are mostly universal. The wide application domain was given and the work performed and the concepts employed in the pre-silicon design phase were fixed and targeted towards that given wide application domain. In other words, the pre-silicon methodology was not independent of the wide application domain. The pre-silicon design phase was performed by the vendor. The user only performs the post-silicon design phase.

In Sec. 2.3.1, commercial reconfigurable CPU supplement modules are reviewed. In Sec. 2.3.2 commercial embedded FPGAs as CPU supplement modules are investigated. Finally, in Sec. 2.3.3 academic research on reconfigurable embedded FPGAs is discussed.

2.3.1 Commercial Reconfigurable CPU Supplement Modules

Commercial microcontrollers (MCUs) optimized for low-power operation are frequently employed in applications like WSNs. All MCUs provide a comprehensive set of peripherals like timers, serial protocol interfaces, ADCs, etc. which operate autonomously and in parallel to the CPU. Therefore these peripherals themselves are CPU supplement modules and most peripherals have some degree of configurability. However, the flexibility is limited and specific to that peripheral, e.g., the direction of a counter, or the number of stop bits of an UART (universal asynchronous receiver and transmitter).

In this section generic reconfigurable CPU supplement modules of current devices on the market are reviewed. The selected devices are microcontrollers, but not (floating-point) co-processors, application processors, or media processors because the focus of this work is on ultra-low-power devices and control-dominated applications.

Microchip Configurable Logic Cell (CLC)

A subset of the low-power 8-bit PIC microcontrollers, e.g., the PIC16(L)F150x series, from Microchip Technology Inc. are equipped with up to four "Configurable Logic Cell" (CLC) modules [Mic14].[4] Each CLC module provides a simple reconfigurable logic function. The inputs and outputs can be selected from pins and internal peripherals. A change in the output signal can further generate an interrupt to notify the CPU. The logic function can be configured as one of eight possible combinational (e.g., "`(A and B) or (C and D)`") and sequential (e.g., D-FF with set and reset) types.

The CLC modules offer only very basic functionality but can reduce the number of external components as well as avoid CPU intervention.

[4] `http://www.microchip.com/clc` [2015-08-20]

Analog Devices Programmable Logic Array (PLA)

Analog Devices Inc. offers the ADuC70xx series of microcontrollers equipped with "Programmable Logic Array" (PLA) peripherals [Ana06]. The MCUs are not low-power devices (slow wake-up from sleep mode, high power consumption even during sleep mode), but the reconfigurable CPU supplement module is of interest in this section.

The CPU supplement module offers 16 PLA elements, which have a two-input LUT and a D-FF, which can be bypassed.[5] This is a fine-grained architecture, very similar to FPGAs. It offers full flexibility (for which it appears to be optimized). The inputs and outputs of the PLA elements can be GPIOs of the MCU, a small set of other peripherals, registers accessible by the CPU, interrupts, and other PLA elements. The PLA operates fully autonomous from the CPU, even in sleep mode.

Using LUTs (like FPGAs) and many possible connections between the PLA elements enables the implementation of a large number of logic functions, although the total number of only 16 elements is a limiting factor.

Energy Micro Low-Energy Sensor Interface (LESENSE)

Silicon Laboratories Inc., former Energy Micro AS, introduced a dedicated reconfigurable sensor interface "Low-Energy Sensor Interface" (LESENSE) as a proprietary peripheral block [Lar11, Ene14] in a subset of the low-power Gecko 32-bit microcontroller series. The LESENSE peripheral is a dedicated low-energy sensor interface and is reviewed here because its application domain is similar to the sensor interface task outlined in Sec. 1.1.1. It is especially tailored to capacitive and inductive proximity sensing (including excitation), e.g., touch buttons, and general analog sensors. A total of 16 external sensors can be monitored.

The LESENSE peripheral is fully autonomous and continues its operation in low-power modes while the CPU is inactive. It can generate a wake-up event for the CPU, thus effectively off-loading the CPU from sensing tasks. Its flexibility is very limited within its application domain of a sensor interface for capacitive and inductive proximity sensors. The LESENSE peripheral is optimized for low power consumption. It is a manually designed module, tailored for specific sensing tasks, i.e., it implements a fixed application domain.

Cypress Programmable System-on-Chip (PSoC)

Cypress Semiconductor Corporation produces MCUs marketed as "Programmable System-on-Chip" (PSoC). The chips only have a small number of hard-wired peripherals, while all other required peripherals should be implemented using the programmable analog and digital blocks. These are configured by the firmware through registers at startup or during runtime.

The older generation of PSoC 1 MCUs [Cyp12b, MSB02] will not be discussed here. The newer generation of PSoC 3 to 5 MCUs [Cyp12a] use an 8-bit 8051 and a 32-bit ARM CPU. The "digital system" and the "analog system" are different from the PSoC 1 MCUs. The chips contain the hard-wired peripherals real time clock, watchdog timer, I^2C master/slave, USB (universal serial bus), timer/counter/PWM (pulse-width modulation), and CAN (controller area network). All

[5] The name "Programmable Logic Array" (PLA) is not related to reconfigurable sum-of-product type devices [Kat94].

other required peripherals should be implemented using the configurable logic. The "digital system" holds an array of up to 24 "universal digital blocks" (UDBs) connected with a flexible routing of custom topology. These allow the implementation of digital functions like timers, UART, I^2C (inter-integrated circuit), SPI (serial peripheral interface) and cyclic redundancy check (CRC).

Cypress offers a large library of prepared configurations with their design tool. These include UART, SPI, pseudo random sequence generators, timer and counter, PWM, etc. The tool also allows to implement custom functions using Verilog [MF14]. It is also possible to design an application by directly setting the appropriate register values.

The approach proposed in this thesis only includes digital logic, therefore the analog system will not be further evaluated. The newer generation of PSoC devices offers generic, fine-grained product-sum-term logic which allows the implementation of (limited) custom logic. Additionally a coarse-grained datapath with dynamic reconfiguration allows the implementation of logic and arithmetic operations but its flexibility is still limited to the application domain of the functional units of the UDB datapath. Many functions require frequent interaction with the CPU. Additionally, the digital blocks as well as the routing are very versatile, therefore cause a high power consumption. In 2012 Cypress introduced[6] the PSoC 5LP series, which is optimized for low power consumption [Cyp15b], and in 2015 the low-power PSoC 4 BLE series with integrated Bluetooth Low Energy radio [Cyp15a] were introduced.[7] Both product series are appropriate for WSN nodes.

Texas Instruments Control Law Accelerator (CLA)

The Texas Instruments Incorporated (TI) microcontrollers of the Piccolo F280xx and the Delfino F283xx series for real-time control include the programmable "Control Law Accelerator" (CLA) peripheral. This peripheral is a small and simplified CPU tailored for single precision floating point processing. Its main purpose is the autonomous handling of control loops with low latency, e.g., motor control, and factory automation.

The CLA has dedicated code and data memories and has access to shared peripherals, especially the ADC and PWM modules, i.e., it is an attached processing unit. Communication with the main CPU is handled through shared memories, messages and events. The CLA is programmed using a special assembly language or using a reduced implementation of the C language [Tex15a]. The binary program code is copied from the main CPU to the CLA code memory and can be changed at any time, analogous to reconfigurable logic. After this setup procedure, the CLA is fully autonomous.

The CLA peripheral is clearly intended to off-load the CPU, but its main purpose is the handling of high performance real-time control loops, so it can not be considered a low-power approach.

2.3.2 Commercial Embedded FPGAs

The five architectures reviewed in the previous section are generic but limited (Microchip CLC, Analog Devices PLA), designed for a specific narrow application domain (Energy Micro LESENSE), or powerful but tailored to a specific wide application domain (Cypress PSoC, TI CLA).

[6] http://www.cypress.com/?rID=73152 [2015-08-20]
[7] http://www.cypress.com/?rID=108151 [2015-08-20]

In this section universal and powerful fine-grained *embedded* FPGAs (eFPGAs) are reviewed. In chapter 3 of his PhD thesis [PSH04], the author divides eFPGAs in

- adding reconfigurable FPGA structures into an SoC, similar to an IP core [GZ97], and
- extending an FPGA chip by typical elements of an SoC, especially CPUs, termed Systems-on-Programmable-Chips (SoPC).

eFPGA fabrics are coupled with the CPU and utilized as CPU supplement modules. It is also possible to couple the eFPGA fabric with other components of the SoC, but here only CPU supplement modules are investigated. In chapter 3 of [PSH04] a survey of a number of architectures is given and assigned to these two categories. Notable products in the second category are Xilinx Virtex II Pro FPGAs and Xilinx Virtex 4 FX FPGAs. These are powerful FPGAs which include up to four hard-wired PowerPC CPUs.

In 2011, Xilinx introduced[8] the *Zynq-7000 All Programmable SoC*, which uses the opposite approach, i.e., an SoC which includes a large eFPGA fabric. The SoC includes two ARM Cortex-A9 CPUs, external memory interfaces, USB 2.0, SD card interface, Gigabit Ethernet, UART, CAN, I^2C, SPI, and ADCs. The interface from the CPU to the eFPGA offers multiple up to 64-bit AMBA AXI bus (ARM advanced microcontroller bus architecture, advanced extensible interface bus) interfaces [Xil14d].

In 2014 the extended version *Zynq UltraScale+ MPSoC* was announced[9] by Xilinx. The devices include four 64-bit ARM Cortex-A53 CPUs plus two 32-bit ARM Cortex-R5 CPUs, a graphical processing unit, external memory interfaces, USB 3.0 and 2.0, SD card interface, SATA, DisplayPort, Gigabit Ethernet, PCI-Express, UART, CAN, I^2C, SPI, and ADC. The interface from the CPU to the embedded FPGA offers multiple up to 128-bit AMBA AXI bus interfaces [Xil15e].

In 2012, Altera introduced[10] the *SoC FPGAs*, which include two ARM Cortex-A9 and eFPGA fabric. The selection of hard-wired IP cores and the interfaces between the CPU and the eFPGA fabric in these devices is similar to the Xilinx Zynq-7000 [Alt15a]. In 2013, Altera announced[11] the *Stratix 10 SoC FPGA*. The devices include four 64-bit ARM Cortex-A53 CPUs, but not much more information is available at the time of writing [Alt15b].

The Xilinx and Altera SoC FPGAs have a high speed system bus interface between the CPU and the reconfigurable fabric. This is beneficial for a wide range of applications. The CPUs can execute an operating system like Linux and application software, while in the eFPGA fabric custom peripherals like graphic controllers, special network interfaces, real-time control, digital signal processing, encryption, etc., can be implemented [Xil15c]. Typical applications are automotive driver assist systems, data center acceleration, software defined radio, motor control, robotics, etc. [Xil15d].

The Xilinx and Altera SoC FPGAs are sold as chips with no possibility for customization of the silicon devices themselves. However, the user utilizes the FPGA fabric to implement the desired custom logic functionality. In contrast, when developing a custom SoC, eFPGA IP cores are available.

[8] http://press.xilinx.com/2011-02-28-Xilinx-Introduces-Zynq-7000-Family-Industrys-First-Extensible-Processing-Platform,1 [2015-08-20]

[9] http://press.xilinx.com/2014-02-24-Xilinx-Introduces-UltraScale-Multi-Processing-Architecture-for-the-Industrys-First-All-Programmable-MPSoCs [2015-08-20]

[10] http://newsroom.altera.com/press-releases/nr-cyclonev-soc-shipping.htm [2015-08-20]

[11] http://newsroom.altera.com/press-releases/nr-altera-arm-a53.htm [2015-08-20]

NanoXplore offers the *NX-eFPGA* embedded FPGA. It uses fine-grained reconfigurable logic with 4-input LUTs and stores the configuration in SRAM cells [Nan14]. The IP core is delivered as a hard macro and generated by a proprietary layout generator. The eFPGA IP core can include memory, digital signal processing functions, and user-specific functions.

ADICSYS offers the *efpga* embedded FPGAs [ADI13]. The product is delivered as a soft IP core and synthesized by the customer using standard ASIC synthesis tools. The FPGA architecture can be customized for its size, LUT style, and routing density.

Menta SAS offers the *eFPGA Core IP* which is delivered either as hard macro or as synthesizable soft macro. The fabric uses LUTs with a customizable number of inputs and can include memory blocks, arithmetic blocks, and custom blocks. The routing resources as well as the size and the outline of the IP can be customized to the customers requirements [Men12, Men10a, Men10b].

There is no dedicated definition of the interface between a CPU (or any other SoC core) and these three eFPGA IP cores. However, the fabrics can be configured for any number of input and output signals, and therefore tailored to the required interface.

The fabric of the above SoC FPGA chips and the eFPGA IP cores contains fine-grained logic blocks and coarse-grained functional units like multipliers and RAMs. The signals are routed through a mesh interconnect. The eFPGAs provide full flexibility which is only limited by the available resources. The application scenario is user defined and can include control-dominated as well as data processing tasks. The fabrics are optimized for performance and for flexibility. The configuration can be applied and changed frequently, and the Xilinx and Altera chips also support partial reconfiguration. The applications realized in the eFPGAs can perform autonomous tasks, can depend on the CPU or can control the CPU. The design entry for the reviewed eFPGAs are hardware description languages (HDLs) like VHDL and Verilog.

2.3.3 Research on Embedded FPGAs

In the previous section, commercial eFPGAs were reviewed. In this section, research on the optimization of characteristics of the (embedded) FPGA fabric are investigated. As in the previous section, the user only performs the post-silicon design phase. However, here the research considers the FPGA fabric itself and therefore represents the pre-silicon design phase.

The Triptych FPGA Architecture

An early example of optimizations of the interconnect of FPGAs is [BEHB95]. They report that the chip area of FPGAs with a mesh interconnect topology is strictly separated between the functional units (FUs) and the routing resources. To allow more complex applications to be routed, the routing resources are increased. On the other hand, the utilization of functional units is decreasing. The *Triptych FPGA* architecture [BEHB95] introduces FUs, which can be utilized either as logic function or as routing resources (termed “routing and logic blocks”, RLBs).

One important observation is, that for combinational logic the fan-in of logic cells often has a triangular shape, i.e., signals from multiple other cells gather at a cell. Also the fan-out of a cell often has a triangular shape, i.e., the output signal is distributed to multiple other cells. To support this typical signal flow, the Triptych FPGA architecture adds diagonal routing resources to connect each RLB with its north-east, north-west, south-east, and south-west neighbors.

This work on fine-grained FPGA fabrics shows important improvements on routing utilization and logic structures. Using FUs for routing can save up to three signals of the interconnect and the integration of diagonal connections is now common in FPGAs. Further, the area utilization and the signal propagation delay were reduced.

Optimization of the Interconnect Power Consumption

While the previous approach optimized the interconnect topology to reduce chip area and increase the utilization of the logic blocks, [KR98] at the University of California, Berkeley optimized the energy consumption with only a minimal impact on speed. The authors measured the capacitance and energy consumption of the internal resources of the Xilinx XC4003A FPGA. 65% of the total energy is dissipated by the interconnect, 21% by the clock distribution, 9% by the IOs, and only 5% by the logic functions.

To reduce the interconnect power consumption, the utilization of connections with different lengths were determined. For a set of benchmark netlists, 50% of the interconnect wiring uses direct connections between complex logic blocks (CLBs) and another 37% of the interconnect wiring uses connections to the second next CLBs. Only 10% of the interconnect wiring uses longer wires. This shows that optimizing short wires has a high impact on the total power consumption.

To reduce the interconnect capacitance and therefore power consumption, a hierarchical interconnect was introduced. At the lowest level, local interconnect between the logic cells was improved for high utilization and low capacitance, also adding diagonal elements as [BEHB95]. For longer connections, wires at the higher levels of the interconnect hierarchy were added.

The authors methodically investigated the main contributions of power consumption and addressed these for the optimization by introducing a hierarchical interconnect.

A similar approach was used by [GZR99], also at the University of California, Berkeley, who also introduced a hierarchical interconnect. The interconnect was organized in three levels. At level 0, direct connections between nearest neighbor CLBs were used. This reduced the energy-delay product by a factor of 3 compared to traditional mesh interconnect. At level 1 a symmetric mesh across the FPGA fabric was used. The problem of this topology is that the energy-delay product increases by the third power of the distance. Therefore at level 2 a binary tree interconnect was used. Shorter connections (less than 10 CLBs) are routed through the level 1 mesh network while longer connections are routed via the level 2 binary tree. The tree topology clusters neighbors, i.e., they can be connected through the lowest level of the tree, which should be routed using the level 1 mesh network. Therefore an inverse clustering was applied, where CLBs with larger distances are connected at the lowest level of the tree hierarchy.

An 4×8 array of this embedded FPGA architecture was used in the Maia chip [ZPG+00] to implement bit-level functions. It was connected via a hierarchical mesh reconfigurable network with the embedded CPU and other coarse-grained reconfigurable units.[12] The 4×8 array of logic blocks required 2.76 mm^2 of the chip manufactured in a 250 nm 6-metal CMOS technology.

The hierarchical interconnect topology and especially the inverse clustering for long connections reduces the number of switch boxes required. Together with other improvements at the technology level, a considerable reduction of the power consumption was achieved.

[12] More details of these coarse-grained reconfigurable units from the Berkeley Pleiades architecture will be discussed in Sec. 2.5.2 on p. 40.

Synthesizable Embedded FPGA

The three exemplary embedded FPGA architectures reviewed in the previous sections are constructed as full custom design, i.e., the layout was constructed manually. [WKW+05, WAWS03] from the University of British Columbia introduced a synthesizable embedded FPGA as an RTL (register transfer level) design which is implemented using standard library cells.

In the reconfigurable architecture many combinational logic loops are present which causes problems for the synthesis tools. Therefore the authors propose a directional architecture which allows signals only to flow from left to right. Additionally, the architecture only allows combinational logic and does not include D-FFs. The configuration of the reconfigurable fabric is stored in a shift register of D-FFs. Due to the high number of D-FFs the clock tree requires considerably more chip area and power as for conventional hard-wired logic circuits. A test chip using the reconfigurable architecture showed an area increase by a factor of 560 but only a doubling of the delay compared to a hard-wired implementation of the logic functions. The area was 6.4 times larger than that of a full custom layout FPGA.

To allow sequential logic in the synthesizable embedded FPGA, [YW04] suggested two architectures but both require a large additional area overhead. To reduce the area of the synthesizable embedded FPGA, [ALS05] suggested to add tactical standard cells for the configuration D-FFs, LUTs, and MUXes. The size of the whole eFPGA core was reduced by an average of 58% and the delay was reduced by 40%. This solution is only 2–2.8 times larger and 10% slower than a full custom eFPGA.

Providing synthesizable FPGA fabrics allows an easy integration in a standard digital ASIC design flow. However, this leads to a large overhead of the chip area compared to full-custom implementation. The concept to add tactical standard cells reduces the area and delay, but the design is not independent of the semiconductor process. Further, limiting the fabric to combinational logic and requiring a directed signal flow places strong constraints on the routability.

XiSystem SoC with PiCoGA and eFPGA

The XiSystem SoC [LCB+06] contains two reconfigurable CPU supplement modules. The *Pipelined Configurable Gate Array* (PiCoGA) is a reconfigurable coarse-grained logic module to extend the instruction set architecture of the XiRisc CPU [CTL+03]. Additionally, a fine-grained eFPGA handles input and output (IO) protocols and pre- and post-processing of signals.

The PiCoGA is directly integrated in the CPU data path and is activated with a special instruction. Its reconfigurable fabric uses two-bit granularity. It is composed of an array of reconfigurable logic cells with LUTs to implement the function. These are connected with a mesh interconnect. Four different configuration contexts can be setup and switched within a single clock cycle. This is performed with two further instructions. In the software, these instructions are placed before and after computational intensive kernels. The kernels then can use the PiCoGA to perform custom operations. The eFPGA is connected to the system bus and has FIFOs for fast data exchange. It is a fine-grained single-bit array of LUTs using a mesh interconnect.

Both reconfigurable fabrics are flexible and only limited by the available resources. However, the two-bit granularity of the PiCoGA fabric still requires a high overhead for logic elements and signal routing. Using two separate CPU supplement modules to split data processing and control allows to optimize each architecture for its wide application domain. The application domain

of the XiSystem SoC is however clearly high-performance data processing and not intended for low-power devices like WSN nodes.

2.3.4 Summary

In this section, universal, application domain *independent* reconfigurable CPU supplement modules were reviewed. Although this only covers one aspect of the approach proposed in this thesis, useful conclusions can be drawn from the reviewed work.

The Energy Micro LESENSE peripheral shows that the autonomous control of a sensor interface as used as example in Sec. 1.1.1 is considered important for low-power operation. The reconfigurable Microchip CLC and the Analog Devices PLA CPU supplement modules are fine-grained approaches with a high degree of flexibility but with limited resources. On the other hand, the Cypress PSoC MCUs delegate the implementation of most peripherals to the flexible and multi-granular reconfigurable logic. Especially the new generation of PSoC MCUs is very flexible, but not tailored to a narrow application domain, which causes an increased power consumption. The Texas Instruments CLA is a simple CPU optimized for floating-point processing. The approach proposed in this thesis aims to off-load the CPU, therefore the CLA is not a suitable solution.

Commercial SoC FPGAs with a CPU tightly coupled to a fine-grained FPGA fabric as well as eFPGA IP cores are flexible and powerful. Due to the large required chip area this approach is only useful for semiconductor processes with high integration density. These also have high leakage currents, which further increases the power consumption. Therefore embedded FPGA fabrics are not suitable for the approach proposed in this thesis.

The Triptych architecture and both works at the UC Berkeley show that the interconnect contributes a considerable amount of area and power consumption. Therefore the topology and the interconnect developed for this thesis must allow short wires with low capacitance.

Soft IP cores of eFPGAs are synthesizable and use standard cells. This is independent of the ASIC design tools as well as the semiconductor process which is a requirement for this thesis (cf. Sec. 1.1.7). However, synthesizable eFPGAs cause a large area overhead (6.4 times [WKW+05, WAWS03]) and delay overhead (2 times). This emphasizes to use coarse-grained logic wherever possible.

[WKW+05, WAWS03] pointed out problems with combinational loops in reconfigurable logic for the static timing analysis (STA) performed by EDA (electronic design automation) tools. The suggested solutions considerably limit the flexibility or require a high area overhead. They also refer to a large area overhead for the clock tree required for the D-FFs to store the configuration data. This includes clock buffers as well as routing and congestions. To reduce this problem, the amount of configuration data must be kept low. The XiSystem SoC supports both, control-dominated tasks and data processing, with the inclusion of separate fine-grained and coarse-grained reconfigurable architectures, respectively.

The reconfigurable architectures reviewed in this section were manually developed and optimized. This means, that the engineers and scientists determined and optimized the functional units and the interconnect by hand. In this thesis an *automated* methodology for the development of reconfigurable CPU supplement modules is proposed. Therefore in the next section, approaches

for the automatic generation of application specific but non-reconfigurable logic circuits are reviewed. This is followed by an in-depth review of automatic generation of application specific and reconfigurable logic circuits in Sec. 2.5.

2.4 Application Specific Non-Reconfigurable CPU Supplement Modules

The approach proposed in this thesis shifts tasks from software to hardware. This topic was coined as "*Hardware/Software Co-Design*" (HW/SW co-design) and "*Hardware/Software Partitioning*" (HW/SW partitioning) and worked on starting from the early 1990ies [Wol03]. Although HW/SW co-design creates non-reconfigurable logic, it is relevant for this thesis because application specific dedicated logic is generated. Therefore key concepts from HW/SW co-design like partitioning and high-level synthesis (discussed below) are also employed by the research on application domain specific reconfigurable logic, which is reviewed in Sec. 2.5. Note that the term "reconfigurable" is also present in HW/SW co-design, but relates to the use of a universal reconfigurable target platform like FPGAs or coarse-grained reconfigurable logic.

The main goal of HW/SW co-design is to split an algorithm in such a way, that, for a given metric (e.g., performance, power), the partitioned implementation is an improvement over the pure software implementation. The optimization is accomplished by design-space exploration, i.e., iteratively shifting the boundary between hardware and software and characterizing the system.

A lot of research was conducted, especially to automate this process. Automatic analysis of the application and design-space exploration are used to identify the computation intensive subtasks and to model the resulting partitioned system. Co-simulation and automatic synthesis of software and hardware was used to profile the application in terms of execution time and performance [Wol03, Vah09].

An important concern is the communication between the processor and the dedicated logic. Implementations range from loosely coupled architectures, where a shared memory is used, up to tightly coupled systems with integration into the processor as application specific instruction-set processor (ASIP) [Vah09] (compare the classification of the coupling in Sec. 2.2). The question of suitable specification languages was discussed. Describing the whole system in a programming language like C would bias toward a software implementation, while using a hardware description language like VHDL or Verilog would bias towards a hardware implementation. Unified languages like SystemC were developed and employed [Wol03].

The target architectures in hardware/software partitioning initially were ASICs. In the beginning, the accelerator was placed as an additional chip to the microprocessor on a PCB. After that, both were integrated in a single chip [Wol03]. Later, FPGAs were used as target architectures. Devices with hard-wired CPUs in addition to the reconfigurable fabric were used [Wol03, Vah09] (cf. Sec. 2.3.2).

2.4.1 High-Level Synthesis

When a computational intensive kernel of an algorithm is selected for implementation in hardware, *high-level synthesis* (HLS) is used for the generation of the logic circuit [WC91, GR94]. The

source code of the algorithm (e.g., in a high level language such as the C programming language) is analyzed for its data and control flow. Then it is mapped onto a pipelined architecture with ALUs, registers, multiplexers, etc. The optimization of the algorithm is performed by exploring different implementations of parallel and serial processing to trade off area, frequency, and latency. The user controls this process by specifying constraints on these parameters.

For a long time, HLS was mostly an academic topic. Examples are the Riverside Optimizing Compiler for Configurable Computing (ROCCC) project [VPNH10][13] at the University of California, Riverside and the LegUp project [FCC+14, CCA+13][14] at the University of Toronto. Lately, HLS is gaining widespread acceptance in industry, partly supported by tools of the major FPGA vendors, such as Xilinx Vivado HLS [Xil14c] and Altera C-to-Hardware Acceleration (C2H) Compiler [Alt09].

2.4.2 Research on HW/SW Co-Design

The research on HW/SW co-design reviewed in this section is summarized in Tab. 2.2 on p. 48.

The *Nymble System* [HLOK13] developed at the Technische Universität Darmstadt uses manual partitioning to extract hardware accelerators. The developer marks regions of functions which should be extracted. Then the C code is analyzed using an open-source compiler infrastructure. The marked regions are automatically extracted as new functions with arguments for all input and output data. The intermediate representation of these functions is converted to a combined control and data flow graph (CDFG) from which Verilog RTL code is generated. The remaining software part is patched to implement the hardware/software interfaces. Finally it is compiled to machine code.

The data transfer between the software and the hardware part is implemented via memory-mapped registers for low-latency applications and via shared memories for high-bandwidth applications. The authors evaluated the Nymble system using several benchmark applications and executed it on a Xilinx Virtex 5 FX FPGA with an included PowerPC CPU. Unfortunately the authors don't report the overhead required for software-to-hardware and hardware-to-software control transfers.

The driving force behind HW/SW partitioning was the optimization of performance. However, it was also used with the main goal of power reduction. [Hen99] from the NEC C&C Research Laboratories presented a methodology to optimize the power consumption of a whole system of a CPU, caches and memories plus application specific cores. The application is divided into clusters, and the cluster with the highest utilization rate is synthesized and evaluated for its energy consumption. The remaining software part is analyzed using an instruction-set simulator with energy estimation including models for caches and memories. In an iterative process, design parameters are modified to optimize the total energy.

Energy reduction utilizing reconfigurable platforms with integrated CPUs was shown by [SV02] at the University of California, Riverside. They manually translated software loops to VHDL modules. The energy consumption of several benchmark applications executed on three commercial FPGA architectures with an integrated CPU were evaluated. The results show an average energy reduction of 25% to 71%.

[13] http://roccc.cs.ucr.edu/, https://github.com/nxt4hll/roccc-2.0 [2015-08-20]
[14] http://legup.eecg.utoronto.ca/ [2015-08-20]

2.4.3 Summary

In this section the generation of application specific but non-reconfigurable CPU supplement modules with HW/SW co-design and HLS was investigated. Such approaches also only cover one aspect of the approach proposed in this thesis (cf. Sec. 2.3.4). Again, important facts can be learned.

Although [SV02] used fine-grained FPGA with integrated CPUs, moving software loops to the FPGA fabric reduced the total power consumption. Even better results were achieved by [Hen99] using hard-wired accelerators. These examples show, that HW/SW partitioning is also a suitable approach for energy reduction [VSPK04, BMM01]. According to the categorization in Sec. 2.1.2, this is a low-power technique at the (hardware) architecture level.

In Sec. 2.3 the methodology applied to develop the reconfigurable architecture was itself specific for the architecture. In contrast, the methodologies reviewed in this section are independent of the application, which complies to the requirements defined in Sec. 1.1.7. However, the methodologies use data flow graphs (DFGs) and similar intermediate representations of the processed algorithms. This limits their use to the wide application domain of data processing, because control-dominated applications and exact timing behavior can not be represented with DFGs.

HW/SW co-design and HLS are primarily applied to application scenarios for data processing. The application scenarios for this thesis include only a small demand for data processing, for example comparing the new sensor value with an old value or calculating the mean value of multiple measurements. Therefore no automatic evaluation and optimization of algorithms is required for this work.

The design entry for HW/SW co-design and HLS use high-level languages like C. In this thesis, data processing *and* control-dominated applications must be supported. However, high-level languages do not support the specification of cycle accurate timing. Therefore high-level languages are not suitable to specify the functionality of the proposed reconfigurable CPU supplement modules.

Unified languages like SystemC can specify cycle accurate timing as well as high-level representations of algorithms. However, SystemC is not suitable for the development of MCU firmware for WSN nodes due to reduced productivity, flat learning curve, and lack of skills of the developers, compared to languages like C.

In this thesis, the boundary between hardware and software is already given by the goal, that the CPU supplement module should perform all simple tasks. The CPU should only be activated, if more complex tasks are required. In other words, the CPU supplement module acts as a filter and preprocessor. Only events which require a complex handling are forwarded to the CPU.

For the mentioned reasons, the methodology proposed in this thesis should not have a unified description of the MCU firmware and the CPU supplement module. This also declines automatic analysis and partitioning of firmware. However, it includes the parallel development of software (application, drivers) and hardware (CPU supplement modules). Therefore co-simulation to verify the hardware implementation as well as the software drivers is required.

The communication between the CPU and the supplement module only involves a small amount of data and interrupts. The communication overhead in terms of latency should be minimized to reduce CPU runtime. Therefore a loosely coupled architecture using memory-mapped registers is adequate [HLOK13].

2.5 Application Domain Specific Reconfigurable CPU Supplement Modules

In the previous sections 2.3 and 2.4 only single aspects of the approach proposed in this thesis were discussed. In this section, approaches and methodologies covering all aspects of this thesis are reviewed. The works are summarized in Tab. 2.2 on p. 48.

2.5.1 The KressArray Family

The *KressArray Family* [Har01a, HKR94] was developed at the University of Kaiserslautern. The initial development was termed "reconfigurable datapath architecture" (rDPA) and later renamed to "KressArray" after the main author. The KressArray family is used for data processing, e.g., in multimedia applications. The architecture is optimized to map deeply pipelined algorithms, e.g., FFT (fast Fourier transform), digital filters, etc.

The architecture is built as a reconfigurable 2D array of reconfigurable datapath units (rDPUs). Each rDPU has two input and two output registers, a 32-bit ALU and a microprogrammable control unit. The latter is programmed with a microassembler and controls the ALU for larger operators (multiplication, division). The rDPUs are connected with a hierarchical interconnect of mesh topology which uses short statically configured connections directly between rDPUs and long connections with dynamic configuration.

The original rDPA was manually developed and optimized (i.e., manual pre-silicon methodology) [HK95, HBH+96, HHHN98] as a universal architecture. [HHHN00a, HHHN00b, HHHN00c, HHN00, Nag01] introduced a pre-silicon methodology and implemented the tool "KressArray Xplorer" to develop an application domain specific rDPA. The automatically generated architectures were termed "KressArray Family". The methodology uses the full KressArray architecture as a starting point and modifies the individual rDPUs and the routing resources. For the specification of the functionality of the reconfigurable architecture a set of applications is used.

Each application is translated to a KressArray specific netlist. From these netlists, the minimum required resources are determined. Then the most complex application is mapped to that architecture. An automatic suggestion generator proposes modifications to the current architecture. This allows an iterative improvement of the architecture. The final architecture is verified with the other applications.

The design methodology is used to optimize a reconfigurable architecture to a narrow application domain. During the optimization, only a single application is considered, but no solution for the selection of this application is provided. If an inadequate application was selected, the whole procedure must be repeated. Additionally, in the verification phase the mappings of the other applications are not checked for adverse results like long routing paths. While the design methodology is not specific to the actual narrow application domain, the overall reconfigurable architecture is optimized for and limited to data processing and does not allow control-dominated applications.

2.5.2 The Pleiades Project

The *Pleiades* architecture was developed at the University of California, Berkeley [AZW+02, AR96, Rab97, RAI+97, ASI+98]. It is based on an architecture template which uses a CPU and

a heterogeneous set of reconfigurable so called "satellite processors" connected with a reconfigurable communication network. The satellite processors implement functionalities like address generator, memory, multiply-accumulate, add-compare-select, but also more complex functionality like programmable datapaths [AR96]. The satellite processors and the interconnect together constitute the reconfigurable CPU supplement module.

The Pleiades architecture is used as a platform for HW/SW co-design. In the pre-silicon design phase, the application software is evaluated to identify compute intensive kernels (e.g., vector dot product). The satellite processors and the interconnect are optimized to execute all computing kernels of the application, one at a time. The algorithm of each kernel is manually converted to a DFG and mapped to the satellite processors.

For the automated development of reconfigurable Pleiades architectures, a pre-silicon design methodology was introduced [WZG+01, Wan01]. The first stage comprises the HW/SW partitioning of an application program using energy as the optimization goal to extract the computationally intensive kernels. With the resulting set of netlists of all kernels, the actual reconfigurable architecture is constructed. For all kernels, the maximum number of each satellite processor type is determined. Then the interconnect implemented as a hierarchical mesh is optimized. As last step, program code for the configuration of the satellite processors and interconnect and to transfer the data for each kernel is generated.

The described methodology was used to develop the Maia processor for CELP-based speech coders (code-excited linear prediction) [ZPG+00]. The dominant kernels identified for these algorithms are vector dot products, and FIR and IIR filters. The chip includes an ARM8 CPU together with 21 satellite processors (multiply-accumulate, ALUs, address generators, memories). A special satellite processor is a low energy embedded FPGA [GZR99] (reviewed in Sec. 2.3.3 on p. 33) to implement infrequent computational functions and bit-level functionality. The Maia processor achieved six times less energy consumption over the lowest power comparable DSP. The energy consumption using satellite processors is twenty times lower than an all-software implementation.

In contrast to the KressArray, the Pleiades architecture optimizes the energy consumption instead of the performance. The architecture is an irregular and heterogeneous arrangement of satellite processors instead of a regular mesh with similar FUs. Both, the KressArray and the Pleiades architecture are used for data processing and therefore need to transfer large amounts of data between the CPU and the accelerator.

The functionality of each satellite processor and therefore the granularity of the reconfigurable Pleiades architecture is manually derived from the operators used in the software kernels. The mapping of the kernels to netlists of satellite processors is also performed manually. This should be automated in the methodology proposed in this thesis.

Similarly to the KressArray family, the functionality of the reconfigurable architecture is specified with a set of netlists, in both cases derived from high-level language. The Pleiades architecture is built using a bottom-up-creation approach. This approach requires to find a "common denominator" of satellite processors required by the set of netlists and to optimize the interconnect between them. This is an applicable basis for the methodology proposed in this thesis.

[Wan01] also implemented automatic code generation for the configuration of the architecture and for the data and control transfer between the CPU and the satellite processors. In this thesis, there is no HW/SW partitioning, so the user has to manually develop software drivers for the reconfigurable module. The design methodology must automatically generate code which allows the implementation of drivers without the knowledge of the actual reconfigurable architecture.

2.5.3 Strategically Programmable System

The previously reviewed Pleiades project used manual mapping of the application to the reconfigurable architecture. The *Strategically Programmable System* (SPS), developed at the University of California, Los Angeles [OMBKS01], implements an approach for automatic mapping and generation of a reconfigurable architecture. It uses a fine-grained island style FPGA fabric with mesh routing topology and embeds coarse-grained hard-wired blocks called "Versatile Parameterizable Blocks" (VPBs). Examples for VPBs are adders, multipliers, multiply-accumulate, etc. These VPBs reduce the total power consumption, improve the performance, and reduce the size of the configuration data and therefore the configuration time. Typical applications are digital signal processing algorithms.

The goal of the methodology is that the VPBs implement as much operations of the applications as possible. The remainder is implemented using the fine-grained FPGA fabric. This project is similar to the pre-silicon design phase methodology suggested in this thesis, because the types of the VPBs should be adapted to an application domain. In the post-silicon design phase, an actual application should be mapped to the reconfigurable SPS.

In [KKOMB02] an algorithm is presented to automatically generate types of VPBs and to map applications to these VPBs (misleadingly called "instructions", "templates", "macros", "clusters" and "patterns" interchangeably). An application written in C, C++, Fortran or SystemC is compiled and converted to DFGs. These DFGs are processed to find frequent patterns. First, a list of all edge types is generated. Edge types are defined as the node types at its head and tail, e.g., a multiplier followed by an adder "$* \rightarrow +$". Then, all occurrences of the most frequent edge types are replaced by new "super-nodes". This clustering of edges is repeated until enough "super-node" types were generated or until a sufficient portion of the DFG is represented by "super-nodes". Note that each "super-node" is specified by its internal DFG, which is the section of the original DFG replaced by this "super-node". An improved algorithm to optimize the DFG coverage was investigated in [BMKS02]. The "super-nodes" are candidates for implementation as VPBs. Note that the algorithm achieves two goals simultaneously: the application DFGs are mapped to the VPBs and concurrently the required VPB types are automatically derived.

The general approach of SPS is to embed custom coarse-grained FUs (VPBs) in a fine-grained FPGA fabric. The surrounding FPGA fabric is used for routing as well as to implement functionality which can not be mapped to the VPBs. However, the fine-grained implementation causes high power consumption, therefore in this thesis, the whole applications should be mapped to coarse-grained FUs with no residual logic.

The algorithms automatically extract coarse-grained FUs. Since the clustering of nodes is only guided by their relative frequency and by the cost function, their boundaries and functionality are random. In other words, the usability of the "super-nodes" is limited to the context, i.e., the location in the DFGs. This also limits the reusability between different applications.

[KKOMB02] observed that simple VPBs with only two DFG nodes achieve sufficient quality of results. However, the total DFG is then split in many simple VPBs which requires a large amount of routing resources. This results in higher power consumption [KR98] (cf. Sec. 2.3.3) and area overhead than using more complex coarse-grained FUs.

From these reasons follows, that the best power and area efficiency is provided when the whole netlist is implemented by coarse-grained FUs. To allow reuse between applications the FUs should be designed manually, but the applications should be mapped automatically to these FUs.

2.5.4 The Totem Project

The *Totem Project* was developed at the University of Washington [Hau05, HCE+06]. It is also dedicated to the wide application domain of data processing and creates a stand-alone architecture not embedded in an FPGA fabric. Contrary to SPS, netlists with instances of arithmetic operators instead of DFGs are used to specify the functionality of the reconfigurable architecture.

The Totem Project is based on the "Reconfigurable Pipelined Datapath" (RaPiD) architecture [ECF96, CFBE98, CFF+99] which is a predefined coarse-grained reconfigurable linear 1D array of 16-bit functional units (e.g., ALUs, multipliers, and registers) and interconnect. The architecture is designed for deeply pipelined datapaths of digital signal processing applications.

The RaPiD architecture is predefined and fixed, i.e., the user only works in the post-silicon design phase. The Totem Project adds the pre-silicon design phase to optimize the cells, their arrangement and the routing of the architecture, depending on the narrow application domain. For that purpose, three topics were investigated: automated generation of a reconfigurable architecture, automatic generation of a chip layout of that reconfigurable architecture, and tools to map an actual application onto the reconfigurable architecture. The first two topics correspond to the pre-silicon design phase as defined in Sec. 1.1.5 while the third topic corresponds to the post-silicon design phase. Here only the architecture generation will be discussed in detail.

The automatic architecture generation uses a set of RaPiD netlists to specify the functionality of the reconfigurable architecture, i.e., which applications have to be supported. These netlists are generated from a high-level language and instantiate and connect RaPiD functional units. From these netlists the reconfigurable circuit is derived.

Two different kinds of reconfigurable architectures are generated. The first kind of reconfigurable architecture (termed "configurable ASIC", cASIC) is limited to the netlists which were used as specification [CH07]. The circuits are highly optimized to only implement MUXes to switch between the specification netlists. Therefore the cASIC approach will not be discussed further. The second kind of reconfigurable architecture is more flexible and termed "RaPiD-like". The methodology for both kinds (mostly) only differs in the generation of the interconnect [CH01, Com03].

As first step of the placement phase the required number of functional units of each type is determined. The user can specify constraints to include additional FUs to increase the flexibility of the resulting reconfigurable architecture. Secondly, the order of the FUs is optimized using simulated annealing. The placement of the FUs along the horizontal axis inter-depends with the binding of the netlist instances to the physical instances. Therefore the physical placement as well as the binding are moved during the simulated annealing [CH01, Com03].

While the functionality of the cASIC architectures is limited to the specification netlists, the RaPiD-like architectures provides more flexibility, for example to fix bugs or implement new applications. The RaPiD-like routing consists of horizontal wires below the FUs, which are connected with vertical wires to the inputs and outputs of the FUs. These tracks are generated iteratively: One track is added and all netlists are placed and routed. From this intermediate result, the number of unroutable signals is determined and used as a cost function of the solution. The optimization terminates, when all netlists can be routed. The user can specify additional routing tracks to increase the post-silicon flexibility [CSPH02, Com03, CH08].

The second topic investigated by the Totem project is the automatic generation of VLSI (very-large-scale integration) layout of the reconfigurable circuits. Three different approaches were

employed: logic synthesis using standard cells, template reduction and circuit generation. For the semi-custom standard cell approach, the designs were generated using Verilog HDL. Commercial logic synthesis and chip layout tools were used to generate the chip [PH02, Phi04]. The other two approaches create highly process dependent layouts and are therefore not further discussed here.

The RaPiD-style as well as the cASIC architectures are tailored to a narrow application domain as specified by a set of netlists. However, these netlists are limited to the wide application domain of coarse-grained signal processing with highly pipelined and regular algorithms. In the Totem project, also fine-grained reconfigurable architectures for control-dominated logic were investigated. The first approach were domain specific CPLDs (complex programmable logic devices) [HH04, HH05, HH07]. Flexible architectures using a full crossbar interconnect as well as tightly tailored architectures with reduced programming points and a sparse crossbar were designed. Secondly, the Totem project also planned the development of application domain specific 2D island-style FPGA architectures ("Totem^2") [Hau05, HCE+06, EHS05], but this area was not further investigated [Hau15].

The Totem project developed both, coarse-grained and fine-grained application domain specific reconfigurable architectures. For both types, a range from architectures tightly tailored to the required functionality up to more flexible architectures and even specifying additional flexibility are provided. The functionality is specified using a set of netlists instead of DFGs. The collection of multiple nodes into bigger "super-nodes" as for the SPS is not required here because the netlists are already generated from a high-level language using HLS, and therefore already instantiate the final node types.

The algorithms presented for (pre-silicon) architecture generation automatically merge multiple netlists into one reconfigurable architecture. The algorithms also allow to specify additional flexibility by increasing the number of functional units and enhancing the interconnect. Interestingly, using a generic interconnect topology (1D bus style for RaPiD and crossbar for CPLDs) itself ensures a certain degree of flexibility over tailored multiplexers and demultiplexers. Although the Totem project provides results for coarse-grained as well as fine-grained reconfigurable architectures, these are separate. The approach proposed in this thesis has to combine both for multi-granular reconfigurable circuits.

2.5.5 Application-Specific Inflexible FPGA and Multi-Mode ASIC

In the previous section, cASIC circuits could only implement the netlists which were used to specify its functionality without any additional flexibility. This kind of reconfigurable circuits is also termed "*multi-mode systems*". One problem common to multi-mode systems and more flexible application domain specific reconfigurable logic circuits is how to merge the netlists used for specification.

[PMKM11] use a set of fine-grained logic netlists which are mapped to a simple and universal FPGA architecture with CLBs, hard blocks like adders or multipliers, and IOs. To merge the specification netlists, the placement algorithm tries to match instances between the netlists and place them on the same physical instance.

In the routing phase, efficient wire sharing is used, so that the number of switches is minimized. Then all unused FUs and routing resources are removed and a VHDL description of the "*Application-Specific Inflexible FPGA*" (ASIF) is generated. Additionally, the bitstreams to configure the ASIF for each specification netlist are generated. This top-down-removal approach

leaves some degree of flexibility, but due to the irregular architecture, place and route algorithms are complex and require long run-times.

Since the remaining flexibility is not efficiently utilizable, the authors moved that approach further to pure *multi-mode ASICs* (mASICs) [KMM14]. The configuration data of the depopulated FPGA architecture is hard-coded using mode (i.e., netlist) select input signals, constants and multiplexers. Afterwards a commercial ASIC synthesis tool is employed to perform constant propagation and logic optimization. This removes all configurable elements and implements efficient logic functions.

For the verification of the results, *logical equivalence checking* is used to compare each specification netlist with the mASIC. In the setup of the equivalence checking, the selection inputs are forced to the appropriate bit pattern to select the according mode.

In principle, this approach only uses the fine-grained FPGA architecture as a platform to find common resources of the specification netlists with the mentioned place and route algorithms. The result is an optimized logic circuit which can switch between the specification netlists using shared resources.

In this thesis, a joint methodology to merge netlists with coarse-grained as well as fine-grained logic is required. The ASIF/mASIC methodology is also applicable for coarse-grained logic circuits, but can not tap the full potential to identify commonalities and wide vector signals. Furthermore, the reduced FPGA fabric is very inflexible and not suitable for the approach of this thesis.

2.5.6 Custom Architecture Design Tool

A joint methodology for coarse-grained and fine-grained reconfigurable logic termed "*Custom Architecture Design Tool*" (CustArD) was shown by [BS14]. They propose to build a reconfigurable circuit as a hierarchy of function blocks. For example, at the lowest level a CLB with a 4-input LUT, a D-FF and a bypass-MUX is defined. At the next level, this CLB is combined with routing channels and connection boxes. This structure is repeated in a 12×12 grid at the next level. Then two such grids are combined with a memory, a DSP slice, and further routing channels. The whole reconfigurable circuit is composed of a 2×2 grid of that structure and finally supplemented with IO cells.

To specify such architectures a visual architecture design tool was implemented. The hierarchical structure poses special requirements for the tools. To efficiently realize coarse-grained cells and routing, the synthesis tool has to preserve vector signals. The place and route tool has to cope with the hierarchical routing structures. To optimize the architecture to a given application domain, an analysis tool is required, which implements several netlists onto the specified architecture and provides statistical evaluation data. Compared to the previously reviewed approaches, this methodology is neither a bottom-up-creation approach from netlists, nor a top-down-removal approach. It is comparable to the KressArray family [Har01a] where a preliminary architecture is setup, analyzed by mapping applications, and then iteratively improved. This work is still at an early stage of development. The authors report findings on several approaches for open problems. Future work will put more emphasis on coarse-grained reconfigurable architectures, e.g., including a configurable ALU similar to the mentioned DSP slices [Bos15].

The CustArD architecture combines fine-grained and coarse-grained reconfigurable logic. The requirements for the synthesis tool to preserve vector signals is relevant for the approach proposed

in this thesis too. However, the CustArD methodology results in very flexible architectures which are only slightly tailored to the application domain. Therefore the area and delay overhead will be higher than applicable for the approach proposed in this thesis.

2.5.7 Summary

In this section application domain specific reconfigurable CPU supplement modules were reviewed. The design methodologies include the pre-silicon design phase to optimize the architecture to a narrow application domain and the post-silicon design phase to map applications to the generated architecture.

The KressArray is a 2D array of coarse-grained DPUs (ALU and control unit). The pre-silicon methodology maps the most complex example netlist to a preliminary architecture. This is analyzed and improved in an iterative process. The resulting architecture is verified to map all other netlists. The Pleiades architecture consists of a CPU with a heterogeneous set of coarse-grained satellite processors. The methodology identifies compute intensive kernels of an application program to optimize this architecture. These kernels are manually mapped to tailored satellites. Finally the interconnect is optimized.

The Strategically Programmable System is based on a 2D island-style mesh FPGA fabric. The fabric includes coarse-grained hard-wired blocks (VPBs). The methodology includes algorithms to automatically combine nodes of DFGs which occur frequently in a pattern. These are implement as VPBs. The Totem architecture is based on the 1D pipelined RaPiD architecture. The individual FUs, their order and the interconnect are optimized by mapping RaPiD netlists and using simulated annealing. For control-dominated applications fine-grained auto-generated optimized CPLDs were developed.

The ASIF/mASIC approach for fine-grained logic uses efficient place and route algorithms to an FPGA fabric. The CustArD project proposes a combined reconfigurable architecture for fine- and coarse-grained logic as a hierarchical circuit with fine-grained CLBs and coarse-grained blocks.

Although none of the reviewed approaches fulfills all given requirements in Sec. 1.1.7, useful details can be learned from the reviewed works. Especially the employed pre-silicon methodologies to develop the reconfigurable architectures provide a valuable basis for this thesis. In the previous subsections, notable conclusions from each reviewed approach individually were discussed. Here inferences from the combination of the approaches are drawn and the most important conclusions are repeated.

- Most work in this area was done for high performance data processing.
- All reviewed works use a set of similar applications to specify the functionality. This fulfills the requirement to specify the flexibility in terms of accomplishable functionality (cf. Sec. 1.1.7).
- The reviewed methodologies use three steps to generate a reconfigurable architecture:
  1. HW/SW partitioning to generate a set of applications,
  2. convert the set of applications to DFGs or netlists, and
  3. merge the DFGs or netlists to a reconfigurable architecture.

- The pre-silicon design methodology must be independent of the narrow application domain. All reviewed approaches comply to this requirement, but most are limited to the *wide* application domain of data processing due to the use of DFGs.
- A part of the approaches for coarse-grained data processing applications compile descriptions in high-level language to generate DFGs. Some approaches further synthesize the DFGs to netlists of FUs. Fine-grained approaches directly use netlists as representation of the applications.
- In the optimization of the KressArray family, only the most complex example application is used. All other approaches use the full set of example applications concurrently which provides improved optimization results and avoids adverse solutions for unused example applications.
- All architectures use a set of reconfigurable functional units and a reconfigurable interconnect. In most approaches, the reconfigurable FUs are heterogeneous, i.e., each FU instance implements a different functionality.
- The SPS architecture uses a fine-grained FPGA fabric for the routing and to implement new functionality. This causes a large overhead in area and power. The same problem exists for the CustArD approach.
- The SPS methodology automatically combines DFG nodes to "super-nodes". In this thesis the "super-nodes" should be defined manually to better support the reuse across netlists.
- The Totem project increases the flexibility of the reconfigurable circuit by including more FUs and more routing resources than required to implement only the specification netlists. It also reveals that the use of a generic interconnect topology instead of MUXes or depopulated FPGAs/CPLDs itself maintains a degree of flexibility.
- The Totem project developed three methods for automatic layout generation. Two of these strongly depend on the semiconductor technology. Only the method using logic synthesis to a standard cell library is independent of the semiconductor process. However this causes an area, delay and power penalty.

All reviewed works from Sec. 2.3, Sec. 2.4, and this section and their properties as defined in Sec. 2.2 are summarized in Tab. 2.2.

2.6 Reconfigurable FSM Architectures

The narrow application domains realized with the proposed methodology contain control-dominated and data processing elements. To combine these domains, [GR94] suggested the generic structure of an FSM which controls a datapath (FSM+D). In this thesis, this approach is extended to also use the FSMs for stand-alone control-dominated tasks. Additionally, multiple communicating FSMs can be utilized to implement hierarchical functionality. For the proposed reconfigurable CPU supplement modules, reconfigurable architectures for FSMs are investigated.

An FSM implemented in digital logic is defined by its n_I input signals I, n_O output signals O, n_S states $S = \{S_1, S_2, \ldots, S_{n_\mathrm{S}}\}$, and the transitions. The output signals depend on the input signals and the current state. The input signals and the current state determine the next state.

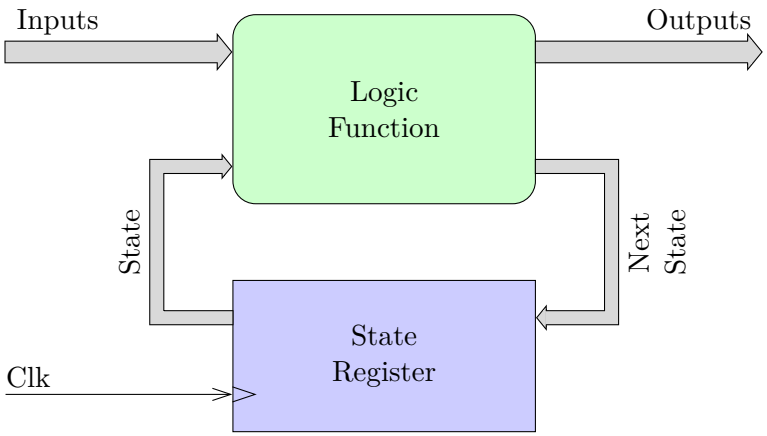
Table 2.2: Classification of the approaches for application domain specific and for reconfigurable CPU supplement modules and methodologies. (n.a.: not applicable, GUI: graphical user interface)

Architecture	Phases	Flexibility	Granularity	Topology	Wide App. Dom.	Optimization	Configuration	Coupling	Autonomy	Design Entry
Commercial Reconfigurable CPU Supplement Modules										
Microchip CLC	Post-Si.	Full	Fine	Crossbar	Bit-Level	Flexibility	Reconfig.	Peripheral	Full	GUI, Regs.
Analog Devices PLA	Post-Si.	Full	Fine	Red. MUX	Bit-Level	Flexibility	Reconfig.	Peripheral	Full	GUI, Regs.
Energy Micro LESENSE	Post-Si.	App. Dom.	Mixed	Custom	Sensor Intf.	Power	Reconfig.	Peripheral	Full	GUI, Regs.
Cypress PSoC 2nd Gen.	Post-Si.	App. Dom.	Coarse	Custom	Peripherals	Flexibility	Config./Dyn.	Peripheral	Full	GUI, Libs., Regs.
Texas Instr. CLA	Post-Si.	n.a.	Coarse	n.a.	Control&Data	Perf./Flex.	Reconfig.	Attached	Full	Assembler, C
Commercial Embedded FPGAs										
Xilinx Virtex II Pro	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	Attached	User-def.	HDL
Xilinx Virtex 4 FX	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	Attached	User-def.	HDL
Xilinx Zynq-7000	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	Attached	User-def.	HDL
Xilinx UltraScale+	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	Attached	User-def.	HDL
Altera SoC FPGAs	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	Attached	User-def.	HDL
NanoXplore NX-eFPGA	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	User-def.	User-def.	HDL
ADICSYS efpga	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	User-def.	User-def.	HDL
Menta eFPGA Core IP	Post-Si.	Full	Mixed	Mesh	User-def.	Perf./Flex.	(Re)Config.	User-def.	User-def.	HDL
Research on Embedded FPGAs										
Triptych FPGA	Pre-Si.	Full	Fine	Mesh	User-def.	Area	unknown	User-def.	User-def.	unknown
UC Berkeley [KR98]	Pre-Si.	Full	Fine	Mesh	User-def.	Power	unknown	User-def.	User-def.	unknown
UC Berkeley [GZR99, ZPG+00]	Pre-Si.	Full	Fine	Mesh	User-def.	Power	unknown	User-def.	User-def.	unknown
U. British Col. [WKW+05, WAWS03]	Pre-Si.	Full	Fine	Mesh	User-def.	Synthesis	unknown	User-def.	User-def.	unknown
U. British Col. [YW04]	Pre-Si.	Full	Fine	Mesh	User-def.	Seq. Logic	unknown	User-def.	User-def.	unknown
U. British Col. [ALS05]	Pre-Si.	Full	Fine	Mesh	User-def.	Area/Perf.	unknown	User-def.	User-def.	unknown
XiSystem PiCoGA [CTL+03]	Pre-Si.	Full	Coarse	Mesh	Data Proc.	Performance	Multi-Context	ISA Ext.	ISA Ext.	C
XiSystem eFPGA [LCB+06]	Pre-Si.	Full	Fine	Mesh	Control Log.	Performance	Reconfig.	Peripheral	Full	HDL
Application Specific Non-Reconfigurable CPU Supplement Modules										
Nymble [HLOK13]	n.a.	n.a.	n.a.	n.a.	Data Proc.	Performance	n.a.	Peripheral	Accelerator	C/C++
NEC C&C [Hen99]	(ASIC)	(ASIC)	n.a.	n.a.	Data Proc.	Power	none (ASIC)	Peripheral	Accelerator	C
UC Riverside [SV02]	Post-Si.	n.a.	Fine	Mesh	Data Proc.	Power	n.a.	Peripheral	Accelerator	C & VHDL
Application Domain Specific Reconfigurable CPU Supplement Modules										
KressArray	Pre+Post	App. Dom.	Coarse	Mesh	Data Proc.	Perf./Flex.	Dynamic	External	Accelerator	C
Pleiades	Pre+Post	App. Dom.	Coarse	Hier. Mesh	Data Proc.	Power	Dynamic	Peripheral	Accelerator	C
SPS	Pre+Post	App. Dom.	Coarse	Mesh	Data Proc.	Perf./Flex.	Reconfig.	undef.	Accelerator	C/C++/Fortan
RaPiD	Post-Si.	App. Dom.	Coarse	Linear	Data Proc.	Performance	Reconfig.	“closely”	Accelerator	manual
Totem	Pre+Post	App. Dom.	Coarse	Linear	Data Proc.	Area/Perf.	Reconfig.	undef.	Accelerator	RaPiD-C
Totem CPLD	Pre+Post	App. Dom.	Fine	Crossbar	Bit-Level	Area/Perf.	Reconfig.	undef.	Full	HDL, BLIF
Totem FPGA	Pre+Post	App. Dom.	Fine	Mesh	Bit-Level	Area	undef.	undef.	Full	undef.
ASIF [PMKM11]	Pre+Post	App. Dom.	Fine	red. Mesh	User-def.	Area	Reconfig.	User-def.	User-def.	HDL
mASIC [KMM14]	Pre+Post	Multi-Mode	Fine	red. Mesh	User-def.	Area	Multi-Mode	User-def.	User-def.	HDL
CustArD [BS14]	Pre+Post	App. Dom.	Mixed	Hierarch.	User-def.	Flexibility	undef.	User-def.	User-def.	GUI & HDL

At the clock edge the next state is activated as current state. Several different digital encodings for the state value are employed, e.g., one-hot (requires $n_{S,1h} = n_S$ signals), binary (requires $n_{S,b} = \lceil \log_2 n_S \rceil$ signals), and Gray coded (also requires $n_{S,g} = \lceil \log_2 n_S \rceil$ signals).

A generic FSM as a synchronous sequential system is shown in Fig. 2.3 [Kae08, RCN03, Kat94]. The "Logic Function" block only contains combinational logic and implements two logic functions: the output function $O = F(I, S)$ and the next state function $S_{next} = G(I, S)$. The "State Register" introduces sequential behavior.

Figure 2.3: Generic FSM structure



Three types of FSMs were defined [Kae08]:

Mealy Type: The output vector is a combinational function of the current state and the inputs.

Moore Type: The output vector is a combinational function of the only the current state.

Medvedev Type: The output vector is exactly the current state vector.

To implement an FSM as a hard-wired digital circuit, the logical synthesis uses logic optimization algorithms to map the combinational logic functions to a logic circuit. Another solution is the implementation with a ROM (read-only memory) [Kat94]. The FSM input signals and the state vector are used as address inputs of the ROM. The data output signals of the ROM are used as FSM output signals and next state vector. This requires $n_I + n_{S,b}$ address signals and $n_{S,b} + n_O$ data bits, hence a total capacity of $(n_{S,b} + n_O) \cdot 2^{n_I + n_{S,b}}$ bits. This shows that the ROM size grows exponentially with the number of input signals and state signals.

Replacing the ROM with a RAM is an approach for a reconfigurable FSM. However, due to the exponential growth, this is an inefficient solution. Additionally, for typical FSMs the state transitions only depend on a subset of the input signals. Logical optimization algorithms for hard-wired FSMs can utilize the information on unobserved signals ("don't care"), but in a ROM and RAM implementation this causes large overhead.

A more efficient reconfigurable implementation of FSMs uses PLAs [RCN03, Kat94]. Each product term has to combine $n_i + n_{S,b}$ (or even less) signals and a total of $n_{S,b} + n_O$ sum terms is required. Besides PLAs, also FPGAs can efficiently implement the next state and the output logical functions [RCN03]. The same logic optimization algorithms as used for hard-wired FSMs

reduce the total amount of logic resources required, which is then mapped to the reconfigurable LUTs and routing resources. For example, [WAWS03] uses an 8×8 embedded FPGA fabric with 3-input LUTs (reviewed in Sec. 2.3.2) to implement the next-state logic of an FSM. Due to the high number of D-FFs for the configuration data of the reconfigurable eFPGA, 45% of the area had to be reserved for the buffers of the clock tree and power supply, compared to 25% for non-reconfigurable logic.

[Buk08, Buk09] reports that FSM implementations using FPGAs are suboptimal. He proposes several methods for architectural decomposition of the FSM logic. These include the conversion of single-level implementations of boolean functions to multi-level implementations and the utilization of the block RAMs available in FPGAs. His final results show only little improvement for actual FPGA implementations of FSMs.

A dedicated architecture for reconfigurable FSMs was presented by [LAKL05]. It uses sequential blocks and logic blocks to implement the next state function and the output function, respectively. These are connected with routing resources. Each logic block contains three levels of PLA blocks and internal interconnect. Each sequential block has an adder and subtracter (to calculate the new state vector) and D-FFs. The authors report a reduction of the area by 43% and of the power consumption by 82% compared to the commercial Virtex-E FPGA. In [LAE06] this architecture was optimized to efficiently support a higher number of states. This approach implements the FSM next state function and the output function using generic logic functions. Together with the reconfigurable interconnect between the sequential and logic blocks this requires large chip area.

An area optimized approach using dynamic reconfiguration of the next-state logic function was presented by [MV07]. Only the functionality for transitions from the current state is implemented in the reconfigurable logic. In other words, the current configuration represents the current state. The authors compared the number of LUTs required to implement the dynamically reconfigured next state logic function for each state with the number of LUTs required to implement the whole FSM next state function and report a reduction of 90%. However, the constant reconfiguration causes high switching activity. Therefore this approach is not suitable for low-power applications.

A power optimized reconfigurable FSM architecture dedicated to WSN nodes was presented by [RV13]. Their general approach is the same as in this thesis: A dedicated CPU supplement module off-loads the CPU, which in turn stays in an in-active low-power mode for longer periods. Compared to the generic FSM model presented above, the “Configurable FSM” (CFSM) adds extensions. For each state, a default next state is defined to reduce the number of total transitions. The CFSM also contains an integrated timer to simplify the implementation of delays and timeouts. Internally four look-up tables map the current state to its default next state, to the non-default transitions, to the output signals and to the counter start value. The authors report an energy reduction of 46% for the whole WSN node SoC when using the CFSM CPU supplement module to performing event related tasks instead of a pure firmware implementation. The extensions for default next states and timers improve the chip area and the usability. Using lookup tables, which grow exponentially with the number of state signals, considerably increases the total chip area. Further, the presented CFSM implements a Moore type FSM, i.e., the output signals only depend on the current state and don’t consider the input signal values. Viewed as a reconfigurable CPU supplement module, this approach lacks further data processing capability, e.g., for sensor value post-processing, and a full pre-silicon design methodology.

In this section, approaches for reconfigurable FSM architectures were reviewed. Implementations using a RAM cause a large area overhead. PLA and FPGA implementations reduce the area but introduce new overhead due to the fine-grained architecture, the storage of the configuration data, and the large interconnect. The approaches presented by [LAKL05] and [RV13] also require large chip area due to the reconfigurable interconnect and the employed lookup-tables, respectively. The dynamic reconfiguration utilized by [MV07] shows a considerable reduction of chip area, but creates high power consumption due to the switching activity caused by the reconfiguration.

2.7 Conclusion

The reviewed works show, that the inclusion of reconfigurable application domain specific CPU supplement modules is a viable approach for energy reduction. The development of a generic methodology for the design of application domain specific reconfigurable logic is a challenging task. The methodology must be independent of the narrow application domain, therefore no assumptions on the actual scenario or the implementation can be made.

In the previous sections a number of approaches for a generic methodology were reviewed. Most approaches are limited to data processing applications and do not cover control-dominated applications. However, the principles employed by the authors provide a foundation for this work. The basic architecture is defined as a collection of reconfigurable functional units connected with an optimized reconfigurable interconnect. The specification of the functionality of a reconfigurable module is defined by a set of applications. These applications are translated to netlists of functional units. All netlists are used together to optimize the reconfigurable module, which is built using a bottom-up-creation approach. Including additional functional units and additional routing resources is used to increase the flexibility.

Besides these principles, solutions for open problems have to be found. The methodology and the designed reconfigurable modules have to explicitly support both, control-dominated and data processing applications. Therefore the architecture must provide multi-granular logic. The user designs the reconfigurable functional units manually, but an automated procedure is required to identify the functional units in the specification applications during the translation to netlists. As a special functional unit, an architecture for reconfigurable FSMs with low power consumption and small chip area has to be developed.

