

Oliver Schliebusch
Heinrich Meyr
Rainer Leupers

Optimized ASIP Synthesis from Architecture Description Language Models

 Springer

OPTIMIZED ASIP SYNTHESIS FROM ARCHITECTURE
DESCRIPTION LANGUAGE MODELS

Optimized ASIP Synthesis from Architecture Description Language Models

by

OLIVER SCHLIEBUSCH

CoWare Inc, Aachen, Germany

HEINRICH MEYR

RWTH Aachen University, Germany

and

RAINER LEUPERS

RWTH Aachen University, Germany

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-10 1-4020-5685-0 (HB)
ISBN-13 978-1-4020-5685-7 (HB)
ISBN-10 1-4020-5686-9 (e-book)
ISBN-13 978-1-4020-5686-4 (e-book)

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

www.springer.com

Printed on acid-free paper

All Rights Reserved

© 2007 Springer

No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

**Dedicated to
my wife Frauke and
my daughter Svenja.**

Contents

Dedication	v
Foreword	xi
Preface	xiii
1. INTRODUCTION	1
1.1 From ASIC to ASIP	1
1.2 Heterogeneous Architectures: Computational Performance vs. Flexibility	4
1.3 Challenges of ASIP Design	7
1.4 Organization of This Book	9
2. ASIP DESIGN METHODOLOGIES	11
2.1 ADL based ASIP Design	12
2.2 (Re)Configurable Architectures	17
2.3 Hardware Description Languages and Logic Representation	19
2.4 Motivation of This Work	20
3. ASIP DESIGN BASED ON LISA	23
3.1 Design Space Exploration	23
3.2 Software Tools Generation	25
3.3 System Simulation and Integration	28
4. A NEW ENTRY POINT FOR ASIP IMPLEMENTATION	29
4.1 Acceptance Criteria for an Automatic ASIP Implementation	30
4.2 From Architecture to Hardware Description	32

5. LISA FRONTEND	41
5.1 Resource Section	42
5.2 LISA Operations	44
5.3 LISA Operation Graph	45
5.4 Representing Exclusiveness in Conflict and Compatibility Graphs	55
5.5 Exclusiveness Information on the Level of LISA Operations	59
5.6 Exclusiveness Information on the Behavioral Level	64
6. INTERMEDIATE REPRESENTATION	67
6.1 Unified Description Layer	67
6.2 Optimization Framework	73
6.3 VHDL, Verilog and SystemC Backend	74
7. OPTIMIZATIONS BASED ON EXPLICIT ARCHITECTURAL INFORMATION	77
7.1 Basic DFG based Optimizations	78
7.2 Resource Sharing	84
7.3 Dependency Minimization	96
7.4 Decision Minimization	98
8. CONFIGURABLE PROCESSOR FEATURES	101
8.1 Processor Features	101
8.2 JTAG Interface and Debug Mechanism Generation	103
8.3 Adaptability of Synthesis Framework	114
9. CASE STUDY: AN ASIP FOR TURBO DECODING	117
9.1 Turbo Decoding based on Programmable Solutions	117
9.2 The Turbo Decoding Principle	120
9.3 The Max-LOGMAP Algorithm	120
9.4 Memory Organization and Address Generation	123
9.5 Instruction-Set and Data-Path Implementation	128
9.6 Instruction Schedule	128
9.7 Pipeline Structure	131
9.8 Results	132
9.9 Concluding Remarks	133

10. CASE STUDIES: LEGACY CODE REUSE	135
10.1 Levels of Compatibility	135
10.2 The Motorola 68HC11 Architecture	137
10.3 The Infineon Technologies ASMD	144
11. SUMMARY	149
Appendices	153
A Case Studies	153
A.1 ICORE	154
A.2 LT Architecture Family	156
B CFG and DFG Conversions	161
B.1 CFG to DFG Conversion	161
B.2 DFG to CFG Conversion	164
C Debug Mechanism	167
C.1 Advanced Features of Debug Mechanisms	167
List of Figures	171
List of Tables	175
References	177
About the Authors	189
Index	191

Foreword

We are presently observing a paradigm change in designing complex SoC as it occurs roughly every twelve years due to the exponentially increasing number of transistors on a chip. The present design discontinuity, as all previous ones, is characterized by a move to a higher level of abstraction. This is required to cope with the rapidly increasing design costs. While the present paradigm change shares the move to a higher level of abstraction with all previous ones, there exists also a key difference.

For the first time advances in semiconductor manufacturing do not lead to a corresponding increase in performance. At 65 nm and below it is predicted that only a small portion of performance increase will be attributed to shrinking geometries while the lion share is due to innovative processor architectures. To substantiate this assertion it is instructive to look at major drivers of the semiconductor industry: wireless communications and multimedia. Both areas are characterized by an exponentially increasing demand of computational power to process the sophisticated algorithms necessary to optimally utilize the limited resource bandwidth. The computational power cannot be provided in an energy-efficient manner by traditional processor architectures, but only by a massively parallel, heterogeneous architecture.

The promise of parallelism has fascinated researchers for a long time; however, in the end the uniprocessor has prevailed. What is different this time? In the past few years computing industry changed course when it announced that its high performance processors would henceforth rely on multiple cores. However, switching from sequential to modestly parallel computing will make programming much more difficult without rewarding this effort with dramatic improvements.

A valid question is: Why should massive parallel computing work when modestly parallel computing is not the solution? The answer is:

It will work only if one restricts the application of the multiprocessor to a class of applications. In wireless communications the signal processing task can be naturally partitioned and is (almost) periodic. The first property allows to employ the powerful technique of task level parallel processing on different computational elements. The second property allows to temporally assign the task by an (almost) periodic scheduler, thus avoiding the fundamental problems associated with multithreading. The key building elements of the massively parallel SoC will be clusters of application specific processors (ASIP) which make use of instruction-level parallelism, data-level parallelism and instruction fusion.

This book describes the automatic ASIP implementation from the architecture description language LISA employing the tool suite "Processor Designer" of CoWare. The single most important feature of the approach presented in this book is the efficient ASIP implementation while preserving the full architectural design space at the same time. This is achieved by introducing an intermediate representation between the architectural description in LISA and the Register Transfer Level commonly accepted as entry point for hardware implementation. The LISA description allows to explicitly describing architectural properties which can be exploited to perform powerful architectural optimizations. The implementation efficiency has been demonstrated by numerous industrial designs.

We hope that this book will be useful to the engineer and engineering manager in industry who wants to learn about the implementation efficiency of ASIPs by performing architectural optimizations. We also hope that this book will be useful to academia actively engaged in this fascinating research area.

Heinrich Meyr, September 2006

Preface

The work presented in this book reflects my PhD thesis accomplished at the Institute for Integrated Signal Processing Systems (ISS) at RWTH Aachen University. The core topic of RTL hardware model generation is only one out of many complex aspects in the field of design automation for Application Specific Instruction-Set Processor (ASIP) development. While several approaches focus on single aspects only, a few methodologies target the whole design flow and have gained acceptance in industry. One of these bases on the architecture description language LISA, which was initially defined to enable the generation of fast instruction-set simulators. Although not foreseen in the beginning, LISA was nevertheless the starting point for a research project that consequently aimed at the needs posed by commercial ASIP design. The development began with simulator generation, then moved on to software tools generation and system integration and has now arrived at C-Compiler generation and RTL model generation. As the RTL model generation was one of the latest aspects targeted, it was a huge challenge to bridge the discrepancy between already commercialized parts of the LISA tool suite and the latest research results. Therefore, the distance between initial research and the final product was extremely short, providing challenge and motivation simultaneously.

I would like to thank Prof. Heinrich Meyr for his support and the numerous inspiring discussions. It was really motivating working in an open and interdisciplinary atmosphere. Also, I would like to express my thanks to Prof. Tobias Noll for his great interest in my work and for his commitment as secondary advisor.

The concepts of a synthesis environment as described in this book can hardly be implemented by a single person. At show-time, shortly before product release, 11+ full time PhD and undergraduate students were working on the project with enthusiasm. Each one of them made it

become a huge success. In particular, I must thank Anupam Chattopadhyay, David Kammler and Martin Witte for their outstanding contribution.

Further, I would like to thank my wife Frauke for her support and understanding during the whole project and especially while writing this book. Last but not least, I would like to give special thanks to my parents Udo and Monika for making my academic profession possible.

This book provides a snapshot not only from a technical point of view, but also a snapshot in time. There are many challenging questions in the emerging field of ESL design. Academic readers might be interested in visiting the web pages of the ISS at RWTH Aachen University (www.iss.rwth-aachen.de), while readers from industry might consider visiting CoWare's web pages (www.coware.com) for latest product information.

OLIVER SCHLIEBUSCH, SEPTEMBER 2006

Chapter 1

INTRODUCTION

The increasing utilization and steadily enhancing functionality of digital devices is noticeable in everyday life. This development is enabled by Systems-on-Chips (SoCs) that comprise various computational building blocks, memories and on-chip communication. However, a crisis in SoC design is posed by the conflicting demands for performance, energy efficiency and flexibility. Thus, hardware designers and the Electronic Design Automation (EDA) industry face extreme pressure to find advanced architectures and appropriate design approaches. The importance of shifting away from architectures with fixed functionality, namely Application-Specific Integrated Circuits (ASICs), to more flexible solutions is discussed in this chapter.

1.1 From ASIC to ASIP

There are four major reasons for today's ASIC design challenges [1]. First, so-called deep-submicron effects are not considered on the abstraction level used for architectural specification. With regard to the interconnect, for example, both an increased influence of the signal propagation delay and a compromised signal integrity due to crosstalk are not considered adequately by traditional design methodologies. An architecture implementation below Register Transfer Level (RTL) is required to take these effects into account and to achieve an accurate estimation of the physical behavior of the architecture. This implies the necessity for several costly design iterations. Second, larger chips can be implemented due to the advances in semiconductor manufacturing. The ITRS [2] expects a growth from 697 million transistors per chip today up to 7,022 million transistors per chip in 2015. As early as 2003, it was predicted that up to 250 million gates per chip will become feasible in 2005,

although it was expected that only 50 million will be used [3]. Third, not only the amount of gates, but also the complexity of the designs is increasing. Analog as well as digital parts are integrated on a single chip and, moreover, different functionalities are combined. Last but not least, the shrinking time-to-market as well as the shortened time-in-market augment the challenges in ASIP design.

Due to the above-mentioned reasons, SoC building blocks need to be designed for reuse and adaptation. Contrary to this requirement, ASICs provide only a fixed function and not the required flexibility. However, ASICs are required in SoC designs due to their high performance as well as energy efficiency.

As shown in figure 1.1, an ASIC can be divided into two components - *data-path* and *control-path*. The data-path covers computational units, storage elements and multiplexers to route the data. The control-path basically consists of a Finite State Machine (FSM) that controls the data-path. Thus, the FSM represents the application dependent schedule of operations. Due to its complexity and irregularity, design errors are more likely to appear in the control-path than in the data-path of the architecture [4]. Although the separation between data-path and control-path is theoretically existent, unfortunately, traditional ASIC design combines these components without a clear interface. This certainly affects the initial specification, but even more the reuse and adaptation of the ASIC. Every variation of the ASIC has to be completely verified even if only the control-path is changed, as shown on the left of figure 1.1. Moreover, the commonly used design entry point is a hardware description on RTL, which already covers numerous hardware details unnecessary for a *functional* verification. However, these hardware details cause a slow simulation performance and render verification extremely costly.

Concluding from the previous paragraphs, a separation of concerns must be achieved. Hot spots of the application must be accelerated by an application-specific and optimized data-path while its control must be flexible and easy to verify. Processors partially address this requirement by being programmable and thus flexible with regard to the schedule of operations. This also implies a separation between fixed data-path and flexible control. Here, errors which ensue from the complex application dependent control can be easily eliminated by software updates. Furthermore, since application software can be executed by more abstract and therefore faster simulators compared to RTL hardware simulators, the time for functional verification is reduced. However, due to their generality, off-the-shelf General Purpose Processors (GPPs), Digital Signal

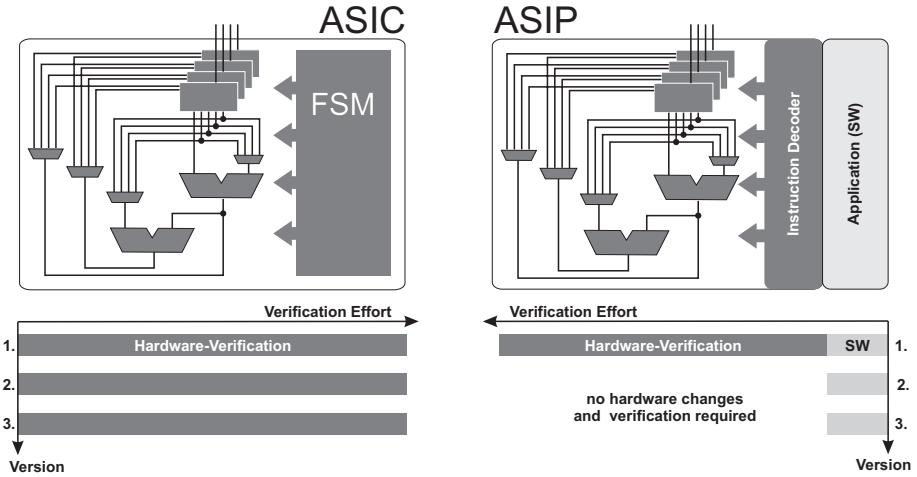


Figure 1.1. Comparison of ASIC and ASIP verification effort

Processors (DSPs) or Microcontrollers (μ Cs) can neither achieve the performance nor the energy efficiency of ASICs.

The inevitable consequence is the use of *Application-Specific Instruction-Set Processors* (ASIPs) which incorporate the flexibility of processors and high performance and energy efficiency of ASICs [5][6][7]. The software, more precisely the sequence of instructions, represents the state transitions of the finite state machine mentioned above. Furthermore, the software can be written and verified independently from the ASIP implementation, which enables an efficient reuse of the ASIP. The orthogonal structure of an ASIP is illustrated on the right of figure 1.1, emphasizing the aspect of verification for the first instantiation and the subsequent reuse (bottom right corner of figure 1.1).

Obviously, not only the hardware implementation of an ASIP but also software development tools are required. As long as they have to be developed manually, the gain with regard to the verification time is neutralized. Therefore, a new ASIP design methodology is indispensable, to preserve the architectural advantages of ASIPs and to reduce the overall design effort.

The complexity of ASIP design results from the fundamental different design tasks and the optimization of the processor concerning a given application. This topic is discussed in detail in section 1.3.

Within the scope of this book the focus is set on application-specific computational elements in SoCs. However, various other aspects of SoC design are also crucially related to the design complexity, such as the on-chip communication. Numerous publications describe the SoC design crisis, e.g. [3], [4], [8], [9], [10] and [11].

This crisis in SoC design currently motivates new heterogeneous architectures, which represent alternatives to ASIPs. A brief overview of the different solutions is given in the following section.

1.2 Heterogeneous Architectures: Computational Performance vs. Flexibility

Heterogeneous architectures combine existing architecture types to satisfy the demands of SoC design. The available architecture types differ regarding performance, energy efficiency and flexibility [12][13], as depicted in figure 1.2.

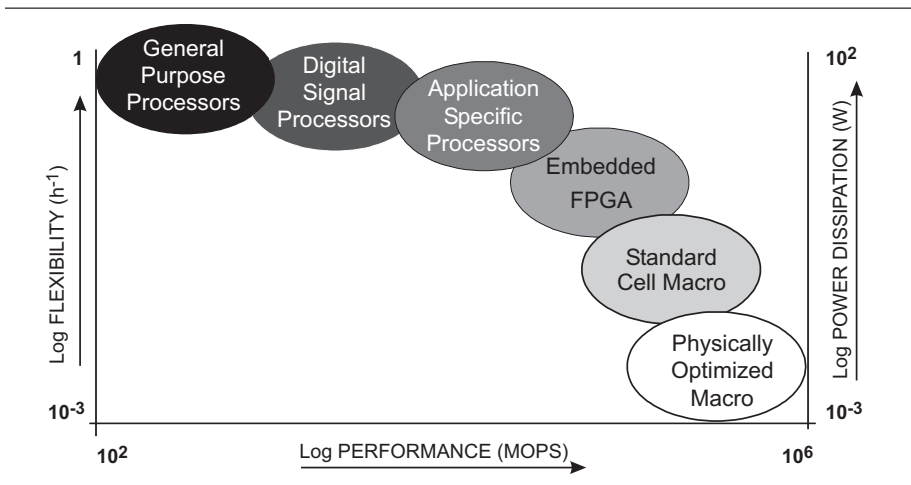


Figure 1.2. Computational building blocks in SoCs. Source: T. Noll, EECS [14]

Off-the-shelf processors provide the most flexibility at the expense of performance and energy efficiency. The processors have fixed instruction-sets, data-paths and interfaces. Software development tools, such as C-compiler, assembler, linker, simulator and debugger are used for application software development. In contrast to this, ASICs provide the best performance at the expense of flexibility. ASIC development, usually started on RTL, requires gate-level synthesis tools, hardware simulators, place and route tools, etc. ASIPs as well as Field Programmable

Gate Arrays (FPGAs) enable a trade-off between flexibility and performance. FPGAs provide flexibility by being reconfigurable to new hardware specifications. Due to the regular structure of FPGA building blocks, logic blocks with a regular structure can be mapped more efficiently to FPGAs than logic with an irregular structure. Therefore, the efficiency of the implementation strongly depends on the architecture.

The architectures types mentioned above are combined to trade off design characteristics in heterogenous SoC design. Architectures currently discussed in academia and industry are presented in the following and summarized in figure 1.3.

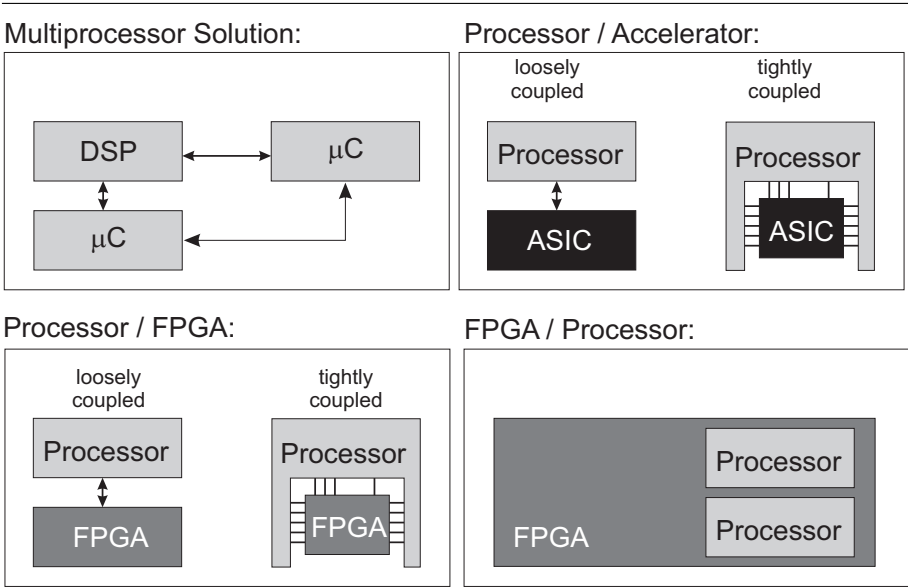


Figure 1.3. Heterogeneous architecture types as SoC building blocks

MultiProcessor (MP) solutions receive significant impetus from the requirement for flexible solutions in embedded systems. They benefit from the high availability of off-the-shelf processors, such as GPPs, μC s and DSPs. The required design tools are software development tools, such as C-compiler, assembler, linker and simulator. A challenging task

is to consider the data and control dependencies while developing multi-tasking/multi-threading applications¹.

However, one of the drawbacks of this approach is that the system performance does not scale linearly with the number of processors, as the on-chip communication has a substantial impact on the overall system performance [16]. Therefore, the design challenges are moved from the computational building blocks to the on-chip communication. Moreover, the potential for application-specific optimization of the computational blocks is completely neglected, which leads to suboptimal solutions.

A **Processor with a loosely/tightly coupled accelerator** addresses the need to speed up certain computational tasks. The data traffic between processor and accelerator is dependent on the given application. A *loosely coupled* accelerator may be used by multiple units in the SoC and accessed, for example, via a bus. This interface is suitable if data transfer rates are low and the computational tasks are independent of other computations executed in the meantime. The *tightly coupled* accelerator is preferable if a computational task depends on the overall control-flow of the application or a heavy data transfer between processor core and accelerator exists. These accelerators are embedded into the processor structure and directly linked to the temporal behavior of the processor. Often the accelerators are triggered by special purpose instructions. The major drawbacks of utilizing accelerators are the generality of the processor, the fixed interface between processor and accelerator and the fixed function of the accelerator. The above-mentioned drawbacks of ASIC design are valid for the accelerator design as well.

A **Processor with loosely/tightly coupled FPGA** partially eliminates the drawbacks of the processor-accelerator coupling. The fixed ASIC accelerator is replaced by an FPGA. The reconfigurable device may be either loosely or tightly coupled. Here, both components provide post-fabrication flexibility. However, the fixed interface between processor and reconfigurable device clearly restricts the architectural design space and prohibits possible optimization of the architecture with regard to a given application.

FPGAs with integrated processors are offered and promoted by FPGA vendors. Here, a particular number of GPPs and peripherals are integrated into the FPGA. Contemporary approaches also integrate

¹Multi-Tasking/Multi-Threading describes the ability to simultaneously execute several tasks/threads on MP-systems or pseudo-simultaneously by sharing the processing power of a single processor. The different terms can be distinguished by the definition given in [15]: A *task* is a semi-independent portion of the application that carries out a specific duty. A *process* is a completely independent program that has its own address space, while a *thread* is a semi-independent program segment that executes within a process.

special hardware as, for example, DSP engines into the FPGA chip. However, even though a high flexibility is provided, off-the-shelf solutions are hardly application-specific and potentially lead to suboptimal solutions.

The heterogeneous solutions mentioned above compete with ASIPs with regard to flexibility, performance, power dissipation and design efficiency. The particular characteristics of ASIPs are explained in the following.

Application-Specific Instruction-Set Processors incorporate the flexibility of programmable solutions and the high performance and energy efficiency of ASICs. The separation between processor and accelerator is completely removed, thus eliminating the drawbacks introduced by interfaces. However, eliminating the interface leads to a more complex design process, since the previously separated application software development and accelerator implementation are now merged into one design process. Furthermore, ASIP-specific application software development tools are required. Thus, the design of ASIPs requires a new design methodology reducing the overall design complexity, as discussed in the following section.

1.3 Challenges of ASIP Design

The development of a processor is a complex task, involving several development phases, multiple design teams and different development languages, as depicted in figure 1.4. The key phase in processor design is the architecture specification since it serves as the basis for all remaining design phases. Although Hardware Description Languages (HDLs) are designed for architecture implementation, in a traditional design flow, these languages are also often used for the initial specification of the processor. In this design phase tasks, such as hardware/software partitioning, instruction-set and micro-architecture definition is performed [17][18]. Based on the architecture specification, both the hardware implementation and the development of software tools is triggered. Both tasks are basically independent and therefore performed by different experts and design methodologies. Hardware designers use HDLs such as VHDL or Verilog, while software designers mostly utilize the C/C++ programming language. In addition, the target processor needs to be integrated into the SoC and the application software needs to be implemented. Communication between the design teams is obviously difficult because of the heterogenous methodologies and languages. Nevertheless, as soon as the above-mentioned phases of processor design are completed, the verification of the hardware, application software and software development tools can be triggered.

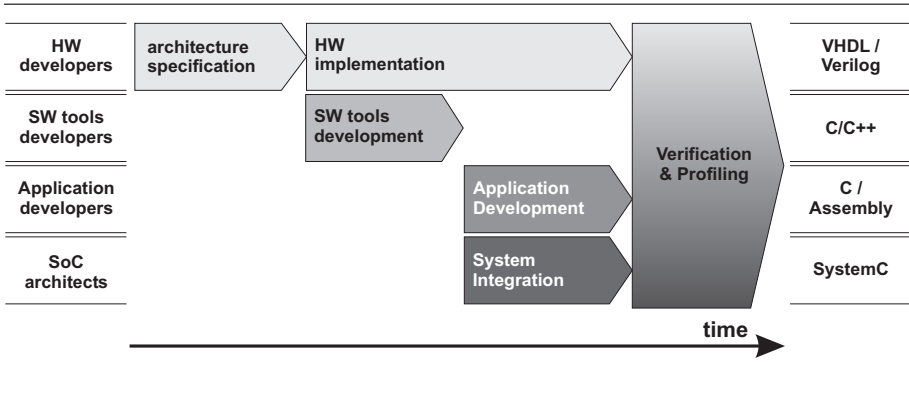


Figure 1.4. ASIP design phases

Considering the traditional processor design flow, the strong dependencies between design phases imply a unidirectional design flow and prevent even minor optimizations. For example, an optimization of the instruction-set due to the application software development requires changes to the software tools and the hardware implementation as well. Due to the different development languages, changes to the architecture are difficult to communicate and inconsistencies are very likely to appear.

The complexity of processor design even increases in ASIP design, since optimizations targeted to a particular application are mandatory. Mapping an architecture to a given application means moving through a design space spanned by axes such as flexibility, power consumption, clock speed, area and more. Every design decision in one dimension constrains other decisions, for example

architectural features	vs. design time,
design time	vs. physical characteristics,
physical characteristics	vs. flexibility,
flexibility	vs. verification effort,
verification effort	vs. architectural features.

It is obviously not possible to move through this design space by applying the traditional processor design methodology. A unified design methodology is required, which provides a common basis for all design phases. It must be suitable for all design engineers involved and naturally consider the dependencies of the design phases.

Today's dilemma in SoC design stems from the requirement of flexible high performance computational building blocks under the constraint of a low design complexity. Most approaches today utilize heterogeneous solutions, which combine already known architecture types and well established design methodologies. Thus, the requirements can only partially be addressed. Contrary to these approaches, this book proposes a new ASIP implementation methodology, which is based on Architecture Description Languages (ADLs) in general and the Language for Instruction-Set Architectures (LISA) in particular. Along with the work performed in the fields of design space exploration, software tools development and system integration based on LISA (cf. chapter 3), this leads to a novel ASIP design methodology. This book addresses the automatic ASIP implementation starting from a given LISA description, which surpasses the traditional processor design flow regarding design efficiency and architectural efficiency. Therefore, a paradigm shift in ASIP implementation is advocated.

1.4 Organization of This Book

This book is organized as follows. In chapter 2 the related work in the field of ASIP design methodologies, RTL hardware implementation and heterogeneous architectures is discussed. In chapter 3, the LISA design environment is described focusing on design space exploration capabilities, software development tools generation and system integration. In chapter 4 impetus is provided regarding the new entry point for ASIP design and implementation. After this, the architecture description language LISA is discussed with regard to automatic hardware implementation (chapter 5). In chapter 6 the intermediate representation used for automatic ASIP implementation is presented. The intermediate representation forms the basis for high level optimizations, described in chapter 7, and for the automatic integration of processor features, described in chapter 8. Three case studies are presented in detail in chapter 9 and 10 to demonstrate the feasibility of the proposed synthesis methodology. The first case study deals with an ASIP for turbo decoding. The second and third case studies discuss architectures derived from existing processors while maintaining application software compatibility. The ISS-68HC11 was devised to achieve a higher instruction throughput compared to the original Motorola architecture. Infineon's ASMD was devised to support new instructions while offering backwards compatibility to existing applications. Finally, the proposed automatic ASIP implementation is summarized in chapter 11.

Chapter 2

ASIP DESIGN METHODOLOGIES

The challenges in ASIP design, as described in the previous chapter, are addressed differently by today's design methodologies. Several approaches are based on Architecture Description Languages (ADLs), which can be grouped according to the following three categories:

Instruction-set centric languages focus on the *instruction-set* and thus represent the programmer's view of the architecture. They are mainly used to describe the instruction encoding, assembly syntax and behavior. The languages nML [19] and ISDL [20] are examples for this category.

Architecture centric languages focus on the *structural aspects* of the architecture. The architecture's behavior is described by functional building blocks and their interconnections, often in an hierarchical description style. Therefore, these languages represent the hardware designer's view of the architecture. An example for this type of ADLs is the MIMOLA [21] language.

Mixed instruction-set and architecture oriented languages combine the description of the instruction-set and the structural aspects in one ADL. Therefore, ASIP design approaches based on mixed-level languages mostly are able to target all domains of ASIP design: architecture exploration, architecture implementation, software development tools generation and system integration. Representatives of this category are e.g. EXPRESSION [22] and LISA [23].

Furthermore, ADLs can be distinguished whether or not a library of predefined building blocks is used (also referred to as *module library approach* [24]). In general, such a set of predefined building blocks enables a high quality of the generated software tools as well as an efficient implementation of the final architecture. However, these advantages are accompanied by a moderate flexibility and therefore possibly suboptimal solutions. This drawback can be eliminated by ADLs which are not limited to a set of predefined components, but allow the semantic and/or behavioral description of architectural building blocks. Due to the enormous flexibility provided by the ADLs, software development tools generation, automatic hardware implementation and system integration become challenging tasks.

In section 2.1 several ADL based approaches, which address all domains of ASIP design, are discussed. Approaches which only target a subset of the complete ASIP design tasks are omitted.

Beside ADL based ASIP design, various approaches are based on (re)configurable architectures. These are presented in section 2.2.

The currently accepted entry point for hardware implementation is an architecture description in an HDL on RTL. The synthesis down to gate-level and logic optimizations are performed by already established tools. Therefore, the RTL hardware model is the interface between the proposed automatic ASIP implementation and the existing implementation flow. HDLs and logic representations are briefly covered in section 2.3.

2.1 ADL based ASIP Design

In contrast to traditional processor design and ASIP design based on (re)configurable architectures, the ADL based design addresses a huge architectural design space while providing a high design efficiency. The different approaches are presented in the following.

EXPRESSION. The EXPRESSION ADL [22][25], developed at the U.C. Irvine, captures the structure and the instruction-set of a processor as well as their mapping. The structural description comprises components and communication between them. Four different types of components exist: units (e.g. ALUs), storage elements (e.g. register files), ports and connections (e.g. buses). Each component is configurable by a list of attributes. The instruction-set is specified by operations that consist of opcode, operands, behavior and instruction format definition. Mapping functions are used to map the structure defined by the components to the operation's behavior specification.

This defines, for each component, the component's set of supported operations (and vice versa).

EXPRESSION originally targeted the software tools generation, such as C-compiler, assembler, linker and simulator. The generation of an RTL hardware model came into focus only in recent years.

The language EXPRESSION provides a set of predefined (sub-)functions to describe the behavior and the instruction-set of the processor. Base on a, so called, *functional abstraction* [26], the functions represent coarse-grained operations (e.g. a program memory access) and sub-functions fine-grained operations (e.g. an addition).

Multiple functions can be nested to describe a behavior which is not represented by the set of functions initially. Considering the hardware generation from an EXPRESSION model, the utilization of predefined (sub-)functions enables a hardware generation based on predefined but configurable RTL building blocks. The major drawback here is given by the reduced flexibility with regard to the behavioral description.

EXPRESSION maps the given architecture description directly to a hardware description [26][27], without applying optimizations. Even if optimized RTL building blocks are instantiated, the hardware generation results in an suboptimal architectural efficiency as a cross template optimization is missing. Also, this framework does currently not support commonly required processor features, such as a debug mechanism or power save modes. Therefore, the generated RTL hardware model only addresses an estimation of the physical characteristics of the architecture during design space exploration.

FlexWare. FlexWare [28] from SGS Thomson/Bell Northern Research consists of the code generator CodeSyn, an assembler generator, a linker generator and the Insulin Simulator. The Simulator is based on a parameterizable VHDL model. Additionally, FlexWare supports the generation of an RTL hardware model, which is not possible in the FlexWare2 environment developed by STMicroelectronics. The FlexWare2 environment [29][30] is capable of generating assembler, linker, simulator (FlexSim) and debugger (FlexGdb) from the, so called, Instruction Description Language (IDL).

ISDL. The Massachusetts Institute of Technology (MIT) developed the language ISDL [20] to target VLIW architecture modeling, simulator generation (GENSIM) and hardware model generation (HGEN).

The published hardware generation approach [31][32] generates a Verilog description from a given ISDL model. The developed hardware

generation only targets the design phase of architecture exploration rather than a final architecture implementation, which therefore must still be manually implemented.

Although resource sharing optimizations are identified as major challenge in automatic hardware implementation from ADLs, their implementation in HGEN only addresses a minimization of computational resources by using a maximum clique search. This abstract view to the resource sharing problem neglects an optimization of the interconnect as well as the timing effects of resource sharing optimizations. Overall, the presented optimizations neither achieve a sufficient architectural efficiency nor trade off different physical characteristics.

ISDL contains only very little structural information about the architecture. The architecture is mainly described from the programmer's point of view, and therefore important information required for an implementation of the architecture is missing or only implicitly available. For example, the information about timing and schedule of instructions is specified in ISDL by annotating the latency. Information about the pipeline structure is not specified but must be derived from the information about the instructions' latencies. Considering the automatic hardware implementation, incomplete or implicit information can only be compensated by assumptions. These certainly lead to a suboptimal architecture implementation if optimizations are not applied adequately.

LISA. The LISA language and the corresponding processor design environment has been developed at the Institute for Integrated Signal Processing Systems (ISS) at the RWTH Aachen University [33] and is commercially available from CoWare Inc. [34]. The LISA design environment is described in chapter 3 regarding architecture exploration, software tools generation and system integration. The information about the underlying hardware which is inherent in the LISA model is described in chapter 5. The optimizations performed during hardware synthesis and the integration of processor features are described in chapter 7 and chapter 8.

MetaCore. The MetaCore system from the Korea Advanced Institute of Science and Technology (KAIST) [35][36] targets the ASIP development for DSP applications. MetaCore divides ASIP design in two phases, first a design exploration and second the design generation. On the basis of a set of benchmark programs and a formal specification of the processor, an estimation of the hardware costs and performance estimation is provided. The formal specification is provided by the so

called MetaCore Structural Language (MSL) and the MetaCore Behavioral Language (MBL). Once the architecture specification is fixed the processor is automatically implemented in an HDL and software tools comprising C-compiler, assembler and instruction-set simulator are generated. The MetaCore system selects and customizes macro-blocks from a library to generate the target architecture. Therefore, the architectural design space is limited to the available macro-blocks.

MIMOLA. The language MIMOLA [21], developed at the Universität Dortmund, describes processors by a hierarchical net-list close to the RTL. Software tools such as C-compiler (MSSQ compiler [37] and RECORD compiler [38]) and instruction-set simulator can be generated. Moreover, MIMOLA supports code-generation, test-generation and the automatic hardware implementation. However, the low abstraction level goes along with numerous architectural details which hinder an efficient design space exploration. The quality of the generated software tools is therefore not sufficient for application software development.

nML. The language nML [19][39] originates from the Technische Universität Berlin and targets the instruction-set description. Several projects utilize nML for software tools generation as well as for hardware generation. Research activities at the TU Berlin led to the generation of the instruction-set simulator SIGH/SIM and the code generator CBC.

Based on nML, a retargetable C-compiler (Chess) and instruction-set simulator generator (Checkers) were developed by IMEC [40] in Leuven, Belgium, and commercialized by Target Compiler Technologies [41]. In addition the commercial software development tool suite comprises a retargetable assembler and disassembler (Darts), linker (Bridge) and a retargetable test-program generator (Risk). The generation of an RTL hardware model is supported by a tool called Go. According to [42], the hardware model is only partially generated and the designer has to plug in the implementation of functional units and memories himself. Due to this semi-automatic approach inconsistencies between the nML model and the RTL hardware model are very likely. In addition, it is hardly possible to evaluate continuously the effects of high level architectural decisions on the physical characteristics of the architecture.

The nML language was also adopted by Cadence in cooperation with the Indian Institute of Technology. The project, called sim-nML [43], targets the software tools generation, while SIM-HS [44] targets the generation of behavioral Verilog models and the generation of structural synthesisable Verilog processor models.

Sim-nML descriptions provide temporal information by the specification of the latency instead of the temporal execution order. Therefore, the synthesis process has to perform scheduling and resource allocation commonly known from behavioral synthesis. The complexity of these techniques is reduced by adopting templates. For example, a four stage pipelined architecture is always assumed. All transformations, which can be performed during synthesis, are either operating on a very coarse-grained level (e.g. pipeline stages) or on a very fine-grained (e.g. single arithmetic operations) level of abstraction. Overall, this approach is lacking flexibility in ASIP implementation by utilizing fixed structural and temporal templates.

PEAS. The PEAS (Practical Environment for ASIP Development) project started in 1989 targeting the configuration of an parameterizable core regarding the instruction-set and architectural parameters [45], such as bit widths or register file size. In the second version of the PEAS system the support for VLIW architectures was added. In the third version of the PEAS environment [24][46], called PEAS-III, also simple pipelined architectures are addressed. The development of the PEAS environment was the basis for ASIP Meister [47], available as beta version first in 2002. The PEAS environment is based on a graphical user interface (GUI), capturing the architecture's specification. According to the architecture specification a hardware model for simulation and synthesis is generated. Even if the gate-level synthesis results are comparable to handwritten implementations, the PEAS approach covers only a small area of the available ASIP design space.

Tipi (Mescal Architecture Development System). The Tipi ASIP design framework has been developed within the scope of the Mescal project at the U.C. Berkeley [48][49][50]. The Tipi framework focuses on the specification of the data-path, which is described in a structural manner close to RTL. The control-path is considered to be overhead in the ASIP design process and thus strongly abstracted or omitted. An assembler, a cycle and bit true simulator and a synthesizable RTL Verilog model are generated from the architecture description. For hardware implementation a one-to-one mapping from the Tipi framework to the hardware model is proposed. High level optimizations and the generation of commonly required processor features are not included in the framework. Details about hardware implementation and their gate-level synthesis results, especially about the control-path, are not available.

The current trends in ASIP design clearly indicate that the design entry point is raised above the RTL. In contrast to this, the Tipi design framework concentrates on an abstraction level close to RTL, which is not suited to perform an efficient design space exploration. Moreover, possible optimizations to the implementation of the complete architecture are highly limited by focusing only on the data-path specification.

2.2 (Re)Configurable Architectures

Although the proposed LISA processor design methodology is completely different from utilizing (re)configurable designer extensible architectures both approaches strongly compete. Compared to ADL based ASIP design, (re)configurable architectures cover a limited architectural design space only but enable a high design efficiency. Configurable architectures can be customized before fabrication, while reconfigurable architecture provide the same architectural flexibility even after fabrication. In the following, three examples are discussed.

ARC. ARC International [51] is an independent company since 1998 and provides configurable microprocessor cores. Also, peripherals, software (such as Real Time Operating Systems and Codecs) and development tools are included in the company's portfolio.

ARC offers several architectures, currently the ARCTangent-A4, ARCTangent-A5, ARC 600 and ARC 700. The RISC cores support the ARCCompact instruction-set, which provides 16 bit instructions for the most frequently used operations and covers all other operations by 32 bit instructions. These two types of instructions can be arbitrarily used within one application. The architecture may be configured with regard to DSP extensions, cache and memory structure, bus interfaces or peripherals. A designer can select from a range of processor configuration options or add custom Intellectual Property (IP) to alter the architecture. For this purpose, ARC provides a Java-based configuration tool, namely ARChitect 2. It enables designers to vary configuration options and to incorporate new instructions. The design environment generates the source code (Verilog or VHDL), synthesis scripts, test files, HDL-simulator support files and HTML-formatted documentation for the design.

Tensilica. Tensilica Inc. [52] provides configurable microprocessor cores, in 2005 the Xtensa LX and Xtensa V. The latter one, for example, is a 32 bit RISC microprocessor architecture made up of five pipeline stages and supports a 16 bit or 24 bit instruction-set. The

instruction-set is not switchable during runtime. The microprocessor is made up of fixed, optional, configurable and designer defined building blocks. The base architecture comprises processor control unit, instruction fetch unit, decode unit and data load/store unit. The memory management unit, all memory blocks and i/o blocks are optional and/or configurable. Other processor features, such as Trace Port, JTAG Tap Control, on-chip debug mechanisms, multiply-accumulate units, multiply units, floating point unit and also a DSP engine are optional. Registers and, so called, hardware execution units can be defined by the designer.

Designer defined building blocks are specified in the Tensilica Instruction Extension (TIE) language [53]. This Verilog-like specification language enables the designer to cover the hardware aspect, such as computational units and additional registers, and the software aspect, for example instruction mnemonics and operand utilization. The TIE description and the selection/configuration of building blocks are the input to the, so called, processor generator. This tool generates on the one hand the synthesisable RTL VHDL or Verilog description and on the other hand the software tools for application development. Further information can be found in [9].

This approach enables a high performance hardware implementation as well as high quality application software development tools. Tensilica's latest development is the Xtensa Processor Extension Synthesis (XPRES) Compiler. This tool identifies hot spots in a given C-application and customizes the Xtensa architecture to accelerate the application execution.

The limited architectural flexibility is the major drawback of this approach. The fixed structure of the on-chip communication as well as the given microprocessor core cause an upper bound to the performance increase.

Stretch. Stretch Inc. [54] offers the reconfigurable processor core family S5000. This architecture family consists of a programmable RISC processor core with a tightly coupled reconfigurable device, so called Instruction-Set Extension Fabric (ISEF). The ISEF is used to implement custom instructions, which extend the original instruction-set of the architecture. Up to three 128 bit operand registers and two 128 bit result registers can be utilized to establish the communication between the fixed RISC architecture and the reconfigurable portion of the architecture. The RISC core of the S5000 family is based on the Tensilica Xtensa RISC processor core, described above.

Available tools are C-compiler, assembler, linker, debugger and profiler, which are combined in an integrated design environment. Moreover, the C-source code of the application can be profiled automatically and hot spots are identified with regard to the application execution time. These portions of the application are mapped to the ISEF. With this approach a simulated speed up of a factor of 191 is achieved for the EEMBC Telemark Score [55], by comparing the S5610 RISC core with and without customized instructions.

The S5000 architecture family provides post-fabrication flexibility with regard to the control by software and with regard to the datapath implementation by an reconfigurable building block. However, application-specific optimizations are limited to this reconfigurable building block. Thus, numerous optimizations in ASIP design are omitted. For example, the pipeline organization cannot be changed in accordance to the targeted application. The memory as well as register organization is fixed and the interface between processor core and the reconfigurable building block is predefined.

2.3 Hardware Description Languages and Logic Representation

A hardware description on RTL is the commonly accepted entry point for architecture implementation. Within this book a new synthesis approach from an ADL to a hardware description language is proposed. Currently, VHDL, Verilog and SystemC are supported. The synthesis framework can easily be enhanced to support future languages, for example SystemVerilog.

2.3.1 VHDL, Verilog, SystemVerilog and SystemC

In the scope of this book the languages VHDL, Verilog, SystemVerilog and SystemC are considered as hardware description languages on RTL, although these languages can be used to describe the target architecture on multiple abstraction levels. Common to all languages is the ability to describe hardware on RTL by providing elements such as concurrency, clocks and sensitivity.

The development of **VHDL** has been started in 1980 by IBM, Texas Instruments, and Intermetrics and became the IEEE 1076 standard in 1987 [56][57]. The initial version (VHDL '87) was continuously discussed and adopted to new requirements (VHDL '93 and VHDL '00). VHDL is mostly used in Europe.

In 1985 **Verilog** originated from Automated Integrated Design Systems, in the meanwhile acquired by Cadence Design Systems [58]. This HDL is mostly used in the USA and Asia. It became IEEE Standard 1364 [56][59] in 1995. The latest revision is Verilog 2001.

SystemVerilog is the latest approach, initiated by Accelera, to establish a language combining aspects of a hardware description and hardware verification [60]. Currently, this language is only supported by few existing tools, thus real-world experience about the efficiency is missing.

SystemC has been invented to close the gap between algorithm specification, often described using the C-programming language, and system simulation [61]. Its current development is driven by the Open SystemC Initiative (OSCI) which is steered by major EDA companies and IP-providers. However, while the SystemC language became famous in the area of system level design, the support for gate-level synthesis in Synopsys' Design Compiler was discontinued in 2003.

2.3.2 Logic Representation and Binary Decision Diagrams (BDDs)

In order to perform gate-level synthesis, the RTL hardware description needs to be transformed into a low level representation made up of building blocks of a technology library. Therefore, gate-level synthesis consists of several steps: logic synthesis, logic optimizations and technology mapping. These steps are based on boolean function representations, such as truth tables or Karnaugh Maps. However, these data structures are insufficient to handle the computational complexity of real world problems. Two-level boolean representation or Multi-Level boolean networks became favored to apply advanced optimization techniques.

An alternative representation are Binary Decision Diagrams (BDDs), which have been discussed since 1959 [62][63]. However, BDDs gained importance in 1986 when Bryant [64] proposed restrictions on BDDs enabling new algorithms.

In order to reduce the computational complexity, the synthesis framework and the optimizations proposed in this book operate on an abstraction level above RTL. Therefore, the concepts commonly known from logic optimizations are not reused.

2.4 Motivation of This Work

As described above, various approaches cover hardware generation from ADLs. However, most approaches map the architectural information of the ADL model directly to a hardware description on RTL. The achieved physical characteristics, in terms of area, timing and power

consumption, are not sufficient for a final architecture implementation. Optimizations are basically not considered for the ADL to HDL synthesis process. Moreover, essential processor features, such as debug mechanism or power save modes, are not covered by these automatic approaches. Therefore, in these approaches hardware generation is only suited for a rough estimation of the physical characteristics during architecture exploration.

Some approaches base on coarse granular building blocks provided in a library or by predefined architecture templates. In these cases, an efficient implementation can be derived from the architecture model. However, the architecture model cannot be optimally tailored to the application, as only a limited architectural flexibility is provided by these approaches.

The contribution of this work is an automatic ASIP implementation from LISA, which fulfills the demands of a final hardware implementation while preserving the desired architectural flexibility. Based on an intermediate representation between the level of ADLs and HDLs and explicit architectural information given in the LISA model, optimizations are performed and processor features are automatically integrated. The achieved architecture implementations are comparable to implementations handwritten by experienced designers. Therefore, the entry point for ASIP implementation is shifted from the currently accepted RTL to a LISA based implementation.

Chapter 3

ASIP DESIGN BASED ON LISA

The LISA Processor Design Platform (LPDP) has been developed at the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University [33]. The LISA design methodology [23][65] has become one of the most comprehensive and powerful ADL based design environments and is internationally recognized by academia and industry. The design environment based on LISA, including the automatic ASIP implementation proposed in this book, is commercially available from CoWare Inc. [34].

In order to tailor the architecture to the special requirements of a given application, ASIP design requires an efficient design space exploration phase. The instruction-set, micro-architecture, register configuration, memory configuration, etc. are subject to optimizations. The LISA based design space exploration, software tools generation and system integration are described in the following sections.

3.1 Design Space Exploration

In ASIP design the key is an efficient exploration of the architectural design space. The LISA language allows to implement changes to the architecture model quickly, as the level of abstraction is higher than RTL. As shown in figure 3.1, the LISA model of the target architecture is used to automatically generate software tools such as C-compiler, assembler, linker, simulator and profiler. These software tools are used to identify

hot spots and to jointly profile the architecture and the application. Both are optimized according to the profiling results, e.g. throughput, clock cycles or execution count of instructions. This exploration loop can be repeated until the design goals are met. With LISA a highly efficient design space exploration is ensured, as the ASIP model can be modified easily and the software tools are re-generated within a negligible amount of time. Moreover, the LPDP provides additional tools that accelerate the design space exploration, such as, for example, an instruction encoding generator [66].

The knowledge about the physical characteristics of the architecture is important at an early design stage already. For example, the information about clock speed substantially influences the number of pipeline stages or even the pipeline organization in general. Ignoring physical parameters in the design space exploration phase leads to suboptimal solutions or long redesign cycles. The automatic ASIP implementation from LISA, also shown in figure 3.1, provides important feedback about the physical characteristics of the architecture. The hardware generation from LISA is discussed in the following chapters.

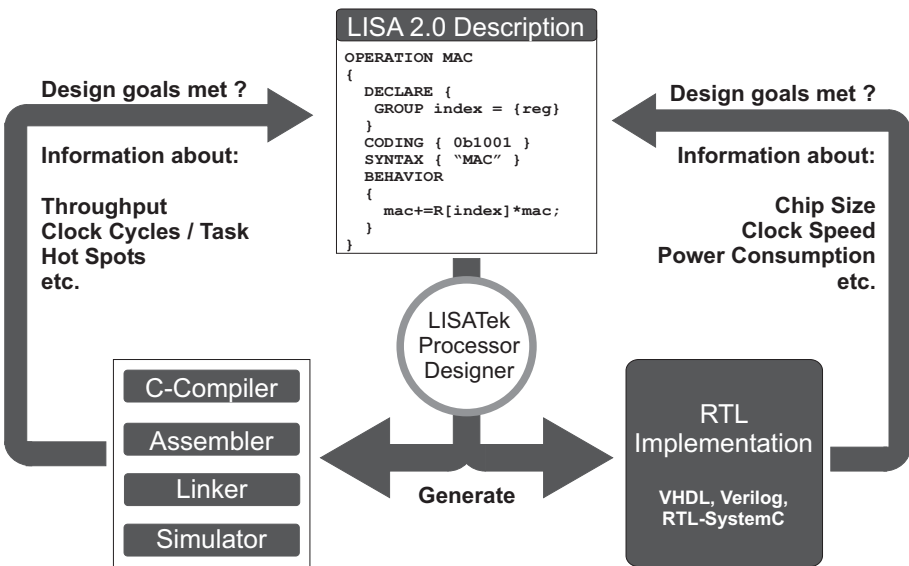


Figure 3.1. Exploration and implementation based on LISA

3.2 Software Tools Generation

The software tools generated from the LISA description are able to cope with the requirements of complex application development. C-compiler, assembler, linker, simulator and profiler are generated from a LISA model.

3.2.1 C-Compiler Generation

The shift from assembly code to the C-programming language is ongoing for software application development for embedded systems. This is basically motivated by the increasing complexity of the embedded software. Considering the number of different versions of an ASIP during design space exploration and the complexity of C-compiler design Complexity, C-Compiler design, the automatic generation of a C-compiler is highly desired [67]. For this reason, the automatic generation of a C-compiler in ASIP design strongly came into focus recently [68].

To retarget a C-compiler, in particular the architecture-specific backend of a C-compiler must be adjusted or rewritten, while the architecture independent frontend and most of the optimizations are kept unchanged. The C-compiler platform CoSy from ACE [69] is used to retarget the C-Compiler based on the information extracted from the LISA model. While some information is explicitly given in the LISA model (e.g. via resource declarations), other information (e.g. resource allocation) is only implicitly given and needs to be extracted by special algorithms. Some further, highly compiler-specific information is not present in the LISA model at all, e.g. data-type bit widths. Thus, compiler information is automatically extracted from LISA whenever possible, while GUI-based user interaction is required for other compiler components. For details about C-compiler generation from LISA the reader is referred to [70][71] [72].

3.2.2 Assembler and Linker Generation

The generated assembler [73] processes the assembly application and produces object code (LISA Object File, LOF) for the target architecture. An automatically generated assembler is required, as the modelled architecture has a specialized instruction-set. A comfortable *macro assembler* exists to provide more flexibility to the designer.

The automatically generated linker combines several pieces of object code to a single executable in ELF format [74]. The linker is architecture-specific since the modelled memory organization needs to be taken into account. Various configuration possibilities are provided to steer the

linking process. Further information about assembler, macro-assembler and linker can be found in [75].

3.2.3 Simulator Generation

Generation, SimulatorThe generated simulator is separated into backend and frontend. The frontend is shown in figure 3.2. It supports application debugging, architecture profiling and application profiling capabilities. The screenshot shows some features such as disassembly view (1) including loop and execution profiling (2), LISA operation execution profiling (3), resource profiling (4) and pipeline utilization profiling (5). The content of memories (6), resources in general (7) and registers in particular (8) can be viewed and modified. Thus, the designer is able to debug both ASIP and application easily. Additionally, the necessary profiling information for design space exploration is provided, such as pipeline and instruction utilization.

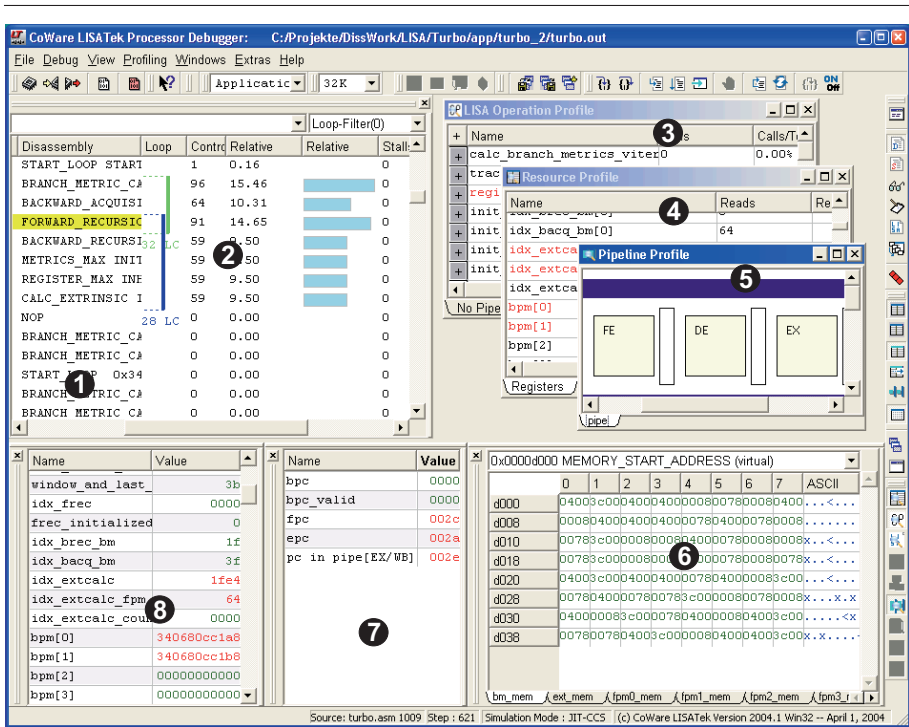


Figure 3.2. The LISA simulator and debugger frontend

The performance of the simulator strongly depends on both the abstraction level of the underlying LISA model and the accuracy of the memory model [76]. Furthermore, various simulation techniques, e.g. compiled simulation [77], interpretive simulation and Just-In-Time Cache Compiled Simulation (JIT-CCS) [78][79] are supported. These techniques are briefly described below.

Interpretive simulation

The interpretive simulation technique, interpretive is based on a software implementation of the underlying decoder of the architecture. For this reason, the interpretive simulation is considered to be a virtual machine performing the same operations as the hardware does, namely fetching, decoding and executing instructions. All simulation steps are performed at simulation runtime. In contrast to a hardware decoder, the control portions of an instruction execution requires a significant amount of time in software.

Compiled simulation

Compiled simulation [80][81] uses the locality of code in order to speed up the execution time of the simulation compared to the interpretive simulation technique. The task of fetching and decoding is performed once for an instruction before simulation run. Thus, the simulation runtime is reduced when instructions are executed multiple times. However, compiled simulation requires the program memory content to be known before simulation runtime and to be static during simulation. Various scenarios are not supported by the compiled simulation technique, such as system simulations with external and thus unknown memory content or operating systems with changing program memory content. Additionally, large applications require a large amount of memory on the target host.

Just-in-time cache compiled simulation (JIT-CCS)

The objective of the JIT-CCS is to provide both a general applicability and a high simulation speed. Techniques from interpretive simulation and compiled simulation are combined in the JIT-CCS [79]. The basic technique is to perform the decoding process *just-in-time* at simulation runtime and to store the results in a cache for later use. In every subsequent simulation step the cache is searched for already existing and valid decoding results. Because of that, the full flexibility of interpretive simulation is provided. Due to the locality of code in typical applications the simulation speed can be significantly improved using JIT-CCS.

Numerous case studies show that with an increasing cache size JIT-CCS performance converges to the performance of compiled simulation. According to the case studies a reasonable maximum of cache lines is 32768, where a line corresponds to one decoded instruction. The maximum amount of cache lines corresponds to a memory consumption of less than 16 MB on the simulator host. Compared to the traditional compiled simulation technique, where the complete application is translated before simulation, this memory consumption is negligible.

3.3 System Simulation and Integration

Today, typical SoCs combine a mixture of several computational elements, such as DSPs, μ Cs, ASIPs, ASICs, etc. Certainly, system level simulation is necessary for both performance evaluation and verification in the system context. The automatically generated LISA processor simulators can easily be integrated into various system simulation environments, such as CoWare ConvergenSC [34] or Synopsys' System Studio [82].

The communication between an ASIP and its system environment can be modelled on different levels of abstraction. First, LISA pin resources can be connected directly to the SoC environment for pin accurate co-simulation. Second, the LISA bus interface allows to use commonly accepted SoC communication primitives and to model the communication on a higher abstraction level, for example the level of TLM [83]. Third, a generic C-programming language interface provides adaptability to arbitrary interfaces.

The system simulation debugger offers all observability and controllability features for multiprocessor simulation as known from standalone processor simulation. Thus, a software centric view to an arbitrary number of ASIPs as well as the system context is provided to the designer [84].

Chapter 4

A NEW ENTRY POINT FOR ASIP IMPLEMENTATION

Due to the high efficiency of design space exploration, the automatic generation of software development tools and the possibility of an early system integration, ASIP *design* nowadays starts with an architecture model in an ADL. The ASIP design phases and their corresponding software tools are presented in the previous chapter. ADLs do not provide a link to the ASIP *implementation* naturally, and thus essential information about the physical characteristics of the architecture is unknown.

Currently, the design phase of hardware implementation is entered on the RTL. Gate-level synthesis tools, such as Synopsys' Design Compiler [85] or Cadence's First Encounter [86], are used to proceed from RTL to gate-level regarding a particular technology library. On this abstraction level, accurate estimates about the physical characteristics, including clock speed, area or power consumption, can be obtained for the first time¹.

A manual implementation of the ASIP on RTL certainly provides information about physical characteristics. However, modifications to the ADL model of the ASIP are often necessary because of either violated physical constraints or possible optimizations. Obviously, in these cases the design efficiency becomes unacceptably low, as two models for software tools generation and implementation have to be written and maintained throughout the entire design process. In particular, adapting an RTL hardware model is a tedious and error-prone task, which

¹Research and development in the field of already established design methodologies below RTL is still ongoing. In particular an accurate estimation of physical characteristics for shrinking technology sizes is targeted.

typically leads to significant inconsistencies between the two models of the same ASIP.

Various approaches target the generation of an RTL hardware model from an ADL model, as described in chapter 2. The generated RTL hardware models are inferior to models that are handwritten by experienced designers [26][87][88][89]. They neither achieve equal physical characteristics nor support well-known processor features, such as built-in debug mechanisms or automatic power save control. Thus, these RTL hardware models are only used during architecture exploration for a rough estimation of the physical characteristics. Manual optimizations are required if these models are used for the final architecture implementation. However, this potentially causes inconsistencies between the final implementation and the automatically generated software development tools.

Obviously, there is a pressing need for an advanced automatic ASIP implementation which closes the currently existing gap between ADL models and RTL hardware models. To gain the acceptance of ASIP designers an automatic approach must fulfill particular criteria, which are discussed in the following.

4.1 Acceptance Criteria for an Automatic ASIP Implementation

Considering several academic and industrial case studies [26][87][88], the following criteria must be fulfilled in order to establish an automatic ADL based ASIP implementation. First of all, the *architectural efficiency* and *implementation flexibility* must be close to the results achieved by manual implementations on RTL. Second, the *processor features*, which are supported by an automatic implementation, must be comparable to the those of handwritten implementations. Third, the goals must be met without compromising architectural flexibility and design efficiency. These three criteria are elaborated in the following sections.

4.1.1 Architectural Efficiency and Implementation Flexibility

In VLSI design, *architectural efficiency* is used to quantitatively compare different architectural alternatives. The physical characteristics, such as timing, area and energy, are used to evaluate an architecture. The valid ranges of the physical characteristics are defined by *boundary constraints*, whereas *precise constraints* must be accurately fulfilled to avoid device failure.

Derived from [90] and [91], physical characteristics² P_i and corresponding weights w_i define the architectural efficiency η_{arch} by

$$\eta_{arch} = \prod_{i=1}^n \frac{1}{P_i^{w_i}}. \quad (4.1)$$

In VLSI design the architectural efficiency is commonly defined by $\eta_{arch} = \frac{1}{AT}$ (or $\eta_{arch} = \frac{1}{ATE}$), which uses equally weighted Area (A), Timing (T) (and Energy (E)).

According to equation 4.1, the same architectural efficiency can be achieved by various implementations each balancing the physical characteristics differently. The ability to trade off the physical characteristics during architecture implementation is called **implementation flexibility** in the following.

In order to optimally tailor the architecture to the application, the architectural efficiency and the implementation flexibility are equally important in ASIP design. In both fields, an automatic approach must achieve results which are close to handwritten implementations by experienced designers.

4.1.2 Processor Features

The recent developments of GPPs, μ C and DSPs have produced sophisticated auxiliary **processor features**, such as built-in debug mechanisms or automatic power save control. Although these features do not influence the core functionality of the architecture, they are nevertheless important for the usability of the ASIP and can considerably influence the implementation. HDLs naturally support their description, while ADLs do not in order to accelerate the design space exploration. Certainly, this negatively affects the acceptance of ADL based ASIP design. Therefore, an automatic ASIP implementation has to cover these processor features in an appropriate way.

4.1.3 Design Efficiency

The **design efficiency** is of major importance in SoC and ASIP design to satisfy today's time-to-market requirements. The architectural efficiency η_{arch} and the overall design effort T_{design} (in man months and weighted by w_{design}) can be used to compare the efficiency of different design methodologies [90][91], as given in equation 4.2.

²In [91] P_i is referred to as architectural parameter.

$$\eta_{design} = \left(\frac{1}{T_{design}} \right)^{w_{design}} \eta_{arch} \quad (4.2)$$

The main goal of an ADL-based ASIP design is a high design efficiency, which has been demonstrated by various case studies, as presented in [23], [87], [92], [93], [94] and [95].

4.2 From Architecture to Hardware Description

ADLs enable a high design efficiency by a *declarative* description style [96] and a significant abstraction or even complete neglect of the interconnect. In LISA, for example, the binary encoding of the instructions is specified explicitly by a directed acyclic graph. The graph structure inherently provides architectural information, such as the information about the exclusive execution of operations. Neither the logic nor the interconnect of the corresponding decoder implementation are specified explicitly in a LISA model, but implicitly predetermined by the LISA simulator behavior.

In contrast to ADLs, HDLs are used to describe logic and interconnect on the RTL. Thus, models in an HDL represent a particular hardware implementation. In general, it is almost impossible to retrieve the semantics and architectural properties from an HDL model. For example, a decoder description in an HDL describes logic and interconnect which sets control signals according to a particular instruction-set. Neither the instruction-set nor information about the exclusive execution of operations can be retrieved from this description with an acceptable computational effort.

According to the character of ADLs and HDLs mentioned above, ASIP models in both language types can be considered to be complementary descriptions. Architectural properties which can be described explicitly by one of the language types can only be described implicitly by the other one.

Targeting an automatic ASIP implementation, an intuitive approach is to map the elements defined in a given ADL model to an equivalent description in an HDL (cf. chapter 2). In this approach, abstract or missing information must be compensated by worst-case assumptions, to achieve a complete and equivalent model on RTL. Naturally, worst-case assumptions lead to redundancy in the implementation and cause insufficient physical characteristics. Therefore optimizations are required to achieve an acceptable architectural efficiency.

For example, elements with a *local scope*, such as signals and ports, are used in RTL hardware descriptions to access storage elements. In contrast to this, in ADL models the storage elements are mostly declared

in a *global scope* and can be instantly accessed from anywhere in the model. In addition, the instructions' behaviors are described almost independently of each other. When mapping them to an RTL description, each access to the storage elements, for example registers, results in an individual connection, as shown in figure 4.1. Here, read accesses are shown, realized on RTL by transmitting the address and retrieving the data. In this case the redundancy is introduced by the instantiation of large multiplexers between registers and logic.

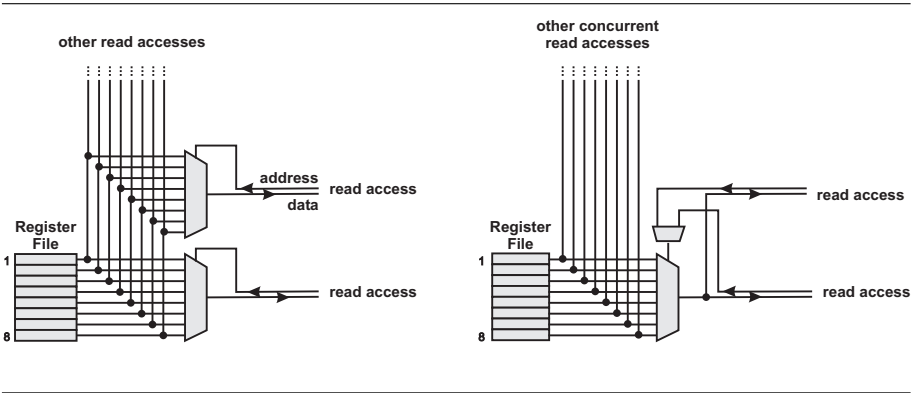


Figure 4.1. Register accesses with and without optimizations

In order to perform optimizations, in particular, information about the exclusive execution of operations is required. In figure 4.1 exclusive read accesses to the registers enable the substitution of one large multiplexer by a smaller one, that only switches the transmitted address.

Several case studies [87][93][89] demonstrate that gate-level synthesis tools are not able to remove the redundancy introduced by the mapping approach mentioned above. Basically four reasons have been identified for this:

1. On RTL the temporal and logical dependencies within the architecture are represented by several independent signals or even bits. Due to this fine-grained representation an analysis of the dependencies becomes unacceptably expensive.
2. On RTL the functionality of the ASIP is described by processes. Several of them commonly represent a cohesive functional block, which is not reflected in the hardware description. Therefore, the relations between processes need to be analyzed before optimizations can be applied.

3. *Semantic information* about logic, storage elements and interconnect does not exist on RTL.
4. In *pipelined architectures* dependencies must be traced back across structural and temporal boundaries, which is not possible regarding the fine-grained description of logic and interconnect.