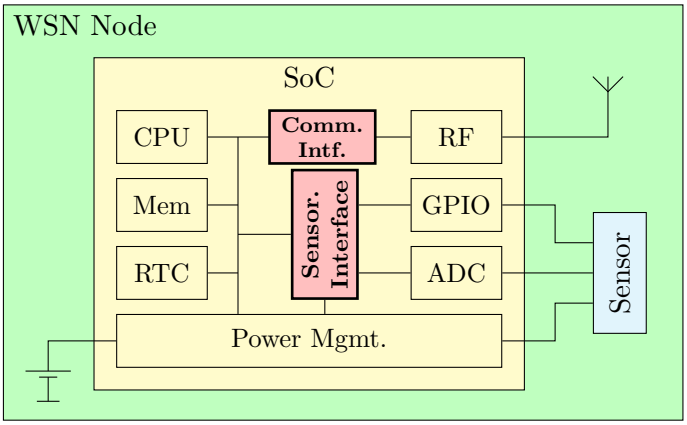
In the remainder of this thesis, a generic methodology for the design of application domain specific reconfigurable CPU supplement modules which fulfills the defined requirements will be developed by combining the basic principles from the related work and contributing solutions for the open problems.

3

Design Methodology

In this thesis an approach for the reduction of the power consumption of ultra-low-power embedded systems by the inclusion of dedicated reconfigurable CPU supplement modules is proposed (cf. Sec. 1.1.3 and Fig. 3.1). These modules off-load the CPU from simple and recurring tasks and therefore act as a filter for events. Only events which require complex processing are forwarded to the CPU (cf. Sec. 1.1.2).

Figure 3.1: WSN node with reconfigurable CPU supplement modules, repeated from Fig. 1.2.



The reconfigurable CPU supplement modules are tailored to a specific application domain and implemented with a heterogeneous and multi-granular reconfigurable architecture (cf. Sec. 1.1.4). The design of such domain-specific reconfigurable logic modules requires a special methodology which is developed in this chapter.

In Ch. 2 the scientific basis was established. The found approaches and concepts are evaluated, integrated, and extended to form a complete design methodology. First in Sec. 3.1 the general principles for the design methodology are arranged. Further details on the generation of a reconfigurable module are discussed in detail in Secs. 3.2, 3.3, and 3.4. Finally, in Sec. 3.5 approaches for the verification of the reconfigurable module at each point of its development are developed.

As stated in Sec. 1.1.5 and depicted in Fig. 1.6 on p. 7 the development and usage of reconfigurable logic is split in the pre-silicon and the post-silicon design phase. Sections 3.1–3.5 are related to the pre-silicon design phase. The post-silicon design phase is discussed in Sec. 3.6.

One important design element in this work are FSMs. A reconfigurable architecture for FSMs is introduced in Sec. 3.7. Finally, the scientific contributions of this thesis are summarized in Sec. 3.8.

3.1 Principle

In this section general principles of the design methodology for reconfigurable modules are established. As stated in Sec. 1.1.5, a demand for a reconfigurable CPU supplement module is the starting point of the design methodology (cf. Fig. 1.5). Therefore the principles for the precise specification of the reconfigurable module are discussed in Sec. 3.1.1. The resulting module is delivered as a soft IP core [GZ97]. The exact deliverables are defined in Sec. 3.1.2.

To separate the functionality of the reconfigurable module from adjustable values and the processed data, the concept of parameterization is introduced in Sec. 3.1.3. In Sec. 3.1.4 the principles of the reconfigurable architecture are refined before the actual generation of the reconfigurable module from the given specification is discussed in Sec. 3.1.5.

To enable the implementation of new applications which were not anticipated during the development of the reconfigurable module, means to increase the flexibility are discussed in Sec. 3.1.6. Finally, in Sec. 3.1.7 the main principles are summarized.

Note that the motivation for this work is the development of reconfigurable CPU supplement modules. However, the design methodology is more general, therefore the term “reconfigurable module” is used starting from here. If an explicit relation to a CPU is given, “reconfigurable CPU supplement module” is used further on.

As defined in Sec. 2.4.3, the design methodology must be independent of the actual application domain. This implies, that the following discussion as well as the design methodology itself can not base on any assumptions regarding the application domain. However, the discussed principles will be illustrated by examples below, but without an influence on the generality of the statements.

3.1.1 Specification

In this section, a suitable format to specify the reconfigurable module is discussed. Generally, there are the following methods to specify a reconfigurable architecture:

a) Specify the available resources provided by the reconfigurable architecture.

b) Use a set of applications, i.e., logic designs, which have to be supported by the reconfigurable architecture [GW14].

c) Describe the functionality of the reconfigurable architecture in abstract terms [GW14].

Ad a) Commercial FPGAs are specified in terms of *available resources*. For example the Xilinx Zynq XC7Z020-CLG484 device (reviewed in Sec. 2.3.2) contains 85k programmable logic cells with 53,200 6-input LUTs, twice as many D-FFs, and provides 200 IOs [Xil14d]. In its datasheet also the routing resources are specified.

The embedded FPGA of the Maia chip [ZPG+00] (cf. Sec. 2.5.2) provides 4×8 logic blocks with 3-input LUTs and a three level hierarchical interconnect [GZR99] (cf. Sec. 2.3.3). This format is also used to specify coarse-grained architectures, e.g., the original 2D rDPA (later renamed to KressArray) with rDPUs [HK95, HBH+96, HHHN98] (cf. Sec. 2.5.1), the 1D RaPiD architecture with a defined set of FUs [ECF96] (cf. Sec. 2.5.4), and the CustArD architecture using a hierarchical specification [BS14] (cf. Sec. 2.5.6).

Ad b) As found in Sec. 2.5.7, most application domain specific reconfigurable architectures were specified using a *set of applications* which span the total range of functionality. For example, the KressArray family of optimized coarse-grained reconfigurable architectures uses a set of applications [HHHN00a, HHHN00b, HHHN00c, HHN00, Nag01] (cf. Sec. 2.5.1). The Pleiades project uses the netlists of the computation intensive kernels of a single application program to specify the satellite processors and the interconnect [WZG+01, Wan01] (cf. Sec. 2.5.2). Contrary to the RaPiD project, the Totem project optimized the 1D architecture, also specified by a set of netlists [CH08] (cf. Sec. 2.5.4).

Ad c) Describing the functionality in *abstract terms* closely resembles the nature of (application domain specific) reconfigurable architectures which provide wide and intangible but limited possibilities. However, this format poses challenges, once how to formulate the specification, and secondly how to verify the compliance to the specification. The specification could include, for example, the number of examined conditions, the number of observable signals per condition, the type of comparisons between numbers, and the type of operations to perform. It further has to declare the relations between these statements

Only b), i.e., using a set of applications to specify the reconfigurable architecture fulfills the requirement of a specification in terms of possibilities and accomplishable functionality (cf. Sec. 1.1.7) and provides the possibility to verify the compliance of the architecture. Therefore, in the methodology discussed in this thesis, this option is used.

The actual range of accomplishable functionality of a reconfigurable module is termed "*application class*" in the remainder of this work. This term is analogous to the narrow application domain used in Sec. 2.2. Each application of the set to specify the functionality of the reconfigurable logic module is termed "*example application*". The term "*new application*" is used for applications introduced in the post-silicon design phase. The superset of example applications and new applications is simply termed "*applications*" or "*(example) applications*" for a special emphasis on both phases.

Next the exact format for (example) applications is derived. When using automatic HW/SW partitioning, the whole application must be implemented in one common language. The design is analyzed and split in hardware and in software partitions. On the other hand, as concluded in Sec. 2.4.3, for the methodology discussed in this thesis the HW/SW partitioning is performed manually by the developer. This implies, that different languages for the software (i.e., MCU firmware) and the hardware partitions (i.e., example and new applications) can be employed. The MCU firmware will most probably be programmed using the high-level language C.

The exact format how to specify the (example) applications of the reconfigurable module has to fulfill several requirements (extended from [GW14]):

- It must support the development of control-dominated as well as data processing tasks.
- It must allow to specify cycle accurate timing to control and to react on other cores (e.g., peripherals) of the SoC.

→ Therefore, in Sec. 2.5.7 the use of DFGs was declined in favor of netlists.

- The format must be easy to learn or at best, not require additional skills of hardware designers.
- It must allow high productivity and should not be error-prone during the development of (example) applications.
- Therefore, the netlists should not be specified directly but in a format which is easy to translate to a netlist.

→ These requirements are best fulfilled by hardware description languages (HDLs).

- Further, the format must allow verification, starting from the initial (example) applications and at every step of the development of the reconfigurable module.
- It must allow early HW/SW co-simulation of the (example) applications together with the firmware and therefore allow early tests of the whole SoC.
- The format must allow the use of commercial as well as free and open-source tools for the design analysis and the design verification.

→ Therefore an existing type of HDL like VHDL and Verilog is used.

To summarize the above discussion, the methodology discussed in this thesis uses a set of example applications developed in VHDL or Verilog to specify the reconfigurable modules (see Fig. 3.2).

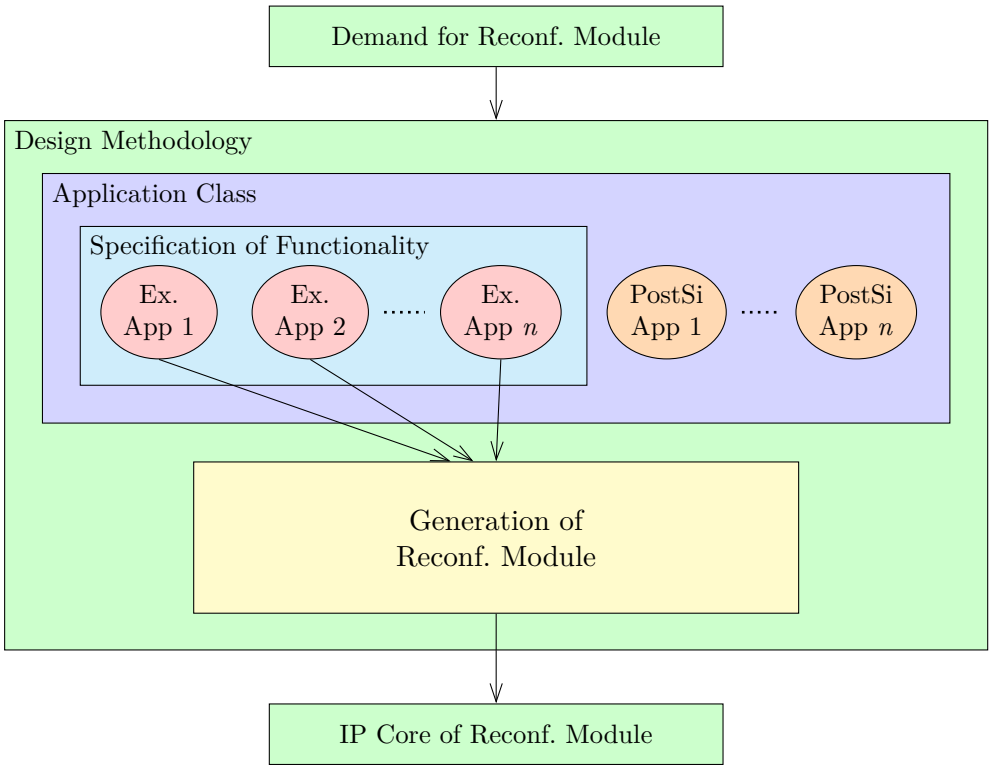
With the example applications, the functionality and therefore the resources of the reconfigurable module are specified. Additionally the input and output ports must be specified. A typical reconfigurable CPU supplement module might have connections to chip internal destinations (e.g., CPU interrupt request, control and data signals to an ADC or an SPI master) as well as to chip pins (e.g., to control external peripherals).

As all example application HDL designs also specify input and output ports, the common denominator of these could be used as the specification of the ports of the reconfigurable module. This would result in a set of universal input and output ports with no dedicated meaning of any port. However, the connections to chip internal peripherals have fixed destinations and therefore meanings and can not be shared between the example applications as universal ports. Therefore the input and output ports of the reconfigurable module are specified separately to reflect the actual requirements of the surrounding SoC. Additional meta information for each example application is used to specify the mapping of its ports to the ports of the reconfigurable module.

3.1.2 Deliverables

In the previous section the starting point of the design methodology was specified. In this section, the deliverables are discussed.

Figure 3.2: The total functionality of the reconfigurable module is denoted by the application class. It is specified using a set of example applications. Due to the flexibility of the reconfigurable module, new applications can be implemented in the post-silicon design phase.



The final reconfigurable module is provided as an IP core. This is integrated with all other modules in the SoC (cf. red CPU supplement modules in Fig. 3.1). Then the work is continued with the realization of the SoC using a standard ASIC flow (synthesis, place and route, etc.) To allow a seamless integration in standard ASIC design flows, the IP core must be independent of the semiconductor process. It must be compatible to commercial and free and open-source ASIC tools as well as to custom in-house design flows (cf. Sec. 1.1.7).

Therefore a *soft* IP core must be generated which is used for synthesis to a standard cell library (cf. soft eFPGAs in Sec. 2.3.4 [WKW+05, WAWS03] and the standard cell layout generation in Sec. 2.5.4 [PH02, Phi04]).

The soft IP core comprises of the following deliverables:

- The reconfigurable module as synthesizable RTL code in Verilog and VHDL.
- All meta-data required for integration of the IP core in the SoC and the ASIC design flow, e.g., synthesis-related scripts, constraints, etc.
- Templates and information to develop firmware drivers for the reconfigurable CPU supplement module.
- Information on the structure and inventory of the reconfigurable module used in the post-silicon design phase to generate the configuration data for new applications.

More details for the concrete realization are discussed in Sec. 4.8.

3.1.3 Configuration and Parameterization

For the configuration of the reconfigurable module, the principle of separation of functionality from data is applied. The functionality of the reconfigurable module is setup by *configuration*, e.g., using a bitstream which is shifted into a configuration chain. On the other hand, to setup data for processing by the reconfigurable module, the concept of *parameterization* is introduced [GHDG10]. Parameters are used to set constant values or to irregularly adjust values, e.g., by writing to memory-mapped registers. Parameterization is independent from configuration, is not integrated in the possibly large configuration data, and has a separate interface from the CPU. This allows the firmware to directly access each parameter and easily carry out changes without interruption of the operation or a full reconfiguration of the circuit.

The parameterization concept is further generalized to bidirectional data transfer, i.e., to return values from the reconfigurable module to the CPU. This means that parameters are either set by the CPU and used by the reconfigurable module, or the reconfigurable module sets parameter outputs and the CPU queries the values.

For example, the functionality of the sensor interface task (cf. Sec. 1.1.1) is split from the values of the period and delay times and from the actual threshold value used to compare the old and the new sensor value. To transfer the new sensor value to the CPU, a parameter output is used. Another example is a reconfigurable module which implements digital filters. The designer uses configuration to setup the topology of the arithmetic operations and connections for the data processing, while the filter coefficients are specified using parameterization. A third example is a reconfigurable module which handles network protocols. It would use configuration to implement the actual network protocol handling and parameterization to set the local address.

In the final SoC the firmware is responsible for the initialization and operation of the reconfigurable CPU supplement module. In a setup sequence, first the configuration data and then the parameterization data are applied. Finally, during the regular operation of the SoC, parameterization is used to adjust and query values of the configured application. In (example) application HDL designs, the parameters are specified as input and output ports.

3.1.4 Reconfigurable Architecture

The reconfigurable architecture has to support control-dominated as well as data processing tasks. Some of the platforms reviewed in Ch. 2 provide facilities for both domains by the inclusion of separate fine-grained and coarse-grained reconfigurable architectures. For example:

- The XiSystem SoC uses the PiCoGA and the eFPGA architectures (Sec. 2.3.3, [LCB+06]).
- The Maia processor of the Pleiades project uses coarse-grained satellites and a dedicated eFPGA (Sec. 2.5.2, [ZPG+00, GZR99]).
- The Totem project developed coarse-grained RaPiD style and fine-grained CPLD architectures (Sec. 2.5.4, [CH08, HH07]).

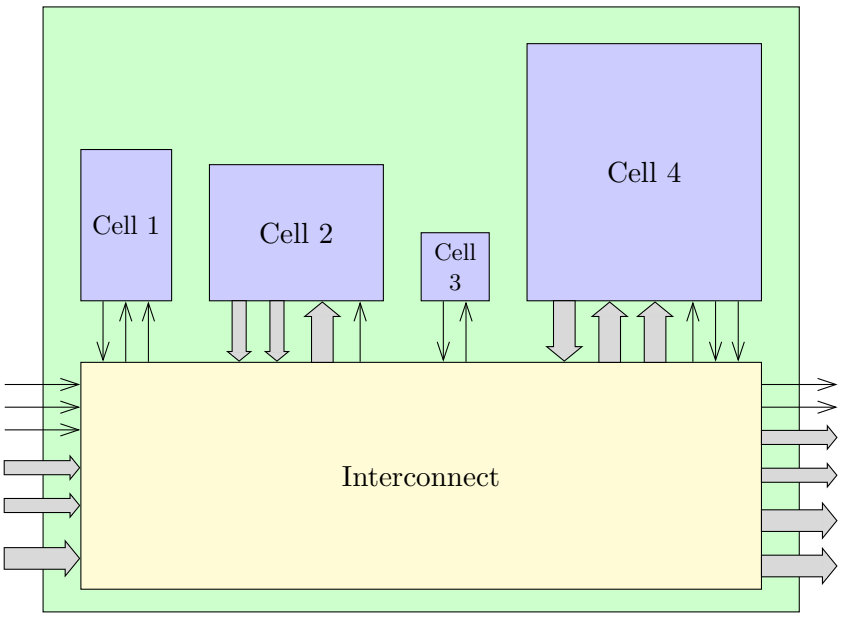
However, these two domains have interdependencies. For example the zero flag from a coarse-grained counter is used as an input of a fine-grained FSM, or the FSM controls a coarse-grained multiplexer of the datapath. Therefore in this work a *unified* multi-granular reconfigurable architecture is required (cf. Sec. 1.1.4 and 1.1.7).

As summarized in Sec. 2.5.7 the basic internal structure of the reviewed reconfigurable architectures use reconfigurable functional units and a reconfigurable interconnect. The same structure is used in FPGAs which have CLBs and a mesh interconnect [GCS+06]. This setup is used as basis for the reconfigurable architecture in this thesis. The internal structure is defined as a pool of FUs connected via a reconfigurable interconnect (see Fig. 3.3) [GHDG10, WGS+12].

In the remainder of this thesis, FUs of the reconfigurable module will be denoted "*cells*" while for general reconfigurable architectures the term "FU" will be used further on. Cells are instances of a "*cell type*". For example, the reconfigurable module can have two instances (cells) of an adder (cell type), among others. The applications which are implemented using the reconfigurable modules also instantiate the cell types, these logical instances are mapped to the physical instances of the reconfigurable module.

In general, the property of granularity applies to FUs as well as to signals of a reconfigurable architecture. Fine-grained FUs implement bit-level functionality (e.g., CLBs in FPGAs) while coarse-grained FUs implement complex functionality (e.g., DSP slice, FSM, CPU, etc.) Fine-grained signals conduct single bit values while coarse-grained signals conduct whole vectors of multiple bits. Fine-grained interconnects route single bit values and use individual switches for each bit, while coarse-grained interconnects route and switch vectors as a whole with an unchanging order of the individual bits. For example, FPGAs contain fine-grained CLBs and coarse-grained FUs (DSP, BlockRAM, etc.) but only provide a fine-grained mesh interconnect. On the other hand the KressArray, Pleiades, and RaPiD/Totem architectures only have coarse-grained FUs and a coarse-grained interconnect (cf. Tab. 2.2). To achieve a unified multi-granular reconfigurable architecture, in this work multi-granular cells as well as a multi-granular interconnect are used.

Figure 3.3: Schematic block diagram of a reconfigurable module: a pool of cells is connected via the interconnect.



Multi-Granular Cells

For a unified multi-granular reconfigurable architecture, fine-grained and coarse-grained cells are included. However, for best area and energy efficiency the designer should put an emphasis on coarse-grained cells (cf. Sec. 2.3.4).

As concluded in Sec. 2.5.3, the cells are designed manually by the developer using an HDL, analogous to the development of example applications. Each cell can implement

- a single function (e.g., an adder),
- can change its function with control signals, or
- its function can be reconfigurable.

The latter two cases represent *multi-function cells* which can change the functionality after production. For example, an add-subtract-compare cell can be used as an adder in one example application, as a subtracter in another example application, and as a comparator in a third example application. It can also be included to provide functionality for future applications in the post-silicon design phase.

Additional to multi-function cells, for each cell also different hardware implementations can be defined. Such cells are termed "*sizable cells*". The implementations can differ numerically, e.g., to set the number of inputs of a MUX, to set the size of a memory, or to set the size of a reconfigurable FSM. The implementations can also differ in categories, for example, a counter cell can offer three different implementations, one which counts in upwards direction, one in downwards direction, and one which can switch the direction using an additional control input.

Multi-Granular Interconnect

The interconnect must be able to route fine-grained as well as coarse-grained signals. For the interconnect signals, the designer specifies a set of "*connection types*", which define compatible signals depending on their width and on their semantics [WGS+12]. The concept of connection types is depicted in Fig. 3.3 using thin and wide arrows. For example, a project could use the connection types "control" (1 bit wide) for control signals, "data" (8 bits wide) for data to and from byte-oriented peripherals, and "value" (16 bits wide) for arithmetic values.

Each port of each cell has to be implemented as one connection type, but a cell can use different connection types for each port. For example, an AND gate can have two "control" inputs and one "control" output, while a combined add-subtract-compare cell could have two "value" inputs, one "value" output, a "control" input to select between addition and subtraction and four "control" outputs to specify the zero, carry, overflow, and sign flags of the result (cf. cell `AddSubCmp` in Fig. 3.8 on p. 71). For the internal design of the cells, the designer is not limited to the connection types.

Input and output ports of the whole reconfigurable module as well as parameter inputs and outputs are directly connected to the interconnect and routed through the interconnect to the cells. This enables to connect the input of a cell with a "global" input port, a parameter input, or an output of another cell. Analogously, the output of a cell can be connected to a "global" output port, a parameter output, or an input of another cell. From this definition follows, that all ports and all parameters have to comply to the defined connection types. Since the reconfigurable module implements an (example) application, the ports of the (example) applications also have to utilize the defined connection types.

The total interconnect of a reconfigurable module is split in dedicated and separate interconnects for each connection type [WGS+12]. To pass signals from one connection type to another, an explicit cell must be included. For example, an external temperature sensor provides a 16-bit value. The value is transmitted with a byte-oriented SPI peripheral. Therefore two signals with connection type "data" have to be converted to a single signal with connection type "value".

The combination of multi-granular cells and multi-granular interconnect is the basis of the realization of a unified multi-granular reconfigurable architecture.

In this section, the basic structure of the reconfigurable architecture was defined. This provides the frame for the generation of a reconfigurable module which is discussed in the next section.

3.1.5 Generation of the Reconfigurable Module

From the defined application class and the set of example applications a soft IP core with the reconfigurable architecture as defined in the previous section is generated. This corresponds to the yellow rectangle "Generation of Reconf. Module" of Fig. 3.2.

Early Approach

In the beginning of the development of the design methodology, a manual approach was used [GHG10, GGHG11]. First the architecture is split to control logic, memory, and arithmetic, analogous to the examples for the connection types used in Sec. 3.1.4. Then the inventory of the reconfigurable module is derived from the defined application class. The required features and functions are anticipated and according cell types are implemented.

To increase the flexibility in the post-silicon design phase, the semantics of the signals are generalized to general-purpose signals and additional cells are included. Then the connections between the cells are constructed according to the expected requirements. This also includes input and output ports of the reconfigurable module as well as interfaces between the connection types. Finally the infrastructure for configuration and parameterization is inserted.

In this early approach the reconfigurable module is designed manually. This does not fulfill the requirements for fast turn-around times and a high degree of automation (cf. Sec. 1.1.7). Therefore this approach was enhanced to an automated approach.

Automated Approach

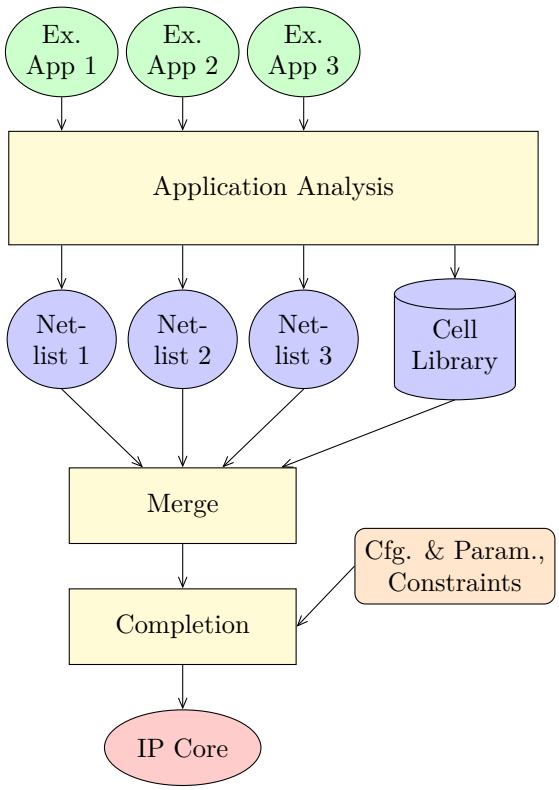
As found in Sec. 2.5.7, previous research uses three steps for the design of reconfigurable modules:

1. HW/SW partitioning of a whole program to extract the functionality which is implemented in reconfigurable logic as a set of example applications.
2. Convert the example applications to DFGs/netlists.
3. Merge the DFGs or netlists to a reconfigurable architecture.

From this procedure, the methodology for the automated generation of the reconfigurable module is derived (see Fig. 3.4) [GW14]. In this thesis, the first step of HW/SW partitioning is performed manually by the developer. He directly implements the set of example applications using a HDL (cf. Sec. 3.1.1).

- Analogous to step 2, in the "Application Analysis" step the example application HDL designs are synthesized to netlists which only instantiate and connect cell types. Concurrently the cell types themselves are derived.
- Analogous to step 3, in the "Merge" step, the netlists are merged to a single reconfigurable circuit which is implemented as a pool of cells and a reconfigurable interconnect (cf. Sec. 3.1.4).
- Finally, in the "Completion" step, the reconfigurable module is further extended with infrastructure for configuration and parameterization and finalized as an IP core as defined in Sec. 3.1.2 for the integration in the SoC.

Figure 3.4: Graphical representation of the design methodology (explanation in the text), reproduced from [GW14] with permission.



The "*Application Analysis*" step has to solve two problems simultaneously:

1. finding the set of cell types which is optimum for the set of example applications, and
2. mapping the example applications to these cell types.

The SPS project (cf. Sec. 2.5.3) [OMBKS01, KKOMB02, BMKS02] proposed an automatic methodology to solve these two problems. However, this approach is based on the frequency of the

combination of node types and therefore leads to cell types with random boundaries and functionality. This limits the reusability of the generated cell types for new applications. Therefore in the design methodology discussed in this thesis, the cell types are developed manually.

The "Application Analysis" step is performed as an iterative process which combines automated and manual actions and will be discussed in further detail in Sec. 3.2. The result of this step is a netlist for each example application and a set of cell types, which constitute the "*cell library*" (cf. Fig. 3.4).

For the "*Merge*" step the commonalities of the netlists are examined. From the number of logical instances of each cell type used by each netlist, the minimum number of physical instances for each cell type is determined. Additionally for sizable cell types, the actual implementations are defined. Afterwards, the reconfigurable interconnect is built to allow the mapping of all example application netlists to the reconfigurable module (cf. Fig. 3.3). These topics and further details on the "Merge" step are discussed in depth in Sec. 3.3.

The "Merge" step obviously requires all netlists simultaneously to determine the number of cell instances and to optimize the interconnect. The "Application Analysis" step also requires all example applications simultaneously, because the definition and optimization of the cell types should enable maximum reuse across all example applications.

The "*Completion*" step is discussed in Sec. 3.4.

3.1.6 Increase Flexibility

After production, the hardware circuits can not be modified (with reasonable costs) so the user has to get by with the available structures to implement the desired application. This means that the pre-silicon design sets restrictions to the post-silicon design-space by the limited set of cells, routing resources, and configuration and parameterization options.

Therefore the resulting reconfigurable module must provide additional flexibility to implement new applications which are different (but similar) to the example applications (cf. Sec. 1.1.7). This is achieved by utilizing the approach of the Totem project (see Sec. 2.5.4 [CH01, CH08]) to include additional cells and additional routing resources. This approach is termed "*oversizing*" in the remainder of this thesis.

Together with the principles defined in the previous sections, the following means to increase the flexibility of a reconfigurable module are applicable:

- include cell types not used by any example application,
- include multi-function cells, which can also provide functionality not used by any example application,
- use a larger implementation of sizable cells,
- increase the number of instances of certain cell types, and
- increase the routing resources.

Note that all means are specified by the user. Further, the use of a generic interconnect topology itself provides a certain degree of flexibility (see Sec. 3.3).

3.1.7 Summary

In this section, the main principles for the design methodology were determined. The reconfigurable module is specified using a set of example applications, each is implemented as an HDL design. The basic structure of the reconfigurable module is a pool of cells connected with a reconfigurable interconnect.

From the example applications an optimized set of cell types is derived in a semi-automated process and each example application is synthesized to a netlist which only instantiates these cell types. The minimum number of instances of each cell type is determined and an interconnect is optimized. To increase the flexibility of the reconfigurable module in the post-silicon design phase, multi-function cells, additional instances, and additional routing resources are included. The module is equipped with infrastructure for the configuration and parameterization and delivered as an IP core.

These main principles set a frame for the detailed development of the design methodology. In the next section, details on the "Application Analysis" step for the synthesis of the example applications and the optimization of the cell library are discussed.

3.2 Application Analysis

In this section, details of the "Application Analysis" step of the generation of a reconfigurable module are discussed (cf. Sec. 3.1.5 and Fig. 3.4). The input for this step is a set of example applications described in an HDL. The result of this step are two products: a netlist for each example application and the cell library. The netlists only instantiate cells from the cell library.

In Sec. 3.2.1 the approach to conjointly find the optimum set of cell types and to map the example applications to these cell types is introduced. This is followed by details on the individual steps and the employed concepts and algorithms.

3.2.1 Application and Cell Library Optimization

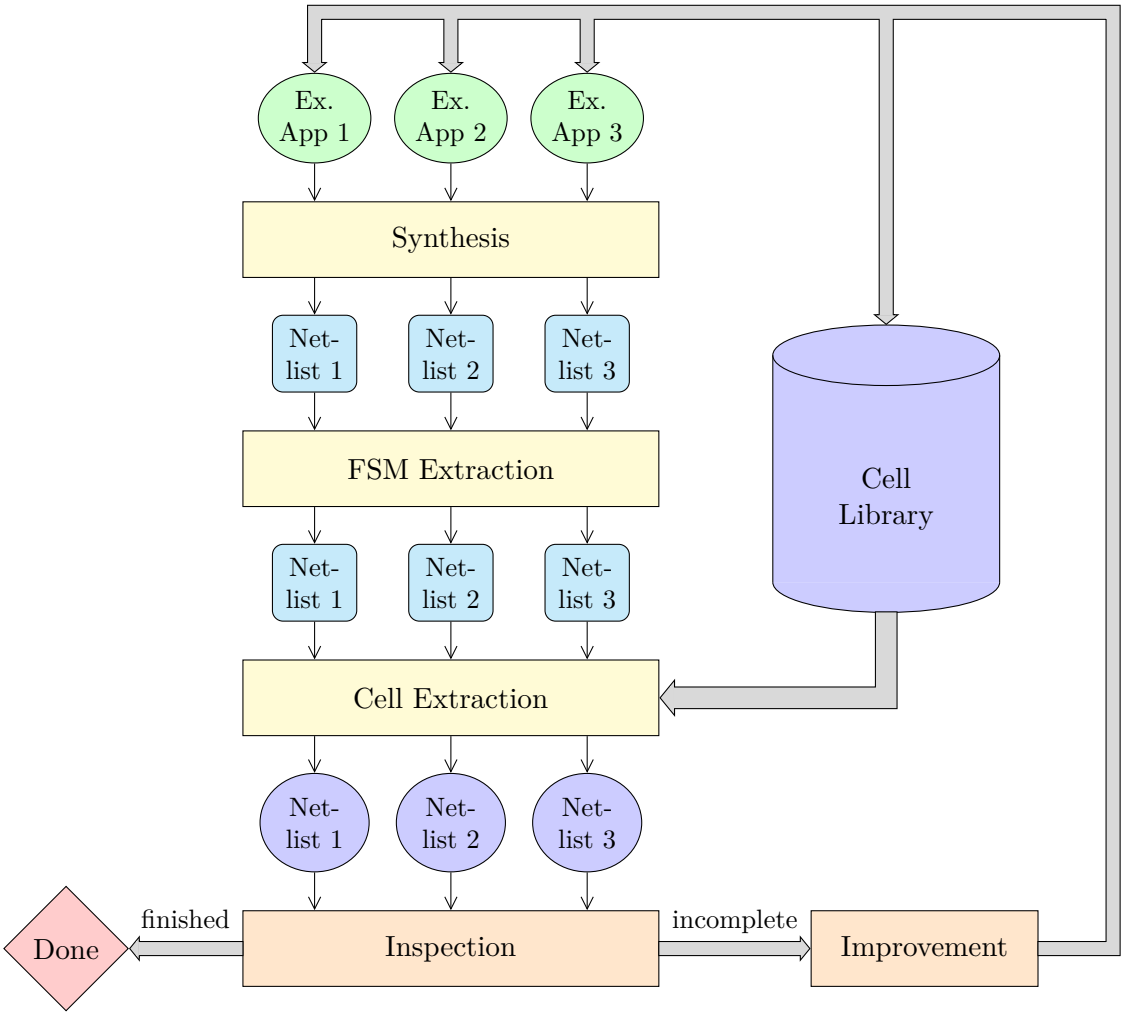
The application and cell library optimization itself is an iterative process displayed in Fig. 3.5 [GW14].

1. "*Synthesis*": First, all example applications are synthesized to netlists using generic cells.[1]
2. "*FSM Extraction*": Then the FSMs are extracted from the netlists and replaced by instances of generic FSM cells.[2]
3. "*Cell Extraction*": In the next step, the netlists of the example applications are processed to automatically extract all occurrences of the cells of the cell library and to replace them with an instance of that cells.

[1] A synthesis tool uses an internal tool-specific set of generic technology cells to represent the synthesized circuit. These implement basic functionality, e.g., multiplexers, D-FFs, and logic gates. Here the term "*generic cell*" is used to distinguish from the (coarse-grained) cells of the cell library.

[2] The generic FSM cell is also a tool-specific cell and will be mapped to a reconfigurable FSM cell in the "Merge" step. The rationale for the intermediate generic FSM cell is discussed in Sec. 3.2.5.

Figure 3.5: Flow diagram of the application and cell library optimization (explanation in the text).



4. "*Inspection*": Afterwards the developer inspects the resulting netlists (ideally as schematics) and the example application HDL code.

5. "*Improvement*":

   (a) If the netlists have groups of generic cells or the HDL designs implement functionality which is common to many example applications, the developer manually creates new cells for the cell library which implement that functionality.

   (b) The developer can modify cells to obtain better results of the "Cell Extraction" step or to achieve better reuse between the example applications.

   (c) He also can modify example applications for these goals.

This procedure is repeated until all example applications can be implemented using only generic FSM cells and cells of the cell library.

At the beginning of the discussed design methodology, the cell library might not contain any cells. In that case, the "Cell Extraction" step is skipped and the "Improvement" step is used to

implement one or more initial cells. Of course, if a previous project was already conducted using the discussed design methodology, the existing cells can be reused in a new project.

In the "*Synthesis*" step, the HDL designs of the example applications are synthesized to netlists with generic cells. To enable the mapping to multi-granular cells and signals, the synthesis tool must preserve vector signals (cf. CustArD [BS14] Sec. 2.5.6) and provide fine-grained as well as coarse-grained generic cells.

In digital design, many applications utilize FSMs combined with a datapath (FSM+D [GR94], cf. Sec. 2.6). Control flow, conditions, sequencing, timing, and operating modes are implemented with FSMs, while any kind of data handing is implemented using other constructs. The reconfigurable implementation of control-dominated tasks requires a high number of small, fine-grained, and general logic cells like LUTs and causes a large routing overhead. Therefore in the discussed design methodology dedicated reconfigurable FSM cells are utilized. This strongly reduces the number of fine-grained cells and fine-grained routing. The FSMs of the (example) applications are extracted in the "*FSM Extraction*" step for the implementation using reconfigurable FSMs, which is discussed in detail in Sec. 3.2.2.

Each cell of the cell library implements functionality which covers a detail of one or more example applications, e.g., an adder, a register, or more complex functionality like calculating the absolute value of the difference of two input values $|a - b|$. Cells can also implement wider functionality as discussed in Sec. 3.1.4

The cells of the cell library are synthesized before their use in the "*Cell Extraction*" step. The resulting netlists also use generic cells. The goal of the "Cell Extraction" step is to identify subcircuits in the netlists of the example applications which match the (smaller) netlists of the cell library. The actual identification of subcircuits is discussed in Sec. 3.2.3.

While the "Synthesis", the "FSM Extraction", and the "Cell Extraction" steps are performed automatically, the "Inspection" and the "Improvement" steps require the specific human intelligence, intuition, and experience. The goal is to identify and develop cells, which best match the example applications, which provide reuse between the example applications, and which prepare for future applications.

The application and cell library optimization is explained with an example: An example application which implements the sensor interface task (cf. Sec. 1.1.1) determines whether the new value differs from the old value, but only if the difference exceeds a given threshold:

$$|t_{\text{new}} - t_{\text{old}}| > \text{threshold}$$

Figure 3.6 shows the according detail of an exemplary netlist of that example application. The netlist contains a subtracter for $d = t_{\text{new}} - t_{\text{old}}$, a comparator of that difference whether it is smaller than 0, a subtracter $\overline{d} = 0 - d$ to negate that value, a MUX $|d| = d$ or $\overline{d}$, and a comparator $|d| > \text{threshold}$.

By inspecting the HDL sources and the netlists and keeping the whole application class in mind, the designer can identify the arithmetic operation $|t_{\text{new}} - t_{\text{old}}|$ as a frequent task (marked with a dashed line in Fig. 3.6) and decide to implement this as a dedicated `AbsDiff` cell (see Fig. 3.7).

As mentioned above, cells can also provide wider functionality to match multiple example applications. For example, an example application calculates the mean value of a sequence of measurements. This requires an adder cell to calculate the sum $a + b$ of consecutive measurements.

Figure 3.6: Detail of the netlist of an implementation of the sensor interface task (described in Sec. 3.2.1) which determines whether the new sensor value t_{new} differs for more than a given threshold from the previous value t_{old}. The netlist instantiates exemplary generic cells (yellow). The region marked with the dashed rectangle is a subcircuit which calculates the absolute difference of two values. The enable signal of the register and the result of the greater-than comparison "$>$" are connected to an FSM.

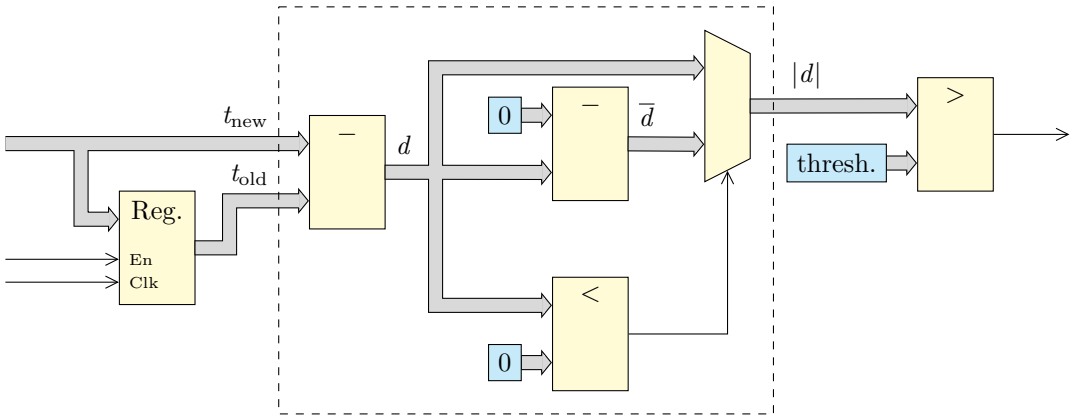
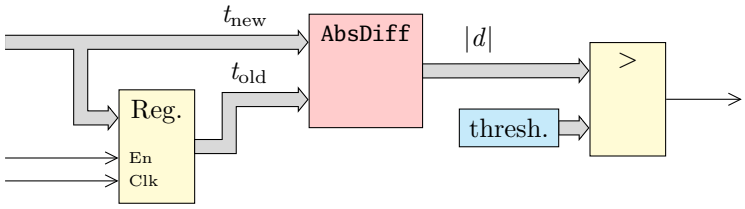


Figure 3.7: Subcircuit of Fig. 3.6 replaced by the cell `AbsDiff`.



This adder cell can be extended to switch between sum and difference $a - b$, which only causes low hardware overhead. A further extension to provide the zero, carry, overflow, and sign flags of the result enables to use the cell as a comparator.[3] This add-subtract-compare cell can also be used by the previous example application to compare $|d|$ with the threshold $|d| >$ threshold.

In summary, the "Inspection" and the "Improvement" steps include

- identifying groups of generic cells and implementing a coarse-grained cell,
- merging similar cells (e.g., adder, subtracter, and comparator) and generalizing to multi-function cells,
- extending cells with features to increase the flexibility of the reconfigurable module in the post-silicon design phase, and
- modifying the example applications to better utilize the available cells while preserving its behavior.

After the "Improvement" step, the refinement cycle is repeated using the automated "Synthesis", "FSM Extraction", and "Cell Extraction" steps before the resulting netlists are inspected again.

[3] A comparison between a and b is performed by subtracting the two values $a - b$ and examining the flags of the result. For example, $a = b$ if the zero flag is '1', and $a >= b$ if the sign flag is '0'.

When all example applications can be implemented using only generic FSM cells and cells of the cell library, the application and cell library optimization is finished. The result is the set of netlists of all example applications and the cell library (cf. Sec. 3.1.5 and Fig. 3.4).

In the next two sections, details on the "FSM Extraction" and "Cell Extraction" steps are discussed.

3.2.2 FSM Extraction

From each example application, the FSMs described in the HDL source code have to be extracted. As an HDL module usually includes more logic together with the FSM, the actual FSM has first to be identified. Then its input and output signals, the state encodings, and the transitions have to be extracted. Finally the FSM logic is replaced by a generic FSM cell which shows the exact same behavior.

For the description of FSMs in HDL, [Gio95] distinguishes between explicit FSMs and implicit FSMs:

- For explicit FSMs, an explicit state register is described and the FSM outputs as well as the FSM transitions depend on that state, e.g., using a VHDL `case` statement.
- On the other hand, using VHDL `wait` statements inside a `process` requires the synthesis tool to implement an implicit FSM.

To extract both types of FSMs, [Gio95] processes the control flow graphs (CFGs) and the DFGs of the logic module. The extracted FSM is represented as state transition graph. This also supports the specification of datapath operations.

A different approach is applied by [SP05] to extract implicit FSMs for cycle accurate IO timing programmed in SystemC. The source code (or its abstract syntax tree, AST) is directly processed to identify the states as delimited by `wait()` statements. The FSM inputs and the transitions are derived from conditional statements (`if`, `while`), and the outputs are recognized from the actions performed in the source code. Although this work is specific to SystemC, the algorithm is applicable to any HDL.

Both approaches of [Gio95] and [SP05] extract an FSM from a logic module, but can not *identify* an FSM in a possibly larger logic circuit. [WG13, Wol15b] identify FSMs by detecting state signals in synthesized logic circuits. These are recognized as output of a register. The input of the register must result from a MUX tree only switching between constant values (i.e., the state encodings) and the current state signal. Further, the state signal is not allowed to connect to a module output port. After identification, the FSM is extracted by processing the MUX tree with the state encodings. The FSM inputs are derived from the MUX select signals, and the FSM outputs are derived from all cells which evaluate the current state signal. The FSM transitions are derived from evaluating the logic circuit for all states and input signals. The extracted FSM is replaced by a generic FSM cell.

The method for FSM identification and extraction directly investigates the logic circuit. This highly depends on the properties and results of the preceding synthesis (especially the MUX tree). However, the FSM extraction is integrated in the synthesis tool, therefore the dependence is justified.

Only the approach by [WG13, Wol15b] provides identification of FSMs inside of larger designs. Therefore it is selected for the "FSM Extraction" step.

3.2.3 Cell Extraction

In Sec. 3.2.1 the "Cell Extraction" step was introduced. In the netlist of each example application ("haystack"), all occurrences of the cells from the cell library ("needles") have to be found. To achieve a fast turnaround time in the refinement process, the "Cell Extraction" step must be automated (cf. Sec. 1.1.7). The task of the "Cell Extraction" step is known as the *subcircuit extraction* problem [ZWH03]. To identify given subcircuits within logical circuits, two approaches are applicable:

- For *logical equivalence checking*, the circuits are represented as logical functions. The equivalence of logical functions is shown analogous to mathematical proofs [Swa97]. This approach can be employed to identify subcircuits within the "haystack" circuit which are logically equivalent to the given "needle" subcircuits.
- The netlists of logical circuits can be represented as graphs. Extracting subcircuits is equivalent to finding sub-graphs in the "haystack" graph, which are isomorphic to the given "needle" graphs (*sub-graph isomorphism*) [OEGS93].

The implementation of a given functionality can lead to different circuits. For example, a ripple-carry adder and a carry-lookahead adder both perform the same functionality, but use different logic circuits. This problem also exists for coarse-grained logic, e.g., the sum $a + b + c$ can be implemented with two-input adders as $(a + b) + c$ and as $a + (b + c)$. Also the above example of the `AbsDiff` cell can be implemented differently: first both differences $d' = t_{\text{new}} - t_{\text{old}}$ and $d'' = t_{\text{old}} - t_{\text{new}}$ are calculated and finally the MUX selects the positive value $|d| = d'$ or d''. In all three examples, the circuits and therefore the netlist graphs are different, but the functionality and therefore the logic functions are identical. Therefore, the logical equivalence approach can identify the subcircuits but the sub-graph isomorphism approach can not.

On the other hand, the logical equivalence approach has to map the circuits to fine-grained single-bit logic before identifying logically equivalent "needle" subcircuits in the "haystack" circuit. This problem is similar to technology mapping of a logical circuit to standard cells during ASIC synthesis [Cha07]. The algorithms used in technology mapping represent the logic functions as graphs (binary decision diagrams, BDDs or and-inverter graphs, AIGs) and therefore implement logical equivalence detection as a sub-graph isomorphism problem. However, the "needle" subcircuits are small and fine-grained with only a few input signals and a single output signal. In contrast, the circuits discussed in this section result in large coarse-grained logic circuits with a high number of graph nodes and edges. Additionally, the algorithms for technology mapping only handle combinational logic and use separate approaches to map registers to D-FFs.

Following from these considerations that the logical equivalence approach is not suitable for the given subcircuit extraction problem, the sub-graph isomorphism approach is chosen. To mitigate the problems to find differing netlists of identical functionality, countermeasures have to be found.

As previously discussed the "Synthesis" step creates netlists with generic cells of all example applications. For the "Cell Extraction" step, all cells of the cell library are synthesized to netlists with generic cells too. These generic cells can be coarse-grained and use vector signals, e.g., adders, comparators, and shift registers. All netlists represent graphs with the generic cells as graph nodes and the signals as graph edges.

To detect all occurrences of the "needle" sub-graphs in the "haystack" graph, the algorithm proposed by [Ull76] is used. This algorithm creates adjacency matrices for each graph. These

specify for each node, whether an edge exists to a given other node or not. The adjacency matrices are then processed to find the occurrences of the sub-graphs. In the discussed subcircuit extraction problem the cell types and the signal types have to be considered additionally to the graph topology. This requires extensions to the algorithm [OMHG11, WG13, Wol15b]. These algorithms introduce extended adjacency matrices which encode the cell and signal types. [WG13, Wol15b] also allow to specify compatible node and signal types as well as rules to compare the ports of the cells.

After the sub-graph isomorphism extraction, all occurrences are replaced by instances of the respective cell (see Fig. 3.7 on p. 67).

3.2.4 Topological Variants and Reduced Variants

To mitigate the problem of differing netlists and therefore graph representations for different implementations of the same functionality, the concept of "*topological variants*" is introduced. The implementation of each cell in the cell library is used twofold: Firstly, in the sub-graph isomorphism detection to *identify* its occurrences in the example application netlists, and secondly, to *replace* the occurrences with a single cell instance. The second usage also implies that the implementation is instantiated in the final reconfigurable module and included in the soft IP core. This implementation is termed the *main variant* of a cell.

The division of the "Cell Extraction" step in two sub-steps is utilized for the concept of topological variants: For each cell an unlimited number of additional logic designs can be prepared. These use a different implementation as the main variant but have to be functionally equivalent. The topological variants are only used for the subcircuit extraction to allow the identification of different implementations. The occurrences are however replaced by an instance of the main variant. Creating topological variants of cells is an element of the tasks performed in the "Improvement" step defined in Sec. 3.2.1.

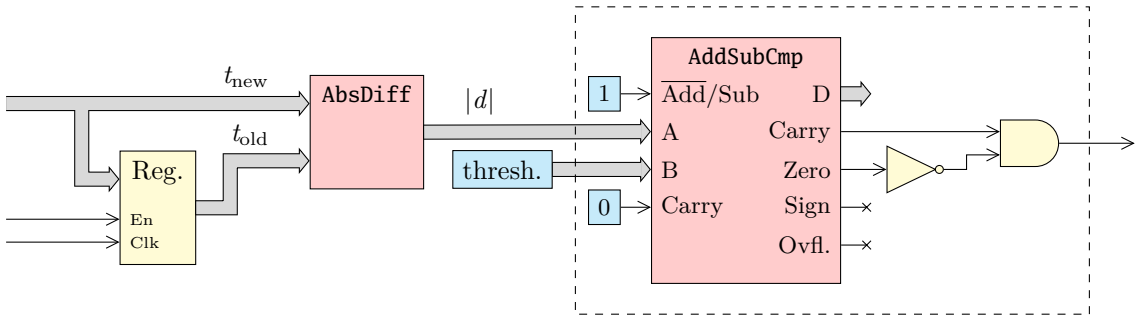
As also mentioned in Sec. 3.2.1, cells can provide wider functionality as required in any example application, e.g., an add-subtract-compare cell. For the automatic detection of such cells in the example application, the concept of "*reduced variants*" is introduced. As a first step after implementing the main variant, reduced variants of the cell are developed, e.g., an adder, a subtracter, a less-than comparator "<", a greater-than comparator ">", etc. These are used by the sub-graph isomorphism algorithm to detect all occurrences in the example applications. Note that the reduced variants implement a different and stripped-down functionality than the main variant, hence the name. Therefore as a second step, wrapper modules for the main variant are implemented. These instantiate the main variant together with the necessary logic to implement the specialized, i.e., reduced functionality. The occurrences of the reduced variants are replaced by the wrapper modules.

The above example is demonstrated in Fig. 3.8: The greater-than comparator ">" is replaced by the `AddSubCmp` add-subtract-compare cell and wrapper logic (cf. Fig. 3.7). Note that this is a special case where the "needle" netlist consists only of the single generic greater-than comparator cell. The `AddSubCmp` cell is set in subtract mode by applying `1` to the $\overline{\text{Add}}$/Sub mode selection input. The result value D is unused, but the flags are logically combined to determine whether $|d| >$ threshold.

Other examples are the use of a four-input MUX cell as a two-input MUX or the instantiation of a counter cell without using its "Reset" control input.

Note that the second step of the reduced variant concept is the opposite of subcircuit extraction: While with subcircuit extraction a collection of cells is replaced by a single cell, here a single cell (the extracted reduced variant) is replaced by the circuit of the wrapper module. The additional logic can be captured by a following subcircuit extraction. In the special case depicted in Fig. 3.8, the single-bit logic can also be realized by the FSM: The condition for the state transition observes both, the carry and zero flag, instead of the single output of the former comparator.

Figure 3.8: The greater-than comparator "$>$" of the circuit in Fig. 3.7 on p. 67 is replaced by the cell `AddSubCmp` add-subtract-compare cell with wrapper logic using the reduced variant concept. The outputs "D", "Sign", and "Ovfl." are unused.



3.2.5 Handling of Configurable Cells

As defined in Sec. 3.1.4, the cells in the cell library can provide a single function or provide multiple functions selected either using control signals or using configuration data. For example, the `AddSubCmp` cell in Fig. 3.8 can switch between addition and subtraction using the $\overline{\text{Add}}$/Sub input. In certain applications it might be required to switch the functionality during operation. Therefore this input should be implemented as a control input, e.g., driven from an FSM or a constant value.

For an example of a reconfigurable cell, a sensor is considered which provides the value as subsection of a larger signal vector (e.g., a 10-bit value located in bits 12 downto 3 of a 16-bit signal, while the other bits are used as flags). This subsection has to be selected from the signal vector before further arithmetic operations. However, the implementation of a dedicated cell would be specific to this one example application. Therefore a cell with a reconfigurable shift-right and a reconfigurable mask is suggested.

The subcircuit extraction requires knowledge on the different specialized functionalities of reconfigurable cells together with the appropriate configuration data. To handle reconfigurable cells by the "Cell Extraction" step, the reduced variants approach is utilized. In the "Improvement" step the developer first implements the main variants of the reconfigurable cells. These are instantiated in the final reconfigurable module, identical to fixed function cells. These also have input and output ports which have to comply to the defined connection types. Additionally, dedicated input ports to supply the configuration data are included.

Secondly, the developer creates reduced variants which implement the specialized functionalities of the reconfigurable cells. These are used by the subcircuit extraction. And thirdly, the developer implements wrapper modules which instantiate the reconfigurable cells and supply the required configuration data to the dedicated input ports. The configuration data has to be transferred to

the "Merge" and "Completion" steps for inclusion in the final configuration stores. The wrapper modules are used to replace the extracted subcircuits of the reduced variants in the application netlists. However, this approach poses the disadvantage that for each specialized functionality a dedicated set of a reduced variant and a wrapper module have to be implemented. Therefore the developer can limit his effort to the cases used by the example applications.

Referring to the previous example of a shift-and-mask cell, the reduced variant directly implements the selection of the subsection of the vector, e.g., as VHDL assignment `Y_o <= "000000" & A_i(12 downto 3);`. The wrapper module instantiates the shift-and-mask cell and supplies the configuration data for the right-shift (in this case `3`) and the mask (in this case `x"03FF"`).

Reconfigurable cells which are also *sizable* (cf. Sec. 3.1.4) require a special handling, because the actual implementation of these cells results from the requirements of all example application concurrently. It is therefore determined during the cell instantiation of the following "Merge" step (cf. Sec. 3.3.1). The generation of the configuration data depends on the actual implementation of the cell and therefore can only be generated after the actual implementation is determined.

To handle reconfigurable and sizable cells, an intermediate cell type which is independent of the actual implementation is used. Instead of the configuration data, an internal format describes its configurable functionality. After the actual implementation of the cell is determined, the configuration data is derived from this internal format.

A special case of reconfigurable and sizable cells are the reconfigurable FSMs. Special algorithms are required for the identification and extraction of the FSM logic from the (example) applications. Therefore instead of the reduced variants approach the separate "FSM Extraction" step is introduced before the "Cell Extraction" step (cf. Sec. 3.2.2). A generic FSM cell is used as intermediate cell type to represent the extracted FSM.

3.3 Merge to Reconfigurable Architecture

In the previous section the synthesis of the example applications to a set of netlists and a cell library was discussed. In this section the procedure to merge the separate netlists to one reconfigurable module is described (cf. "Merge" step in Sec. 3.1.5 and Fig. 3.4 on p. 62). The result of the procedure is the reconfigurable module with input and output ports, instances of the (reconfigurable) cells of the cell library, and a reconfigurable interconnect, as defined in Sec. 3.1.4 (cf. Fig. 3.3 on p. 59).

In Sec. 3.3.1, the preparation to instantiate the cells is discussed. In Sec. 3.3.2 the requirements for the interconnect topology are evaluated and a suitable topology is selected. Finally, in Sec. 3.3.3 the actual interconnect optimization is discussed.

3.3.1 Cell Instantiation

The procedure to generate the reconfigurable module consists of the following steps (extending [WGS+12, GW14]):

1. determine the required implementation of sizable cell types

2. determine the minimum number of instances of each cell type
3. include additional instances for yet unknown applications (oversizing)
4. generate and optimize the interconnect

Before the interconnect can be generated and optimized (step 4), the number of instances of each cell type has to be determined. For fixed cell types the number of instances results from the maximum number of instances used by each netlist (step 2). For sizable cell types the actual implementations have to be determined (step 1) beforehand. If only a single instance is used in any netlist, the biggest implementation is selected. Multiple cell instances offer the optimization potential to map each netlist instance to a fitting cell of the interconnect. In the discussed methodology this task is performed manually by the developer. For sizable cell types which are also reconfigurable, the deferred generation of the configuration data (cf. Sec. 3.2.5) is performed here.

To increase the flexibility of the reconfigurable module in the post-silicon design phase, in the "Merge" step three means are taken (cf. Sec. 3.1.6):

- increase the size of sizable cells (step 1)
- increase the number of instances of cell types (step 3)
- increase the routing resources (step 4, discussed below)

As defined in Sec. 3.1.4, for each connection type a separate interconnect is generated. Cell types which implement ports for a mixture of connection types are connected to each of the used interconnects. The following discussion is applicable to all parallel interconnects.

After the cell instantiation (steps 1–3) is completed, in the next section the interconnect topology is defined, before in Sec. 3.3.3 the optimization of the interconnect (step 4) is discussed.

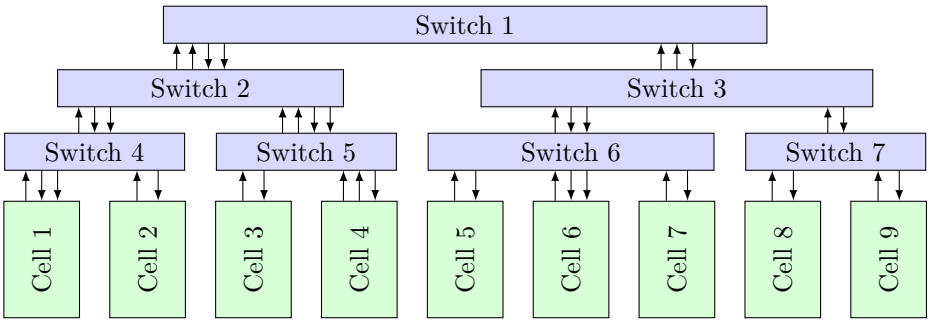
3.3.2 Interconnect Topology

As found in Sec. 2.5.4, the use of a generic topology alone leads to increased flexibility in the post-silicon design phase. The interconnect topology used in the reconfigurable module discussed in this thesis has to fulfill a set of requirements (extending [WGS+12]):

1. allow connections between all cells
2. low power consumption
3. low propagation delay
4. low chip area
5. little configuration data
6. allow optimization
7. prohibit over-optimization
8. allow oversizing
9. usable for synthesis (cf. Sec. 1.1.7)

From the topologies reviewed in Ch. 2 and in [WGS+12], the tree topology is selected for its given optimization potential and its universality. The cells are instantiated as leaf nodes of the tree (cf. Fig. 3.9). The inner nodes of the tree are implemented as reconfigurable switches. The connections between the cells and the switches represent the edges.

Figure 3.9: Exemplary tree interconnect for nine cells, arranged in three levels (reprinted with modifications from [WGS+12, GW14] with permission).



The tree topology can connect any cell with any other (requirement 1). The tree topology provides hierarchical clustering of nodes. According to [KR98] and [GZR99] this reduces the power consumption (requirement 2) and delay (requirement 3). The number of switches is small which reduces the amount of configuration data (requirement 5). The chip area (requirement 4) is reduced firstly due to the low number of switches, and secondly due to the reduced clock tree for the configuration D-FFs (cf. Sec. 2.3.3 [WKW+05, WAWS03]). The clustering allows the optimization of the interconnect by utilizing the locality of connections (requirement 6) but still allows connections to any cell (requirement 1).

The tree topology also allows the optimization of the number of up and down connections between the tree levels (requirement 6). In the discussed methodology whole cells are connected as leaf nodes instead of the individual cell ports. This limits the potential for optimization and leaves flexibility for the post-silicon design phase (requirement 7). The tree topology allows oversizing by including additional up and down connections between the switches (requirement 8). The switches of the tree topology can be implemented using unidirectional MUXes, therefore the topology is suitable for ASIC synthesis (requirement 9).

As stated above, for each connection type a separate interconnect is implemented. For each of these interconnects, each cell is member of a single cluster. To improve the connectivity within the interconnect, the generation of multiple trees for each connection type is enabled. This can be envisioned as stacked trees in the z-axis of the drawing layer of Fig. 3.9. However, each cell is placed at a different leaf position in every tree resulting in a different clustering. Therefore each cell can be member of multiple clusters which improves the routability of signals [WGS+12]. Note that each interconnect for each connection type can consist of multiple trees.

3.3.3 Interconnect Optimization

With the tree topology selected, the actual optimization of the interconnect is discussed in this section. The input to this procedure consists of:

- the cell library which defines the *cell types*,
- the set of netlists, which instantiate and connect the cell types from the cell library,
- the actual implementations of sizable cells, and
- the number of instances of each cell type in the reconfigurable module (cf. steps 1–3 in Sec. 3.3.1).

For better distinction, in this section the instances of cell types in the netlists (logical instances) are termed *nodes* while the instances of cell types in the reconfigurable module (physical instances) are termed *cells*.

Note that for linear or mesh topologies the cells are at given spatial positions in a 1D or 2D arrangement. This results in spatial proximity relations between the cells. On top of the cells, the interconnects for each connection type are placed. This means that the position of a cell is the same within each interconnect. Therefore all interconnects have to be jointly optimized. In contrast, an interconnect with tree topology only produces *logical* proximity individually in each tree and therefore allows independent optimization.

In relation to the reviewed approaches in Sec. 2.5, in this thesis a bottom-up-creation approach is realized. In contrast to the KressArray methodology (cf. Sec. 2.5.1), all netlists should be used concurrently for the optimization to achieve best results.

For the optimization, two problems have to be solved concurrently:

1. mapping physical cells to leaf positions of the tree (cell-to-leaf mapping), and
2. mapping netlist nodes to physical cells (node-to-cell mapping).

To achieve short turnaround times (cf. Sec. 1.1.7) the optimization process must be automated.

The design methodology developed by the Pleiades project (cf. Sec. 2.5.2) uses manual clustering and placement of the cells ("satellites"). A local interconnect is created within each cluster. For the connections between the clusters a hierarchical mesh interconnect is generated. For the optimization, the solutions for different topologies are compared. The nets are routed with graph-based algorithms and simulated annealing using energy and delay as a cost metric [Wan01]. For the mesh topology bidirectional wires are assumed which is not applicable for synthesized designs. The cost function considers hardware implementation details (energy, delay), thus the approach is not independent of the semiconductor process (cf. Sec. 1.1.7).

The Totem project (cf. Sec. 2.5.4) proposed a methodology to generate RaPiD-like 1D linear array architectures. To optimize the placement of the cells, simulated annealing moves the cell-to-leaf mapping and the node-to-cell mapping. In the following routing phase, bidirectional routing tracks are generated. An iterative approach is used to determine the number of tracks and the mapping of signals to the tracks [CH08]. This methodology poses the disadvantage that the placement phase can not consider the routing cost.

Both approaches use simulated annealing which assumes a spatial area at which the cells are moved within their vicinity. This does not apply to the hierarchical nature of the tree topology because of its non-uniform distances (e.g., compare the distance between cells 3 and 4 and between 4 and 5 in Fig. 3.9).

The optimization approach proposed by [WGS+12] was specifically developed for tree topologies. It uses the Kerninghan-Lin algorithm to optimize the node-to-cell mappings and the cell-to-leaf mapping: First the node-to-cell mappings of all netlists are concurrently optimized so that nodes with similar connections in all netlists are mapped to the same cell. Then the total routing length, i.e., the sum of the number of switches each signal has to pass, is minimized by optimizing the cell-to-leaf mapping and the node-to-cell mapping.

Considering the disadvantages of the algorithms used by the Pleiades and the Totem project and that [WGS+12] was specifically developed for the task discussed in this section, the latter is selected for the interconnect optimization.

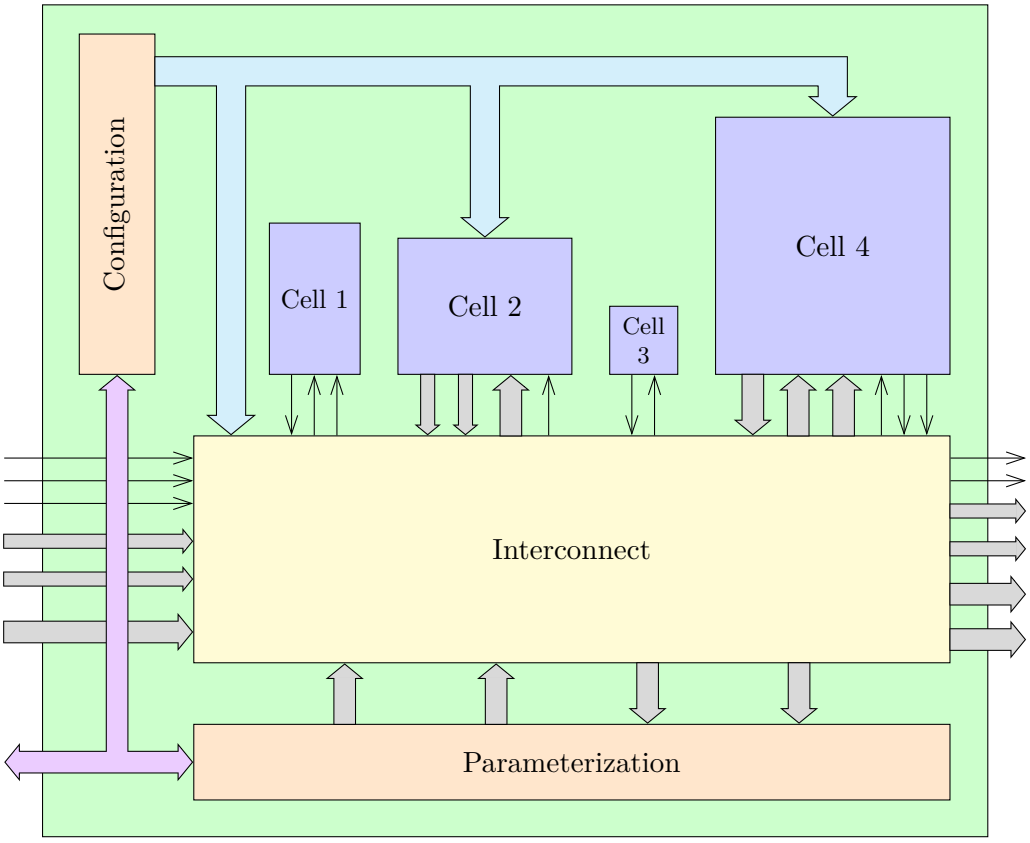
The result of the "Merge" step is an HDL module which instantiates cells and creates MUXes as the reconfigurable interconnect (see Fig. 3.3). Additionally the configuration data for the reconfigurable cells as well as for the reconfigurable interconnect is created for all example applications. To map new applications to the reconfigurable module in the post-silicon design phase and to create the appropriate configuration data, meta-information about the interconnect is stored.