Since the required optimizations can obviously not be performed during gate-level synthesis, they must be applied during the generation of the RTL hardware model. Requirements to an optimization framework can be derived from the four reasons above, as discussed in the following.

4.2.1 Motivation for a Unified Description Layer

The first and second reason in section 4.2 result from the low abstraction level which is associated with gate-level synthesis tools. Therefore, optimizations during an automatic ASIP implementation require a higher abstraction level. For later use, this requirement is referred to as *raised abstraction level*.

The third reason in section 4.2 results from the ability of HDLs to describe arbitrary logic and to automate the architecture implementation. For architecture implementation, the semantics of the logic and interconnect is not required and thus not contained in an RTL hardware model. In order to perform optimizations, the semantics of the logic and interconnect needs to be analyzed. Utilizing the semantics provided by an ADL model enables optimizations by significantly reducing the analysis effort. This requirement is referred to as *preserved semantics of hardware description*.

The first, second and third reason define requirements which lead to a new (intermediate) abstraction level intended for optimizations (and transformations) of the ASIP, as shown in figure 4.2. This abstraction level represents a hardware model for implementation, which retains additional information derived from the ADL model. Therefore, this intermediate representation is called *Unified Description Layer (UDL)*.

The UDL is independent from existing ADLs and HDLs. According to common software design techniques, the proposed synthesis tool is subdivided into *frontends*, *transformations* and *backends*. The frontends depend on particular ADLs, the backends are specific to the supported HDLs, whereas the transformations are language independent. The UDL is described in detail in chapter 6.

The definition of a suitable abstraction level to perform optimizations is only one of two important aspects when defining a synthesis framework for automatic ASIP implementation. The second aspect is that the analysis of ADL models is of significantly less complexity compared to

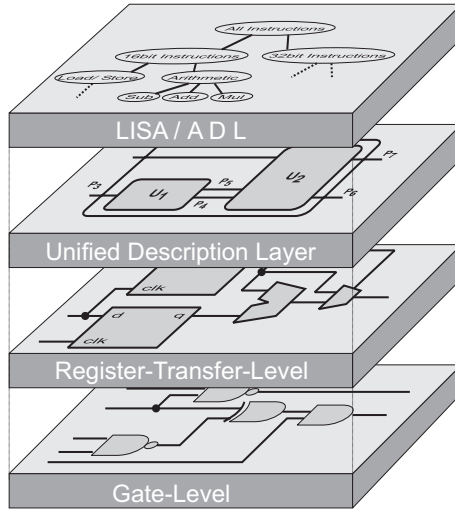


Figure 4.2. ADL to RTL synthesis flow based on an UDL

RTL models. This is of major importance, in particular, when applying optimizations and will be discussed in more detail in the following section.

4.2.2 Analysis of Architectural Information

The fourth reason in section 4.2 results from the high complexity to retrieve and analyze architectural information on the RTL. In pipelined architectures, the instructions' behaviors are distributed over several pipeline stages. This introduces structural and temporal dependencies, which are only *implicitly* modelled in a hardware description on RTL. ADLs separate the structural specification of the pipeline, the temporal relation of operations and the instructions' behavioral descriptions. Therefore the required information to perform optimizations is *explicitly* available in an ADL model. Consequently, its extraction is effortless even across register and module boundaries. The extraction of architectural information from a LISA model is described in detail in chapter 5.

In general, the analysis effort is strongly dependent on whether the required information is explicitly or implicitly available in the architecture model. The different analysis efforts between LISA models (as one representative of ADLs) and RTL hardware models is elaborated by the respective worst case scenarios in the following. Even a simple scenario, which neglects temporal and structural boundaries, shows the significant advantage of the LISA model analysis.

In this scenario n_{op} operations are decoded. Information about the exclusiveness of operations is defined by the binary encoding of the instruction-set. An example of a binary encoding in LISA is given in figure 4.3, including the explicit information about the exclusive execution of operations. On RTL, the information about exclusive execution needs to be determined by computation.

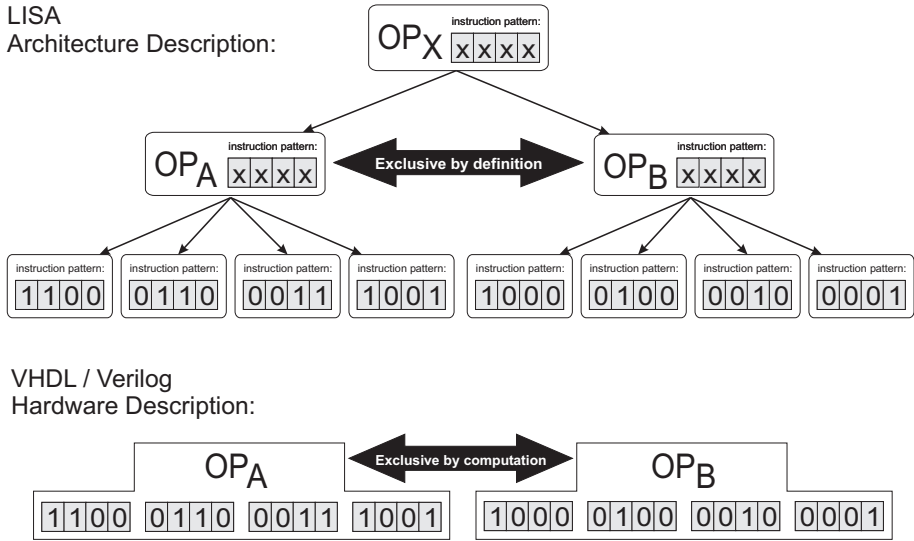


Figure 4.3. Exclusiveness in architecture descriptions and hardware descriptions

LISA model analysis

In LISA, the instruction-set is defined by a directed acyclic graph. Information about the exclusiveness of operations is provided inherently by the graph structure. In figure 4.3, the exclusiveness of the operations OP_A and OP_B is represented in their common ancestor OP_X without any computation. In the worst case scenario, the LISA instruction encoding specification might be degenerated to a binary tree, where all left children are leaf nodes. The depth of the tree is $\lceil n_{op}/2 \rceil$ operations. The exclusiveness detection algorithm has to iterate over the complete tree, giving a linear complexity $\mathcal{O}(n_{op})$ for a single exclusiveness decision between two LISA operations.

The analysis of exclusiveness information for all pairs of LISA operations requires about $n_{op}^2/2$ decisions, which leads to a total complexity

of $\mathcal{O}(n_{\text{op}}^3)$. However, this estimation completely ignores the structural information provided by a directed acyclic graph. In section 5.5 an algorithm, which obtains the exclusiveness information with a lower complexity, is presented. The total **LISA exclusiveness decision complexity** is $\mathcal{O}(n_{\text{op}}^2)$.

RTL hardware model analysis

On RTL, the link given by the common ancestor OP_X is lost. The information whether OP_A and OP_B are exclusively executed can only be retrieved by examining separate control signals, for example A_{decoded} and B_{decoded} . These signals depend on the decode logic and thus represent the result of boolean functions. The operations OP_A and OP_B are exclusively executed if the conjunction of their boolean functions is zero, as given in statement 4.3.

$$OP_A \text{ exclusive to } OP_B \Leftrightarrow A_{\text{decoded}} \wedge B_{\text{decoded}} \equiv 0 \quad (4.3)$$

In general, the control signals A_{decoded} and B_{decoded} might depend on several instruction encodings (cf. figure 4.3). The boolean functions representing the decoder signals depend on $b_A = \lceil \log_2(n_A) \rceil$ and $b_B = \lceil \log_2(n_B) \rceil$ binary variables, where n_A and n_B denote the number of represented instruction encodings. In the worst case, the boolean functions of both signals cannot be minimized. In this case, the boolean functions comprise 2^{b_A} and 2^{b_B} different terms, each representing a particular bit pattern. If both functions are combined in a conjunction for exclusiveness detection (statement 4.3), $2^{b_A} \cdot 2^{b_B}$ bit pattern comparisons are required by the distributive law. A single check for exclusiveness of A_{decoded} and B_{decoded} requires $n_A \cdot n_B$ operations. Therefore, the complexity for a single exclusiveness decision on RTL is $\mathcal{O}(n_{\text{op}}^2)$.

The total exclusiveness detection problem on RTL is most complex for a binary decision tree, degenerated into two subtrees, in combination with non-minimizable boolean functions for all control signals. Since no hierarchical relation between the subtrees exist, the complexity of exclusiveness analysis cannot be reduced. Such a case is given in figure 4.4. In each subtree the disjunctions can be associated with a particular *level*, which can be determined by the number of represented instruction encodings n_A by $L = n_A - 1$. For each disjunction the control signal is a non-minimizable boolean function with $L + 1$ conjunctions. The number of comparisons for all pairs of control signals from different subtrees is given by the iteration over all levels i and j as shown in equation 4.4.

$$\begin{aligned}
\text{boolean term comparisons: } & \sum_{i=1}^L \sum_{j=1}^L (i+1) \cdot (j+1) \\
& = \sum_{i=1}^L (i+1) \cdot \frac{(L+2) \cdot (L+1)}{2} \quad (4.4) \\
& = \frac{(L+2)^2 \cdot (L+1)^2}{4}
\end{aligned}$$

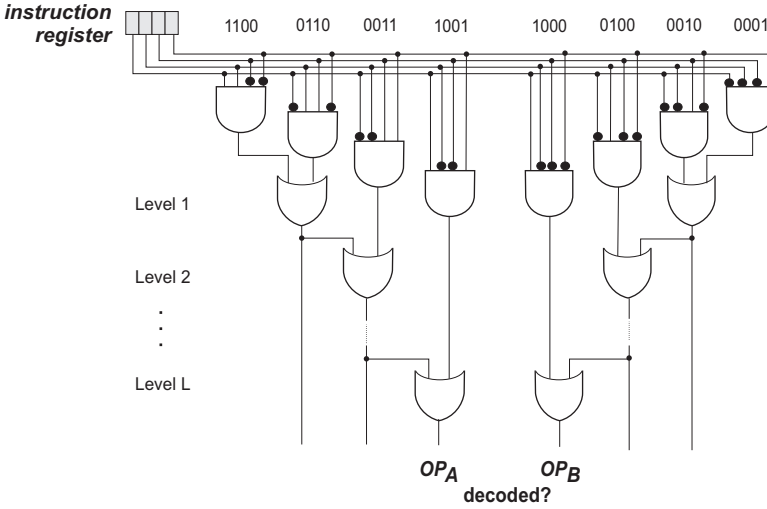


Figure 4.4. Example of decode logic for a degenerated encoding specification

The complexity of this comparison is $\mathcal{O}(L^4)$ and $\mathcal{O}(n_{\text{op}}^4)$ with the relation $L \sim n_{\text{op}}$. To obtain the total complexity also the complexity of comparisons within subtrees must be added. Due to the hierarchical structure of the subtrees, the complexity is less than $\mathcal{O}(n_{\text{op}}^4)$ and thus negligible. The **complexity of the exclusiveness computation on RTL** is $\mathcal{O}(n_{\text{op}}^4)$.

This complexity rule for the analysis on RTL ignores all other signals that are present in the processor model. Synthesis tools cannot distinguish the signals in an architecture naturally, but need to compute the influence of *all* signals to the exclusiveness of operations. Also, the logic, that implements the decoding decisions, is often distributed over several pipeline stages. Thus, the analysis must cross both temporal

and structural boundaries. In real-world architectures this is often not possible with an acceptable computational effort.

Comparison between the complexities of the LISA and RTL model analysis

With regard to the complexities given for the ADLs and RTL model analysis, even this simplified scenario demonstrates the vital performance advantage of a LISA model analysis:

$$\frac{\text{Total RTL Exclusiveness Decision Complexity}}{\text{Total LISA Exclusiveness Decision Complexity}} = \mathcal{O}(n_{\text{op}}^2). \quad (4.5)$$

Because of this analysis, gate-level synthesis cannot be expected to perform the strongly required optimizations.

Both an efficient analysis of the architectural properties and the appropriate abstraction level (and data structure) are required for optimizations. The following chapter presents the extraction of architectural properties from LISA, while chapter 6 introduces the UDL used for optimizations.

Chapter 5

LISA FRONTEND

This chapter discusses the LISA language elements with regard to the generation of an RTL hardware model. The information contained in a LISA model is sufficient to generate a *complete* RTL hardware model as well as to automatically perform optimizations. Thus all requirements discussed in section 4.1 can be satisfied.

With LISA an ASIP can be described on different levels of abstractions. The model abstractions range from *HLL algorithmic kernel models* down to *cycle based models* [23]. A smooth transition from one abstraction level to the next lower level ensures a highly efficient design flow. Regarding the RTL hardware model generation from LISA, the most accurate ASIP specification on the lowest abstraction level enables the most efficient implementation. Therefore, the automatic hardware implementation, proposed in this book, is only possible from cycle accurate LISA models.

A LISA model basically consists of two parts, on the one hand the specification of the resources of the target architecture and on the other hand the description of the instruction-set, behavior and timing. The first one is captured in the *resource section*, while the latter one is described by the *LISA operation graph*¹. Both are described in the following sections.

The frontend of the synthesis tool extracts the architectural information from the LISA model and maps this information to the elements of the UDL. However, these elements are represented by an internal data-structure and thus not suitable for an explanation of the LISA frontend.

¹In publications sometimes referred to as LISA operation tree.

The extraction of architectural information is explained on the basis of pseudo hardware descriptions instead.

5.1 Resource Section

The resource section defines the resources of the ASIP such as storage elements, functional units and pipelines. All storage elements are declared with a global scope by specifying a data-type, identifier and optionally a resource-type. In example 5.1 the declaration “`REGISTER int R[0..15];`” instantiates 16 elements of type integer, which can be accessed by the identifier `R`. The keyword `REGISTER` identifies the type of this resource. A specialization of registers are `PROGRAM_COUNTER` and `CONTROL_REGISTER`. The keyword `PIN` describes the interface resources of the processor. Resources without resource-type specifier are considered to be signals which change their value *immediately* after value assignment.

The memory is specified accordingly, here, the keyword `RAM` is used. The size is defined by the number of blocks the memory consists of (`SIZE`) and the bit size of each block (`BLOCKSIZE`). The second parameter of the blocksize specification defines the sub-blocksize, the smallest accessible portion of the memory.

The pipeline definition comprises the pipeline name, an ordered list of pipeline stage names and the pipeline register elements between pipeline stages. As shown in example 5.1, the definition of pipeline registers is not tailored to a particular transition from one pipeline stage to another. In order to simplify ASIP modeling as well as to achieve speedups in simulation time the specification of the pipeline registers is unified. The specified pipeline register elements represent the union of all required elements per pipeline. Additionally, functional units are declared by the keyword `UNIT` followed by a unique name and a set of LISA operations.

The RTL hardware model structure can be derived explicitly from the specification of the pipeline and implicitly from the global scope of the defined resources. The LISA frontend maps the architecture to a hierarchical model structure: The first level comprises three modules grouping all registers, grouping all memories and encapsulating the pipeline structure. The pipeline module is further made up of modules each representing a particular pipeline stage or pipeline register. Finally, the pipeline stages cover an arbitrary number of modules, which represent the functional units defined in the resource section.

Further information about the RTL hardware model generation from the resource section is presented in [23], [97], [98] and [99].

```

RESOURCE {

    /* Memory definition: */
    RAM int mem_prog {
        SIZE(0x1000);
        BLOCKSIZE(32,32);
        FLAGS(R|X);
    };

    RAM int mem_data {
        SIZE(0x1000);
        BLOCKSIZE(32,32);
        FLAGS(R|W);
    };

    /* Global Register Resources: */
    PROGRAM_COUNTER unsigned short FPC;
    REGISTER         unsigned short BPC;
    REGISTER         bool          BPC_valid;
    REGISTER         int           R[0..15];

    /* Input/Output pins of the core: */
    PIN IN           int           input;
    PIN OUT          int           output;

    /* Global signals (unlocked) */
    int op_1, op_2;

    /* Pipeline and pipeline register definition: */
    PIPELINE pipe = { FE; DE; ME; EX; WB };
    PIPELINE_REGISTER IN pipe {
        unsigned int pc;
        int operand_1;
        int operand_2;
        int operand_1_reg_num;
        int operand_2_reg_num;
        int result;
        int result_reg_num;
    };

    /* UNIT definition to form functional units: */
    UNIT Fetch      { fetch; };
    UNIT Forward    { forward_operand_1_mem; };
    UNIT MemAccess  { data_mem_read, cm_mem_read, mac_mem; };
    UNIT Alu        { ADDI, SUBI, MOVI, MOVN, CMPI; };
    UNIT Branches   { BNE, BGT, BUNC; };
    UNIT AddrGeneration { mem_reg_mov, imm_disp_instr; };
    UNIT WriteBack  { MMR, writeback; };
}

```

Example 5.1. Example LISA resource section

5.2 LISA Operations

The architecture, including the instruction-set, is defined by a directed acyclic graph. The nodes of this graph are LISA operations op . Each node may contain information about the assembly syntax, the binary encoding of the instruction-set, the behavior and timing of the architecture. Generally, this information is specified independently of each other, therefore, focusing one type of information only, it is useful to refer to the *coding graph*, the *behavior graph*, the *syntax graph*, etc. This is illustrated in figure 5.1. The architectural information represented by a single LISA operations might be independent from other operations, which is reflected by unconnected nodes in the graphs of figure 5.1.

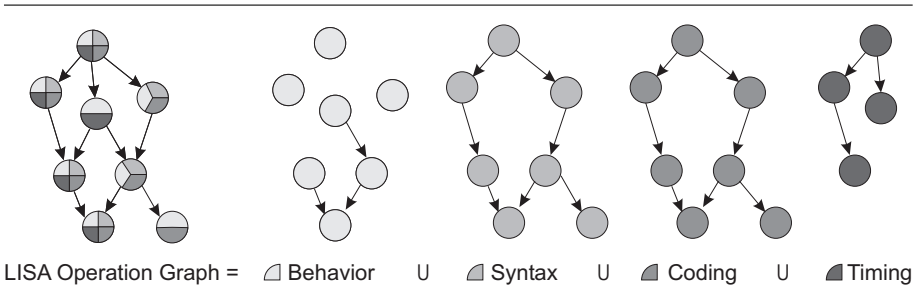


Figure 5.1. Combining information: LISA operation graph

The different types of information covered by a LISA operation is captured in *sections*, as listed in table 5.1.

Table 5.1. List of LISA sections

Section		Purpose
Declare	S_D	Declarations
Coding	S_C	Description of the binary representation of the instruction-set
Syntax	S_Y	Description of the assembly syntax
Activation	S_A	Timing description
Behavior	S_B	State transition functions
Expression	S_E	Extraction of resource values

The declare section S_D is used to establish the LISA operation graph, as defined in the following.

LISA 5.1: Declare Section

The declare section covers the definition of *instances*, *groups* and *labels*.

An instance refers to one particular LISA operation, whereas a group refers to one LISA operation of a specified set. Labels represent a certain part of the instruction coding, as defined in the coding section (section 5.3.1).

The instances, groups and labels defined in this section are used within other sections of the same LISA operation. In this way, sections of different type, for example the behavior section and the coding section, are connected to each other.

5.3 LISA Operation Graph

The instruction-set, behavior, timing, etc. of the architecture is captured by a graph structure, as defined in 5.2.

LISA 5.2: LISA Operation Graph

\mathcal{OP} describes the set of LISA operations op . The LISA operation graph is defined as a directed acyclic graph by $\mathcal{G} = \langle \mathcal{OP}, (\text{Instances, Groups}) \rangle$ with nodes op and relation (edges) of type instance or of type group. The LISA operation graph describes the target architecture with regard to the instruction-set, the behavior and the timing.

There are two special operations in the graph, each representing a root node for a path through the graph. The root node of the instruction-set description is called *coding root* operation op_{CR} . The directed acyclic graph for the instruction-set description is spanned by the coding section S_C and the syntax section S_Y . The root node for the behavioral (S_B , S_E) and temporal (S_A) description is called *main* operation m . Given a particular path, to each operation in this path the *level* n , written op^n , can be annotated. The annotation of the level in a path is also used for all following definitions and naming conventions. For example, it can be written for the path $op_a, op_b, op_c, op_d, op_e, op_f$:

$$\langle op_a^1, op_b^2, \dots, op_e^{n-1}, op_f^n \rangle = \left\langle \begin{pmatrix} S_{D_a}^1 \\ S_{C_a}^1 \\ S_{Y_a}^1 \\ S_{A_a}^1 \\ S_{B_a}^1 \\ S_{E_a}^1 \end{pmatrix}, \begin{pmatrix} S_{D_b}^2 \\ S_{C_b}^2 \\ S_{Y_b}^2 \\ S_{A_b}^2 \\ S_{B_b}^2 \\ S_{E_b}^2 \end{pmatrix}, \dots, \begin{pmatrix} S_{D_e}^{n-1} \\ S_{C_e}^{n-1} \\ S_{Y_e}^{n-1} \\ S_{A_e}^{n-1} \\ S_{B_e}^{n-1} \\ S_{E_e}^{n-1} \end{pmatrix}, \begin{pmatrix} S_{D_f}^n \\ S_{C_f}^n \\ S_{Y_f}^n \\ S_{A_f}^n \\ S_{B_f}^n \\ S_{E_f}^n \end{pmatrix} \right\rangle$$

The index representing the level of an operation or section in one particular path is of course not a unique identifier, as the same LISA operation might be part of different paths in the graph. In order to distinguish between several paths, the term *context* is used to identify the set of constraints defined by a particular path through the graph.

5.3.1 Coding Section

The *coding section* S_C covers information about the binary representation of the instruction-set and consists of a sequence of coding elements. The coding elements might be either a *terminal* or a *non-terminal* coding element. Coding element, terminal Coding element, non-terminal:

LISA 5.3: Terminal Coding Element

A *terminal coding element* t is a sequence of bits “0”, “1” or “x”. The value “x” represents a don’t care bit which may be either “0” or “1”. The length of the coding element $Len(t)$ is defined as the number of bits in the sequence.

LISA 5.4: Non-Terminal Coding Element

A non-terminal coding element w is a wildcard for an operation referred to by an instance or a group declared in section S_D of the same operation. The referred operation must contain a coding section S_C that represents an unambiguous bit pattern in the scope of coding element w . The length of the element on level n is $Len(w^n) = Len(S_C^{n+1})$.

Therefore, all coding sections S_C^{n+1} grouped by a non-terminal coding element $S_C^{n+1} \epsilon w^n$ are of equal bit width. The coding section is defined as follows:

LISA 5.5: Coding Section

A coding section S_C is defined as an ordered concatenation (\leftarrow) of coding elements c , denoting either non-terminal coding elements w , defined in 5.4, or terminal coding elements t , defined in 5.3. The concatenation is written

$$S_C(c_k, c_{k-1}, \dots, c_1) = c_k \leftarrow c_{k-1} \leftarrow \dots \leftarrow c_1, k \in \mathbb{N}.$$

The length of the coding section S_C is defined by $Len(S_C) = \sum_{i=1}^k Len(c_i)$.

The properties of coding elements and coding sections, such as the *coding position*, *coding coverage* and *coding mask*, are explained in the following.

Position of a coding element or coding section

A coding element or coding section generally represents only a portion of the whole instruction word. Its position within the complete instruction word can be recursively calculated by:

$$Pos(S_{CR}^1) = 0, \quad (5.1)$$

$$Pos(c_k^n) = Pos(S_C^n) + \sum_{i=1}^{k-1} Len(c_i^n), n \geq 1, n \in \mathbb{N}, \quad (5.2)$$

$$Pos(S_C^{n+1}) = Pos(c^n), n \geq 1, n \in \mathbb{N}. \quad (5.3)$$

The LISA operation that represents the starting point of an instruction encoding description is called *coding root*, its coding section is referred by S_{CR} . The instruction bit width is equal to $Len(S_{CR})$, the coding position is $Pos(S_{CR}) = 0$.

Coding coverage of a coding element or coding section

The number of bit-patterns, which are represented by a particular coding element or coding section, is called *coding coverage*. The knowledge about the coding coverage is required to determine the necessity to decode particular bits. For example, a terminal coding element only consisting of 0s and 1s covers exactly one particular bit pattern of the specified portion in the instruction word, written $Cov(t) = 1$. A terminal coding element that consists of two don't care bits represents

$Cov(t) = 2^2 = 4$ different bit patterns. The coverage of a coding element or coding section can be recursively calculated from the leave nodes to the root of the LISA operation graph.

The coverage of a terminal coding element is calculated by

$$Cov(t) = 2^{\text{number of don't cares}} \quad (5.4)$$

The coverage of a non-terminal coding element on level n can be calculated by

$$Cov(w^n) = \sum_{\forall S_C^{n+1}} Cov(S_C^{n+1}) \quad (5.5)$$

The coverage of a coding section can be calculated with 5.4 and 5.5 by

$$Cov(S_C^n) = \prod_{i=1}^k Cov(c_i^n) : i, k \in \mathbb{N} \quad (5.6)$$

A so-called *full* coding coverage is given if

$$Cov(S_C^n) = 2^{Len(S_C^n)} \quad (5.7)$$

Coding mask of a coding element or coding section

The *coding mask* of a coding element or coding section represents the possible values at particular bit positions within the instruction word. The values “0” and “1” represent bits with a fixed value. They are specified by either non-terminal coding elements or by the subgraph. In other words, the coding mask is propagated from the leave nodes to the root nodes of the LISA operation graph. The value “?” represents an undeterminable bit on this level in the coding graph. The specification of different bits within the coding section is uncorrelated, thus more bit patterns may be covered by a particular coding mask, than actually existent in the subgraph.

Coding graph example

A simple coding graph is given in figure 5.2. The coding mask of section S_{C3} is “??” as both bit positions cannot be determined. This coding mask represents four bit patterns (“00”, “01”, “10”, “11”), although only two bit patterns are covered by the subgraph. The coding coverage is $Cov(S_{C3}) = 2$.

The coding mask for section S_{C1} is “?????1”. The bit at position zero is specified by the non-terminal coding element. Furthermore, the remaining bits of the coding mask are derived from coding masks on

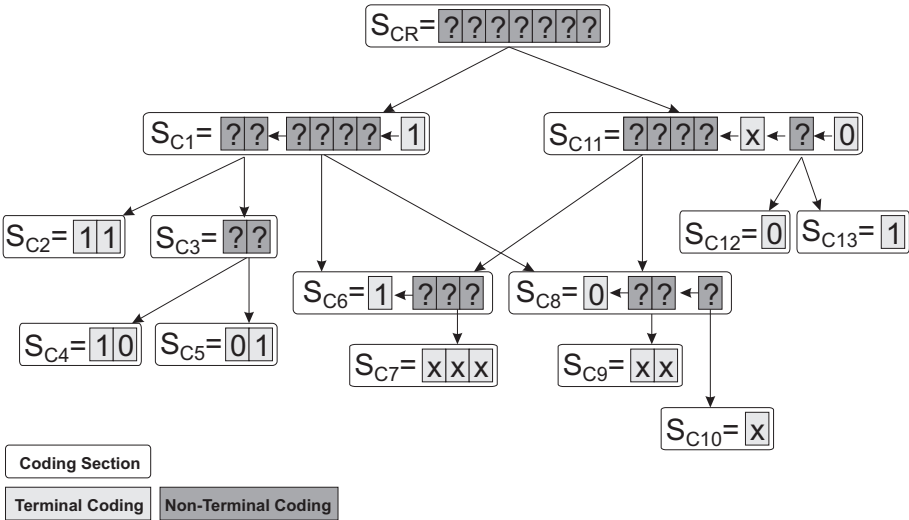


Figure 5.2. LISA coding graph

lower levels in the hierarchy. Thus, within section S_{C1} all other bits are undeterminable (they may be “0” or “1”).

There are two paths to the coding section S_{C6} in the exemplary coding graph. The coding section S_{C6} can be reached by the path S_{CR}, S_{C1} leading to a coding position $Pos(S_{C6}) = 1$ or S_{CR}, S_{C11} leading to a coding position $Pos(S_{C6}) = 3$. In this example, two different contexts exists for coding section S_{C6} .

Decoder generation

The LISA coding graph is utilized to generate the corresponding decode logic. The bit pattern of a terminal coding element can be compared directly with the instruction word. The bit patterns represented by a non-terminal coding element are derived from the corresponding subgraph. Only one of these bit patterns is valid regarding a given instruction word. Thus, the boolean OR operator is used to determine whether a non-terminal coding element is decoded. The concatenation of coding elements within a coding section is translated to the boolean AND operator, since all elements must be decoded. If a full coding coverage of coding section S_C is given (cf. equation 5.7), the decoding results from the subgraph need not be taken into account.

Decoder example

The pseudo-code in example 5.2 depicts the algorithm used to decode an instruction with regard to the instruction-set, which is given in figure 5.2.

Each coding section is represented by a variable d_i to represent the information whether the coding section has been decoded ($d_i = 1$) or not decoded ($d_i = 0$). All variables are initialized in line 02 and line 04. Coding sections which only cover don't care bits are always decoded, as any instruction matches this pattern. In this example this is the case for coding sections S_{C7} , S_{C9} and S_{C10} . Coding sections which only cover terminal coding elements can be decoded by comparing their coding mask to the corresponding bits of the instruction word. For example, coding section S_{C4} and S_{C5} are translated to lines 08 and 09 and coding section S_{C2} is translated to line 12.

In line 32 several decoding results are combined to decide whether coding section S_{C11} is decoded. Additionally, the coding mask needs to be compared to the corresponding bits of the instruction word. Certainly, don't care bits are not taken into account for this comparison, thus, in line 32 only the bit at position zero of the instruction word is compared to the value "0".

The instruction-set defined by example 5.2 covers only a subset of all possible coding bit patterns. The coverage can be recursively calculated, which leads to $Cov(S_{CR}) = 112$ different instructions. The maximum number of different instructions is $2^7 = 128$. Thus 16 instructions are invalid regarding the given definition in figure 5.2 (e.g. instruction "0011111"). An invalid instruction can be easily determined by evaluating the variable d_R , which equals "1" for every valid instruction and "0" for every invalid instruction.

The decoding information represented by the variables d_i is utilized by other portions of the generated hardware, which are derived, for example, from the behavior section or activation section.

5.3.2 Syntax Section

The S_Y covers the assembly syntax of the instruction-set and is comparable to the coding section defined above. The non-terminal syntax elements represent the mnemonics of the instruction-set. The syntax section is not required for an RTL hardware model generation. Interested readers may refer to [23] or [100].

```

01 // variables to represent the decoding result:  $d_i$  corresponds to  $S_{C_i}$ 
02  $d_R = d_2 = d_3 = d_4 = d_5 = d_6 = d_8 = d_{11} = d_{12} = d_{13} = 0$ ;
03 // full coding coverage, therefore always decoded (equation 5.7):
04  $d_7 = d_9 = d_{10} = 1$ ;
05
06 // Instruction bits at position 5 and 6 are compared:
07 switch( instruction(6..5) ){
08     case "10":  $d_4 = 1$ ;
09     case "01":  $d_5 = 1$ ;
10 }
11
12 if( instruction(6..5) == "11" ){  $d_2 = 1$ ; }
13
14 // First context of  $S_{C_6}$ 
15 if( instruction(4) == "1" ){  $d'_6 = d_7$ ; }
16
17 // First context of  $S_{C_8}$ 
18 if( instruction(4) == "0" ){  $d'_8 = d_9 \parallel d_{10}$ ; }
19
20 // Second context of  $S_{C_6}$ 
21 if( instruction(6) == "1" ){  $d''_6 = d_7$ ; }
22
23 // Second context of  $S_{C_8}$ 
24 if( instruction(6) == "1" ){  $d''_8 = d_9 \parallel d_{10}$ ; }
25
26 if( instruction(0) == "1" ){  $d_1 = (d_2 \parallel d_3) \&\& (d'_6 \parallel d'_8)$ ; }
27
28 if( instruction(1) == "0" ){  $d_{12} = 1$ ; }
29
30 if( instruction(1) == "1" ){  $d_{13} = 1$ ; }
31
32 if( instruction(0) == "0" ){  $d_{11} = (d''_6 \parallel d''_8) \&\& (d_{12} \parallel d_{13})$ ; }
33
34  $d_{S_{C_R}} = d_{S_{C_1}} \parallel d_{S_{C_{11}}}$ ;

```

Example 5.2. Decoder implementation for coding tree in figure 5.2

5.3.3 Behavior Section

The *behavior section* S_B covers the state update functions of the architecture. They are described using the

- C-programming language,
- the elements defined in the resource section and
- the instances and groups defined in the declare section S_D .

In this context, instances and groups refer to the behavior section of the respective LISA operations.

The C-code given in the LISA behavior section can be represented by Control Flow Graphs (CFGs). Figure 5.3 gives an example of behavior sections and the relation between different operations established via the behavior section [88].

```

// Elements defined in the          // Coding Section,
// resource section are:           // Syntax Section, etc. are omitted.
// use_bypass, reg_bypass,
// pipe_reg, instr, operand

OPERATION load_operand {          OPERATION ld_reg {
  DECLARE{                        BEHAVIOR {
    GROUP load = {ld_reg || ld_imm};      operand = R[instr.Extract(18,10)];
  }                                     }
  BEHAVIOR{                          }
    if(use_bypass==1)
      operand = reg_bypass;
    else
      load();
    pipe_reg = operand;
  }
}

```

Figure 5.3. LISA behavior sections

The operation `load_operand` describes the following behavior: An operand is read and then written to a pipeline register (`pipe_reg`). The operand may be either a register value (`R[]`) or an immediate value, which is extracted from the current instruction word (`instr`). Whether the register value or the immediate value is loaded depends on the coding specification (cf. section 5.3.1), not shown in the example. If the bypass is enabled (`use_bypass`), the operand value is read from the bypass register (`reg_bypass`).

The left CFG of figure 5.4 represents the behavior code given in operation `load_operand`. The group `load` within the behavior section represents behavior descriptions which are referred to by this group (either `ld_reg` or `ld_imm`). The syntax is derived from a C-function call, and also the behavior is equal to the execution of a function in the C-programming language. In general, behavior descriptions are sequentially executed, which also includes function calls and behavioral descriptions represented by groups or instances.

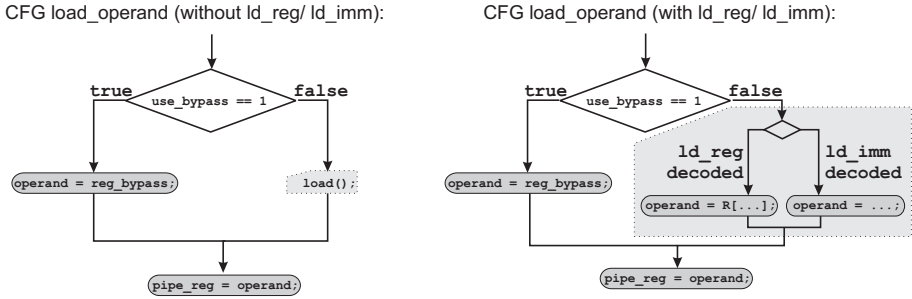


Figure 5.4. CFG representation of LISA behavior sections

As a particular behavioral description might be utilized in several operations, even within the same clock-cycle, the code is automatically duplicated to represent the corresponding hardware. The behavioral description is *inlined* to ensure the correct timing². Therefore, the CFGs of the operation `load_operand`, `ld_reg` and `ld_imm` are merged to one CFG. As can also be seen on the right of figure 5.4, a control flow statement is inserted into the `load_operand` CFG to control the execution of the `ld_reg` CFG and `ld_imm` CFG. The decision about which operation to call can be carried out accordingly to the information covered in the coding section (cf. section 5.3.1).

Data transfer between the operation `load_operand`, `ld_reg` and `ld_imm` is enabled by the resource named `operand`. The resource `operand` is defined in the resource section as signal because the value has to be transmitted within the current clock cycle.

5.3.4 Expression Section

The *expression section* S_E is part of the behavioral description of the architecture. Different from a behavior section, the expression section returns a particular value, for example the content of a register. Arithmetic or boolean calculations are not allowed within this section. Groups and instances are used to refer to an expression section from a behavior section. For example, the behavior sections of operations `ld_reg` and `ld_imm` in figure 5.3 can be replaced by expression sections.

²The LISA simulator executes the same code multiple times within a single control step, for example within one clock cycle, which is obviously not possible in the real hardware implementation.

5.3.5 Activation Section

The *activation section* S_A describes the temporal relation between LISA operations.

LISA 5.6: Activation Section

The LISA activation section S_A is a set of activation elements a_i , each of them an instance or group declared in section S_D of the same operation. An activation schedules an operation for execution. The activation may be conditional to compile-time and/or run-time conditions.

The decoding results, as described in section 5.3.1, are used to activate one operation out of group. Thus, an activation always refers to one operation only. The correct point in time for execution depends on the pipeline organization and the current pipeline status. In general, operations are assigned to certain pipeline stages, thus a temporal relation between operations is described, which is utilized for simulation as well as hardware generation. Pipeline control functions are used to modify the regular execution of operations, such as pipeline flushes, which prevent the execution, and pipeline stalls, which delay the execution.

Figure 5.5 gives an example of a LISA activation graph. The operation main, which is called once by the simulator for every control step, activates the operation op_1 . As the operation main is only the entry point for the LISA simulator, the behavior section S_B of operation main is not considered for hardware generation (the operation main is comparable to a VHDL testbench, which is, in general, also not synthesisable). In this example, op_2 and op_3 are members of the same group and therefore exclusively executed. The operations op_5 and op_6 are activated in parallel. The activation of op_4 is sensitive to run-time conditions. Beside that graph, the assignment of operations to pipeline stages and the resulting temporal relation is also illustrated. For example, the temporal difference between the execution of op_3 and op_5 is $\Delta t = 2$ cycles, if no pipeline control, such as flush or stall, influences the execution.

The execution order defined explicitly by the activation section and implicitly by the LISA simulator behavior must be translated to control logic in hardware. For every operation contained in the LISA activation graph, it must be checked whether the operation is activated. Depending on its activation, the activation must be propagated to the control logic that represents the subsequent activation sections. Pipeline registers are inserted automatically to ensure a timing, which corresponds to the

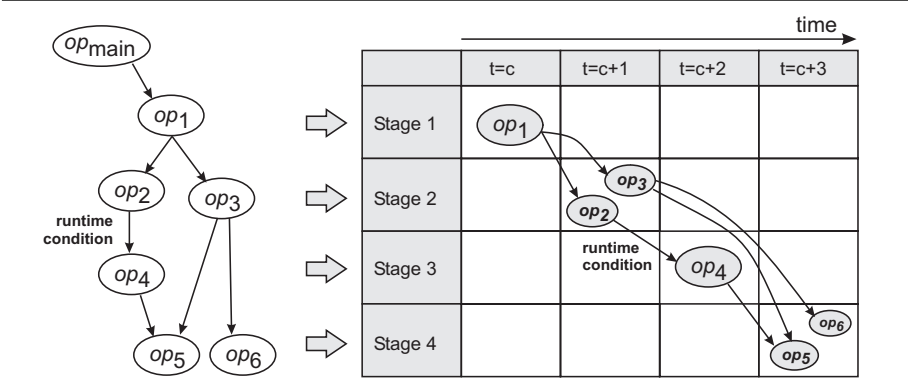


Figure 5.5. LISA activation graph and the spatial and temporal relation

LISA simulator behavior. Additionally, the activation may depend on run-time conditions, for example, particular register values. In LISA such conditions are expressed using the C-programming language within the activation section.

Figure 5.6 illustrates a possible implementation. The control signal a_1 indicates that operation op_1 has to be executed. The control signal a_1 is always set to 1, which corresponds to the activation from main to op_1 in every clock cycle. The decoding results (cf. section 5.3.1) are represented by the variables d_2 and d_3 which are utilized to set either a_2 or a_3 to 1. External conditions are evaluated in the case of control signal a_4 . As operations may be activated in several contexts, the or-gate in the last stage combines all control signals from different contexts to a single signal a_5 . Whenever possible, multiplexers are replaced by appropriate signal assignments in the real implementation. For the reason of clarity this optimization is omitted here. The signals a_1 to a_6 are used to steer the execution of behavior sections of the corresponding operations. As can be seen in figure 5.6, the correct temporal behavior is achieved by preserving the pipeline structure and introducing registers between consecutive pipeline stages.

5.4 Representing Exclusiveness in Conflict and Compatibility Graphs

The information extracted so far is required to instantiate a first hardware representation of the target architecture. To perform optimizations, especially, but not only resource sharing, information about exclusiveness is essential. It has been shown in chapter 4 that this infor-

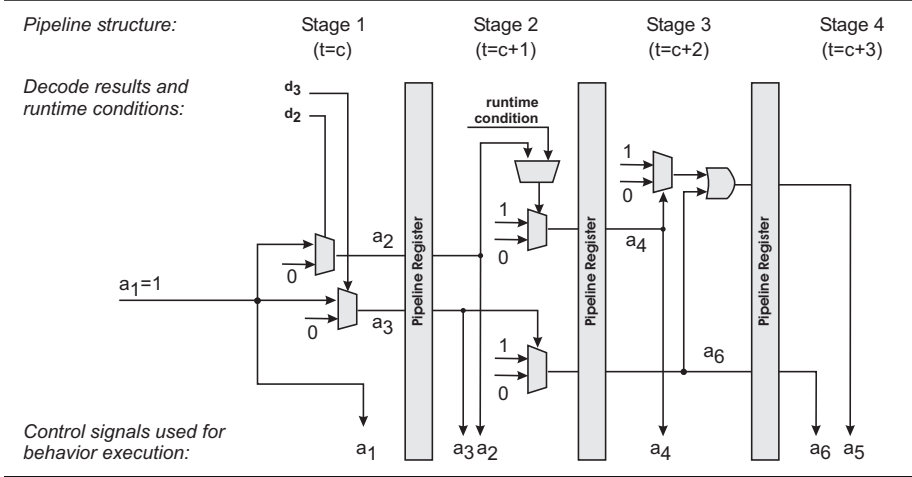


Figure 5.6. Pseudo hardware implementation of the LISA activation graph

mation is hardly extractable on the RTL but explicitly available in the LISA model.

A data structure is required that preprocesses exclusiveness information and allows queries with a low computational complexity. This demand is fulfilled by an undirected graph $\mathcal{G} = \langle \mathcal{OP}, E \rangle$. \mathcal{OP} is the set of vertices, each vertex represents a LISA operation. E is the set of edges, each edge represents a commutative relation between two LISA operations.

This section focuses on the extraction of exclusiveness information and the appropriate graph implementation to store this information. A query to this information is performed in $\mathcal{O}(1)$ time [101].

5.4.1 Graph Definitions

Two sorts of relations are possible when handling exclusiveness information: “Operations A and B are executed exclusively” and “Operations A and B can be executed concurrently”. On the resource sharing background, exclusiveness is called *compatibility*, the opposite relation *conflict* (definition 5.8). Conflict and compatibility relations are complementary to each other (equation 5.9).

$$\begin{aligned}
 op_x \otimes op_y &:= \text{Operation } op_x \text{ is compatible with Operation } op_y \\
 op_x \ominus op_y &:= \text{Operation } op_x \text{ conflicts with Operation } op_y
 \end{aligned} \tag{5.8}$$

$$\begin{aligned} \overline{op_x \otimes op_y} &\Leftrightarrow op_x \ominus op_y \\ \overline{op_x \ominus op_y} &\Leftrightarrow op_x \otimes op_y \end{aligned} \quad (5.9)$$

With this concept two classes of graphs are possible: Compatibility graphs $\mathcal{G}_{\text{compat}}$ (equation 5.10) and conflict graphs $\mathcal{G}_{\text{conflict}}$ (equation 5.11).

$$\begin{aligned} \mathcal{G}_{\text{compat}} &= \langle \mathcal{OP}, E_{\text{compat}} \rangle \\ E_{\text{compat}} &= \{(op_x, op_y) | op_x, op_y \in \mathcal{OP}, op_x \otimes op_y\} \end{aligned} \quad (5.10)$$

$$\begin{aligned} \mathcal{G}_{\text{conflict}} &= \langle \mathcal{OP}, E_{\text{conflict}} \rangle \\ E_{\text{conflict}} &= \{(op_x, op_y) | op_x, op_y \in \mathcal{OP}, op_x \ominus op_y\} \end{aligned} \quad (5.11)$$

Due to the complementary property of both relations, a conflict graph is the complementary graph of the compatibility graph and vice versa (equation 5.12). Figure 5.7 gives an example of conflict and compatibility graphs.

$$\begin{aligned} \mathcal{G}_{\text{conflict}} &= \langle \mathcal{OP}, \{(op_x, op_y) | (op_x, op_y) \notin E_{\text{compat}}\} \rangle = \overline{\mathcal{G}_{\text{compat}}} \\ \mathcal{G}_{\text{compat}} &= \langle \mathcal{OP}, \{(op_x, op_y) | (op_x, op_y) \notin E_{\text{conflict}}\} \rangle = \overline{\mathcal{G}_{\text{conflict}}} \end{aligned} \quad (5.12)$$

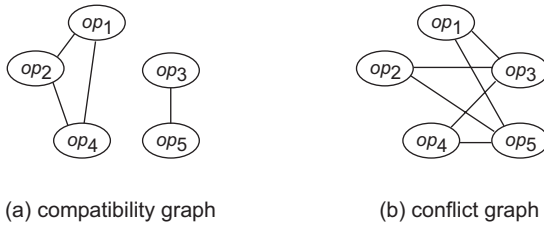


Figure 5.7. Compatibility graph and conflict graph

5.4.2 Creating Compatibility and Conflict Graphs

The information about exclusiveness is extracted by recursively processing the directed acyclic LISA operation graph with a depth-first strategy. Thus the construction of conflict or compatibility graphs from

the LISA operation graph requires the merging of subgraphs, which represent relations between sub-sets of LISA operations, into a common graph. Due to the complementary property of conflict and compatibility graphs it is sufficient to describe the merging operations for one class only. Here, the conflict graphs $\mathcal{G}_{\text{conflict},1}$ and $\mathcal{G}_{\text{conflict},2}$ are selected to be merged into one global graph $\mathcal{G}_{\text{conflict,global}}$ (equation 5.13).

$$\begin{aligned}\mathcal{G}_{\text{conflict},1} &= \langle \mathcal{OP}_1, E_1 \rangle \\ \mathcal{G}_{\text{conflict},2} &= \langle \mathcal{OP}_2, E_2 \rangle \\ \mathcal{G}_{\text{conflict,global}} &= \langle \mathcal{OP}_{\text{global}}, E_{\text{global}} \rangle\end{aligned}\tag{5.13}$$

For the merging process it is necessary to distinguish whether *compatibility* or *conflict* relations should be established for pairs of vertices of different graphs. The merging of *compatible* conflict graphs is the very basic merge operation as no new conflict edges have to be inserted. All conflicts will persist by creating the union of both sets of operations \mathcal{OP}_1 and \mathcal{OP}_2 and the union of edges E_1 and E_2 . For the merge of *compatible* conflict graphs the operator \cup_{\oplus} is defined according to equation 5.14. An example of a merge of two compatible conflict graphs is given in figure 5.8. In this figure, the LISA operations op_1 and op_2 appear in both source graphs, because they are referred to by multiple operations in the coding graph.

$$\mathcal{G}_{\text{conflict},\oplus} = \mathcal{G}_{\text{conflict},1} \cup_{\oplus} \mathcal{G}_{\text{conflict},2} := \langle \mathcal{OP}_1 \cup \mathcal{OP}_2, E_1 \cup E_2 \rangle\tag{5.14}$$

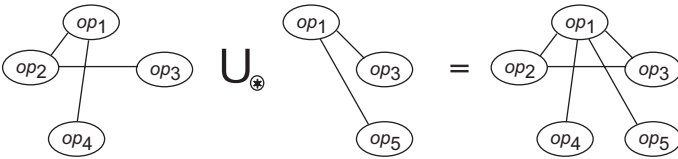


Figure 5.8. Example of merging compatible conflict graphs

The complexity of the merge of compatible conflict graphs depends on the graph implementation. If adjacency lists are used, only graph nodes and no edges must be considered during the merging. Therefore, the complexity is given by $\mathcal{O}(|\mathcal{OP}_1| + |\mathcal{OP}_2|)$. If the result is stored in $\mathcal{G}_{\text{conflict},1}$ the complexity is only $\mathcal{O}(|\mathcal{OP}_2|)$, because the vertex of $\mathcal{G}_{\text{conflict},2}$ can be inserted without considering any vertex in $\mathcal{G}_{\text{conflict},1}$.

If conflicts should be established between the two graphs $\mathcal{G}_{\text{conflict},1}$ and $\mathcal{G}_{\text{conflict},2}$, new conflict edges have to be inserted for each pair of nodes from different subgraphs. All conflicts existing in the original graphs will persist. The operator \cup_{\ominus} is defined for the merging of *conflicting* conflict graphs according to equation (5.15). An example is given in figure 5.9.

$$\begin{aligned}
 \mathcal{G}_{\text{conflict},\ominus} &= \mathcal{G}_{\text{conflict},1} \cup_{\ominus} \mathcal{G}_{\text{conflict},2} \\
 &:= \langle \mathcal{OP}_1 \cup \mathcal{OP}_2, \\
 &\quad E_1 \cup E_2 \cup \{(op_x, op_y) \mid op_x \in \mathcal{OP}_1 \wedge op_y \in \mathcal{OP}_2\} \rangle
 \end{aligned}
 \tag{5.15}$$

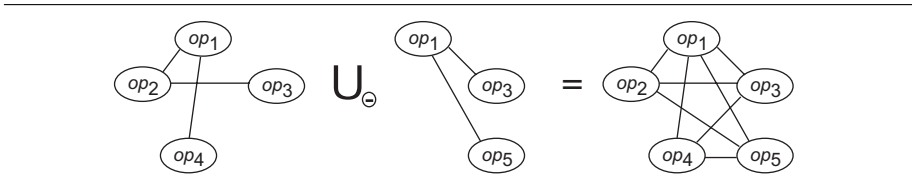


Figure 5.9. Example of merging conflicting conflict graphs

For an analysis of the complexity a graph representation based on adjacency lists is assumed. Therefore, the complexity for merging conflicting conflict graphs relies on the number of newly inserted edges, given by the product of $|\mathcal{OP}_1|$ and $|\mathcal{OP}_2|$. Thus the complexity is $\mathcal{O}(|\mathcal{OP}_1| \cdot |\mathcal{OP}_2|)$.

5.5 Exclusiveness Information on the Level of LISA Operations

The conflict and compatibility graphs defined in section 5.4 are used to retain information about exclusiveness on the level of LISA operations. This exclusiveness graph has to be built by using the structural information given in the LISA operation graph. The LISA operation graph is traversed step by step with the depth-first strategy. Local graphs, which represent the exclusiveness information provided by LISA operation subgraphs, are merged into one graph until the root of the LISA operation graph is reached. With this strategy, a large number of merge operations on local graphs are necessary. Under the constraint that conflicts have to be preserved the merge of conflict graphs is less complex than the merge of compatibility graphs. Therefore, conflict graphs are chosen for building up the global LISA operation graph.

5.5.1 Conflict Graph Creation

A recursive algorithm is implemented to create the LISA operation conflict graph. Here, a single recursion step operates on a particular LISA operation and has to merge all conflict graphs of subsequent LISA operations, as defined by the directed acyclic LISA operation graph. Therefore, the LISA operation sections have to be analyzed whether they create conflicts or compatibilities.

Coding section S_C

According to 5.3.1, non terminal coding elements w refer to a group of LISA operations $\mathcal{OP}_{\text{Group}}$. All pairs of LISA operations $\{(op_x, op_y) | op_x, op_y \in \mathcal{OP}_{\text{Group}}\}$ define local mutually exclusiveness by definition. Therefore, the conflict graphs \mathcal{CG}_{op} of all group elements $op \in \mathcal{OP}_{\text{Group}}$ can be merged into the coding element conflict graph $\mathcal{CG}_{\mathcal{OP}_{\text{Group}}}$ without insertion of additional conflicts (equation 5.16).

$$\mathcal{CG}_{\mathcal{OP}_{\text{Group}}} = \bigcup_{\forall op \in \mathcal{OP}_{\text{Group}}}^{\otimes} \mathcal{CG}_{op} \quad (5.16)$$

Concatenated coding elements c (both instances and groups) each represent a portion of the same instruction and thus encode hardware which is concurrently executed. Thus, conflicts instead of exclusiveness relations must be established on the level of a coding section S_C for coding elements. Therefore, the local conflict graphs $\mathcal{CG}_c, c \in S_C$ have to be merged with conflicts (equation 5.17).

$$\mathcal{CG}_{S_c} = \bigcup_{\forall c \in S_c}^{\ominus} \mathcal{CG}_c \quad (5.17)$$

Activation section S_A

Activation elements $a \in S_A$ are executed concurrently. The conflict graph for the activation section \mathcal{CG}_{S_A} has to be built up by merging the local conflict graphs for all activation elements with conflicts:

$$\mathcal{CG}_{S_A} = \bigcup_{\forall a \in S_A}^{\ominus} \mathcal{CG}_a \quad (5.18)$$

If an activation element a leads to a single LISA operation op , the activation element's conflict graph contains only the conflict graph for LISA operation op : $\mathcal{CG}_a = \mathcal{CG}_{op}$. For an activation a of a group $\mathcal{OP}_{\text{Group}}$, the same exclusiveness rules apply as described by (5.16). Therefore, the conflict graph for this activation element is given by $\mathcal{CG}_a = \mathcal{CG}_{\mathcal{OP}_{\text{Group}}}$.

Activations can be conditional by the utilization of if-else statements, as explained in section 5.3.5. Therefore exclusiveness can be concluded between operations activated by a_{if} and operations activated by a_{else} , with $a_{\text{if}}, a_{\text{else}} \in S_A$. For a single conditional activation $a_{\text{conditional}} = \langle a_{\text{if}}, a_{\text{else}} \rangle$ the conditional activation element's conflict graph $\mathcal{CG}_{a_{\text{conditional}}}$ is the compatible merge of both conflict graphs $\mathcal{CG}_{a_{\text{if}}}$ and $\mathcal{CG}_{a_{\text{else}}}$:

$$\mathcal{CG}_{a_{\text{conditional}}} = \mathcal{CG}_{a_{\text{if}}} \cup_{\otimes} \mathcal{CG}_{a_{\text{else}}} \quad (5.19)$$

This rule has to be applied recursively for all nested conditional activation elements.

Behavior section S_B

A link between LISA operations may be established via the behavior section S_B , as described in section 5.3.3. Here, the referred behavior code is duplicated for each reference and inlined in the behavior section of the referring operation. Therefore, exclusiveness aspects on the level of LISA operations are *not* influenced by the behavior section S_B .

Recursive conflict graph creation

The steps of a conflict graph creation, elaborated above, are now combined in a single recursive algorithm. The resulting conflict graph contains the entire information about exclusive execution with a granularity of LISA operations. The recursive algorithm for the creation of a conflict graph of a particular LISA operation op is given in figure 5.10. In order to use the exclusiveness information given by conditional activations, conflicts between exclusively activated coding elements must be avoided. Therefore, the conflict graph \mathcal{CG}_{S_A} must be created before \mathcal{CG}_{S_C} and coding elements used in the activation section must be omitted for the creation of \mathcal{CG}_{S_C} . The recursive algorithm for the analysis of activation sections is described in figure 5.11. The analysis of behavior calls is the last step after all exclusiveness relations for coding elements have already been analyzed. Thus, behavior calls of coding elements must be ignored in the behavior call analysis step.

After discussing one recursion step, the creation of the global conflict graph is discussed in the following. For its creation, the topology of the pipeline needs to be taken into account. Until now, the LISA operation conflict graph may contain compatibility relations for operations in different pipeline stages, as only the decoding of a single instruction is analyzed. It must be considered that different instructions are executed concurrently in different pipelines and pipeline stages. Therefore,

```

CREATELISAOPERATIONCONFLICTGRAPH(op)
1  // conflict graph for operation op
2   $\mathcal{CG}_{op} \leftarrow \{\{op\}, \{\}\}$ 
3
4  // Conflict Graphs for Activation Section
5   $\mathcal{CG}_{S_A} \leftarrow \text{CREATEACTIVATIONCONFLICTGRAPH}(S_A)$ 
6   $\mathcal{CG}_{op} \leftarrow \mathcal{CG}_{op} \cup_{\oplus} \mathcal{CG}_{S_A}$ 
7
8  // Conflict Graphs for Coding Section
9   $\mathcal{CG}_{S_C} \leftarrow \emptyset$ 
10 for  $\{\mathcal{OP}_{\text{Group}} \mid \mathcal{OP}_{\text{Group}} \in S_C \wedge \mathcal{OP}_{\text{Group}} \notin S_A\}$  do
11   // non terminal coding element conflict graph
12    $\mathcal{OP}_{\text{Group}} \leftarrow \emptyset$ 
13   for  $\{op \mid op \in \mathcal{OP}_{\text{Group}}\}$  do
14      $\mathcal{CG}_{op} \leftarrow \text{CREATELISAOPERATIONCONFLICTGRAPH}(op)$ 
15      $\mathcal{CG}_{op} \leftarrow \mathcal{OP}_{\text{Group}} \cup_{\oplus} \mathcal{CG}_{op}$  // (equation 5.16)
16   endfor
17    $\mathcal{CG}_{S_C} \leftarrow \mathcal{CG}_{S_C} \cup_{\oplus} \mathcal{CG}_{\mathcal{OP}_{\text{Group}}}$  // (equation 5.17)
18 endfor
19  $\mathcal{CG}_{op} \leftarrow \mathcal{CG}_{op} \cup_{\oplus} \mathcal{CG}_{S_C}$ 
20
21 // Conflict Graphs for Behavior Section
22  $\mathcal{CG}_{S_B} \leftarrow \emptyset$ 
23 for  $\{op_{bc} \mid op_{bc} \in S_B \wedge op_{bc} \notin S_C\}$  do
24    $\mathcal{CG}_{S_B} \leftarrow \mathcal{CG}_{S_B} \cup_{\oplus} \mathcal{CG}_{op_{bc}}$ 
25 endfor
26  $\mathcal{CG}_{op} \leftarrow \mathcal{CG}_{op} \cup_{\oplus} \mathcal{CG}_{S_B}$ 
27 return  $\mathcal{CG}_{op}$ 

```

Figure 5.10. Conflict graph creation for LISA operations

conflicts for all pairs of operations in different pipelines or pipeline stages must be added. The algorithm for the creation of the global conflict graph is given in figure 5.12.

5.5.2 Algorithmic Complexity of Conflict Analysis

The complexity for the global conflict graph creation can be determined with the help of an operation graph degenerated to a list. The list length is n_{op} , the *level* of a node is its distance to the root node plus one. Thus the coding root has $level_{\text{root}} = 1$, the leaf $level_{\text{leaf}} = n_{op}$. The recursion step on the first level has to work on $n_{op} - 1$ child operations. For this case the recursion complexity due to conflict graph creation is given by equation (5.20).

```

CREATEACTIVATIONCONFLICTGRAPH( $S_A$ )
1  // creates a conflict graph for an activation section
2   $\mathcal{CG}_{S_A} \leftarrow \emptyset$ 
3  for  $\{a \mid a \in S_A\}$  do
4    // Conflict Graph for the current Activation Element
5     $\mathcal{CG}_a \leftarrow \emptyset$ 
6    if  $a \equiv op_{\text{Group}}$  then
7      // Activation Element is a group
8       $\mathcal{CG}_w \leftarrow \emptyset$ 
9      for  $\{op \mid op \in \mathcal{OP}_{\text{Group}}\}$  do
10      $\mathcal{CG}_{op} \leftarrow \text{CREATELISAOPERATIONCONFLICTGRAPH}(op)$ 
11      $\mathcal{CG}_w \leftarrow \mathcal{CG}_w \cup_{\otimes} \mathcal{CG}_{op}$ 
12   endfor
13    $\mathcal{CG}_a \leftarrow \mathcal{CG}_w$ 
14   else if  $a \equiv a_{\text{conditional}} = \langle a_{\text{if}}, a_{\text{else}} \rangle$  then
15     // Activation Element is a Conditional Activation
16      $\mathcal{CG}_{a_{\text{if}}} \leftarrow \text{CREATEACTIVATIONCONFLICTGRAPH}(a_{\text{if}})$ 
17      $\mathcal{CG}_{a_{\text{else}}} \leftarrow \text{CREATEACTIVATIONCONFLICTGRAPH}(a_{\text{else}})$ 
18      $\mathcal{CG}_a \leftarrow \mathcal{CG}_{a_{\text{if}}} \cup_{\otimes} \mathcal{CG}_{a_{\text{else}}}$ 
19   else
20     // Activation Element is a single LISA Operation
21      $\mathcal{CG}_a \leftarrow \text{CREATELISAOPERATIONCONFLICTGRAPH}(op)$ 
22   endif
23    $\mathcal{CG}_{S_A} \leftarrow \mathcal{CG}_{S_A} \cup_{\oplus} \mathcal{CG}_a$ 
24 endfor
25 return  $\mathcal{CG}_{S_A}$ 

```

Figure 5.11. Conflict graph creation for activation elements

$$\text{Recursion Complexity: } \sum_{L=0}^{n_{\text{op}}-1} L = \frac{(n_{\text{op}}) \cdot (n_{\text{op}} - 1)}{2} \Rightarrow \mathcal{O}(n_{\text{op}}^2) \quad (5.20)$$

In this equation, the possible insertion of conflicts during the graph merging process is omitted. The complexity of all conflict insertions is given by the total number of possible conflicts in the global LISA operation conflict graph. This complexity already includes the insertion of conflicts for operations in different pipeline stages. Therefore, the total complexity of conflict insertion is given by $\mathcal{O}(n_{\text{op}}^2)$.

The complexity of the combination of recursion and conflict insertion results in a total complexity of $\mathcal{O}(n_{\text{op}}^2)$ for the **global LISA operation conflict graph creation**.

```

CREATEGLOBALCONFLICTGRAPH ()
1   $\mathcal{CG}_{\text{global}} \leftarrow \emptyset$ 
2
3   $\mathcal{CG}_{\text{global}} \leftarrow \text{CREATELISAOPERATIONCONFLICTGRAPH}(\text{main})$ 
4
5  // Add Conflicts for Operations in different Pipeline Stages
6  for  $\{op_1 \mid op_1 \in \mathcal{CG}_{\text{global}}\}$  do
7    for  $\{op_2 \mid op_2 \in \mathcal{CG}_{\text{global}}\}$  do
8      if  $(\text{PIPE}(op_1) \neq \text{PIPE}(op_2)) \vee (\text{STAGE}(op_1) \neq \text{STAGE}(op_2))$  then
9         $\mathcal{CG}_{\text{global}} \leftarrow \mathcal{CG}_{\text{global}} \cup \{\{\}, \{(op_1, op_2)\}\}$ 
10       endif
11     endfor
12   endfor
13
14  return  $\mathcal{CG}_{\text{global}}$ 

```

Figure 5.12. Global conflict graph creation

5.6 Exclusiveness Information on the Behavioral Level

The exclusiveness information was only created for the level of LISA operations. In addition, information about the exclusive execution of operations also exists on the level of single assignments, arithmetic or logic operations, etc., represented in CFGs. These CFGs contain explicit information about exclusiveness by conditional statements, such as *IF* and *ELSE* blocks or *SWITCH* statements handling multiple cases. An example of the different relations between blocks is given in figure 5.13. An edge between two vertices is either an explicit compatibility or explicit conflict. If no edge exists between two vertices, it is a weak conflict that may become a compatibility by evaluating implicit exclusiveness information.

Implicit exclusiveness information is given by the data-dependency of conditions in conditional statements. Conditions are usually given by boolean functions of one or more variables. The main task for determining the exclusiveness between two conditional blocks is to compare their conditions. In general, the comparison of boolean functions is \mathcal{NP} complete [102]. Sophisticated algorithms have been developed to raise the number of manageable boolean inputs and complexity of boolean functions that are comparable with reasonable effort [103].

In the scope of this book, however, the addresses conditions are top level conditions which check the single bit control signals. Therefore, the condition comparison based on iterations through a condition truth table

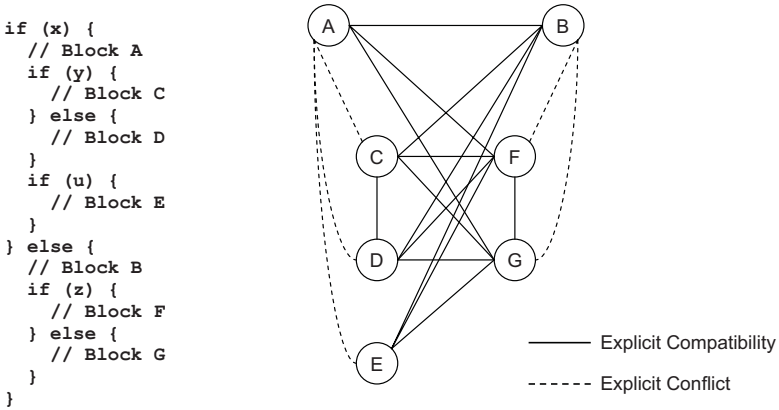


Figure 5.13. Example of explicit relations between conditional blocks

is sufficient as long as a maximum number $n_{inp,max}$ of boolean inputs is defined in order to guarantee an upper limit for comparison runtime. The truth table comparison is able to handle complex models with thousands of conditional blocks with $n_{inp,max} = 16$ boolean condition inputs within few minutes.

Overall, the analysis of CFGs for an evaluation of explicit and implicit exclusiveness is not specific to the LISA language. Therefore, this topic is not further discussed here. Interested readers might refer to [104].

Chapter 6

INTERMEDIATE REPRESENTATION

The previous chapters explain the urgent need for an automatic ASIP implementation from high level ADL models. As discussed in section 4.2.1, an intermediate representation is required which can be used to perform the optimizations, transformations and to generate the hardware description on RTL, using HDLs [105]. Furthermore, two main requirements to the intermediate representation are derived in section 4.2.1:

1. The intermediate representation must describe hardware on an abstraction level above RTL.
2. The intermediate representation must contain the semantics of the hardware description.

The *Unified Description Layer (UDL)* represents a hardware model for implementation while additional semantic information, derived from the ADL model, is retained.

6.1 Unified Description Layer

HDLs, such as VHDL, Verilog and RTL-SystemC¹ basically use the same elements to describe hardware. These are *processes* (Verilog: always-blocks, RTL-SystemC: methods) and *signals* (Verilog: wire, reg). They are used to explain the UDL in the following sections.