3.4 Completion of the Reconfigurable Module

In the "Completion" step the module from the "Merge" step is extended with infrastructure for the configuration and parameterization (see Fig. 3.10). The interconnect and all reconfigurable cells (e.g., a reconfigurable FSM) are supplied with configuration data. For parameterization (e.g., the threshold value used in Fig. 3.6), read and write circuits are included.

To access the configuration data and the parameters, interfaces for the CPU are added to the reconfigurable module. Finally a soft IP core including further data is created (see Sec. 3.1.2). However, this procedure is strongly related to the actual implementation of the discussed design methodology and therefore is presented in Sec. 4.8.

Figure 3.10: The basic reconfigurable module of Fig. 3.3 is extended with infrastructure for configuration and parameterization.

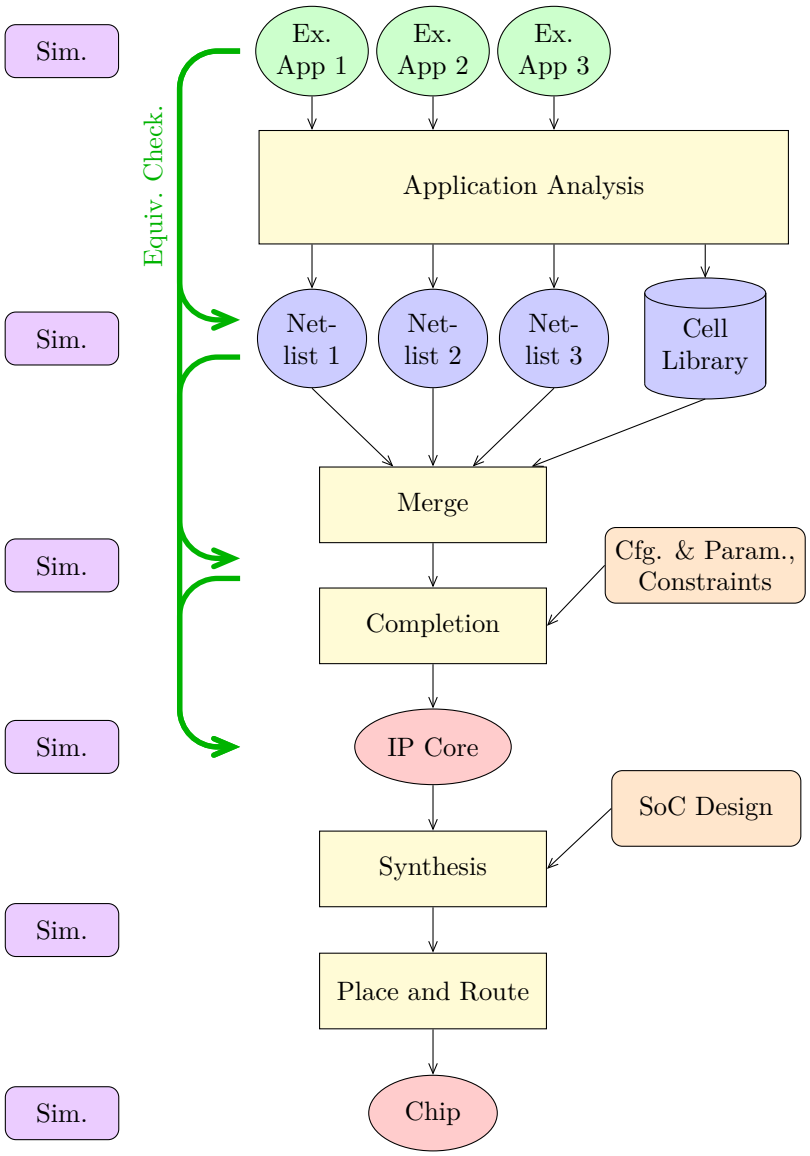


3.5 Verification

The requirements for the discussed design methodology include verified results (cf. Sec. 1.1.7). This means that the reconfigurable module must be verified to correctly implement each example application. Further, each example application has to be verified itself to correctly implement the given functionality. To avoid long revision cycles, additionally the result of each step of the design methodology must be verified (see Fig. 3.11). For clarity all verification related discussion is collected in this section. The methods employed for verification include simulation, formal verification (model checking), and logical equivalence checking.

The first designs to be verified are the manually developed example applications. As defined in Sec. 3.1.1 each example application is implemented as RTL design using a HDL like VHDL or

Figure 3.11: Verification in the design methodology using simulation and logical equivalence checking (extension of Fig. 3.4).



Verilog. This allows to use industry standard as well as free and open-source verification tools. For each example application, the developer creates a testbench. This is used in simulation to check the behavior of the example application. Additionally the developer can use methods like formal verification to verify the design.

To enable the early development of firmware for each example application, *HW/SW co-simulation* is used (cf. Sec. 2.4.3). Wrapper modules which have the same module interfaces as the final reconfigurable module are generated. The wrapper modules instantiate the RTL implementation of the example applications and connect its ports to the module ports appropriately. Additionally infrastructure for parameterization and the interface to the CPU are included. Note that for the *early* HW/SW co-simulation no configuration data is required. However, parameters are directly accessible and handled by the firmware drivers.

These wrapper modules are used together with a CPU model which executes the firmware driver and application code. Alternatively the wrapper modules are integrated in the full SoC design instead of the reconfigurable module. Then the whole SoC including an RTL design of the CPU is simulated. Before that, the compiled firmware is loaded to the simulated code memory.

For the design of cells for the cell library during the "Application Analysis" step (cf. Sec. 3.2.1), a testbench for each cell is developed for the verification using simulation. Analogous to the development of the example applications, the developer can also use formal verification. Additionally the topological variants are compared to the main variant and the reduced variants are compared to their according wrapper modules using logical equivalence checking to ensure identical functionality.

After the "Application Analysis" step (cf. Sec. 3.2), for each example application an implementation using only cells of the cell library exists. These implementations have the same interfaces as the RTL implementations. Therefore the original testbench is used to verify the behavior using simulation. Besides simulation, *logical equivalence checking* is utilized to verify that the generated implementations are logically equivalent to the RTL implementations [GW14] (and cf. Sec. 2.5.5). Before the verification, the configuration data of all instantiated reconfigurable cells is applied.

In the "Merge" step (cf. Sec. 3.3), the reconfigurable module without the infrastructure for configuration and parameterization is finished. For each example application, a wrapper module with the interfaces of their original RTL implementations and an instance of the reconfigurable module is generated. For the simulation and for the logical equivalence checking first the configuration data for the reconfigurable cells as well as for the interconnect is applied. For simulation the original testbench is used.

After the "Completion" step (cf. Sec. 3.4), the reconfigurable module contains the infrastructure for configuration and parameterization. Therefore, compared to the previous case, slightly different wrapper modules are required. These modules are verified analogous to the previous case.

After the IP core is integrated in the SoC design a modified testbench is required for simulation. The reason is that in general the whole chip can not be wrapped to only the interfaces of each example application. Additionally, ports of the reconfigurable module which correspond to ports of the example applications have to be accessed across the module hierarchy. Similar to the above discussion of the HW/SW co-simulation, the firmware code is applied to the simulated code memory. The only difference is that here the firmware has to supply the configuration data to the reconfigurable module. The simulation of this setup is identical to the silicon chip. It can

also be used for post-synthesis and post-place-and-route simulations, even with included delay annotation generated from parasitics extraction.

For the whole SoC design no logical equivalence checking with the example applications is possible because the SoC includes a number of other modules. However, logical equivalence checking between the full RTL design of the SoC and the netlists resulting from synthesis and place and route is possible.

3.6 Post-Silicon Design Phase

In the previous sections the pre-silicon design phase of the design methodology was discussed. This section covers the details on the post-silicon design phase for the implementation of new applications. This is analogous to common FPGA design, however, since the reconfigurable module is tailored to a specific application class, the pre-silicon design phase limits the design-space of the post-silicon design phase. That means that new applications have to be within the application class of the reconfigurable module.

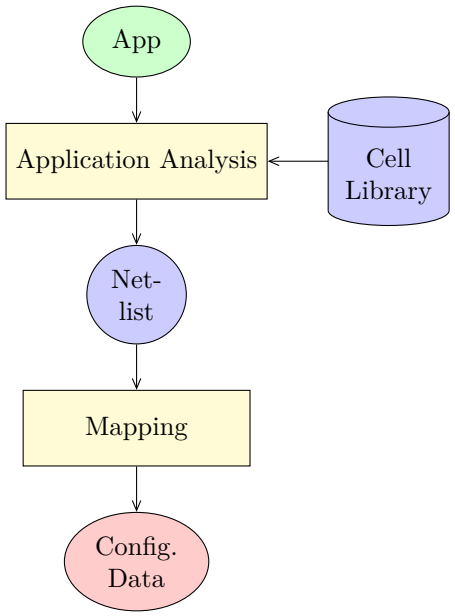
In the pre-silicon design phase, the generation of the configuration data is already included. The configuration data for reconfigurable cells is generated during the "Cell Extraction" step in the "Application Analysis" step (cf. Sec. 3.2.5 and Fig. 3.5) and for sizable cells during the cell instantiation of the "Merge" step (cf. Sec. 3.3.1). The configuration data for the interconnect for each example application is generated from the routed designs in the final optimization iteration (cf. Sec. 3.3.3).

In the post-silicon design phase new applications are implemented. In the discussed methodology, the design procedure of the post-silicon design phase is a special case of the pre-silicon design phase (see Fig. 3.12 and cf. Fig. 3.4 on p. 62). First the new application is developed, identically to the pre-silicon design phase.

In the "Application Analysis" step (cf. Sec. 3.2 and Fig. 3.5), the application is synthesized and the FSMs are extracted. In the "Cell Extraction" step, the cells contained in the existing and immutable cell library are extracted from the new application and the configuration data for reconfigurable cells is generated. In the "Improvement" step, the application can be modified, e.g., if the "Cell Extraction" did not identify certain constructs. It is also possible to create new topological variants and reduced variants of existing cells. However, no new cell types can be created. The goal of the "Application Analysis" is the complete extraction of the application with no residual logic. If the application can not be extracted, the reconfigurable module does not provide the necessary resources and therefore can not implement the application.

The resulting netlist using only cells of the cell library is then mapped to the reconfigurable architecture (cf. Fig. 3.12). This step is a subset of the "Merge" step of the pre-silicon design phase (cf. Sec. 3.3). For all sizable and reconfigurable cells the configuration data is generated (cf. Sec. 3.3.1). Then the meta-information on the interconnect, which was stored in the pre-silicon design phase, is used to map the nodes of the netlist to the leafs of the interconnect tree (node-to-cell mapping [WGS+12], cf. Sec. 3.3.3) and to route the signals via the interconnect. Finally the configuration data for the interconnect is generated. If the reconfigurable architecture does not provide the required amount of cells or routing resources, the application can not be implemented.

Figure 3.12: The post-silicon design methodology is a special case of the pre-silicon design methodology (cf. Fig. 3.4) (explanation in the text).



3.7 Transition-Based Reconfigurable FSM

During the "Application Analysis" and "Merge" steps the FSMs of the example applications are mapped to reconfigurable FSM cells. In Fig. 3.13 an exemplary state diagram of an FSM is given. The approaches for reconfigurable FSM architectures reviewed in Sec. 2.6 focus on the implementation of the output function and the next state function. These logic functions compute the values for the output and the next state signals from the current state and the input signals. To implement such combinational logic functions a universal fine-grained reconfigurable architecture is required which causes high power consumption, large area overhead, and large configuration data. To remedy these shortcomings the *Transition-Based Reconfigurable FSM* (TR-FSM) was developed [GDHG10, GDHG11]. This architecture directly focuses on the state transitions which allows a compact and efficient implementation.

Figure 3.13: Exemplary state diagram of an FSM with four states, four input signals, and five output signals. The conditions for the input pattern to activate a transition are printed above the transitions. The output signal patterns are printed below the transitions.

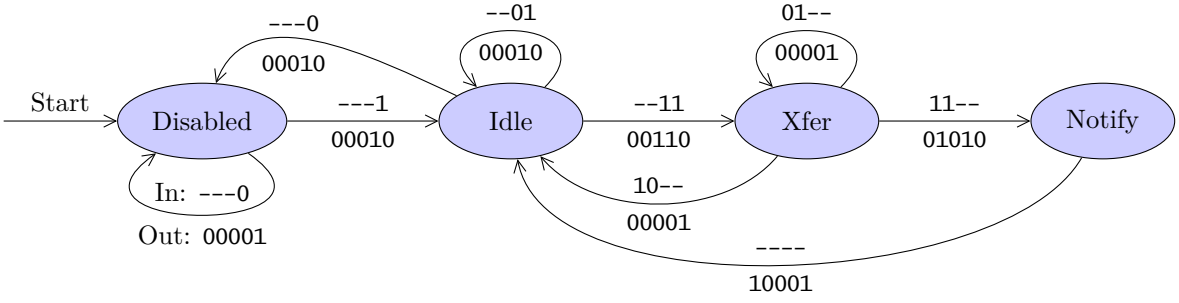
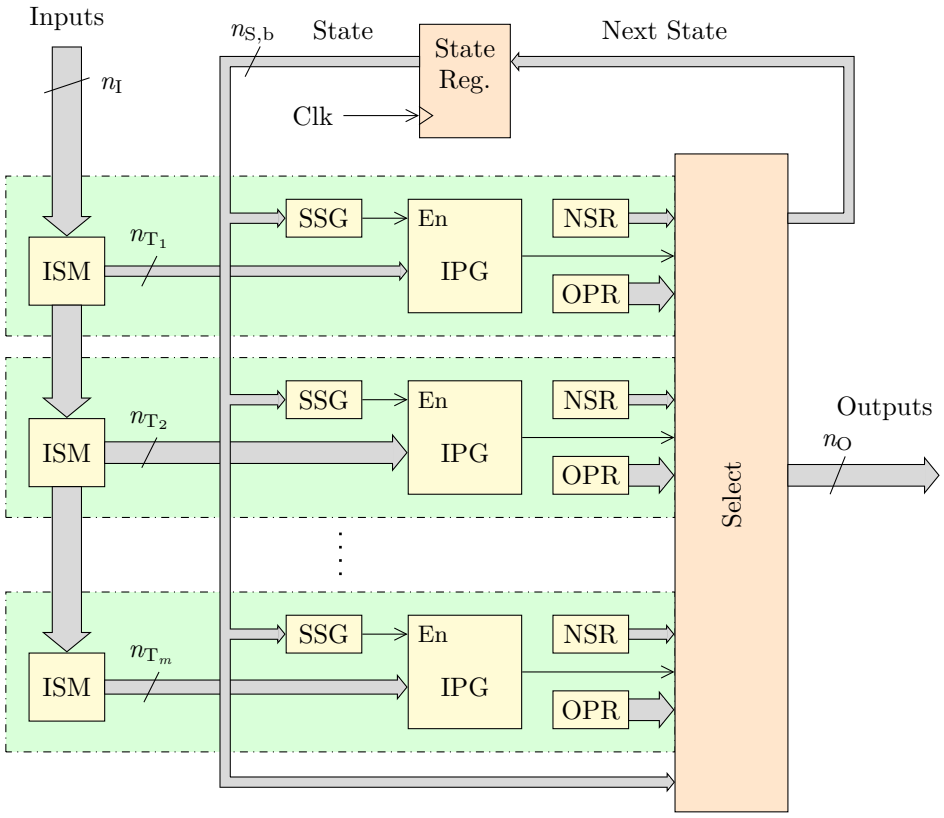


Figure 3.14: The TR-FSM architecture is built as a collection of transition rows (description is given in the text, figure reproduced with modifications from [GDHG10, GHDG10, GHG10, GDHG11, GGHG11] with permission).



The TR-FSM implements a Mealy type FSM, i.e., the output signals depend on the current state and the input signals. In each state, the input signals are evaluated and the according state transition is activated to select the next state and the output value.

The TR-FSM is built as a collection of *transition rows* (TRs) (see Fig. 3.14). Each TR implements one transition of the FSM, i.e., one input pattern as condition, one next state, and one output pattern. The state selection gates (SSGs) enable only those TRs, which implement transitions leaving from the current state. In each TR, a subset of the n_{I} input signals is selected by the input switching matrix (ISM). The maximum number n_{T_m} of observable signals is a characteristic of each TR and is termed "width". A TR-FSM instance consists of a different number of TRs with different widths. The subset of inputs is evaluated by the input pattern gate (IPG). If it matches a given pattern, this transition is active. The IPG can match only a single pattern or a set of different patterns. Only a single TR can be active at a time, because in FSMs only a single transition is performed. The next state registers (NSRs) and the output pattern registers (OPRs) of the TRs hold the next state and output signal. The values of the active TR are selected as next state and output, respectively. At the clock edge, the next state is stored by the state register and becomes the current state.

The configurable elements of a TR-FSM comprise the SSG, ISM, IPG, NSR, and OPR. In the pre-silicon design phase, the actual implementation of the sizable TR-FSM cell is specified by the number of input signals, the number of output signals, the width of the state vector, and the

number of TRs of each width. Typically, TRs with a width of zero to four are implemented. Zero width TRs are used for sequences of consecutive states.

Since all TRs provide the same functionality (except for the number of observed input signals), the TR-FSM architecture allows to trade off the total number of states of an FSM against the number of transitions per state. The TR-FSM architecture efficiently handles transitions with unobserved input signals. Further, the mapping of an abstract FSM description to the TR-FSM configuration is a straight forward process [GGHG11].

Finally, several improvements over the original TR-FSM architecture are suggested. The signal path from the input ports to the output ports is purely combinational. While the architecture shows low propagation delay, especially the implementation of a TR-FSM using an FPGA as hardware platform, e.g., for testing purposes, can lead to unacceptably long critical paths. Therefore an optional output register was added by Martin Schmölzer in his diploma thesis [Sch14, p. 13]. The IPG is implemented as a LUT and therefore grows exponentially with the number of observed signals. Most transitions are triggered by a single or a small number of input patterns. Therefore for wide TRs this can be improved by using PLA elements with only a few product terms instead of LUTs. Another improvement for the reduction of the switching power disables the input signals of the ISM for all TRs which don't match the current state. To reduce the total number of transitions within an FSM, [RV13] uses a default next state for each state which is selected if no other transition is active.

The TR-FSM is used for the discussed methodology to implement the reconfigurable FSMs.

3.8 Scientific Contribution

In this chapter a design methodology for multi-granular, heterogeneous, application domain specific reconfigurable logic modules was developed. It covers the complete design process starting from the specification up to the final reconfigurable architecture including the mapping of actual applications and the generation of configuration data. In the scientific development, existing approaches and concepts were integrated and the following extensions beyond the state-of-the-art were contributed:

- The design methodology is the first approach which is independent of the application domain, does not make assumptions on the implemented domain, and is not limited to any domain. It introduces the development of heterogeneous reconfigurable modules which concurrently perform control-dominated and data processing tasks. The design methodology uses a set of example applications to accurately specify the reconfigurable module. The use of an HDL guarantees the universality to realize any application domain and includes the specification and implementation of cycle accurate timing.
- A novel unified multi-granular reconfigurable architecture with multi-granular functional units and multi-granular routing is introduced. It is the first reconfigurable architecture using the concept of parameterization, which is isolated from the configuration, to separate the functionality from the data. The design methodology delivers the reconfigurable module as a soft IP core which is independent of the semiconductor process and is compatible to commercial as well as free and open-source tools and to custom in-house design flows.

- An approach for the iterative optimization of the functional units with automatic identification and extraction from the example applications is introduced. The novel concept of topological variants of functional units is used to improve the automatic extraction. For the support of multi-function cells and reconfigurable cells, the concept of reduced variants is introduced.
- The design methodology is the first to provide full verification of the example applications, all intermediate steps, and the final resulting reconfigurable module. It introduces verification at the early stage of specification, including HW/SW co-simulation for driver and application development. With the generation of wrapper modules at all intermediate steps and for the resulting reconfigurable module, simulation can reuse the original testbench and logical equivalence checking can proof the correctness of the results.
- For the implementation of cycle accurate control tasks and sequences, a novel architecture for reconfigurable FSMs is introduced. The so called TR-FSM directly implements the FSM state transitions instead of the transition function and the output function. It provides a reduction of the configuration data, area, power consumption, and propagation delay compared to existing architectures.

4

Realization

In Ch. 3 the scientific development of the proposed design methodology was discussed. This methodology has been implemented as a design flow, which is discussed in the current chapter. The design flow was used to develop an SoC with a reconfigurable module to demonstrate the feasibility of the design methodology and the practicability of the design flow (see Sec. 5.1). Several examples provided in this chapter use excerpts from this design.

To fulfill the requirement of an automated design flow, a set of tools were developed and external tools are utilized. All tools are integrated to provide seamless data management and to deliver reproducible results. The tools, the data management, and the limitations of the design flow are described in Sec. 4.1.

The designed reconfigurable module is customized to the requirements of the complete SoC. Therefore the module which will instantiate the reconfigurable module, i.e., its parent module, is prepared in a first step (Sec. 4.2). From the parent module, the interfaces of the reconfigurable module are derived. The developer has to set up characteristics of these interfaces, which are defined in Sec. 4.3. Afterwards the example applications are developed and verified (Sec. 4.4). Although the development of the cells of the cell library is integrated in the "Application Analysis" step of the design methodology, it is described separately in Sec. 4.5 for clarity. This is followed by the discussion of the "Application Analysis" step in Sec. 4.6.

After all example applications are successfully extracted and the cell library is finished, the netlists are merged (Sec. 4.7). This is followed by the "Completion" step to build the infrastructure for configuration and parameterization (Sec. 4.8) and to finalize the reconfigurable module as IP core. The implementation of new applications in the post-silicon design phase is discussed in Sec. 4.9. Finally, in Sec. 4.10 a summary of the design flow is given.

4.1 Basis of the Design Flow

Analogous to the design methodology, the design flow starts with the demand for a reconfigurable module with a certain functionality. For the development and generation of the reconfigurable module, the design flow uses a heterogeneous set of tools to automate the procedure. These tools process and generate data in different formats. Additionally the user has to enter information (e.g., specifying the interfaces of the reconfigurable module) and to develop HDL designs (e.g., the example applications and the cells). All information and all HDL designs are stored in files organized in a systematic directory hierarchy which is discussed in Sec. 4.1.3.

Before the directory structure is discussed, the employed tools are introduced. The individual tools and the filesystem locations of the data files are controlled and managed by the frontend FlowCmd. The main processing and handling of the data, including the automatic generation and transformation of data files, is performed by the backend FlowProc. Both tools are discussed in Sec. 4.1.1.

Besides these main tools, the design flow uses the three tools Yosys for synthesis and extraction, TrfsmGen to manage TR-FSMs, and InterSynth to merge the example applications. These tools are discussed in Sec. 4.1.2.

Due to limited manpower, the design flow does not implement all features as defined by the design methodology. The limitations of the design flow are summarized in Sec. 4.1.4.

4.1.1 Flow Tools

The central command interface for the design flow is realized as the command line tool FlowCmd (see Fig. 4.1). It manages the paths and filenames according to the directory structure. FlowCmd is implemented as a Bash[1] script with approximately 2,300 lines of code. The user executes FlowCmd from a shell and supplies a command and optional parameters as arguments, e.g.,

```
$ flow extract -interactive
```

to perform the "Cell Extraction" step for an example application. The resulting schematic is displayed graphically and the user can further examine the synthesized design. For the creation of, for example, a new project, or an example application, an according template subdirectory tree is copied. This contains exemplary files which are edited and customized by the user. The majority of the FlowCmd commands execute other tools discussed in this section.

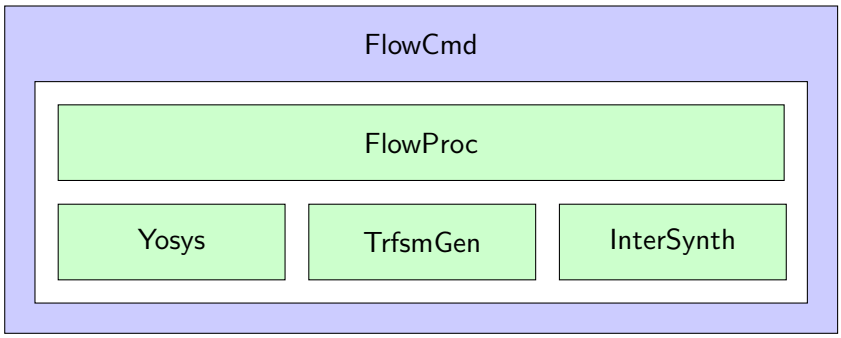
The main tool to manage and generate a reconfigurable module is FlowProc (see Fig. 4.1). It handles all information of the reconfigurable module (e.g., the configuration interface), of the example applications (e.g., their parameters), and of the cells (e.g., their interfaces). Therefore it implements a complete internal representation of these entities and their relationships. Additionally, FlowProc extensively checks all information for its validity and consistency and prints detailed information of any issues. It reads the output generated by the other tools (e.g., netlists of the example applications), writes the input for the other tools (e.g., InterSynth command files), and converts between file formats. FlowProc generates configuration data, templates for the example application HDL designs and testbenches, firmware drivers, and especially the HDL design of the reconfigurable module including the configuration and parameterization interfaces.

[1] http://www.gnu.org/software/bash/ [2015-08-20]

For the set up of the information and for the execution and automation of the tasks, a flexible and powerful method of specification is required. It must provide reproducible results and allow full customization. Commercial EDA tools face the same requirements. Many EDA tools are controlled using a Tcl scripting interface.[2] The specification of information and the execution of commands in a script ensures reproducible results, allows powerful processes, and the user can add comments with notes to explain and justify design decisions.

FlowProc is implemented as an object-oriented Pascal program[3] with approximately 25,000 lines of code. It uses an object-oriented wrapper for the Tcl libraries[4] and for the GNU Readline library.[5] To implement the functionality described above, it includes parsers for various file formats and a library to represent and generate netlists and HDL designs.

Figure 4.1: Overview of the tools employed for the design flow.



4.1.2 Embedded Tools

Besides FlowCmd and FlowProc three further tools are closely integrated in the design flow (see Fig. 4.1). As derived in Sec. 3.1.1, the example applications are developed in a HDL like VHDL or Verilog. Therefore a logic synthesis tool to translate the example applications to logic netlists is required. It must preserve multi-bit vector signals and utilize coarse-grained generic cells (cf. footnote 1 on p. 64).

These requirements are fulfilled by the Yosys Open SYnthesis Suite [Wol15b, WG13, Wol15a].[6] Yosys is free and open-source software. The control and data flow is organized using "frontends" to read designs from files, a large set of "passes" which process the design in memory, and "backends" to write the design to files. Yosys only supports Verilog and does not support VHDL.[7] Yosys internally represents the netlist with tool-specific generic cells, e.g., `$and` for an AND gate, `$mul` for a multiplier, and `$reduce_or` for a wide-input single-output OR gate. Most cells have parameters to configure the width of the input and output signals. Several passes are available

[2] The Tcl (tool command language) scripting language actually originates from tools for integrated circuit design (http://www.tcl.tk/about/history.html [2015-08-20]), although, most vendors use an in-house customized version instead of the official open-source libraries (http://www.tcl.tk/ [2015-08-20]).

[3] http://www.freepascal.org/ [2015-08-20]

[4] https://github.com/hansiglaser/pas-tcl [2015-08-20]

[5] http://www.gnu.org/software/readline/, https://github.com/hansiglaser/pas-readline [2015-08-20]; However, FlowPas is controlled with Tcl script in the design flow, therefore the interactive interface is unused.

[6] http://www.clifford.at/yosys/ [2015-08-20]

[7] In the meantime Yosys can use an external tool and a commercial library for VHDL support but these were not available during the development of the design flow.

for technology mapping of the design to a gate-level netlist. These utilize the external Berkeley ABC logic optimization tool [Ber15].

Yosys also supports FSM extraction (cf. Sec. 3.2.2) with the `fsm*` passes and "Cell Extraction" (Sec. 3.2.3) with the `extract` pass. An extracted FSM is represented using the generic FSM cell `$fsm` of the internal cell library. This is used at several places in the discussed design flow. With the flexible passes Yosys is modular and extensible. Custom passes can be implemented and included as plug-ins.

Several backends are available to save the current design in different file formats including Verilog netlist, SPICE, and EDIF. Yosys also provides the custom "ILang" file format (usually saved to files with the extension `.il`), which is designed to represent the design in full detail. FlowProc and TrfsmGen include a parser for the ILang file format to process example applications and cells.

To replace generic FSM cells with TR-FSM instances and to generate the appropriate configuration data, the tool TrfsmGen was developed. It is an object-oriented Pascal program with approximately 9,300 lines of code, additionally using parts of the FlowProc code. TrfsmGen reads a synthesized design in ILang format. For each `$fsm` cell a wrapper submodule with the same ports is created. Inside of the wrapper module a TR-FSM module (cf. Sec. 3.7) is instantiated. The `$fsm` cells also include the FSM specification as parameters (states and transitions). These FSM specifications are mapped to the configuration data of the TR-FSMs and written in a number of file formats, e.g., as VHDL vector, or as C array. This procedure is also executed as a Tcl script.

The tool InterSynth was specifically designed to perform the interconnect generation and optimization (cf. Sec. 3.3.3) [WGS+12, GW14].[8] InterSynth reads scripts in a custom language which specify connection types, cell types, netlists of example applications, and oversizing rules. Afterwards the interconnect is generated and optimized. This also includes the mapping of all example applications. Finally a Verilog module with instances of the cells and with MUXes is generated. Additionally the configuration data of all example applications as well as meta-information on the generated interconnect for the post-silicon design phase are stored.

These three tools are integrated in the design flow.[9] During the development of a reconfigurable module, additional tools are used by the designer which are directly executed and not managed by FlowCmd. Besides obvious programs like a text editor and command line tools, other tools for simulation (e.g., Mentor ModelSim/Questa Sim), logical equivalence checking (e.g., Cadence Encounter Conformal Equivalence Checking, LEC), and prototyping (FPGA tools) are used.

4.1.3 Directory Structure

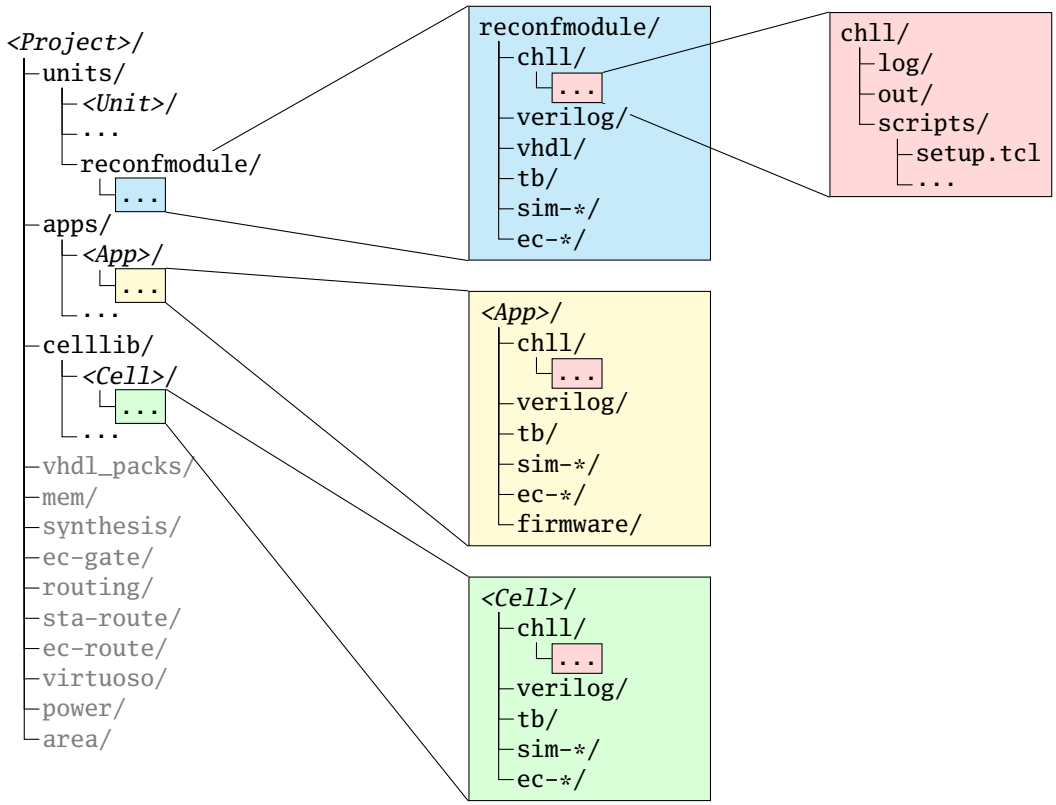
The development of the reconfigurable module involves a large number of files of different kind. Therefore a directory structure to systematically arrange these files was set up. The resulting IP core is integrated in a superordinate SoC, therefore the development of the complete SoC is supported in the directory structure. The main structure of a project is depicted in the left

[8] http://www.clifford.at/intersynth/ [2015-08-20]

[9] InterSynth was developed by Clifford Wolf as a student project, defined and supervised by the author of this thesis. Further discussions about the design methodology and possible solutions for a suitable format to specify the example applications used by InterSynth motivated Clifford Wolf to develop Yosys as a general-purpose, feature-rich, extensible, and verified synthesis tool. Since then it has gained wide-spread use in numerous open-source and commercial projects.

tree in Fig. 4.2. In the `units/` directory, the design units of the chip are placed in individual subdirectories. Each unit is a basic building block of the chip, e.g., the chip core, the CPU, a timer, and serial protocol interfaces. The units are HDL modules with optional submodules and are themselves instantiated hierarchically.

Figure 4.2: Directory structure used by the design flow (explanation in the text).



All example applications and new applications for the reconfigurable module are placed in the `apps/` directory. The cells of the cell library are placed in the `celllib/` directory. The developer can use any number of additional subdirectories, e.g., for VHDL packages, memory macros for the chip, the synthesis, etc. These optional directories are printed gray in Fig. 4.2.

The reconfigurable module generated using the discussed design flow is itself a unit and is placed in the directory `units/reconfmodule/`. The reconfigurable module unit, all applications, and all cells have a similar internal directory structure (cf. the blue, yellow, and green boxes in Fig. 4.2). In the `chll/` subdirectory all files related to the actual design flow are located. The HDL designs are placed in the subdirectories `vhdl/` and `verilog/` while the testbenches are placed in `tb/`.

For each simulation setup, a dedicated subdirectory `sim-*/` is created. For example, the main RTL simulation is performed in `sim-rtl/`, while the early HW/SW co-simulation of an example application is performed in `sim-chip-rtl/`, and the simulation of an extracted example application is performed in `sim-yosys-extract/`. Similarly, for equivalence checking the `ec-*/` subdirectories are used, e.g., to compare an extracted example application (with the `$fsm` cell replaced by a preliminary TR-FSM instance) to the original RTL design, the subdirectory `ec-yosys-extract-trfsm/` is used. For applications an additional `firmware/` subdirectory is used to develop the firmware driver.

All `chll/` directories have the same structure. The `scripts/` subdirectory contains all Tcl scripts which are edited by the user and executed by FlowProc, Yosys, and TrfsmGen. The data files and HDL designs generated by these tools as well as InterSynth are stored in the `out/` subdirectory. The screen output and reports of all runs started using FlowCmd is stored in `log/`. The reconfigurable module is completely auto-generated with no manually created designs. Therefore all subdirectories except `chll/` are unused. All other units are not related to the reconfigurable module and therefore do not have a `chll/` subdirectory and do not have to comply to the presented structure.

4.1.4 Limitations

The discussed design flow has limitations compared to the design methodology as defined in Ch. 3.

- Only a single reconfigurable module can be generated within a project. Extending the design flow to support multiple different reconfigurable modules requires the introduction of an additional level of indirection in FlowCmd.
- Due to the use of Yosys as synthesis tool, Verilog is the only supported HDL for a subset of the designs developed during the discussed design flow. However, this is not a principal restriction but only due to tool support. Further, all testbenches can be implemented in VHDL, and FlowProc and TrfsmGen can write generated designs in both, VHDL and Verilog.
- During the development of example applications, scripts to set up simulation and equivalence checking are generated with FlowProc (cf. Secs. 4.4.3 and 4.4.5). Currently the only supported tools are Mentor ModelSim/Questa Sim and Cadence Encounter Conformal Equivalence Checking (LEC).
- The only supported type of sizable cell is the reconfigurable TR-FSM.
- Currently only shift registers are supported for the storage of the configuration data.
- Additionally, for the bus interface from the CPU to the reconfigurable module only the OpenMSP430 is supported [Gir13] (cf. Sec. 4.8.2).
- The resulting IP core is not as elaborate as a commercial IP core. However, the reconfigurable module is a custom IP core and not a product intended for high reusability.
- The design flow and all tools are fully compatible to version control systems like Subversion (SVN)[10] and Git[11]. However the FlowCmd `commit-pre-si` command (cf. Sec. 4.8.3) currently only supports Subversion.

4.2 Preparation of the Parent Module

In this section the start of the design flow is discussed which prepares the definition of the reconfigurable module. In terms of a reconfigurable CPU supplement module, it is customized

[10] https://subversion.apache.org/ [2015-08-20]
[11] https://git-scm.com/ [2015-08-20]

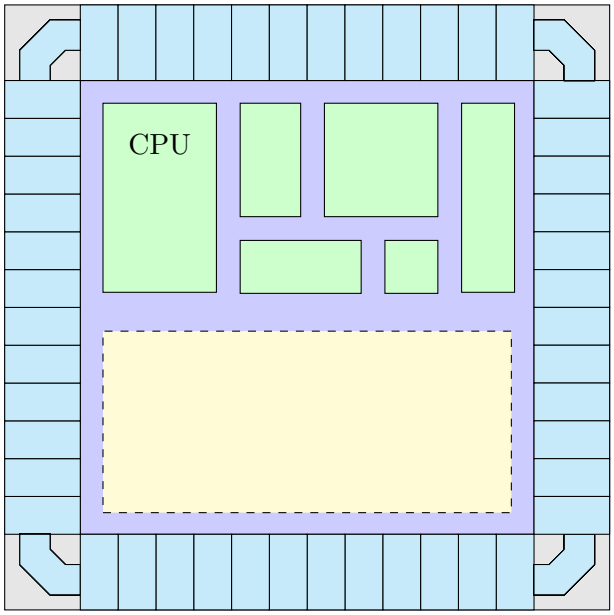
to an SoC. Therefore the SoC is first planned, including its usage scenario, its application, and its requirements. The next step is the manual HW/SW partitioning: the designer defines which tasks are performed by the firmware executed by the CPU and which tasks are performed by hardware.

Note that the hardware partition is only partly covered by the reconfigurable module. Some tasks are assigned to specialized modules like serial protocol peripherals (e.g., SPI, I^2C), ADC, and RF network physical layer. From the hardware tasks assigned to the reconfigurable module, its application class is defined (cf. Fig. 3.2 on p. 57). Note that the application class is a verbal description of the reconfigurable module used by the designer in the further design process.

The reconfigurable module will be instantiated in a higher-level module which is called its "*parent module*". The reconfigurable module is directly customized to this parent module, i.e., its ports are connected to other modules and to ports of its parent module. Therefore, in contrast to commercial IP cores, where the integrator adapts the surrounding module to the IP core, here the reconfigurable module IP core is customized to its surrounding.

To begin the design flow, a new project is created and a template directory structure is copied to the project directory by FlowCmd. Afterwards the designer plans and manually develops the parent module with all ports and the instances of all submodules. This also includes the preparation of the interfaces between the CPU and the reconfigurable module, e.g., a bus interface, a small number of "internal" GPIO signals for direct signaling, and interrupt request signals. For example, the parent module can be the whole chip core module and contain a CPU, memories, timers, GPIO, and serial protocol peripherals (see Fig. 4.3). Some of the modules are exclusively used by the CPU, and some are provided for the reconfigurable module. The reconfigurable module itself is still missing (dotted yellow rectangle).

Figure 4.3: Exemplary SoC with the chip core (blue) surrounded by the pad frame (cyan pad and corner cells). The SoC contains a CPU and several other modules like RAM, Flash, Timer, and GPIO (green). The dashed yellow rectangle denotes the demand for the reconfigurable module.



During the development of the parent module, the developer is in no way limited by the design flow, including naming conventions, data types, modules, in-house design guidelines, etc. The only current limitation is the use of Verilog for the parent module because it is synthesized using Yosys (see Sec. 4.3.1). All other modules, including the instantiated modules, are not limited to Verilog, because in the parent module only their interfaces are relevant.

To enable the later instantiation of the reconfigurable module, a file, which will be auto-generated during the development of the reconfigurable module, is included in the parent module (see Lst. 4.1). However, this is optionally disabled using the macro `ReconfModuleNone`.[12] The result of the described preparation is the verbal definition of the application class and the implementation of the parent module which still misses the reconfigurable module.

Listing 4.1: Verilog fragment used in the parent module to include a generated file with the instantiation of the reconfigurable module.

```
// ...
`ifndef ReconfModuleNone
  `include "reconflogic-instance.inc.v"
`endif
// ...
```

4.3 Definition of the Reconfigurable Module

The specification of the reconfigurable module comprises its functionality (defined by example applications) and its interfaces (customized to the parent module). In this section, the definition of the interfaces is discussed. The development of example applications is discussed in the next section.

After the parent module is developed as discussed in the previous section, the directory tree for the reconfigurable module is created with the FlowCmd `new-reconf-module` command.

The interfaces of the reconfigurable module are automatically derived from the parent module. Then the user has to specify properties of the individual signals. This is discussed in Sec. 4.3.1. Besides the reconfigurable logic, the reconfigurable module also includes infrastructure for configuration and parameterization, which is discussed in Sec. 4.3.2. Finally in Sec. 4.3.3 the automatic generation of the source code for the instantiation of the reconfigurable module and other files is discussed.

4.3.1 Setup of the Reconfigurable Signals

In this section the definition of the interfaces, i.e., the ports, of the reconfigurable module is discussed. To avoid the error-prone and tedious task to manually set up the list of all inputs and outputs, these ports are automatically derived from the parent module. In the parent module, the reconfigurable module is missing (cf. Fig. 4.3) which results in unused signals. To find these

[12] Note that the logic is inverted: The macro *disables* the inclusion of the instantiation, because commercial EDA tools later used in the design flow, which should include the file, might not support the definition of macros, while Yosys properly supports this feature.

signals, the parent module is synthesized using Yosys with the macro `ReconfModuleNone` defined, i.e., the `‘include` statement is disabled. Then the `stubnets.so` plugin is utilized to identify all signals without a driver or without a sink.

Afterwards the netlist of the parent module (used to identify the types of the signals) and the list of the unused signals are read by FlowProc.[13] Note that the automatically detected list of unused signals is not complete. Some inputs and outputs of modules can be intentionally unused. Further, some signals can have both, a driver and a sink, but should be branched and used as input of the reconfigurable module, e.g., the global clock and reset signals. Therefore individual signals are removed from or added to the list. This procedure is programmed with a set of Tcl scripts which are customized by the developer. The finally adjusted signals are termed "*reconfigurable signals*".

As next step the *connection types* are defined (cf. Sec. 3.1.4) and assigned to the reconfigurable signals. However, not all reconfigurable signals should be routed via the interconnect. For example, to adhere to a clean synchronous logic design, the clock and reset signals are directly connected to the cells. Further, some reconfigurable signals should be directly set by configuration, e.g., the clock polarity of an SPI SCK signal, or directly set by parameterization, e.g., a clock divider to set the frequency of the SPI SCK signal.

To accommodate this requirement, "*usage types*" as generalization and hierarchically higher level as connection types are introduced. In the design flow, five usage types are defined:

- `dynamic`: The reconfigurable signal is routed via the interconnect, i.e., it is connected as an input or output to the interconnect (cf. Fig. 3.3 on p. 59). These are the "normal" reconfigurable signals which were assumed in the development of the design methodology. Only reconfigurable signals with this usage type are assigned a connection type. The other four usage types are an extension to the design methodology introduced for the design flow.

- `direct`: The reconfigurable signal is directly connected to the cells, e.g., clock and reset signals.

- `config`: Outputs of the reconfigurable module are directly driven by a separate configuration store to supply the values.

- `param`: Inputs and outputs of the reconfigurable module are directly connected to the parameterization infrastructure and can be read or written as parameters by the CPU.

- `const`: Outputs of the reconfigurable module are directly driven by a constant value. This usage type is expected to be used scarcely and is only provided for completeness.

Signals with a usage type `config`, `param`, and `const` are not connected to the interconnect or any cells but are input and output ports of the reconfigurable module. In the final automatic generation of the reconfigurable module the appropriate infrastructure to connect all reconfigurable signals depending on their usage type is generated.

[13] The unused signals could also be detected by FlowProc but the `stubnets.so` plugin was previously developed for this task by Clifford Wolf.

4.3.2 Setup of the Reconfigurable Module

In the "Merge" step of the design methodology (cf. Sec. 3.3), the reconfigurable module is generated with instances of the cells and a reconfigurable interconnect. In the design flow discussed in this chapter, InterSynth is used for this task, which produces a Verilog HDL module. More details are discussed in Sec. 4.7.2.

In the "Completion" step (cf. Sec. 3.4) the reconfigurable module is extended with the infrastructure for configuration and parameterization (cf. Fig. 3.10 on p. 76). However, instead of the automatic modification of the HDL source code generated by InterSynth, a wrapper module is generated. This contains an instance of the unmodified module generated by InterSynth (subsequently termed "*interconnect module*") as well as all other logic required for configuration and parameterization and for the reconfigurable signals with a usage type `config`, `param`, and `const`. The term "*reconfigurable module*" is subsequently used for this wrapper module.

Additionally to the reconfigurable signals, the reconfigurable module requires an interface to its infrastructure for configuration and parameterization. Typically this is implemented as memory-mapped peripheral and accessed via a bus from the CPU. With the specification of that interface, the entity, i.e., the input and output ports, of the reconfigurable module is completely defined. However, at this early stage the number of configuration stores and their size as well as the number of parameterization registers is still undefined. More details are discussed in Sec. 4.8.

The Tcl scripts with the setup of the reconfigurable signals and of the reconfigurable module are executed by FlowProc at the beginning of the scripts used in most of the following steps to prepare the respective tasks. For example, for the early HW/SW co-simulation of example applications, a wrapper module is generated, which has the exact same ports as the final reconfigurable module. The definition of the reconfigurable signals and the reconfigurable module are required by FlowProc to generate this wrapper module.

4.3.3 Generation of HDL Modules

During the preparation of the parent module, the instantiation of the final reconfigurable module is deferred using the Verilog `` `include `` statement to include a generated file with the instantiation as described in Sec. 4.2. The next step of the design flow is the generation of that include-file `reconflogic-instance.inc.v`.

To assist the developer, additional files and HDL modules can be generated by FlowProc. If the parent module is the chip core module (blue in Fig. 4.3 on p. 91), the higher-level module (gray in Fig. 4.3) which instantiates the pad cells (cyan in Fig. 4.3) can be generated. For this task the library with the pad cells provided by the chip manufacturer in the Liberty format [Lib15] is loaded by FlowProc. For each port of the parent module, the developer specifies the appropriate pad cell.

Note that all ports of modules inside the chip including the core module have to be defined as either input or output, because no bidirectional or undriven signals are allowed. To implement pins of the chip with special behavior like open-drain or bidirectional ports, e.g., used as GPIOs or for I^2C, separate input, output, and enable signals are internally used. For the instantiation of such pad cells, these separate signals are explicitly stated (see Lst. 4.2).

Another scenario during the development is to test the reconfigurable module with real hardware, e.g., an SPI sensor, instead of simulation models of these chips. This test is best carried out using

Listing 4.2: Definition of a bidirectional pin "P1_b" and an open-drain pin "I2CSDA_b" for the chip top-level module. Note that "P1_b" is actually an 8 bit wide port, therefore eight individual pad cells are automatically instantiated. Further, the pad library does not contain I^2C pads, therefore a bidirectional output pad is instantiated with its output value set to '0' and its enable input controlled with the inverted I2CSDA_o output signal.

```
# ...
chip_add_pin -pad_cell "BBC16P" -out "P1_DOut_o" -enable "P1_En_o" -in "P1_DIn_i" -portname "P1_b"
# ...
chip_add_pin -pad_cell "BBC16P" -out '0' -enable_n "I2CSDA_o" -in "I2CSDA_i" -portname "I2CSDA_b"
# ...
```

an FPGA to implement the reconfigurable module and its parent module and to connect it to the other chips (cf. "Verification with Prototype Testing" in Sec. 5.1.3). For this scenario, another top-level module can be generated. Input and output ports are directly connected and ports with special behavior are implemented using assignments with tri-state values 'Z'. The FPGA implementation tools translate these constructs to the appropriate control signals for the FPGA pad cells.

The generated HDL modules are prepared here at the beginning of the design process of the reconfigurable module for the use at end of the design process when the reconfigurable module is finished. However, these modules are also required at an early stage for the development of the example applications, which is discussed in the next section.

4.4 Development of Example Applications

The functionality of the reconfigurable module is specified using a set of example applications. In this section, the development of these example applications is discussed. Example applications are developed as common HDL logic designs. All signals, including parameters, are implemented as ports of the logic module. The first step is the creation and set up using the defined reconfigurable signals and connection types (Sec. 4.4.1). Then the developer manually develops and verifies the example application (Sec. 4.4.2).

Before the design flow is continued, the developer checks the example application for its suitability for the further tasks (Sec. 4.4.3). Additionally a firmware driver is implemented (Sec. 4.4.4) and HW/SW co-simulation is used for its verification (Sec. 4.4.5). The result of this process is the HDL design of the example application, which is functionally verified and suitable for the further design flow, and a firmware driver.

4.4.1 Setup of Example Application

As first step of the design of an example application, the FlowCmd `new-app` command is used to create a new example application. This creates a subdirectory with the appropriate structure and template files (represented by the yellow box in Fig. 4.4). Afterwards the developer manually customizes the setup script `./chll/scripts/setup.tcl` (① in Fig. 4.4).

Lst. 4.3 shows an exemplary setup script for the example application "ADT7310". This script is executed by FlowProc at the beginning of each task which processes this example application to

set up its properties. Before that script the scripts which define the reconfigurable signals and the reconfigurable module are executed. The FlowProc Tcl command `create_application` registers a new application. The input and output ports are declared using `app_add_port`. These must be chosen from the list of reconfigurable signals. However, using the `-map` parameter, the port can use a different name (e.g., the reconfigurable signal `SPI_DataOut` coming from an SPI master peripheral in the parent module is used as an input port named `SPI_Data_i`). Additionally, the `-index` parameter is used to select single bits of vector signals like `ReconfModuleIn_s`, which is an 8 bit wide GPIO output of the CPU dedicated for direct signaling with the reconfigurable module.

The command `app_add_param` is used to add parameters. Note that here the direction and connection type are explicitly specified. For the reconfigurable signals this information is already set up. Reconfigurable signals which should carry a constant value while the example application is active are defined using the `app_set_port_value` command. The actual implementation in the reconfigurable module depends on the usage type of that signal.

After the developer has finished the setup script, the FlowCmd `app-templates` command uses FlowProc to create templates HDL modules for the example application and its testbench (② in Fig. 4.4). The example application HDL file contains the module and port definitions and assignments of the constant values as set with `app_set_port_value` for signals with a usage type `dynamic`. The testbench HDL file contains an instantiation of the example application module, proper handling of constant reconfigurable signals, clock signal generation, and a template stimulus process. Note that the testbench can be implemented in VHDL because it is not synthesized with Yosys.

Further, template files for the firmware driver are created, which are discussed in Sec. 4.4.4. To prepare HW/SW co-simulation (cf. Sec. 4.4.5), a wrapper module is generated, to instantiate the example application HDL module as the reconfigurable module.

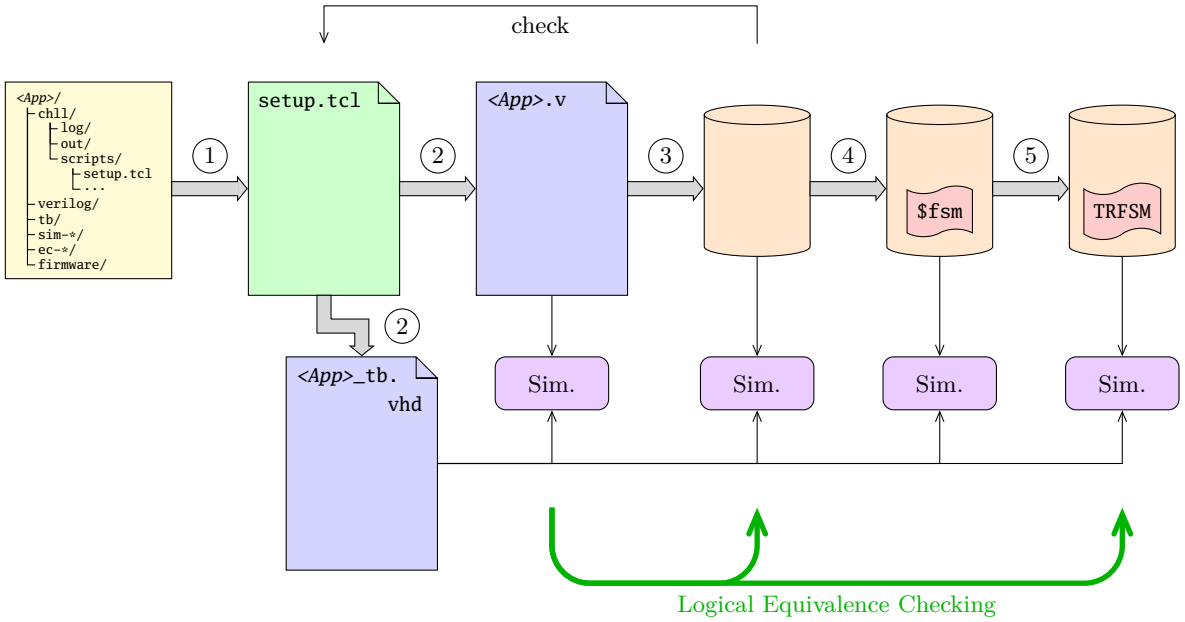
4.4.2 Development and Verification

After the example application is set up and the template files are generated, the developer manually designs the HDL module. This development is identical to common logic design. The design flow does not put any restrictions on the developer, e.g., on naming conventions or signal types. He can adhere to in-house design guidelines and utilize familiar methods. The ports of the example application are fixed, but the internal logic design can be fully customized. It is also possible to use submodules for the implementation of the example application.

The development of the example application also includes verification of the HDL module (cf. Sec. 3.5). The main approach for functional verification is simulation. The testbench instantiates the HDL module and supplies stimulus input signals and examines the output signals (depicted in the left-most “Sim.” box in Fig. 4.4, which corresponds to the top “Sim.” box in Fig. 3.11 on p. 77). The design flow does not include any dedicated facilities for simulation except from the generated testbench template. The developer can utilize any suitable simulation tool. Besides simulation, the developer can also utilize methods of formal verification.

To summarize, the development and verification of the example applications allows full freedom to comply with any in-house design guidelines and with the developer’s experience.

Figure 4.4: Development of example applications (explanation in the text).



Listing 4.3: Exemplary `setup.tcl` file of an example application (explanation in the text).

```
# define application
create_application "ADT7310"
# system signals
app_add_port "Reset_n_i"
app_add_port "Clk_i"
# interface to CPU
app_add_port "Enable_i"      -map "ReconfModuleIn_s"   -index 0
app_add_port "CpuIntr_o"     -map "ReconfModuleIRQs_s" -index 0
# interface to sensor
app_add_port "ADT7310CS_n_o" -map "Outputs_o" -index 0
# parameters
app_add_param -in -conntype "Word" -default 0 "SPICounterPreset_i"
# ...
# result value
app_add_param -out -conntype "Word" "SensorValue_o"
# interface to SPI master
app_set_port_value "SPI_LSBFE" '0'
# ...
app_set_port_value "SPI_SPPR_SPR" {"00000000"}
app_add_port "SPI_Data_i"         -map "SPI_DataOut"
app_add_port "SPI_Write_o"        -map "SPI_Write"
# ...
```

4.4.3 Check Example Application

After the example application is developed and functionally verified, its suitability for the further design flow is checked. Therefore the example application is synthesized with Yosys (③ in Fig. 4.4). Before that, the developer optionally customizes the template synthesis script, especially to read in all individual files of submodules. The resulting netlist is loaded by FlowProc to check the ports against the definition in the setup script.

Afterwards the FSMs in the example application logic design are extracted (cf. Sec. 3.2.2 and ④ in Fig. 4.4) which results in netlists with instances of the generic FSM cell `$fsm`. Finally, TrfsmGen is used to replace the generic FSM cells with TR-FSMs (⑤ in Fig. 4.4). Additionally the configuration data to implement the according functionality is generated. Note that these TR-FSM instances are preliminary and only for testing purposes. During the cell instantiation of the "Merge" step (cf. Sec. 4.7.1) the final TR-FSM instances suitable for all example applications are instantiated.

Each of the three different netlists has the same ports and the same behavior as the original HDL module. Therefore the original testbench can be used for simulation (see Fig. 4.4). The pure logic netlist and the netlist with the generic FSM cells are self-contained, while the netlist with the TR-FSMs requires additional configuration data to fully specify its functionality. Before the start of the simulation, this configuration data has to be applied to the TR-FSM instances. Instead of simulating the download of the configuration data using the configuration ports, this is performed using simulation setup scripts generated with TrfsmGen. These directly set the values of all configuration stores before the start of the simulation. Currently only the generation of ModelSim/Questa Sim scripts with `force -freeze` *<object_name>* *<value>* commands is supported.

Besides simulation, the design flow provides facilities for logical equivalence checking to compare the generated netlists with the original HDL design (see Fig. 4.4). The generic FSM cell `$fsm` is a custom Yosys extension and therefore this netlist can not be used in logical equivalence checking. Similarly to simulation, when TR-FSMs are included, the configuration data has to be applied. For that purpose TrfsmGen is used to generate setup scripts for Cadence LEC with `add instance constraint` *<0|1>* *<instance_pathname>* `-revised` commands.

One additional issue in logical equivalence checking are the state encodings and the state registers of FSMs. These are likely different between the original HDL design and the TR-FSMs. To inform the logical equivalence checking tool about the different state encoding, a state encoding translation file is generated by TrfsmGen and processed by Cadence LEC with the `read fsm encoding` *<filename>* `-golden` command.[14]

4.4.4 Firmware Development

The developed reconfigurable module is used as CPU supplement module. To initialize the reconfigurable module and to control the example application implemented with it, a firmware driver is required. The firmware driver is specific for each example application and is therefore developed manually. It defines functions and interrupt service routines (ISRs) to interact with the

[14] Besides Cadence LEC also Synopsys Formality was evaluated for logical equivalence checking. Unfortunately the reencoding of FSMs in Formality is not flexible enough for TR-FSMs. This is the main reason why only Cadence LEC is supported for logical equivalence checking.

running example application, e.g., via "internal" GPIOs for direct signaling, via parameters, and via interrupts. The main firmware program utilizes this driver and implements the functionality of the complete SoC.

Before the actual operation of the example application, the reconfigurable module is initialized, i.e., the configuration data is applied and the parameters are set. The initialization code for each example application is generated during the "Completion" step of the reconfigurable module (cf. Sec. 4.8.3) and invoked by the according driver. It includes the configuration data generated with InterSynth, FlowProc, and TrfsmGen for all configuration stores. The generated code itself relies on further drivers specific to the configuration and parameterization interfaces of the reconfigurable module.

The initialization code as well as the manually developed firmware driver access the individual configuration stores, the parameterization registers, and reference elements of vector signals (cf. `Enable_i` and `CpuIntr_o` in Lst. 4.3). Therefore an additional include-file with constants for indexes is generated.

The firmware is used in the manufactured SoC, but also for testing the SoC design in an FPGA, for power analysis of the SoC design to generate representative switching activity, etc.

4.4.5 HW/SW Co-Simulation

To verify the firmware driver and its interaction with the example application, the discussed design flow provides mechanisms for HW/SW co-simulation. For the implemented approach, the complete chip or chip core module, including an HDL design of the CPU and including the reconfigurable module, is simulated. The complete chip design or the core module is instantiated in a testbench. This is only slightly modified compared to the original testbench used to verify the example application. Before the simulation is started, the firmware is loaded into the program memory. This corresponds to the programming of the flash memory of the manufactured chip. The same method as discussed in Sec. 4.4.3 to apply the configuration data for TR-FSMs is used.

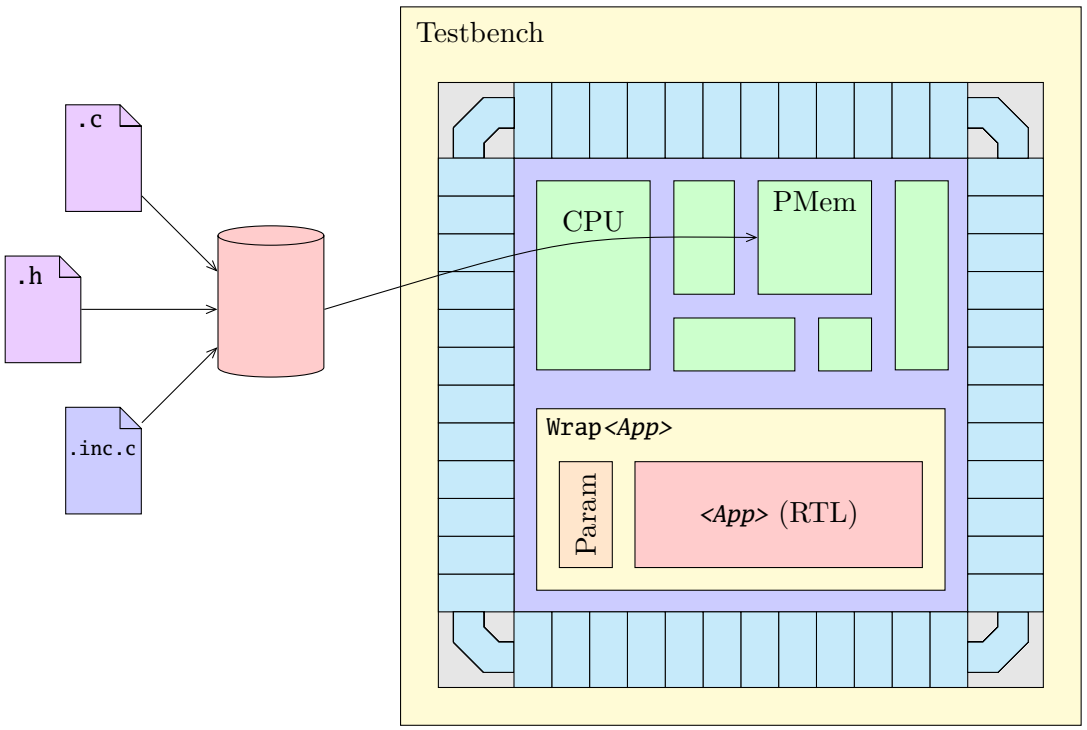
However, this setup for HW/SW co-simulation requires the final reconfigurable module, which is completed late in the design process. To allow early development of the firmware, the design flow additionally provides *early* HW/SW co-simulation, i.e., during the development of the example application and therefore before the reconfigurable module is generated.

For this scenario, the example application is instantiated in the parent module instead of the reconfigurable module. For this purpose, a wrapper module is generated directly after the setup of the example application (cf. Sec. 4.4.1). It has the same ports as the final reconfigurable module and internally instantiates the example application HDL design together with preliminary infrastructure for parameterization (see Fig. 4.5). The example application HDL design itself does not require configuration and the reconfigurable signals with usage type `config` are implemented as constant assignments in the wrapper module with the values defined in the setup. Therefore no configuration infrastructure is included.

This setup also requires a slightly different initialization procedure executed by the firmware driver. It does not include configuration data and uses preliminary indexes for the parameterization registers. Its source code is also generated after the setup of the example application. The manually developed driver functions are not changed for the early HW/SW co-simulation.

Note that this setup for the early HW/SW co-simulation can also be used to test the complete design in an FPGA as outlined in Sec. 4.3.3.

Figure 4.5: For the early HW/SW Co-Simulation of an example application, its RTL design (<App>) is wrapped in a module with the same interface as the reconfigurable module (Wrap<App>) and instantiated in the parent module, e.g., the chip core. The complete chip or its core are instantiated in the simulation testbench. Before the simulation is started, the firmware (violet sheets represent the sources which are compiled to a binary file represented by the red cylinder) is loaded in the program memory (PMem).



4.5 Development of Cells

After all example applications have been developed, the "Application Analysis" step of the design methodology is performed (cf. Sec. 3.2 and Fig. 3.5 on p. 65). One main task is the development of cells. For clarity, the related features and tasks of the design flow are discussed in this section, separately from the "Application Analysis" step.