¹A subset of SystemC is synthesisable by some Synopsys' Design Compiler versions. Proposing SystemC as hardware description language is often confusing, as SystemC is well established in the domain of system-level design. Therefore, in the scope of this book, the term RTL-SystemC is used to refer to the synthesisable subset of SystemC.

6.1.1 Basic Elements of the UDL

In a hardware description on RTL, processes are used to model combinatorial or sequential logic in a procedural description style. Beyond the similarity to programming languages processes also describe the concurrent and reactive nature of hardware. In general, multiple processes are used to model one architectural building block.

Data, in its most general meaning, is transferred with a granularity of bits summarized by signals between processes. Signals are read and written within processes. Shown in figure 6.1, several processes A_1 , A_2 and A_3 are connected by signals S_1, \dots, S_{11} .

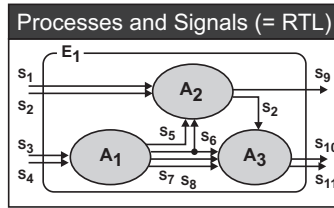


Figure 6.1. Hardware description on RTL, based on entities, processes and signals

In addition to processes and signals, *entities* are commonly used to modularize an RTL hardware model, which enables maintainability as well as reusability. These entities do not necessarily represent boundaries within the architecture, but only refer to the model organization. However, entities indirectly influence the architectural efficiency since optimizations performed by gate-level synthesis tools are usually limited by entity boundaries (cf. chapter 4). In figure 6.1, the hardware description consisting of processes and signals is covered by entity E_1 .

In the following, the first demand for a raised abstraction level is addressed. Both key elements of a hardware description, namely signals and processes, are considered. On the one hand, several signals between two processes are grouped to a *path*, which corresponds to an *abstraction of the interconnect*. On the other hand, several processes within one entity are summarized to a *unit*, which represents a *functional abstraction*. Semantic information, derived from the ADL model, is utilized to group signals and processes respectively. The UDL also provides entities to modularize the hardware model, similar to the RTL.

An example for the functional abstraction is a unit which represents the complete decode logic. An example for the abstraction of the

interconnect is the write access to a register file. In this case, the path contains the address signal, the control signal and the data signal.

Figure 6.2 illustrates the abstraction of several signals and processes. For example, path P_1 groups the signals S_1 and S_2 and path P_5 consists of signals S_5 and S_6 . Also, the processes A_2 and A_3 are represented by unit U_2 and unit U_1 represents the process A_1 .

The abstract elements of the UDL, path and unit, are *containers* to the underlying elements of an RTL hardware description. Thus, once the complete implementation is instantiated, an abstract view to the architecture is still possible. As depicted in figure 6.2, the same UDL model can be viewed on the highest level of abstraction, made up of entities, units and paths and also on the lowest level, the RTL, made up of entities, processes and signals. Certainly, a mixed abstraction with regard to both function and interconnect is also possible.

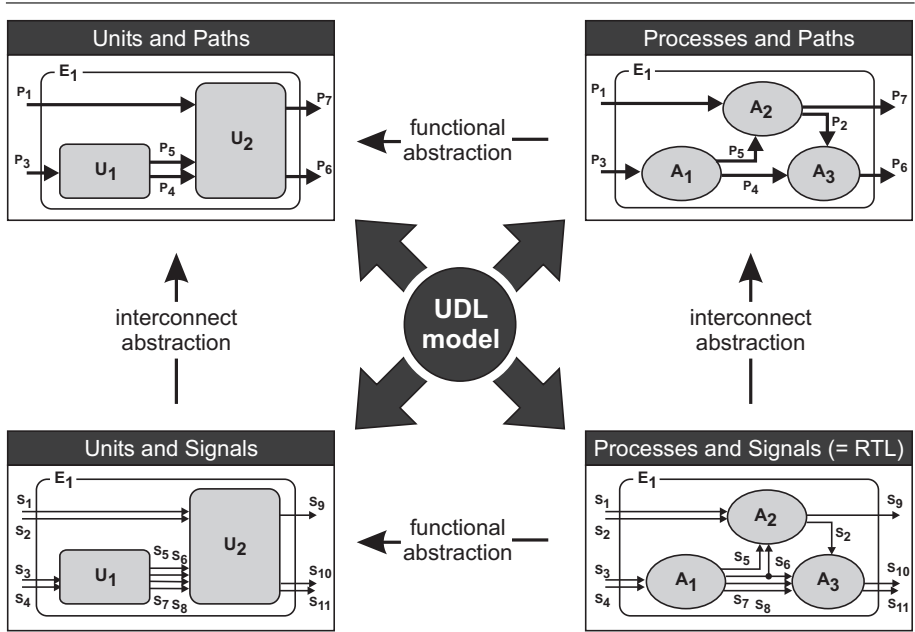


Figure 6.2. Different views to the same architecture model

The derivation above only explains how the UDL represents the ASIP on an abstraction level higher than RTL. In the following it is described how the UDL retains the semantic information derived from an ADL model. Both signals and processes are grouped to paths and units with

regard to their semantics. The UDL elements naturally correspond to the semantics of the underlying hardware description and are associated with explicit *types*, which represent a well defined semantic category.

The types of the UDL elements are listed in table 6.1. For example, the `pattern_matcher` unit recognizes particular instruction bit patterns and sets the `coding_path` accordingly. The information provided by the coding path as well as run-time conditions are evaluated by the `decoder` unit to set the control signals represented by the `activation_path`. These control signals are, for example, utilized by the `data_path` units. The types of the UDL are not fixed and can be extended in future work as needed.

Table 6.1. Exhaustive list of entities, units and paths

List of UDL Elements		
Entities	Units	Paths
testbench	data_path	resource
architecture	reset	mem
registers	pipe_reg	mem_r
memories	pipe_reg_coding	mem_r_w
pipeline	pipe_reg_activation	mem_w
pipe_stage	resource	clock
pipe_register	pin	reset
funct.-block	decoder	activation
	pattern_matcher	coding
	multiplexer	ft_pipe_reg
	pipe_controller	pipe_reg_ctrl
	pipe_register	pipe_reg_ctrl_flush
	ctrl_stall_flush	pipe_reg_ctrl_stall
	clock	

The types of the UDL elements, as listed in table 6.1, do not represent predefined implementations but repositories of semantic information. This information is retrieved by the frontend and assigned to the UDL elements. Especially, the exclusiveness information, as described in chapter 5.4, is directly linked to the elements units and paths.

6.1.2 UDL Model Hierarchy

The elements *path* and *unit* are embedded in a hierarchical model. An exemplary data structure is presented in figure 6.3.

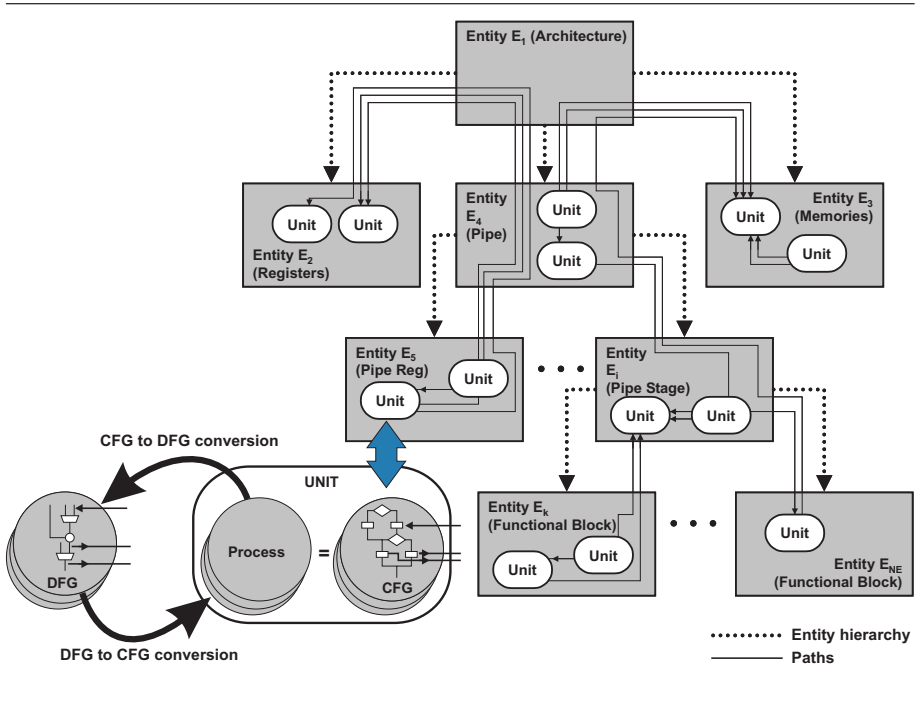


Figure 6.3. Example UDL of an ASIP

The model hierarchy is represented by a directed tree \mathcal{T} of entities. An edge corresponds to a single hierarchical relation between two entities, as indicated by dotted arrows in figure 6.3 (for example, entity E_1 contains the entities E_2 , E_3 and E_4). The functional model is represented by a graph \mathcal{G} , where the vertices are the units. The edges of this graph are UDL paths, as indicated by the solid lines in figure 6.3. The relation between \mathcal{T} and \mathcal{G} is well defined since a unit is uniquely assigned to a particular entity. Paths are bound to the entity hierarchy, since they represent the interconnect on RTL. Processes are represented by Control Flow Graphs (CFGs) or Data Flow Graphs (DFGs) depending on the transformation or optimization to perform.

6.1.3 UDL Instantiation and Refinement

The UDL covers different abstraction levels of the target architecture. As shown in figure 6.4 the highest level is made up of entities, units and paths. The lowest abstraction level consists of entities, processes and signals. The complete synthesis flow from the first instantiation to the RTL hardware representation comprises the following ordered steps:

1. **Structuring:** Instantiation of entities
2. **Functional Mapping:** Instantiation of units
3. **Mapping of the Interconnect:** Instantiation of paths
4. **Functional Refinement:** Conversion of units to processes
5. **Refinement of the Interconnect:** Conversion of paths to signals
6. **HDL Generation:** Generation of an RTL hardware model

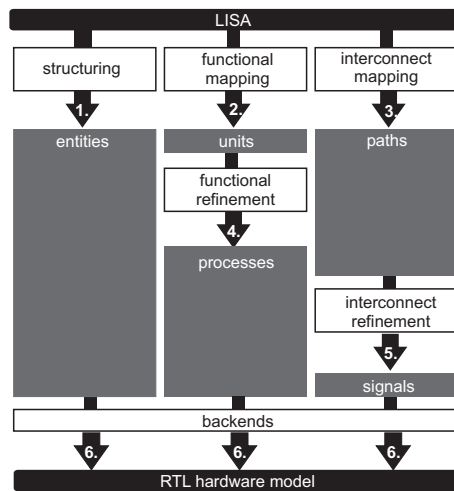


Figure 6.4. Synthesis steps from LISA to RTL

For instantiation and refinement of the model, information about the underlying hardware is extracted from the ADL model. Information directly available from the ADL model is called *explicit* information (cf. chapter 4). However, missing information must be compensated by assumptions about the ASIP implementation. These assumptions concern both the functional model and the interconnect on arbitrary levels of abstraction. The presented UDL can be perfectly used to include these assumptions at any step in the synthesis process. Three examples about

the usage of explicit information and assumptions about the implementation are given in the following.

The LISA frontend assumes an initial model of the hierarchy because this structural information is not existent in a LISA model. The frontend instantiates the entities `registers`, `memories` and `pipe` within the entity `architecture` (see figure 6.3). Optimizations and transformations may change this hierarchy in subsequent synthesis steps.

Every process needs to be represented by a CFG for architecture implementation. If the information about the implementation is omitted in the LISA model, a CFG template must be tailored to the provided information. For example, LISA specifies the instruction-set including the binary encoding but does not specify the corresponding decoder implementation. Here, CFG templates must be tailored to the specified instruction-set.

LISA provides the architecture's behavior on the basis of the C-programming language (the behavior is specified within so called behavior sections S_C , cf. chapter 5.3.3). These behavior sections are utilized to implement processes and to derive the CFG representation without assumptions.

6.1.4 UDL Visualization

The elements of the UDL can be used to provide a structural view on the ASIP. A screenshot of the synthesis tool is given in figure 6.5. The highest level of abstraction, which consists of entities, units and paths, is used for visualization. This level provides the most suitable view for design space exploration as unnecessary hardware details are omitted. The architecture can be explored on different levels of the model hierarchy while displaying the selected entity as well as all embedded entities. The type of paths might be used to visualize only a subset of all paths. For example, the resource access, pipeline control, decoder control or clock tree can be separately displayed. Paths can be selected and traced through the whole architecture.

6.2 Optimization Framework

The transformations and optimizations can be performed on all representations provided by the UDL. In general, the most suitable representation depends on the optimization to perform. In all cases, the optimizations benefit from the decreased model complexity, which is achieved by both the higher abstraction level and the semantic information. The available optimizations are discussed in chapter 7. Several of the optimizations and in particular resource sharing require a DFG,

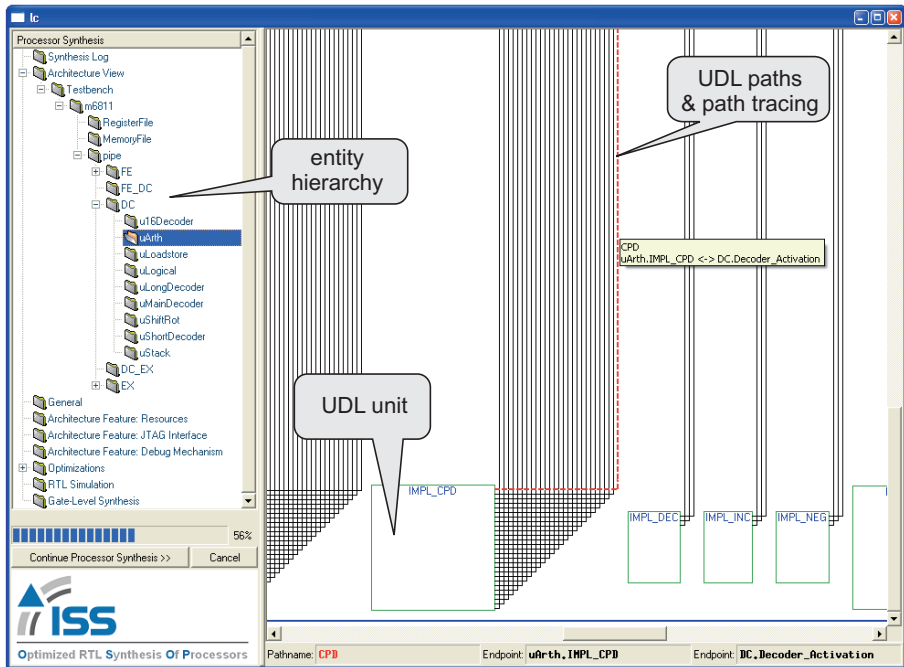


Figure 6.5. Visualizing the UDL in the synthesis GUI

which can be derived from a CFG. The corresponding conversions are described in appendix B.

6.3 VHDL, Verilog and SystemC Backend

The currently existing backends generate a hardware description in either VHDL, Verilog or RTL-SystemC [106]. Beyond syntactical differences, the major difference between these languages is the different handling of type conversions. In Verilog and RTL-SystemC logic description is loosely typed and the required type conversions are automatically performed by design tools. For example, in RTL-SystemC the type conversions of the ANSI-C standard is applied [107], whereas, in VHDL the data types are explicitly imposed on expressions. Although proper type conversion is an essential part in backend development, its implementation is of minor interest in the scope of this book and therefore omitted.

The LISA to RTL synthesis tool and in particular the backends are developed according to sophisticated software design techniques [108] [109][110]. Design patterns [111] are used to implement the software architecture. The UDL and backends are implemented according to the *visitor-pattern*, which equips the UDL with a generic backend interface. Arbitrary language backends can be hooked to this interface to iterate over the data-structure, collect required information and write out the hardware description.

The generated hardware description is usable for simulation as well as synthesis. The hardware behavior is compliant with the LISA simulator behavior. Thus, a bit accurate and cycle accurate co-simulation between the LISA model and the generated RTL hardware model is possible. Also the verification of the gate-level model, based on a co-simulation with the LISA model, is enabled as well. In addition to the RTL hardware description, the scripts for RTL simulation and gate-level synthesis are generated automatically for VHDL, Verilog and RTL-SystemC.

Chapter 7

OPTIMIZATIONS BASED ON EXPLICIT ARCHITECTURAL INFORMATION

In chapter 4 the urgent need for optimizations during the automatic ASIP implementation based on ADLs is discussed. These optimizations are motivated by the requirements for an architectural efficiency and implementation flexibility close to those of handwritten implementations (cf. section 4.1.1). To satisfy these two requirements two types of optimizations are required, the *constraint-independent optimizations* and *constraint-dependent optimizations*.

- **Constraint-independent** optimizations improve one or more physical characteristics without compromising any other characteristic.
- **Constraint-dependent** optimizations improve one or more physical characteristics at the cost of another characteristic.

Figure 7.1 gives two abstract optimization examples on the basis of the variables Area (A), Timing (T) and Energy (E) as well as the assumption $ATE = \text{const}$. The theoretically existing Pareto points are represented by surfaces in figure 7.1. *Constraint-independent* optimizations improve the architectural efficiency, which corresponds to shifting the surface of Pareto points. On the contrary, *constraint-dependent* optimizations trade off the physical characteristics and select a particular design point at the surface. Therefore, these optimizations (only) affect the implementation flexibility while the architectural efficiency remains unchanged.

In general, the optimizations benefit from the explicit architectural information specified in the LISA model. However, each optimization requires an appropriate abstraction level and suitable data-structures to retain the architectural information. As described in chapter 6, the

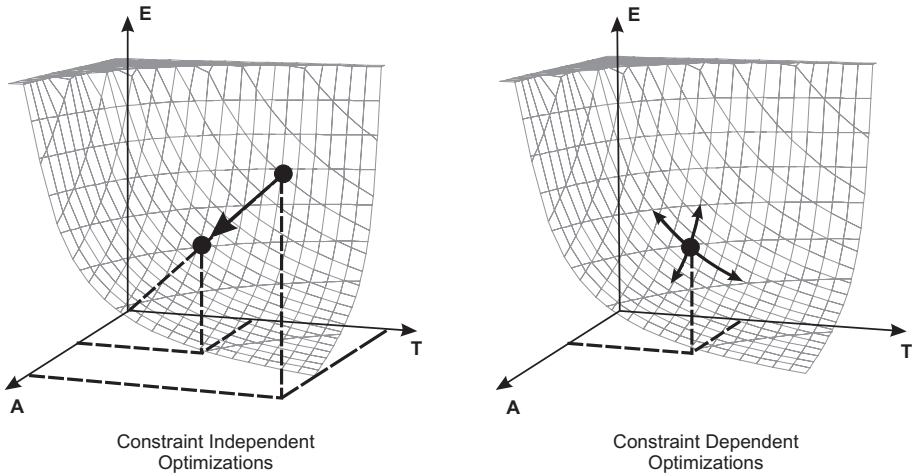


Figure 7.1. Constraint-independent and constraint-dependent optimizations

UDL provides several abstraction levels to perform optimizations. The optimizations presented in this chapter considerably gain from the UDL paths and their semantics, which are hooked into DFGs and CFGs [112].

The LISA behavior description (cf. section 5.3.3) is originally represented by a CFG and the HDL backends require a CFG to generate a hardware description, too. Since most optimizations are applied to DFGs, the conversions between both representations are explained in appendix B.

7.1 Basic DFG based Optimizations

Usually there is potential for removing redundancy after the initial creation of the DFG. The redundancy (e.g. unnecessary multiplexers, duplicate operators, constant inputs, etc.) is basically caused by the declarative description style and an significant abstraction or even complete neglect of the interconnect (cf. chapter 4). This redundancy complicates resource sharing optimizations unnecessarily. Therefore, constraint-independent optimizations have to be performed in order to obtain a DFG which is as simplified as possible. For this simplification the optimizations *constant propagation*, *multiplexer simplification*, *structural simplification* and *merging identical nodes* are applied.

In table 7.1, the effect of these basic optimizations on the ICORE architecture [91] (cf. appendix A) is given. The achieved gain is quite low,

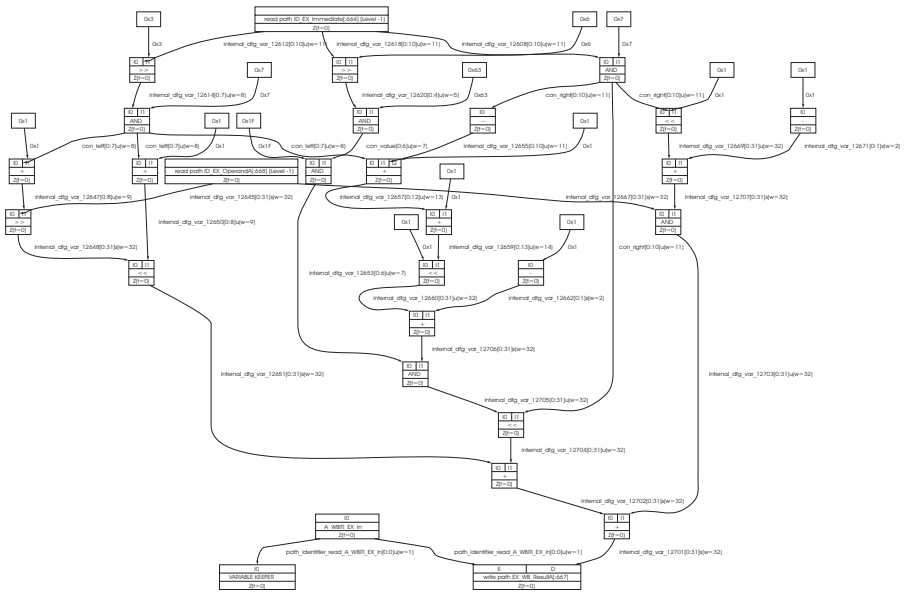
but nevertheless notable, if the optimizations are applied to the DFG derived from a *single* LISA operation. Obviously, the basic optimizations become more effective if they are applied to DFGs representing several LISA operations grouped by a functional unit. The effects of the optimizations for several functional units are mentioned in table 7.1, too. For these units, the basic optimizations achieved an important gain. The negligible improvement for the unit `Bitmanip` results from the fact that only few LISA operations are grouped by this unit.

The largest portion of the gain is achieved by the optimization *merging identical nodes*, described in section 7.1.4. This is a strong indication of redundancy caused by the LISA behavioral description and motivation to perform resource sharing in a later step.

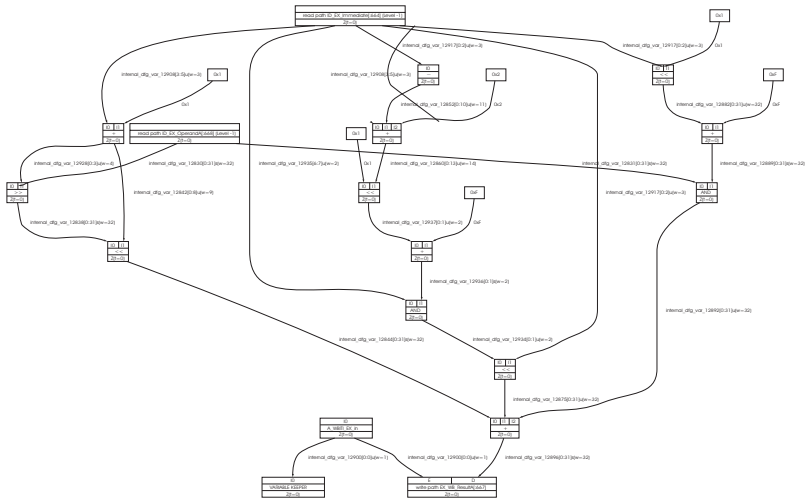
Table 7.1. Effect of basic optimizations on graph vertex count for the ICORE model

LISA Operation		Number of vertices w/o and with basic optimizations	
<code>pc_calc</code>	Program counter calculation	119	90 (75.6 %)
<code>CMP</code>	Compare	33	27 (81.8 %)
<code>Cordic01</code>	Special cordic instruction	39	37 (94.9 %)
<code>Abs</code>	Absolute value calculation	40	18 (45.0 %)
Functional Unit			
<code>ALU</code>	Arithmetic and logic unit	100	56 (56.0 %)
<code>DAG</code>	Address generation unit	161	109 (67.7 %)
<code>Shifter</code>	Shifter unit	92	67 (72.8 %)
<code>ZOLP</code>	Zero overhead loop	161	64 (39.8 %)
<code>Bitmanip</code>	Bit-manipulation unit	34	32 (94.1 %)

In figure 7.2 an example of the effects of the basic optimizations is given that depicts the unoptimized (a) and the optimized (b) DFGs for the `WBITI` instruction (write constant in bit field of register) of the ICORE architecture. The basic optimizations are described in the following sections.



(a) unoptimized DFG



(a) optimized DFG

Figure 7.2. Unoptimized and optimized DFG for ICORE's WBITI instruction [91]

7.1.1 Constant Propagation and Constant Folding

Constants are often used in CFG expressions, e.g. for constant shifting or comparisons. The *constant propagation* can be performed if all inputs $C_{in,P}$ for an operator P are constant. In this case the output $C_{out,P}$ can be replaced by the resulting constant. If only a single input is constant, some binary operators can be replaced by one of its inputs or even a constant. Especially boolean operators allow this simplification called *constant folding*. Table 7.2 lists the implemented constant folding optimizations.

Table 7.2. Simplifications by constant folding

Operator	Constant Input	Simplification
a AND b	a = 1	b
a AND b	a = 0	0
a OR b	a = 1	1
a OR b	a = 0	b
a XOR b	a = 1	\bar{b}
a XOR b	a = 0	b
a SHL b	b	a CONCAT $0_{width=b}$
a SHR b	b	a[$width_a - 1$ downto b]

7.1.2 Multiplexer Simplification

The implementation of multiplexers has a significant influence on the overall architectural efficiency. An important optimization is the simplification of the multiplexer implementation if the input data is at least partially constant.

This optimization targets multiplexers with a single control signal cc_{mux} and two input signals of boolean type $C_{in,mux}$. Due to the constant input the truth table of the multiplexer can be simplified, initially given by the set CC_v of associations $(v_{cc}, C_{in,mux})$ between condition values v_{cc} and inputs $C_{in,mux}$. The most common simplification rules are given in table 7.3. These simplifications can be applied, for example, if a constant single bit is assigned to a control signal within several nested conditional blocks.

7.1.3 Structural Simplifications

Structural simplifications are mainly intended to combine trees of binary commutative and associative operators into a single n-ary operator,

Table 7.3. Simplification for multiplexers with boolean control and data inputs

Assignment Patterns	Simplification
$CC_v = \{(v_{cc} = 0, 0), (v_{cc} = 1, 1)\}$	$C_{out,mux} = cc_{mux}$
$CC_v = \{(v_{cc} = 1, 0), (v_{cc} = 0, 1)\}$	$C_{out,mux} = \overline{cc_{mux}}$
$CC_v = \{(v_{cc} = 0, 0), (v_{cc} = 1, C_{in,mux})\}$	$C_{out,mux} = cc_{mux} \wedge C_{in,mux}$
$CC_v = \{(v_{cc} = 0, 1), (v_{cc} = 1, C_{in,mux})\}$	$C_{out,mux} = \overline{cc_{mux}} \vee C_{in,mux}$
$CC_v = \{(v_{cc} = 0, C_{in,mux}), (v_{cc} = 1, 0)\}$	$C_{out,mux} = \overline{cc_{mux}} \wedge C_{in,mux}$
$CC_v = \{(v_{cc} = 0, C_{in,mux}), (v_{cc} = 1, 1)\}$	$C_{out,mux} = cc_{mux} \vee C_{in,mux}$

e.g. successive additions, multiplications or identical boolean operators. No immediate advantage is provided by this simplification. Nevertheless, this transformation simplifies the identification of common inputs if two n-ary commutative and associative operators are candidates for resource sharing. This finally leads to a reduced instantiation of multiplexers.

Chains of multiplexers can be merged if they have one common input. This case occurs, for example, if the same value is assigned to a variable in several nested *IF* statements. However, this optimization is only possible if several conditions are met:

- All multiplexers in the chain must have one common input.
- Each multiplexer in the chain has only one output connection, namely the one to the next multiplexer in the chain.
- Each multiplexer has exactly two multiplexed inputs. If a default input is already present, it must be the common input. This condition is usually true for *IF/ELSE* statements.

An example of a hypothetical program counter calculation is given in figure 7.3. After the initial DFG creation, a chain of two multiplexers exists. Both multiplexers can be combined into a single multiplexer. The control input for the merged multiplexer is the concatenation of all control signals in the chain (here: cc_1 and cc_2). Therefore, the condition value associated with the special input (**branch**) is the concatenation of all condition values along the chain. The common input is used as default multiplexer input.

7.1.4 Merging Identical Nodes and Path Sharing

In general, the instructions' behaviors are independently described in several LISA operations to enable an efficient design space exploration. However, this concept introduces redundancy, in particular for

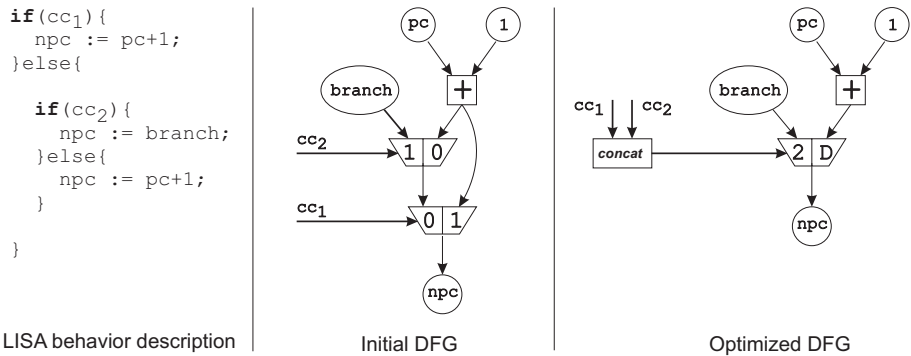


Figure 7.3. Multiplexer chain simplification

basic operations commonly used. This is of special importance for LISA resources accessed by multiple LISA operations. Here, it is possible to reduce the interconnect, which is caused by the use of individual paths for each single resource access (cf. example in section 4.2). This optimization is referred to as *path sharing*.

The redundancy can be removed easily with the help of a DFG by comparing all pairs of nodes for common operands and same functionality. For some classes of operations the sign of the operand may be ignored. This sign tolerance is allowed for additions and bitwise boolean operators, because no special sign handling is required. For comparators, shifters and multipliers, however, the sign of the operands is relevant for the correct computation. Therefore, operators of these classes with common but differently signed operands must not be merged. Only if the operators are mutually exclusive, it is possible to insert a conditional sign extension and to merge these nodes, too.

For behavioral descriptions it is common to use complementary operators, e.g. comparisons for equality or disparity, additions and subtractions, etc. In order to reduce the optimization complexity, the DFG creation process instantiates only one class for each pair of complementary operators and inserts an additional complement operator where necessary. Subtractions are converted into additions by using the binary complement of the subtrahend and using a carry-in of one. By this unification of operators, the potential for merging identical nodes and resource sharing is improved.

7.2 Resource Sharing

Resource sharing is an optimization that primarily addresses a reduction of the silicon area. However, the required multiplexer insertion potentially increases the minimum clock speed. Therefore, resource sharing is a constraint-dependent optimization. The effects on the timing must be considered by an appropriate cost model, as discussed in section 7.2.1. Based on the cost model, optimization constraints can limit the range of the resource sharing and thus restrict the effect on the timing.

Furthermore, the goals of a high architectural and implementation flexibility raise two questions:

- How can two resources be shared efficiently (cf. 7.2.2)?
- Which resources should be shared (cf. 7.2.3)?

Both questions will be answered with the help of DFGs and compatibility or conflict graphs containing exclusiveness information (cf. section 5.4ff).

7.2.1 Cost Estimation

Resource sharing algorithms have to estimate costs in order to ensure an improvement of the architectural efficiency. These costs result from additional multiplexers which increase the timing delay and chip area and thus lower the gain of sharing. The increase of the timing delay can also be caused by changed data flows. Cost estimation is particularly important for incremental sharing algorithms that have to select the most promising pairs of sharing candidates. The DFG has the potential of providing the necessary information of area and timing on a high abstraction level.

The synthesis of an RTL hardware description from an ADL is completely different from gate-level synthesis. Gate-level synthesis selects a particular implementation of complex arithmetic operations such as additions or multiplications, applies bit level optimizations and maps the logic and storage elements onto cells of a specific technology library. These technology libraries provide a highly diversified set of cells with manifold complex logic, different driver strengths and power consumption. Each cell provides detailed information on its timing behavior, the cell area, capacitance and power consumption. This information is only available because each library cell represents a specific physical implementation for a single chip production process.

When generating DFGs from the LISA behavior description, no information on the physical implementation is available at all, as both the used technology library and the bit level representation are unknown.

Not even the implementations of arithmetic operations are known and optimizations on bit level are not predictable.

Therefore, abstract instead of physical models for timing and area are required for the proposed optimizations. These abstract models can take into account heuristics of logic implementations for arithmetic operations and general aspects relevant for physical implementations.

Abstract timing approximation

In this work, a first and basic model for the delay estimation is used to avoid unacceptable timing by improper sharing. The values for the delay are normalized to a virtual gate with a delay of one unit. All logical operations such as AND, OR, XOR and NOT are approximated by a delay of one unit.