In the design flow the development of cells is implemented similarly to the development of example applications. The setup, development, and verification of a cell is discussed in Sec. 4.5.1. For each cell additional topological variants and reduced variants can be implemented (Sec. 4.5.2). Finally, all cells are assembled in the cell library (Sec. 4.5.3). Cells developed in a previous project using the discussed design flow can be reused and added to the cell library of a new project.

4.5.1 Setup and Development of Cells

For the development of a new cell, the directory hierarchy and template files are created with the FlowCmd `new-cell` command. Analogous to the development of example applications, afterwards the developer customizes the `setup.tcl` script. Lst. 4.4 shows an exemplary setup script of the cell "`Byte2WordSel`", which was used as an example for a reconfigurable cell in Sec. 3.2.5 to merge two 8-bit values into a 16-bit value and select a subsection of that.

The command `create_cell` creates a new in-memory representation of a cell in FlowProc. With the command `cell_add_port` input and output ports and configuration ports are declared. In the final reconfigurable modules, the ports `Reset_n_i` and `Clk_i` will be directly connected to reconfigurable signals with usage type `direct` without going through the interconnect.[15] The ports `H_i`, `L_i`, and `Y_o` will be routed through the reconfigurable interconnect and are therefore assigned a connection type. The ports `Shift_i` and `Mask_i` will be supplied with configuration data, four bits each.

Listing 4.4: Exemplary `setup.tcl` file of the cell "`Byte2WordSel`" which is used as an example for a reconfigurable cell in Sec. 3.2.5.

```
# define cell
create_cell "Byte2WordSel"
# add ports
cell_add_port "Reset_n_i" -map "Reset_n_i"
cell_add_port "Clk_i"     -map "Clk_i"
cell_add_port "H_i"       -in  -conntype "Byte"
cell_add_port "L_i"       -in  -conntype "Byte"
cell_add_port "Shift_i"   -config -width 4
cell_add_port "Mask_i"    -config -width 4
cell_add_port "Y_o"       -out -conntype "Word"
```

After the customization, the FlowCmd `cell-templates` command executes the `setup.tcl` script, among others, with FlowProc to generate template files for the HDL module and the testbench. Then the developer manually designs the cell, including verification and checking for suitability for the further design flow. However, in contrast to example applications (cf. Fig. 4.4 on p. 97), FSM extraction and TR-FSM insertion are not applicable for cells.

In this section, the setup and the development of the *main variant* of the cell was discussed. In the next section, the development of topological variants and reduced variants is explained.

4.5.2 Topological Variants and Reduced Variants

In Sec. 3.2.4 the concept of topological variants and reduced variants was introduced. It is based on the division between identification of a subcircuit and the replacement of that subcircuit by the actual implementation (see Fig. 4.6).

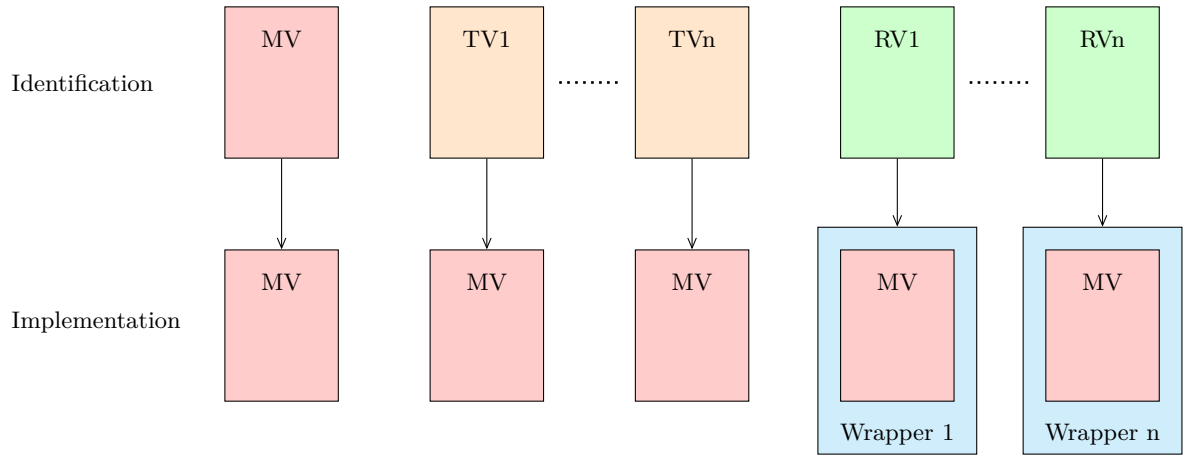
For the implementation of *topological variants*, the developer creates additional HDL modules with the same ports as the main variant but with a different logic design. During cell extraction, this module is only used for identification and is replaced by the main variant (TV1, ..., TVn in Fig. 4.6) as its implementation. The easiest and most practical way is to simply copy the Verilog file of the main variant and change its logic contents.

For the implementation of *reduced variants*, the developer has to create two HDL modules:

- The first module is only used for identification. It can have custom ports and implements the functionality of the reduced variant (RV1, ..., RVn in Fig. 4.6).

[15] Actually these two ports are included in this example only for demonstration purposes because the described module is better implemented purely combinational.

Figure 4.6: For each cell one main variant and an arbitrary number of topological variants and reduced variants are implemented. For the identification of the cells in the (example) applications, all variants are used. These are replaced by the main variant for the implementation. The reduced variants are replaced by wrapper modules which internally instantiate the main variant. (MV: main variant, TV: topological variant, RV: reduced variant).



- The second module is used to replace the identified subcircuit. It is a wrapper module with the same ports as the reduced variant which instantiates the main variant together with the possibly additionally required logic.

The same approach is used for *reconfigurable cells*.

Afterwards, the Verilog files of all variants are added to the `synthesize.tcl` script. This script instructs Yosys to synthesize the main variant, all topological variants, and all reduced variants including their wrapper modules. Additionally, simple wrapper modules for all topological variants are generated which are used for their replacement by the main variant. Finally, the Yosys built-in SAT solver[16] is used to check the logical equivalence of each topological variant with the main variant and of each reduced variant with its according wrapper module.

4.5.3 Building the Cell Library

After a new cell is created, it is added to the cell library and provided for the cell extraction. The developer adds the cell to the global list of cells[17] and uses the FlowCmd `check-celllib` command to generate two files:

- In the first file the netlists of all variants of all cells are collected, each in a separate module, which are used for the subcircuit extraction.
- In the second file the netlists of all wrapper modules are collected, which are used to replace the identified subcircuits with the main variant and possibly required additional logic.

[16] http://www.clifford.at/yosys/cmd_sat.html [2015-08-20]
[17] realized as a Tcl list located in the script `./units/reconfmodule/chll/scripts/setup-celllib-arr.tcl`

During the "Application Analysis" step, additional topological variants and reduced variants can be added as required. The design flow does not stipulate the developer to complete a cell and all its variants at a time.

One special cell in the cell library is the TR-FSM (cf. Sec. 3.7). Contrary to the other cells, it is not automatically detected in the "Cell Extraction" step, but introduced in a two-step procedure via "FSM Extraction" and without subcircuit extraction. Therefore the TR-FSM does not have to comply to the directory structure and setup of the other cells. It is included as finished and verified HDL module in the cell library which allows to adjust the number of its inputs and outputs, width of the state signal, and the number of transition rows of varying width. TrfsmGen is used to generate customized wrapper modules for the TR-FSM design and to generate its configuration data.

4.6 Application Analysis

In Sec. 3.2 and Fig. 3.5 on p. 65 the "Application Analysis" step of the design methodology was developed. It is an iterative and repetitive process, split into the steps "Synthesis", "FSM Extraction", "Cell Extraction", "Inspection", and "Improvement". While the first three steps are performed automatically, the last two steps require the specific human intelligence, intuition, and experience. The input of the "Application Analysis" is a set of example applications and possibly a set of cells reused from a previous project. The results are a netlist for each example application and the cell library. The resulting netlists only instantiate cells from the cell library and generic FSM cells.

The following discussion only considers the procedure and the iterations for a single example application. However, the sequence is performed multiple times within the outer iteration cycles for all example applications. Additionally the individual steps can be interleaved between all example applications.

The "*Synthesis*' and the "*FSM Extraction*" steps are implemented in the same way as already used for the verification, whether the example application is suitable for the further design flow (cf. Sec. 4.4.3 and Fig. 4.4 on p. 97). These steps are identical to the steps ③ and ④, and result in a netlist with `$fsm` cells.

This is followed by the "*Cell Extraction*" step which is implemented using the Yosys `extract` pass (cf. Sec. 3.2.3). First the netlist of the example application is loaded. Then the `extract` pass uses the first file defined in Sec. 4.5.3 with the collection of netlists of all variants of all cells of the cell library to identify the subcircuits and replace each occurrence with an instance of its according module.

Afterwards, the `techmap` pass is used to replace these instances by the modules defined in the second file with the wrapper modules. If the `extract` pass identified the main variant, this module is retained. Instances of the topological variants are directly replaced by the main variant using the automatically generated wrapper modules. Instances of reduced variants and reconfigurable cells are replaced by the manually designed wrapper modules. This might introduce some additional logic. Therefore in a further step the logic can be merged into the FSMs or replaced by a cell using a second execution of the `extract` pass.

This extraction procedure is fully automated. Afterwards, the resulting netlist is visualized for the "*Inspection*" step (cf. Fig. 5.1 on p. 124). The developer inspects the schematic of the (partially) extracted netlist to

- identify groups of generic cells which are candidates for implementation using a new cell, to
- identify problems during extraction, e.g., subtle differences in the implementation of a cell and the example application, and finally to
- verify that the netlist is completely implemented using only cells of the cell library and no remaining generic cells except `$fsm` are present.

The developer can also inspect the original netlist at the beginning of the "Application Analysis" step to identify initial cells.

In the "*Improvement*" step the developer creates new cells, adds topological variants or reduced variants to existing cells, or slightly modifies the example application to achieve better extraction results.

As mentioned above, the developer can manually verify that the extraction is complete. However, this is better checked automatically. Therefore the design flow implements the FlowCmd `check-extract` command which executes FlowProc. The setup scripts of the example application and of all cells are executed to set up the internal representation of these elements. Then the netlist of the extracted example application is loaded and checked. Only the defined cells and `$fsm` generic FSM cells and only connections of signals and ports with consistent usage types and connection types are allowed.

When the example application is completely extracted, the result has to be verified (cf. Sec. 3.5). The netlist has the same input and output ports as the original HDL design, therefore, the original testbench is used for simulation. This is shown with the second "Sim." box from top in Fig. 3.11 on p. 77. For logical equivalence checking, which corresponds to the top-most green "Equiv. Check." arrow, first the `$fsm` generic FSM cells are replaced by preliminary TR-FSM implementations and the required configuration data is generated. The configuration for other reconfigurable cells, e.g., the `Byte2WordSel` used as an example above, is included in the extracted netlist as constant value assignment.

Each example application is appended to the list of example applications in a Tcl script similar to the list of cells. The design flow also allows to use the "Application Analysis" step even before the full set of example applications is designed. The developer can add new example applications at any time.

Finally, when all example applications are extracted, these are processed concurrently and statistics on the resource usage is reported (see Lst. 5.1 on p. 122). This also includes the requirements for the common TR-FSMs (number of inputs, outputs, states, and transitions). That information is used in the Cell Instantiation step of the following "Merge" step, which is discussed in the next section.

4.7 Merge to Reconfigurable Architecture

When all example applications are developed and successfully extracted, in the "Merge" step the reconfigurable module is generated (cf. Sec. 3.3 and Fig. 3.4 on p. 62). More precisely, the interconnect module as defined in Sec. 4.3.2 is generated. The input of this step are the extracted netlists of the example applications. The result is an RTL HDL design of the interconnect module, configuration data for all example applications, and an internal description of the interconnect.

To generate the interconnect module, first the sizable cells and the oversizing rules are defined (Sec. 4.7.1). This is followed by the automated optimization of the interconnect (Sec. 4.7.2). Finally, the result is verified to correctly implement each example application (Sec. 4.7.3).

4.7.1 Cell Instantiation

Before the interconnect optimization, the implementations of sizable cells and the oversizing rules are defined (cf. Sec. 3.3.1). Currently the design flow only supports TR-FSMs and does not support other types of sizable cells. The input of the cell instantiation are the extracted netlists of all example applications which contain `$fsm` generic FSM cells. The result are the netlists with the `$fsm` cells replaced by TR-FSM wrapper modules and the configuration data.

The developer inspects the resource utilization reports generated at the end of the "Application Analysis" step and specifies the number and parameters of the TR-FSM physical instances. Then the FlowCmd `insert-trfsms` command is used to execute TrfsmGen to create the TR-FSM wrapper modules according to the specified parameters. The `$fsm` generic FSM cells in the netlists of all example applications are automatically mapped to and replaced by the adequate physical instances with the appropriate size and the configuration data is generated. This is analogous to the preliminary TR-FSM insertion performed for a single example application as discussed in Sec. 4.4.3. Afterwards the resulting netlists are verified using simulation and logical equivalence checking.

The second task of the cell instantiation step is to specify the *oversizing rules* to increase the flexibility of the final reconfigurable module. These are stored in the file `./units/reconfmodule/chll/-scripts/presilicon.txt` which is processed by InterSynth. Lst. 4.5 illustrates the oversizing rules. The commands specify rules which are applied by InterSynth during the interconnect optimization, e.g., the minimum number of physical instances of cell types (`min`), the number of instances added to the automatically determined number of minimum instances (`abs`), and the relative number by which the minimum number is multiplied (`rel`). Besides rules for instances of cells (`cellup`), also rules to increase the interconnect resources are used (`switchup`).

4.7.2 Interconnect Optimization

For the interconnect optimization (cf. Sec. 3.3.3) InterSynth is used. All script files for InterSynth are automatically generated with FlowProc, except `presilicon.txt`, which specifies the oversizing rules as shown in the previous section. These scripts define the connection types, the ports of the interconnect module, the cell types, and the netlists of the example applications. Special constructs are introduced to handle cells with internal configuration stores, cells with configuration inputs, constant values in the netlist, and parameters.

Listing 4.5: Exemplary oversizing rules specified for InterSynth. For each connection type (`Bit`, `Byte`, and `Word`) the interconnect switches should use one more up and down connection than required. For each of the two different TR-FSM sizes at least one instance is included. Finally, for the cell types `CONST_Byte`, `WordMuxDual`, etc., the minimum number of instances is specified.

```
### Oversizing rules ##################################################################
# headroom switchup <conntype> [abs <num>] [rel <num>] [min <num>] [depth <num>] [up|down]
# headroom cellup   <celltype> [abs <num>] [rel <num>] [min <num>]

# add one up and one down connection for each switch
headroom switchup Bit  abs 1
headroom switchup Byte abs 1
headroom switchup Word abs 1

# one instance of each TR-FSM size
headroom cellup TRFSM0 min 1
headroom cellup TRFSM1 min 1

# more instances of useful cells
headroom cellup CONST_Byte        min 6
headroom cellup WordMuxDual       min 2
headroom cellup WordRegister      min 3
headroom cellup CONST_Word        min 1
headroom cellup CellParamOut_Word min 2
```

InterSynth processes the scripts, optimizes the interconnect, and generates a Verilog file of the interconnect module which instantiates the cells and implements the interconnect. The module has the specified ports for the reconfigurable signals with usage types `dynamic` and `direct` plus an additional input `bitdata`, which is a wide vector to supply the configuration data. Further, it instantiates the HDL designs of the cells, i.e., the main variant. Therefore, in all further steps including the final chip synthesis, the HDL designs of the cells are used. The example applications and all netlists generated with Yosys are only intermediate products of the design flow.

A section of the result of InterSynth is illustrated in Lst. 4.6. It shows an instance named `cell_62` of the cell type `Counter` with the signals connected to its ports. In the second half, the MUX for the signal `cell_62_4`, which is connected to the input `PresetVal_i` of the counter, is shown. The design implements two parallel trees for the connection type `Word`, therefore the outputs of two MUXes are combined with a logical OR. The bitstream ensures that the unused MUX contributes an all-zeros vector. Each MUX selects between the output signals from other cells and from a switch at a higher tree level.

Besides the Verilog HDL design, InterSynth also generates files which specify the structure and the cell-to-leaf-mapping of the interconnect tree for the mapping of new applications in the post-silicon design phase (cf. Sec. 4.9). Additionally, for each example application the configuration data, a listing of the node-to-cell-mapping, images of the interconnect with the routed signals (cf. Fig. 5.2 on p. 126), and an image of the netlist are generated.

Listing 4.6: Typical Verilog output generated by InterSynth showing an instantiated cell and a MUX (explanation in the text).

```
// ...
// Counter[1]
wire [15:0] cell_62_5; // D_o
wire cell_62_3; // Direction_i
wire cell_62_2; // Enable_i
wire cell_62_6; // Overflow_o
wire [15:0] cell_62_4; // PresetVal_i
wire cell_62_1; // Preset_i
wire cell_62_0; // ResetSig_i
wire cell_62_7; // Zero_o
Counter cell_62 (
        .D_o(cell_62_5),
        .Direction_i(cell_62_3),
        .Enable_i(cell_62_2),
        .Overflow_o(cell_62_6),
        .PresetVal_i(cell_62_4),
        .Preset_i(cell_62_1),
        .ResetSig_i(cell_62_0),
        .Zero_o(cell_62_7),
        .Clk_i(Clk_i),
        .Reset_n_i(Reset_n_i)
);
// ...
assign cell_62_4 =
   (bitdata[1054:1052] == 3'b101 ? sw_2_0_3_down0 :
    bitdata[1054:1052] == 3'b100 ? cell_63_6 :
    bitdata[1054:1052] == 3'b010 ? cell_63_7 :
    bitdata[1054:1052] == 3'b110 ? cell_93_0 :
    bitdata[1054:1052] == 3'b001 ? cell_94_0 :
    bitdata[1054:1052] == 3'b0 ? 16'b0 : 16'bx) |
   (bitdata[1203:1201] == 3'b001 ? sw_2_1_8_down0 :
    bitdata[1203:1201] == 3'b100 ? cell_61_5 :
    bitdata[1203:1201] == 3'b010 ? cell_64_6 :
    bitdata[1203:1201] == 3'b110 ? cell_64_7 :
    bitdata[1203:1201] == 3'b0 ? 16'b0 : 16'bx);
// ...
```

4.7.3 Verification

The verification of the results generated by InterSynth is performed in three steps: sanity checks, simulation, and logical equivalence checking. First, all results generated by InterSynth (except the Verilog module) are loaded with FlowProc and verified. This includes the connection types, the cell types, and the ports. Additionally, wrapper modules and setup scripts for the following two steps are generated. Finally, Yosys is used to synthesize the Verilog design to an actual ASIC standard cell library and to estimate the chip area.

The second step of the verification is the *simulation* of each example application implemented with the interconnect module (cf. Sec. 3.5 and the third "Sim." box in Fig. 3.11 on p. 77). Therefore in the previous step for each example application a wrapper module is generated, that has the

same ports as the original example application. It instantiates the interconnect module, assigns a constant value with the configuration data to its `bitdata` input, and connects its ports to the wrapper module ports or assigns constant all-zero values to unused inputs. This wrapper is then instantiated in the original testbench of the example application.

Note that two kinds of wrapper modules for the example applications are used in the design flow:

- The first kind of wrapper modules is used for HW/SW co-simulation. It instantiates the original example application HDL design to create a preliminary reconfigurable module.
- The second kind of wrapper modules is reversed: The interconnect module (as in this section) and the reconfigurable module (see Sec. 4.8.4) are wrapped to create a surrogate module of the original example application HDL design.

The original testbench can access signals across the module hierarchy from inside the example application design using Verilog and VHDL external names. In the interconnect module, these signals have different drivers and sinks. To use the same testbench for both simulation setups, the external name statements are placed in a separate module named "`extnames`". This is instantiated in the testbench and provides the internal signals as its ports. This allows to reuse the original testbench while only different implementations of the "`extnames`" module are instantiated. The "`extnames`" module used for the example application HDL design is created manually. The implementation used to simulate the interconnect module is generated with a Tcl script executed by FlowProc using the information on the node-to-cell-mapping to access the renamed signals.

For simulation, the testbench, all cells of the cell library including the TR-FSM and its wrapper modules, the interconnect module, the wrapper module, and the module to access internal signals are compiled. The configuration data of the interconnect module is hard-wired as a constant value in the wrapper module. This includes the configuration of the interconnect and of all cells with configuration inputs. The configuration of cells with internal configuration stores, especially the TR-FSMs, are applied before the start of the simulation using startup scripts as described in Sec. 4.4.3.

The third step of the verification is the comparison of the interconnect module with each example application using *logical equivalence checking*. This is illustrated with the green arrow from the example applications to the output of the "Merge" box in Fig. 3.11 on p. 77. The original example application HDL module is used as golden design. The revised design consists of the wrapper module, the interconnect module, and the cells of the cell library. The procedure is analogous to that used in Sec. 4.4.3 including the setup of the configuration data for cells with internal configuration stores and the FSM encoding.

4.8 Completion of the Reconfigurable Module

With the verified interconnect module, the reconfigurable module is completed (cf. Sec. 3.4) and is provided as IP core. First, in Sec. 4.8.1 the exact deliverables for the IP core are defined. As reasoned in Sec. 4.3.2, the reconfigurable module is a wrapper for the interconnect module. Details on the infrastructure in the reconfigurable module to support the interconnect module and the final module hierarchy are discussed in Sec. 4.8.2. In Sec. 4.8.3 the actual generation of the reconfigurable module as HDL design and the specific problems and solutions are described.

After the reconfigurable module is generated, it is verified to implement each example application (Sec. 4.8.4). Finally, the completed and verified reconfigurable module is integrated in an SoC. The associated steps and tools are briefly summarized in Sec. 4.8.5.

4.8.1 Deliverables

The result of the design flow is an IP core of the reconfigurable module for the integration in an SoC. In this section the exact deliverables are defined (cf. Sec. 3.1.2). These comprise the following parts:

a) HDL design of the reconfigurable module

b) meta-information for the integration in the SoC

c) run-time data of each example application

d) meta-information to develop new applications in the post-silicon design phase

ad a) The HDL design of the reconfigurable module consists of the module itself, the modules used for the configuration and parameterization infrastructure (defined in the next section), the interconnect module, and the HDL designs of the cells of the cell library including the TR-FSM and its wrapper modules.

ad b) The meta-information for the integration in the SoC consists of instructions for the synthesis and the layout (place and route). For the synthesis, a list of all files of the HDL design and information to define the constraints are required.

- High fan-out nets are treated as ideal networks during synthesis and are implemented with dedicated buffer trees during place and route.
- Manually gated clocks also have to be treated in a similar way. Both cases are especially relevant for the configuration infrastructure.
- One major problem during synthesis and the accompanied static timing analysis (STA) arises from combinational logic loops through the reconfigurable interconnect. This was also reported by [WKW+05, WAWS03] (cf. Sec. 2.3.3). [WKW+05] used a directional architecture which only allows connections in one direction to solve this problem. However, this is not practical for this thesis. In Sec. 2.3.4 and in [WGS+12, GW14] the use of timing constraints to break these loops is suggested. This requires information on the interconnect structure, on the instantiated cells, and on their ports.
- When all combinational logic loops are disabled for the STA, separate timing constraints for combinational cells and for the paths through the reconfigurable interconnect are required.
- During synthesis, all cells which directly connect an output to an input port (e.g., to convert between connection types or cells for constant values) are ungrouped to avoid the automatic insertion of redundant buffers and to allow improved logic optimization.

The deliverables have to include information to appropriately handle these cases.

ad c) The run-time data of each example application comprises the configuration data and the firmware. During compilation, the configuration data is embedded in the firmware binaries (cf.

Sec. 4.4.4). At run-time, the driver applies the configuration data to the reconfigurable module and subsequently activates its functionality.

ad d) The meta-information to develop new applications in the post-silicon design phase includes:

- the definition of the reconfigurable signals and the reconfigurable module (cf. Sec. 4.3),
- the cell library with all topological variants and reduced variants,
- the specification of all sizable cells which were used for the cell instantiation (cf. Sec. 4.7.1),
- information on the structure and the cell-to-leaf-mapping of the interconnect tree to map new applications to the existing interconnect module (cf. Sec. 4.7.2), and
- constants and functions for the configuration and parameterization in the firmware.

The deliverables include a large set of different kinds of information which is dispersed across the directory structure. Therefore the design flow does not implement the creation of a stand-alone package of the IP core. Instead, the complete directory structure is delivered as IP core.

4.8.2 Reconfigurable Module Contents

The reconfigurable module is set up at the beginning of the design process (cf. Sec. 4.3.2). However, the details on the infrastructure for configuration and parameterization and the interface to the CPU were left open. These details are developed in this section.

The infrastructure for configuration comprises the configuration stores and the logic to apply the configuration data. The configuration stores can be implemented as EEPROM/Flash memory, as SRAM, or as shift register using D-FFs. The configuration data can be applied from firmware using memory-mapped registers via a CPU peripheral bus interface to the reconfigurable module (e.g., OpenMSP430 External Peripheral Interface, [Gir13, p. 15], ARM AMBA AXI used in the Xilinx Zynq-7000 FPGA [CEES14, p. 353ff]), or with an external interface (e.g., in-system programming with a special hardware tool via JTAG). The current implementation of the design flow only supports shift registers as configuration stores and the OpenMSP430 CPU peripheral bus as its interface. However, the implementation allows a simple extension to other options. In the remainder of this section, only the supported option is discussed.

The infrastructure for parameterization is always accessed from the CPU to set and query values during run-time with memory-mapped register via a peripheral bus.

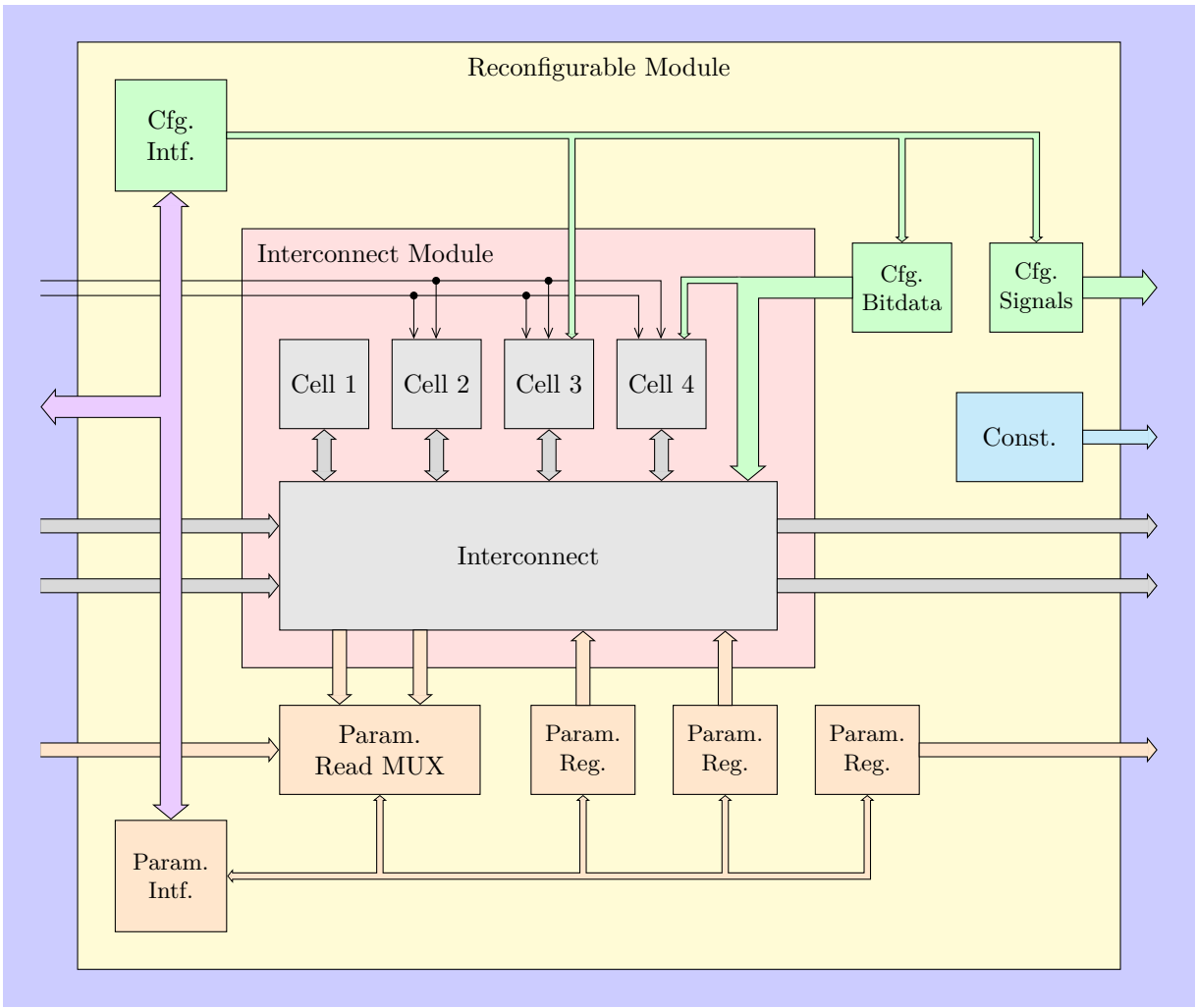
In the setup of the reconfigurable module the interface from the CPU to the infrastructure for configuration and parameterization is defined. For the discussed design flow the assumption is made, that both, the configuration and the parameterization, are accessed via the same type of interface (i.e., CPU peripheral bus). Therefore both use the same signals of the bus interface and only differ in the address of the according memory-mapped registers. To reduce the complexity, dedicated modules for the configuration interface and for the parameterization interface are introduced in the reconfigurable module. These separate the interface from the storage and convert the CPU interface to a simpler internal interface.

The final module hierarchy is shown in Fig. 4.7. The reconfigurable module (yellow) is embedded in its parent module (blue). Its main contents is the interconnect module (red) with the

cells and the interconnect (gray). All cells are connected to the interconnect. Cells 2–4 also use reconfigurable signals from the parent module with the usage type `direct` (thin arrows). Cell 3 additionally has an internal configuration store and is therefore connected to the configuration interface (green). Cell 4 is also reconfigurable but uses the configuration data included in the common configuration of the interconnect module. This is provided by the "Bitdata" configuration store. The "Signals" configuration store provides the values of the reconfigurable signals with the usage type `config`.

The reconfigurable signals with the usage type `const` are directly driven with constant assignments from the reconfigurable module (cyan). The interconnect is also connected to reconfigurable signals with the usage type `dynamic` (gray arrows from and to the parent module) and to parameters (orange arrows). For parameters written by the CPU and used by the reconfigurable module, parameter registers are instantiated (orange). For parameters provided by the reconfigurable module and read by the CPU, a read MUX is used. Reconfigurable signals with the usage type `param` are handled analogously. The parameters are accessed from the CPU via the parameterization interface. Together with the configuration interface, it is connected to the CPU via a peripheral bus interface (violet arrow).

Figure 4.7: Module hierarchy of the reconfigurable module (explanation in the text).



4.8.3 Generation

In the previous section the contents of the reconfigurable module was derived. In this section its automated generation is discussed. Based on the setup and on the results of the previous steps, a complete internal representation of the reconfigurable module is prepared in FlowProc. The "Bitdata" configuration store to provide the configuration data for the interconnect module is instantiated. The size of the configuration data is determined from the output of InterSynth.

The "Signals" configuration store is instantiated, which holds the values for the reconfigurable signals with the usage type `config`. For each reconfigurable signal a section of the configuration data is allocated. The total size results from the sum of all reconfigurable signals. For the cells with internal configuration stores, the according sizes are set up during the cell instantiation.

Parameter registers are instantiated for the according inputs of the interconnect module and for the reconfigurable signals with the usage type `param`, which are driven by the reconfigurable module. The parameter read MUX is instantiated for the according outputs of the interconnect module and for the reconfigurable signals with the usage type `param`, which are inputs of the reconfigurable module.

At this point the list of configuration stores and of parameters is complete, therefore indexes for the individual access from the CPU are allocated. Finally the interface to the CPU is constructed.

With the complete internal representation of the reconfigurable module, an HDL module is generated. FlowProc supports the generation of Verilog and VHDL designs of the reconfigurable module. Besides the generation of the above structures, additional details have to be implemented. Reconfigurable signals with a differing width have to be converted to the according connection type, either by padding with zeros (e.g., a 10-bit ADC value input to a 16-bit vector of connection type `Word`) or by subranges (e.g., an output with connection type `Byte` with eight bits to a four bit wide read count signal for an I^2C master). All signals have to be connected between the modules and with the ports of the generated reconfigurable module.

The configuration infrastructure using shift-registers results in a large clock tree to supply the clock signal to each D-FF. For a low-power design (cf. Sec. 2.1.2), an individual manual clock gating cell is instantiated for each configuration store. Therefore, during normal operation, the according sections of the clock tree are disabled. During configuration, only the section of the clock tree of the current configuration store is enabled.

Besides the reconfigurable module, additional files are generated individually for each example application. For the verification of the reconfigurable module, wrapper modules and setup scripts are generated (see next section).

As mentioned in Secs. 4.4.4 and 4.4.5, a preliminary driver is used for the firmware development and for the early HW/SW co-simulation. Here the final firmware driver is generated with the above mentioned final indexes for configuration stores and parameters. This also includes the complete configuration data: the configuration data for the reconfigurable signals with usage type `config`, for the cells with internal configuration stores, and for the interconnect module. The developer has to recompile the firmware, e.g., by using different settings for the `Makefile`. It is not necessary to modify the manually developed parts of the firmware driver because the details on the configuration and parameterization are encapsulated in the auto-generated driver.

In the post-silicon design phase the tools rely on the ample set of information which was set up and generated in the pre-silicon design phase. If any piece is lost after the SoC is produced, the

development of new applications is impossible. Therefore all original files and all generated files have to be safely stored. The FlowCmd `commit-pre-si` command checks that all information is up-to-date and commits the final reconfigurable module to a version control system. Currently only Subversion (SVN) is supported. Additionally a file is generated which specifies that the project directory is switched to the post-silicon design phase. This disables most FlowCmd commands to prevent accidental modifications of the generated reconfigurable module or any files it is based on.

4.8.4 Verification

The verification of the generated reconfigurable module is performed individually for each example application and comprises three approaches: simulation, logical equivalence checking, and HW/SW co-simulation.

The *simulation* of the reconfigurable module corresponds to the fourth "Sim." box at the "IP Core" in Fig. 3.11 on p. 77. It is similar to the verification of the interconnect module in Sec. 4.7.3 with minor difference. A wrapper module is generated with the same inputs and outputs as the original example application. It instantiates the reconfigurable module and connects the signals with usage type `dynamic` and `direct`. However, parameters of the example applications, which are also ports of its HDL module, are realized as parameter registers and a parameter read MUX, which are embedded inside of the reconfigurable module. To access these internal signals, VHDL-2008 external names [AL07] are used in the wrapper module.[18] Additionally special VHDL configurations are required to disconnect the drivers of the parameter registers inside the reconfigurable module from these signals, which are driven externally in this setup.

The simulation setup scripts include the configuration data for the interconnect module and for the reconfigurable signals with usage type `config` additional to the configuration data of cells with an internal configuration store. These measures ensure that the original testbench of the example application is applicable.

The verification of the reconfigurable module using *logical equivalence checking* to compare it with the example application HDL design is also similar to the procedure discussed in Sec. 4.7.3 with minor difference. This corresponds to the green arrow from the example applications to the "IP Core" box in Fig. 3.11 on p. 77. The employed tool Cadence Encounter Conformal Equivalence Checking (LEC) does not support the features of the VHDL-2008 standard. Instead, setup scripts are generated which expose internal signals of the reconfigurable module as primary inputs and outputs and which define the appropriate mapping points between the golden and the revised netlists. With the complete configuration data applied, this setup allows to verify the logical equivalence of the configured reconfigurable module with the original example application HDL design.

The third approach for verification is *HW/SW co-simulation*. This is similar to the procedure developed in Sec. 4.4.5. Instead of the wrapper module (cf. Fig. 4.5 on p. 100) the final reconfigurable module is instantiated in its parent module. Additionally, the final firmware including all configuration data is used.

[18] Note that in the discussion of the verification of the interconnect module (cf. Sec. 4.7.3), external names were used in the testbench to access internal signals. Here external names are additionally used in the wrapper module of the unit under test.

4.8.5 SoC Integration and Implementation

The generated and verified IP core of the reconfigurable module is integrated in the SoC (cf. Fig. 3.11 on p. 77). The SoC is implemented using synthesis, place and route, and sign-off verification before the production data is delivered (tape-out) and the SoC is produced. At the initial operation of the produced SoC, the developed firmware is directly applicable. However, contrary to HW/SW co-simulation, it has to be transferred to the program memory (e.g., flash memory) with in-system programming.

The methods of verification during the implementation of the SoC include post-synthesis (i.e., gate-level) and post-place-and-route simulation. The netlists instantiate the semiconductor process specific standard cells. Additionally the estimated delays of the cells and the routing can be considered using standard delay format (SDF) files. In all setups the same testbench and the same firmware as used for the HW/SW co-simulation are applicable. During the simulation, the signal values and the switching activity can be recorded in value change dump (VCD) and switching activity interchange format (SAIF) files, respectively. This can be used for power analysis.

The above mentioned estimated delays in SDF files are generated using static timing analysis (STA). This loads the netlist of the SoC and parasitics information generated during place and route. To estimate the power consumption of the SoC or of the reconfigurable module itself, power analysis is used. For example, this feature is included in Synopsys Design Compiler and accessed with the `report_power` command. It utilizes the switching activity for all signals of the SoC as determined with simulation. More details on power analysis are discussed in Sec. 5.3.3.

4.9 Post-Silicon Design Phase

After the reconfigurable module is generated and verified, the project directory is switched to the post-silicon design phase (cf. Sec. 4.8.3). This prevents changes on the reconfigurable module but allows the development of new applications (cf. Sec. 3.6 and Fig. 3.12 on p. 80). The development process is similar to the procedures used in the pre-silicon design phase. Therefore several of the FlowCmd commands support the `-post-si` argument.

First the new application is developed, including verification, firmware development, and early HW/SW co-simulation. In the discussed design flow this is implemented identically as in the pre-silicon design phase. After the new application is developed, the “Application Analysis” step follows. This is also identical to the pre-silicon design phase except that no new cells can be created. However, modifications of the new application as well as new topological variants and new reduced variants are allowed, because these do not imply changes of the silicon circuit. If the new application can not be fully extracted, it can not be implemented with the reconfigurable module.

For new applications, the “Merge” step is replaced by mapping the extracted netlist to the existing reconfigurable module. This step first uses TrfsmGen to map the generic FSM cells to TR-FSMs. If the TR-FSMs integrated in the reconfigurable module do not provide the necessary resources for the generic FSM cells, the example application can not be implemented. Afterwards InterSynth is used for the node-to-cell-mapping and routing. It relies on the information on the structure and the cell-to-leaf-mapping of the interconnect tree stored in the pre-silicon design phase. At this

point, the resources provided by the reconfigurable module can also preclude the implementation of the new application.

The next step is the verification of the new application implemented by the interconnect module alone and by the reconfigurable module. For both cases the according wrapper modules and setup scripts are generated. The verification is performed identical to the procedure described in Secs. 4.7.3 and 4.8.4.

Finally the firmware with the included configuration data is deployed to the SoC and executes the new application.

4.10 Summary

In this chapter a design flow based on the design methodology for the development of reconfigurable modules introduced in Ch. 3 was presented. It starts with the demand for a reconfigurable module and with the development of its parent module. The parent module is evaluated to extract the ports of the reconfigurable module. Then a set of example applications is developed as the specification of its functionality. These example applications are analyzed and an optimized cell library is developed. The example applications implemented using only the cell library are afterwards merged to form the interconnect module. This is further encapsulated to include the infrastructure for configuration and parameterization to form the reconfigurable module. After the chip production, the design flow supports the development of new applications and their implementation using the reconfigurable module.

The design flow provides a high degree of automation. It combines a set of tools and ensures seamless integration. The designer is relieved from error-prone and monotonic tasks by the automatic generation of template files, wrapper modules, and setup scripts. The design flow ensures reproducible and validated results by the use of simulation and logical equivalence checking at crucial points throughout the design process. It is independent of the realized application class and does not introduce any limitations, apart from those listed in Sec. 4.1.4. However, these are mostly missing features due to time constraints and can be easily implemented.

In the next chapter the feasibility of the design methodology and the practicability of the design flow are demonstrated by designing an SoC including a reconfigurable module. This is evaluated for its power consumption, chip area, and the size of the configuration data.

5

Evaluation and Results

In this chapter, the developed design methodology, the design flow, and the results obtained from their utilization are evaluated quantitatively and qualitatively. First, the four hypotheses defined in Sec. 1.2 are evaluated.

In Sec. 5.1 the feasibility of the design methodology (hypothesis 1) is evaluated. Therefore a reconfigurable module and an SoC were developed and manufactured. Hypothesis 2 compares the power consumption of a reconfigurable module to a CPU. At an early stage of the design methodology a different reconfigurable module was manually developed for this comparison. That module is presented in Sec. 5.2. In Sec. 5.3 the evaluation of the power consumption of the SoC and the evaluation of hypothesis 2 are presented.

Hypothesis 3 compares the chip area of the reconfigurable module with the parallel implementation of the example applications and with eFPGA implementations. The chip area and the hypothesis are evaluated in Sec. 5.4. In Sec. 5.5 the size of the configuration data of the reconfigurable module is compared to eFPGA implementations to evaluate hypothesis 4.

After the evaluation of the four hypotheses, the design methodology is further evaluated for qualitative measures including correctness of results, productivity, and flexibility (see Sec. 5.6). This is followed by Sec. 5.7 with an examination of the design methodology regarding the requirements defined in Sec. 1.1.7. Additionally, in Sec. 5.8 the TR-FSM architecture is evaluated. Finally the results are thoroughly discussed in Sec. 5.9.

5.1 Feasibility of the Design Methodology

In this section, hypothesis 1 "Feasibility of Design Methodology" as defined in Sec. 1.2 is evaluated. Therefore, the design methodology and its realization as a design flow are utilized to design a reconfigurable module which is integrated in an SoC. This procedure is used to test whether the design methodology is applicable and produces valid results which are suitable as an IP core.

As concrete application, a low-power WSN SoC with a reconfigurable module for an autonomous sensor interface is chosen. This demand for a reconfigurable module is the starting point of the design methodology (cf. Sec. 5.1.1). As first step, the SoC is specified and implemented in Sec. 5.1.2. Then the development of the reconfigurable module is discussed in Sec. 5.1.3. Its integration, the further development of the complete SoC, the manufacturing of a test chip, and the characteristics of the SoC are presented in Sec. 5.1.4. The reconfigurable module itself is briefly characterized in Sec. 5.1.5. Further details are discussed in Secs. 5.3–5.6 and in appendix A. Finally, the actual evaluation of hypothesis 1 is discussed in Sec. 5.1.6.

5.1.1 Demand for a Reconfigurable Module

The WSN SoC should be similar to an MCU with an added reconfigurable module. With manual HW/SW partitioning, the tasks of the WSN SoC are split between its CPU and its reconfigurable module. The communication via the wireless network using suitable MAC and routing layer protocols is assigned to the firmware. This also includes the transmission of the measured sensor values to a central station.

The application of the reconfigurable module is the autonomous handling of external sensors and to perform the sensor interface task as described in Sec. 1.1.1. To support analog and digital sensors, an ADC and bus masters for the SPI and I^2C protocols are required. However, these modules are not part of the reconfigurable module but instantiated in its parent module and controlled by the reconfigurable module. Additionally, digital input and output signals to control the external sensors are required. Further, the post-processing of the retrieved measurement values is included in the application class. This includes the detection of a change and the calculation of the mean value of consecutive measurements.

Besides the interfaces towards the external sensor, the reconfigurable module requires interfaces to and from the inner of the WSN SoC. Additional to the reset and clock signals and the configuration and parameterization interface, an interrupt request signal to the CPU, digital signals for single-bit communication (e.g., to enable and disable the configured application), and control and data signals to the ADC and the serial bus masters are required.

As the application class sets requirements for the SoC (included modules, signals, pins), the SoC sets requirements for the reconfigurable module (ports, interface to the CPU, etc.). Therefore in the next section the according details of the SoC are discussed briefly. This is followed by a detailed discussion of the application of the design methodology and design flow for the development of the reconfigurable module.

5.1.2 Wireless Sensor Network SoC

The first step of the design flow is the specification and development of the parent module. For the discussed WSN SoC, the parent module is the complete chip core (see Fig. 4.3 on p. 91). The WSN

SoC should be similar to the Texas Instruments MSP430F1232 microcontroller [Tex04]. This chip is an ultra-low-power MCU and is used in various WSN nodes, e.g., the TinyMote [MB04]. It was also used in Sec. 2.1.1 and will be used in Sec. 5.3 for comparison. The MSP430F1232 includes 8 kB Flash, 256 bytes RAM, three 8-bit GPIO ports, a 10-bit ADC, a watchdog timer, a 16-bit timer, and a universal synchronous/asynchronous receive transmit (USART) module.

For testing purposes and practical reasons, restrictions and simplifications are employed:

- The WSN SoC does not contain RF communication.
- Instead of an integrated ADC module, its control signals are implemented as pins of the chip. This allows to externally emulate an ADC and to simulate exact values to test the behavior of the (example) application in relation to defined changes of the measured value.
- Further, no watchdog timer is required.

The WSN SoC uses the OpenMSP430[1] free and open-source CPU design, which is fully compatible to the commercial products [Gir13]. For debugging, the included debug interface is configured to use the I^2C protocol (instead of UART). For the production of a test chip, only static RAM (SRAM) but no Flash memory was available. Therefore the program memory is implemented as SRAM and set up via the debug interface before the operation of the WSN SoC.

The OpenMSP430 provides up to six 8-bit GPIO ports compatible with the commercial product series. The MSP430F1232 implements three 8-bit GPIO ports. To save chip area, the WSN SoC implements two 8-bit GPIO ports and uses its third GPIO port as internal signals for direct single-bit communication with the reconfigurable module. Additionally, the special functions of the GPIO pins (e.g., timer signals) are implemented identical to the commercial product, where applicable.

The OpenMSP430 also provides a compatible "TimerA" module which is included in the WSN SoC. To communicate with the WSN SoC, a UART is required. The OpenMSP430 project does not provide a USART module compatible with the commercial MCUs. Therefore, a simplified UART module, which is included in an example project,[2] is used.

For the firmware implementation of the sensor interface task, a simple SPI master peripheral termed "`SimpleSPI`" was implemented. This module is also suitable for a later extension to communicate with an external RF module. The ADC interface and the serial bus masters used by the reconfigurable module (see below) use direct signaling instead of a peripheral bus interface and are therefore not suitable for the OpenMSP430 CPU.

The OpenMSP430 supports the same low-power modes as implemented by the commercial MSP-430 MCUs. For their discussion, the clocking scheme of the WSN SoC is described. The main clock signal, which enters the WSN SoC, is distributed to the OpenMSP430 CPU and the reconfigurable module and its peripherals. The OpenMSP430 CPU internally creates three gated clocks:

- MClk is used internally by the CPU, its debug interface, and the RAMs.
- SMClk is used by the CPU peripherals: GPIO, TimerA, UART, SimpleSPI.
- AClk is unused.

All clocks are directly gated from the main clock signal without clock dividers. Further, no independent clock domains are implemented. The GPIO module needs the SMClk signal to

[1] `http://opencores.org/project,openmsp430` [2015-08-20]

[2] The simplified UART model is located at `./fpga/xilinx_avnet_lx9microbard/rtl/verilog/omsp_uart.v` in the OpenMSP430 source tree.

synchronize incoming signals and the TimerA module needs the SMClk signal to synchronize the external INClk and TAClk signals, i.e., these signals are not implemented as separate clock domains.

Low-power mode 0 (LPM0) switches off the MClk signal and therefore deactivates the CPU and the RAMs, while the peripherals are still active and can activate the CPU using interrupts. Due to the simplified SoC design, LPM1 is equivalent to LPM0. LPM2 additionally switches off SMClk and therefore deactivates the CPU peripherals GPIO, TimerA, UART, and SimpleSPI. LPM3 and LPM4 are equivalent to LPM2 because the affected systems are not included in the WSN SoC.

For the *reconfigurable module*, dedicated SPI and an I^2C bus master peripherals are included in the SoC design.[3] Note that the SimpleSPI peripheral is connected to the OpenMSP430 CPU via its External Peripheral Interface while the SPI bus master peripheral is connected to the reconfigurable module via control and data signals. Additionally, pins for the external ADC and direct digital input and output pins to control external sensors are included. As mentioned above, the third 8-bit GPIO port of the OpenMSP430 is used for single-bit communication between the CPU and the reconfigurable module. Further, five interrupt request signals and the OpenMSP430 peripheral bus for the configuration and parameterization interfaces of the reconfigurable module are prepared.

The mentioned modules and signals are implemented in the chip core design together with the Verilog `‘include` definition for the instantiation of the generated reconfigurable module (cf. Lst. 4.1 on p. 92). For the verification of the chip core design without the reconfigurable module, a substitute module was created which drives all signals with a constant value. This was used to simulate various test cases, e.g., a blinking LED, low-power modes, and RAM tests. Additionally, the configuration and parameterization interfaces of the reconfigurable module and the according firmware drivers were developed and verified as peripherals connected via the OpenMSP430 peripheral bus.

In this section, the development of the chip core module as the parent module was described. The requirement of the design flow to develop the parent module before the reconfigurable module emphasizes a clean top-down design strategy.

5.1.3 Design of the Reconfigurable Module

After the development of the parent module, the reconfigurable signals were set up (cf. Sec. 4.3.1). For the reconfigurable module, the three connection types “`Bit`”, “`Byte`”, and “`Word`” were defined. The design flow automatically recognizes unused signals and issues errors for missing or duplicate usage type and connection type definitions. To select the signals in these definitions, regular expressions enable a precise and economic specification.

The next step is the set up of the reconfigurable module (cf. Sec. 4.3.2). The interfaces for the configuration and parameterization infrastructure are specified as memory-mapped peripherals connected via the OpenMSP430 bus interface.

[3] These peripherals were developed by students supervised by the author of this thesis.

Example Applications

To specify the functionality of the reconfigurable module, six *example applications* were manually developed as Verilog RTL designs (cf. Sec. 4.4): "`ADT7310`", "`MAX6682`", "`MAX6682Mean`", "`ADT7410`", "`ExtADC`", and "`SlowADT7410`". The names reflect the employed sensors: The Analog Devices ADT7310 16-bit temperature sensor and the Maxim Integrated MAX6682 10-bit temperature sensor provide an SPI interface. The Analog Devices ADT7410 16-bit temperature sensor provides an I^2C interface. The "`ExtADC`" example application uses an external ADC.

"`MAX6682Mean`" periodically performs four consecutive measurements, sums the result values and then notifies the CPU. The division by four (e.g., implemented as a shift-right operation) to acquire the mean value is delegated to the firmware.[4] The other five example applications periodically perform a single measurement and compare the value to the previous value. Only if the difference is above a parameterized threshold, the new value is stored and the CPU is notified, analogous to the sensor interface task described in Sec. 1.1.1.[5]

The ADT7310 and ADT7410 sensors require two separate requests to initiate the measurement and, after a delay of at least 240 ms, to query the value. The MAX6682 sensor continuously performs measurements and the latest value is returned on each SPI transfer.

The delays in all example applications are implemented using counters. The delay values are set up using parameterization. The "`ADT7410`" and the "`SlowADT7410`" example only differ in the width of the counters. The former uses two 16-bit counters while the latter uses two 32-bit counters.

All example applications except "`ExtADC`" use two FSMs. One FSM controls the communication via the employed serial bus master and stores the returned sensor values to "`Byte`" registers. The second FSM controls the first FSM and implements the datapath for the sensor value post-processing. The "`ExtADC`" example application is implemented using a single FSM.

For each example application, a separate subdirectory was created according to the directory structure (cf. Sec. 4.1.3). Then the `setup.tcl` script was customized to define the ports, including the parameters. Listing 4.3 on p. 97 shows a slightly simplified excerpt of the setup script for the example application "`ADT7310`". The design flow supports the development of the example applications by the automatic generation of a Verilog HDL template and a testbench template. These were used to develop and verify the example applications.

After functional verification, the Verilog HDL designs were synthesized and automatically checked against the specification in `setup.tcl`. Then the FSMs were extracted and replaced by preliminary TR-FSMs (cf. Fig. 4.4 on p. 97). The verification using simulation as well as logical equivalence checking of these automatically generated designs did not reveal any discrepancies. The process was assisted by the design flow, which automatically generated setup scripts, configuration data, and mappings of FSM state registers. For all simulations, the same testbench as implemented for the manual HDL design could be used unmodified.

[4] Note that actually the division is not necessary because the measurements themselves are provided in an arbitrary unit, which has to be transformed to an actual temperature reading. This calculation can include the division at no extra cost.

[5] To simplify any example or new application and therefore to reduce the power consumption, the above use of raw sensor values can be extended further. Instead of scaling the value to a real value, the threshold value can be scaled to the arbitrary unit of the raw sensor values. Additionally, suppose an application performing a control loop, e.g., to precisely control the temperature, all controller parameters can be scaled to the arbitrary unit of the raw sensor values.

For each example application a firmware driver was developed. The MSP430-GCC[6] tool chain was used for the OpenMSP430 CPU with the `-mmcu=msp430f1232` option. For the early HW/SW co-simulation two new testbenches were created which instantiate the chip core module and the chip top module, respectively. The stimulus process and verification of the output signals were copied from the original testbench and were only slightly adapted. With the early HW/SW co-simulation each example application could be developed and integrated in its entirety. It was not necessary to interrupt the development of an example application and continue when the final reconfigurable module was generated.

Application Analysis

During the "Application Analysis" step (cf. Sec. 4.6), the cell library was developed (cf. Sec. 4.5) and the example applications were extracted. The final resource utilization report in Lst. 5.1 lists all cells as columns and the example applications as rows.

Listing 5.1: Resource utilization of the example applications shown as the output of FlowProc. The hyphenations of the column titles were introduced manually to fit the table to the page.

App	$fsm	Counter	Counter32	AbsDiff	Word-Register	Byte-Register	Add-SubCmp	ByteMux-Quad	ByteMux-Dual	Byte2-Word	Byte2-WordSel	WordMux-Dual
ADT7310	2	1	1	1	1	2	1	1		1		
MAX6682	2		1	1	1	2	1				1	
MAX6682Mean	2	1	1	1	2	2	2				1	1
ADT7410	2	2		1	1	2	1	1	2	1		
SlowADT7410	2		2	1	1	2	1	1	2	1		
ExtADC	1	1		1	1		1					
Min	1	1	1	1	1	2	1	1	2	1	1	1
Max	2	2	2	1	2	2	2	1	2	1	1	1

The development of the individual cells of the cell library is supported by the design flow similarly as the development of example applications. Verilog HDL and testbench templates are generated according to the specification of the ports in `setup.tcl`. Additional to synthesis and verification of the ports, the logical equivalence of topological variants and of reduced variants is automatically verified. For example, the `AddSubCmp` cell demonstrated in Fig. 3.8 on p. 71 is instantiated in most example applications via the reduced variant of a ">" comparator.

During the refinement cycles of the application and cell library optimization, several iterations with manual improvements were performed. One repeatedly occurring problem came from the use of immediate constant values in the Verilog designs. For example, the counters used to implement delays are initialized with the value 0 at reset using the Verilog statement "`Counter <= 0;`". In contrast, in the `Counter` cell, the statement "`Counter <= 16'd0;`" is used. Yosys correctly interprets the first statement according to the Verilog standard as a 32-bit integer of which only the lower 16 bits are used. On the other hand, it interprets the second statement as a 16-bit integer. Therefore these two values did not match in the extraction process. In the "Improvement" step, the example application HDL design was modified to match the cell design. A similar difficulty appeared during the development of another example application for the decrement of the counter, using the different assignments "`Counter <= Counter - 1;`" and "`Counter <= Counter - 1'b1;`".

[6] http://www.ti.com/tool/msp430-gcc-opensource [2015-08-20]

Another more complex case was solved for the example application "MAX6682Mean". Here an accumulator is implemented, which either loads a new value at the beginning of a measurement series, adds the new value to the old value, or keeps the old value. This is implemented with two dedicated signals "StoreValue" and "AddValue" generated by the FSM. Yosys creates a MUX topology where one MUX selects between the old value coming from the register output and the added value, and the second MUX decides between this result and the new value. The MUX structure does not directly fit the "WordRegister" cell. Its "Enable" input is used as select signal for an internal MUX which selects between the old value at the output and the new value at the input.

Therefore, a reduced variant was implemented, which actually is an "extended variant". It swaps these two MUXes and generates the appropriate control signals. The introduced combinational logic is integrated into the FSMs as discussed in Sec. 3.2.4 using the Yosys fsm_expand pass and the separate MUX is extracted as a "WordMuxDual" cell.

During the refinement cycles, the schematic of the extracted design of the example application can be visualized for the "Inspection" step. In Fig. 5.1 an exemplary schematic of an extracted application is shown. Unfortunately the schematics of the example applications do not fit the page or would require unreadable small fonts, therefore the post-silicon application "ExtADCSimple" is shown. This is a simplified version of the example application "ExtADC" which does not perform post-processing but notifies the CPU after every measurement (cf. Sec. 5.6.3). To visualize schematics, Yosys uses the Graphviz[7] dot tool, which is a general graph visualization software. Therefore the signals are drawn as curved lines instead of orthogonal lines. The program supports interactive zoom and pan functions to display and inspect all regions of the schematic.

Additionally, after the extraction, the resource usage and the remaining cells, which were not extracted, are listed. This can also be listed for the set of all example applications as previously shown in Lst. 5.1. The design flow can only be continued, if all example applications are successfully extracted and represented with only cells of the cell library and generic $fsm cells.

Merge

To prepare the "Merge" step (cf. Sec. 4.7), two TR-FSM instances (cf. Sec. 3.7) were specified with the implementation as shown in Tab. 5.1. The values were derived from the requirements of the example applications and increased to improve the post-silicon flexibility. Afterwards, two parallel trees for each connection type and oversizing rules were specified. An excerpt of these rules was used as an example in Lst. 4.5 on p. 106.

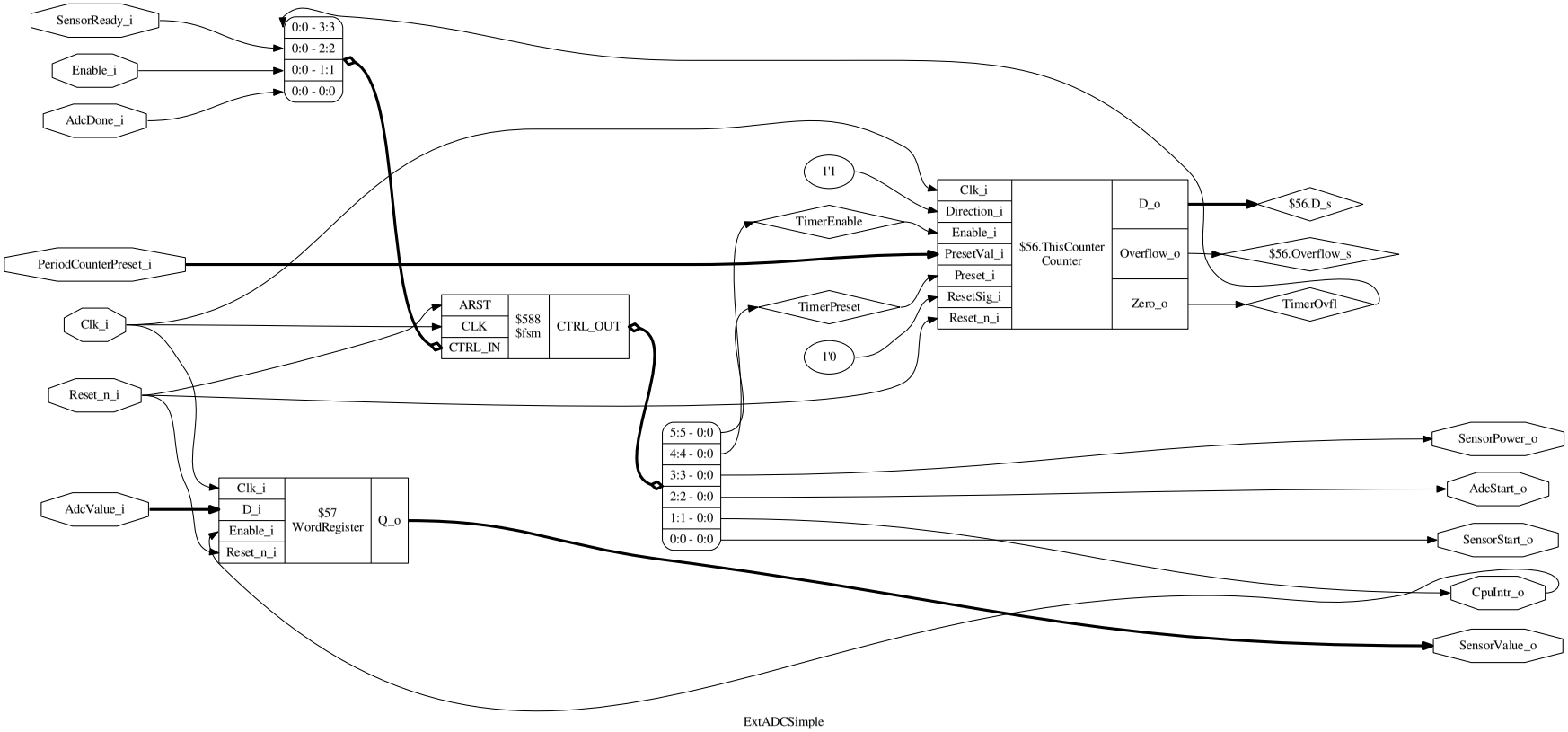
Table 5.1: TR-FSM specification for the WSN SoC. The columns n_i specify the number of transition rows with a width of i, and $\sum n_i$ is the total number of transition rows and therefore the maximum number of transitions which can be implemented with the respective TR-FSM instance.

	Inputs	Outputs	State Width	n_0	n_1	n_2	n_3	n_4	$\sum n_i$
TRFSM0	6	10	5	10	10	4	4	2	30
TRFSM1	10	15	6	10	20	6	6	4	46

With these specifications, the interconnect was generated and optimized with InterSynth. Listing 4.6 on p. 107 shows two sections of the Verilog HDL code of the resulting interconnect module.

[7] http://www.graphviz.org/ [2015-08-20]

Figure 5.1: Exemplary schematic of the extracted post-silicon application "`ExtADCSimple`". The "`WordRegister`" cell at the bottom left stores the sensor value coming from the "`AdcValue_i`" input and provides it at the "`SensorValue_o`" output, which is queried as a parameter by the CPU. The "`CpuIntr_o`" output notifies the CPU of a new value. Additionally, the same signal is used as the "`Enable_i`" input of the "`WordRegister`". The main "`$fsm`" generic FSM cell controls the execution of the application, including the "`Counter`" used to periodically initiate the measurement. The boxes with rounded corners connected to the "`CTRL_IN`" and "`CTRL_OUT`" ports of the "`$fsm`" cell denote the assembly and disassembly of vector signals. For example, the first row of the top left box labeled "`0:0 - 3:3`" connects the bit range `0:0` of the left signal (which is a single-bit signal) to the bit range `3:3` of the right signal. These assembly and disassembly boxes are only used for visualization and do not exist in the actual netlist.



Additionally, InterSynth generates LaTeX Ti*k*Z[8] pictures of all interconnect trees. In Fig. 5.2 the two trees of the connection type "`Byte`"[9] are shown together with the signals routed for the "`ADT7410`" example application.

Further, for each example application, a wrapper module for the interconnect module with the exact ports of the example application and setup scripts for simulation using the original testbench and for logical equivalence checking were generated. The verification has to be performed individually for every example application. This should be automated in a future version of the design flow. Verification showed correct results in all design iterations for all example applications.

Completion

In the Completion step (cf. Sec. 4.8) the final reconfigurable module and all further deliverables for the IP core were generated. The reconfigurable module contains the interconnect module and the infrastructure for configuration and parameterization (see Fig. 4.7 on p. 111). As mentioned in Sec. 4.8.1, the combinational loops in the interconnect module have to be handled with constraints for the static timing analysis (STA) during synthesis. Therefore a Tcl script was generated with InterSynth which describes the interconnect. However, this approach did not provide the expected results. Therefore a different approach was implemented (see Sec. 5.1.4).

Besides this, no other meta-information for the integration in the SoC was generated. Instead, the information was manually collected and setup.

The firmware and the configuration data for each example application were generated. The meta-information to develop new applications in the post-silicon design phase was stored using the appropriate InterSynth commands in the "Merge" step above.

Additionally, wrapper modules for each example application and setup scripts were generated to use the reconfigurable module instead of the original example applications for simulation and for logical equivalence checking. To access internal signals of the reconfigurable module (especially the parameters), special constructs are required (cf. Sec. 4.8.4). These allow to reuse the original testbench for simulation and to directly compare the example application HDL design with the completed reconfigurable module. Verification showed correct results for all example applications.

Besides the pure hardware-level verification, HW/SW co-simulation was used to verify the interactions between the firmware and the example application implemented with the reconfigurable module. In the completion step, the final driver with the complete set of configuration and parameterization data was generated. For the firmware development, two separate Makefile targets were used to automatically compile and link the two slightly different binaries: one for the early HW/SW co-simulation using the preliminary driver, and one for the final firmware using the final driver. This was seamlessly used with the firmware and the testbench manually developed together with the example application and showed correct results.

During the development of the reconfigurable module Mentor Graphics Questa Sim 10.0a was used for simulation and Cadence Encounter Conformal Equivalence Checking (LEC) 08.10-s440 was used for logical equivalence checking.

[8] `http://pgf.sourceforge.net/` [2015-08-20]
[9] The trees of the other connection types are much larger and therefore do not fit on a page.

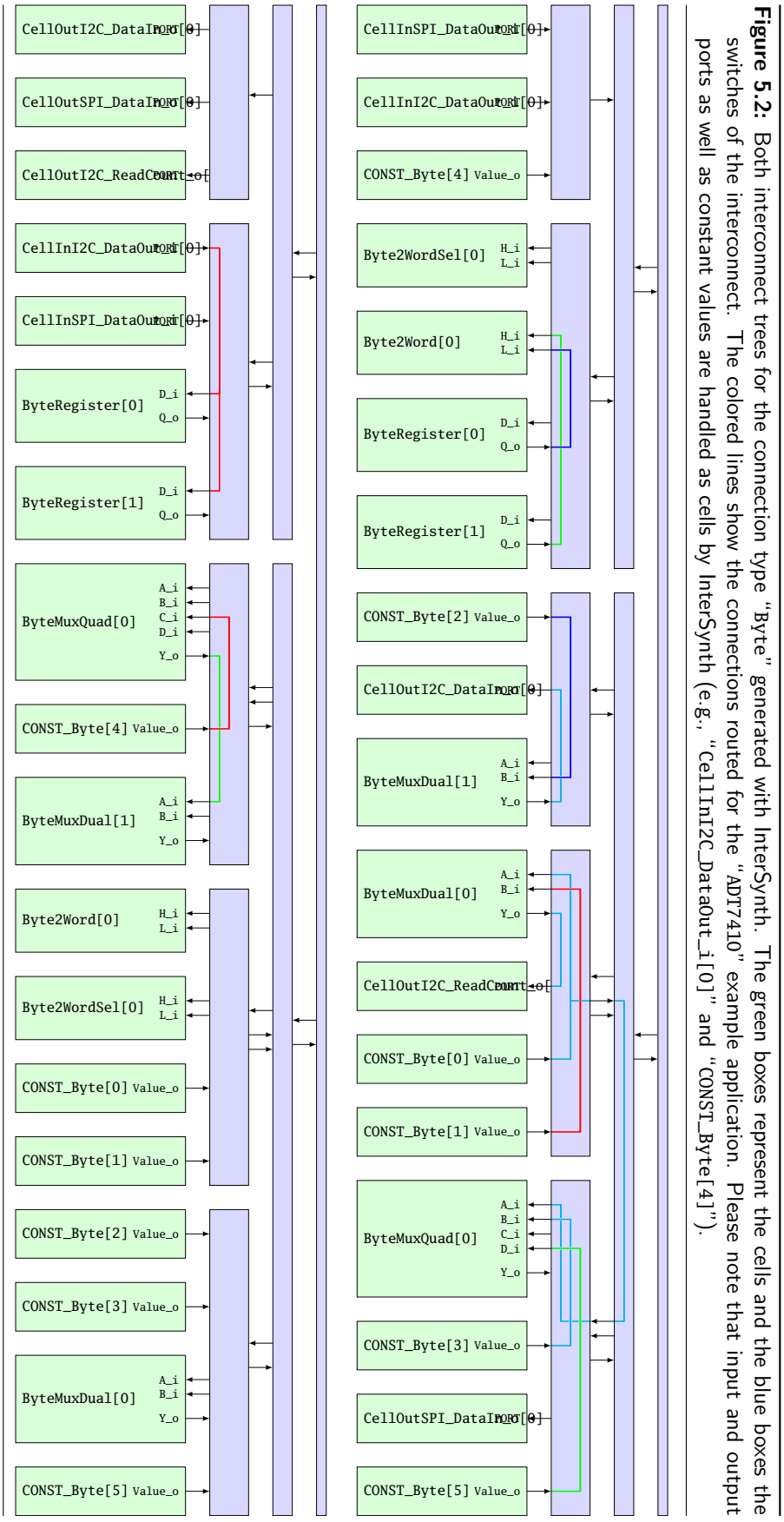
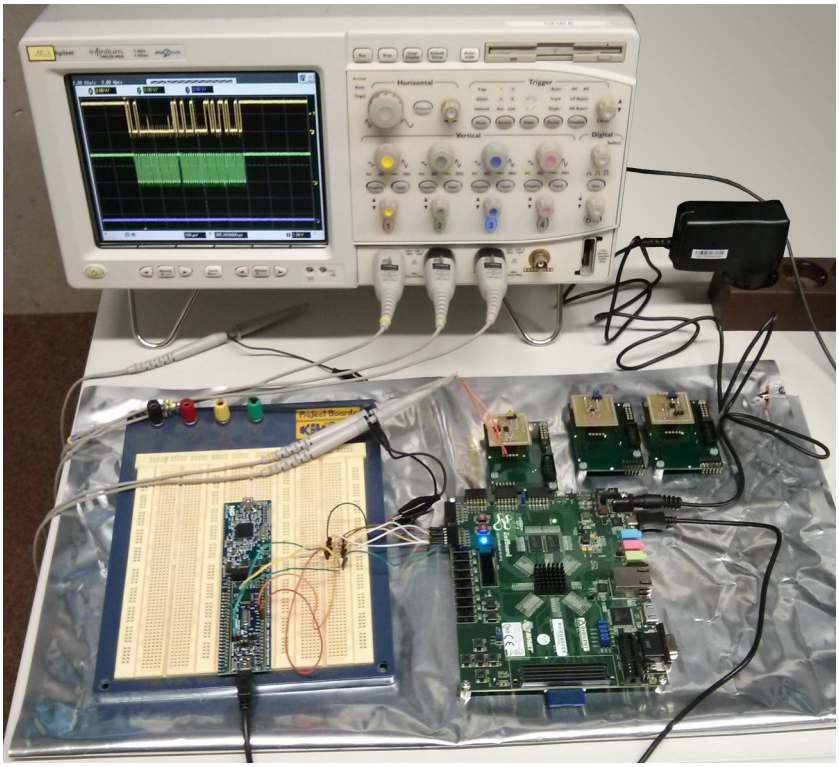


Figure 5.2: Both interconnect trees for the connection type "Byte" generated with InterSynth. The green boxes represent the cells and the blue boxes the switches of the interconnect. The colored lines show the connections routed for the "ADT7410" example application. Please note that input and output ports as well as constant values are handled as cells by InterSynth (e.g., "CellInI2C_DataOut_i[0]" and "CONST_Byte[4]").

Verification with Prototype Testing

Before the WSN SoC HDL design was entered in the ASIC design flow, it was verified using real hardware. The main purpose was to test the example applications with actual external sensors. For the simulation of the example applications, manually developed VHDL models of the external sensors were used. While these models were carefully developed in accordance to the datasheets, no independent verification was carried out. If these models were incorrect, the simulation of the example applications controlling these models would lead to the false assumption of correct example applications. To eliminate this potential fallacy, the WSN SoC HDL design was implemented using an FPGA and connected to actual sensor chips (Fig. 5.3).

Figure 5.3: Verification of the WSN SoC HDL design with prototype testing (explanation in the text).



For these tests an additional wrapper module for the chip core module was generated during the setup of the reconfigurable module to correctly implement bidirectional and open-drain pins (cf. Sec. 4.3.3). This wrapper module was synthesized together with the complete WSN SoC HDL design and the generated reconfigurable module and interconnect module using Xilinx Vivado v2014.2. The program and data memories were modeled using Verilog arrays which were automatically implemented as Block RAMs by the Vivado synthesis software. Note that the design in the FPGA is identical to the produced WSN SoC except the pad cells and the memory macros. The hardware tests were carried out using the ZedBoard[10] (bottom right in Fig. 5.3) which contains a Xilinx Zynq XC7Z7020-CLG484 FPGA (cf. Sec. 2.3.2). The CPUs in this chip were not used for the tests. Note that this setup has two levels of configuration: Firstly the FPGA as underlying hardware platform, and secondly, on top, the reconfigurable module. The FPGA con-

[10] http://zedboard.org/ [2015-08-20]

figuration was not changed during the tests. Only the configuration of the reconfigurable module was replaced for each example application.

To access the debug interface of the OpenMSP430 CPU and the UART from a PC, an adapter was developed. It is based on the LPCXpresso Board for the NXP LPC11U14 microcontroller[11] which is located on the bread board together with I^2C pull-up resistors in Fig. 5.3. The debug adapter firmware implements two USB communications device class (CDC) interfaces, one for the UART and one for the I^2C debug interface.

To test the example applications, first the appropriate sensor was connected to the ZedBoard. These were soldered on sensor module PCBs (small square FR4 PCBs above the ZedBoard) and inserted into Pmod adapter sockets (dark green PCBs).[12] After enabling the power supply of the ZedBoard, the FPGA configuration bitstream was applied using the on-board USB interface. Then the adapter with the LPC11U14 MCU was used to access the debug interface of the synthesized OpenMSP430 CPU to download the WSN SoC firmware into the program memory and to start the execution.

The tests showed proper behavior of all example applications, except `ExtADC`, which was not tested because no emulation model of the ADC was implemented. One problem with the MAX6682 sensor appeared, because it immediately stops any ongoing conversion if the value is queried via SPI. This was resolved with a longer delay between the queries.

The FPGA test was conducted after the completion of the reconfigurable module. However, the design methodology and the design flow also allow to perform prototype tests already during the development of the example applications, similarly to the early HW/SW co-simulation. The setup only differs from the above setup by the use of the example application HDL design together with its wrapper module and the preliminary firmware driver.

5.1.4 Characterization of the WSN SoC

An industry standard ASIC design flow was used to implement the verified WSN SoC design as a test chip. The first step was *logic synthesis* using Synopsys Design Compiler C-2009.06-SP5-2 to generate a gate-level netlist. The chip top-level module with an instance of the chip core module and the appropriate pad cells for all pins was generated using FlowProc during the generation of the reconfigurable module (see Lst. 4.2 on p. 95).

As mentioned in Sec. 5.1.2, no flash memory was available, therefore the program memory as well as the data memory were implemented with SRAM cells. The 8 kB program memory was implemented with four 2k×8 area optimized memory cells, and the 256 bytes data memory was implemented with two 128×8 speed optimized SRAM cells. Both cells were available from a previous project. The 2 kB SRAM cells required customized interface logic for clock and data synchronization.

The clock frequency was set to 10 MHz, which is limited by the timing requirements of the size optimized 2 kB SRAM cells and the according interface logic. To break the combinational loops of the interconnect, all MUXes were grouped into a new sub-module named "`Muxes`" and all timing arcs through that module were disabled. Additionally, maximum and minimum delay and input and output delay constraints were applied to the signals.

[11] `http://www.nxp.com/demoboard/OM13014.html` [2015-08-20]
[12] The Pmod adapter sockets were created by Martin Schmölzer [Sch14].

The OpenMSP430 includes manual clock gating cells to implement the low-power modes (cf. Sec. 5.1.2). The configuration registers, which were implemented as shift registers, contain a large number of D-FFs. Therefore in the configuration interface also clock gating cells were instantiated manually. To improve the synthesis results, these high fan-out nets were set up as ideal networks. Additionally, automatic insertion of clock gating was used to reduce the total power consumption.

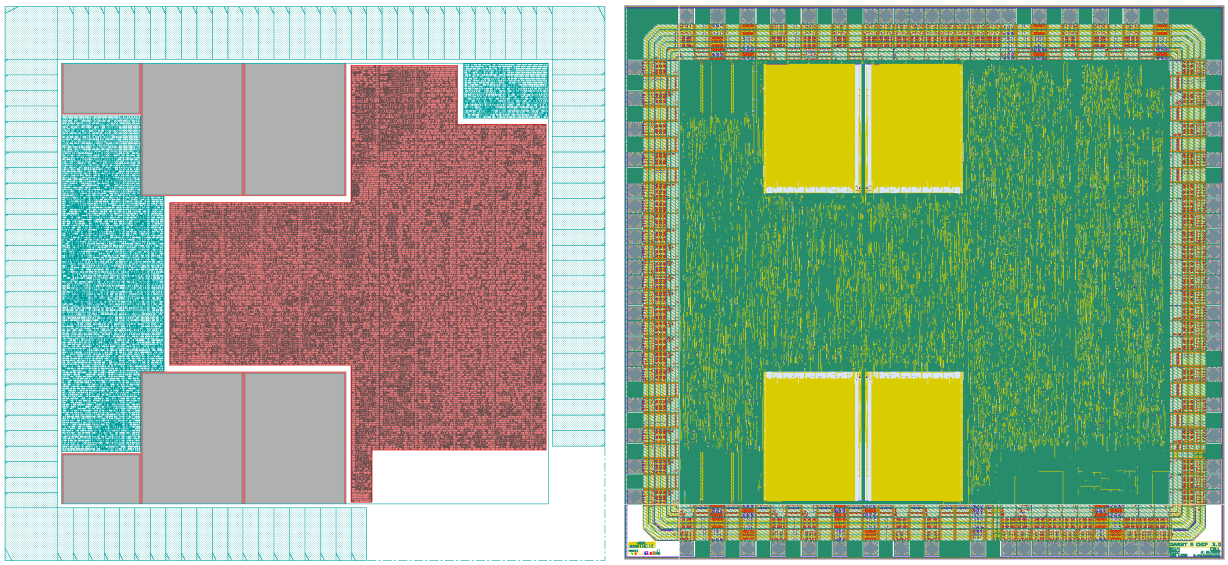
The reconfigurable module and the interconnect module generated with the design flow were fully synthesizable without problems.

After the synthesis, the resulting gate-level netlist was compared for logical equivalence with the original HDL design using Synopsys Formality C-2009.06-SP3.

The *layout (place and route, P&R)* was generated with Cadence Encounter v08.10-s273_1. The WSN SoC floorplan is shown in the left image in Fig. 5.4. The free area in the bottom right corner is reserved for a different project. To enable precise measurements of the power consumption of the reconfigurable module, a separate power domain was introduced for it. The power was distributed using rings around the core area, the memory cells, and around the reconfigurable module as well as vertical stripes across the core area.

Then all standard cells were placed and the clock tree was synthesized. Additional buffer trees were inserted for the reset signal and for the control signals to the configuration registers. Afterwards all signals were automatically routed, filler cells were inserted to fill the spaces between logic gates, and the whole design was optimized and verified.

Figure 5.4: Floorplan (left) and final layout (right) of the WSN SoC. The layout contains fill structures on all metal layers. The bottom right corner of the floorplan is reserved for a different project and inserted in the layout. The pad cells are placed at the perimeter of the WSN SoC. Its core contains the digital logic. The gray boxes in the floorplan are SRAM cells (two instances of 128×8 bits and four instances of 2k×8 bits). The four 2 kB SRAMs are visible as yellow boxes in the layout because no fill structures on metal layer 4 are allowed while the two 128 byte SRAMs are covered with fill structures. The cyan polygon on the left of the floorplan contains the digital logic of CPU and its peripherals. The red/gray polygon contains the reconfigurable module. The cyan rectangle on the top right of the floorplan contains the peripherals of the reconfigurable module.



The final design was checked for logical equivalence to the original HDL design as well as to the gate-level netlist from synthesis using Synopsys Formality. Synopsys PrimeTime C-2009.06-SP3 was used for static timing analysis (STA) to generate standard delay format (SDF) files with signal delay information from parasitics information for the following post-P&R simulation with accurate delays.

The simulation was carried out using Mentor Questa Sim 10.0a on an Intel Core i7-860 CPU with 2.80 GHz, 64 bit architecture, 8 GB memory, and a Debian GNU/Linux operating system. The Questa Sim kernel only used a single core. The simulation performance in number of simulated clock cycles per second wall-clock time strongly depends on the simulation setup. For the WSN SoC RTL design without the reconfigurable module and its SPI and I^2C peripherals this achieved approximately 14.0 kCycles/s for the simulation of a blinking LED firmware program using a delay loop.

The early HW/SW co-simulation of the "`ADT7310`" example application using its RTL code instead of the reconfigurable module achieved 276.6 kCycles/s. The CPU was in the inactive LPM3 state. Final HW/SW co-simulation using the generated RTL design of the reconfigurable module with everything else being equal achieved 180.6 kCycles/s.

HW/SW co-simulation of the post-P&R netlist with accurate delays achieved approximately 3 kCycles/s for the pure firmware implementation of the sensor interface task with delays implemented in LPM3, i.e., the WSN SoC is mostly inactive. This setup was also used to generate the VCD and SAIF files for the power analysis (cf. Sec. 5.3.3). When this logging was active, the simulation performance dropped to approximately 1 kCycles/s. The simulation of a 200 ms interval (cf. Sec. 5.3.2) at 10 MHz operating frequency produced a 30.6 GB VCD file, which was compressed to 7.6 GB. Shorter intervals and lower frequencies produced smaller VCD files.

For the implementation of the delays with permanently polling the timer overflow flag, the performance further dropped to approximately 200 cycles/s. A 200 ms interval at 1 MHz produced a 13.1 GB VCD file (compressed 4.8 GB). The simulation of the "`ADT7310`" example application performed by the reconfigurable module achieved 0.9 kCycles/s, which is mostly a 32-bit counter to implement the period delay (cf. Sec. 5.3.1). A 200 ms interval at 10 MHz produced a 34.0 GB VCD file (compressed 9.2 GB). All VCD files were converted to SAIF files with approximately 30 MB (compressed 0.8 MB).

The layout and the netlist were imported to Cadence Virtuoso icfb 5.10.41.500.6.143. This supplemented the design with the detailed layout of the standard cells and with reduced layouts of the memory cells.[13] The WSN SoC design was manually amended with the layout of the second project and with logos at the bottom corners. Additionally, manufacturing process specific fill structures for empty areas in the metal layers were generated with Mentor Calibre v2011.2_27.20. The same program was used for sign-off verification (ERC: electrical rule check, DRC: design rule check, LVS: layout vs. schematic). The final layout (see right image in Fig. 5.4) was sent to manufacturing (tape-out).

For the design flow, the WSN SoC design including the reconfigurable module, and all scripts to automate the mentioned tools, the following amount of source code was developed (lines of code):

- Tools, Scripts: Pascal: 45.700, Tcl: 18.900, Bash: 13.200, C: 2.500, Makefile: 900, C++: 500

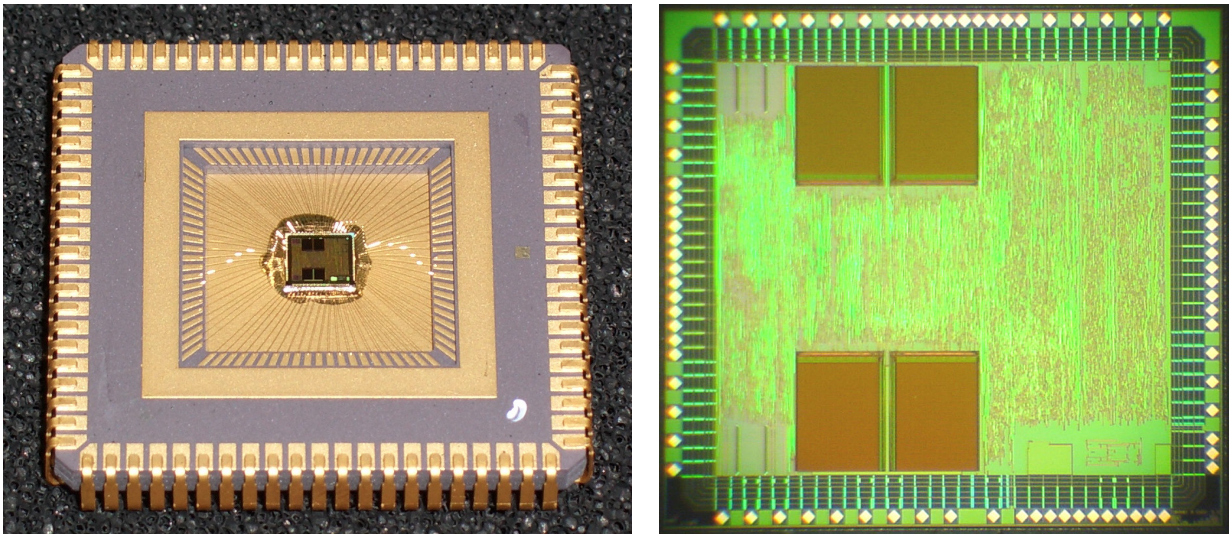
[13] The foundry provides reduced layouts of the SRAM cells which include all metal layers but no information on the doping. Before production, the reduced layouts are replaced with the full layouts by the foundry.

- Design: VHDL: 33.600 (5.700 third party), Verilog: 18.700 (12.200 third party)
- Generated: VHDL: 21.900, Verilog: 135.300 (97.000 lines is the interconnect module), Tcl: 800, C: 2.500

The WSN SoC was produced end of 2014 at ams AG[14] (see Fig. 5.5). The above described ASIC design flow used the HitKit 3.80 process design kit. The chip was manufactured in the C35B4C3 350 nm silicon CMOS process with 4 metal layers (Al). The process supports 3.3 V core and pad supply. The standard cell library implements cells with a regular threshold voltage and a row height of 13 µm. It uses pad cells with a size of 340.4 µm × 100 µm.

The WSN SoC size is 3,910 µm × 3,610 µm = 14.1 mm^2. The chip core area inside the pad cells is 3,200 µm × 2,900 µm = 9.28 mm^2. More details on the chip area are discussed in Sec. 5.4. The chip was placed in a CLCC "J" package with 84 pins.

Figure 5.5: WSN SoC die in the package with bond wires (metal lid removed, left image) and die photo (right).



5.1.5 Characterization of the Reconfigurable Module

The reconfigurable module instantiates cells from the cell library as given by the resource utilization report in Lst. 5.1 on p. 122 plus additional cells as specified by the oversizing rules in Lst. 4.5 on p. 106, as well as two TR-FSM instances as shown in Tab. 5.1. It holds four configuration registers with a total of 3,889 bits. More details on the configuration are discussed in Sec. 5.5.

The reconfigurable module further contains seven parameterization registers written by the CPU, of which two are used for reconfigurable signals with the usage type "`param`" and the other five are connected to the interconnect module. It also contains three parameters read by the CPU, of which one is used for a reconfigurable signal. All parameters are implemented as 16-bit signals and used with the connection type "`Word`". The report on the characteristics of the interconnect generated with InterSynth is shown in Lst. 5.2.

[14] http://asic.ams.com/ [2015-08-20]

Listing 5.2: Characterization of the interconnect generated with InterSynth. Note that the high number of celltypes and cells results from the fact that each input and output port (including parameters) is internally represented as an individual celltype by InterSynth.

```
Number of conntypes:    3
Number of trees:        6
Number of celltypes:   80
Number of cells:       98
Number of netlists:     6
Number of unroutable netlists: 0
Number of nodes per netlist:      26 ..   44
Number of nets per netlist:       29 ..   64
Number of up-links per switch:     0 ..   10
Number of down-links per switch:   0 ..    6
Number of up-ports per switch:     0 ..   25
Number of down-ports per switch:   0 ..   18
Interconnect config bits per cell port: 1208 / 220 =   5.49
Interconnect mux2 per cell port:        2670 / 220 =  12.14
```

The leakage current of the reconfigurable module was measured as 15.4 nA and the active current without configuration is 59.4 µA/MHz. More details on the power consumption are discussed in Sec. 5.3 and appendix A. The chip area of the reconfigurable module is 4.2 mm^2 (see red/gray polygon in the floorplan in Fig. 5.4). For more details see Sec. 5.4.

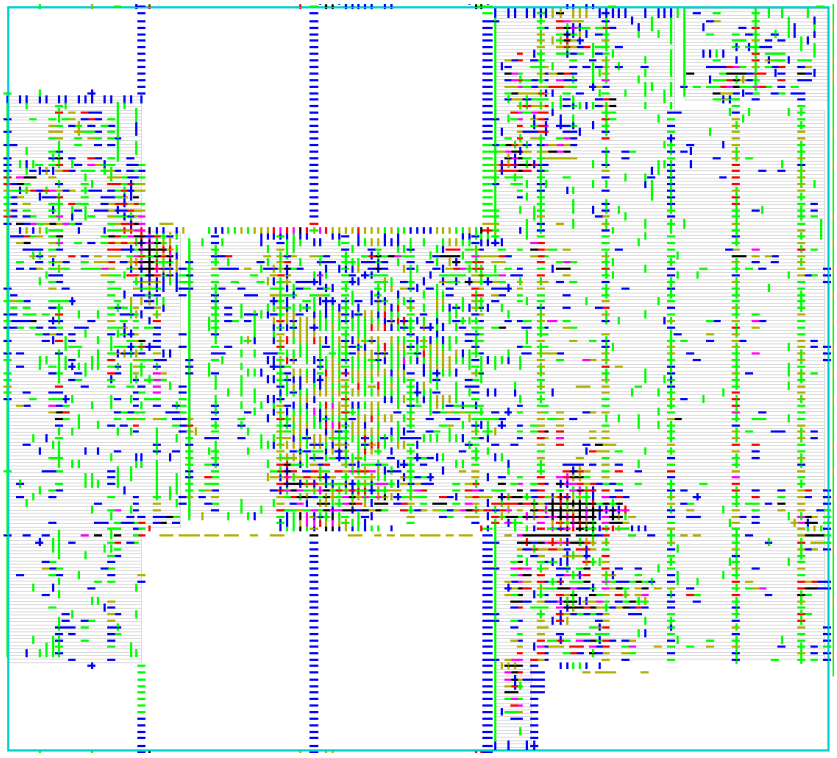
The routing of the layout revealed considerable congestion, i.e., more area for the routing tracks was required (see Fig. 5.6). Especially the region of the interconnect was strongly congested (cf. Fig. 5.18 on p. 154). This is a special problem for the interconnect of connection types with wide signals, because only a low number of large configuration D-FF cells is connected to a high number of small combinational cells and routing tracks. This problem could not be relaxed by the reduction of the area utilization because the placement algorithm still arranged the combinational cells in close vicinity while leaving only the corners of the reconfigurable module with unused area. During the optimization of the design, the tool successfully routed all signals, but had to use longer wires.

Two possibilities to relax this problem are suggested. Firstly, partial placement blockages which specify the placement *density* at specific regions can be utilized. However, this either requires a-priori knowledge where the interconnect and its individual trees will be placed, or the interconnect must be constrained to a specified region. The second possibility to relax the congestion problem is to utilize a method similar to scan chain reordering for the configuration shift registers. This would ignore the serial connections between all configuration D-FFs and therefore only use the connections to the combinational logic of the interconnect to guide the placement. Unfortunately, scan chain reordering is a very specific feature tailored to the scan logic ports of the according scan D-FFs. More work is required to investigate this approach.

5.1.6 Evaluation of the Hypothesis

In this section, hypothesis 1 as defined in Sec. 1.2 is evaluated. Therefore the design methodology introduced in this thesis and its implementation as a design flow were used to implement a reconfigurable module. This was integrated in a WSN SoC and manufactured as a test chip.

Figure 5.6: Congestion map of the WSN SoC routing: Each line fragment represents a routing "Gcell" for horizontal or vertical tracks. The colors denote the filling degree: blue: only one track is unused, green: all tracks are used, yellow: one more track is required than available, red: two more tracks, magenta: three more tracks, black: four or more tracks.



The manufactured WSN SoC was successfully put into operation. It was tested with the firmware of the example applications, which also include the configuration data, and correctly executed the defined tasks. Additionally, new applications, which are discussed in Sec. 5.6.3 and more detailed in appendix A, were developed. These also correctly executed the defined tasks.

This test shows that the design methodology leads to a working reconfigurable CPU supplement module which provides the full specified functionality plus additional flexibility to implement new functionality. Therefore hypothesis 1 is provisionally accepted.

5.2 Manually Developed Reconfigurable Module

At an early stage of the development of the discussed design methodology, a reconfigurable module was manually developed and manufactured as a test chip (cf. "Early Approach" in Sec. 3.1.5) to examine the potential for the reduction of the power consumption compared to an MCU [GDHG11, GGHG11]. This reconfigurable module is used in the evaluation of hypothesis 2 in the next section and therefore briefly introduced here. Contrary to the previously discussed WSN SoC, the early test chip only contains the reconfigurable module and its peripherals but without a CPU. It was implemented for the same application class as the WSN SoC.

The structure of the manually developed reconfigurable module is shown in Fig. 5.7. It is built of three reconfigurable sections to separate control, data storage, and arithmetic, each with a manually developed internal interconnect and with connections among them. The "Control" section includes four TR-FSM instances. In the "Byte" section 16 8-bit registers with a flexible and reconfigurable address decoder from control signals are implemented. The "Word" section uses similar cells as used in the WSN SoC (cf. Sec. 5.1.3). These are connected with a comprehensive interconnect, including feedback paths and dedicated conversions from and to the "Byte" section.

Additionally, six bus master peripherals for serial protocols are included. Similarly to the WSN SoC, an interface to an external ADC as well as digital IOs are provided. The configuration registers are accessed via a JTAG interface. This implements boundary scan for the pins of the test chip as well as special JTAG data registers which actually are the configuration registers. The reconfigurable module holds eleven configuration registers for the TR-FSMs, the different interconnects, and the configuration of the input and output pins. For the parameterization interface an UART interface is included.[15]

The reconfigurable module was developed manually in VHDL. It was verified with simulation using Mentor Questa Sim 6.5d. The reconfigurable module was implemented as a test chip in the Infineon Inway 5.2.2m4 design environment and produced using the automotive grade Infineon C11N 130 nm CMOS process with six copper and one aluminum metal layers. The JTAG interface was implemented with Mentor Graphics BSDArchitect v8.2006_3.10. The synthesis was performed with Synopsys Design Compiler Version X-2005.09-SP4 to standard cells with a row height of 4 µm and high threshold voltage for reduced leakage power. The layout was generated using Magma Design Automation Inc. Talus version 1.0.84-linux24_x86_64.

The resulting test chip was produced in 2010 and is shown in Fig. 5.8. Its total area is 1,646.44 µm $\times$1,486.44 µm = 2.45 mm^2. The core area inside of the pad frame is 1,120 µm$\times$960 µm = 1.08 mm^2. The chip has 52 pins and has a core supply voltage of 0.8–1.5 V and two separate pad supplies with 2.5–3.3 V.

Besides the manual development of the reconfigurable module, the configuration data is also generated mostly manually. A set of Pascal classes were developed which represent the design. To develop a design, a small program is written which instantiates these classes and executes the according methods. For the TR-FSMs, the states and the transitions with input and output patterns are specified. The signal connections between the TR-FSMs, the "`Byte`" registers, and between the individual "`Word`" arithmetic cells are set up with another set of methods.

Finally, a set of "synthesis" methods is executed which map the design to reconfigurable module resources and generate configuration data. These designs can only be verified at the end of their development with simulation by downloading the generated configuration data to the VHDL code of the reconfigurable module with JTAG signals.

To operate the test chip, an evaluation platform was developed which will be discussed in Sec. 5.3.2. The configuration data is downloaded into the test chip with JTAG and the parameters are set via UART. Then the operation of the design is activated. More details on the manually developed reconfigurable module were published in [GDHG11, GGHG11].

[15] The serial bus master peripherals and the UART module were developed by students supervised by the author of this thesis. The I^{2}C and SPI peripherals were also used in the WSN SoC.

Figure 5.7: Structure of the manually developed reconfigurable module (reproduced from [GDHG11, GGHG11] with permission, explanation in the text).

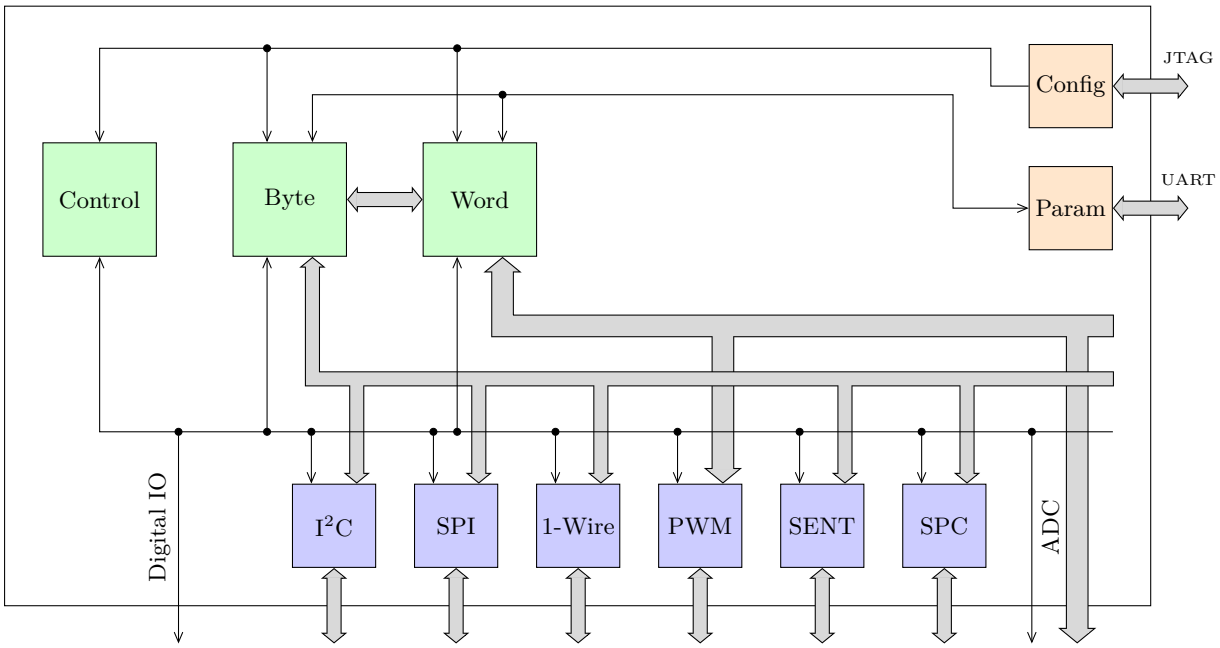
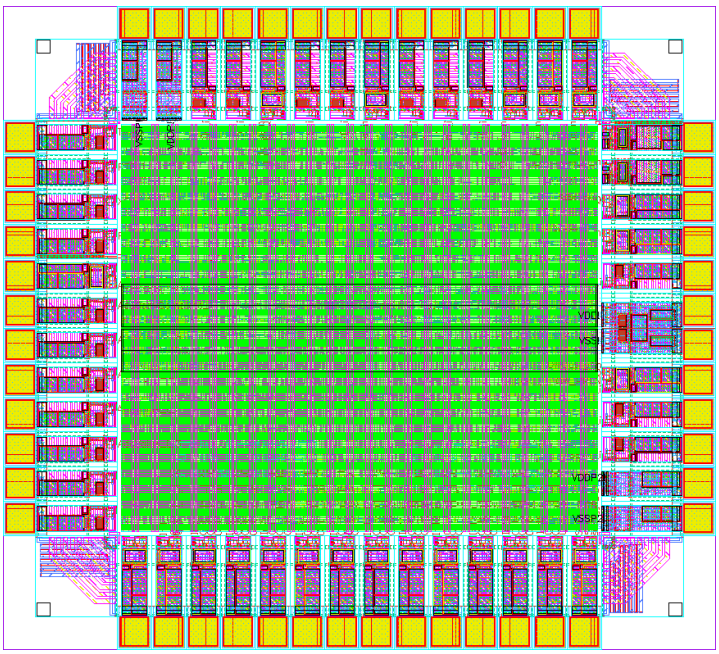
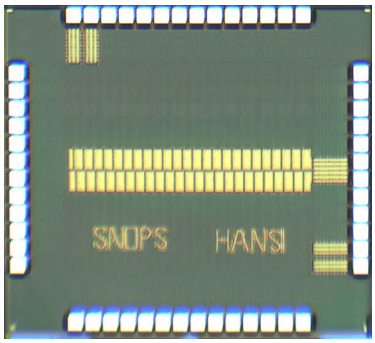
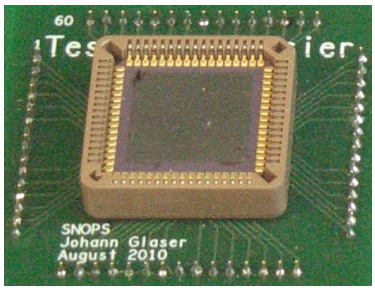


Figure 5.8: Layout (left), die photograph (top right), and chip package (bottom right) of the manually developed reconfigurable module. The yellow horizontal line in the die photograph are two wide wires to connect the supply pads on the right across the chip core. These are connected to vertical wires (magenta in the layout) which further distribute the power supply to the standard cell rows. No other features except the bond pads and logos are visible in the die photograph due to fill structures.







5.3 Power Consumption

In this section the power consumption of the WSN SoC and hypothesis 2 are evaluated. As first step, in Sec. 5.3.1 a representative task is selected which is performed by both, a firmware implementation and the reconfigurable module. Additionally a model of the energy consumption, a method for its measurement, and a calculation to extract the requested values are derived.

In Sec. 5.3.2 the hardware platform for the generation of the stimuli and the measurements as well as its automated operation are discussed. To identify the individual contributions to the total energy consumption, power analysis was performed for the WSN SoC. This method is presented in Sec. 5.3.3.

Finally, in Sec. 5.3.4 hypothesis 2 is evaluated for the WSN SoC as well as for two additional sources which evaluated the manually developed reconfigurable module. Additional information and details on the power consumption of the WSN SoC and of the reconfigurable module are presented in appendix A.

5.3.1 Definition of the Energy Consumption

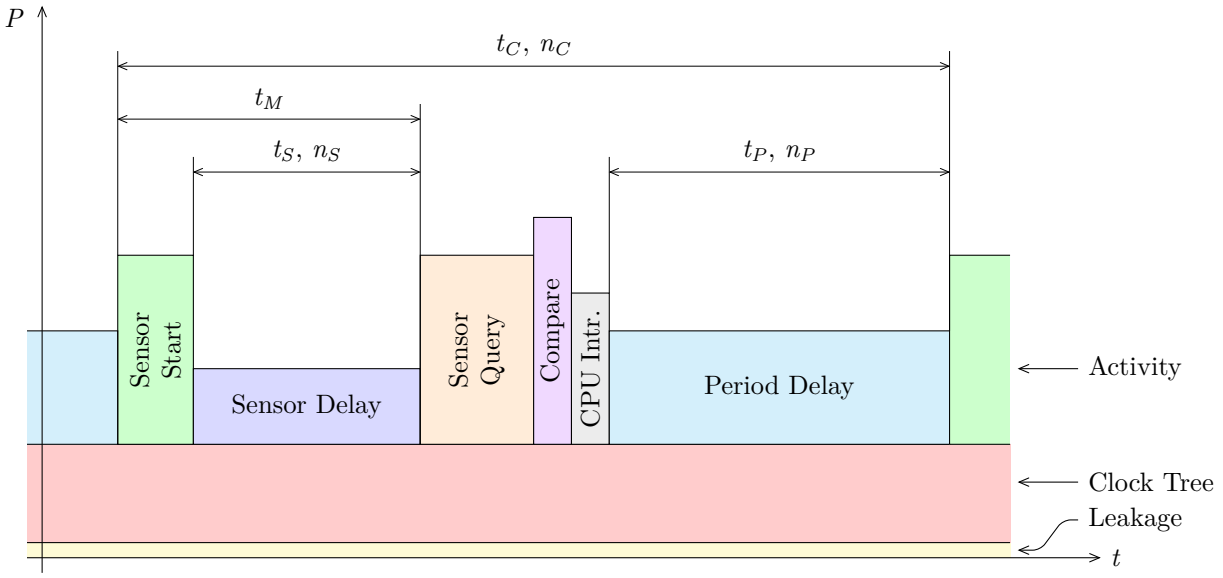
To evaluate hypothesis 2, an identical task is once performed by the reconfigurable module, and once by the CPU, both in the manufactured WSN SoC. That task has to be located within the application class of the reconfigurable module. Therefore, the task is defined as periodic measurements conducted with the Analog Devices ADT7310 16-bit temperature sensor with SPI interface [Ana09]. This sensor was used in an example application to develop the reconfigurable module (cf. Sec. 5.1.3). It is selected, because it is a moderately complex sensor which requires separate SPI transmissions to initiate a measurement and to query the resulting value. This ensures that the evaluation of the hypothesis is not corrupted by oversimplification. Additionally, the sensor itself has a low power consumption and is therefore also a typical use case in the application class.

Hypothesis 2 is provisionally accepted, if the energy consumption of the reconfigurable module is lower than the energy consumption of the CPU. The energy consumption while performing the above defined task is the sum of multiple contributions (see Fig. 5.9). First the sensor is started to initiate a measurement in one-shot mode. Then a delay of $t_S = 240\,\text{ms}$ is required until the measurement value is available. Afterwards, this value is queried, compared to the old value, and, if it differs from the old value, the CPU is notified with an interrupt request. These sensor measurements are performed periodically with a cycle time t_C.

From these individual contributors, only the actual activity to perform the measurement is relevant, because the leakage power and the power consumption of the clock tree only depend on the implementation, the synthesis and place and route settings, and the semiconductor process technology. Additionally, the delay periods t_S and t_P are identical for the CPU and the reconfigurable module cases. The according energy consumption is however implementation dependent. For example, an ultra-low-power RTC can be used, or the time can be utilized to perform other tasks. Therefore, the energy consumption used to evaluate the hypothesis is defined as the contributions of the sensor start, sensor query, comparison, and CPU interrupt operations.

The individual operations transition rapidly which results in steep changes of the current consumption. However, due to parasitic inductances and capacitances in the power supply wiring and the power supply circuitry, only a smoothed mean current can be measured. From this the

Figure 5.9: Measurement cycle and power consumption of the ADT7310 example application (explanation in the text).



total energy consumption of one measurement cycle is calculated as $E_C = U \cdot I \cdot t_C$. Additionally, in a separate testcase, the current consumption of the fully set up but deactivated CPU and reconfigurable module are measured and denoted as I_0. This only includes the leakage current and the current consumption of the clock tree. With I_0, the energy consumption of the actual activity is given by

$$E_A = U \cdot (I - I_0) \cdot t_C.$$

To further separate E_A into the individual contributions, *multiple linear regression* is used. A model for the total energy E_A as the linear combination of the individual contributions as regression coefficients b_i is set up with the basic formula

$$E_A = b_0 + \sum_{i=1}^{n} b_i x_i + \epsilon.$$

Then multiple testcases are measured with a variation of the parameters x_i. The individual contributions b_i are finally determined using least squares.

The total energy E_A is the sum of the energy used for the evaluation of the hypothesis and the energy consumed for the two delays. The energy consumption for the evaluation of the hypothesis is constant for a given implementation and represented by b_0. The energy consumptions of the delays are proportional to the number of the clock cycles $n_S = f \cdot t_S$ and $n_P = f \cdot t_P$ and entered in the equation with $b_1 \cdot n_S$ and $b_2 \cdot n_P$.

The mathematical evaluation showed that a term to include the cycle duration t_C improves the model fitting, although the leakage current and the current consumption of the clock tree I_0 were already subtracted to calculate E_A. The reason are variations of the leakage current due to uncontrolled temperature changes during the measurements. The constant power is entered in the equation with $b_3 \cdot t_C$.

In the WSN SoC the delays are implemented as counter registers which are preset to a start value and decrement until zero is reached. The energy consumption depends on the switching activity, which also depends on the current counter value. It is highest for the least significant bit, which toggles at every clock cycle, and decreases towards the most significant bit. A longer delay with a larger start value involves more switching activity at the higher bits. Therefore using additional quadratic terms $b_4 \cdot n_S^2$ and $b_5 \cdot n_P^2$ greatly improves the fitting of the model. A further improvement of the model fitting is achieved with the terms $b_4 \cdot n_S \cdot n_C$ and $b_5 \cdot n_S \cdot n_C$ instead of the exactly quadratic terms.

The final model for the energy is given in Eqn. 5.1.

$$E_A = b_0 + b_1 \cdot n_S + b_2 \cdot n_P + b_3 \cdot t_C + b_4 \cdot n_S \cdot n_C + b_5 \cdot n_P \cdot n_C \tag{5.1}$$

- b_0 denotes the energy consumption used for evaluation of the hypothesis.
- b_1 and b_2 are the energy consumption per clock cycle of the two delays.
- b_3 is the changed leakage power in comparison to the I_0 measurement.
- b_4 and b_5 are correction coefficients to improve the model fitting.

To use the linear regression to separate the individual contributions of the energy consumption, testcases with linear independent variations of n_S, n_P, and t_C are required. This is achieved with the measurement procedure developed by Georg Blemenschitz [Ble15], which is described in more detail in the next section. The interval t_M between the sensor start and the sensor query is fixed at 2 ms for each testcase. The testcases differ in the operating frequency f and the cycle time t_C. This results in a concurrent variation of n_S and n_P. Note that t_S is lower than the required 240 ms. This is however no problem, because the ADT7310 sensor is emulated with an FPGA to supply a precise series of measurement values. One additional testcase for each frequency f is required with the CPU or the reconfigurable module disabled to determine I_0.

The evaluation of the measurement results also showed that the energy consumption values vary in a wide range depending on the testcase by a factor of approximately 300. Therefore the larger energy consumption values have a disproportional influence to the smaller energy consumption values in the model fitting. To mitigate this problem, *weighted* least squares are used to determine the regression coefficients

$$\mathbf{b} = \left(\mathbf{X}^T\mathbf{W}\mathbf{y}\right)^{-1}\left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)$$

with $\mathbf{b}$ being the vector of the searched regression coefficients b_i, $\mathbf{X}$ the matrix of the x_i values (n_S, n_P, t_C, $n_S \cdot n_C$, and $n_P \cdot n_C$) for all testcases, $\mathbf{y}$ the vector of the measured energies E_A for all testcases, and $\mathbf{W}$ the weighting matrix [KNNL05]. $\mathbf{W}$ is a diagonal matrix with $W_{ii} = \frac{1}{y_i^2}$, i.e., the weights are inversely proportional to the square of the energy E_A.

To perform the measurements an evaluation platform and a measurement procedure were developed. These are discussed in the next section.

5.3.2 Measurement Setup

For the characterization of the manually developed reconfigurable module, an evaluation platform was developed (see Figs. 5.10 and 5.11) [GGHG11]. This is also used to characterize the WSN SoC. It provides a socket for a test chip carrier module and two sockets for MCU carrier modules.

These are connected to a Microsemi/Actel SmartFusion Evaluation Kit[16] [Act10b] via a crossbar switch for arbitrary connections.

The SmartFusion FPGA [Act10a] includes a hard-wired ARM Cortex-M3 MCU with a tight coupling to the FPGA fabric as well as analog and mixed signal functionality. The FPGA is used to create and observe timing accurate signals to stimulate and test the test chip or MCU, e.g., to simulate an external ADC or sensors via SPI or I^2C. The SmartFusion evaluation kit required modifications to access more digital signals as available through its mixed signal header. It is connected to the PC via USB to remotely control its operation.

For tests with real sensors the evaluation platform additionally provides connectors to extend the serial busses to separate sensor carrier modules which can hold the sensor modules shown in Fig. 5.3 on p. 127. It further provides adjustable power supplies for the module sockets with current measurement. However, these are not accurate, therefore external power supplies and current measurement were utilized. This was set up for two different supply voltages, each with a four-wire connection for separate supply and voltage sensing (see the red and black wires from the coaxial cables at the left side in Fig. 5.11). The evaluation platform also allows to simultaneously operate an MCU together with the manually developed reconfigurable module as a co-processor to simulate a complete WSN node.

The evaluation platform was improved and extended by Georg Blemenschitz in his diploma thesis [Ble15]. Originally a simple multiplexer for UART signals between the PC, the SmartFusion FPGA, and the test chip or the MCUs was implemented. This was replaced by a flexible any-to-any multiplexer using a CPLD (top right in Fig. 5.11 with colored wires). Furthermore, the FPGA design and its driver in the ARM Cortex-M3 firmware were greatly improved and extended with a clock generator, general purpose IOs, a generic SPI sensor simulator, and facilities to trigger and to synchronize the current measurements.

The major extension is the development of a complete measurement setup including external power supplies and voltage and current measurement. These are controlled and automated with comprehensive Matlab programs via GPIB, Ethernet, RS232, and USB. This is accompanied by a set of Matlab programs to analyze the measured values.

Georg Blemenschitz also developed a universal measurement procedure to characterize a test chip or an MCU [Ble15]. First a test series is set up with approximately 100 to 200 testcases, depending on the evaluated chip and its frequency and supply voltage operating range. Then, for each testcase, a preliminary supply voltage and operating frequency is set and the parameters (e.g., f, t_C) are communicated to the chip via UART. After the chip starts operation, the target voltage and frequency are set, including a control loop to accommodate the voltage drop of the wires and connectors. Then the mean current consumption is measured and stored to a datafile.

The activity of the chip is defined as the periodic execution of the sensor measurement task with a cycle time t_C. The current measurement is performed with an integration window of 200 ms (NPLC = 10) and repeated 50 times to calculate a mean value. Therefore the cycle time t_C is limited to values which are an integer fraction of the integration window, ranging from 5 ms to 200 ms. The SmartFusion FPGA simulates the sensor and provides a defined set of values. From the individual sensor queries, it can determine the start of a sensor measurement cycle. This is used to generate a pulse to synchronize the current measurement.

[16] http://www.microsemi.com/products/fpga-soc/design-resources/dev-kits/smartfusion/smartfusion-evaluation-kit [2015-08-20]

Figure 5.10: Evaluation platform overview. The test chip and MCUs are connected via a crossbar switch to the MCU and FPGA evaluation device. This is controlled via a PC (reproduced from [GGHG11] with permission).

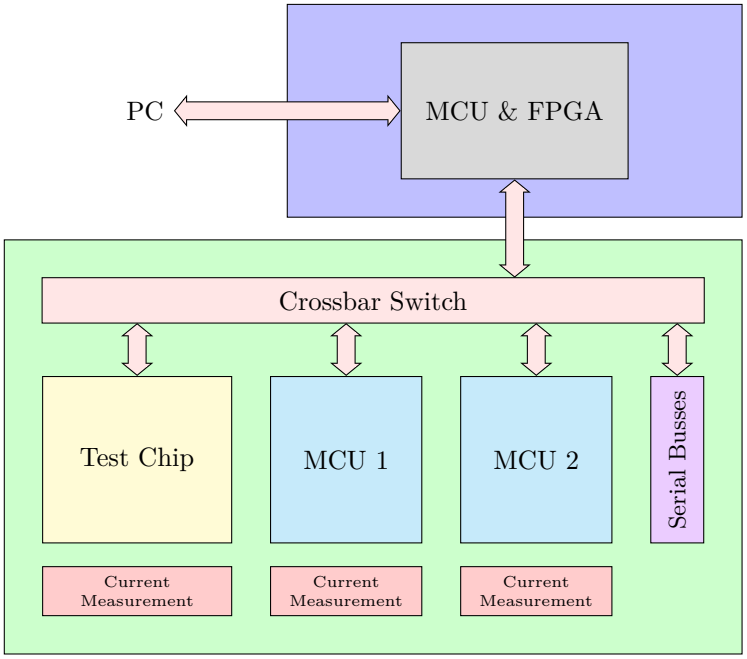
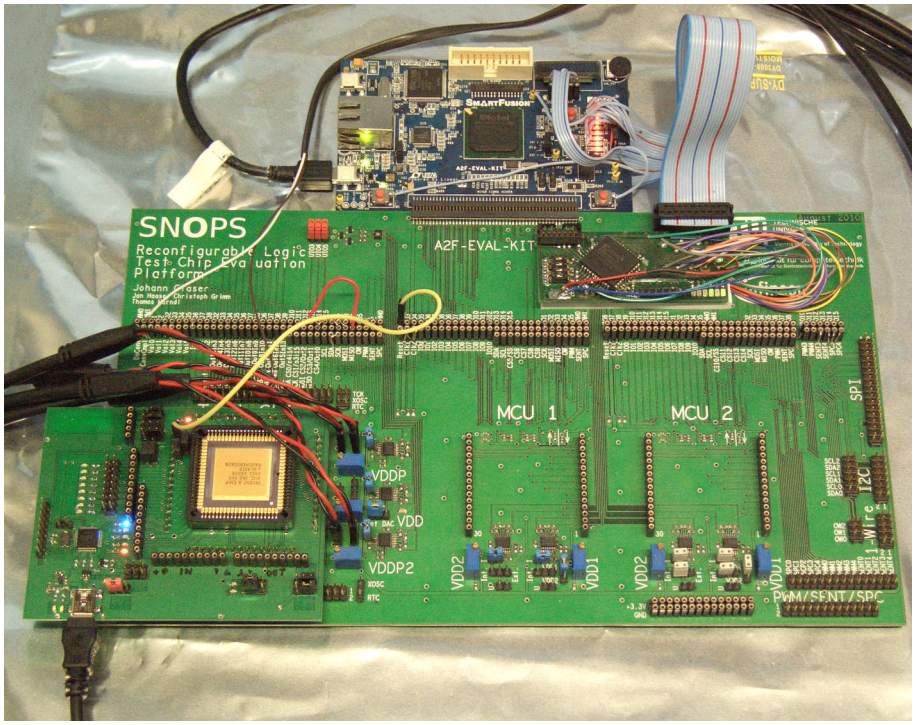


Figure 5.11: Evaluation platform with the WSN SoC carrier module and cables for the external supply and measurement.



For the characterization of the WSN SoC, the measurement setup was further extended by the author of this thesis. A WSN SoC carrier module PCB was developed to fit into the test chip module socket (see bottom left in Fig. 5.11). Besides the WSN SoC, it includes an NXP LPC11U34 MCU to download the firmware, for debugging, and for stand-alone testing. The MCU firmware is the same as used in the LPCXpresso board for the verification with prototype testing (cf. Sec. 5.1.3). The WSN SoC is connected to the LPC11U34 MCU and the SmartFusion FPGA without level-shifters. Therefore its supply voltage can not deviate from 3.3 V, otherwise current paths through the ESD diodes of input pins would confound the current measurements or destroy the chip.

In contrast to the MCUs characterized in [Ble15], the WSN SoC does not provide flash memory. Therefore its firmware must be downloaded after each power cycle. This feature was added to the Matlab programs which perform the measurements. The firmware used for the measurements is different from the firmware developed for the example applications. Two different versions were derived from the firmware developed by Georg Blemenschitz for the MSP430F1232 [Ble15]. One version performs the sensor measurement task as firmware and therefore was only slightly modified due to the different UART and SPI peripherals. The second version performs the sensor measurement using the reconfigurable module. This was extended with the driver for the reconfigurable module and the appropriate setup and interrupt service routines.

Furthermore, the measurement setup and the Matlab programs were changed to use a single Keithley 2602A SourceMeter instead of separate power supplies and multimeters for voltage and current measurement (see Fig. 5.12).[17] The SourceMeter provides two independent channels, each with concurrent supply and measurement. This enabled the concurrent measurement of both power domains of the WSN SoC. Previously two supply voltages were provided but only one current could be measured.

The measurement setup used a time consuming control loop to counterbalance the voltage drop of the wires and connectors. The author of this thesis extended the measurement setup and the Matlab program to utilize four-wire connections for supply and measurement. With this feature the SourceMeter automatically performs the control loop. This improvement reduced the duration to measure a single testcase from 20.3 s to 12.4 s, of which 10 s are the actual current measurement (50 times 200 ms). Considering that previously each power domain had to be measured separately, a speedup by a factor of 3.3 was achieved.

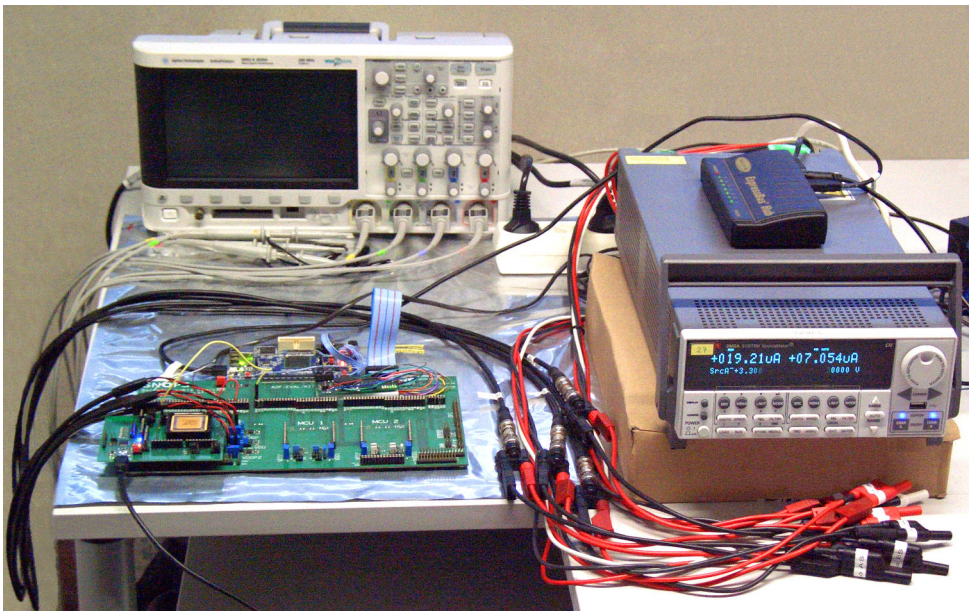
A third improvement was added which allows to continue the measurement of a test series, which was previously interrupted. For the analysis of the measurements, new routines were implemented according to the model derived in the previous section.

5.3.3 Power Analysis

To identify the individual contributions of the total power consumption inside the WSN SoC, power analysis is used. Note that power analysis is used to divide the total power into the contributions of the hierarchical modules of the WSN SoC, while the energy model and multiple linear regression are used to separate the contributions of the individual tasks of the measurement cycle. Firstly, from the SoC layout, the parasitics are extracted and saved as standard parasitic

[17] This functionality was already developed by Georg Blemenschitz but he had to change the measurement setup to separate devices when the SourceMeter was stolen from the lab. The measurements of the WSN SoC were performed at a different site where a SourceMeter was available.

Figure 5.12: Measurement setup used to characterize the WSN SoC. The evaluation platform on the bottom left is connected to the SourceMeter on the right with coax cables. Due to the pre-built measurement leads attached to the SourceMeter, sub-optimal extensions were required.



exchange format (SPEF) file with Cadence Encounter. Then the delays of the gates and the wires are calculated using Synopsys PrimeTime and saved as SDF file.

This file is used for post-P&R simulation to determine the switching activity of each individual net, which is written to switching activity interchange format (SAIF) files. The simulation is identical to the discussed HW/SW co-simulation but uses the post-P&R netlist and the above mentioned firmware for the measurements. A special testbench was developed which emulates the set up of the testcase parameters via UART, the synchronization of the measurements, and the processing of a test series. Additional Tcl scripts for Questa Sim were designed to start and stop the logging of the switching activity, analogous to the triggered measurement.

The SAIF file is loaded with Synopsys Design Compiler together with the post-P&R netlist and the SPEF file. Then the power analysis is performed with the `report_power` command for different operating conditions. Unfortunately no power models of the RAMs were available, therefore the according power consumption is not included. The resulting power reports separately state the cell and net power consumption as well as the static leakage power of the complete WSN SoC. These are finally parsed with Matlab programs, which were used to generate the following tables and diagrams.

To identify the power consumption of the individual components, hierarchical power reports are generated. These break down the total power to all individual hierarchical modules of the design. This also allows to determine the power consumption of the reconfigurable module in the separate power domain. Unfortunately the hierarchical power reports only specify three significant digits, which can not be increased. This leads to errors in the further analysis of the power consumption with the Matlab programs. To solve this problem, the power analysis was extended to generate a non-hierarchical power analysis report of only the reconfigurable module, which provides up to seven significant digits.

The comparison of the power analysis reports for different testcases revealed a bug in Questa Sim 10.0a when generating SAIF files. For signals with short glitches the values of `T0` and `T1`, which specify the total time the signal has the values '0' or '1', respectively, were reported wrong. This leads to erroneous assumptions of Design Compiler when propagating the switching activity through the netlist and therefore inaccurate power reports. The bug was still present in the then newest version 10.4 of Questa Sim. To solve this problem, the actual signal values were logged to value change dump (VCD) files during the simulation with Questa Sim. Afterwards these were converted to SAIF files with the Synopsys `vcd2saif` program and further used for power analysis. However, this heavily increased the file size and simulation run-time.[18]

5.3.4 Evaluation of the Hypothesis

In this section hypothesis 2 is evaluated. It states that the implementation of a task using a reconfigurable module requires less energy than the implementation using a CPU. To evaluate this hypothesis, three different approaches are used. First, the manually developed reconfigurable module is compared to commercial MCUs. Secondly, the same comparison is performed using an improved measurement setup and a more complex task. Finally, the CPU and the reconfigurable module of the WSN SoC are compared.

The first two comparisons use different chips which are manufactured with different semiconductor processes, which confounds the comparisons. One possible approach to mitigate this problem is to scale the results to a common semiconductor process. Unfortunately the required information on the employed process is only scarcely available for the MCUs.[19] Therefore the raw results are used for the comparisons. The influence of the different semiconductor processes is assumed smaller than the ratios of the power consumption so that the relations are not reversed and the main statements are preserved.

The third comparison using the WSN SoC is not confounded by different semiconductor processes and therefore provides the most valid result. The other two comparisons provide weaker evidence but are included here, because the underlying work and the evaluations are building blocks of this thesis.

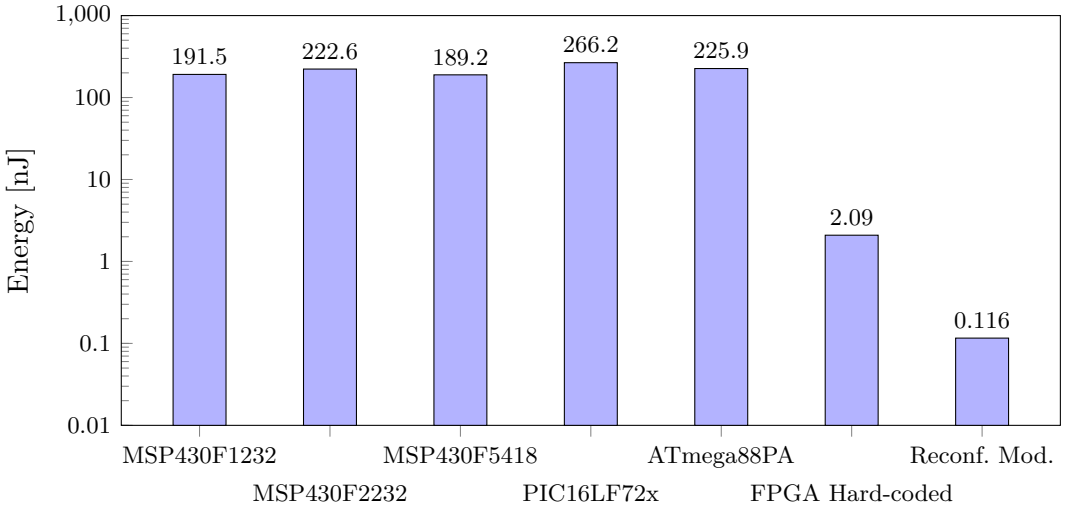
Manually Developed Reconfigurable Module

The approach to reduce the energy consumption of a WSN node by the inclusion of a reconfigurable CPU supplement module which off-loads the CPU was first investigated for five MCUs and an FPGA in [GHDG09]. The results were already discussed in Sec. 1.1.2, more detailed in Sec. 2.1.1, and are shown in Tab. 2.1 on p. 21 and Fig. 2.1. The implementation of the sensor interface task assumed a simple analog sensor connected to an ADC. The implementation using an FPGA reduced the energy consumption by a factor of 90.5 compared to the lowest power MCU.

[18] The VCD files grew to a total size of more than 50 GB for only a few testpoints.

[19] Inquiries were sent to the according vendors, but only Atmel answered the question, TI stated that the information is proprietary and cannot be shared, and Microchip did not respond. For documentation purposes: The ATmega88PA is manufactured in a 250 nm process. The letter "A" at the end denotes the "newer" process. Chips without "A" are manufactured in an older 350 nm process. http://electronics.stackexchange.com/q/169587/71799 [2015-08-20]

Figure 5.13: Energy consumption of one sensor measurement performed by ultra-low-power MCUs, an FPGA and the manually developed reconfigurable module [GGHG11] (reproduced with permission). Please note the logarithmic scale of the vertical axis.



This evaluation used a commercial high-performance FPGA as reconfigurable architecture which is fine-grained and optimized for speed. In this thesis the inclusion of a mixed-grained and application domain specific reconfigurable module is proposed. Therefore at an early stage of the development of the design methodology, a reconfigurable module was manually developed (cf. Sec. 5.2).

Its energy consumption was measured and compared to the MCU and FPGA implementations in [GGHG11]. The results are shown in Fig. 5.13, which adds one bar with the power consumption of the manually developed reconfigurable module to Fig. 2.1. The reconfigurable module consumes 0.116 nJ per sensor measurement, which is a reduction by a factor of 18 compared to the FPGA implementation (2.09 nJ), a reduction by a factor of 1,631 compared to the lowest power MCU MSP430F5418 (189.2 nJ), and a factor of 1,889 compared to the mean of the MCUs (219.1 nJ).

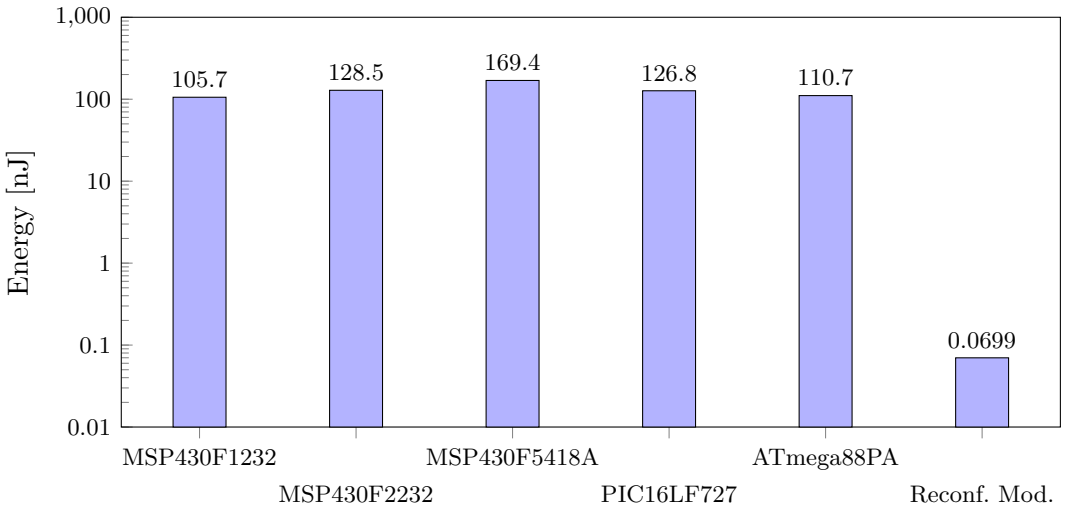
These results demonstrate a reduction of the energy consumption for the employment of a reconfigurable CPU supplement module.