All other operations used in the DFG are modelled as multiples of basic gates. For adders with the bit width W , the assumed model is a carry ripple implementation with a carry chain of $W - 1$ basic gates with a delay of 1 unit plus a full adder delay of 3 units. This model is also applied to all comparison operations except for equality. The comparison for equality can be implemented as a balanced binary tree of AND gates that merges results of bit comparisons. This results in a delay proportional to $\log_2(W)$.

For multiplications, booth multipliers are an efficient implementation that is commonly used by gate-level synthesis. Booth multipliers do a recoding of the multiplicands in order to reduce the number of partial products from W to $W/2$. This step is approximated with 3 delay units. After this recoding, the partial products are generated and fed into a carry save tree adder with a height of $\log_2(W/2)$ and a full adder delay of 3 units on each tree level. This results in an additional delay of $3 \cdot \log_2(W/2)$ for the tree adder and $2 + W$ for the final carry ripple adder.

Multiplexers for the 1-of- n selection are approximated by a tree of 1-of-2 elementary multiplexers with the height equal to the number of control bits W_{control} . For each 1-of-2 multiplexer a delay of two units is approximated.

An overview on the timing approximations currently used is given in table 7.4. These approximations use a basic approach to introduce timing aspects into the high level resource sharing algorithms. However, they are very coarse estimations and thus require more refinement and deeper investigation. Yet the current estimation is already usable as the comparisons for several units of the ICORE architecture demonstrate in table 7.5. For the three functional units *ALU*, *Shifter* and *Branch Detection* the ratios of the approximated relative delays versus the synthesis

Table 7.4. Models for a basic abstract timing approximation

Function	Model	Timing Approximation
AND, OR, XOR, NOT	Single Gate	1
Addition, Subtraction	Carry Ripple Adder	$W + 2$
Comparison (<i>Not</i>) <i>Equal</i>	Tree Comparator	$ld(W) + 1$
Comparisons <i>Greater</i> ,...	Carry Ripple Adder	$W - 1 + 3$
Multiplication	Booth Multiplier, Carry Save Tree Adder	$5 + 3 \cdot \lceil ld(W/2) \rceil + W$
Multiplexer	Tree of 1-of-2 multiplexers	$2 \cdot W_{\text{control}}$

Table 7.5. Timing approximation vs. synthesis results for the ICORE architecture

Functional Unit	Approximation	Synthesis Results	Ratio
ALU	78	1.97 ns	39.59 1/ns
Shifter	106	2.56 ns	41.40 1/ns
Branch Detection	87	2.21 ns	39.37 1/ns
MinMax	42	2.14 ns	19.62 1/ns
Mult	33	2.51 ns	13.14 1/ns

results are around $40 \frac{1}{ns}$ with a low variance, showing that the current approximation already leads to acceptable results. The delay of the functional units *MinMax* and *Mult* however is weighted too low in the approximation, stressing the need for further research on the aspect of timing approximation.

Abstract area approximation

When sharing two resources, multiplexer insertions are usually necessary. These multiplexers occupy additional chip area A_{mux} . If the area saving A_{sav} , is smaller than A_{mux} the sharing will result in a negative gain. Thus, the sharing of these two operations will not give any advantage.

In many cases, a priori assumptions can be made on the relations between the saved area by sharing two operators of a specific class and the area of necessary multiplexers. This is particularly true for single bitwise operators such as *AND*, *OR* and *XOR*. No multiplexer implementation will be smaller than the area saved by removing a single bitwise operator. On the other hand, adder and multiplier implementations consume more area than the multiplexers inserted for the sharing.

For these reasons, a priori assumptions are used to decide if an operator class is worth sharing or not. Therefore, the area approximation

is currently not considered. Further research in this field is necessary in order to use area approximations for improved sharing algorithms.

7.2.2 Resource Sharing Methods

The implemented resource sharing optimizations targets arithmetic and logical operators, while storage elements are explicitly defined in LISA by the designer and therefore not considered.

All operators can be classified by the commutativity and associativity of their inputs. For operators with commutative and associative inputs, any permutation of the input order is possible without changing the operator's result. For non-commutative operators each input has its own semantics which prohibits the permutation of the operator inputs.

In terms of sharing efficiency, commutative operators provide much more flexibility in the search for common inputs which reduce the number of multiplexers added by the sharing. In order to exploit this flexibility, different algorithms have to be used for commutative and non-commutative operators. A differentiation between unary, binary and n-ary operators is not necessary, because binary operators always fit in one of the commutativity classes of n-ary operators. For unary operators commutativity does not matter.

Non-commutative operators

Examples of non-commutative operators are subtraction, division, selection, etc. Two exclusive executed operators P_x and P_y can be shared if they implement the same function \mathcal{F} and have the same number of inputs. When P_x and P_y are merged, there is no other possibility than sharing their inputs \mathcal{C}_{in} paired in the order given by their parameter position in function \mathcal{F} : $\{(C_{in,P_x,i}, C_{in,P_y,i}) \mid 1 \leq i \leq |\mathcal{C}_{in,P_x}|\}$.

The example given in figure 7.4 shows three subtractions that have some common inputs. For efficient sharing it is important not to introduce multiplexers for common inputs, such as input b.

In the second sharing step the importance of the knowledge about existing multiplexers is stressed. An unfavorable possibility is the use of chained multiplexers. Recognizing equal inputs to already established multiplexers leads to a balanced solution, as shown in figure 7.4.

Commutative operators

Examples of operators that are both commutative and associative are addition, multiplication and the basic logical operators *AND*, *OR* and *XOR*. The sharing of two commutative operators P_x and P_y is possible if both operators implement the same function \mathcal{F} and have the same

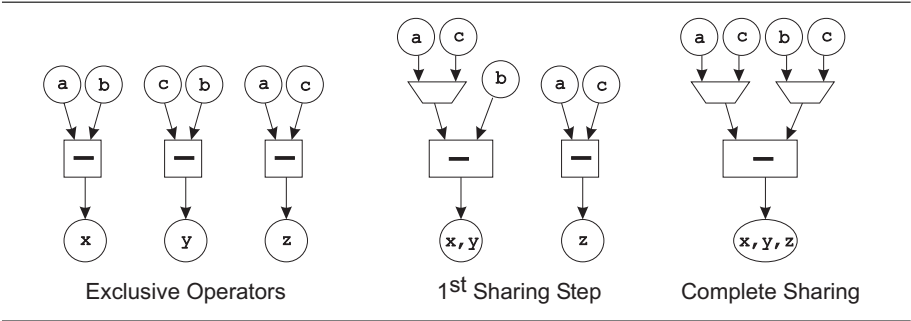


Figure 7.4. Sharing of non-commutative operators

number of inputs. The numbers of inputs does not need to be identical, because missing inputs can be completed by the identity element n with $\mathcal{F}(n, x_1, x_2, \dots) \equiv \mathcal{F}(x_1, x_2, \dots)$.

A common example of this completion with identity elements is sharing of addition and subtraction. The subtraction is represented as addition with the binary complement plus a *carry-in* set to one (figure 7.5). This operation is shareable with a binary addition by inserting the identity element $n_{\text{add}} = 0$ as carry-in input.

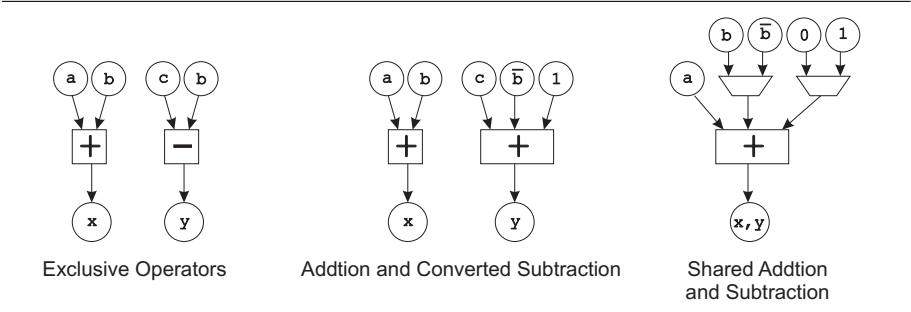


Figure 7.5. Sharing with insertion of identity elements

For two commutative operators, any pairing of their inputs is possible, resulting in $N!$ different sharing possibilities for two operators with N inputs each. An optimal pairing has to pair as many identical inputs as possible. For this purpose an input $C_{\text{in}, P_x, i}$ of operator P_x has to be compared with any input $C_{\text{in}, P_y, j}$ of operator P_y . If an identical input

$C_{in,P_y,j}$ is found for $C_{in,P_x,i}$, they are paired. If more than one input of P_y matches, it doesn't matter which one is chosen. Thus the decision can be made already for the first match. Therefore, only $\sum_{i=1}^N N - i = \sum_{k=N-1}^0 k = N(N - 1)/2$ comparisons are necessary. However, it is important that in a first iteration no decision is made for an input of P_x without a counterpart in P_y , because it could bind an input of P_y that has a counterpart in P_x . The inputs of P_x skipped in the first iteration may be paired with any input of P_y in a second iteration.

In figure 7.6 an example of sharing two adders with one common and two different inputs is given. The common input a is found and the multiplexer insertion is avoided.

When sharing two commutative operators with already existing sharing multiplexers, it is important to include the inputs of these multiplexers in the decision for the grouping of the original unshared inputs into the shared inputs. The sharing that results in the DFG given in figure 7.6 detected that input b is already present in the first multiplexer of figure 7.6 and that input e is already present in the second multiplexer. Thus only the input f is left and therefore shared with input a .

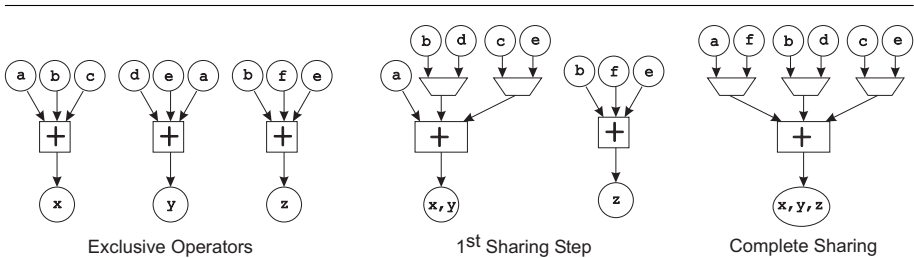


Figure 7.6. Sharing of commutative operators

The sharing of more than two operators can run into problems, if

- two or more inputs are used for at least two different operators
- and two or more of these inputs are used concurrently for at least one operator.

In figure 7.6 this is true for the inputs b and e . If the first sharing step produces a different pairing, as depicted in figure 7.7, the greedy search for common inputs in the set of existing sharing multiplexers will result in a suboptimal multiplexer assignment. In the example given, the

inputs b and e are already grouped by the same multiplexer $M1$, but for the remaining operator P both inputs are used concurrently. Therefore, b may be shared by $M1$, but e must be assigned to another multiplexer $M3$ with three inputs now, instead of only two (figure 7.7).

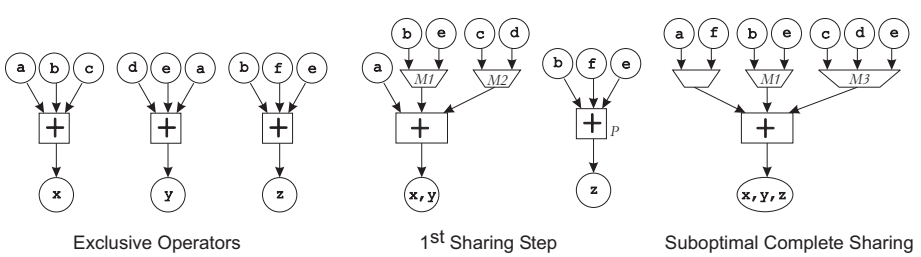


Figure 7.7. Suboptimal sharing of commutative operators

This problem can be described by graph theory. A graph G has to be created that describes which inputs are used for an operator $P \in \mathcal{P}$ within the set \mathcal{P} of all operators to be shared. The vertices of graph G are the different inputs used for the operators, an edge (I_i, I_j) indicates that the inputs I_i and I_j are used concurrently for at least one operator. Thus graph G is the input conflict graph for the sharing of the set \mathcal{P} of operators. Figure 7.8 shows this graph for the example given in figure 7.6.

$$\begin{aligned}
 G &:= \langle V, E \rangle \\
 V &:= \{I \mid I \text{ is input of any operator } P \in \mathcal{P}\} \\
 E &:= \{(I_i, I_j) \mid I_i, I_j \in \mathcal{C}_{in,P}, P \in \mathcal{P}\}
 \end{aligned} \tag{7.1}$$

In order to achieve the optimal grouping for an n -ary operator, the input conflict graph G has to be partitioned into n partitions. All inputs covered by a partition will be shared by one multiplexer. If a partition contains conflicts, one or more inputs involved in the conflicts of this partition have to be duplicated and used in other multiplexers, too. This was the case in figure 7.7, the partitioned conflict graph for this situation is given in figure 7.8. Therefore, the number of conflicts within the partitions has to be minimal for an optimal sharing. Figure 7.8 is the partitioned conflict graph for the example from figure 7.6.

The optimal partitioning can be found by the solution of the maximum induced n -partite subgraph problem for the conflict graph G . This problem is already NP-complete for the bipartite problem ($N = 2$) [113].

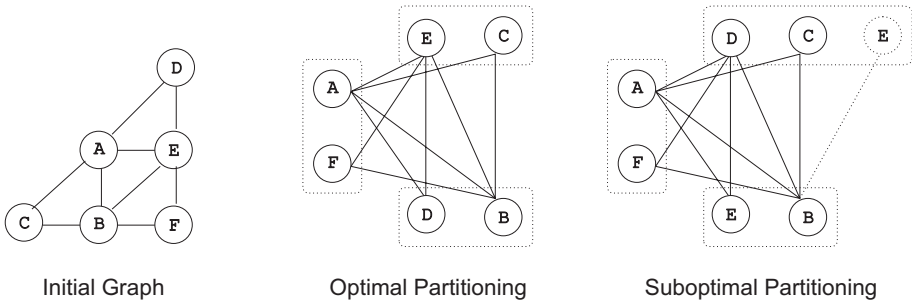


Figure 7.8. Input conflict graphs

Pearson et al. propose a linear time heuristic for the bipartite problem based on the breadth first search in the graph but without measuring the heuristic's quality [114]. Nevertheless, this heuristic is applied by Brisk et al. in a study on data-path based resource sharing [115].

7.2.3 Resource Sharing Decision

In addition to the necessity to efficiently share two hardware resources, it is also important to select those resources with the most promising gain. For this purpose the knowledge about common inputs, as discussed above, will be helpful and extended by further criteria.

There are two contrary approaches for resource sharing. The first approach is based on graph coloring. It focusses on the absolute minimization of the number of resources without considering the data flow structure. The missing consideration of the data flow structure might cause disadvantageous multiplexer insertions and elongated critical paths. The second approach applies heuristics for DFG matching in order to consider the data flow structure. It might not achieve the minimization of utilized resources but much better results in terms of timing and multiplexer reduction.

Clique covering and conflict graph coloring

The problem of finding the minimum number of required hardware resources can be solved with the help of compatibility graphs and conflict graphs. For compatibility graphs all nodes of a fully connected subgraph called clique can be mapped on a single resource, for conflict graphs all nodes of a completely unconnected subgraph can be shared. For minimal resource usage the minimal number of such subgraphs has to be found.

In graph theory this problem is known as the *clique covering* problem for compatibility graphs and as the *graph coloring problem* for conflict graphs. The graph coloring problem stems from the goal of using the minimum number of colors for coloring the different countries on a map without using the same color for neighboring countries. Translated to conflict graphs, the vertices represent the different countries while edges connect pairs of neighboring countries.

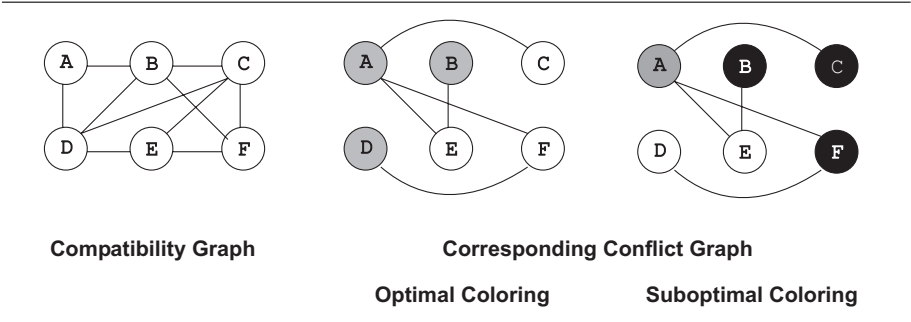


Figure 7.9. Clique covering and graph coloring

In figure 7.9 an example of an optimal and a suboptimal graph coloring is depicted. The coloring scheme with two colors is the optimal solution for the bipartite conflict graph. Also a suboptimal solution is shown which is caused by the decision to color the three compatible vertices *B*, *E* and *F* with the same color.

The problem of finding a minimum coloring of a graph is NP-hard. The corresponding decision problem whether there is a coloring with at maximum *n* colors is NP-complete [116]. Thus optimal algorithms have a runtime which increases exponentially with the number of graph vertices. Therefore, heuristics have to be applied for large graphs.

In this work two algorithms for graph coloring are implemented and used depending on the number of graph vertices. For small graphs a backtracking technique proposed by Brown is used to find the optimal coloring [116][117]. With this algorithm the optimal coloring of graphs up to about 40 vertices is feasible.

For more than 40 vertices, the exponential runtime increase of Brown’s algorithm is unacceptable. In this case the heuristic called “Recursive Largest First” proposed by Leighton [118] is selected, which is known to be one of the most efficient. It has a runtime of $\mathcal{O}(|V| \cdot |E|)$ for a graph $G = \langle V, E \rangle$ [116].

A common problem of the use of graph coloring for resource sharing is the limited focus on single resources ignoring problems and advantages given by data dependencies. The most critical problem is the creation of *false loops*. There are several approaches for the prevention and elimination of false loops [119][120][121]. In general additional edges have to be inserted into the conflict graph. This insertion however might remove good sharing possibilities [122].

For this reason, in this work resource sharing by graph coloring is only utilized for operators that do not allow loop creation or require the aggressive resource minimization provided by an optimal coloring.

Heuristics for data flow graph matching

The problem of graph coloring is that its focus is limited to single resources ignoring data dependencies. Constructive heuristics avoid this problem by sharing pairs of operators of the DFG iteratively. With this approach it is possible to concentrate on the area reduction and timing issue instead of minimizing only the number of resources without considering the logic overhead caused by multiplexers.

The algorithm described by Raje and Bergamaschi proposes several criteria that cover structural similarities in the DFG up to a vertex distance of two [122]. Brisk et al. try to increase the coverage by comparing complete data-paths, however, their number may increase exponentially in complex data flows [115], while Bhattacharya et al. introduce timing constraints into their resource sharing algorithm [123].

In this work concepts from Raje, Bergamaschi and Bhattacharya are considered to accomplish a resource sharing algorithm that covers both structural similarities and timing issues. For this purpose several criteria are introduced to compare pairs of shareable resources.

Input Similarity: As shown in section 7.2.2 multiplexers can be avoided if two operators have common inputs. The number of common inputs is defined as *input similarity*. Therefore the sharing of the pair with the highest input similarity is desired. Furthermore configurable constraints can be introduced in order to define a minimum input similarity necessary for sharing.

Output Similarity: If the outputs of two operators serves as input for the same multiplexer, the sharing of both operators will remove the subsequent multiplexer and increase the gain of sharing (figure 7.10). Therefore, the number of common subsequent operators is defined as *output similarity*. If two pairs have different output similarities, the pair with the higher output similarity is preferred.

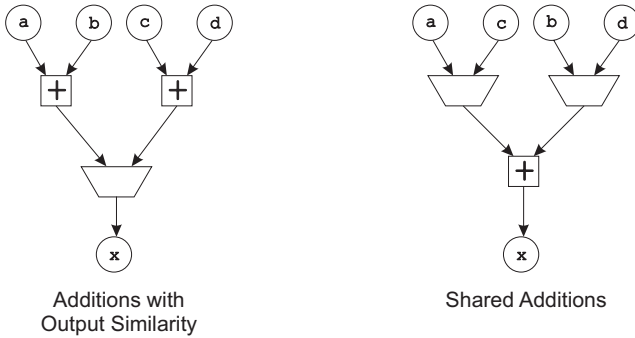


Figure 7.10. Sharing nodes with output similarity

Timing Difference: When sharing two operators, the critical path might be elongated significantly (figure 7.11). Therefore, the timing approximation given by the DFG has to be used to decide whether the increase of the timing delay is acceptable. A configurable constraint is introduced defining the *maximum* relative timing difference allowed for two operators to be shared. In the example given in figure 7.11 the operators OP_5 and OP_B are sharing candidates with a maximum relative timing difference of $\max\{|5 - 2|/2, |2 - 5|/5\} = \max\{150\%, 60\%\} = 150\%$. Recommended values for this constraint, with regard to the current implementation of the optimizations, are 30% and below.

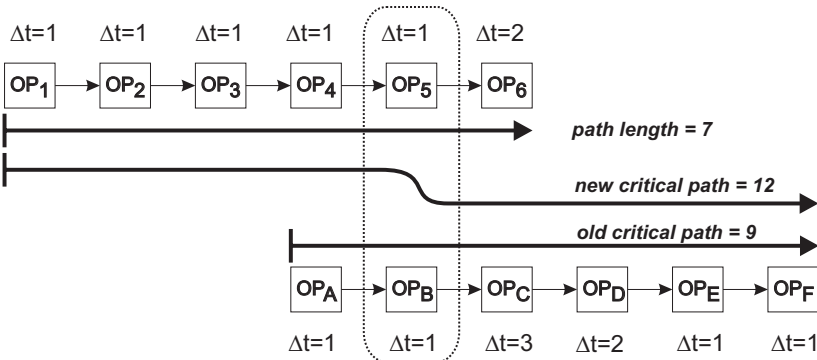


Figure 7.11. Worse timing by sharing operators with widely differing timing

The implemented resource sharing algorithm groups all operators P with the same functionality \mathcal{F} to sets $\mathcal{P}_{\mathcal{F}}$, e.g. a set of adders, a set of multipliers, etc. The resource sharing optimization proceeds with one of the sets $\mathcal{P}_{\mathcal{F}}$ and selects a pair of shareable resources $S = (P_i, P_j)$ with $P_i, P_j \in \mathcal{P}_{\mathcal{F}}$. A pair $S = (P_i, P_j)$ is shareable if:

- Both operators are mutually exclusive. This information is derived from the exclusiveness information on the LISA operation level (cf. section 5.5) and behavioral level (cf. section 5.6).
- Both operators have no data dependency. If there exists a data flow from the output of P_i to an input of P_j or vice versa, the pair S is not shareable, because a false loop would be created otherwise.
- The maximum relative timing difference for P_i and P_j is less than the configured constraint.
- \mathcal{F} is one of the operators listed in table 7.6. Boolean operators, such as *AND*, *OR*, etc., are not shared, since they always occupy less area than any multiplexer inserted due to the resource sharing. Read accesses to one- or multidimensional LISA resources with constant addresses are not shared either, because no combinatorial logic is instantiated for constant addresses.

Table 7.6. List of sharable operations and their implementation

operation		implemented by
addition subtraction	+ −	+ − and two's complement
multiplications	signed * signed unsigned * unsigned signed * unsigned unsigned * unsigned	signed multiplication and sign extension
comparisons	> < ≤ ≥ = ≠	> > and switched operands <i>not</i> > <i>not</i> > and switched operands = <i>not</i> =
shifts	shl shr	shl shr
division modulo	div mod	div mod

If multiple pairs $S_1 = (P_i, P_j)$ and $S_2 = (P_k, P_l)$ can be shared, a rating is required to perform the most promising resource sharing step. The rating is applied by performing prioritized comparisons:

- 1 If the input similarity for S_1 is higher than the input similarity for S_2 , S_1 is the better rated pair and vice versa. If the input similarities are equal, proceed with further comparisons.
- 2 If the output similarity for S_1 is higher than the output similarity for S_2 , S_1 is the better rated pair and vice versa. If the input similarities are equal, proceed with further comparisons.
- 3 If the relative timing difference for S_1 is lower than the relative timing difference for S_2 , S_1 is the better rated pair and vice versa.

This rating constitutes the heuristic applied in this work. It includes the structural properties in an environment with a radius of one DFG vertex into the sharing decision. A task of future work will be the extension by further criteria allowing the structural analysis for a larger radius as well as the estimation of the sharing gain in terms of effective area reduction.

7.3 Dependency Minimization

Often nested conditions, which are mutually exclusive, are derived from the ADL model. In the default implementation the conditions are mapped onto IF/ELSE statements. During the gate-level synthesis the nested IF/ELSE statements have different effects on signals written within the conditional statements.

For signals that are written in all cases an encoder can be used to efficiently steer the multiplexer, as shown in the example given by figure 7.12.

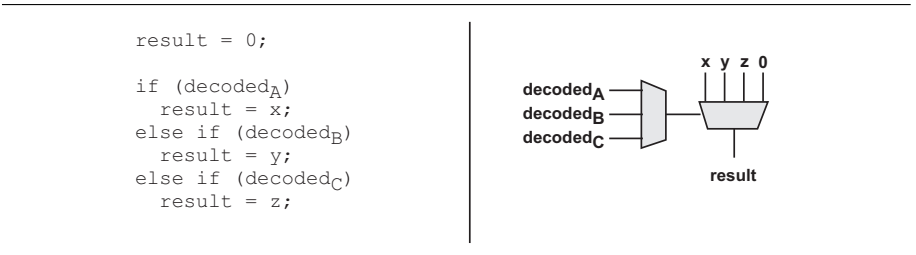


Figure 7.12. Default condition check implementation

If a signal is written in only one or few cases, all preceding conditions of the nested IF/ELSE statements are unnecessarily evaluated. In general, gate-level synthesis tools are not able to recognize the mutually exclusiveness of the control signals and thus do not resolve the nested structure. In the example given in figure 7.13, the signal `result` only depends on the control signal `decodedC`, but due to the nested conditions all preceding conditions are checked as well.

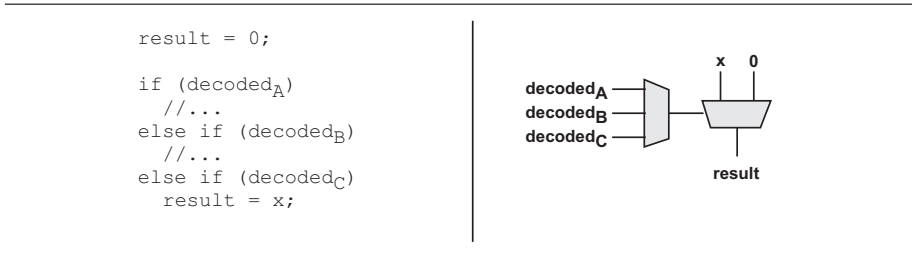


Figure 7.13. Unnecessary dependencies in default condition check implementation

The result of this suboptimal gate-level synthesis is an increased fanout for the control signals `decodedA` and `decodedB`. The signal delay for the multiplexer selection input is also increased due to unnecessary comparisons.

For mutually exclusive conditions, these drawbacks can be removed by using separated IF statements instead of nested conditions as depicted in figure 7.14. With this modification, the priority encoder only checks the conditions that are really necessary, for assignments of single constant bits they can be omitted completely.

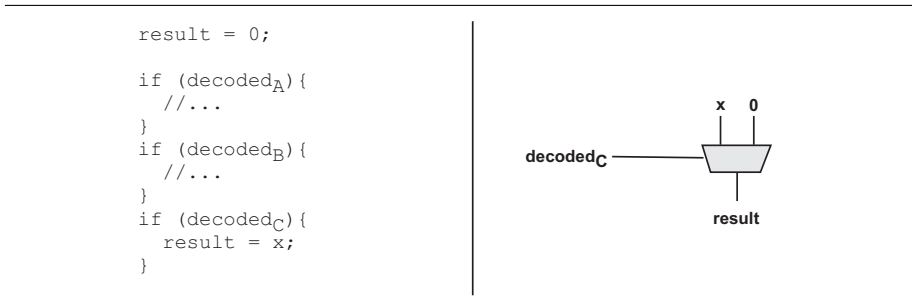


Figure 7.14. Improved condition check implementation

However, there are some cases in which this implementation causes worse results. First, signals that are both read and written in different conditional blocks create long chains of combinatorial logic that are not generated when using nested conditions. Second, the RTL code within separated conditional blocks is no longer explicitly specified to be mutually exclusive. Therefore, optimizations applied by the gate-level synthesis tool might be less successful.

Obviously, this is a constraint-dependent optimization, which must be carefully applied with regard to the targeted architecture and conflicting optimizations.

7.4 Decision Minimization

The most common and relevant paths used in the UDL are paths from the functional units to the resource units. These resource accesses can be either read or write accesses. Generally, these paths are mapped to three signals on RTL:

- address signal (for one- and multidimensional resources)
- data signal
- read/write enable signal

For write accesses, all three signals are driven by the functional unit. For read accesses, the data signal is driven by the resource unit. The implementation of the resource unit ensures, that operations only take place if the enable signal is set true.

In general all signals of a path are written within the same block, which might be a conditional block, as shown in figure 7.15. The register R is accessed by writing to the signals $R_{\text{write,addr}}$, $R_{\text{write,data}}$ and $R_{\text{write,enable}}$ within a conditional block. Thus, for all signals the corresponding condition is evaluated and the signals are only written if necessary. However, the resource implementation checks the enable signal, and therefore implicitly the original condition again. Therefore the preceding evaluation of the condition is redundant for the address signal and the data signal. Considering the knowledge about the resource unit implementation a more efficient implementation can be found.

The redundancy can be removed by separating the assignments of the path signals. This idea is depicted on the right of figure 7.15. Only the enable signal needs to be written within the conditional block, thus all other assignments can be moved outside the conditional block. This optimization results in a significantly lower fanout of the logic driving the control signal as well as a significantly improved timing delay for the written signals.

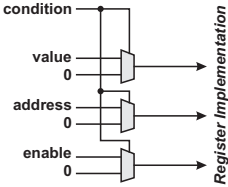
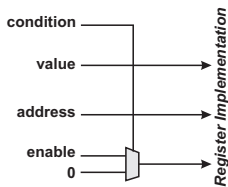
LISA	HDL (unoptimized)	HDL (optimized)
<pre>if(condition){ R[idx] = value; }</pre>	<pre>Rwrite,value = 0; Rwrite,address = 0; Rwrite,enable = 0; if(condition){ Rwrite,value = value; Rwrite,address = idx; Rwrite,enable = 1; }</pre>	<pre>Rwrite,value = value; Rwrite,address = address; Rwrite,enable = 0; if(condition){ Rwrite,enable = 1; }</pre>
		

Figure 7.15. Decision minimization

Moving assignments out of the associated conditional block is only possible if all input dependencies can be resolved. Since variables introduce such dependencies, an efficient decision minimization algorithm must also move variable assignments out of conditional statements. In this case the variables are renamed to avoid the interference between the data dependencies inside and outside the conditional block. If several statements are moved out of a conditional block, their order has to be preserved.

This optimization aims at a timing improvement and multiplexer reduction. By moving complete statements, which might include arithmetic operations, out of conditional blocks explicit exclusiveness information is lost. This prevents any further resource sharing optimizations. Therefore, this optimization can only be applied as very last optimization.

Decision minimization, which represents a constraint-independent optimization, results in significant improvement of timing and area as described in chapter 9, section 10.2 and appendix A. The gain depends on the timing constraints and further optimizations.

Chapter 8

CONFIGURABLE PROCESSOR FEATURES

Various auxiliary processor features, such as debug mechanism, power save modes or bus interfaces, are commonly incorporated in off-the-shelf embedded processors, but completely omitted in ADL based ASIP design. However, for the development of competitive ASIPs it is of great importance that ASIPs keep up with the features provided by off-the-shelf processors.

ADLs enable an efficient design space exploration by describing the core functionality of ASIPs in a declarative manner (cf. section 4). Since most processor features are not part of the core functionality, they are not covered by ADLs. Additionally, the declarative description style precludes an implicit description of processor features. Also a manual integration of processor features into an existing RTL hardware model is hardly possible as this potentially introduces inconsistencies with the generated software tools. Therefore, an automatic integration of these processor features during the synthesis of an RTL hardware model is proposed.

In order to satisfy the requirements of high design efficiency, architectural efficiency, performance and flexibility, the processor features, that are considered for an automatic integration, must be carefully selected. This topic is discussed in detail in the following section. Afterwards, the generation of debug mechanism and JTAG interface is presented.

8.1 Processor Features

Processor features differ from each other regarding their applicability in an automatic synthesis process from an ADL. The following questions must be answered when considering a processor feature for an automatic integration.

- **Does the processor feature influence the architectural efficiency or performance?**

A limited influence on the architectural efficiency and performance is required to avoid a costly design space exploration concerning the processor feature. Also, the influence of the processor feature must be predictable accurately.

- **How much is the processor feature interweaved with the architecture?**

Configurable processor features are based on templates, which are automatically integrated in the ASIP. Processor features that are strongly interweaved with the architecture core conflict with an unrestricted design space exploration. Therefore, only those processor features which are moderately interweaved with the architecture core are candidates for an automatic integration.