Improved Measurement Setup

The energy consumption of the MCUs used in Fig. 5.13 was determined by manually investigating the number of clock cycles required to execute the sensor interface task. Therefore, for each MCU architecture the C source code was compiled to assembler code. Then for each instruction the according number of clock cycles was added up. The duration of the execution was calculated by the total number of clock cycles multiplied by the clock period plus constant delays of the ADC and the sensor. The total duration was multiplied by the supply voltage and the typical operating current specified in the datasheets. For more details see Sec. 2.1.1 and [GHDG09].

To obtain measurements, the author of this thesis supervised the diploma thesis [Ble15] of Georg Blemenschitz. Additionally the more complex sensor ADT7310 was selected, which was also used for the evaluation of the WSN SoC. The MCUs and the manually developed reconfigurable

Figure 5.14: Energy consumption of one sensor measurement performed by ultra-low-power MCUs and the manually developed reconfigurable module [Ble15]. The values were determined for the ADT7310 sensor interface task using the measurement setup discussed in Sec. 5.3.2. The MCUs are supplied with 1.8 V, the reconfigurable module with 0.8 V. The MSP430F5418A and the PIC16LF727 operate at 10 MHz, the other MCUs at 4 MHz, and the reconfigurable module at 100 kHz. Please note the logarithmic scale of the vertical axis.



module executed the measurement procedure described in Sec. 5.3.1. The measurements were performed with the evaluation platform and the measurement setup discussed in Sec. 5.3.2.

For the analysis a different approach than discussed in Sec. 5.3.1 was used: For each MCU and for the manually developed reconfigurable module the optimum operating point (frequency and supply voltage) with the lowest energy consumption per measurement was determined. The resulting values are given in Fig. 5.14.[20] The reconfigurable module shows a reduction by a factor of 1,512 compared to the lowest power MCU MSP430F1232 and by a factor of 1,834 compared to the mean 128.2 µA of the MCUs.

These results also demonstrate a reduction of the energy consumption for the employment of a reconfigurable CPU supplement module.

WSN SoC Developed with the Design Methodology

In the previous two comparisons, the test chip was manufactured in a different process technology than the MCUs and the reconfigurable module was developed manually. To enable a direct comparison with identical process technology and to evaluate the outcome of the design methodology discussed in this thesis, the WSN SoC is evaluated. The energy consumption to perform a sensor measurement is compared for an implementation using the CPU and an implementation using the reconfigurable module. Please refer to appendix A for a detailed discussion and selection of different variants of the sensor interface task for both implementations.

[20] In the previous comparison the results were determined for MCU families, e.g., the PIC16LF72x, while in this comparison actual MCU chips were used, e.g., the PCI16LF727. This is the reason for the slightly different labels shown at the horizontal axes of Figs. 5.13 and 5.14.

Both implementations were evaluated at an operating frequency of 1 MHz, 4 MHz, and 10 MHz with eight different cycle times t_C of 5 ms, 10 ms, 20 ms, 33.33 ms, 40 ms, 50 ms, 100 ms, and 200 ms. The reconfigurable module implementation was additionally evaluated at the lower operating frequencies 32.765 kHz[21], 100 kHz, and 500 kHz. Therefore a total of 24 testcases for the CPU implementation and 48 testcases for the reconfigurable module implementation were analyzed.

The test chip samples of the WSN SoC were measured using the evaluation platform and measurement setup discussed in Sec. 5.3.2. Additionally, power analysis was used to determine the power consumption of all testcases. The resulting current consumption values were evaluated with the energy model discussed in Sec. 5.3.1, see Eqn. 5.1 on p. 138. Additionally the measurements performed by [Ble15] were evaluated with this model.

The resulting regression coefficients are shown in Tab. 5.2. For the WSN SoC 24 test chip samples and for the manually designed reconfigurable module 12 test chip samples were measured. The regression coefficients were calculated for each sample individually. In Tab. 5.2 the mean and standard deviation of each coefficient are shown.

In the column "Residuals" the difference between the measured and the predicted (i.e., calculated) energy E_A of a total sensor measurement cycle is given. These show a very good fitting with only 0.036% to 2.659% error.

The large difference between the values determined with measurement and with power analysis for the CPU implementation are caused by the missing energy model for the memories. The accuracy of the power analysis is also limited for the reconfigurable module. More details are discussed in appendix A.

The energy consumption per sensor measurement is represented by b_0. These values are shown in Fig. 5.15 and will be discussed below. The factor b_1 and b_2 describe the energy per clock cycle for the sensor delay and for the period delay, respectively. Both values are effectively zero for the CPU implementation of the WSN SoC because, due to a design issue, the delays had to be implemented externally. An interrupt was used to notify the CPU to exit LPM3. More details on this workaround are discussed in Sec. A.1.3. The reconfigurable module requires 33.91 pJ and 32.06 pJ per clock cycle for the counters. The values are similar because the 16-bit counter itself requires less energy than the 32-bit counter, but due to the higher fan-out of the counter value into the interconnect, it causes more switching activity and this difference is over-compensated. For the MCUs partially large negative values were calculated for an unknown reason.

The factor b_3 describes the leakage and active power (not energy) which is different from the inactive testcase. This is also reflected by the high standard deviation (±144.8% and ±31.18%) of the factor. The values of 14.65 nW and 8.16 nW determined by power analysis are non-zero for an unknown reason. For the manually developed reconfigurable module no inactive testcases were measured, therefore b_3 has the high value 1,963.5 nW. The large values for the five MCUs of 342.8 nW–943.3 nW occur because the compensation of the voltage drop of the supply leads and connectors was performed by a less precise algorithm than the measurements performed with the WSN SoC using the SourceMeter. The factors b_4 and b_5 are quadratic terms and cover the dependency on the number of active bits used by the counter start values.

The SPI and I^2C master peripherals for the reconfigurable module in the WSN SoC are supplied from the CPU power domain. However, the respective energy consumption is eliminated by the

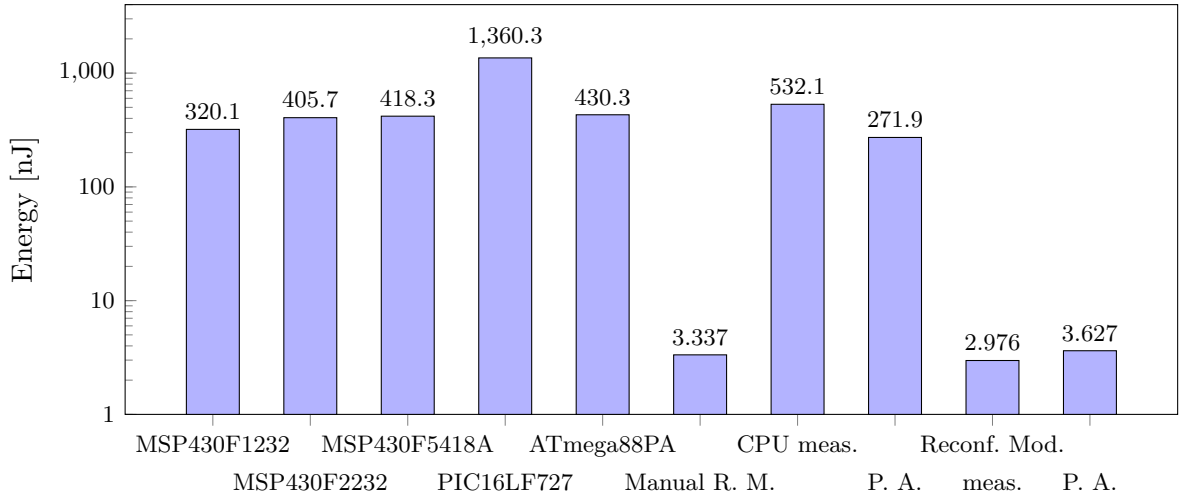
[21] The typical crystal frequency of 32.768 kHz for RTCs was generated from a 100 MHz with a frequency divider of 3,052, which results in the small deviation.

Table 5.2: Regression coefficients of the energy consumption per sensor measurement (explanation in the text, R. M.: Reconfigurable Module, P. A.: Power Analysis).

Chip	V_{DD}	#	b_0	b_1 (n_S)	b_2 (n_P)	...	Residuals
MSP430F1232	3.0 V	1	320.1 nJ	-3.760 pJ	-0.073 pJ	...	0.507 %
MSP430F2232	3.0 V	1	405.7 nJ	-2.131 pJ	-0.196 pJ	...	1.341 %
MSP430F5418A	3.0 V	1	418.3 nJ	-20.27 pJ	-0.045 pJ	...	1.086 %
PIC16LF727	3.0 V	1	1360.3 nJ	-149.3 pJ	-0.044 pJ	...	0.688 %
ATmega88PA	3.0 V	1	430.3 nJ	-8.411 pJ	-0.345 pJ	...	2.659 %
Manual R. M.	1.2 V	12	3.337 nJ ± 1.736%	35.03 pJ ± 1.580%	34.95 pJ ± 1.661%	...	0.036 % ± 42.23%
CPU meas.	3.3 V	24	532.1 nJ ± 0.485%	-0.499 pJ ± 81.07%	-0.035 pJ ± 158.7%	...	0.910 % ± 30.01%
CPU P. A.	3.3 V	1	271.9 nJ	-0.063 pJ	-0.006 pJ	...	0.323 %
Reconf. Mod. meas.	3.3 V	24	2.976 nJ ± 0.466%	33.91 pJ ± 0.427%	32.06 pJ ± 0.438%	...	0.156 % ± 9.594%
Reconf. Mod. P. A.	3.3 V	1	3.627 nJ	37.22 pJ	33.47 pJ	...	0.246 %

Chip	V_{DD}	#	...	b_3 (t_C)	b_4 ($n_S \cdot n_C$)	b_5 ($n_P \cdot n_C$)	b_4 ($n_S{}^2$)	b_5 ($n_P{}^2$)
MSP430F1232	3.0 V	1	...	854.6 nW	-5.712e-18 J	1.181e-19 J		
MSP430F2232	3.0 V	1	...	943.3 nW	7.450e-18 J	1.010e-20 J		
MSP430F5418A	3.0 V	1	...	342.8 nW			6.889e-16 J	1.528e-20 J
PIC16LF727	3.0 V	1	...	597.0 nW			5.109e-15 J	2.545e-20 J
ATmega88PA	3.0 V	1	...	402.9 nW	1.514e-17 J	-1.068e-20 J		
Manual R. M.	1.2 V	12	...	1963.5 nW ± 7.437%	-6.835e-18 J ± 183.2%	-8.107e-20 J ± 539.0%		
CPU meas.	3.3 V	24	...	-58.42 nW ± 144.8%	1.891e-18 J ± 140.8%	-4.811e-21 J ± 511.5%		
CPU P. A.	3.3 V	1	...	14.65 nW	2.824e-19 J	-3.244e-21 J		
Reconf. Mod. meas.	3.3 V	24	...	6.323 nW ± 31.18%	2.545e-18 J ± 36.79%	3.800e-20 J ± 36.35%		
Reconf. Mod. P. A.	3.3 V	1	...	8.160 nW	-8.457e-18 J	-4.472e-20 J		

Figure 5.15: Energy consumption for the actual sensor measurement (values from Tab. 5.2). Please note the logarithmic scale on the vertical axis.



subtraction of I_0 and therefore does not confound the results of the CPU implementation. On the other hand, the power consumption of the "Simple SPI" CPU peripheral is included in b_0 because it is only active during the sensor start and sensor query operations (cf. Fig. 5.9 on p. 137). However, this peripheral consumes only 0.5327 µA when performing 200 meas./s at 1 MHz (cf. Tab. A.6 on p. 188), i.e., 2.664 nA per meas./s, compared to 160.7 nA per meas./s (cf. Tab A.5 in p. 186) and is therefore neglected.

The differences of the values shown in Tab. 5.2 and Fig. 5.15 to the values of [Ble15] shown in Fig. 5.14 are mainly caused by the differing supply voltage. Here 3.0 V is used because some MCUs do not support the higher operating frequencies at lower supply voltages. That would reduce the number of testcases and datapoints available for the linear regression used in the analysis. In [Ble15] 1.8 V are used at the optimum operating point. As discussed in Sec. 2.1.1 the power consumption is proportional to V^2 which results in a difference by a factor of 2.778 which approximately relates the two sets of values. The PIC16LF727 requires four clock cycles for each instruction, while the other MCUs execute one instruction every clock cycle. Due to its lower current consumption at a given frequency, the difference to the other MCUs is less than a factor of four. The reason why this did not appear in the results of [Ble15] is unknown. The large difference of the results for the manually developed reconfigurable module are caused by the different method for the analysis used in [Ble15].

As mentioned above, the factor b_0 represents the energy consumption of the actual sensor measurement which is used to compare the different implementations. The energy consumption for a sensor measurement performed by the WSN SoC and implemented as firmware is 532.1 nJ while the implementation with the reconfigurable module consumes 2.976 nJ. This means a reduction by a factor of 178.8. Therefore the design methodology leads to a reconfigurable module which consumes less energy than an implementation with a CPU. Thus, hypothesis 2 is provisionally accepted.

5.4 Chip Area

In this section, the chip area of the reconfigurable module is evaluated and compared to the area of all example applications in parallel to evaluate the first sub-hypothesis of hypothesis 3. The second sub-hypothesis is evaluated by a comparison with the chip area of (embedded) FPGAs. Therefore, the chip area of the basic cells of different FPGA architectures was determined (see Sec. 5.4.1).

Before the comparison, the chip area of the WSN SoC is characterized in Sec. 5.4.2. This is followed by a characterization of the reconfigurable module in Sec. 5.4.3. Finally, in Sec. 5.4.4 both sub-hypotheses are evaluated.

5.4.1 FPGA Area

To compare the area of the reconfigurable module to the size of an (embedded) FPGA, the size of the basic cells of FPGAs has to be determined. The architectures reviewed in Ch. 2 and additional architectures are investigated. The results are summarized in Tab. 5.3 on p. 150. Additionally the area is scaled to the same process node 350 nm as used for the WSN SoC. Therefore the area is multiplied by the square of the ratio of 350 nm and the feature size L of the FPGA

$$A_{350\,\mathrm{nm}} = A_L \cdot \left(\frac{350\,\mathrm{nm}}{L}\right)^2 .$$

This does not take the number of metal layers into account which are an important factor in FPGA architectures. Therefore the comparison between the reconfigurable module and the FPGA architectures favors the latter. From the scaled value the equivalent area of the smallest entity of the FPGA is calculated.

[GZR99] reported an FPGA with 8×8 CLBs with four 3-input LUTs and three D-FFs manufactured in a 250 nm process with 6 metal layers on an area of 2 mm×2 mm. This results in an area of 62,500 µm^2 per CLB. The Pleiades Maia process contains a 4×8 array of logic blocks with the same architecture as [GZR99] with a total area of 2.76 mm^2 in a 250 nm process with 6 metal layers [ZPG+00]. The area of a logic block is 86,250 µm^2.

[AS06] developed an eFPGA with tiles with a size of 2,070 µm^2 to 4,450 µm^2 in a 90 nm CMOS process. Unfortunately the number of LUTs per tile is not documented. An automatic FPGA generator was presented by [KER05]. They designed an 8×8 array of CLBs with three 4-input LUTs each. The array required a total area of 1,041 µm×1,225 µm = 1.275 mm^2 in a 180 nm CMOS process with 6 metal layers. The area of a single CLB is therefore 19,925 µm^2. [Kuo04] reported the area of a CLB of the Xilinx Virtex-E FPGA with 35,462 µm^2. However, it is not clear whether this number was determined for the real device or an estimate by replicating the circuit. Each CLB contains four 4-input LUTs and D-FFs.

[WAWS03] developed an 8×8 array of 3-input LUTs but without D-FFs in a 180 nm process to implement the next-state logic of reconfigurable FSMs, which required 684,600 µm^2. Therefore each LUT has a size of 10,697 µm^2.

In contrast to academic FPGAs, the size of commercial FPGAs is kept as a trade secret. Therefore the area was estimated with indications gathered from different sources. For the Lattice/SiliconBlue iCE65 architecture, the size of a logic cell with a 4-input LUT and a D-FF was estimated as 1,750 µm^2 in [GDHG11] from die photographs and coordinate specifications of the bonding pads.

The Xilinx 7 Series FPGAs (Artix-7, Kintex-7, Virtex-7) use identical CLB designs manufactured with a 28 nm high-κ metal gate CMOS process [Xil14a, Xil15b]. Each CLB holds eight 6-input LUTs which each can be used as two 5-input LUTs with shared inputs, i.e., a total of 16 5-input LUTs. Additionally, each CLB holds 16 D-FFs. Therefore the CLB size is divided by 16 to calculate the size of the smallest entity.

The Artix-7 XC7A200T die size is 11.10×12.05 mm [Xil14b, p. 220] which is divided in 2 clock regions in the horizontal direction and five clock regions in the vertical direction according to the Vivado v2014.2 placement editor. Each clock region has 50 rows with 42 CLBs plus other cells like RAMs and DSP cells. Only 55.3% of the total width of a clock region is used by CLBs and their associated switch boxes. This was estimated from the Vivado placement editor with the assumption that the visualization is to scale. This results in an estimated size of a CLB as 73.1 µm×48.2 µm = 3,522 µm^2.

The Xilinx Kintex-7 XC7K70T die size is 5.99 mm×9.68 mm and the XC7K160T die size is 8.54 mm×12.05 mm [Xil14b, p. 230f]. The Kintex-7 XC7K70T is built of four clock regions in vertical direction and two asymmetric clock regions in horizontal direction. Each clock region has 50 rows of CLBs with 31 CLBs across the total width of the chip. With the visualization of the FPGA floorplan of Vivado, the width occupied with CLBs together with the associated switchboxes was estimated as 41.4% across the total width of the chip. This results in an estimated CLB size of 80.1 µm×48.4 µm = 3,876 µm^2. The Kintex-7 XC7K160T is built of 2×5 clock regions with 28×50 CLBs each. The CLBs occupy 54% of the width. This results in an estimated CLB size of 82.3 µm×48.2 µm = 3,967 µm^2.

Table 5.3: Size of FPGA basic cells at the FPGA chip, scaled to a 350 nm process, and divided to give the smallest entity.

Architecture	Cell Size	Process	Cell Size at 350 nm	
[GZR99]	62,500 µm^2/CLB	250 nm	30,625 µm^2/3-LUT	40,833 µm^2/3-LUT+D-FF
[ZPG+00]	86,250 µm^2/CLB	250 nm	42,263 µm^2/3-LUT	56,350 µm^2/3-LUT+D-FF
[AS06] min.	2,070 µm^2/Tile	90 nm		31,306 µm^2/Tile
[AS06] max.	4,450 µm^2/Tile	90 nm		67,299 µm^2/Tile
[KER05]	19,925 µm^2/CLB	180 nm	75,335 µm^2/CLB	25,112 µm^2/4-LUT
[WAWS03]	10,697 µm^2/3-LUT	180 nm		40,443 µm^2/3-LUT
Virtex-E	35,462 µm^2/CLB	180 nm		33,519 µm^2/4-LUT+D-FF
iCE65	1,750 µm^2/D-FF	65 nm		50,740 µm^2/4-LUT+D-FF
XC7A200T	3,522 µm^2/CLB	28 nm	550,303 µm^2/CLB	34,394 µm^2/5-LUT+D-FF
XC7K70T	3,876 µm^2/CLB	28 nm	605,658 µm^2/CLB	37,854 µm^2/5-LUT+D-FF
XC7K160T	3,967 µm^2/CLB	28 nm	619,846 µm^2/CLB	38,740 µm^2/5-LUT+D-FF

The three estimates for the Xilinx 7 Series vary from 3,522 µm^2–3,967 µm^2 but should be identical. Therefore the mean value 3,788 µm^2 is used in the following comparisons. The individual cell sizes of all reviewed architectures vary considerably due to the logic contents and the process technology (see Tab. 5.3). The sizes of the smallest entity (LUT+D-FF) scaled to the common process node 350 nm only show a small variation.

5.4.2 Characterization of the WSN SoC

As already briefly mentioned in Sec. 5.1.4, the WSN SoC has a size of 3,910 µm×3,610 µm = 14.1 mm^2. The core area inside the pad frame is 3,200 µm×2,900 µm = 9.28 mm^2. The actual core logic area is slightly smaller (8.98 mm^2) due to the surrounding power ring.

The individual modules of the core are highlighted in Fig. 5.16, and the respective area and colors are listed in Tab. 5.4. The top bar diagram in Fig. 5.17 shows the proportions. The largest units are the reconfigurable module with 4.20 mm^2 and the 8 kB program memory with 2.17 mm^2.

The OpenMSP430 CPU itself requires 0.49 mm^2. The "GPIO", "Timer", "UART", and "SPI" modules are OpenMSP430 peripherals. "SPI Master" and "I2C Master" are exclusively used by the reconfigurable module and therefore placed in the top right corner of the chip core.

The clock tree root also contains the reset tree root. Note that all modules also contain buffers of the clock and the reset trees, therefore at the top level only the according tree *roots* are included.

To improve the routability of the logic design, the P&R tool keeps spaces between the standard cells. These are filled with filler cells. Additionally, below the vertical power stripes no logic cells are placed because the power wires block the logic routing wires. These stripes are also filled with filler cells and are visible as thin vertical lines in Fig. 5.16. The non-reconfigurable part of the WSN SoC contains a total of 0.49 mm^2 filler cells which corresponds to 12.51%.

In Tab. 5.4 the row "Core Logic" is the sum of the rows listed above, which is only 57.48% of the total chip area due to the size of the pad frame. "Total Core" specifies the core area inside the pad frame of 32×29 pads with 100 µm width each. The row "Pads" consists of 60 digital IO pads, five GND and five VDD pads (three for the CPU power domain and two for the reconfigurable module power domain, each), 36 pad filler cells and 3 corner cells (see the full layout in Fig. 5.4 on p. 129). The row "Chip" specifies the whole chip area and includes the process dependent scribe border and the free area used by the project located at the bottom right, which is not included in any of the above numbers.

5.4.3 Characterization of the Reconfigurable Module

The total area of the reconfigurable module is 4.20 mm^2. Its individual modules are highlighted in Fig. 5.18 and the respective area values and colors are listed in Tab. 5.5 and shown in the bottom bar diagram in Fig. 5.17.

The largest contributors are the interconnect with 0.69 mm^2 (16.52%) and its configuration registers with 0.47 mm^2 (11.24%). This also includes the configuration for the reconfigurable signals with the usage type `config`. Further, the two TR-FSM instances with 0.39 mm^2 (9.27%) and 0.88 mm^2 (20.92%). The TR-FSM area values include the configuration registers. See Tab. 5.1 on p. 123 for the specification of the TR-FSM instances. All cells only require a small fraction of the area with a total of 158,704.03 µm^2 (3.78%).

The row "Param" specifies the parameter registers written by the CPU and used by the reconfigurable module. The read multiplexer for the parameters in the opposite direction is contained in the "Reconf. Mod. Local". The interface for configuration and parameterization, which are connected to the OpenMSP430 External Peripheral Interface, also only require a small area of 1.23%. The row "Reconf. Mod. Local" further contains the clock and reset tree root. The reconfigurable module also contains 1.39 mm^2 of filler cells, which equals 33.10%. In other words, the

Figure 5.16: WSN SoC core area with the individual units of the non-reconfigurable section highlighted. The colors and the area are listed in Tab. 5.4.

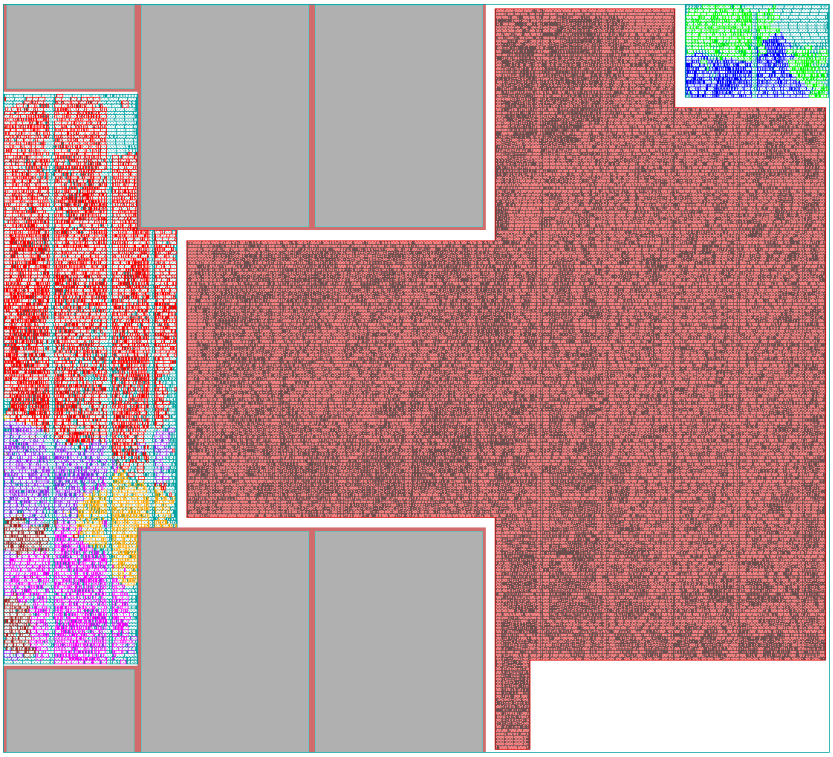


Table 5.4: Division of the WSN SoC core area as shown in Fig. 5.16 plus the pad frame (cf. Fig. 5.4 on p. 129).

Module	Color	Area	% Core	% Chip
OpenMSP430	red	0.493 mm^2	6.08%	3.50%
Program Memory	gray	2.167 mm^2	26.71%	15.35%
Data Memory	gray	0.317 mm^2	3.90%	2.24%
GPIO	violet	0.084 mm^2	1.03%	0.59%
Timer	magenta	0.103 mm^2	1.27%	0.73%
UART	orange	0.060 mm^2	0.74%	0.42%
SPI	brown	0.039 mm^2	0.48%	0.27%
SPI Master	green	0.061 mm^2	0.75%	0.43%
I2C Master	blue	0.065 mm^2	0.80%	0.46%
Clock Tree Root		0.057 mm^2	0.70%	0.40%
Filler	cyan	0.490 mm^2	6.03%	3.47%
Reconf. Mod.	red/gray	4.200 mm^2	51.76%	29.75%
Core Logic		8.114 mm^2	100.00%	57.48%
Total Core		9.280 mm^2		65.75%
Pads		3.956 mm^2		28.03%
Chip		14.115 mm^2		100.00%

Table 5.5: Division of the area of the reconfigurable module as shown in Fig. 5.18.

Module	Color	Area	%
Interconnect	black	693,608 µm^2	16.52%
TRFSM0	green	389,135 µm^2	9.27%
TRFSM1	blue	878,457 µm^2	20.92%
WordMuxDual0	cyan	2,075 µm^2	0.05%
WordMuxDual1	cyan	2,093 µm^2	0.05%
Byte2WordSel	cyan	8,973 µm^2	0.21%
ByteMuxDual0	cyan	1,056 µm^2	0.03%
ByteMuxDual1	cyan	1,056 µm^2	0.03%
ByteMuxQuad	cyan	2,257 µm^2	0.05%
AddSubCmp0	cyan	7,790 µm^2	0.19%
AddSubCmp1	cyan	7,207 µm^2	0.17%
ByteRegister0	cyan	3,003 µm^2	0.07%
ByteRegister1	cyan	2,948 µm^2	0.07%
WordRegister0	cyan	5,806 µm^2	0.14%
WordRegister1	cyan	5,806 µm^2	0.14%
WordRegister2	cyan	5,624 µm^2	0.13%
AbsDiff	cyan	12,594 µm^2	0.30%
Counter320	cyan	29,994 µm^2	0.71%
Counter321	cyan	29,648 µm^2	0.71%
Counter0	cyan	15,379 µm^2	0.37%
Counter1	cyan	15,397 µm^2	0.37%
Config	magenta	472,188 µm^2	11.24%
Param	violet	34,507 µm^2	0.82%
Config&Param Intf.	orange	51,615 µm^2	1.23%
Reconf. Mod. Local		131,338 µm^2	3.13%
Filler	gray	1,390,262 µm^2	33.10%
Total		4,199,813 µm^2	100.00%

area utilization is as low as 66.9%. However, these filler cells are poorly distributed which causes the congestion problems discussed in Sec. 5.1.5.

5.4.4 Evaluation of the Hypothesis

Hypothesis 3 states two sub-hypotheses which compare the area of the reconfigurable module, firstly to a non-reconfigurable but switchable implementation, and secondly to an implementation using an embedded FPGA. To evaluate the first sub-hypothesis, a new function was added to FlowProc which generates an HDL module with instances of the HDL modules of all example applications and MUXes for the ports. This module was synthesized together with the HDL code of the example applications for the same semiconductor process and standard cell library as the WSN SoC.

The synthesis was performed using boundary optimization, i.e., the logic was optimized across the module boundaries. Instead of the generation of a layout, the additional area required for clock buffers, reset buffers, and net congestions was estimated by assuming 70% area utilization.

Figure 5.17: Division of the WSN SoC area and of the reconfigurable module area into the individual modules. The colors correspond to Figs. 5.16 and 5.18 and Tabs. 5.4 and 5.5.

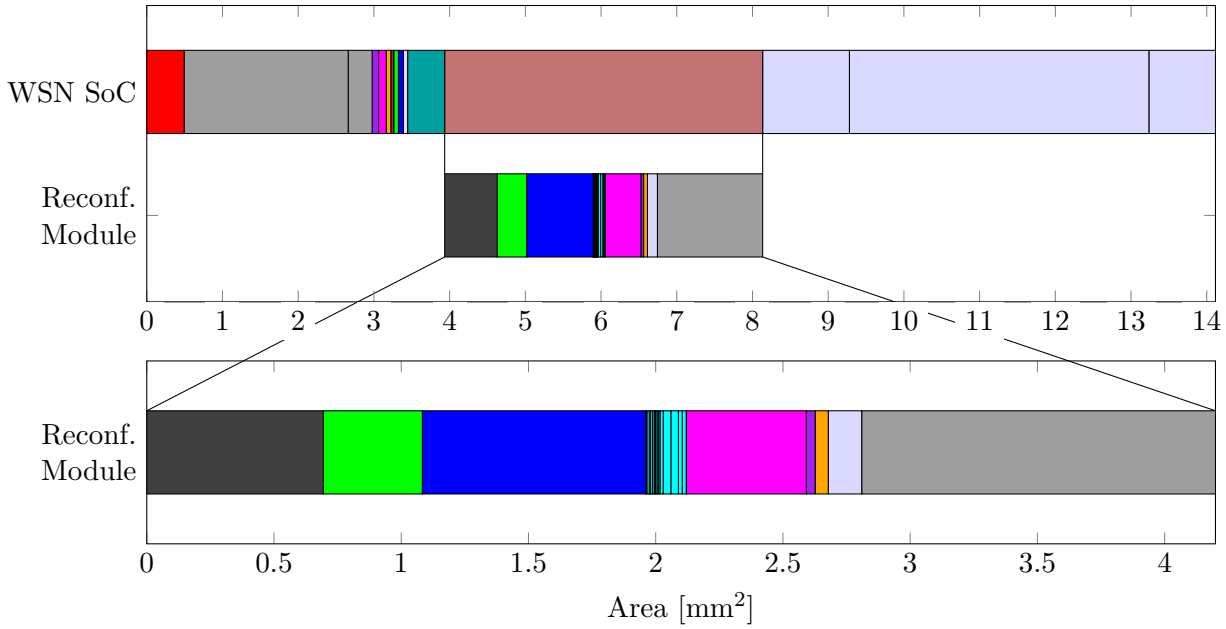
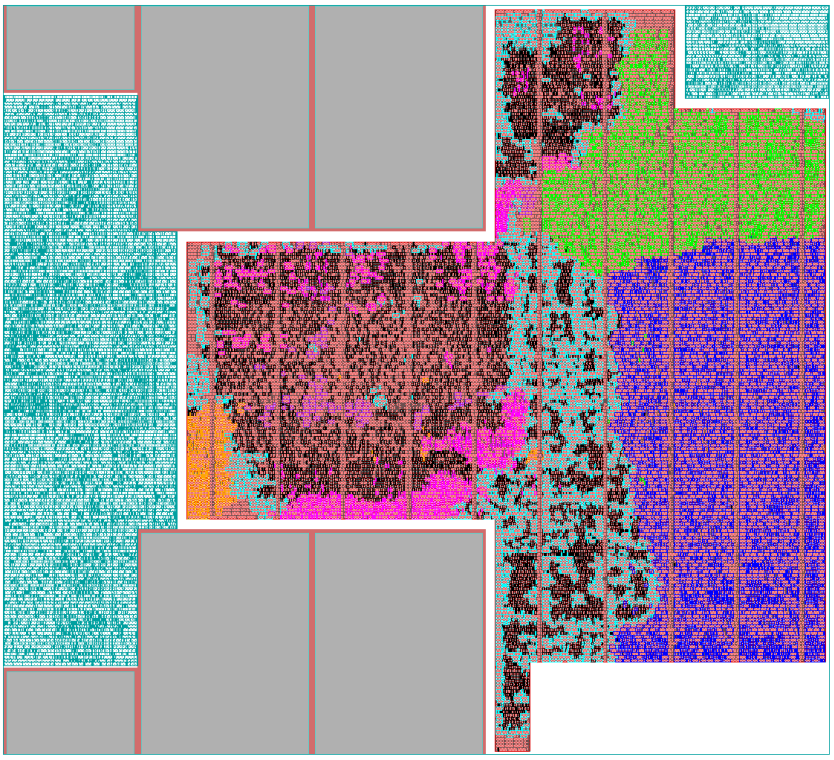


Figure 5.18: WSN SoC core area with the individual units of the reconfigurable module highlighted. The colors and the area are listed in Tab. 5.5.

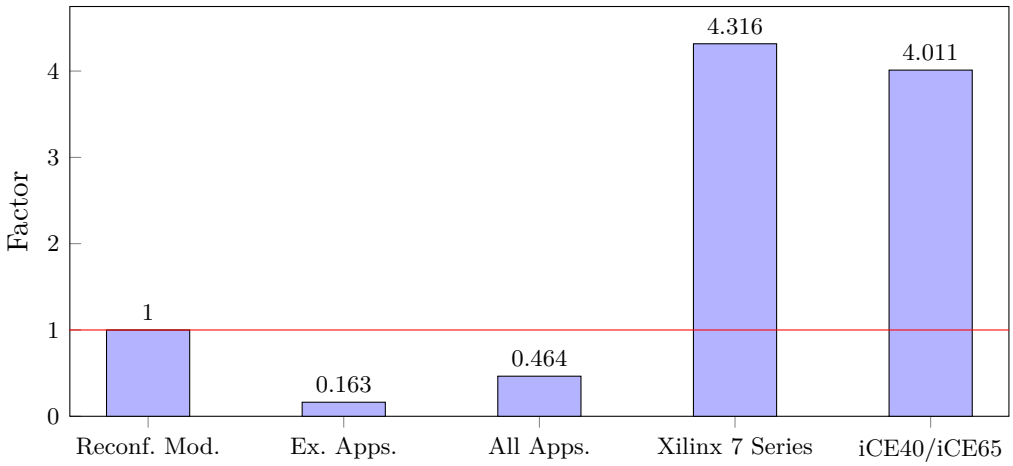


The area reported after synthesis was scaled accordingly. The resulting area for each example application, the MUXes, and the total area are given in the fourth column of Tab. 5.6.

The second column shows the area of the reconfigurable module. The third column shows the area of each application synthesized individually. The differences to the area for each example application in the switchable implementation is caused by the use of stronger and therefore larger output drivers of the cells and by the boundary optimization.

The area of the switchable implementation is 0.68 mm^2. The reconfigurable module (4.20 mm^2) requires 6.14 times of that area (see Fig. 5.19).

Figure 5.19: Area comparison of the reconfigurable module with the parallel implementation of all example applications, all applications, and an implementation using two FPGA architectures (values from Tab. 5.6).



However, the module is reconfigurable and can implement new applications in the post-silicon design phase. The same procedure as described above was performed using also the new applications. The results are stated in the fifth column of Tab. 5.6. The switchable implementation which provides all applications requires a total area of 1.95 mm^2. The reconfigurable module requires 2.15 times of that area (see Fig. 5.19).

Both factors mean, that the design methodology leads to a reconfigurable module which chip area is *larger* than the parallel implementation of all (example) applications. Therefore the first sub-hypothesis is rejected.

The second sub-hypothesis compares the reconfigurable module to an implementation using an (e)FPGA. Unfortunately no *embedded* FPGA architecture could be used for the comparison because no synthesis and implementation tools were available. Instead, stand alone FPGA architectures are used.

To evaluate this hypothesis, each application was individually synthesized and implemented using Xilinx Vivado v2014.2 with default settings for the Artix XC7A200TFBG484-3 and the two Kintex XC7K70TFBG484-3 and XC7K160TFBG484-3 FPGAs. These produced identical resource utilization results for all three devices, which are therefore summarized as “Xilinx 7 Series” in Tab. 5.6. The table lists the number of LUT+D-FF combinations. For some of these combinations only the D-FF or only the LUT is used. The area for an eFPGA which provides exactly the required resources is calculated by multiplying the resource utilization with the mean of the estimated area of the three architectures in Tab. 5.3.

Table 5.6: Area comparison of the reconfigurable module with the parallel implementation of all example applications, all applications, and an implementation using two FPGA architectures.

Application	**Reconf. Mod.**	**Each App.**	**Ex. Apps.**	**All Apps.**	**Xilinx 7 Series**		**iCE40/iCE65**	
	µm²	µm²	µm²	µm²	LUT+D-FFs	µm² @ 350 nm	LUT+D-FFs	µm² @ 350 nm
ADT7310		109,571 µm²	95,420 µm²	95,420 µm²	211	15,612,077 µm²	276	14,004,142 µm²
MAX6682		77,634 µm²	62,348 µm²	62,348 µm²	145	10,728,678 µm²	205	10,401,627 µm²
MAX6682Mean		125,901 µm²	110,344 µm²	110,344 µm²	231	17,091,895 µm²	283	14,359,320 µm²
ADT7410		95,130 µm²	82,914 µm²	82,888 µm²	206	15,242,122 µm²	247	12,532,692 µm²
SlowADT7410		131,496 µm²	113,074 µm²	113,074 µm²	232	17,165,885 µm²	331	16,794,822 µm²
ExtADC		58,329 µm²	45,578 µm²	45,708 µm²	113	8,360,970 µm²	165	8,372,041 µm²
ADT7310P32S16		109,999 µm²		95,602 µm²	200	14,798,177 µm²	295	14,968,195 µm²
ADT7310P16S16		91,213 µm²		80,418 µm²	174	12,874,414 µm²	242	12,278,994 µm²
ADT7310P32S32		128,253 µm²		110,604 µm²	217	16,056,022 µm²	332	16,845,562 µm²
ADT7310P32LS16		110,012 µm²		95,368 µm²	245	18,127,767 µm²	290	14,714,497 µm²
ADT7310P32S16L		109,870 µm²		95,914 µm²	166	12,282,487 µm²	296	15,018,935 µm²
ADT7310P32LS16L		109,896 µm²		96,044 µm²	197	14,576,204 µm²	291	14,765,237 µm²
ADT7310P16LS16L		91,486 µm²		80,990 µm²	160	11,838,542 µm²	244	12,380,473 µm²
ADT7310P16LS32L		109,636 µm²		95,836 µm²	187	13,836,296 µm²	281	14,257,840 µm²
ADT7310P32LS32L		127,890 µm²		110,760 µm²	227	16,795,931 µm²	329	16,693,343 µm²
ExtADCSimple		33,027 µm²		18,694 µm²	69	5,105,371 µm²	66	3,348,817 µm²
blinki		38,963 µm²		31,694 µm²	76	5,623,307 µm²	79	4,008,432 µm²
ExtIntr		130 µm²		78 µm²	0	0 µm²	0	0 µm²
TMP421		79,228 µm²		41,756 µm²	146	10,802,669 µm²	180	9,133,136 µm²
Ex.Apps.: MUXes			50,726 µm²					
Ex.Apps.: Sum/Max			684,141 µm²		232	17,165,885 µm²	331	16,794,822 µm²
All Apps.: MUXes				135,721 µm²				
All Apps.: Sum/Max	4,199,813 µm²			1,950,671 µm²	245	18,127,767 µm²	332	16,845,562 µm²
Factor	1.000		0.163 = 1/6.139	0.464 = 1/2.153		4.316		4.011

Additionally, each example application was synthesized and implemented using Lattice iCEcube2 2014-12 with default settings for the iCE40LP-4k-CM225 and iCE65L04-CB196 FPGAs.[22] For both architectures identical results were obtained. This was expected because the datasheets show identical internal circuits [Sil10, Lat12b]. Therefore both architectures are summarized as "iCE40/iCE65" in Tab. 5.6. The area of an eFPGA with exactly the required resources was calculated with the estimate for the iCE65 device in Tab. 5.3. As the internal logic of the iCE65 and iCE40 architectures seems to be identical, it is assumed that the iCE40 architecture was developed solely as a process shrink from 65 nm to 40 nm. Therefore, scaling the area to the 350 nm process leads to identical values.

For each FPGA architecture, the application with the highest resource utilization and therefore area requirement was selected. The Xilinx 7 Series requires 18.13 mm^2 and the iCE40/iCE65 architecture requires 16.85 mm^2 to implement the largest application. This is 4.32 and 4.01 times more area than the reconfigurable module, respectively (see Fig. 5.19).

These factors show, that the design methodology leads to a reconfigurable module which chip area is *smaller* than an (e)FPGA which can implement its full functionality. Therefore the second sub-hypothesis is provisionally accepted.

Note that perfect packing of LUTs and D-FFs into the higher order logic cells (i.e., CLBs and PLBs) are assumed, as well as perfect place and route results to the available resources without requiring additional logic cells for routing. A real (e)FPGA implementation should provide additional logic cells and therefore require a larger area to accommodate these requirements.

5.5 Configuration Data

To evaluate hypothesis 4 (see Sec. 1.2), the number of configuration bits used in the reconfigurable module is compared to the number of configuration bits required in different FPGA architectures. Therefore each example and new application is synthesized to these FPGA architectures and the resource utilization is multiplied by the number of configuration bits per logic entity. Here again only commercial FPGA chips are considered, because no dedicated eFPGA architecture and synthesis and implementation tools were available. In Sec. 5.5.1 the number of configuration bits per logic entity of FPGA architectures is acquired. The hypothesis is evaluated in Sec. 5.5.2.

5.5.1 FPGA Architectures

The format of the configuration data and the meaning of the individual bits is kept as a trade secret for commercial FPGAs. This prevents to acquire the exact number of configuration bits for the individual logic entities. Therefore here the numbers are estimated. The results of this section are summarized in Tab. 5.7.

The configuration data of Xilinx 7 Series FPGAs (and most other architectures) also contains commands for the configuration infrastructure of the FPGA [Xil15a, p. 76], which are estimated as 1% overhead. To estimate the number of configuration bits per CLB and its associated switch box and routing infrastructure, the configuration data dedicated for all other resources is subtracted from the total size of the configuration data with the following coarsely estimated amounts:

[22] The iCE65 FPGA architecture is not directly accessible with the iCEcube2 GUI, but the synthesis was automated using Bash scripts which supplied the appropriate options and library file names to the tools.

- Block RAM: memory size plus 1,000 bits
- DSP slice: 1,000 bits
- Clock Management Tile (CMT): 1,000 bits
- PCIe block: 1,000 bits
- Gigabit Transceiver (6.6 Gb/s, GTP): 1,000 bits
- Gigabit Transceiver (12.5 Gb/s, GTX): 1,000 bits
- User Configurable Analog Interface (XADC): 1,000 bits
- PS (Processing System): 100,000 bits (including AXI bus interface to the FPGA fabric)
- OCM (On-Chip Memory): memory size (no overhead because it is hard-wired in the processing system)
- IO: 100 bits

The Xilinx Artix-7 XC7A200T FPGA contains 33,650 logic slices (=16,825 CLBs), 740 DSP slices, 365 36 kBit Block RAMs, 10 CMTs, 1 PCIe block, 16 GTPs, 1 XADC, and 500 IOs [Xil15b, p. 2]. Its total configuration data including overhead has a size of 77,845,216 bits [Xil15a, p. 14]. Therefore the configuration data for the 16,825 CLBs is estimated as 62,428,403 bits and 3,710 bits per CLB.

The Xilinx Kintex-7 XC7K70T FPGA contains 10,250 logic slices (=5,125 CLBs), 240 DSP slices, 135 36 kBit Block RAMs, 6 CMTs, 1 PCIe, 8 GTXs, 1 XADC, and 300 IOs [Xil15b, p. 3]. Its total configuration data including overhead has a size of 24,090,592 bits [Xil15a, p. 14]. Therefore the configuration data for the 5,125 CLBs is estimated as 18,452,046 bits and 3,600 bits per CLB.

The Xilinx Kintex-7 XC7K160T FPGA contains 25,350 logic slices (=12,675 CLBs), 600 DSP slices, 325 36 kBit Block RAMs, 8 CMTs, 1 PCIe, 8 GTXs, 1 XADC, 400 IOs [Xil15b, p. 3]. Its total configuration data including overhead has a size of 53,540,576 bits [Xil15a, p. 14]. Therefore the configuration data for the 12,675 CLBs is estimated as 40,041,370 bits and 3,159 bits per CLB.

The Xilinx Zynq-7 XC7Z020 SoC FPGA contains 53,200 6-input LUTs and 106,400 D-FFs (=6,650 CLBs), 220 DSP slices, 140 36 kBit Block RAMs, 1 PS with 256 kB OCM, and 200 IOs [Xil14d]. The configuration file generated with Vivado v2014.2 has a size of 4,045,649 bytes = 32,365,192 bits. Therefore the configuration data for the 6,650 CLBs is estimated as 24,303,428 bits and 3,655 bits per CLB. The estimated values for the four Xilinx 7 Series devices vary from 3,159–3,710 bits. Therefore in the following comparison, the mean value 3,531 is used.

The configuration of the Xilinx Virtex 4 FPGA architecture is organized in frames with a constant length of 1,312 bits. The XC4VFX20 FPGA is configured with 5,488 configuration frames, i.e., with 7,200,256 bits configuration data [Xil09, p. 87f]. The configuration file generated with Xilinx ISE 13.4 has a size of 905,418 Bytes = 7,243,344 bits. Hence, it contains an overhead of 0.595%. The value of 1% assumed above for the Xilinx 7 Series is reasonable similar.

The XC4VFX20 FPGA contains 8,544 logic slices, 32 XtremeDSP slices, 68 18 kBit block RAMs, 1 PowerPC processor block, 2 Ethernet MACs, 8 RocketIO Blocks, and 320 IOs. Each CLB is built of four slices, which each contain two 4-input LUTs and D-FFs [Xil07]. To estimate the configuration data per CLB, again 1,000 bits per block and 100 bits per IO for the routing connections are estimated. For the PowerPC processor block and its bus interface to the FPGA fabric, 50,000 bits are estimated. This results in a total of 5,754,880 bits for the CLBs, and 2,694 bits per CLB. The resource utilization report generated with Xilinx ISE does not allow to use the number of LUT+D-FF combinations, therefore a slice, which contains two of these, is used as smallest entity.

Project IceStorm[23] aims to document the bitstream format of Lattice/SiliconBlue iCE40 FPGAs [Lat12a]. Each logic tile, which is equivalent to a programmable logic block (PLB) and contains eight 4-input LUT and D-FF pairs, is configured with 864 bits.[24] Due to the similarity of the iCE40 and iCE65 architectures, it is assumed that this value is also valid for iCE65.

Table 5.7: Size of the configuration for different FPGA architectures.

Architecture	Configuration Size
XC7A200T	3,710 Bits / CLB = 232 Bits / 5-LUT+D-FF
XC7K70T	3,600 Bits / CLB = 225 Bits / 5-LUT+D-FF
XC7K160T	3,159 Bits / CLB = 197 Bits / 5-LUT+D-FF
XC7Z020	3,655 Bits / CLB = 228 Bits / 5-LUT+D-FF
XC4VFX20	2,694 Bits / CLB = 674 Bits / Slice
iCE40	864 Bits / PLB = 107 Bits / 4-LUT+D-FF

5.5.2 Evaluation of the Hypothesis

The configuration data of the reconfigurable module integrated in the WSN SoC is grouped in four configuration registers associated with individual modules:

Reconfigurable Signals	9 bits
Interconnect Module	1,282 bits
TRFSM0	820 bits
TRFSM1	1,778 bits
Total	3,889 bits

The FPGA resource usage values acquired for the evaluation of hypothesis 3 in Sec. 5.4.4 are also used to evaluate hypothesis 4. Additionally the applications were synthesized and implemented for the Xilinx Virtex 4 XC4VFX20 FPGA using Xilinx ISE 13.4 to determine the resource usage. Default settings were used, except that the module hierarchy was flattened to improve optimization results. The total size of the configuration data was calculated with the estimated size of configuration data per FPGA cell of Tab. 5.7. The results are depicted in Tab. 5.8 and Fig. 5.20.

For the Xilinx 7 Series architecture 54,071 bits of configuration data are estimated to implement any of the example and new applications. This is 13.90 times more than the reconfigurable module. The Xilinx Virtex 4 architecture requires 90,930 bits, which is 23.38 times more. The Lattice iCE40 and iCE65 architectures require 35,441 bits, which is 9.11 times more. Note that here again perfect packing of the netlist to the FPGA cells is assumed.

The design methodology leads to a reconfigurable module which requires less configuration data than the evaluated FPGA architectures. Therefore hypothesis 4 is provisionally accepted.

5.6 Qualitative Measures

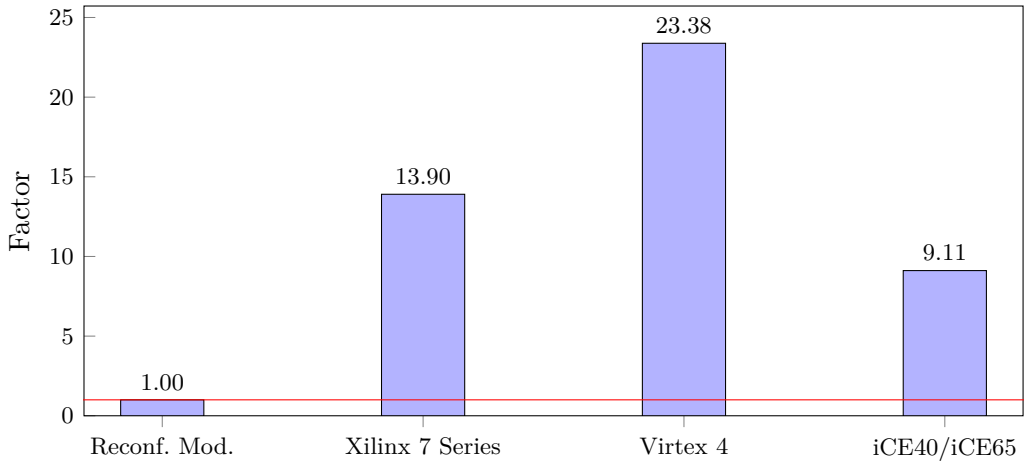
Additional to the evaluation of the four hypotheses, in this section the design methodology is evaluated regarding qualitative properties. In Sec. 5.6.1 the means to ensure correct results are

[23] http://www.clifford.at/icestorm/ [2015-08-20]
[24] This value is exact and no estimate.

Table 5.8: Configuration data size comparison.

Application	Reconf. Mod.	Xilinx 7 Series		Virtex 4		iCE40/iCE65	
	Bits	LUT+D-FFs	Bits	Slices	Bits	LUT+D-FFs	Bits
ADT7310		211	46,567	115	77,459	276	29,463
MAX6682		145	32,001	101	68,029	205	21,884
MAX6682Mean		231	50,981	95	63,988	283	30,210
ADT7410		206	45,464	98	66,009	247	26,367
SlowADT7410		232	51,202	133	89,583	331	35,334
ExtADC		113	24,939	62	41,761	165	17,614
ADT7310P32S16		200	44,139	119	80,153	295	31,491
ADT7310P16S16		174	38,401	100	67,356	242	25,834
ADT7310P32S32		217	47,891	135	90,930	332	35,441
ADT7310P32LS16		245	54,071	116	78,133	290	30,958
ADT7310P32S16L		166	36,636	111	74,765	296	31,598
ADT7310P32LS16L		197	43,477	108	72,744	291	31,064
ADT7310P16LS16L		160	35,311	93	62,641	244	26,047
ADT7310P16LS32L		187	41,270	100	67,356	281	29,997
ADT7310P32LS32L		227	50,098	116	78,133	329	35,121
ExtADCSimple		69	15,228	34	22,901	66	7,046
blinki		76	16,773	38	25,595	79	8,433
ExtIntr		0	0	0	0	0	0
TMP421		146	32,222	60	40,413	180	19,215
Ex.Apps.: Max		232	51,202	133	89,583	331	35,334
All Apps.: Max	3,889	245	54,071	135	90,930	332	35,441
Factor	1.00		13.90		23.38		9.11

Figure 5.20: Configuration data size comparison (values from Tab. 5.8).



discussed. The implications on productivity and design time as well as the learning curve for the design methodology are reviewed in Sec. 5.6.2. Finally, in Sec. 5.6.3 the flexibility of the generated reconfigurable module is investigated.

5.6.1 Correctness of Results

Correctness of the results is crucial in the development of semiconductors. This was expressed as the requirement for the compliance of the reconfigurable module with its specification (cf. Sec. 1.1.7). To ensure the correctness of the results, verification was taken into account from the beginning of the development of the design methodology (cf. Sec. 3.5).

After the Completion step, the generated reconfigurable module is verified against each example application using simulation and logical equivalence checking (cf. Sec. 4.8.4). Additionally the reconfigurable module and the surrounding SoC are prototype tested using an FPGA.

Besides the verification of the final reconfigurable module, verification is also integrated at every step of the development (cf. Fig. 3.11 on p. 77). This ensures that errors appear early in the design process and can be corrected in short iterations. Note that automatically generated results also have to be verified due to possible bugs or missing features in the employed tools and because user supplied options can inadvertently produce undesired results. The firmware and driver development are supported with HW/SW co-simulation. Early HW/SW co-simulation directly instantiates the example application HDL code, while final HW/SW co-simulation uses the generated reconfigurable module (cf. Sec. 4.4.5).

The use of a HDL for all logic designs (example applications, cells) also enables the utilization of wide-spread and proven commercial and open-source tools for verification during the development. The use of Tcl scripts to specify information and to perform the course of the design flow ensures reproducible and documented results.

During the development of the reconfigurable module for the WSN SoC, each example application was verified with a self-checking testbench using VHDL `assert` statements. Each firmware driver was tested using early HW/SW co-simulation. Additionally, the synthesized design, after FSM extraction, after TR-FSM insertion, and after extraction were verified using the original testbench as well as logical equivalence checking. Finally, the generated reconfigurable module was verified without errors against each example application using simulation, logical equivalence checking, and HW/SW co-simulation.

This shows that the discussed design methodology and its implementation as a design flow lead to a reconfigurable module that matches the example applications, and therefore lead to correct results.

5.6.2 Productivity

To successfully compete in the ever evolving world, the time to market of a new product is crucial. The productivity of the designer who utilizes the design methodology and its implementation as a design flow is a special concern. Therefore the productivity is investigated in terms of the actual design time in relation to the *minimum possible* design time. This is limited by the *essentially manual tasks*, which can not be automated.

In the discussed design methodology, there exist two categories of essentially manual tasks. The first category encompasses all points where information in the most general sense is specified. These are especially the development of the parent module, the specification of the reconfigurable signals, the development of the example applications and of the cells, and the specification of the oversizing rules. The second category comprises tasks which require the specific human intelligence and experience. These include most of the tasks also assigned to the first category and the "Inspection" and "Improvement" steps of the "Application Analysis".

The design flow currently lacks the automation of one task: The designer has to manually create the testbench for the HW/SW co-simulation of each example application, despite the necessary information to automatically create a template with an instantiation of the parent module is available in FlowProc. However, the design methodology discussed in this thesis does not require this task to be manual. Therefore it will be automated in a future version of the design flow.

All other not essentially manual tasks of the design methodology are fully automated.

To further reduce the design time and the workload of the developer, the essentially manual tasks are *assisted* by the design flow.

- During the application of the design flow, a large number of Tcl scripts are used. These are prepared as templates which only require a small amount of customization.
- For the definition of the reconfigurable signals the unused signals of the parent module are automatically determined.
- For the development of applications and cells, templates for the HDL design and for its testbench are generated. The designer can focus on the actual implementation.
- For the parent module, e.g., the chip core, a wrapper module can be generated which instantiates the pad cells.
- The design flow also includes numerous tests and checks for the work of the designer and issues warnings or errors if problems are detected.

The essentially manual tasks mainly comprise tasks exclusively required for the development of the reconfigurable module, e.g., the development of the example applications and of the cells. However, some tasks are part of the development of the complete SoC. Therefore this work is *shared*. These are the development of the parent module and of the firmware for the example applications.

The total design time is the sum of the individual tasks plus the required design *iterations*. The main iterative task of the design methodology is the "Application Analysis" (cf. Fig. 3.5 on p. 65), which can greatly increase the total design time. The number of iterations only depends on the experience of the designer and the complexity of the developed reconfigurable module. The duration of each iteration is optimized by providing full automation of the "Synthesis", "FSM Extraction", and "Cell Extraction" steps. The designer can devote his full attention to the "Inspection" and "Improvement" steps with his experience.

A special type of iterations is caused by mistakes and problems of the design. These iterations are kept as short as possible by the extensive facilities for *verification* (cf. the previous section), including (example) applications, early HW/SW co-simulation of the firmware, cells, the interconnect module, and the reconfigurable module.

Besides the reduction of manual work, an important factor in the application of the discussed design methodology is the *learning curve*, i.e., whether it provides a user-friendly interface and is easy to learn. This is supported by the use of common and wide-spread languages, especially Verilog for HDL designs,[25] and Tcl for scripts. These also enable the use of wide-spread tools, e.g., for verification, and interoperability with in-house design flows or custom tools. Tcl scripts additionally provide a user-friendly interface to specify information, to automate tasks, and to use comments to explain procedures and to justify decisions.

The tools Yosys and InterSynth provide extensive *documentation*. The tools FlowCmd, FlowProc, and TrfsmGen are documented with elaborate comments in the scripts and the source code, command line help, and with this thesis.

This analysis indicates that the design methodology provides an environment which enables high productivity.

5.6.3 Flexibility

The design methodology discussed in this thesis should lead to reconfigurable modules, which provide additional flexibility in the post-silicon design phase to implement new applications which were not anticipated before the production. To investigate the flexibility of the WSN SoC, new applications were implemented and verified.

A simplified version of the "`ExtADC`" example application without post-processing of the sensor measurements was designed (see Fig. 5.1 on p. 124 for its schematic). Since this is only a reduction in the required resources, a completely different application which blinks an LED with a parameterizable frequency was developed. To overcome the limitation of missing single-bit D-FF and inverter cells, it was implemented using a small FSM.

Additionally a new application within the application class of the reconfigurable module was developed for the Texas Instruments TMP421 I^2C temperature sensor. It is manufactured by a different vendor as the sensor chips used by the example applications and it differs from these chips because it provides two sensors, an internal and an external one. The application periodically queries and stores both sensor values and subsequently notifies the CPU. Logical equivalence checking was used to verify the correctness of the results.

As mentioned in Sec. 5.3.4 and discussed more detailed in Sec. A.1.3, a design bug of the WSN SoC makes the use of the timer to wake-up from LPM3 impossible, because its clock signal is turned off in that low-power mode. The external clock inputs INClk and TAClk for the timer are synchronized with that clock signal, therefore these also can not be used. Additionally, all GPIO inputs are synchronized with the that clock signal. Hence, the OpenMSP430 CPU also can not be activated with an external interrupt.

In contrast, the interrupt signals from the reconfigurable module do not require synchronization and therefore can be used to activate the CPU from LPM3. This bug was not discovered before production, because no firmware implementation of the sensor interface task was developed at that time and all tests with LPM3 were conducted using the reconfigurable module. The problem was

[25] This is only limited due to the missing support of VHDL by Yosys. All HDL designs which are not synthesized with Yosys, e.g., testbenches, the generated reconfigurable module, and the surrounding SoC, can also be developed using VHDL.

solved by the implementation of a new application for the reconfigurable module which directly connects an external input of the reconfigurable module to the internal interrupt signal.

Besides this bug in the WSN SoC core design, two design issues in the ADT7310 example application were discovered during the measurements. For a detailed discussion of the findings please see Sec. A.2. The ADT7310 example application uses counters which are not suitable for the period delay t_P and sensor delay t_S required in the measurement procedure. Additionally, when the counters were stopped, they were controlled to permanently preset the start value, which caused unnecessary energy consumption. These bugs were not found because in the pre-silicon design phase these internal signals were not inspected, and no power analysis was conducted. These problems were solved by the implementation of new applications with the proper counter size (32 bits vs. 16 bits) and control signals generated by the FSMs. The stopped counters reduced the current consumption from 76.39 µA ("P32 S16") to 70.14 µA ("P32L S16L"), i.e., by 8.18% (see Tab. A.9 and Fig. A.6 on p. 195) for 200 meas./s at 1 MHz operating frequency.

This shows that the design methodology leads to a reconfigurable module with flexibility in the post-silicon design phase. It allows to implement new applications within the defined application class, to resolve design issues, and to some extent even applications with unrelated functionality.

However, the flexibility is limited. The 16-bit and the 32-bit counter cells connect the counter value to the output "`D_o`" (cf. Lst. 4.6 on p. 107) and therefore to the interconnect. This causes high switching activity and energy consumption in the interconnect, even if the signal is not routed to another cell. This problem was not found in the pre-silicon design phase for the same reasons the permanent presetting was missed, and because no low-power-audit for undesired switching activity was performed. Due to the fixed design of the cells, a redesign of the reconfigurable module is required to solve this problem.

5.7 Requirements

In Sec. 1.1.7 a set of requirements for the design methodology was defined. In this section the design methodology is checked to comply to these requirements.

- The design methodology must be independent of the wide and narrow application domain. This is best shown by the implementation of two or more reconfigurable modules for diverse application domains. Unfortunately limited resources do not allow this approach. Therefore an analysis of the design methodology and the design flow was performed to identify assumptions or limitations for the application domain. This did not reveal any restrictions, hence the requirement is fulfilled.

- The specification in terms of possibilities and accomplishable functionality is realized using example applications (cf. Sec. 3.1.1).

- The inclusion of additional flexibility is enabled with increasing the size of sizable cells, increasing the number of instances of cell types, and increasing the routing resources (cf. Sec. 3.3.1).

- The unified multi-granular reconfigurable architecture is implemented using different connection types. Additionally, cells can offer ports with a mixture of connection types.

- The design methodology provides extensive facilities for verification, including means to prove the compliance of the generated reconfigurable module to its specification. This was discussed in detail in Sec. 5.6.1.
- The design methodology enables high productivity by the implementation of a fully automated design flow. Only essentially manual tasks have to be performed by the designer. This was discussed in detail in Sec. 5.6.2.
- The design flow provides a user-friendly interface by using a Unix style command line interface and Tcl as scripting language. This ensures fast turn-around times, documented and reproducible results, and easy customization. It is easy to learn and only requires skills readily available among HDL designers. This was discussed in detail in Sec. 5.6.2.
- The design flow provides short iteration times by the automation of all tasks not essentially manual and by early verification of all manual and automated results. This was discussed in detail in Sec. 5.6.2.
- The design methodology and the generated reconfigurable module are independent of the semiconductor process. No assumptions on the production are included and the reconfigurable module is delivered as a soft IP core with fully synchronous plain HDL RTL designs. Apart from the special handling of combinational loops in the interconnect, the synthesis is straight forward. No special standard cells are required.
- The design flow is compatible to commercial ASIC tools as demonstrated with the implementation and production of the WSN SoC. This process included the tools Mentor Graphics Questa Sim, Cadence LEC, Synopsys Design Compiler, Synopsys Formality, Synopsys PrimeTime, Cadence Encounter, Cadence Virtuoso, and Mentor Graphics Calibre. Finally, a chip was produced using a commercial semiconductor process.

  The directory structure used by the design flow is stand-alone and enables the development of a complete SoC. Therefore it is in conflict with in-house ASIC design flows. However, the HDL designs of the cells and the generated HDL design of the interconnect module and the reconfigurable module can be easily copied or imported into an in-house design flow. Additionally, FlowCmd and all Tcl scripts can be adapted to and integrated in an in-house design flow.

5.8 TR-FSM

In this section the reconfigurable TR-FSM architecture (cf. Sec. 3.7) is evaluated. The results published in [GDHG10], which are based on the LGSynth93 FSM benchmark suite [McE93], and the results published in [GDHG11], which are based on the manually developed reconfigurable module (cf. Sec. 5.2), are summarized. Additionally new results based on the WSN SoC are introduced.

In Sec. 5.8.1 the power consumption is investigated. In Sec. 5.8.2 the chip area is compared to implementations using FPGAs, SRAMs, and non-reconfigurable synthesized FSMs. The size of the configuration data is compared to FPGA implementations in Sec. 5.8.3. In Sec. 5.8.4 the propagation delay time is compared to FPGA implementations and to non-reconfigurable synthesized FSMs. Finally, in Sec. 5.8.5 the results are summarized.

5.8.1 Power Consumption

For the evaluation of the power consumption of the TR-FSM architecture, the three different silicon implementations in the manually developed reconfigurable module (cf. Sec. 5.2) were supplied with an alternating pattern of all-zeros and all-ones [GDHG11]. The three instances consumed 2.53 µA/MHz/V, 7.92 µA/MHz/V, and 15.3 µA/MHz/V, respectively. This was compared to the active power consumption of the lowest-power MCU investigated in [GHDG09] (ATmega88PA) with 100 µA/MHz/V. The results show that the TR-FSM silicon implementations requires 6.6–39.5 times less power compared to the MCU, depending on their size. Note that the TR-FSM directly implements a given function while an MCU requires a larger number of clock cycles and thus causes a much larger difference in the total *energy* consumption.