- **How large is the design space of the processor feature?**

The multiple realization and implementation alternatives of a processor feature must be configurable in order to fit smoothly into the overall ASIP design process.

According to these questions, a debug mechanism, for example, is a perfect candidate for the automatic integration, as discussed in the following.

Due to the increased complexity of SoCs in general, as well as its computational building blocks, a debug mechanism is required. Design errors often just become visible in a system context, which renders on site debug capabilities indispensably. A debug mechanism enables the designer, for example, to examine the current state of the processor, to stepwise execute the application software and to force the processor into a new state. The debug mechanism generation, including a JTAG interface, is exemplarily used to explain the automatic processor feature integration [124].

The basic functions of a debug mechanism require access to all storage elements, already. Reading the current state of the processor, for example, requires an interruption of the program execution, the suspension of the processor status and the access to the storage elements. Thus, the debug mechanism is interweaved with the processor core at least regarding the storage elements.

A JTAG interface commonly serves as the interface to the debug mechanism. The processor core and the JTAG interface operate with different clock speeds, which leads to separate *clock domains* in the architecture.

Therefore, a suitable handover mechanism also needs to be applied, to prevent the architecture from running into undefined states. The automatically generated JTAG interface, handover mechanism for clock domain boundaries and debug mechanism are presented in the following sections.

8.2 JTAG Interface and Debug Mechanism Generation

The debug mechanism as well as the JTAG interface are configurable within the boundaries given by the ASIP’s specification. The ADL model is read by the frontend and mapped to the basic elements of the UDL, entity, unit and path (cf. chapter 6). Based on this, the designer is able to configure the desired debug mechanism and JTAG interface in a GUI, as shown in figure 8.1.

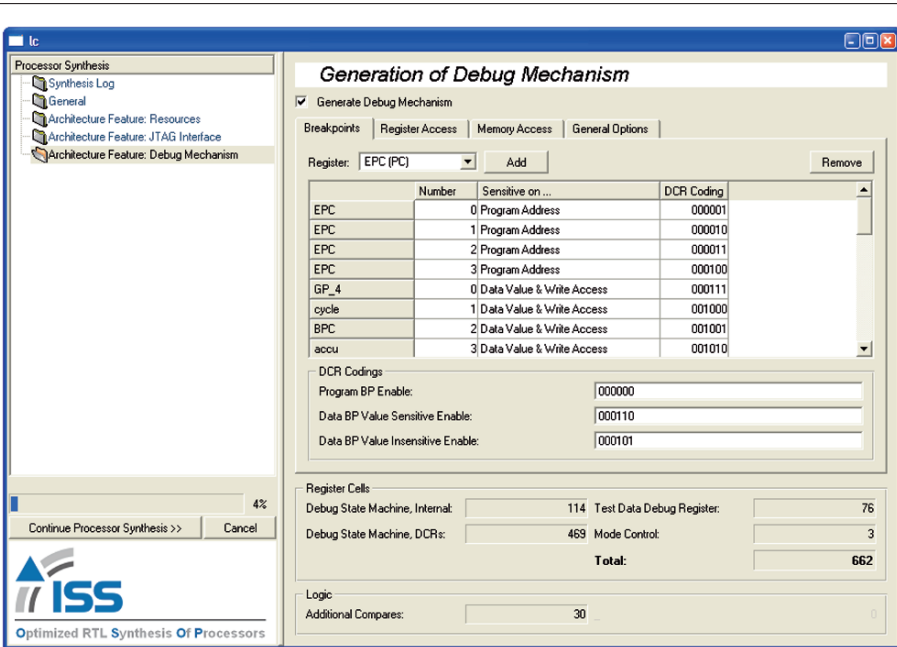


Figure 8.1. Debug configuration dialog

8.2.1 IEEE Standard Test Access Port (JTAG)

The IEEE standard 1149.1 “Standard Test Access Port and Boundary-Scan Architecture” was originally introduced in February 1990 by the Joint Test Action Group (JTAG). It has been modified several times up to its current version 1149.1-2001 [125]. The standard defines a test access port and a boundary-scan architecture for digital integrated circuits and for the digital portions of mixed analog/digital integrated circuits. The acronym JTAG has become a synonym for this standard over the years.

The boundary-scan

The boundary-scan architecture enables a test of the interconnections between integrated circuits on a board without physical test probes. Boundary-scan cells, which consist of multiplexers and latches, are added to the pins of the chip. A detailed description of a boundary-scan cell can be found in [125]. According to the IEEE standard, the integration of a boundary-scan is mandatory for a JTAG implementation. However, ASIPs are mostly integrated in SoCs and thus not directly connected to the pins of the chip. Nevertheless, the JTAG interface has also become famous for the communication with the debug mechanism. In this case, the required component is the Test Access Port (TAP), as discussed in the following section.

The test access port

The TAP is the key component of a JTAG interface and comprises a physical pin interface and the TAP controller, as shown in figure 8.2. The pins provided by the TAP are the following:

- Test Data Input (TDI) and Test Data Output (TDO)
The test data pins are used for a serial transfer of data. The instructions to control the debug mechanism, for example, are set through the TDI pin. The debug information, for example a register value, is received through the TDO pin.
- Test Clock input (TCK)
The test clock input receives the clock for the test logic defined by the JTAG standard. Therefore, the JTAG interface can be used independently from the architecture’s specific clock. Usually, the frequency of the test clock is much lower than the one of the system clock. Hence, timing violations are usually not introduced by the test logic.

- Test ReSeT input (TRST)

The test reset input is used for an asynchronous initialization of the TAP controller.
- Test Mode Select input (TMS)

The signal received at the test mode select input is decoded by the TAP controller to control test operations.

The basic principle is the utilization of an arbitrary number of *test data registers* to control, for example, a debug mechanism. The *boundary-scan register (boundary-scan chain)* is also one of these test data registers. The boundary-scan register and the bypass register, as depicted in figure 8.2, are required by the JTAG standard. All test data registers are shift registers connected in parallel between TDI and TDO. One of the test data registers is selected by a command sequence written to the instruction register. In addition to the TAP controller, further control logic is required, which is omitted in figure 8.2.

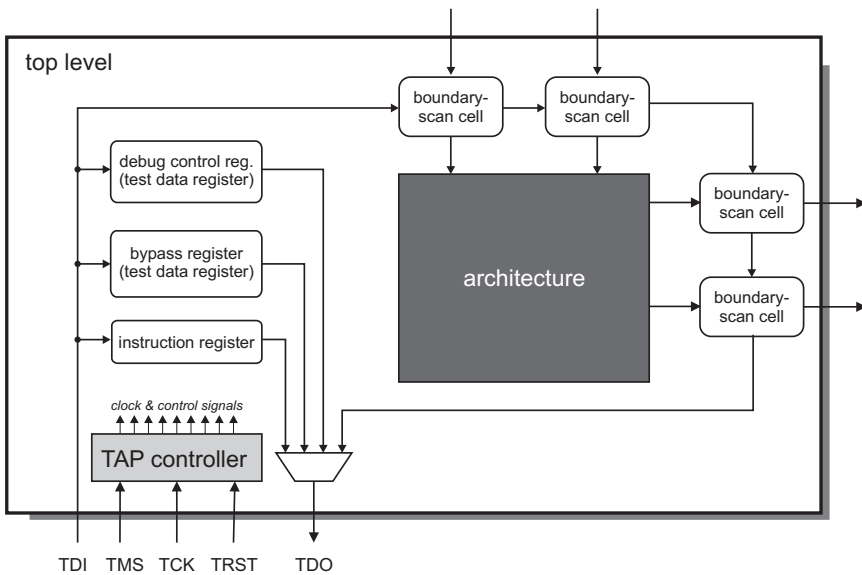


Figure 8.2. Structure of the TAP

The interface between JTAG interface and debug mechanism is basically given by a *debug control register*. This is of special importance since ASIPs are mostly one of many building blocks in a SoC. In general, this SoC only contains one single JTAG interface and TAP controller.

In this case, separate debug control registers, which are all connected to the same TAP controller, must be instantiated for each architecture with debug mechanism. Therefore, the generation of a TAP controller is optional and not required for the generation of the debug mechanism.

As mentioned above, the JTAG interface and the processor operate at different clock speeds. The consequences for hardware implementation are discussed in the following section.

8.2.2 Multiple Clock Domains

The JTAG interface and processor, including the debug mechanism, run at different clock speeds. Data transfers across clock domain boundaries potentially cause *metastable* states of the architecture. This can be prevented by instantiating appropriate circuits in the RTL hardware model.

Metastability

The correct operation of a flip-flop can only be guaranteed if the setup and hold time regarding the D input are not violated. As illustrated in figure 8.3, the level of input D must not change during setup time t_{su} and during hold time t_h .

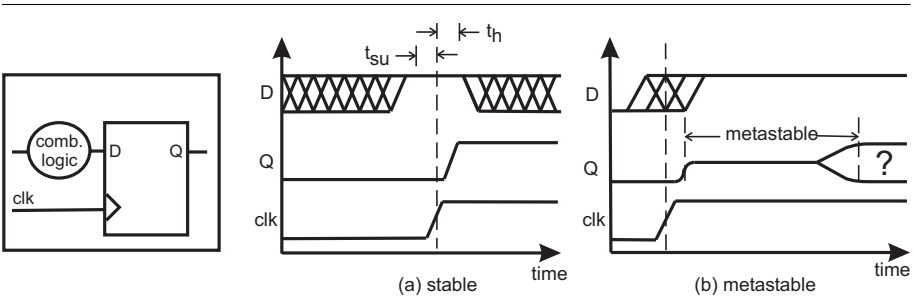


Figure 8.3. Setup and hold time for a D-flip-flop

A timing violation results in an undefined state of the flip-flop. As also illustrated in figure 8.3, the output Q of the flip-flop adopts a level between the low and high levels. This state is called *metastable*. After an unpredictable duration, the flip-flop reverts to one of the two possible states because of the noise of transistors and the interference penetrating from the exterior.

In [126] an equation is derived to calculate the Mean Time Between Failures (MTBF) resulting from metastable states in non-synchronized circuits with multiple clock domains. An example is given for a system clock frequency of 10 MHz, a mean frequency of the asynchronous input signal of 10 kHz and a setup time of the following circuit of 25 ns. In this case the MTBF is only about 54 minutes. Obviously, the resulting error rate is too high.

The MTBF can be drastically increased by latching the asynchronous input value by two flip-flops, which are connected to the clock of the targeted domain. The MTBF increases up to 2 million years [126], which is acceptable in most cases. However, this concept is not suited for transmitting a whole signal representing several bits over a clock domain boundary, as not only metastability must be avoided but also data consistency must be guaranteed. This means that all bits must be simultaneously transmitted to the targeted clock domain. Therefore, a more sophisticated synchronization circuit needs to be utilized.

Synchronization circuit

The synchronization circuit is derived from the *Flancter Circuit* by Rob Weinstein [127]. It utilizes a handshake mechanism based on a one bit flag to transfer data from one clock domain to the other. The flag is set to logical high whenever a new value has been written to the source register. A successful data transfer is acknowledged by setting the flag back to logical low. The status of the flag clearly indicates whether new data can be written or read. Figure 8.4 depicts the Flancter Circuit.

The Flancter Circuit is made up of two D-type flip-flops with clock enable (FF1 and FF2), an inverter and an exclusive-or (xor) gate. The flag is completely asynchronous to both clock domains. Therefore, the additional flip-flops (FF3 to FF6) are used to reduce the probability of metastable states. The asynchronous reset inputs to the flip-flops are unconnected for clarity.

The operation sequence of the Flancter Circuit is as follows. The flag is set to *high* from clock domain `clk1` by the signal `clk_en_set`. The flag is set to *low* (reset) from clock domain `clk2` by the signal `clk_en_reset`. The flag can be read from both domains.

Based on the synchronization circuit mentioned above, figure 8.5 depicts the data transmission across clock domain boundaries. The first step is to write the information to the data register (1). In the next clock cycle, the flag is set from the source clock domain (2). The flag can be read two clock cycles later from the destination clock domain (3) at the

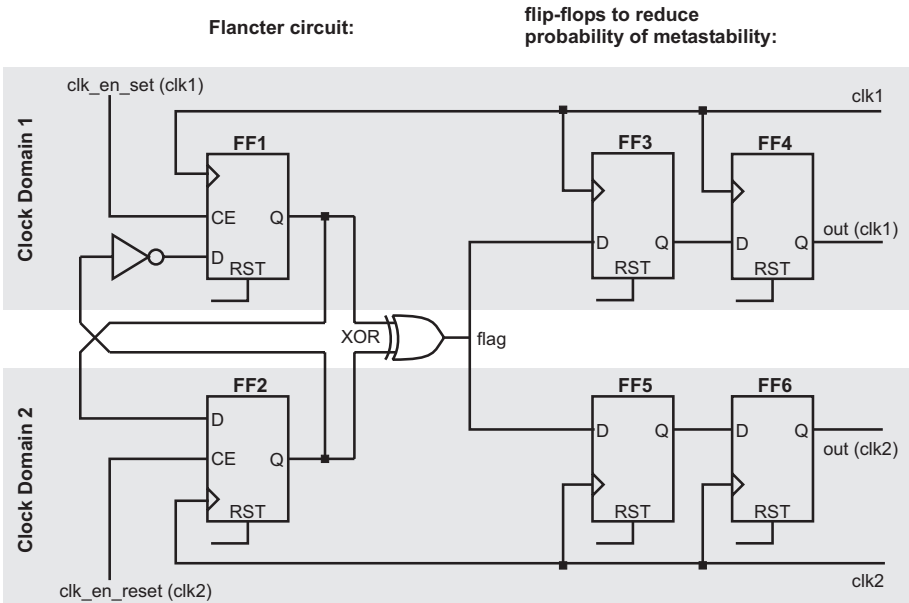


Figure 8.4. The Flancter Circuit

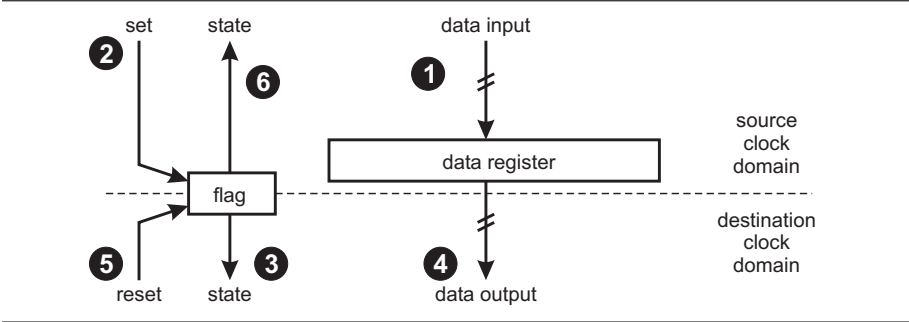


Figure 8.5. Data transmission across clock domain boundaries

earliest. In the fourth step the data register is read from the target clock domain (4) and the flag is reset from the destination clock domain (5). As soon as the reset of the flag is observed from the source clock domain (6), new values can be transmitted by this handshake mechanism.

8.2.3 Debug Mechanism

Today, almost every off-the-shelf processor contains a debug mechanism to enable software debugging even within the final hardware environment. A processor with debug interface is capable of switching to a special operation mode for debugging purpose, a so called *debug mode*. On the contrary, the usual operation mode is called *user mode*. The features of the generated debug mechanism are the following:

Entering debug mode and returning to user mode: There are several possibilities to switch to debug mode and back to user mode. For example, it can be either requested by (re-)setting a particular pin of the architecture, or the JTAG interface can be used to set a debug request instruction. The debug mode is entered automatically whenever a breakpoint is hit.

Single step execution in debug mode: A single step execution is required to observe state-transitions in the smallest possible temporal granularity. In this case, the processor automatically changes into debug mode after executing one step.

Access to storage elements during debug mode: The current values of the processor's storage elements must be read to determine the current state of the processor core. This comprises registers (in this context called core registers), pipeline registers, memories, etc. A write access to the storage elements is useful to force a particular processor state.

Hardware breakpoints: Whenever a hardware breakpoint is hit, the processor core switches to debug mode. Hardware breakpoints are distinguished between *program breakpoints* and *data breakpoints*. Program breakpoints (sometimes also called instruction breakpoints) specify a programm address and take effect *before* the corresponding instruction is executed. Data breakpoints refer to core registers and take effect when the corresponding core register is written. Additionally, these data breakpoints may be sensitive to predefined values.

Further information about debug mechanisms, the Nexus standard and debug mechanisms in industry can be found in appendix C.

8.2.4 Architecture Structure

The automatically generated debug interface uses a JTAG interface and three additional ports to control the debug mechanism [128]. As shown in figure 8.6, the three ports are *debug request* (DBG_REQUEST), *debug mode* (DBG_MODE) and *ready read*

(RDY_READ). The DBG_REQUEST pin is used to switch between user and debug mode. DBG_MODE signals the processor’s current mode. The RDY_READ pin is set to high when a value (e.g. register value) can be read from the debug control register.

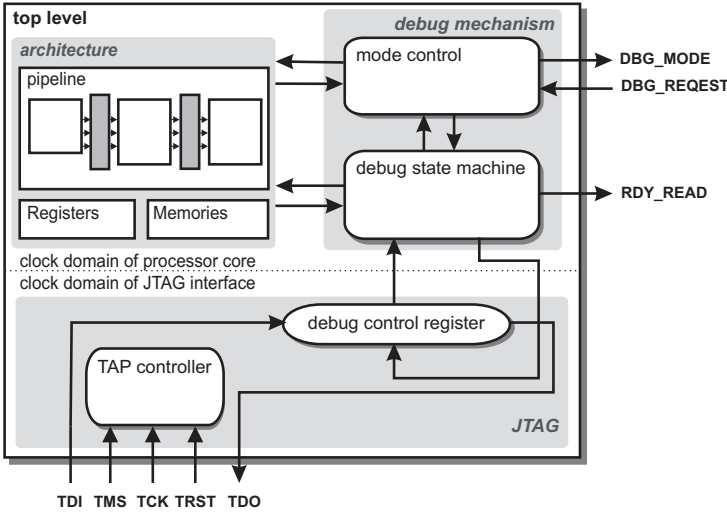


Figure 8.6. The synthesized RTL hardware model structure

The automatic generation of a debug mechanism demonstrates the usability of the proposed synthesis framework to automatically integrate processor features. The debug mechanism basically consists of two components, called *mode controller* and *debug state machine*. These are basically connected to the storage elements of the ASIP core. The debug mechanism can be controlled via the JTAG interface mentioned above. The following section briefly describes the generated components of the debug mechanism.

8.2.5 Debug State Machine and Mode Controller

Debug state machine

Debug instructions can be written into the debug control register via the JTAG interface. The debug state machine decodes these instructions and controls their execution. Furthermore, the debug state machine stores the configuration of the debug mechanism, for example data and program breakpoints. These breakpoints are configured by debug

instructions written to the JTAG interface. The current state of the debug mechanism is stored in so called Debug Configuration Registers (DCRs). The debug state machine is hardly application-specific and thus can be easily configured according to given design constraints.

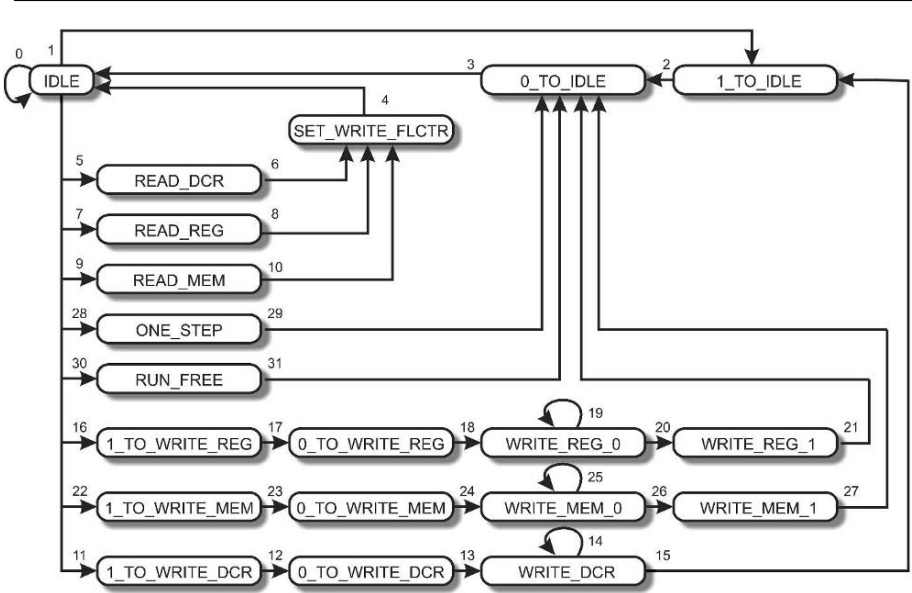


Figure 8.7. State transition diagram of the debug state machine

The state transition diagram of the debug state machine is depicted in figure 8.7. The default state of the debug state machine is the IDLE state, in which new debug commands are accepted. These instructions are decoded and executed according to the state transitions in figure 8.7. For example, read accesses to the DCRs (transition 5), to the registers (transition 7) or to the memories (transition 9) are initiated. In order to process write accesses to these elements, several intermediate states are passed to ensure proper synchronization with the JTAG interface via the hand-over mechanism described in section 8.2.2. If an unknown debug instruction is decoded, the state transition 1 is initiated to reset the Flancter Circuit and to accept new debug instructions. A detailed description of the debug state machine can be found in [128].

Mode controller

Several events, such as breakpoint hits, external debug mode requests and one step execution requests, might cause a switch between user and debug mode. The mode controller monitors these events and selects the appropriate processor mode. This functionality is not architecture-specific, however, its implementation is highly optimized to minimize the influence on the architecture’s timing characteristics. The interface and the state transition diagram of the mode controller are depicted in figure 8.8.

As shown on the left in figure 8.8, the most important output of the mode controller is the *register enable path*, which is utilized to suspend the execution of the application. This path is directly driven by a breakpoint detection path and the external debug request path (DBG_REQUEST). Due to this bypass of the debug state machine, the propagation delay is minimized and the timing characteristic of the architecture are hardly influenced. Certainly, the implementation of the one step execution control needs to be carefully considered in order to avoid an increased critical path.

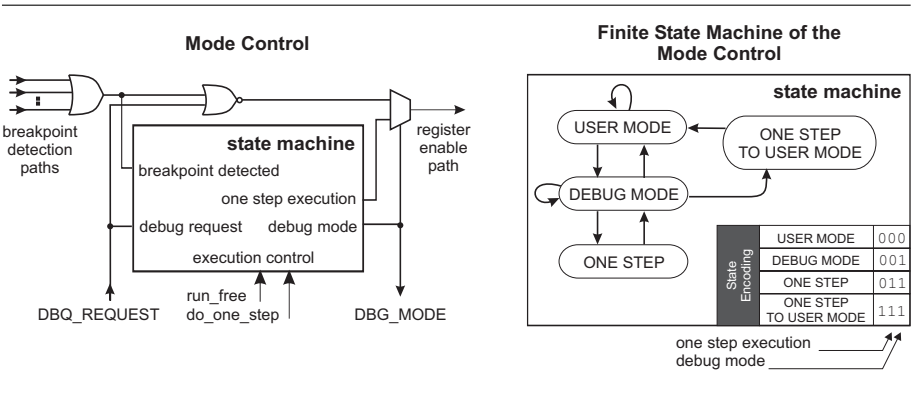


Figure 8.8. Mode controller of the debug mechanism

The four states are encoded by three bits to restrict the influence on the timing of the ASIP. On the right of figure 8.8, the state transition diagram and the state encoding is depicted. The particular bits of the state encoding can be directly used to drive the control-path, which renders additional control logic unnecessary. A detailed description of the mode controller is given in [128].

8.2.6 Changes to the Processor Core

The above-mentioned JTAG interface, the debug state machine and the mode controller are supplemental components and instantiated outside to the processor core. This section describes the necessary changes to the processor core, which mainly motivate the requirement of an *automatic* integration of the processor feature. Several elements are changed or added to support the JTAG interface and the debug mechanism generation as given in table 8.1, section 8.3. The extended implementation of core registers and memories are discussed in detail in the following.

Register generation with debug support

The register implementation is shown in figure 8.9. This register diagram describes the fundamental principle of the register access in user and debug mode. In the given example, n read and write paths access a register file, which comprises m elements. The white boxes represent the logic required for register access in user mode. The write paths are routed to the elements of the register file through a cross-connect. The synchronous register file implementation is connected with the read paths through a cross-connect.

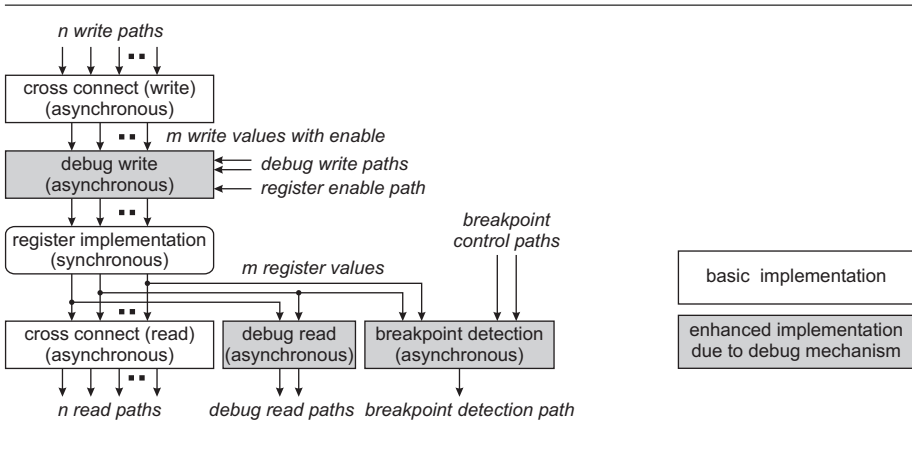


Figure 8.9. Register implementation

The gray boxes are instantiated to realize functions of the debug mechanism. They contain combinatorial logic for writing a register, reading a register and detecting breakpoints. When the critical path runs to a register with debug write access, it is only elongated by a single multiplexer (selecting 1-of-2) and thus hardly influenced by the debug mechanism.

However, the size definitely increases as shown in the case studies presented in chapter 10.2 and appendix A.

As it can be seen here, a debug mechanism does have significant influence on the architecture implementation, which prevents an efficient manual integration of the debug mechanism into the architecture once the architecture is generated on RTL.

Memory generation with debug support

The debug support for memories is comparable to the concepts applied to the implementation of registers. However, the basic difference is that the debug logic cannot be embedded in the memory implementation. Therefore, the debug functionality is constrained by the available number of memory ports. As shown in figure 8.10, the default read and write ports must be used for debug access. However, the functionality provided is not affected.

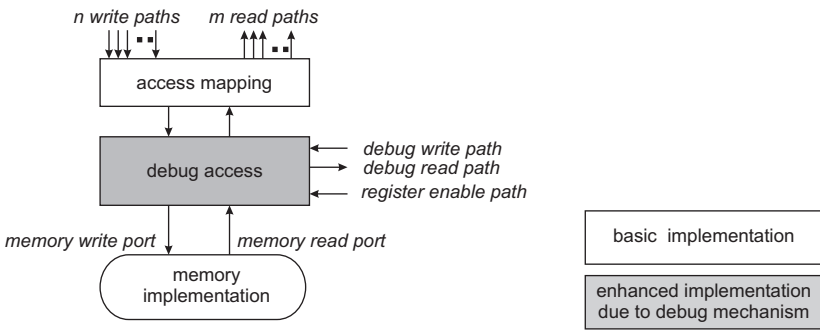


Figure 8.10. Memory implementation

The following section briefly describes the changes and enhancements to the UDL, which are required to support debug mechanism and JTAG interface generation.

8.3 Adaptability of Synthesis Framework

In general, the advantages of the UDL compared to the RTL are a higher abstraction level and additional semantic information. These properties of the UDL simplify the automatic integration of processor features as the computational complexity is significantly reduced. For example, the architecture structure needs to be transformed due to the generation of a debug mechanism and JTAG interface. The transformation complexity depends on the overall model complexity and thus

Table 8.1. UDL elements for debug mechanism and JTAG interface generation

enhanced UDL elements		added UDL elements	
Entities	Units	Entities	Units
-	register memory pipe_register pipe_reg_coding pipe_reg_activation	dbg_state_machine dbg_mode_control	dbg_testbench dbg_state_machine dbg_mode_control dbg_dw_tap dbg_testdata_reg dbg_mux_reg_read dbg_mux_mem_read dbg_tdo_generation

benefits from the UDL. This topic has been elaborated in detail in chapter 4 and chapter 6.

The synthesis framework must be extensible easily regarding to further processor features. Fundamental changes or even substitutions of existing elements are error-prone and must be avoided. Therefore, a reasonable choice of the UDL elements and their interfaces is essential to easily enhance the functionality.

About 20 UDL paths have been added, their development time is negligible due to the applied software design techniques. Table 8.1 lists the UDL units and entities, which have been enhanced or added due to the debug mechanism and JTAG interface generation. The register unit, for example, originally only covered a basic register implementation. It has been extended to cover debug access and breakpoint functionality as well. Obviously, none of the complex units, such as the decoder unit or the data-path unit, has been changed due to the support for JTAG interface and debug mechanism generation.

Chapter 9

CASE STUDY: AN ASIP FOR TURBO DECODING

Over several years Turbo decoding implementations have been exclusively realized using ASIC designs. However, even in this application domain which is characterized by very high data throughput requirements, the benefits of programmable solutions are wanted. Today's 3GPP standard limits the throughput to 2 MBit/s, while future systems are predicted to require above 10MBit/s. Scalable and energy efficient architectures are required to catch up with this challenge. GPPs or even domain-specific architectures do not satisfy the requirements given by the Turbo decoding principle (table 9.1), because their fixed structures of *memory* and *arithmetic units* are not suited to exploit the algorithm's inherent potential for architectural optimizations. Therefore, insisting on a programmable solution, ASIPs are becoming the architecture of first choice. In this chapter, an ASIP design for Turbo decoding is outlined, spotting several important design issues, as e.g. loop control, addressing schemes and data-path.

Detailed information about the algorithm's fundamentals can be found in [129]. A further discussion of implementation issues can be found in [130] and [131].

9.1 Turbo Decoding based on Programmable Solutions

Various solutions that are based on programmable and sometimes also reconfigurable architectures exist, each of them marking a particular point in the design space. Table 9.1 presents an overview of existing programmable solutions including special architectural features, the type of instruction-set, clock frequency and throughput. Obviously, the major

issue here is the achieved throughput, which is far below even today's telecommunication standards. For example, the 3GPP standard defines a maximum throughput of 2 MBit/s.

The solutions 1 to 15 in table 9.1 are based on either general purpose or domain-specific processors without an application-specific configuration. Considering a normalized frequency, the throughput rates are unacceptably low.

The solutions 16 to 20 in table 9.1 are based on (re)configurable architectures. Custom instructions are utilized to steer an optimized data-path. The work presented in [139] and [140], mentioned in line 20, elaborates the possibilities of data-path configuration. The configurable portion of the architecture is used to implement an instruction-set extension for efficient add-compare-select execution. Therefore, the bottleneck imposed by general purpose arithmetic units is overcome. The solution of adding a custom instruction is realized utilizing design environments from ARC [51] and Tensilica [52]. In both designs the embedded FPGA limits the clock frequency to 100 MHz, which leads to equal gate-level synthesis results and a throughput far below the 3GPP standard. In order to meet these constraints, a multiprocessor solution is discussed in [141]. Here, the design space exploration also needs to consider possible communication structures, which satisfy the algorithm's inherent data dependencies. However, the search for such a communication structure is often neglected [137].

Based on an Altera Excalibur EPXA1 platform, Blume, Gemmeke and Noll [136] investigated acceleration possibilities for Turbo Decoding. Here, different application hot spots are mapped on the FPGA, which is loosely coupled to an ARM922T. The utilization of an ACS-hardware-unit decreases the throughput compared to a pure software solution, as the processor/FPGA interface does not meet the data transfer requirements (line 15 and 16). The data transfer rate between processor and FPGA is reduced by mapping a recursion unit or a Max-LOGMAP unit on the FPGA in solutions 17 and 18. Therefore, the Turbo decoding throughput is increased up to 400 kbit/s.