5.8.2 Chip Area

In this section the chip area of the TR-FSM architecture is compared to reconfigurable FSM implementations using FPGAs and SRAMs, and to non-reconfigurable directly synthesized FSMs.

Comparison to FPGA

The chip area of the three different silicon implementations of the TR-FSM in the manually developed reconfigurable module (cf. Sec. 5.2) were compared to the area of an equivalent FPGA implementation [GDHG11]. From the LGSynth93 suite [McE93] those FSMs were selected which could be implemented by the TR-FSMs in accordance to the number of inputs, outputs, states, and transitions. For each TR-FSM the FSM which required the most FPGA resources as determined by [Buk08] was selected and used to estimate the chip area of an FPGA with the exact amount of resources and implemented in the same semiconductor process. The chip area of the three FPGA implementations was estimated as 98,000 µm^2, 371,000 µm^2, and 574,000 µm^2, respectively, for a 130 nm CMOS process. The three TR-FSM implementations required a chip area of 29,900 µm^2, 94,700 µm^2, and 187,800 µm^2, respectively. The results show that the total area of each TR-FSM implementation is only 25.5–32.7% compared to an FPGA implementation of the FSMs.

Comparison to SRAM

To compare the area requirement of the TR-FSM architecture with an alternative implementation using SRAMs, the according sizes are calculated. TRFSM0 of the WSN SoC provides six inputs, a state vector with five bits, and ten outputs (cf. Tab. 5.1 on p. 123). An equivalent SRAM therefore requires eleven address inputs and a word width of 15 bits, i.e., a total of $15 \cdot 2^{11} = 30,720$ bits. The TRFSM1 instance provides ten inputs, a state vector with six bits, and 15 outputs. The size of an equivalent SRAM is therefore $(15 + 6) \cdot 2^{10+6} = 21 \cdot 2^{16} = 1,376,256$ bits. However, the number of transitions which can be implemented with the TR-FSM instances is limited by the number of the included transition rows (30 and 46, respectively). The SRAM implementations can implement all possible transitions, i.e., $2^{n_\mathrm{I}+n_\mathrm{S}}$.

To determine the chip area of these SRAMs, a memory generator to create the actual cells is required. Unfortunately this was not available, therefore the size is estimated by scaling the size of the memory cells used as program and data memory. The program memory is a size optimized

Table 5.9: Area comparison of the two TR-FSM instances with an alternative implementation using SRAMs.

	Area	**2k×8: 0.536 mm² 32.71 µm²/bit**		**8k×8: 3.571 mm² 54.49 µm²/bit**		**128×8: 0.157 mm² 153.75 µm²/bit**	
TRFSM0	0.39 mm²	1.00 mm²	×2.58	1.67 mm²	×4.30	4.72 mm²	×12.14
TRFSM1	0.88 mm²	45.02 mm²	×51.25	74.99 mm²	×85.37	211.60 mm²	×240.88

2 kB SRAM with a size of 0.536 mm². The data memory is a speed optimized 128 byte SRAM with a size of 0.157 mm². Additionally, an SRAM cell of a different project realized with the same semiconductor process is used. It is a speed optimized 8 kB SRAM with a size of 3.571 mm².

From these sizes the area per bit is calculated and multiplied by the required number of bits to replace the TR-FSMs. The result is shown in Tab. 5.9 and compared to the TR-FSM instances. For the most area efficient SRAM architecture, the required size would be 2.58 times and 51.25 times to replace the two TR-FSM instances. This shows that the TR-FSM is an area-efficient solution for the implementation of reconfigurable FSMs. Note that the TR-FSM area values do not contain filler cells and other overhead, therefore the factors are slightly lower for an actual chip implementation.

Comparison to Non-Reconfigurable Synthesized FSMs

For a comparison of the reconfigurable TR-FSM architecture to non-reconfigurable synthesized FSMs, a number of random FSMs were generated.[26] Two different sets of random FSMs were generated with the parameters chosen to match and fully utilize the two TR-FSM instances included in the WSN SoC (cf. Tab. 5.1 on p. 123). These FSMs were mapped to the TR-FSM instances using TrfsmGen. This eliminated a large part of the generated FSMs which could not be implemented with the TR-FSM instances, e.g., if too many wide TRs were required. At the end a total of 115 FSMs to compare to the TRFSM0 instance, and 121 (different) FSMs to compare to the TRFSM1 instance (see Tab. 5.10) were arranged.

A new function was added to TrfsmGen which exports an FSM as an ILang file for Yosys using its `$fsm` generic FSM cell (cf. Sec. 4.1.2). Each of the 115 and 121 random FSMs were exported as ILang files and then implemented as Verilog RTL FSM designs with the Yosys `fsm_map` command. Finally each Verilog RTL FSM design was synthesized to the same semiconductor process as the WSN SoC using Synopsys Design Compiler.

The mean and standard deviation of the chip area of the random FSMs are shown in the fourth column of Tab. 5.10.[27] The results show a large area penalty of the TR-FSM instances compared to non-reconfigurable synthesized FSMs by a factor of 31.12 and 43.21. However, the TR-FSM is reconfigurable and can implement a large number of different FSMs. Therefore in the sixth column of Tab. 5.10 the sums of the area of all random FSMs are shown. Although these exemplary random FSMs only cover a fraction of the total functionality of the TR-FSM instances, they would require 3.58 and 2.80 more area than the TR-FSM instances.

[26] http://ddd.fit.cvut.cz/prj/Circ_Gen/index.php?page=kiss [2015-09-20]

[27] The area values are not scaled to a given area utilization as in Sec. 5.4.4, because the TR-FSM area values are also given without overhead.

Table 5.10: Area comparison of the two TR-FSM instances in the WSN SoC with non-reconfigurable synthesized FSMs.

		Non-Reconfigurable FSMs				
	Area	**Num.**	**Area**	**Factor**	**$\sum$ Area**	**Factor**
TRFSM0	0.39 mm^2	115	12,115±600 µm^2	÷31.12	1.39 mm^2	×3.58
TRFSM1	0.88 mm^2	121	20,328±818 µm^2	÷43.21	2.46 mm^2	×2.80

5.8.3 Configuration Data

To evaluate the size of the configuration data, 19 FSMs of the LGSynth93 suite were selected and assigned to three groups depending on the similarity of their number of inputs, outputs, states, and transitions [GDHG10]. Note that these three groups are different from the TR-FSM implementations used previously for the comparison of the power consumption and chip area. For each group the smallest possible TR-FSM and the size of its configuration data were determined. In addition, for each of the 19 FSMs the required FPGA resources as determined by [Buk08] were used to find the biggest FSM in each group. This was used to calculate the size of the FPGA configuration data. The largest FSM in each group required 145,152, 102,816, and 61,344 bits configuration data, respectively, for the FPGA implementation. The three TR-FSM implementations required 17,058, 13,150, and 4,507 bits. The results show that the TR-FSM requires only 7.3–12.7% configuration data compared to an FSM implementation using an FPGA, i.e., 7.82–13.61 times less.

5.8.4 Propagation Delay

In this section the propagation delay time of the TR-FSM architecture is compared to reconfigurable FSM implementations using FPGAs and to non-reconfigurable directly synthesized FSMs.

Comparison to FPGA

To compare the signal propagation delay, static timing analysis was used for the three TR-FSM silicon implementations of the manually developed reconfigurable module [GDHG11]. These introduce a delay of 6.83 ns, 8.28 ns, and 9.87 ns, respectively. The values were compared to the propagation delay of a 4-input LUT (as typically employed in FPGAs) manufactured in the same semiconductor process using standard cells. A 4-input LUT introduces a delay of 2.94 ns–3.27 ns, depending on the surrounding circuit. The results show that the total TR-FSM propagation delay is 2.2–3.2 times the delay of a single 4-input LUT. While FPGAs use an optimized full-custom design which is faster than the LUTs implemented using standard cells, the flexible routing resources introduce considerable delay. The logical depth of a typical FSM input to output path implemented in an FPGA uses two to three LUTs. Therefore the propagation delay time of a TR-FSM is comparable or even faster than an FPGA implementation.

Comparison to Non-Reconfigurable Synthesized FSMs

The propagation delay time of the TR-FSM instances of the WSN SoC as well as of the random non-reconfigurable synthesized FSMs as described in Sec. 5.8.2 are summarized in Tab. 5.11. The

Table 5.11: Comparison of the propagation delay time of the two TR-FSM instances in the WSN SoC with non-reconfigurable synthesized FSMs.

		Non-Reconfigurable FSMs		
	Delay	**Num.**	**Delay**	**Factor**
TRFSM0	10.59 ns	115	5.28±0.81 ns	÷2.00
TRFSM1	14.15 ns	121	6.75±0.67 ns	÷2.10

results show that the TR-FSM instances cause a longer signal delay than the directly synthesized FSMs by a factor of 2.00 and 2.10.

5.8.5 Summary

In this section the reconfigurable TR-FSM architecture was evaluated. The power consumption with maximum input switching activity is 6.6–39.5 times lower than an ultra-low-power MCU. The chip area is only 25.5–32.7% compared to an FPGA implementation. In comparison to an SRAM, the area is 2.58 and 51.25 times smaller for the two instances of the WSN SoC. However, directly synthesized non-reconfigurable FSMs require 31.12 and 43.21 times less area than the TR-FSM.

The configuration data of the TR-FSM architecture is 7.82–13.61 times smaller than the configuration data of an FPGA with exactly the required resources. The signal delay of both architectures is approximately the same. In comparison to directly synthesized non-reconfigurable FSMs, the TR-FSM causes 2.00 and 2.10 times more delay.

5.9 Discussion

The introduced design methodology enabled the development of an exemplary WSN SoC with a reconfigurable module for the application class of a sensor interface. Its functionality was verified during the development using simulation and logical equivalence checking, using an FPGA, and with a test chip. All verification procedures and tests with the manufactured WSN SoC showed proper operation, which proves the feasibility of the design methodology.

In Sec. 5.3.4 three different comparisons between an implementation of the sensor interface task using a microcontroller and using a reconfigurable module showed considerable energy reduction. The most reliable comparison using the WSN SoC, which is not confounded by a difference of the semiconductor process and supply voltage, shows a reduction of the energy consumption per sensor measurement by a factor of nearly 180. This factor describes only the sensor measurement itself. The factor of energy reduction is lower for the complete operation of the WSN node, because of other activities.

The comparison of the chip area of the reconfigurable module to the parallel implementation of all example applications leads to an increase by a factor of 6.1 (see Sec. 5.4.4). Additionally, the reconfigurable module provides flexibility to implement new applications. In comparison to the parallel implementation of all applications developed to date, the reconfigurable module requires

2.2 times more chip area. However, even more new applications can be developed than this set of applications, which would further reduce the factor.

A comparison of the reconfigurable module to (embedded) FPGAs shows that the latter require 4.0 to 4.3 times more chip area. These values assume perfect packing and P&R results, hence a real eFPGA should provide additional resources and therefore require more area. Further, the FPGA architectures used in the comparison are implemented as full custom designs. A soft core eFPGA using standard cells, which is independent of the semiconductor process, would require more area than a full custom eFPGA. [WAWS03] reported a factor of 6.4 (cf. Sec. 2.3.3). Additionally, eFPGAs increase the power consumption and cause licensing cost.

The size of the configuration data of the reconfigurable module was compared to commercial FPGA architectures in Sec. 5.5.2. These require 9.1 to 23.4 times more configuration data. Since the configuration data has to be stored twice (in non-volatile memory and in the configuration stores) and it has to be loaded at startup, the reduced configuration data results in reduced chip area, reduced flash memory, and reduced loading time.

Verification is integrated in the design methodology from the start of the development with module-testing of each example application, using simulation and logical equivalence checking for the generated intermediate netlists and the reconfigurable module, up to the final post-P&R chip netlist (cf. Sec. 5.6.1). Only two similar testbenches for each example application are required throughout the design process. This shows that the design methodology provides a reliable development process.

The productivity of the designer is supported by the automated design flow (cf. Sec. 5.6.2). The designer is relieved from error-prone and tiring tasks. Manual work is only required where the specific human intelligence and experience of the designer are required. Throughout the design flow, known and wide-spread languages (Verilog, VHDL, Tcl) reduce the learning effort and enable the use of in-house flows and proven tools.

The design methodology leads to a reconfigurable module which provides enough flexibility to implement new applications (cf. Sec. 5.6.3). These can implement new functionality within the application class. Additionally, design issues of the pre-silicon design phase can be resolved in the post-silicon design phase without the need for the production of a new chip. The creative use of the available resources even extends the range of the application class which was shown with the LED blinking application. However, the flexibility is limited as was revealed by the high switching activity caused by the counter cells used to implement delays.

For the development of the design methodology discussed in this thesis, several design decisions were made. In Sec. 3.1.1 the use of example applications as format for the specification was selected. If a specification in terms of the available resources or in abstract terms was selected, the development of example applications could be removed. This would considerably reduce the design time. On the other hand, the reconfigurable module could not be verified to provide the necessary resources or to comply to the specification.

Additionally, DFGs were declined and existing HDLs were selected to represent the example applications. If DFGs were selected, only data processing applications without the ability to control peripherals were possible. If a custom HDL was used for the example applications, the designers would need to learn this new language and additional tools for the processing and verification were required.

In Sec. 3.1.5 the automatic identification of cells was declined in favor of manual development of cells. If the cells were automatically identified as implemented by the SPS project [OMBKS01]

(cf. Sec. 2.5.3), the "Application Analysis" could be fully automated. However, the individual functionalities of each cell would be arbitrary sections of the example applications. This would cause severe difficulty for the implementation of new applications and therefore degrade the flexibility of the reconfigurable module.

In Sec. 3.2.1 the decision to utilize specific FSM cells was discussed and in Sec. 3.7 the TR-FSM as a dedicated architecture for reconfigurable FSMs was introduced. If control-dominated tasks were implemented using fine grained logic cells, a large number of small cells would be required. This would have simplified the implementation of the blinking LED application, but on the other hand would have caused a large increase in the number of ports and configuration data and therefore chip area of the interconnect. The evaluation of the TR-FSM architecture in Sec. 5.8 shows considerable advantages in terms of power consumption, chip area, and the size of the configuration data, compared to reconfigurable FSM implementations with FPGAs and SRAMs. The propagation delay is comparable to an FPGA implementation. However, the chip area of a directly synthesized non-reconfigurable FSM is considerably smaller. The propagation delay differs only by a factor of approximately 2. These advantages directly result from the implementation of a dedicated architecture for FSMs, which avoids the large overhead that pertains to general purpose logic and memory architectures. However, the chip area, delay, and power consumption of a TR-FSM is inferior to an FSM directly synthesized to a non-reconfigurable gate netlist with standard cells.

A comparison of the design methodology with commercial embedded FPGA architectures (cf. Sec. 2.3.2) shows that these require less work in the pre-silicon design phase. The required resources can be estimated from previous projects or from a single "example" application. The vendor tools also support wide-spread languages and verification. The eFPGA is delivered as IP core with extensive documentation and support for the integration in the SoC. However, eFPGAs cause a higher power consumption, chip area, and require considerably more configuration data. Using a hard IP core is not independent of the semiconductor process, while a soft IP core further increases the chip area. Additionally, license fees increase the cost of the product.

The KressArray Family [Har01a, HKR94] (cf. Sec. 2.5.1) analyzes the results of experimental mappings and automatically generates suggestions for the improvement of the reconfigurable architecture. The design methodology discussed in this thesis does not provide suggestions but only relies on the experience of the developer. However, for the optimization of the KressArray Family architecture only a single example application is used. Example applications are developed with a custom high-level language. The architecture only supports data processing and offers no mixed granularity cells or control-dominated tasks.

The Berkeley Pleiades Project [AZW+02, AR96, Rab97, RAI+97, ASI+98] (cf. Sec. 2.5.2) also provides suggestions to iteratively improve the architecture. These are derived by power estimation of the implementation. The design methodology discussed in this thesis does not analyze an intermediate solution and does not offer a metric for the optimization, e.g., power consumption or chip area. The Berkeley Pleiades Project uses all example applications in parallel, but also uses a custom high-level language and only supports data processing tasks.

The Totem RaPiD Project [Hau05, HCE+06] (cf. Sec. 2.5.4) requires RaPiD netlists which instantiate the cells implemented in the reconfigurable architecture, i.e., there is no cell library optimization. These RaPiD netlists are used to optimize a 1D cell arrangement and interconnect. This architecture is also limited to data processing.

Comparing the design methodology to the manual development of a reconfigurable module (cf. Sec. 5.2) shows a significant improvement in the productivity. The extensive facilities for verification ensure a reliable methodology. The automatic selection of the cells for the interconnect module and the automatic generation of the routing resources ensure sufficient flexibility without expendable overhead.

The design methodology discussed in this thesis leads to functional reconfigurable modules which comply to the specification. It is universally applicable for any application domain and not limited to low-power or WSN applications. The mixed granularity reconfigurable architecture supports data processing as well as control-dominated tasks. The resulting reconfigurable modules provide a reduction of the power consumption compared to a firmware implementation, but require larger chip area than the parallel implementation of all example applications. In comparison to eFPGAs, the chip area and the size of the configuration data is reduced. The design methodology provides extensive facilities for verification. The use of wide-spread languages and the automation of all tasks which do not require the specific human experience lead to high productivity. The resulting reconfigurable modules provide enough flexibility to implement new applications.

6

Conclusion and Future Work

In this thesis a novel design methodology for reconfigurable CPU supplement modules was introduced. The work is summarized in Sec. 6.1. In Sec. 6.2 the implications and possibilities of the new design methodology are discussed. The challenges which occurred during the course of this work are briefly discussed in Sec. 6.3. The present work includes limitations and open issues, which are discussed in Sec. 6.4 with proposed solutions. Finally, in Sec. 6.5 future work to continue the findings of this thesis as well as further research stimulated and enabled by this work are given.

6.1 Summary

In the field of wireless sensor networks (WSNs), low power consumption of a node is a crucial design goal. In this thesis, the inclusion of CPU supplement modules as a technique for power reduction is proposed. The modules autonomously handle tasks like sensor measurements or the network protocol while the CPU stays in an inactive low-power mode for extended periods. These CPU supplement modules must be reconfigurable to adapt to different environments. To further reduce the power consumption, these reconfigurable modules must be tailored to the domain of its application scenario and use mixed-grained reconfigurable logic.

Therefore, a design methodology for application domain specific mixed-grained reconfigurable CPU supplement modules is required, which is the goal of this thesis. In Sec. 1.2 four hypotheses were postulated:

- The design methodology must be feasible for the development of reconfigurable module.
- It must lead to reconfigurable modules, which reduce the energy consumption compared to a CPU-only implementation.
- It must lead to reconfigurable modules which require less chip area than the parallel implementation of all example applications and less area than fine-grained embedded FPGA implementations.
- It must lead to reconfigurable modules which require less configuration data than embedded FPGAs.

The design methodology was developed using a scientific method. It is based on prior research and includes original contributions. The implementation as a design flow was used to develop an exemplary reconfigurable module, which was investigated to evaluate the hypotheses.

Previous research on application domain specific reconfigurable logic (cf. Ch. 2) focused on data processing tasks and did not include control-dominated tasks. No reconfigurable architecture for mixed granularity and no design methodology for the combined tasks, which is independent of the application domain and provides verification of the results, were available. Further, no suitable reconfigurable architecture for FSMs to implement cycle accurate control-dominated tasks was available.

To remedy these shortcomings, a novel design methodology was developed in this thesis (cf. Ch. 3). The functionality of reconfigurable modules is specified with a set of example applications as logic designs developed in an HDL like VHDL or Verilog. The reconfigurable module is delivered as an IP core for the integration in an SoC. The reconfigurable architecture is defined as a pool of cells which are connected via an interconnect. It supports mixed-grained logic by separate interconnects for different connection types and allows cells with ports of different connection types. The concept of parameterization was introduced to separate the operating data from the configuration.

The development of a reconfigurable module is performed in two steps: “Application Analysis” and “Merge”. The “Application Analysis” is an iterative process with automated and manual tasks to develop and optimize the cell library. First the example applications are synthesized to netlists and the FSMs and the cells are extracted. In the following manual “Inspection” and

"Improvement" steps, the designer identifies residual logic, designs new cells, and improves the example applications and the existing cells. Then the next iteration is started. To support the extraction of cells from the example applications, the concepts of topological variants and reduced variants of cells were introduced.

When all example applications can be implemented by only cells of the cell library, these netlists are merged to the reconfigurable architecture. This determines the minimum number of instances of cells required, maps all example applications to these cells, and optimizes an interconnect with tree topology. To increase the flexibility of the reconfigurable module, additional cells and increased routing resources in the interconnect can be included ("oversizing").

The design methodology includes the verification of the initial example applications and of the generated results at every intermediate step. For simulation, only a single testbench for all steps is required. Additionally, logic equivalence checking is used. The design methodology also provides HW/SW co-simulation to test the interactions between the reconfigurable module and the firmware driver. The extensive features for verification ensure that the generated reconfigurable module is correct. While the generation of the reconfigurable module is denoted the pre-silicon design phase, the post-silicon design phase is analogous to the use of FPGAs. The design methodology supports the implementation of new applications, which were not anticipated in the pre-silicon design phase.

Additional to the design methodology, a reconfigurable architecture for FSMs termed TR-FSM was introduced. It focuses on the transitions instead of the next state and output functions, and provides a reduction of the chip area, configuration data, power consumption, and delay.

The design methodology was implemented as a design flow (cf. Ch. 4). This provides a user-friendly environment for the development of reconfigurable modules. It integrates custom, open-source, and commercial tools and is fully automated using Tcl scripts, which ensure documented and reproducible results.

To evaluate the design methodology and its implementation as a design flow, an exemplary WSN SoC with a reconfigurable module was developed, manufactured, and the hypotheses were evaluated.

- The results show, that the design methodology is feasible for the development of reconfigurable modules.
- The energy consumption of the reconfigurable module to perform a sensor measurement is nearly 180 times lower than an implementation using solely the CPU.
- Its chip area is 2.2 times larger than the parallel implementation of the example applications and the new applications, but it requires 4.0–4.3 times less area than embedded FPGAs.
- The size of the configuration data is 9.1–22.4 times smaller than for embedded FPGAs.

The TR-FSM requires 2.6–51.3 times less chip area than an implementation using SRAM cells. The design methodology facilitates high productivity of the developer by the automation of all non-essentially manual tasks.

6.2 Impact

The discussed design methodology enables the extension of WSN nodes with reconfigurable CPU supplement modules to reduce the power consumption. The results achieved with the design methodology are verified, i.e., the integration of reconfigurable modules is a secure design decision which does not cause additional risks. This is emphasized by the successful development, production, and evaluation of the WSN SoC discussed in Sec. 5.1.

The design methodology is universal and independent of the application domain. Besides a sensor interface, other tasks can be implemented which are neither related to WSNs nor to low power consumption, e.g., network protocol handlers, CPU accelerators, digital filters, control loops, computer vision preprocessors, etc. The development of a reconfigurable module and its integration into a chip design is straightforward and uncomplex. Therefore it is also suitable for small designs, wherever post-silicon flexibility is required, for example to avoid the use of an embedded or external FPGA, or within an FPGA if no partial reconfiguration is available or desired.

The use of free and open-source tools and common EDA tools does not require the purchase of additional tools or license fees. Although the motivation for this work was the development of CPU supplement modules, the reconfigurable modules can also be employed to supplement other logic designs or as dedicated semiconductor product. Additionally, the TR-FSM is itself an independent design unit which can be integrated if an efficient reconfigurable FSM implementation is required.

6.3 Challenges

For the development of the design methodology and its implementation as a design flow, a wide spectrum of different tasks had to be preformed:

- digital logic design using VHDL and Verilog
- verification using simulation and logical equivalence checking
- implementation of logic designs targeting FPGAs
- ASIC design flow and the according tools for synthesis, place and route, sign-off verification, and tapeout
- programming using different languages (Pascal, C, C++, Tcl, Bash, Makefiles)
- integration of a scripting language in programs
- automatic generation of VHDL, Verilog, and Tcl
- development of MCU firmware, including communication via USB and the according host software drivers
- PCB design and manufacturing for analog and digital circuits
- remote-control of measurement instruments via GPIB, Ethernet, RS232, and USB
- analysis of the results using Matlab
- automatic generation of LaTeX code for Ti*k*Z, pgfplots, and PgfplotsTable

6.4 Open Issues

The work provided with this thesis has certain limitations. However, these only have a small impact on the usability and did not interfere with the development of the WSN SoC:

- The design flow does not implement all features of the design methodology (see Sec. 4.1.4). These limitations are due to time constraints and are not principle problems.
- The design methodology, the design flow, and the evaluation do not consider the propagation delay time of the reconfigurable module. This is most likely inferior to the parallel implementation of all example applications, but possibly better than for embedded FPGAs due to their extensive routing resources with a high number of switch boxes.
- The feasibility of the design methodology and the design flow were demonstrated with the development of a reconfigurable module for a sensor interface (cf. Sec. 5.1). The findings were generalized from this single application domain. Additionally, in Sec. 5.7 the requirement for independence of the application domain was discussed with an analysis of the design methodology and the design flow. To increase the robustness of these statements, reconfigurable modules for more and different application domains should be implemented.
- In Sec. 5.1.5 considerable routing congestions of the chip layout, especially for the interconnect, were reported. To relax the congestions, two solutions were suggested: placement density constraints and configuration chain reordering. An additional improvement could be achieved by the use of D-FFs without reset. This would considerably reduce the number of buffers and wires of the reset tree. However, it must be ensured that at power-on the random configuration values do not produce undesired results, e.g., by activating combinational loops or a ring oscillator. All three improvements require further investigation.

6.5 Future Work

The discussed design methodology and the design flow were completed and validated. In this section future research on extensions and improvements are proposed:

- In the “Application Analysis” step, the optimization of the example applications and the cell library are performed by the developer. The goal is to map the digital logic of all example applications to cells of the cell library. Future research should investigate means for additional guidance for the optimization using the metrics power consumption, chip area, and delay. This should analyze the current results, estimate its power consumption, chip area, and delay, and provide suggestions for improvements. After the “Application Analysis”, in the “Merge” step, the interconnect is optimized. Further research should investigate the potential of these metrics for improvements of the generation of the interconnect as well as for the mapping of applications.
- In the design methodology, the determination of the oversizing rules for the number of instances and the routing resources of the interconnect module are based on the developers experience. Further research should investigate methods to characterize the flexibility of the resulting circuit using a quantitative approach, e.g., [Com03, CH04]. This should guide

the developer to find a suitable trade-off between limitations in the post-silicon design phase and surplus resources causing increased chip area.

- Another important area for future research is the reduction of the chip area of the interconnect and of the TR-FSMs. This could be achieved by the design of tactical standard cells as implemented by [ALS05]. For example, this could include the use of transmission gates to implement the MUXes of the interconnect and of the TR-FSM input switching matrices (ISMs). However, custom standard cells interfere with the requirement for independence of the semiconductor process and require high effort to characterize. Therefore, research should also take the trade-off between chip area and independence of the semiconductor process into account.

- In Sec. 3.7 further improvements of the TR-FSM were suggested: sum-of-product elements for input pattern gates (IPGs) with many inputs, several possibilities for a reduction of its power consumption, and default next states as implemented by [RV13]. They also directly include counters in the FSM to implement delays.

The design methodology enables and stimulates further research in different areas:

- Following the starting point of this work to reduce the power consumption of WSN nodes, the inclusion of reconfigurable modules enables the design and optimization of new network protocols. It can be based on frequent operations which were previously precluded due to too high power consumption of the CPU.

- In contrast to firmware programs, reconfigurable modules enable fast responses and constant latency to react on incoming information. With this feature, research on rapid and ultra-low-power real-time control loops can be conducted.

- In the research area of software defined radio (SDR), the power consumption of the extensive digital signal processing is a major concern. The implementation of digital filters and higher-level protocol handling using reconfigurable modules can considerably reduce the power consumption. Concurrently this approach conserves the flexibility to changes the filter topology and its coefficients as well as the entire communication protocol. This approach fills the gap between SDR and hard-wired radios and facilitates the implementation of power-efficient and flexible multi-protocol transceivers. For example, this enables research on WSNs, which can adapt to new communication standards with only a firmware upgrade.

- For the development of WSNs, simulation is used to optimize the node hardware, the firmware, and the network protocols. For example, in this research area the PAWiS framework [WGM07, GWMM08], SystemC-based frameworks [HDG+09], or PAWiS based frameworks [MHMS13, Mös15] are used. To include simulation models of reconfigurable modules, the functionality, timing, power consumption, as well as the interaction with the CPU have to be considered. Additionally, a trade-off between detailed models for accurate results and abstract models for higher simulation performance has to be investigated.

Conclusion

In this thesis, a novel design methodology for application domain specific mixed-grained reconfigurable modules was introduced. It is the first methodology to universally support both, control-dominated tasks and data processing, and to include verification of the final and all intermediate results. Additionally, the TR-FSM as a reconfigurable architecture for FSMs was introduced. This work enables the save development and employment of reconfigurable modules. For example, as a CPU supplement module for the reduction of power consumption of WSN nodes, or generally to provide reconfigurable functionality in a semiconductor product. The design methodology facilitates research in new areas which were previously precluded because of power consumption, chip area, or cost.

A

WSN SoC Power Consumption

In Sec. 5.3 the power consumption of the WSN SoC and the reconfigurable module were investigated. In this appendix, more details are given. The power consumption of the MCU part of the WSN SoC is investigated in Sec. A.1. This is followed by the analysis of the power consumption of the reconfigurable module in Sec. A.2.

A.1 Characterization of the WSN SoC

In this section the power consumption of the WSN SoC is characterized. As mentioned in Sec. 5.1.4, two power domains were implemented. The CPU power domain includes the CPU, the RAMs, the peripherals for the CPU and the reconfigurable module, the clock tree root, and the pads. The reset tree root is also contained in the CPU power domain but only contributes to the static leakage current. The reconfigurable module itself is placed in a separate power domain.

A total of 24 chip samples were measured with 101 testcases for the firmware implementation of the sensor interface task and with 163 testcases for the implementation using the reconfigurable module. For each testcase, the mean and standard deviation of the current consumption over the 24 chip samples was calculated. The current consumption of the CPU power domain shows a variation of 9.50% for the testcase to measure the leakage current and below 0.733% for all active testcases. The variation of the reconfigurable module current consumption is below 0.315% for all testcases. Note that these standard deviation values describe the raw current measurements while the standard deviation values given in Sec. 5.3.4 and in Tab. 5.2 on p. 147 describe the variation of the calculated regression coefficients and residuals.

Due to the small variation, the values reported in Secs. A.1 and A.2 were only measured with one chip sample labeled "01". The values used for the evaluation of the hypothesis in Sec. 5.3.4 were determined for all 24 samples and the mean and standard deviation are reported. Note that for the following characterization and for the evaluation of the hypothesis only a subset of the testcases mentioned above were used.

Sec. A.1.1 presents the leakage current of the SoC and the reconfigurable module. The active current of the MCU part of the SoC is discussed in Sec. A.1.2 for an infinite loop and for the inactive low-power modes. Finally, the active current for the ADT7310 sensor interface task is presented in Sec. A.1.3.

A.1.1 Leakage

Table A.1 shows the leakage current of the WSN SoC, separately for both power domains. The measurement was performed at room temperature of approximately 25°C with a supply voltage of 3.3 V. The power analysis was performed for the three operating conditions "`BEST-MIL`", "`TYPICAL`", and "`WORST-MIL`" as defined by the standard cell library. The power analysis results in the following sections are reported for the typical operating condition, if not otherwise noted. No clock signal was supplied, all inputs of the WSN SoC were either tied to 0 V or 3.3 V, and all outputs were unloaded. However, the large value of 1,204.3 nA measured for the CPU power domain points to an unobserved path through the WSN SoC pins.

Table A.1: Leakage current of the WSN SoC for both power domains.

	Measured	Power Analysis		
		Best	Typ.	Worst
Power Domain	3.30 V, 25°C	3.63 V, -50°C	3.30 V, 25°C	3.00 V, 150°C
CPU	1,204.30 nA	13.88 nA	15.26 nA	16.79 nA
Reconf. Mod.	15.42 nA	6.70 nA	7.37 nA	8.11 nA

A.1.2 Active Current

The active current is determined for three operating modes of the CPU

- performing an infinite loop,
- inactive in low-power mode 1 (LPM1, CPU and RAMs deactivated), and
- inactive in low-power mode 3 (LPM3, CPU, RAMs, CPU peripherals deactivated).

For more details on the low-power modes and the active units see Sec. 5.1.2. Additionally the active current of the reconfigurable module without configuration is measured. The power consumption for each operating mode was measured at 1 MHz, 4 MHz, and 10 MHz. The current was modeled with the equation $I = I_0 + I_f \cdot f$ and separated using linear regression. The results are summarized in Tab. A.2. Note that I_0 deviates from the leakage current previously reported because of uncontrolled temperature variation between the measurements and the statistical approach.

Table A.2: Active current of the WSN SoC for both power domains.

Measured		
Infinite Loop:	I =	630.5 nA + 744.5 µA/MHz ·f
LPM1:	I =	3852.3 nA + 184.4 µA/MHz ·f
LPM3:	I =	3530.3 nA + 107.8 µA/MHz ·f
Reconf. Mod.:	I =	-238.0 nA + 59.4 µA/MHz ·f
Power Analysis: Best Case		
Infinite Loop:	I =	13.9 nA + 456.6 µA/MHz ·f
LPM1:	I =	13.9 nA + 193.6 µA/MHz ·f
LPM3:	I =	13.9 nA + 121.1 µA/MHz ·f
Reconf. Mod.:	I =	6.7 nA + 56.4 µA/MHz ·f
Power Analysis: Typical		
Infinite Loop:	I =	15.3 nA + 477.7 µA/MHz ·f
LPM1:	I =	15.3 nA + 208.1 µA/MHz ·f
LPM3:	I =	15.3 nA + 131.8 µA/MHz ·f
Reconf. Mod.:	I =	7.4 nA + 58.6 µA/MHz ·f
Power Analysis: Worst Case		
Infinite Loop:	I =	16.8 nA + 531.9 µA/MHz ·f
LPM1:	I =	16.8 nA + 231.7 µA/MHz ·f
LPM3:	I =	16.8 nA + 150.1 µA/MHz ·f
Reconf. Mod.:	I =	8.1 nA + 61.8 µA/MHz ·f

Power analysis was performed only for an operating frequency of 1 MHz because the reported active current linearly scales with the frequency and the leakage current is independent of the frequency. In Tab. A.2 the reported active current as well as the separately reported leakage current are shown. Note that no power models of the RAMs were available, therefore the difference between measurement and power analysis represents the RAMs. However, for LPM1 and LPM3, the measured current is *lower* than the result of the power analysis, hence the difference can not be used. A detailed investigation of the power analysis reports revealed, that the error results from too high power estimates of the clock tree. On the other hand, the power analysis result at typical operating conditions of the active current of the reconfigurable module of 58.6 µA/MHz only differs by 1.3% from the measured value of 59.4 µA/MHz.

In Tab. A.3 the current consumption of the WSN SoC is compared to commercial MCUs. The current consumption is 1.9 to 6.8 times higher for an infinite loop and 43 to 278 times higher in LPM3. This difference is caused on one hand because only one clock domain and only coarse manual clock gating are implemented. On the other hand, the WSN SoC is manufactured in a standard CMOS process without special low-power features and it is operated at a slightly higher voltage.

Table A.3: Current consumption of the WSN SoC compared to commercial MCUs at 1MHz using typical values specified in the datasheets. The power modes of the MCUs were selected equivalent to the WSN SoC. The active current of the MSP430FR6972 FRAM MCU depends on the cache hit rate. For the PIC16LF727 no dedicated numbers for low-power states are provided in the datasheet. The active and LPM0 current of the ATmega88PA are specified at 2.0 V and the LPM3 and LPM4 values are specified at 3.0 V in the datasheet.

MCU	Inf. Loop	LPM0	LPM3	Leakage/LPM4	V_{DD}	Reference
WSN SoC	745 µA	188 µA	111.3 µA	1.20 µA	3.3 V	Sec. A.1
MSP430F1232	300 µA	55 µA	1.6–2.3 µA	0.10–0.80 µA	3.0 V	[Tex04]
MSP430F2232	390 µA	90 µA	0.9–2.6 µA	0.10–1.50 µA	3.0 V	[Tex10a]
MSP430F5418A	290 µA	69 µA	1.7–2.6 µA	0.10 µA	3.0 V	[Tex10b]
MSP430FR6972	110-370 µA	80 µA	0.4 µA	0.02 µA	3.0 V	[Tex15b]
PIC16LF727	150 µA				3.0 V	[Mic09]
ATmega88PA	200 µA	30 µA	0.9 µA	0.10 µA	2.0 V	[Atm10]

In Tab. A.4 and Fig. A.1 the power analysis results for the individual contributions to the total current consumption are shown. The columns “meas.” show the measured values and “P.A.” show the power analysis results at typical operating condition. Both power domains were measured separately and are shown in the rows “Total CPU (meas.)” and “Reconf. Mod.”. The leakage currents were measured in separate testcases (see above) and are shown for reference, but not included in the grand total at the bottom row. On the other hand, the values determined with power analysis only specify the active current, therefore the leakage current is included in the grand total of the respective columns. The row “Memories” only reflects the interface logic circuit but not the actual memory cells. The reconfigurable module is not configured, therefore its power consumption is solely caused by its clock tree.

The power analysis results show that the main contribution is caused by the clock tree. In both low-power modes, the clock of the OpenMSP430 CPU is turned off. The small remaining power consumption is caused by the clock gates for MClk and SMClk. The differing power consumption of the peripherals and the reconfigurable module between infinite loop and LPM1 is caused by the activity of the address and data bus, which is connected to the memories and the other modules. The “SPI Master” and the “I2C Master” are peripherals only used by the reconfigurable module and are therefore not affected by the bus activity.

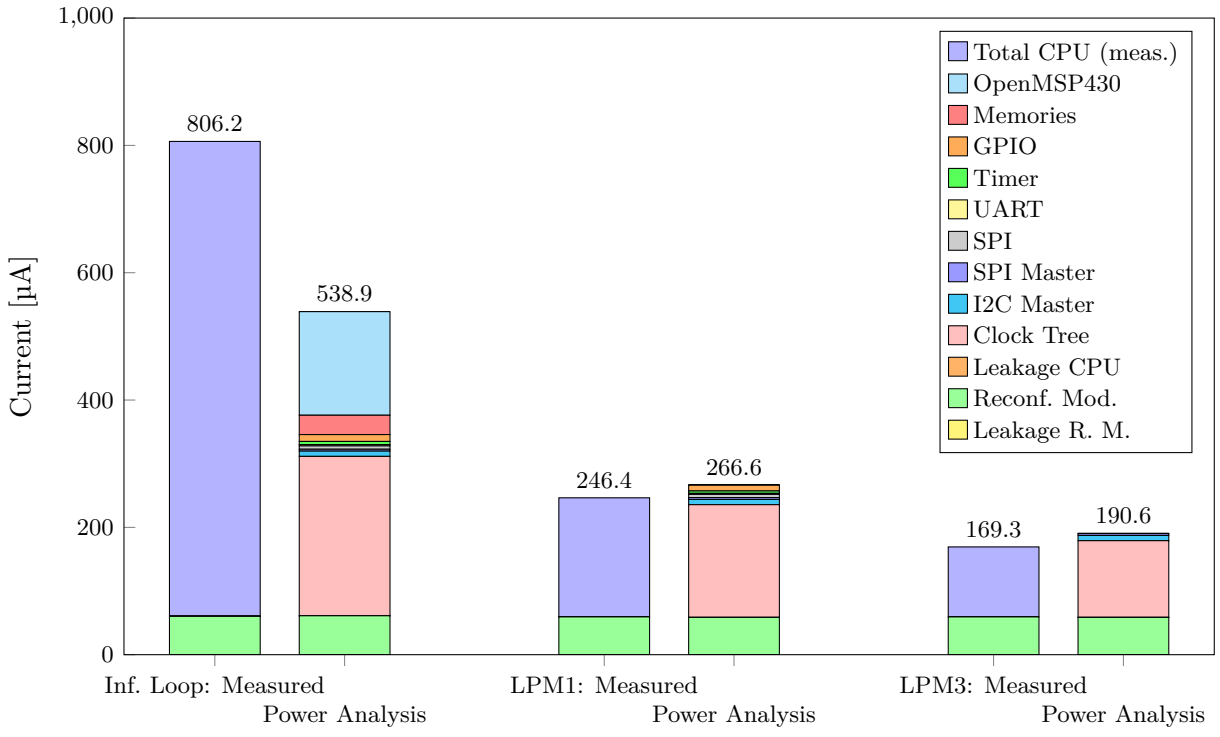
A.1.3 Sensor Interface Task: ADT7310

Additional to the inactive case and the infinite loop, the WSN SoC is characterized while executing the ADT7310 sensor interface task in firmware. In the following analysis an MCU without a reconfigurable module is assumed. Therefore only the CPU power domain is considered. The delay periods t_S and t_P are implemented with three different operating modes: polling, LPM1, and LPM3. These three modes are similar to the previous section, but differ in the fact that apart from the delay periods the CPU is executing productive firmware code.

Table A.4: Active current consumption in µA of the WSN SoC in the three operating states of the CPU at 1 MHz clock frequency (explanation in the text).

	Infinite Loop		LPM1		LPM3	
	meas.	P.A.	meas.	P.A.	meas.	P.A.
Total CPU (meas.)	746	–	187.1	–	110	–
OpenMSP430	–	162.7	–	0.4149	–	0.2836
Memories	–	30.39	–	0	–	0
GPIO	–	10.78	–	9	–	0
Timer	–	4.858	–	3.727	–	0
UART	–	2.151	–	1.664	–	0
SPI	–	5.227	–	4.963	–	0
SPI Master	–	2.93	–	2.93	–	2.93
I2C Master	–	8.288	–	8.288	–	8.288
Clock Tree	–	250.4	–	177	–	120.5
Leakage CPU	1.204	0.01526	1.204	0.01526	1.17	0.01526
Reconf. Mod.	60.22	61.14	59.34	58.58	59.33	58.58
Leakage R. M.	0.01542	0.007372	0.01542	0.007372	0.01632	0.007372
Total	806.2	538.9	246.4	266.6	169.3	190.6

Figure A.1: Active current consumption of the WSN SoC in the three operating states of the CPU at 1 MHz clock frequency (values from Tab. A.4).



The polling and LPM1 implementation of the delay periods use the integrated timer module. The implementation of the delay periods using LPM3 revealed a design issue of the WSN SoC which was not considered during the development. The clock signal of the timer module and the GPIO module is turned off in LPM3 (cf. Sec. 5.1.2). Therefore neither the timer nor an external interrupt via the GPIO pins can be used to wake-up the CPU from LPM3. However, the interrupt request signals from the reconfigurable module to the CPU are not synchronized and can be used to exit from LPM3. To solve this problem, a new post-silicon application was implemented which connects an input of the reconfigurable module to the interrupt request signal. The input was driven by an external pulse generated with the SmartFusion FPGA. The pulse generator was implemented by Georg Blemenschitz, because the ATmega88PA also requires an external interrupt to wake-up from its low-power mode [Ble15]. For more details on this workaround see Sec. 5.6.3.

For the three different implementations of the delay periods, one common firmware source code with C `#ifdef`s to realize the small differences was developed and compiled to separate binaries with Makefile settings. These required 1,404, 1,388, and 1,878 bytes program memory and 6, 4, and 4 bytes of data memory, respectively. The firmware using the LPM3 includes 508 bytes of configuration data and the driver for the reconfigurable module and excludes initialization routines for the timer.

The power consumption is measured as well as determined with power analysis at an operating frequency of 1 MHz with a variation of the number of sensor measurements per second from 5 to 200, i.e., t_C varied from 200 ms to 5 ms, respectively. The current consumption depending on the number of measurements per second is shown in Tab. A.5. The second column shows the current consumption when performing no measurements. In the third and fourth column the current consumption at the least and the most activity are shown for comparison. Measurements and power analysis were performed for intermediate values as well. For reference, in the fifth column the current consumption of an infinite loop is shown. This is considerably higher than the sensor interface task implemented with polling, because the tight infinite loop causes higher switching activity than the polling loop.

Table A.5: Current consumption of the WSN SoC at 1MHz performing the ADT7310 sensor interface task with three different implementations of the delay periods.

Mode	0 meas./s	5 meas./s	200 meas./s	Inf. Loop	Current
Measurement					
Polling:	679.5 µA	667.9 µA	673.9 µA	746.0 µA	I = 667.7 µA + 29.9 nA·meas./s
LPM1:	187.1 µA	187.8 µA	216.4 µA	746.0 µA	I = 187.1 µA + 146.5 nA·meas./s
LPM3:	110.0 µA	110.7 µA	142.1 µA	745.5 µA	I = 110.0 µA + 160.7 nA·meas./s
Power Analysis: Typical					
Polling:	434.2 µA	426.4 µA	430.1 µA	477.7 µA	I = 426.3 µA + 19.3 nA·meas./s
LPM1:	208.1 µA	210.9 µA	223.4 µA	477.7 µA	I = 210.6 µA + 64.3 nA·meas./s
LPM3:	131.8 µA	133.9 µA	149.8 µA	477.7 µA	I = 133.5 µA + 81.9 nA·meas./s

The values of the equation in the last column are calculated with linear regression as $I = I_0 + I_m \cdot f_m$ with f_m as the number of sensor measurements per second. The base current I_0 decreases with the implementation of the delay periods from 667.7 µA to 110.0 µA, but the current proportion per additional measurement I_m increases from 29.9 nA per meas./s to 160.7 nA per meas./s due to the overhead of the ISR and the statistical approach. The current consumption of an infinite loop for

the implementation using LPM3 (745.5 µA) slightly differs from the other two implementations (746.0 µA) due to measurement inaccuracies and uncontrolled temperature changes.

The current consumption of the implementation using polling is higher without performing measurements (679.5 µA) than the current consumption at 200 measurements per second (673.9 µA). The reason are different memory locations of the two polling loops which cause different switching activity of the address bus and therefore power consumption. The loop for t_P is located at a favorable address, while in the testpoint with 0 meas./s the firmware is executing the loop for t_S, which is located at `0xE2DE` = `1110 0010 1101 1110` to `0xE2E5` = `1110 0010 1110 0101`. Since the memory is addressed word-wise, the five address signals A5 down to A1 require switching activity.

If the loop is shifted (e.g., using NOP instructions) to `0xE2E2` = `1110 0010 1110 0010` to `0xE2E9` = `1110 0010 1110 1001`, the switching activity is limited to the three address signals A3 down to A1. The influence of the code address results in the difference of 11.8 µA between the base current I_0 calculated with the linear regression and the measured current, which is 1.77%. However, the implementation using polling is not relevant for the evaluation of the hypothesis and only evaluated here for the characterization of the WSN SoC. For both implementations using low-power modes, the measured current consumption without sensor measurements is identical to the base current calculated with the linear regression (187.1 µA and 110.0 µA) although this data was not included in the regression.

The increased current consumption of the implementation using polling is also observed for the power analysis results (434.2 µA vs. 426.3 µA), but the difference is only 7.9 µA. This points to the fact, that the program memory cell is also affected by the memory location. For both implementations using low-power modes, the current consumption without sensor measurements and the result of the linear regression differ (208.1 µA vs. 210.6 µA and 131.8 µA vs. 133.5 µA). No explanation could be found for this result. Note that here the power analysis results are also higher than the measurement results, but show a lower slope.

Table A.6 and Fig. A.2 show the individual contributions of the total power consumption at 1 MHz and 200 meas./s (t_C = 5 ms). The power analysis result for the implementation using polling does not include the RAM power consumption and is therefore 36.15% lower than the measurement. The power analysis results for both other implementations are slightly higher than the measurements, despite the power consumption of the RAMs is not included. However, they show that the clock tree is the major component of the power consumption which accounts for 37.1% for the implementation using polling (referred to the measured current) and 82.1% and 86.4% for the implementations using LPM1 and LPM3, respectively (referred to the power analysis total current).

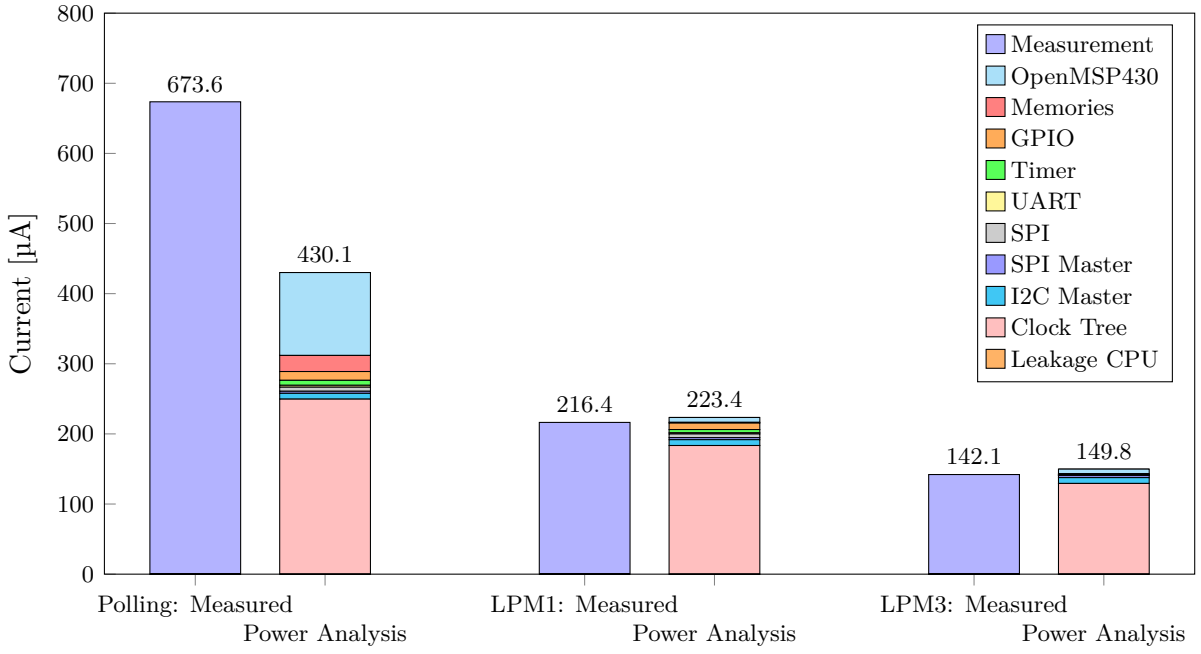
In contrast to Tab. A.4, the current consumption of the CPU peripherals is non-zero, because of the address and data bus activity during the code execution. Additionally, the "SPI" module is used for communication with the ADT7310 sensor. In the implementations using polling and LPM1, the "Timer" module is utilized to implement the delay periods and therefore has a higher power consumption than for LPM3, where an external interrupt is used to exit the low-power mode (see above).

The two peripherals of the reconfigurable module "SPI Master" and "I2C Master" consume 2.93 µA and 8.288 µA, respectively, independent of the operation of the CPU. Both are included in the CPU power domain and directly supplied with the main clock without gating. The values

Table A.6: Active current consumption in µA of the CPU performing the ADT7310 sensor interface task at 1 MHz clock frequency and 200 meas./s with three different implementations of the delay periods. The reconfigurable module is not included. For more explanation see Tab. A.4.

	Polling		LPM1		LPM3	
	meas.	P.A.	meas.	P.A.	meas.	P.A.
Total CPU (meas.)	673.6	–	216.4	–	142.1	–
OpenMSP430	–	118.2	–	6.939	–	6.546
Memories	–	22.97	–	1.061	–	1.021
GPIO	–	12.55	–	9.212	–	0.6712
Timer	–	7.094	–	4.579	–	0.3133
UART	–	2.696	–	1.713	–	0.1321
SPI	–	5.809	–	5.256	–	0.5327
SPI Master	–	2.93	–	2.93	–	2.93
I2C Master	–	8.288	–	8.288	–	8.288
Clock Tree	–	249.6	–	183.4	–	129.4
Leakage CPU	1.246	0.0153	1.204	0.0153	1.17	0.0153
Total	673.6	430.1	216.4	223.4	142.1	149.8
Difference	–	−36.15%	–	3.257%	–	5.456%

Figure A.2: Active current consumption of the CPU performing the ADT7310 sensor interface task at 1 MHz clock frequency and 200 meas./s with three different implementations of the delay periods (values from Tab. A.6).



are included here to enable the comparison of measurement and power analysis results. However, these are not included in the final comparison of the energy consumption in Sec. 5.3.4.

To summarize, the CPU implementation of the ADT7310 sensor interface task consumes 142.1 µA executing 200 meas./s at 1 MHz with delay periods implemented using LPM3. In the next section, the current consumption of the reconfigurable module is investigated.

A.2 Characterization of the Reconfigurable Module

In this section the power consumption of the reconfigurable module is characterized. Two design issues were revealed in the utilized example application which were fixed with new applications. These are discussed in Sec. A.2.1. In Sec. A.2.2 the total supply current is investigated. The individual contributions to that current are discussed in Sec. A.2.3.

A.2.1 Example and New Applications

For the characterization of the reconfigurable module it is configured with the "`ADT7310`" example application. This uses a 16-bit counter for the period delay t_P and a 32-bit counter for the sensor delay t_S. During the development, the sensor delay t_S was simulated as 240 ms at 1 MHz which requires a 32-bit counter. The period delay t_P was simulated very short and therefore a 16-bit counter was implemented.

For the measurement setup (cf. Sec. 5.3.2) the sensor delay is fixed to 2 ms and the period delay is varied with the cycle time t_C from 5 ms to 200 ms. The long period delays can not be reached with the 16-bit counter, especially at higher frequencies. Therefore three new post-silicon applications were developed to evaluate all combinations of 16-bit and 32-bit counters. The applications use the suffix "Pxx Syy" to specify the width of the period and the sensor delay counters, e.g., "P16 S32" for the original example application with a 16-bit period delay counter and a 32-bit sensor delay counter.[1]

The measurements revealed another bug in the example application which was also copied to the three new applications. In the time periods when a counter was unused (cf. Fig. 5.9 on p. 137), the FSM which controls the counter did not stop the counter but permanently activated the `Preset_i` input (cf. Lst. 4.6 on p. 107). This caused increased current consumption. For further investigation, three additional new applications were developed (derived from "P32 S16") which properly stopped one or both of the counters while not used.

The naming scheme was extended to append "L" to the stopped counter, e.g., "P32L S16" for the application which properly stops the period delay counter and uses the unmodified control of the sensor delay counter. Additionally, for all combinations of counter widths, new applications with properly stopped counters were developed. This results in a total of ten slightly different implementations of the ADT7310 sensor interface task: "P16 S32" (original), "P32 S16", "P16 S16", "P32 S32", "P32L S16", "P32 S16L", "P32L S16L", "P16L S16L", "P16L S32L", and "P32L S32L".

[1] Please note the reversed order: For a sensor measurement and in the energy model equation, the sensor delay is first, but in the suffix the period delay is stated first.

Additionally power analysis was used to investigate the current consumption of the counters in these cases. In Fig. A.3 the current consumption of the "P32 S16" variant which permanently presets the unused counters is shown for 5–200 meas./s. In Fig. A.4 "P32L S16L" with properly stopped counters is shown (note the changed vertical scale). The current consumption of the 16-bit "Counter 1", which implements the sensor delay of 2 ms within each measurement cycle, requires 4.78 µA for 5 meas./s to 5.65 µA for 200 meas./s with permanent preset, but in the "P32L S16L" improved implementation 2.82 µA for 200 meas./s and approaches zero with 0.11 µA for 5 meas./s.

The period delay counter implemented with the 32-bit "Counter32 1" requires 9.91 µA for 5 meas./s to 9.16 µA for 200 meas./s in the "P32 S16" case and is reduced to 5.65 µA for 5 meas./s to 4.78 µA for 200 meas./s for "P32L S16L". The current decreases because it is used a smaller fraction of the time. More details on these post-silicon applications are discussed in Sec. 5.6.3.

The firmware for the evaluation of the reconfigurable module was derived from the firmware used in the previous section. The program logic which implements the sensor interface task was removed and the driver for the reconfigurable module was integrated. After the activation of the reconfigurable module, the firmware enters LPM3. The CPU is only activated with an interrupt, if the measured value differs from the previous value for more than a certain threshold. In the ISR, the new value is queried via the parameterization interface.

`#ifdef`s and Makefile settings were used to compile ten slightly different version for the varying counter implementations. These required 2,214 to 2,250 bytes of program memory and 4 bytes of data memory. The program memory includes 508 bytes of configuration data. This is slightly more than the 3,889 bits (487 bytes) would require due to the use of full bytes to store the data for each of the four configuration registers and meta-data like the length. The code size varies slightly because the 32-bit counters use two parameters with a connection type "`Word`" and therefore a different number of parameters has to be set for the initialization.

A.2.2 Total Supply Current

Measurements and power analysis for the ten different implementations of the ADT7310 sensor interface task were conducted. The total supply current of the reconfigurable module for an operating frequency of 1 MHz is shown in Tab. A.7.

The current consumption of the disabled reconfigurable module with permanently presetting counters in the second column of the first four rows show as expected that the 32-bit counters require more current than the 16-bit counter. The difference between "P16 S32" and "P32 S16" is caused by a differing fan-out into the interconnect of the mapped counter cells. In the next three rows the improvement with stopped counters is gradually introduced which considerably reduces the current consumption. In the last four rows the counters are properly stopped. This results in a reduction by 9.1 µA for two 16-bit counters, 13.0 µA and 12.9 µA for mixed width counters, and 16.8 µA for two 32-bit counters. The measured values 60.05 µA–61.76 µA vary slightly and differ a small amount from the power analysis result 58.59 µA which is (nearly) identical to the current consumption of the unconfigured reconfigurable module (see Tab. A.4).

The next two columns show that the active reconfigurable module requires more current than when disabled. In the last column, a linear equation for the current consumption is given. The base current I_0 is higher than for the disabled reconfigurable module because this reflects the running period delay counter and the presetting sensor delay counter in the according rows.

Figure A.3: Composition of the current consumption of the reconfigurable module at 1 MHz with the "P32 S16" variant of the ADT7310 application for different frequencies of sensor measurements. Only the components which vary are shown.

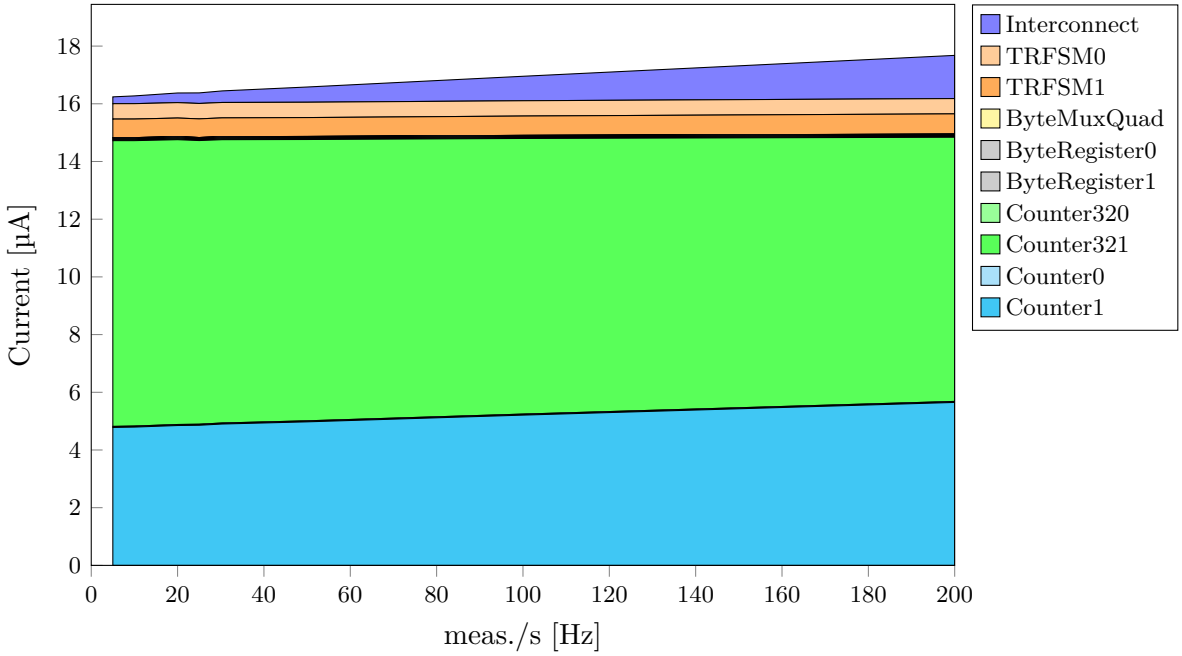


Figure A.4: Composition of the current consumption of the reconfigurable module at 1 MHz with the "P32L S16L" variant of the ADT7310 application for different frequencies of sensor measurements. Only the components which vary are shown. Note the different vertical scale compared to Fig. A.3.

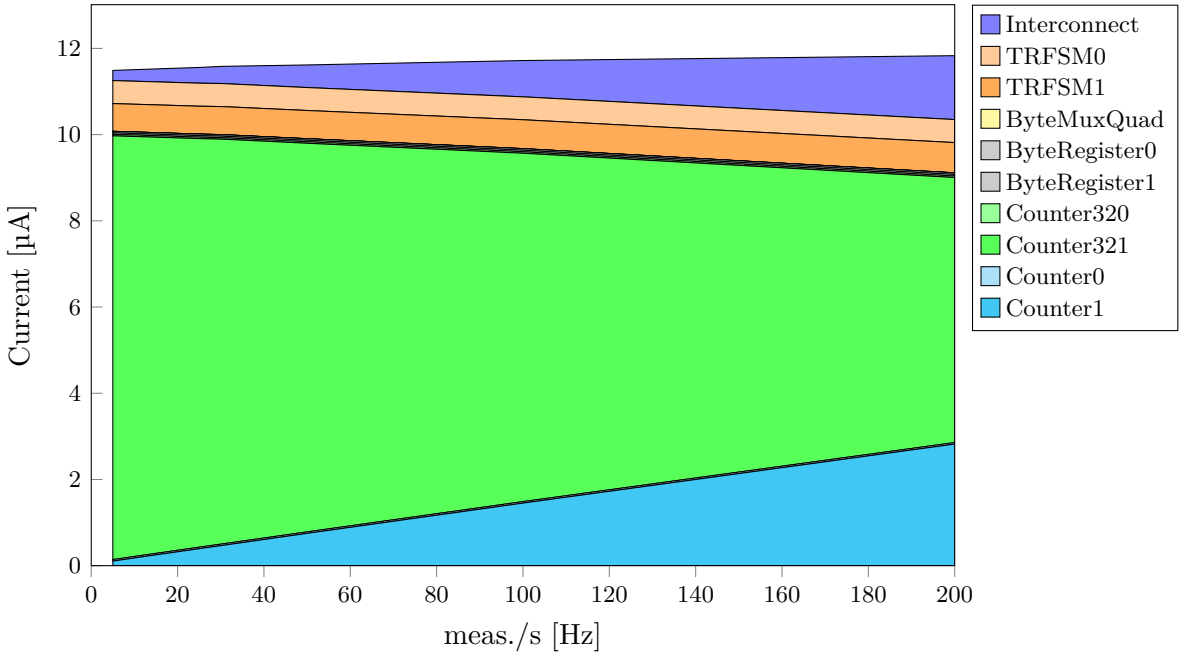


Table A.7: Current consumption of the reconfigurable module at 1 MHz for the ADT7310 application with different implementations of the delay periods. The "P16 S*yy*" implementations can only reach a maximum period delay of 50 ms which limits the minimum activity to 20 meas./s instead of 5 meas./s.

Mode	Disabled	5/20 meas./s	200 meas./s	Current
Measurement				
P16 S32	72.96 µA	76.60 µA	76.09 µA	I = 76.66 µA − 2.829 nA·meas./s
P32 S16	73.27 µA	74.82 µA	76.39 µA	I = 74.78 µA + 8.108 nA·meas./s
P16 S16	70.58 µA	74.56 µA	74.09 µA	I = 74.61 µA − 2.579 nA·meas./s
P32 S32	78.57 µA	78.57 µA	78.79 µA	I = 78.57 µA + 1.134 nA·meas./s
P32L S16	65.14 µA	74.75 µA	73.17 µA	I = 74.80 µA − 8.101 nA·meas./s
P32 S16L	68.40 µA	70.00 µA	73.38 µA	I = 69.91 µA + 17.377 nA·meas./s
P32L S16L	60.27 µA	69.92 µA	70.14 µA	I = 69.92 µA + 1.128 nA·meas./s
P16L S16L	61.45 µA	70.28 µA	69.51 µA	I = 70.37 µA − 4.272 nA·meas./s
P16L S32L	60.05 µA	68.11 µA	69.04 µA	I = 68.01 µA + 5.195 nA·meas./s
P32L S32L	61.76 µA	69.89 µA	70.18 µA	I = 69.89 µA + 1.463 nA·meas./s
Power Analysis: Typical				
P16 S32	71.00 µA	75.22 µA	74.71 µA	I = 75.28 µA − 2.806 nA·meas./s
P32 S16	71.10 µA	73.45 µA	75.02 µA	I = 73.42 µA + 8.050 nA·meas./s
P16 S16	67.40 µA	73.26 µA	72.84 µA	I = 73.32 µA − 2.365 nA·meas./s
P32 S32	74.70 µA	77.02 µA	77.26 µA	I = 77.02 µA + 1.229 nA·meas./s
P32L S16	63.31 µA	73.38 µA	72.02 µA	I = 73.43 µA − 6.945 nA·meas./s
P32 S16L	66.37 µA	68.78 µA	72.17 µA	I = 68.71 µA + 17.399 nA·meas./s
P32L S16L	58.59 µA	68.70 µA	69.18 µA	I = 68.71 µA + 2.404 nA·meas./s
P16L S16L	58.59 µA	69.18 µA	68.61 µA	I = 69.26 µA − 3.170 nA·meas./s
P16L S32L	58.59 µA	67.08 µA	68.12 µA	I = 66.98 µA + 5.776 nA·meas./s
P32L S32L	58.59 µA	68.70 µA	69.19 µA	I = 68.71 µA + 2.454 nA·meas./s

In some cases the slope I_m is negative, i.e., more measurements per second *reduce* the current consumption. The reason is that each counter has an output which provides its current value to the interconnect. The 16-bit counters have a larger fan-out to the interconnect and therefore cause higher switching activity. Complex interaction between the number of toggling bits, the start value, and the fan-out of the counter result in these unexpected values.

A.2.3 Supply Current Contributions

To identify the individual contributions of the total power consumption, power analysis was performed for an operating frequency of 1 MHz and 200 meas./s. In Tab. A.8 and Fig. A.5 the current consumption of the first four cases with permanently presetting counters of different width are split into the contributions of the interconnect, the cells, the infrastructure for parameterization ("Param") and configuration ("Config"), the clock tree and the SPI master. Note that the SPI master is shown for reference but not included in the sum because it is supplied from the CPU power domain.

In Tab. A.9 and Fig. A.6 the current consumption with gradually stopped counters is shown. "P32 S16" is repeated for reference. These show that the total reduction is mainly achieved by "Counter32 1" and "Counter 1". The current consumption of the interconnect as well as the clock

tree is slightly reduced, while the two TR-FSM instances show a minimal increase due to the improved FSMs.

In Tab. A.10 and Fig. A.7 the current consumption with different counter widths for properly stopped counters is shown. "P32L S16L" is repeated for reference. The two logical counter instances are mapped to the four physical counter instances as shown in Tab. A.11. The period delay counter requires 6.148 µA when implemented as 32-bit counter and 4.373 µA or 4.0 µA as 16-bit counter. The sensor delay counter requires 4.167 µA as 32-bit counter and 2.591 µA or 2.821 µA as 16-bit counter.

The results for all implementations show that for the reconfigurable module also the clock tree causes the major current consumption with 72.1% ("P32 S32") to 81.8% ("P16L S32L"). Although during the synthesis, clock gating was automatically inserted, this only affects the sections of the clock tree which are closest to the inactive registers. The hierarchically higher levels of the clock tree are still active. This should be investigated in more detail for future implementations.

The counters and the interconnect cause almost the entire residual current consumption. The current consumption of the interconnect is mainly caused by the counter output ports. The TR-FSMs only require a small portion of the current consumption.

Note that the power analysis was performed for a single sensor measurement cycle. This used the same measurement value as already stored as old value. Therefore no switching activity is observed in the datapath. On the other hand, the measurement was performed for 50 cycles with a set of different sensor measurement values. This causes (at least partially) the small differences between measurement and power analysis.

For the evaluation of the hypothesis, one implementation with the CPU and one implementation with the reconfigurable module have to be selected. For the CPU, the implementation using LPM3 is selected because it requires the lowest current. For the reconfigurable module, "P32L S16L" is selected, because it supports all cycle time values from 5 ms to 200 ms and uses the improved counter control.

Table A.8: Current consumption in µA of the reconfigurable module with an operating frequency of 1 MHz and performing 200 meas./s with different counter widths. The measured leakage current was determined in a separate testcase and is shown here for reference but not added to the total current consumption. The power analysis totals in this and the following two tables as well as the according bar diagrams in Figs. A.5–A.7 differ slightly from the values given in Tab. A.7 for 200 meas./s due to insufficient resolution of the hierarchical power analysis report used to calculate this table (cf. Sec. 5.3.3). The last row specifies the difference of the current consumption determined with measurement and with power analysis.

	P16 S32		P32 S16		P16 S16		P32 S32	
	meas.	P.A.	meas.	P.A.	meas.	P.A.	meas.	P.A.
Measurement	76.09	–	76.39	–	74.09	–	78.79	–
Interconnect	–	1.379	–	1.497	–	2.867	–	0.2964
TRFSM0	–	0.5302	–	0.5301	–	0.5301	–	0.5301
TRFSM1	–	0.7009	–	0.7009	–	0.7009	–	0.7009
WordMuxDual0	–	0	–	0	–	0	–	0
WordMuxDual1	–	0	–	0	–	0	–	0
Byte2WordSel	–	0	–	0	–	0	–	0
ByteMuxDual0	–	0	–	0	–	0	–	0
ByteMuxDual1	–	0	–	0	–	0	–	0
ByteMuxQuad	–	0.002442	–	0.002442	–	0.002442	–	0.002442
AddSubCmp0	–	0	–	0	–	0	–	0
AddSubCmp1	–	0	–	0	–	0	–	0
ByteRegister0	–	0.03583	–	0.03583	–	0.03583	–	0.03583
ByteRegister1	–	0.03613	–	0.03613	–	0.03613	–	0.03613
WordRegister0	–	0.03455	–	0.03455	–	0.03455	–	0.03455
WordRegister1	–	0.03455	–	0.03455	–	0.03455	–	0.03455
WordRegister2	–	0.03455	–	0.03455	–	0.03455	–	0.03455
AbsDiff	–	0	–	0	–	0	–	0
Counter320	–	9.224	–	0.03455	–	0.03455	–	9.224
Counter321	–	0.03455	–	9.158	–	0.03455	–	9.158
Counter0	–	5.527	–	0.03485	–	5.039	–	0.03485
Counter1	–	0.03455	–	5.648	–	6.148	–	0.03455
Param	–	0.4806	–	0.4806	–	0.4806	–	0.4806
Config	–	0.8848	–	0.8848	–	0.8848	–	0.8848
Clock Tree	–	55.69	–	55.97	–	55.98	–	55.87
SPI Master	–	2.994	–	2.994	–	2.994	–	2.994
Leakage	0.01632	0.007364	0.01632	0.007364	0.01632	0.007364	0.01632	0.007364
Total	76.09	74.67	76.39	75.13	74.09	72.89	78.79	77.4
Difference	–	−1.861%	–	−1.656%	–	−1.628%	–	−1.762%

Table A.9: Continuation of Tab. A.8 with gradually using stopped counters, "P32 S16' repeated for reference.

	P32 S16		P32L S16		P32 S16L		P32L S16L	
	meas.	P.A.	meas.	P.A.	meas.	P.A.	meas.	P.A.
Measurement	76.39	–	73.17	–	73.38	–	70.14	–
Interconnect	–	1.497	–	1.494	–	1.482	–	1.482
TRFSM0	–	0.5301	–	0.5346	–	0.5301	–	0.5346
TRFSM1	–	0.7009	–	0.7052	–	0.7	–	0.7012
WordMuxDual0	–	0	–	0	–	0	–	0
WordMuxDual1	–	0	–	0	–	0	–	0
Byte2WordSel	–	0	–	0	–	0	–	0
ByteMuxDual0	–	0	–	0	–	0	–	0
ByteMuxDual1	–	0	–	0	–	0	–	0
ByteMuxQuad	–	0.002442	–	0.002442	–	0.002442	–	0.002442
AddSubCmp0	–	0	–	0	–	0	–	0
AddSubCmp1	–	0	–	0	–	0	–	0
ByteRegister0	–	0.03583	–	0.03583	–	0.03583	–	0.03583
ByteRegister1	–	0.03613	–	0.03613	–	0.03613	–	0.03613
WordRegister0	–	0.03455	–	0.03455	–	0.03455	–	0.03455
WordRegister1	–	0.03455	–	0.03455	–	0.03455	–	0.03455
WordRegister2	–	0.03455	–	0.03455	–	0.03455	–	0.03455
AbsDiff	–	0	–	0	–	0	–	0
Counter320	–	0.03455	–	0.03455	–	0.03455	–	0.03455
Counter321	–	9.158	–	6.148	–	9.158	–	6.148
Counter0	–	0.03485	–	0.03485	–	0.03485	–	0.03485
Counter1	–	5.648	–	5.648	–	2.821	–	2.821
Param	–	0.4806	–	0.4806	–	0.4806	–	0.4806
Config	–	0.8848	–	0.8848	–	0.8848	–	0.8848
Clock Tree	–	55.97	–	55.89	–	55.88	–	55.82
SPI Master	–	2.994	–	2.994	–	2.994	–	2.994
Leakage	0.01632	0.007364	0.01632	0.007364	0.01632	0.007364	0.01632	0.007364
Total	76.39	75.13	73.17	72.04	73.38	72.19	70.14	69.13
Difference	–	−1.656%	–	−1.55%	–	−1.625%	–	−1.44%

Table A.10: Continuation of Tab. A.9 using stopped counters, "P32L S16L" repeated for reference.

	P32L S16L		P16L S16L		P16L S32L		P32L S32L	
	meas.	P.A.	meas.	P.A.	meas.	P.A.	meas.	P.A.
Measurement	70.14	–	69.51	–	69.04	–	70.18	–
Interconnect	–	1.482	–	2.858	–	1.376	–	0.2897
TRFSM0	–	0.5346	–	0.5346	–	0.5346	–	0.5346
TRFSM1	–	0.7012	–	0.7012	–	0.7012	–	0.7012
WordMuxDual0	–	0	–	0	–	0	–	0
WordMuxDual1	–	0	–	0	–	0	–	0
Byte2WordSel	–	0	–	0	–	0	–	0
ByteMuxDual0	–	0	–	0	–	0	–	0
ByteMuxDual1	–	0	–	0	–	0	–	0
ByteMuxQuad	–	0.002442	–	0.002442	–	0.002442	–	0.002442
AddSubCmp0	–	0	–	0	–	0	–	0
AddSubCmp1	–	0	–	0	–	0	–	0
ByteRegister0	–	0.03583	–	0.03583	–	0.03583	–	0.03583
ByteRegister1	–	0.03613	–	0.03613	–	0.03613	–	0.03613
WordRegister0	–	0.03455	–	0.03455	–	0.03455	–	0.03455
WordRegister1	–	0.03455	–	0.03455	–	0.03455	–	0.03455
WordRegister2	–	0.03455	–	0.03455	–	0.03455	–	0.03455
AbsDiff	–	0	–	0	–	0	–	0
Counter320	–	0.03455	–	0.03455	–	4.167	–	4.167
Counter321	–	6.148	–	0.03455	–	0.03455	–	6.148
Counter0	–	0.03485	–	2.591	–	4	–	0.03485
Counter1	–	2.821	–	4.373	–	0.03455	–	0.03455
Param	–	0.4806	–	0.4806	–	0.4806	–	0.4806
Config	–	0.8848	–	0.8848	–	0.8848	–	0.8848
Clock Tree	–	55.82	–	55.78	–	55.85	–	55.67
SPI Master	–	2.994	–	2.994	–	2.994	–	2.994
Leakage	0.01632	0.007364	0.01632	0.007364	0.01632	0.007364	0.01632	0.007364
Total	70.14	69.13	69.51	68.46	69.04	68.25	70.18	69.13
Difference	–	−1.44%	–	−1.513%	–	−1.151%	–	−1.492%

Figure A.5: Current consumption of the reconfigurable module with an operating frequency of 1 MHz and performing 200 meas./s with different counter widths (values from Tab. A.8).

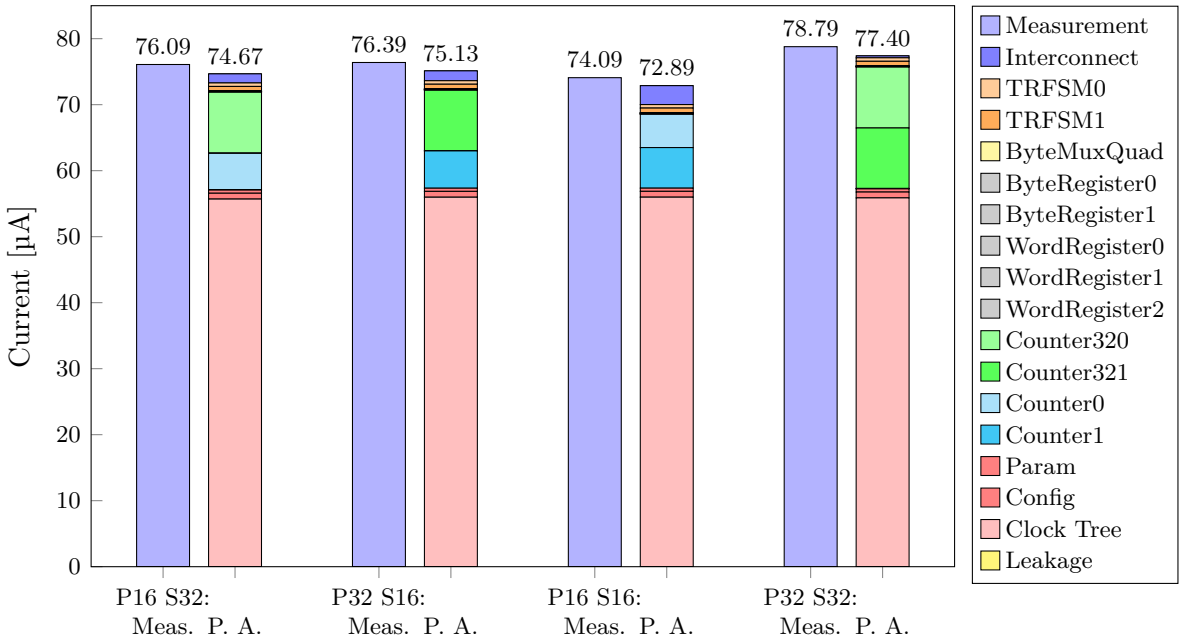


Figure A.6: Current consumption of the reconfigurable module with an operating frequency of 1 MHz and performing 200 meas./s with gradually using stopped counters, "P32 S16' repeated for reference (values from Tab. A.9).

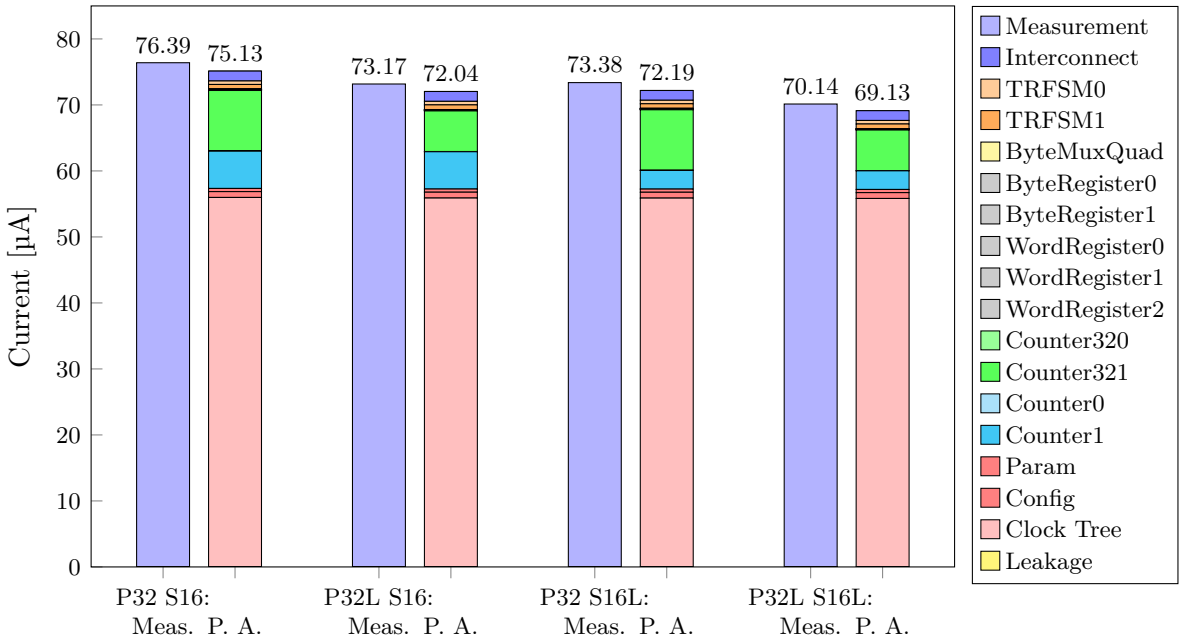


Figure A.7: Current consumption of the reconfigurable module with an operating frequency of 1 MHz and performing 200 meas./s using stopped counters, "P32L S16L" repeated for reference (values from Tab. A.10).

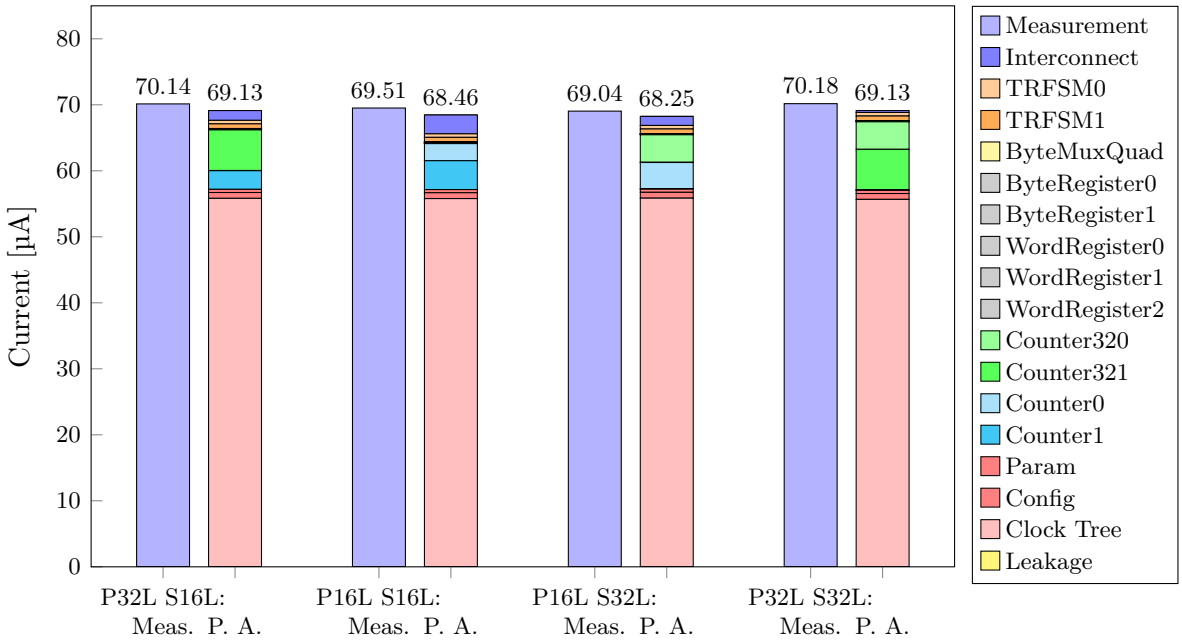


Table A.11: Mapping of the logical counter instances to the physical counter instances.

	Period Delay	Sensor Delay
P32L S16L	Counter32 1	Counter 1
P16L S16L	Counter 1	Counter 0
P16L S32L	Counter 0	Counter32 0
P32L S32L	Counter32 1	Counter32 0

Bibliography

[Act10a] Actel Corporation, Mountain View, CA 94043. *Actel's SmartFusion Intelligent Mixed-Signal FPGAs*, May 2010. Revision 2.

[Act10b] Actel Corporation, Mountain View, CA 94043. *SmartFusion Evaluation Kit User's Guide*, March 2010.

[ADI13] ADICSYS. *Synthetizable Programmable Core IP*, 2013.

[AL07] Peter J. Ashenden and Jim Lewis. *VHDL-2008: Just the New Stuff*. Systems on Silicon. Elsevier Science, 2007.

[ALS05] Victor Aken'Ova, Guy Lemieux, and Resve Saleh. An Improved "Soft" eFPGA Design and Implementation Strategy. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, pages 179–182, 18.–21. September 2005.

[Alt09] Altera Corporation. *Nios II C2H Compiler, User Guide*, November 2009. 9.1.

[Alt15a] Altera Corporation. *Altera SoCs Overview*, 2015. https://www.altera.com/products/soc/overview.html [2015-08-20].

[Alt15b] Altera Corporation. *Stratix 10 Overview*, 2015. https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html [2015-08-20].

[Ana06] Analog Devices, Inc., One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A. *ADuC70xx Precision Analog Microcontroller*, 2006. Rev. A.

[Ana09] Analog Devices, Inc., One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A. *ADT7310: ±0.5° C Accurate, 16-Bit Digital SPI Temperature Sensor*, 2009. Rev. 0.

[AR96] Arthur Abnous and Jan Rabaey. Ultra-Low-Power Domain-Specific Multimedia Processors. In *IX Workshop on VLSI Signal Processing*, pages 461–470, 30. October–1. November 1996.

[AS06] Victor Aken'Ova and Resve Saleh. A "Soft++" eFPGA Physical Design Approach with Case Studies in 180nm and 90nm. *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, 2.–3. March 2006.

[ASI+98] Arthur Abnous, Katsunori Seno, Yuji Ichikawa, Marlene Wan, and Jan M. Rabaey. Evaluation of a Low-Power Reconfigurable DSP Architecture. In *IPPS/SPDP Workshops*, pages 55–60, 1998.

[Atm10] Atmel Corporation, San Jose, CA. *8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash: ATmega48PA, ATmega88PA, ATmega168PA, ATmega328P*, April 2010. 8271B-AVR-04/10.

[AZW+02] Arthur Abnous, Hui Zhang, Marlene Wan, Varghese George, Vandana Prabhu, and Jan M. Rabaey. The Pleiades Architecture. In Alan Gatherer and Edgar Auslander, editors, *The Application of Programmable DSPs in Mobile Communications*, chapter 17. John Wiley & Sons, Ltd, Chichester, UK, 2002.

[BAPZ09a] Hichem Belhadj, Vishal Aggrawal, Ajay Pradhan, and Amal Zerrouki. Power-aware FPGA design. *Programmable Logic DesignLine*, 2009. http://www.pldesignline.com/213001682, http://www.pldesignline.com/213403832, http://www.pldesignline.com/214303718 [2015-08-20].

[BAPZ09b] Hichem Belhadj, Vishal Aggrawal, Ajay Pradhan, and Amal Zerrouki. Power-Aware FPGA Design. Technical report, Actel Corporation, February 2009.

[BEHB95] Gaetano Borriello, Carl Ebeling, Scott A. Hauck, and Steven Burns. The Triptych FPGA Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(4):491–501, December 1995.

[Ber15] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification, 2015. http://www.eecs.berkeley.edu/~alanmi/abc/ [2015-08-20].

[BFS04] Carlo Brandolese, William Fornaciari, and Fabio Salice. Source-Level Models for Software Power Optimization. In Enrico Macii, editor, *Ultra Low-Power Electronics and Design*, pages 156–171. Kluwer Academic Publishers, Dordrecht, 2004.

[Ble15] Georg Blemenschitz. Evaluierung der Reduktion der Leistungsaufnahme durch eine rekonfigurierbare Architektur. Master's thesis, Technische Universität Wien, Institut für Computertechnik, 2015. (in preparation).

[BMKS02] Elaheh Bozorgzadeh, Seda Ogrenci Memik, Ryan Kastner, and Majid Sarrafzadeh. Pattern Selection: Customized Block Allocation for Domain-Specific Programmable Systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, pages 190–196, June 2002.

[BMM01] Luca Benini, Giovanni De Micheli, and Enrico Macii. Designing Low-Power Circuits: Practical Recipes. *IEEE Circuits and Systems Magazine*, 1(1):6–25, First Quarter 2001.

[Bos15] Timm Bostelmann. personal communication, 2015.

[BR96] Stephen Brown and Jonathan Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design Test of Computers*, 13(2):42–57, Summer 1996.

[BS14] Timm Bostelmann and Sergei Sawitzki. A Conceptual Toolchain for an Application Domain Specific Reconfigurable Logic Architecture. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–4, December 2014.

[Buk08] Arkadiusz Bukowiec. *Synthesis of Finite State Machines for Programmable Devices Based on Multi-Level Implementation*. PhD thesis, University of Zielona Góra, Poland, 2008. http://zbc.uz.zgora.pl/Content/14528/PhD-ABukowiec.pdf [2015-08-20].

[Buk09] Arkadiusz Bukowiec. *Synthesis of Finite State Machines for FPGA Devices Based on Architectural Decomposition*, volume 13 of *Lecture Notes in Control and Computer Science*. University of Zielona Góra Press, Zielona Góra, Poland, 2009.

[Cal04] Edgar H. Callaway, Jr. *Wireless Sensor Networks, Architectures and Protocols*. Auerbach Publications, 2004. ISBN 0-8493-1823-8.

[CCA+13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24:1–24:27, September 2013.

[CEES14] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, first edition, 2014. http://www.zynqbook.com/ [2015-08-20].

[CFBE98] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and Compiling Applications for RaPiD. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 116–125, 15.–17. April 1998.

[CFF+99] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 23–40, 21.–24. March 1999.

[CH01] Katherine Compton and Scott Hauck. Totem: Custom Reconfigurable Array Generation. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM '01*, pages 111–119, 29. March–2. April 2001.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, June 2002.

[CH04] Katherine Compton and Scott Hauck. Flexibility Measurement of Domain-Specific Reconfigurable Hardware. In *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pages 155–161, 2004.

[CH07] Katherine Compton and Scott Hauck. Automatic Design of Area-Efficient Configurable ASIC Cores. *IEEE Transactions on Computers*, 56(5):662–672, May 2007.

[CH08] Katherine Compton and Scott Hauck. Automatic Design of Reconfigurable Domain-Specific Flexible Cores. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(8):493–503, May 2008.

[Cha07] Satrajit Chatterjee. *On Algorithms for Technology Mapping*. PhD thesis, University of California, Berkeley, 2007.

[Com03] Katherine Leigh Compton. *Architecture Generation of Customized Reconfigurable Hardware*. PhD thesis, Northwestern University, Dept. of ECE, 2003.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-Power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[CSPH02] Katherine Compton, Akshay Sharma, Shawn Phillips, and Scott Hauck. Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems. In *International Conference on Field Programmable Logic and Applications*, pages 59–68, 2002.

[CT00] Jae-Hwan Chang and Leandros Tassiulas. Energy Conserving Routing in Wireless Ad-hoc Networks. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM 2000. Proceedings*, volume 1, pages 22–31, 26.–30. March 2000.

[CTL+03] Fabio Campi, Mario Toma, Andrea Lodi, Andrea Cappelli, Roberto Canegallo, and Roberto Guerrieri. A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications. In *International Solid-State Circuits Conference*, 2003.

[Cyp12a] Cypress Semiconductor Corporation. *PSoC® 5LP Architecture TRM Technical Reference Manual*, 21. November 2012.

[Cyp12b] Cypress Semiconductor Corporation. *PSoC® Programmable System-on-Chip Technical Reference Manual (TRM)*, 26. July 2012.

[Cyp15a] Cypress Semiconductor Corporation. *PSoC 4 BLE Product Overview*, 2015. http://www.cypress.com/psoc4ble/ [2015-08-20].

[Cyp15b] Cypress Semiconductor Corporation. *PSoC 5LP Product Overview*, 2015. http://www.cypress.com/psoc5lp [2015-08-20].

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 16.–18. November 2004.

[DM95] Srinivas Devadas and Sharad Malik. A Survey of Optimization Techniques Targeting Low Power VLSI Circuits. In *32nd Conference on Design Automation, 1995. DAC '95*, pages 242–247, 1995.

[EAA13] Minar El-Aaasser and Mohamed Ashour. Energy Aware Classification for Wireless Sensor Networks Routing. In *15th International Conference on Advanced Communication Technology (ICACT)*, pages 66–71, January 2013.

[ECF96] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *The 6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[EHS05] Ken Eguro, Scott Hauck, and Akshay Sharma. Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement. In *42nd Design Automation Conference, Proceedings*, pages 439–444, 13.–17. June 2005.

[Ene14] Energy Micro AS, Sandakerveien 118, N-0484 Oslo, Norway. *EFM32GG Reference Manual*, 2. July 2014. Rev1.10 http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG-RM.pdf, http://www.silabs.com/support/pages/document-library.aspx?p=MCUs--32-bit&f=Giant%20Gecko [2015-08-20].

[Est02] Gerald Estrin. Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. *IEEE Annals of the History of Computing*, 24(4):3–9, October–December 2002.

[FCC+14] Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen, Mathew Hall, Bain Syrowik, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis. In *12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 120–129, August 2014.

[FH05] Michael J. Flynn and Patrick Hung. Microprocessor Design Issues: Thoughts on the Road Ahead. *Micro, IEEE*, 25(3):16–31, May–June 2005.

[GCS+06] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*, 2006(1):1–19, 2006.

[GDHG10] Johann Glaser, Markus Damm, Jan Haase, and Christoph Grimm. A Dedicated Reconfigurable Architecture for Finite State Machines. In *Reconfigurable Computing: Architectures, Tools and Applications, 6th International Symposium, ARC 2010*, volume LNCS 5992 of *Lecture Notes on Computer Science*, pages 122–133, Bangkok, Thailand, March 2010. Springer Berlin Heidelberg.

[GDHG11] Johann Glaser, Markus Damm, Jan Haase, and Christoph Grimm. TR-FSM: Transition-based Reconfigurable Finite State Machine. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(3):23:1–23:14, August 2011.

[GGHG11] Johann Glaser, Klaus Gravogl, Jan Haase, and Christoph Grimm. A Reconfigurable Architecture for Ultra-Low Power Wireless Sensors. *The Mediterranean Journal of Electronics and Communications (MEDJEC)*, 7(3):255–266, 2011.

[GHDG09] Johann Glaser, Jan Haase, Markus Damm, and Christoph Grimm. Investigating Power-Reduction for a Reconfigurable Sensor Interface. In *Proceedings of Austrochip 2009*, Graz, Austria, 7. October 2009.

[GHDG10] Johann Glaser, Jan Haase, Markus Damm, and Chritoph Grimm. A Novel Reconfigurable Architecture for Wireless Sensor Networks. In *Tagungsband zur Informationstagung Mikroelektronik 10*, pages 284–288, Vienna, Austria, 7.–8. April 2010. OVE.

[GHG10] Johann Glaser, Jan Haase, and Christoph Grimm. Designing a Reconfigurable Architecture for Ultra-Low Power Wireless Sensors. In Zabih Ghassemlooy and Wai Pang Ng, editors, *Proceedings of the Seventh IEEE, IET International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 343–347, Northumbria University, Newcastle upon Tyne, United Kingdom, 21.–23. July 2010.

[Gio95] Jean-Charles Giomi. Finite State Machine Extraction from Hardware Description Languages. In *Proceedings of the Eighth Annual IEEE International ASIC Conference and Exhibit*, pages 353–357, September 1995.

[Gir13] Olivier Girard. *openMSP430*, 24. February 2013. Rev. 1.13.

[GL00] Ajay Chandra V. Gummalla and John O. Limb. Wireless Medium Access Control Protocols. *IEEE Communications Surveys & Tutorials*, 3(2):2–15, Second Quarter 2000.

[GM02] Robert Graybill and Rami Melhem, editors. *Power Aware Computing*. Kluwer Academic / Plenum Publishers, 2002.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to High-Level Synthesis. *IEEE Design & Test*, 11(4):44–54, October 1994.

[Gre02] Jack Greenbaum. Reconfigurable Logic in SoC Systems. In *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*, pages 5–8, 2002.

[GW14] Johann Glaser and Clifford Wolf. Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In Jan Haase, editor, *Models, Methods, and Tools for Complex Chip Design*, volume 265 of *Lecture Notes in Electrical Engineering*, pages 201–221. Springer International Publishing, 2014.

[GWMM08] Johann Glaser, Daniel Weber, Sajjad A. Madani, and Stefan Mahlknecht. Power Aware Simulation Framework for Wireless Sensor Networks and Nodes. *EURASIP JES*, 2008.

[GZ97] Rajesh K. Gupta and Yervant Zorian. Introducing Core-Based System Design. *IEEE Design & Test of Computers*, 14(4):15–25, October–December 1997.

[GZR99] Varghese George, Hui Zhang, and Jan Rabaey. The Design of a Low Energy FPGA. In *International Symposium on Low Power Electronics and Design*, pages 188–193, 1999.

[Har01a] Reiner Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 642–649, 2001.

[Har01b] Reiner Hartenstein. Coarse Grain Reconfigurable Architecture. In *Proceedings of the Asia South Pacific Design Automation Conference*, pages 564–570, Yokohama, Japan, 2001.

[Hau05] Scott Hauck. The Totem Project. http://www.ee.washington.edu/faculty/hauck/Totem/ [2015-08-20], 2005. University of Washington, Dept. of EE.

[Hau15] Scott Hauck. personal communication, 2015.

[HBH+96] Reiner W. Hartenstein, Jürgen Becker, Michael Herz, Rainer Kress, and Ulrich Nageldinger. A Synthesis System for Bus-based Wavefront Array Architectures. In *Proceedings of International Conference on Application Specific Systems, Architectures and Processors ASAP 96*, pages 274–283, 12.–21. August 1996.

[HCE+06] Scott Hauck, Katherine Compton, Ken Eguro, Mark Holland, Shawn Phillips, and Akshay Sharma. Totem: Domain-Specific Reconfigurable Logic. Technical report, University of Washington, Dept. of EE, 2006.

[HD08] Scott Hauck and André DeHon, editors. *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann Publishers, Amsterdam, Boston, 2008.

[HDG+09] Jan Haase, Markus Damm, Johann Glaser, Javier Moreno, and Christoph Grimm. SystemC-based Power Simulation of Wireless Sensor Networks. In *Proceedings of the Forum of Design Languages (FDL)*, Sophia Antipolis, 2009.

[Hea03] Steve Heath. *Embedded Systems Design*. EDN Series for Design Engineers. Newnes, Oxford, 2nd edition, 2003.

[Hen99] Jörg Henkel. A Low Power Hardware/Software Partitioning Approach for Core-based Embedded Systems. In *36th Design Automation Conference*, pages 122–127, New Orleans, LA, USA, 21.–25. June 1999.

[HH04] Mark Holland and Scott Hauck. Automatic Creation of Reconfigurable PALs/PLAs for SoC. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *Proceedings of the 14th International Conference on Field Programmable Logic and Application, FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 536–545, Leuven, Belgium, 30. August – 1. September 2004. Springer.

[HH05] Mark Holland and Scott Hauck. Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC. In *International Conference on Field Programmable Logic and Applications*, pages 95–100, 24.–26. August 2005.

[HH07] Mark Holland and Scott Hauck. Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):291–295, February 2007.

[HHHN98] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Using the KressArray for Reconfigurable Computing. In *Proceedings of SPIE Vol. 3526, Conference on Configurable Computing: Technology and Applications*, pages 150–161, Boston, USA, 2.–3. November 1998.

[HHHN00a] Rainer Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC 2000*, 2000.

[HHHN00b] Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Synthesis and Domain-specific Optimization of KressArray-based Reconfigurable Computing Engines. In *Proceedings of the 2000 ACM/SIGDA 8th International Symposium on Field Programmable Gate Arrays*, 2000.

[HHHN00c] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Generation of Design Suggestions for Coarse-Grain Reconfigurable Architectures. In *10th International Workshop on Field Programmable Logic and Applications, FPL'2000*, pages 389–399, Villach, Austria, 27.–30. August 2000.

[HHN00] Reiner W. Hartenstein, Thomas Hoffmann, and Ulrich Nadeldinger. Design-Space Exploration of Low Power Coarse Grained Reconfigurable Datapath Array Architectures. In *PATMOS 2000 International Workshop – Power and Timing Modeling, Optimization and Simulation*, pages 118–128, Göttingen, Germany, 13.–15. September 2000.

[HK95] Reiner W. Hartenstein and Rainer Kress. A Datapath Synthesis System for the Reconfigurable Datapath Architecture. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC '95, CHDL '95, VLSI '95*, pages 479–484, 29. August–1. September 1995.

[HKR94] Reiner W. Hartenstein, Rainer Kress, and Helmut Reinig. A Dynamically Reconfigurable Wavefront Array Architecture for Evaluation of Expressions. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 404–414, 22.–24. August 1994.

[HLOK13] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. Hardware/-Software Co-Compilation with the Nymble System. In *8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, July 2013.

[HM99] R.W. Hartenstein and V. Milutinovic. Introduction to the Configware Minitrack – From Glue Logic Synthesis to Reconfigurable Computing Systems. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, 1999. HICSS-32*, volume Track 3, pages 326–326, 1999.

[HXS+13] Pei Huang, Li Xiao, Soroor Soltani, Matt W. Mutka, and Ning Xi. The Evolution of MAC Protocols in Wireless Sensor Networks: A Survey. *Communications Surveys Tutorials, IEEE*, 15(1):101–120, February 2013.

[Jos14] Anand Joshi. *Embedded Systems: Technologies and Markets*. BCC Research, Wellesley MA USA, September 2014. Report Code: IFT016E http://www.bccresearch.com/market-research/information-technology/embedded-systems-report-ift016e.html [2015-08-20].

[Kae08] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2008.

[Kat94] Randy H. Katz. *Contemporary Logic Design*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[KER05] Ian Kuon, Aaron Egier, and Jonathan Rose. Design, Layout and Verification of an FPGA using Automated Tools. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays FPGA'05*, pages 215–226, Monterey, CA, USA, 20.–22. February 2005.

[KJA+14] Fakhri Karray, Mohamed W. Jmal, Mohamed Abid, Mohammed S. BenSaleh, and Abdulfattah M. Obeid. A Review on Wireless Sensor Node Architectures. In *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, 26.–28. May 2014.

[KKOMB02] Ryan Kastner, Adam Kaplan, Seda Ogrenci-Memik, and Elaheh Bozorgzadeh. Instruction Generation for Hybrid Reconfigurable Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):605–627, October 2002.

[KMM14] Alp Kilic, Zied Marrakchi, and Habib Mehrez. A Top-Down Optimization Methodology for Mutually Exclusive Applications. *International Journal of Reconfigurable Computing*, 2014:18, 2014.

[KNNL05] Michael H. Kutner, Christopher J. Nachtsheim, John Neter, and William Li. *Applied Linear Statistical Models*. McGraw-Hill/Irwin, fifth edition, 2005.

[KR98] Eric Kusse and Jan Rabaey. Low-Energy Embedded FPGA Structures. In *International Symposium on Low Power Electronics and Design, Proceedings*, pages 155–160, 10.–12. August 1998.

[Kuo04] Ian Carlos Kuon. Automated FPGA Design, Verification and Layout. Master's thesis, University of Toronto, Graduate Department of Electrical and Computer Engineering, 2004.

[LAE06] Zhenyu Liu, Tughrul Arslan, and Ahmet T. Erdogan. An Embedded Low Power Reconfigurable Fabric For Finite State Machine Operations. In *International Symposium on Circuits and Systems, ISCAS*, pages 4374–4377, 2006.

[LAKL05] Zhenyu Liu, Tughrul Arslan, Sami Khawam, and Iain Lindsay. A High Performance Synthesisable Unsymmetrical Reconfigurable Fabric For Heterogeneous Finite State Machines. In *Proceedings of the ASP-DAC – Asia and South Pacific Design Automation Conference*, volume 1, pages 639–644, 18.–21. January 2005.

[Lar11] Rasmus Christian Larsen. Using hardware to save energy in MCU-based sensing applications. http://www.embedded.com/design/power-optimization/4230298/Using-hardware-to-save-energy-in-MCU-based-sensing-applications [2015-08-20], 2. November 2011. Energy Micro.

[Lat12a] Lattice Semiconductor Corporation. *iCE40™ HX Series Ultra Low-Power FPGA Family Data Sheet*, 1.32 edition, 3. October 2012.

[Lat12b] Lattice Semiconductor Corporation. *iCE40™ LP Series Ultra Low-Power FPGA Family Data Sheet*, 1.32 edition, 3. October 2012.

[LCB+06] Andrea Lodi, Andrea Cappelli, Massimo Bocchi, Claudio Mucci, Massimilano Innocenti, Claudia De Bartolomeis, Luca Ciccarelli, Roberto Giansante, Antonio Deledda, Fabio Campi, Mario Toma, and Roberto Guerrieri. XiSystem: A XiRisc-Based SoC With Reconfigurable IO Module. *IEEE Journal of Solid-State Circuits*, 41(1), January 2006.

[Lib15] The Liberty Library Modeling Standard. Synopsys, Inc., 2015. http://www.opensourceliberty.org/ [2015-08-20].

[LMP+05] Phil Levis, Sam Madden, Joe Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An Operating System for Sensor Networks. In Werner Weber, Jan M. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[LWj11] Li Li and Li Wei-jia. The analysis of data fusion energy consumption in WSN. In *International Conference on System Science, Engineering Design and Manufacturing Informatization*, volume 1, pages 310–313, Guiyang, 22.–23. October 2011.

[Mac04] Enrico Macii, editor. *Ultra Low-Power Electronics and Design*. Kluwer Academic Publishers, Dordrecht, 2004.

[Mah04] Stefan Mahlknecht. *Energy-Self-Sufficient Wireless Sensor Networks for Home and Building Environment*. PhD thesis, Institute of Computer Technology, Vienna University of Technology, 2004.

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, Boston, 2003.

[MB04] Stefan Mahlknecht and Michael Böck. CSMA-MPS: A Minimum Preamble Sampling MAC Protocol for Low Power Wireless Sensor Networks. In *Proceedings of the IEEE International Workshop on Factory Communication Systems. WFCS*, pages 73–80, 22.–24. September 2004.

[McE93] Kenneth McElvain. LGSynth93 Benchmark Set: Version 4.0. http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.cbl.ncsu.edu/CBL_Docs/lgs93.html [2015-08-20], 1993.

[Men10a] Menta SAS. eFPGA Core Datasheet Brief, June 2010. DSB01 (v2.0).

[Men10b] Menta SAS. eFPGA Creator Product Brief, June 2010. Rev. 1.

[Men12] Menta SAS. eFPGA Core Product Brief, June 2012. Rev. 3.

[MF14] Vijay Kumar Marrivagu and Antonio Rohit De Lima Fernandes. *PSoC 3, PSoC 4, and PSoC 5LP – Implementing Programmable Logic Designs with Verilog*. Cypress Semiconductor Corporation, 2014. AN82250.

[MGH05] Stefan Mahlknecht, Johann Glaser, and Thomas Herndl. PAWiS: Towards a Power Aware System Architecture for a SoC/SiP Wireless Sensor and Actor Node Implementation. In *Proceedings of 6th IFAC International Conference on Fieldbus Systems and their Applications*, pages 129–134, Puebla, Mexiko, 14.–15. November 2005.

[MHMS13] Georg Möstl, Richard Hagelauer, Gerhard Müller, and Andreas Springer. STEAM-Sim – Filling the Gap between Time Accuracy and Scalability in Simulation of Wireless Sensor Networks. In *Proceedings of the 8th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, pages 61–66, November 2013.

[Mic09] Microchip Technology Inc., 2355 West Chandler Blvd., Chandler, AZ 85224-6199. *PIC16F72X/PIC16LF72X Data Sheet*, March 2009. DS41341C.

[Mic14] Microchip Technology Inc., 2355 West Chandler Blvd., Chandler, AZ 85224-6199. *PIC16(L)F1503 Data Sheet*, 2014.

[MMR06] Stefan Mahlknecht, Sajjad Ahmad Madani, and Matthias Rötzer. Energy Aware Distance Vector Routing Scheme for Data Centric Low Power Wireless Sensor Networks. In *4th International IEEE Conference on Industrial Informatics INDIN'06*, 16.–18. August 2006.

[Mös15] Georg Möstl. *Enabling Accurate, Energy-Aware, and Scalable Simulation of Industrial Wireless Sensor Networks*. PhD thesis, Johannes Kepler Universität (JKU) Linz, February 2015.

[MPS98] Enrico Macii, Massoud Pedram, and Fabio Somenzi. High-Level Power Modeling, Estimation, and Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1061–1079, November 1998.

[MSB02] Monte Mar, Bert Sullam, and Eric Blom. An Architecture for a Programmable Mixed-Signal Device. In *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*, pages 55–58, 2002.

[MV07] Graeme Milligan and Wim Vanderbauwhede. Implementation of Finite State Machines on a Reconfigurable Device. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 386–396, Edinburgh, 5.–8. August 2007.

[Nag01] Ulrich Nageldinger. *Coarse-Grained Reconfigurable Architecture Design Space Exploration*. PhD thesis, University of Kaiserslautern CS department (Informatik), 2001.

[Nan14] NanoXplore. *NX-eFPGA*, 2014. http://www.nanoxplore.com/categories/17-efpga.html [2015-08-20].

[Nie98] Ralf Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, Boston, 1998.

[OEGS93] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm. In *30th Conference on Design Automation*, pages 31–37, 14.–18. June 1993.

[OMBKS01] Seda Ogrenci-Memik, Elaheh Bozorgzadeh, Ryan Kastner, and Majid Sarrafzadeh. SPS: A Strategically Programmable System. In *Proceedings of Reconfigurable Architecture Workshop*, April 2001.

[OMHG11] Jiong Ou, Farooq Muhammad, Jan Haase, and Christoph Grimm. A Technique for the Identification of Reconfigurable Resources of Flexible Communication Systems. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 256–263, June 2011.

[Ozk12] Murat Ozkan. Fundamentals Of Low-Power Design. http://electronicdesign.com/boards/fundamentals-low-power-design [2015-08-20], February 2012.

[PH02] Shawn A. Phillips and Scott Hauck. Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip. In *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2002.

[Phi04] Shawn A. Phillips. *Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip*. PhD thesis, University of Washington, Dept. of EE., 2004.

[PMKM11] Husain Parvez, Zied Marrakchi, Alp Kilic, and Habib Mehrez. Application-Specific FPGA Using Heterogeneous Logic Blocks. *ACM Transactions on Reconfigurable Technology and Systems*, 4(3):24:1–24:14, August 2011.

[PSH04] Shawn A. Phillips, Akshay Sharma, and Scott Hauck. Automating the Layout of Reconfigurable Subsystems Via Template Reduction. In *International Symposium on Field-Programmable Logic and Applications*, pages 857–861, 2004.

[PTD13] Kenneth Pocek, Russell Tessier, and Andre DeHon. Birth and Adolescence of Reconfigurable Computing: A Survey of the First 20 Years of Field-Programmable Custom Computing Machines. In *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–17, 28.–30. April 2013.

[Rab97] Jan M. Rabaey. Reconfigurable Processing: The Solution to Low-Power Programmable DSP. In *IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP-97*, volume 1, pages 275–278, Munich, Germany, 21.–24. April 1997.

[RAI+97] Jan M. Rabaey, Arthur Abnous, Yuji Ichikawa, Katsunori Seno, and Marlene Wan. Heterogeneous Reconfigurable Systems. In *IEEE Workshop on Signal Processing Systems, SIPS 97 – Design and Implementation*, pages 24–34, 3.–5. November 1997.

[RCN03] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits*. Prentice Hall, Upper Saddle River, 2003.

[RN00] M. Srikanth Rao and S. K. Nandy. Power Minimization Using Control Generated Clocks. In *37th Design Automation Conference, Proceedings*, pages 794–799, 2000.

[Roe04] Matthias Roetzer. Routing in energieautarken Funksensornetzwerken. Master's thesis, Vienna University of Technology, 2004.

[RP96] Jan M. Rabaey and Massoud Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, 1996.

[RPL96] Jan M. Rabaey, Massoud Pedram, and Paul E. Landman. Low Power Design Methodologies: Introduction. In Jan M. Rabaey and Massoud Pedram, editors, *Low Power Design Methodologies*, pages 1–18. Kluwer Academic Publishers, Norwell, 1996.

[RSZ04] C.S. Raghavendra, Krishna M. Sivalingam, and Taieb Znati. *Wireless Sensor Networks*. Kluwer Academic Publishers, July 2004.

[RV13] Juan Carlos Peña Ramos and Marian Verhelst. Flexible, Ultra-Low Power Sensor Nodes through Configurable Finite State Machines. In *8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, July 2013.

[Sar12] Rahul Sarpeshkar. Universal Principles for Ultra Low Power and Energy Efficient Design. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59(4):193–198, April 2012.

[Sch14] Martin Schmölzer. Design of a Flexible Data Path for Heterogeneous Coarse-Grain Reconfigurable Logic Circuits. Diploma thesis, Vienna University of Technology, 2014.

[Sil10] SiliconBlue Technologies Corporation. *iCE65 Ultra Low-Power mobileFPGA Family*, 26. May 2010.

[SP05] Vikram Singh Saun and Preeti Ranjan Panda. Extracting Exact Finite State Machines from Behavioral SystemC Descriptions. In *18th International Conference on VLSI Design*, pages 280–285, 3.–7. January 2005.

[SV02] Greg Stitt and Frank Vahid. Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. *IEEE Design & Test of Computers*, 19(6):36–43, 2002.

[SVKS01] Patrick Schaumont, Ingrid Verbauwhede, Kurt Keutzer, and Majid Sarrafzadeh. A Quick Safari Through the Reconfiguration Jungle. In *Proceedings Design Automation Conference*, pages 172–177, 2001.

[Swa97] Gitanjali Swamy. Formal Verification of Digital, Systems. In *Tenth International Conference on VLSI Design. Proceedings*, pages 213–217, 4.–7. January 1997.

[TCW+05] Tim J. Todman, George A. Constantinides, Steven J. E. Wilton, Oscar Mencer, Wayne Luk, and Peter Y. K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings of Computers and Digital Techniques*, 152(2):193–207, March 2005.

[Tew10] Girma S. Tewolde. Current Trends in Low-power Embedded Computing. In *IEEE International Conference on Electro/Information Technology (EIT)*, 20.–22. May 2010.

[Tex04] Texas Instruments, Dallas, Texas. *MSP430x11x2, MSP430x12x2 Mixed Signal Microcontroller*, August 2004. SLAS361D.

[Tex10a] Texas Instruments, Dallas, Texas. *MSP430F22x2, MSP430F22x4 Mixed Signal Microcontroller*, March 2010. SLAS504D.

[Tex10b] Texas Instruments, Dallas, Texas. *MSP430F543xA, MSP430F541xA Mixed Signal Microcontroller*, March 2010. SLAS655A.

[Tex15a] Texas Instruments, Dallas, Texas. *C2000 CLA C Compiler*, 2015. http://processors.wiki.ti.com/index.php/C2000_CLA_C_Compiler [2015-08-20].

[Tex15b] Texas Instruments, Dallas, Texas. *MSP430FR697x(1), MSP430FR687x(1), MSP430FR692x(1), MSP430FR682x(1) MCUs*, May 2015. SLASE23C.

[TSV07] George Theodoridis, Dimitrios Soudris, and Stamatis Vassiliadis. A Survey of Coarse-Grain Reconfigurable Architectures and CAD Tools. In Stamatis Vassiliadis and Dimitrios Soudris, editors, *Fine- and Coarse-Grain Reconfigurable Computing*, chapter 2, pages 89–149. Springer, 2007.

[Tua01] Tim Tuan. Platform-Based Design of Low-Energy, Hybrid Reconfigurable Fabric for Wireless Protocol Processing. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2001.

[Ull76] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[Uni14] Unified EFI, Inc. *Advanced Configuration and Power Interface Specification*, July 2014. Revision 5.1.

[Vah09] Frank Vahid. What is Hardware/Software Partitioning? *SIGDA Newsletter*, 39(6):1–1, June 2009.

[VPNH10] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, 2010.

[VS07] Stamatis Vassiliadis and Dimitrios Soudris, editors. *Fine- and Coarse-Grain Reconfigurable Computing*. Spring, 2007.

[VSPK04] Ingrid Verbauwhede, Patrick Schaumont, Christian Piguet, and Bart Kienhois. Architectures and Design Techniques for Energy Efficient Embedded DSP and Multimedia Processing. In Enrico Macii, editor, *Ultra Low-Power Electronics and Design*, pages 141–155. Kluwer Academic Publishers, Dordrecht, 2004.

[Wan01] Marlene Wan. *Design Methodology for Low Power Heterogeneous Reconfigurable Digital Signal Processors*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2001.

[WAWS03] James C.H. Wu, Victor Aken'Ova, Steven J.E. Wilton, and Resve Saleh. SoC Implementation Issues for Synthesizable Embedded Programmable Logic Cores. In *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003*, pages 45–48, 21–24 September 2003.

[WC91] Robert Al Walker and Raul Camposano. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, Boston, 1991.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1991.

[WG13] Clifford Wolf and Johann Glaser. Yosys – A Free Verilog Synthesis Suite. In *Austrochip Workshop on Microelectronics*, pages 47–52, Linz, Austria, October 2013.

[WGM07] Daniel Weber, Johann Glaser, and Stefan Mahlknecht. Discrete Event Simulation Framework for Power Aware Wireless Sensor Networks. In *Proceedings of the 5th International Conference on Industrial Informatics, INDIN 2007*, volume 1, pages 335–340, Vienna, Austria, 23.–26. July 2007.

[WGS+12] Clifford Wolf, Johann Glaser, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-Driven Interconnect Synthesis for Heterogeneous Coarse-Grain Reconfigurable Logic. In *Forum on Specification and Design Languages (FDL)*, pages 194–201, Vienna, Austria, 18.–20. September 2012.

[WKW+05] Steven J. E. Wilton, Noha Kafafi, James C. H. Wu, Kimberly A. Bozman, Victor O. Aken'Ova, and Resve Saleh. Design Considerations for Soft Embedded Programmable Logic Cores. *IEEE Journal of Solid-State Circuits*, 40(2):485–497, February 2005.

[Wol03] Wayne Wolf. A Decade of Hardware/Software Codesign. *Computer*, 36(4):38–43, April 2003.

[Wol15a] Clifford Wolf. *Yosys Manual*, 2015.

[Wol15b] Clifford Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/ [2015-08-20], 2015.

[WZG+01] Marlene Wan, Hui Zhang, Varghese George, Martin Benes, Arthur Abnous, Vandana Prabhu, and Jan Rabaey. Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System. *Journal of VLSI Signal Processing*, 28(1-2):47–61, May–June 2001.

[Xil07] Xilinx, Inc. *Virtex-4 Family Overview*, 23. January 2007. DS112 (v2.0).

[Xil08] Xilinx, Inc. *ML405 Evaluation Platform User Guide*, 10. March 2008. UG210 (v1.5.1).

[Xil09] Xilinx, Inc. *Virtex-4 FPGA Configuration User Guide*, 9. June 2009. UG071 (v1.11).

[Xil14a] Xilinx, Inc. *7 Series FPGAs Configurable Logic Block – User Guide*, 17. November 2014. UG474 (v1.7).

[Xil14b] Xilinx, Inc. *7 Series FPGAs Packaging and Pinout – Product Specification*, 13. November 2014. UG475 (v1.13).

[Xil14c] Xilinx, Inc. *Vivado Design Suite, User Guide, High-Level Synthesis*, 1. October 2014. UG902 v2014.3.

[Xil14d] Xilinx, Inc. *Zynq-7000 AP SoCs Product Table*, 2014.

[Xil15a] Xilinx, Inc. *7 Series FPGAs Configuration – User Guide*, 24. June 2015. UG470 (v1.10).

[Xil15b] Xilinx, Inc. *7 Series FPGAs Overview – Product Specification*, 17. December 2015. DS180 (v1.16.1).

[Xil15c] Xilinx, Inc. *Zynq-7000 Family Use Cases and Markets*, 2015. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/use-cases-and-markets.html [2015-08-20].

[Xil15d] Xilinx, Inc. *Zynq UltraScale+ MPSoC Application Spaces*, 2015. http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc/application-spaces.html [2015-08-20].

[Xil15e] Xilinx, Inc. *Zynq UltraScale+ MPSoC Product Selection Guide*, 2015.

[YW04] Andy Yan and Steven J. E. Wilton. Sequential synthesizable embedded programmable logic cores for system-on-chip. In *Proceedings of the IEEE 2004 Custom Integrated Circuits Conference, 2004*, pages 435–438, 3.–6. October 2004.

[ZPG+00] Hui Zhang, Vandana Prabhu, Varghese George, Marlene Wan, Martin Benes, Arthur Abnous, and Jan M. Rabaey. A 1V Heterogeneous Reconfigurable Processor IC for Wireless Baseband Applications. *IEEE Journal of Solid-State Circuits*, 35(11):1697–1704, November 2000.

[ZWH03] Nian Zhang, Donald C. Wunsch, II, and Frank Harary. The subcircuit extraction problem. *IEEE Potentials*, 22(3):22–25, August 2003.

Abbreviations

ACPI	advanced configuration and power interface
ADC	analog to digital converter
AIG	and-inverter graph
ALU	arithmetic and logical unit
ASIC	application specific integrated circuit
ASIP	application specific instruction-set processor
AST	abstract syntax tree
ATA	advanced technology attachment
BDD	binary decision diagram
CAN	controller area network
cASIC	"configurable ASIC" variant of Totem project
CDC	USB communications device class
CDFG	control and data flow graph
CELP	code-excited linear prediction speech coding
CFG	control flow graph
CLA	Texas Instruments "Control Law Accelerator"
CLB	complex logic block
CLC	Microchip "Configurable Logic Cell" modules
CMOS	complementary metal-oxide-semiconductor
CPLD	complex programmable logic device
CPU	central processing unit
CRC	cyclic redundancy check
CSMA-MPS	carrier sense multiple access with minimum preamble sampling
D-FF	D-type flip-flop
DFG	data flow graph
DMA	direct memory access
DRC	design rule check

DVFS	dynamic voltage and frequency scaling
EDA	electronic design automation
EEPROM	electrically erasable PROM
eFPGA	embedded FPGA
ERC	electrical rule check
FFT	fast Fourier transform
FPGA	field programmable gate array
FSM	finite state machine
FSM+D	FSM with datapath
FU	functional unit
GPIO	general purpose input and output
GUI	graphical user interface
HDL	hardware description language
HLS	high-level synthesis
HW	hardware
HW/SW	hardware/software
I^2C	Inter-Integrated Circuit serial bus
IO	input and output
IP	intellectual property
IPG	TR-FSM input pattern gate
ISA	instruction set architecture
ISM	TR-FSM input switching matrix
ISR	interrupt service routine
LESENSE	Energy Micro "Low-Energy Sensor Interface"
LUT	look-up table
LVS	layout vs. schematic
MAC	media access control
MCU	microcontroller unit
MIN	multistage interconnect network
MOS	metal-oxide-semiconductor
MOSFET	metal-oxide-semiconductor field-effect transistor
MUX	multiplexer
NMOS	n-channel MOSFET
NPLC	number of power-line cycles
NRE	non-recurring engineering (cost)
NSR	TR-FSM next state register
OPR	TR-FSM output pattern register

P&R place and route
PCB printed circuit board
PCI peripheral component interconnect bus
PiCoGA XiSystem “Pipelined Configurable Gate Array”
PLA Analog Devices “Programmable Logic Array”
PLA programmable logic array
PLB Lattice/SiliconBlue iCE65/iCE40 FPGA “Programmable Logic Block”
PMOS p-channel MOSFET
PROM programmable read-only memory
PSoC Cypress “Programmable System-on-Chip”
PWM pulse-width modulation

RAM random access memory
RaPiD “Reconfigurable Pipelined Datapath” project
rDPA KressArray “reconfigurable datapath architecture”
rDPU KressArray “reconfigurable datapath unit”
RF radio frequency
RLB Tryptich FPGA “routing and logic block”
ROCCC Riverside Optimizing Compiler for Configurable Computing
ROM read-only memory
RTC real time clock
RTL register transfer level

SAIF switching activity interchange format
SATA serial ATA
SD secure digital card
SDF standard delay format
SDR software defined radio
SoC system on chip
SPEF standard parasitic exchange format
SPI Serial Peripheral Interface bus
SPS “Strategically Programmable System” project
SRAM static RAM
SSG TR-FSM state selection gate
STA static timing analysis
SW software

Tcl tool command language
TI Texas Instruments Incorporated
TR TR-FSM transition row
TR-FSM Transition-Based Reconfigurable FSM

UART universal asynchronous receiver and transmitter
UDB PSoC “universal digital block”
USART universal synchronous/asynchronous receive transmit

USB	universal serial bus
VCD	value change dump
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuit
VLSI	very-large-scale integration
VPB	SPS project “Versatile Parameterizable Block”
WSN	wireless sensor network

Curriculum Vitae

Name: Johann Glaser

Date of birth: June 1, 1979

Nationality: Austria

Address: Ebenhochstraße 8/EG/10, 4020 Linz

EMail: Johann.Glaser@gmx.at

Education:

1998:	School leaving graduation with distinction, HTBLA Linz Paul-Hahn-Straße, Electrical Engineering
1998–2004:	Academic Studies: Electrical Engineering, specialization on Embedded Systems, masters degree with distinction, Vienna University of Technology
2004–2012:	Academic Studies: Psychology, specialization on clinical psychology and social psychology, masters degree, University of Vienna
2004–2015:	PhD in Electrical Engineering, specialization on digital logic design, Vienna University of Technology

Employment:

2005–2013:	Project assistant at the Institute of Computer Technology, Vienna University of Technology, WSN simulation framework, ASIC development, publications, supervision of students, organization of the "Invent a Chip" competition for pupils
2013–2015:	Project assistant at the Research Institute for Integrated Circuits (RIIC), Johannes Kepler University Linz, organization of "Invent a Chip"
2015:	R&D Engineer at the Linz Center of Mechatronics GmbH, real-time systems, software engineering, embedded systems

Publications

Diploma Thesis in Electrical Engineering

- **Johann Glaser**. *Burst Mode Receiver for Passive Optical Networks*. Diploma thesis, Vienna University of Technology, 2004.

- **Johann Glaser**. *Burst Mode Receiver for Passive Optical Networks*. VDM-Verlag, ISBN 363901720X, 2008. (extended version of diploma thesis published as book)

Simulation of Wireless Sensor Networks

- Stefan Mahlknecht, **Johann Glaser**, and Thomas Herndl. PAWiS: Towards a Power Aware System Architecture for a SoC/SiP Wireless Sensor and Actor Node Implementation. In *Proceedings of 6th IFAC International Conference on Fieldbus Systems and their Applications*, pages 129–134, Puebla, Mexiko, 14.–15. November 2005.

- Daniel Weber, **Johann Glaser**, and Stefan Mahlknecht. Discrete Event Simulation Framework for Power Aware Wireless Sensor Networks. In *Proceedings of the 5th International Conference on Industrial Informatics, INDIN 2007*, volume 1, pages 335–340, Vienna, Austria, 23.–26. July 2007.

- **Johann Glaser**, Daniel Weber, Sajjad A. Madani, and Stefan Mahlknecht. Power Aware Simulation Framework for Wireless Sensor Networks and Nodes. *EURASIP JES*, 2008.

- Jan Haase, Markus Damm, **Johann Glaser**, Javier Moreno, and Christoph Grimm. SystemC-based Power Simulation of Wireless Sensor Networks. In *Proceedings of the Forum of Design Languages (FDL)*, Sophia Antipolis, 2009.

Design Methodology for Custom Reconfigurable Logic Architectures

- **Johann Glaser**, Jan Haase, Markus Damm, and Christoph Grimm. Investigating Power-Reduction for a Reconfigurable Sensor Interface. In *Proceedings of Austrochip 2009*, Graz, Austria, 7. October 2009.

- **Johann Glaser**, Markus Damm, Jan Haase, and Christoph Grimm. A Dedicated Reconfigurable Architecture for Finite State Machines. In *Reconfigurable Computing: Architectures, Tools and Applications, 6th International Symposium, ARC 2010*, volume LNCS 5992 of *Lecture Notes on Computer Science*, pages 122–133, Bangkok, Thailand, March 2010. Springer Berlin Heidelberg.

- **Johann Glaser**, Jan Haase, Markus Damm, and Chritoph Grimm. A Novel Reconfigurable Architecture for Wireless Sensor Networks. In *Tagungsband zur Informationstagung Mikroelektronik 10*, pages 284–288, Vienna, Austria, 7.–8. April 2010. OVE.

- **Johann Glaser**, Jan Haase, and Christoph Grimm. Designing a Reconfigurable Architecture for Ultra-Low Power Wireless Sensors. In Zabih Ghassemlooy and Wai Pang Ng, editors, *Proceedings of the Seventh IEEE, IET International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP)*, pages 343–347, Northumbria University, Newcastle upon Tyne, United Kingdom, 21.–23. July 2010.

- **Johann Glaser**, Markus Damm, Jan Haase, and Christoph Grimm. TR-FSM: Transition-based Reconfigurable Finite State Machine. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(3):23:1–23:14, August 2011.
- **Johann Glaser**, Klaus Gravogl, Jan Haase, and Christoph Grimm. A Reconfigurable Architecture for Ultra-Low Power Wireless Sensors. *The Mediterranean Journal of Electronics and Communications (MEDJEC)*, 7(3):255–266, 2011.
- Clifford Wolf, **Johann Glaser**, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-Driven Interconnect Synthesis for Heterogeneous Coarse-Grain Reconfigurable Logic. In *Forum on Specification and Design Languages (FDL)*, pages 194–201, Vienna, Austria, 18.–20. September 2012.
- Clifford Wolf and **Johann Glaser**. Yosys – A Free Verilog Synthesis Suite. In *Austrochip Workshop on Microelectronics*, pages 47–52, Linz, Austria, October 2013.
- **Johann Glaser** and Clifford Wolf. Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In Jan Haase, editor, *Models, Methods, and Tools for Complex Chip Design*, volume 265 of *Lecture Notes in Electrical Engineering*, pages 201–221. Springer International Publishing, 2014.

Psychology

- **Johann Glaser**. *fMRI Artifact Correction in EEG and EMG Data.* Diploma thesis, University of Vienna, 2012.
- **Johann Glaser**. *fMRI Artifact Correction in EEG and EMG Data: Introducing FACET: A Flexible Artifact Correction and Evaluation Toolbox for EEG/fMRI Data.* LAP Lambert Academic Publishing, ISBN 3659376078, 2013. (extended version of diploma thesis published as book)
- **Johann Glaser**, Roland Beisteiner, Herbert Bauer, and Florian Ph.S Fischmeister. FACET – a "Flexible Artifact Correction and Evaluation Toolbox" for Concurrently Recorded EEG/fMRI Data. *BMC Neuroscience 14:138*, 2013.
- **Johann Glaser**, Veronika Schöpf, Roland Beisteiner, Herbert Bauer, and Florian Ph.S Fischmeister. Optimum Gradient Artifact Removal from EEG-Data using FACET. *Journal of the Neurological Sciences, 333, E622*, 2013.