However, utilizing configurable and thus optimized data-paths obviously increase the data throughput, while the implementations still fail to meet today's 3GPP standard. The difference in data-throughput between a tightly coupled and loosely coupled accelerator indicates the importance of data routing through the architecture. Common to all approaches is a general purpose memory interface. The memory bandwidth required for turbo decoding is much higher compared to the bandwidth available in processors. ASIC design provides the freedom to define a specialized memory interface. The ASIP design methodology includes

Table 9.1. Turbo decoding implementations based on programmable solutions

Architecture		Algorithm	Additional Information	Instruction Set	Clock Freq.	Throughput at (x iterations)	Ref.
					MHz	kbit/s	
1	STM 120	Max-LOGMAP	2 ALUs	VLIW	200	540 (5)	[132]
2	STM 120	LOGMAP	2 ALUs	VLIW	200	200 (5)	[132]
3	Starecore 140	Max-LOGMAP	4 ALUs	VLIW	300	1875 (5)	[132]
4	Starecore 140	LOGMAP	4 ALUs	VLIW	200	600 (5)	[132]
5	ADI TS	LOGMAP	2 ALUs	VLIW	180	666 (5)	[132]
6	SP-5 DSP	Max-LOGMAP	32 bit	SIMD	250	291 (8)	[133]
7	SP-5 DSP	Max-LOGMAP	16 bit	SIMD	250	582 (8)	[133]
8	SP-5 DSP	LOGMAP	32 bit	SIMD	250	158 (8)	[133]
9	SP-5 DSP	LOGMAP	16 bit	SIMD	250	158 (8)	[133]
10	TMS320C62x			VLIW	300	487 (8)	[134]
11	Pentium III	Max-LOGMAP		486 comp.	933	366 (1)	[135]
12	Pentium III	Const.-LOGMAP		486 comp.	933	296 (1)	[135]
13	Pentium III	Lin.-LOGMAP		486 comp.	933	262 (1)	[135]
14	Pentium III	LOGMAP		486 comp.	933	91 (1)	[135]
15	ARM922T	Max-LOGMAP			200	33 (4)	[136]
16	ARM922T+FPGA	Max-LOGMAP	ACS unit		65	20 (4)	[136]
17	ARM922T+FPGA	Max-LOGMAP	Recursion unit		50	60 (4)	[136]
18	ARM922T+FPGA	Max-LOGMAP	Max-LOGMAP unit		53	400 (4)	[136]
19	XiRisc (eFPGA)			CISC	100	270 (-)	[137][138]
20	RISC+eFPGA (A)	LOGMAP		CISC	100	330 (5)	[139][140]
21	RISC+eFPGA (B)	LOGMAP		CISC	133	1480 (5)	[141]

an exploration of the memory organization and therefore fulfills the demanded data throughput and programmability, as discussed in the following sections.

9.2 The Turbo Decoding Principle

The turbo decoding principle is depicted in figure 9.1. The Soft Input Soft Output (SISO) decoders operate on Log-Likelihood Ratios (LLRs) as defined in equation 9.1, where I_k denotes an information bit and y_0^{N-1} represents the complete sequence of received symbols.

$$L_k = \log \frac{P(I_k = 1 | y_0^{N-1})}{P(I_k = 0 | y_0^{N-1})} \quad (9.1)$$

The decoder input is the systematic information ($y_{0,k}$), parity information ($y_{1,k}$) and a-priori information (z_k). The a-priori information is derived from the a-posteriori output of the previous constituent decoder. The Turbo decoding principle is based on *several (nearly) statistically independent sources of reliability information on the received data-symbols*. Interleaver (π) and deinterleaver (π^{-1}) are used to remove the correlation between neighboring information bits.

For SISO decoder implementation the Maximum-A-Posteriori (MAP) decoding algorithm is the decoder of choice [142][143]. In order to reduce the computational complexity, the MAP algorithm is transformed to the logarithmic domain (LOGMAP). As shown in [129] and given in equation 9.2 the arithmetic complexity is further reduced by approximating the \max^* operation (or “E” operation) by a maximum selection and an additive correction function $f_c(|u - v|)$. A widely known approximation for $f_c(|u - v|)$ is $f_c(|u - v|) \approx 0$ [129].

$$\begin{aligned} \max^*(u, v) &:= \log(\exp(u) + \exp(v)) \\ &= \max(u, v) + f_c(|u - v|) \approx \max(u, v) \end{aligned} \quad (9.2)$$

9.3 The Max-LOGMAP Algorithm

The SISO decoders are based on the Max-LOGMAP algorithm, which is exhaustively described in [129]. The different steps of the Max-LOGMAP algorithm are presented here in order to justify the architectural decisions.

The whole algorithm is based on a butterfly decomposition of the trellis diagram, where m denotes a particular state and m' the successor state. The index k represents the time stamp. The corresponding

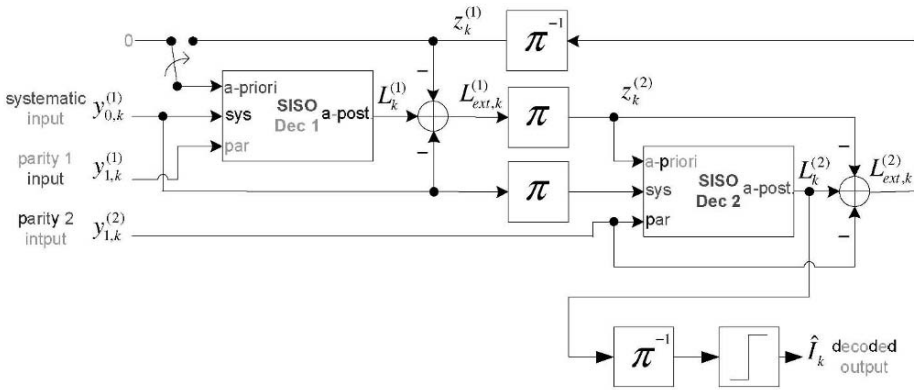


Figure 9.1. Turbo decoder structure. Source: F. Munsche [129], modified

trellis butterfly structure is depicted in figure 9.2. Within the trellis diagram the forward and backward path metrics represent the probability of a certain state, whereas the branch metrics represent the transition probabilities between states. The branch metrics are denoted by $t_k(m', m)$, forward path metrics by a_k , backward path metrics by b_k , the a-posteriori LLR by L_k , extrinsic LLR by $L_{ext,k}$, the received Symbol by $y_{j,k}$, the transmitted symbol by $x_{j,k}$ and finally, the a-priori information (LLR) by z_k .

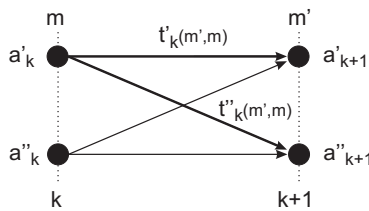


Figure 9.2. Trellis butterfly structure

Equation 9.3 represents the **Branch Metric Calculation (BMR)**, which benefits from the utilization of symmetric branch metrics.

$$t_k(m', m) := x_{0,k}(m', m) \cdot (y_{0,k} + z_k) + \sum_{j=1}^{n-1} (y_{j,k} \cdot x_{j,k}(m', m)) \quad (9.3)$$

The **Forward Recursion** (FR) given in equation 9.4 and the **Backward Recursion** (BR) in equation 9.5 calculate the path metrics in the trellis diagram.

$$a_{k+1}(m) = \max_{m'} (a_k(m') + t_k(m', m)) \quad (9.4)$$

$$b_k(m') = \max_m (b_{k+1}(m) + t_k(m', m)) \quad (9.5)$$

The **a-posteriori LLR Calculation**, also called Metric Combining (MC), is described by equation 9.6.

$$L_k := \max_{m', m | I_k=1} (a_k(m') \cdot t_k(m', m) \cdot b_{k+1}(m)) - \max_{m', m | I_k=0} (a_k(m') \cdot t_k(m', m) \cdot b_{k+1}(m)) \quad (9.6)$$

Finally, the **extrinsic LLR Calculation** (LLRC) is given in equation 9.7.

$$L_{ext,k} = \frac{L_k}{2} - y_{0,k} - z_k \quad (9.7)$$

As implied in the description of Max-LOGMAP algorithm, the forward recursion and backward recursion cause a large amount of intermediate results. Therefore, a crucial aspect of Turbo decoder implementations is to reduce the memory requirement. This is achieved by dividing the received data into sub-blocks (windows). The operations captured in equations 9.3 to 9.7 are now executed per window. Due to the sub-block based processing order, the initial data required for backward recursion needs to be approximated by the so called *backward acquisition* (BA), performed on the subsequent window. Therefore, the reduced memory requirements goes at the cost of computational effort.

The schedule of operations depicted in figure 9.3 and described in [129] is chosen for further discussion. This schedule provides two advantages compared to alternative schedules. First, the lifetime of intermediate results (and therefore memory requirement) is low. As indicated in figure 9.3 by the grey areas, the forward path metrics as well as the branch metrics needs to be stored for a maximum of two windows. Second, the latency counts only two times the windows size.

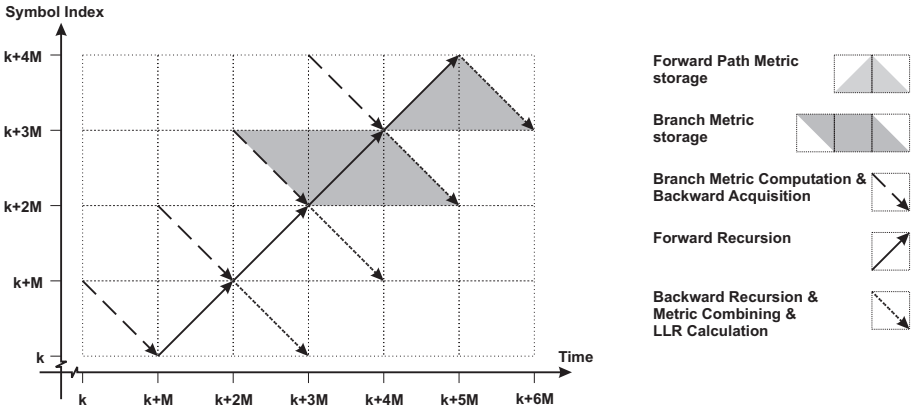


Figure 9.3. Schedule of operations, Source: F. Munsche [129], modified

9.4 Memory Organization and Address Generation

The evaluation of given programmable solutions for Turbo decoding in table 9.1 reveals that unsuited memory interfaces are the major reason for the low throughput. This section discusses the high demands on the memory organization, which are not fulfilled by general purpose solutions. Similar to the possibilities of ASIC development, the ASIP approach also allows to chose any desired memory organization.

Closely coupled to the question of memory organization is the address calculation issue. ASIC implementations utilize state machines determining the next address for each memory access. However, considering a programmable solution the address generation should be generalized and decoupled from a pure Turbo decoding purpose to allow its efficient usage in similar applications.

9.4.1 Memory Organization

The memory organization is derived from the given algorithm and the schedule of operations. Table 9.2 lists the memories implemented, their quantity and bit width.

The size of the three input memories, which store systematic and parity bits, and the size of the extrinsic memory is defined by the maximum blocksize of 5114 (5120 incl. tail bits) given by the 3GPP standard. The utilized memories support 2^n elements, thus the input memories have

Table 9.2. Memory organization for Turbo decoding

Memory Usage	Quantity	Bit width	Elements	Size
Systematic and Parity Bits	3	5 bit	8192	40960 bit
Extrinsic Interleaver	1	6 bit	8192	49152 bit
Branch metrics	1	13 bit	8192	114688 bit
Path metrics	1	$2 * 7 = 14$ bit	64	896 bit
Program	4	$4 * 11 = 44$ bit	64	2816 bit
	1	32 bit	4096	

$2^{\lceil \log_2(5120) \rceil} = 2^{13} = 8192$ elements. The size of the branch metrics memory as well as the path metrics memory is determined by the duration required to store intermediate values. A window size of 32 samples multiplied with a duration of two windows for this schedule results in 64 storage elements.

The memory organization as well as the algorithm steps lead to the specific memory utilization (R=Read/W=Write) given in table 9.3. The algorithm's operations are ordered according their execution from left to right. The input data, namely systematic, parity and extrinsic information is processed by the branch metric computation and stored in the respective memory. Both forward recursion and backward recursion utilize the calculated branch metrics and store their results in dedicated memories. Also the backward acquisition utilizes the branch metrics and stores the final acquisition result in the backward path metrics memory. The metric combining step reads the path metrics and hands over the results (via registers) to the extrinsic LLR calculation. The extrinsic memory is used to store the results.

Table 9.3. Memory utilization by operations (R=Read/W=Write)

Memory	Operation					
	BMC	FR	BA	BR	MC	LLRC
Systematic and Parity Bits	R/-					
Extrinsic	R/-					-/W
Branchmetrics	-/W	R/-	R/-	R/-		
Pathmetrics (forward)		R/W			R/-	
Pathmetrics (backward)			-/W	R/W	R/-	

Adopting the schedule given in figure 9.3 the execution of branch metric calculation, backward acquisition and forward recursion are not conflicting with other steps of the algorithm, as they are each exclusively

executed for different windows. Data dependencies are existent between backward recursion, metric combining and extrinsic LLR calculation. These dependencies are considered while mapping the equations to processor instructions including operand load and result storage. Data forwarding (and bypassing respectively) is a typical mechanism in processor design to resolve data dependencies and is therefore not further discussed here.

Additional to the memory organization, an efficient addressing mechanism is required, which is discussed in the following section.

9.4.2 Address Generation

Derived from the schedule of operations in figure 9.3, there are two special addressing schemes given and depicted in figure 9.4. The circular buffer schemes in forward and backward direction are derived from path metric and branch metric calculation. Also, metric combining utilizes this addressing scheme. The access to input memories and extrinsic data memory requires a more complicated addressing scheme. Here, the memory pointer is moved forward by an increment of two times the window size after processing one window in backward direction.

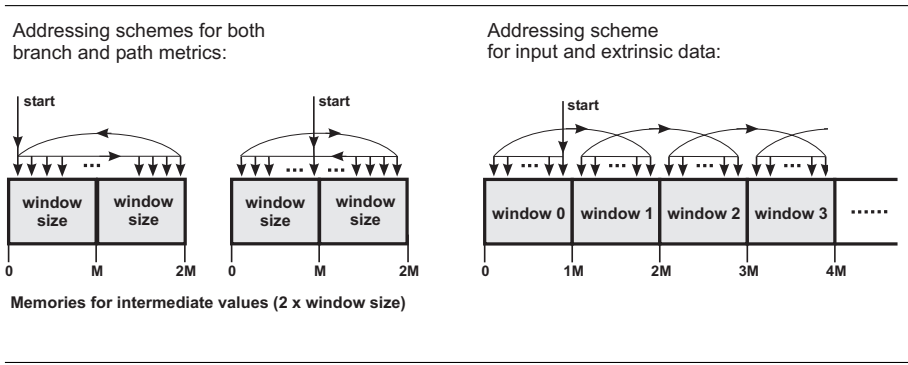


Figure 9.4. Addressing schemes derived from the turbo decoding algorithm

The required addressing schemes can be combined to one flexible address calculation scheme within the architecture. The underlying idea is to use a circular buffer and extend the address calculation by generalized features. A circular buffer is fully defined by the start and end address of the memory, which stores the data of the circular buffer. In a generalized version of a circular buffer, a starting pointer needs to be

initialized. Operations allowed on a circular buffer are address increment and decrement.

The basic realization of a circular buffer is extended to support a sliding memory range for the circular buffer. This is specified by an additional parameter, that represents the positive or negative shift of memory bounds and start address. The shift is initiated if the address pointer has completed one circulation on the current memory range. This generalized addressing scheme is a superset of the addressing schemes demanded by the turbo decoding algorithm and thus may also be advantageous for other applications.

The generalized concept is depicted in figure 9.5. Additionally, this figure illustrates the handling of a corner case where the memory size is not a multiple of the circular buffer size. To handle this corner case an additional parameter representing the upper bound (termination address) is required.

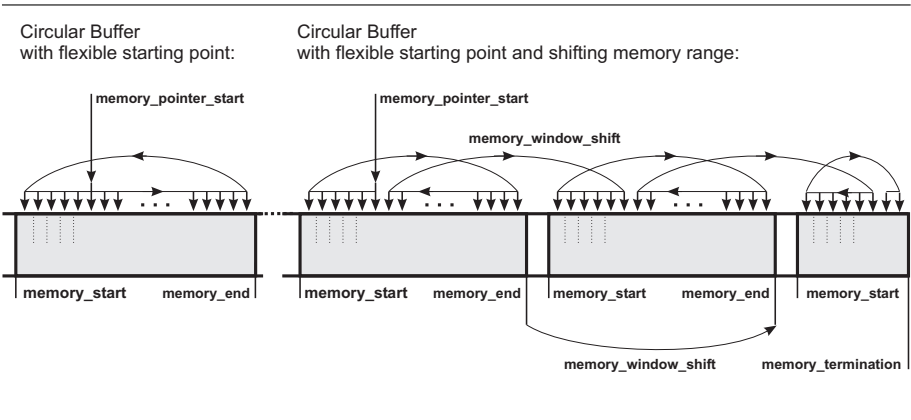


Figure 9.5. Supported addressing schemes by the address registers

The LISA operation which covers the address calculation is given in example 9.1. The variables used are directly derived from the illustration in figure 9.5. The address calculation is initiated by pre- or post-modifying access to address registers, such as $AR[0]++$, $AR[2]--$ or $--AR[1]$.

Therefore, address calculation is transparent to the software designer. The addressing scheme is defined by an address register initialization. The initialization is based on two instructions. The first one specifies the upper and lower bound of the circular buffer. All other parameters are initialized to reasonable values, thus the address register can already be used to realize a circular buffer. The second initialization instruction

```

OPERATION address_calculation {

    // Address calculation supports pre-/post increment/decrement
    // Address calculation is initiated by address register access
    // Example:      post-increment:  AR[0]++
    //              reading, no modification:  AR[2]
    //              pre-decrement:  --AR[0]

    BEHAVIOR {
        unsigned int ai = g_address_register_index;
        unsigned int address = memory_pointer[ai];

        // Publish the current address for further usage

        old_address = address;
        if(g_do_post_modification)
            new_address = memory_pointer[ai];

        if(g_do_decrement)
        {
            // Check whether address points to the lower bound:

            if(address == memory_start[ai])
                address = memory_end[ai];
            else
                address--;
        }
        else if (g_do_increment)
        {
            // Check whether address points to the upper bound:

            if(address == memory_end[ai])
                address = memory_start[ai];
            else
                address++;
        }
        if((g_do_increment || g_do_decrement) && address == memory_pointer_start[ai])
        {
            // Move the window only if we are around => check increment / decrement

            int shift = memory_window_shift[ai];
            if(memory_end[ai] + shift < memory_termination[ai])
            {
                memory_start[ai]      = memory_start[ai]      + shift;
                memory_end[ai]        = memory_end[ai]        + shift;
                memory_pointer_start[ai] = memory_pointer_start[ai] + shift;
                address                = memory_pointer_start[ai] + shift;
            }
            else
            {
                // Exception handling for termination ...
            }
        }
        memory_pointer[ai] = address;
        if(g_do_pre_modification)
            new_address = address;
    }
}

```

Example 9.1. LISA code realizing the address calculation

specifies the parameters required to benefit from the extended capabilities, such as start pointer, memory shift and termination address. Different from the naming in figure 9.5, example 9.1 uses arrays for all parameters to efficiently model multiple address registers. The variables prefixed with “g_” specify the type of address calculation to be performed. In Turbo decoding as well as other sequential data processing algorithms, often two successive data samples are accessed. Therefore, this implementation provides the unmodified address (`old_address`) and modified address (`new_address`) to the special purpose Turbo decoding instructions.

9.5 Instruction-Set and Data-Path Implementation

The currently implemented instruction-set is strictly tailored to optimally fit the Turbo decoding algorithm and the Max-LOGMAP realization. Additionally, general purpose instructions can be used to alter the Turbo decoding algorithm.

The data-path implementation implicitly results from the instructions’ behavior descriptions. LISA provides the possibility to group certain operations (instructions) to one particular unit. The optimizations presented in chapter 7 enable an efficient hardware implementation, as multiple arithmetic resources are shared. The definition of units within this particular architecture is derived from commonly known concepts, such as butterfly segmentation and Add-Compare-Select (ACS) units. These concepts are also utilized in ASIC design [129] and the implementation of (re)configurable solutions [137][141][143] and therefore not discussed here.

9.6 Instruction Schedule

The schedule of operations and thus instructions is already given for the kernel execution in figure 9.3. However, a quite important fact is the requirement for prologue and epilogue processing in Turbo decoding. Figure 9.6 indicates the required prologue and epilogue processing on the basis of windows. Furthermore, the overall schedule is a superposition of multiple iterations, a window based processing scheme and data processing based on a *Trellis diagram* [129]. This topic is closely related to the loop implementation in general.

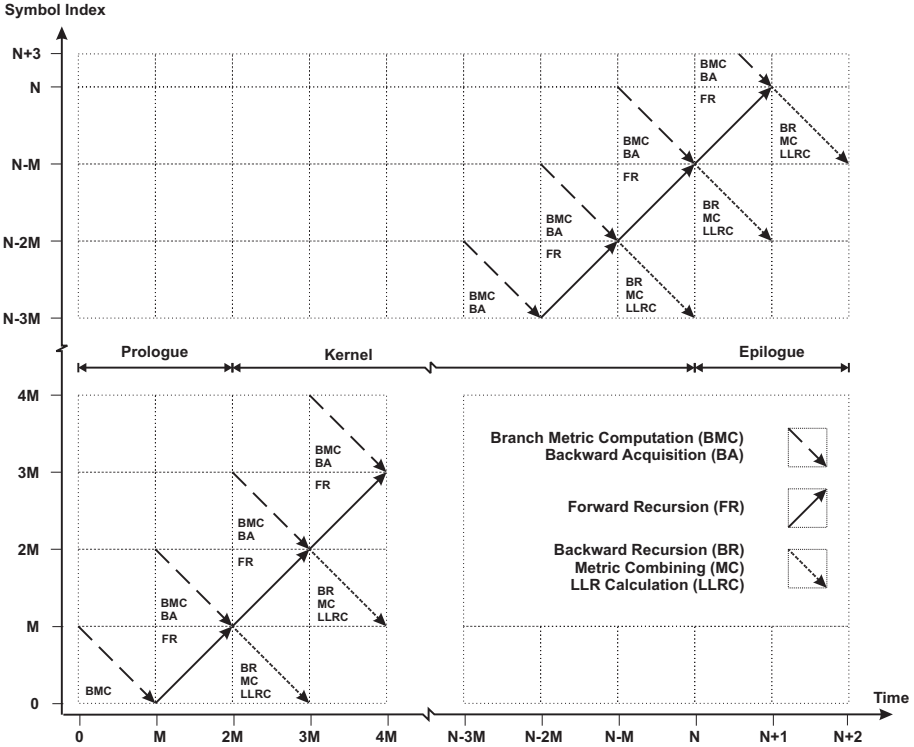


Figure 9.6. Prologue and epilogue in Turbo decoding

9.6.1 Zero Overhead Loops

Zero Overhead Loops (ZOLPs) are used to realize loops without additional latency introduced by jump or branch instructions. A loop is initialized by a start and end label, representing the boundaries of the loop, and a number of jumps or execution counts. The execution of a jump is controlled by dedicated logic, and thus conditional branch instructions within the loop body are eliminated. The concept of ZOLPs outperforms conditional branch instructions especially for small loop bodies as the ratio of data processing instructions to loop control instructions is improved. In this architecture ZOLPs are also utilized to realize an efficient and flexible prologue and epilogue handling.

The realized architecture not only supports a single ZOLP implementation, but a stack of several ZOLPs. This means, several nested loops can be successively initialized and executed. The top most loop of the

stack is active and taken for the ongoing pc calculation. Once a loop is completely processed, the loop is removed from the stack and the next (or none) loop gets activated.

9.6.2 Prologue and Epilog Handling

Figure 9.6 depicts the boundaries of the instruction schedule for Turbo decoding. The prologue and epilogue depicted here result from the window based processing scheme, which utilizes the independent execution of branch metric calculation, backward acquisition and forward recursion.

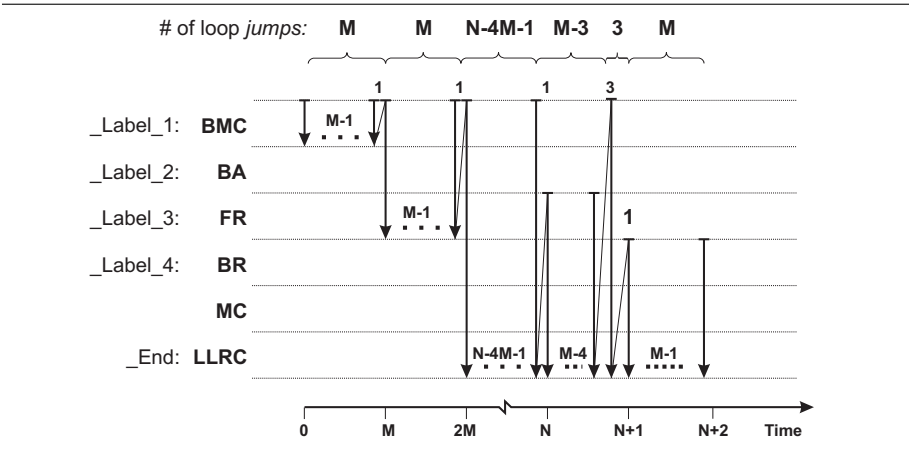


Figure 9.7. Prologue and epilogue handling

A generalized view to the challenge of prologue and epilogue handling is given in figure 9.7. The instruction sequences are indicated by the abbreviations BMC, BA, FR, BR, MC, LLRC (cf. figure 9.6). The loops executed due to the window based processing scheme are also depicted. Loops because of the execution of multiple iterations or interleaved and non-interleaved access to memories are omitted for the reason of clarity.

The concept of a ZOLP stack is reused to realize an efficient prologue and epilogue handling. To implement the schedule of instructions given in figure 9.7 several loop initializations are required. For example, the BMC needs to be performed M times, thus M-1 jumps are executed. Furthermore, a single jump is required to proceed to the next window and its calculation, here BMC, BA and FR. These jumps are combined to one ZOLP initialization with the start address `_Label_1`, the end address `_Label_2` and a jump count of M. The initialization of all required loops is given in example 9.2.

```

; ===== Loop stack is build up: last loop first! =====
; ===== Epilogue =====
START_LOOP START=_Label_4 END=_End      JUMP=M
START_LOOP START=_Label_1 END=_End      JUMP=#3
START_LOOP START=_Label_3 END=_End      JUMP=M-3

; ===== Kernel =====
START_LOOP START=_Label_1 END=_End      JUMP=N-4M-1

; ===== Prologue =====
START_LOOP START=_Label_1 END=_Label_4  JUMP=M
START_LOOP START=_Label_1 END=_Label_2  JUMP=M

```

Example 9.2. Loop initialization for prologue and epilogue handling

9.7 Pipeline Structure

Computations as well as data-routing (section 9.4) are well balanced over a four-stage pipelined structure, comprising the stages fetch, decode, execute and writeback. The distribution of dedicated units to the pipeline stages is listed in the following and depicted in figure 9.8.

Fetch stage

This pipeline stage comprises the instruction fetch mechanism and program counter calculation. Logic to control the zero overhead loop stack is also located in this pipeline stage.

Decode stage

The primary task of this pipeline stage is to decode the instruction, set the corresponding control signals to the subsequent stages and to load the operands from the memories. Therefore, also logic for the address calculation presented in section 9.4.2 is located in this stage. The current realization can calculate two addresses in parallel. Moreover, operands may be routed by a switchbox [129] according to the butterfly processing scheme.

Execute stage

The execute stage contains four ButterFLy units (BF units), each of them made up of two Add-Compare-Select (ACS) units. Thus, up to eight states within the trellis diagram can be processed simultaneously. These units are utilized for every data processing task.

Writeback stage

The writeback stage writes the results to the dedicated memories, also considering the butterfly processing scheme in Turbo decoding. Execution and writeback has been separated into two different stages to shorten the critical path.

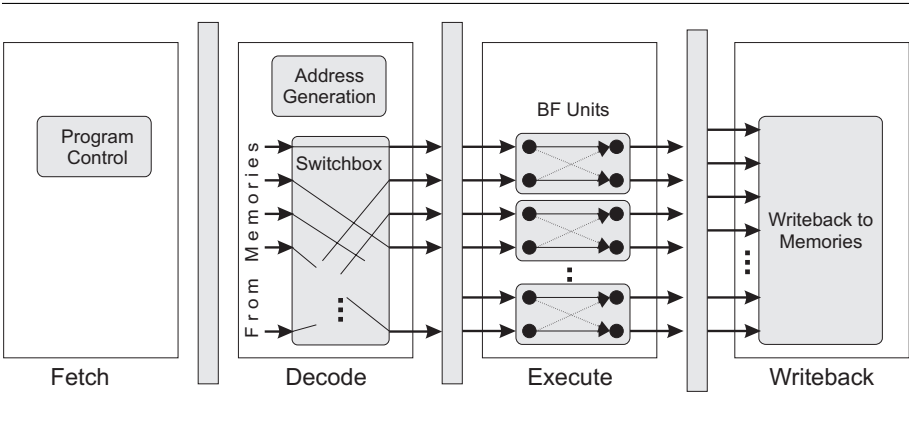


Figure 9.8. Pipeline organization

9.8 Results

The iterative design of this ASIP for Turbo decoding is fully based on the LISA processor design methodology. This comprises LISA model development, its verification within Synopsys' CoCentric System Studio, automatic implementation and gate-level synthesis. The LISA model counts 64 LISA operations and 5842 lines of code. The generated VHDL code comprises 33651 lines of code. Within six months of development time the architecture has been steadily optimized up to the results presented in the following.

The technology library and synchronous SRAMs from Virtual Silicon Technology (VST) [144] is designed for a 0.13 μm process of the foundry United Microelectronics Corp. (UMC) [145]. Gate-level synthesis with Synopsys' Design Compiler [85], worst case conditions, achieved a clock frequency of 164 MHz (6.1 ns critical path). The whole architecture requires 352.5 kGates. The complete memory consumes 326 kGates, the register file 10.5 kGates and the pipeline structure, including data-path and control-path, consumes 16.0 kGates. The memory dominated area distribution can be seen in table 9.4.

Table 9.4. Architecture size of the Turbo decoding architecture

Module (and hierarchically included modules)	Module Area		Average Power Consumption (@ 125 MHz)	
Pipeline Stage Fetch (FE)	0.59 kGates	0.17%	0.07 mW	0.89%
Pipeline Stage Decode (DC)	4.68 kGates	1.33%	0.89 mW	11.37%
Pipeline Stage Execute (EX)	3.13 kGates	0.89%	3.06 mW	38.92%
Pipeline Stage Writeback (WB)	1.68 kGates	0.48%	0.61 mW	7.82%
Pipeline Register FE/DC	0.35 kGates	0.10%	0.22 mW	2.77%
Pipeline Register DC/EX	3.98 kGates	1.13%	1.87 mW	23.76%
Pipeline Register EX/WB	1.60 kGates	0.45%	1.12 mW	14.29%
Register File	10.50 kGates	2.98%	2.49 mW	22.31%
Systematic and Parity Memory	101.00 kGates	28.65%		n/a
Extrinsic Memory	38.00 kGates	10.78%		n/a
Branchmetrics Memory	3.00 kGates	0.85%		n/a
Pathmetrics Memory	28.00 kGates	7.94%		n/a
Interleaver Memory	70.00 kGates	19.86%		n/a
Program Memory	85.00 kGates	24.11%		n/a
Memory Control Logic	1.00 kGates	0.28%	0.42 mW	3.74 %
ALL	352.50 kGates	100.00%	11.17 mW	100.00 %

The throughput achieved was 2.34 MBit/s at 5 iterations, thus fulfilling the 3GPP standard.

The average power consumption is also depicted in table 9.4. The results are measured with Synopsys' Prime Power and based on a Turbo decoding application with two iterations at 125MHz. An average power consumption of 11.17 mW and the application runtime of 49072 ns lead to 2.49 nJ per sample (Energy per sample).

9.9 Concluding Remarks

The ASIP presented here is a starting point for future research. Based on a specialized realization the data throughput of 2.3 MBit/s at 5 iterations outperforms the existing implementation on a processor. The high data throughput is achieved by realizing a specialized memory organization, a sophisticated address calculation, zero overhead loop control and a customized data-path.

Still a challenging task is to further improve the flexibility of the architecture. In general, both the flexibility and the high performance of ASIPs are reflected in the ASIP's instruction-set [91]. The flexibility is enabled by general purpose instructions which steer simple data-path operations. The high performance is achieved by specialized complex instructions which simultaneously trigger multiple arbitrary data-path operations. In the current implementation, the complex instructions

are extensively used to achieve a high data-throughput. An altered implementation of the Turbo decoding algorithm can only be achieved by the additional utilization of general purpose instructions which significantly degrades the performance. Therefore, future research will focus on architectures which provide flexibility at a lower cost of performance.

Chapter 10

CASE STUDIES: LEGACY CODE REUSE

The increasing fraction of software based solutions implies a growing interest of efficient application software reuse. One possibility is to simply move to the next processor generation of the same architecture family. This provides the advantage of compatibility between the existing software and the new processor. However, this is often accompanied by a moderate performance increase since supplemental features are not utilized by the legacy code. Therefore, the initial processor must be revised in order to achieve a higher performance increase.

With the LISA processor design platform, including an optimized hardware implementation, the efficient reuse of software becomes possible. After introducing the various possibilities of legacy code reuse, two case studies are presented, proving the benefits of software IP reuse.

10.1 Levels of Compatibility

Software may be written either in a high level programming language (e.g. C) or in an architecture-specific assembly language. A high level programming language provides the advantage of architecture independence and higher design efficiency. Contrary to this, a realization on assembly level offers the potential for a more efficient implementation in terms of code size and code quality. In general, software tools, such as C-compiler, assembler and linker, are utilized to convert the software from one level to the next. Either way, the software must be converted to a binary representation in order to be executed by the processor. Figure 10.1 shows the different levels of software development.

The basic idea is to redesign the original processor respecting software compatibility as well as increased performance. Due to advances in processor design, such as novel pipelining techniques, increased memory

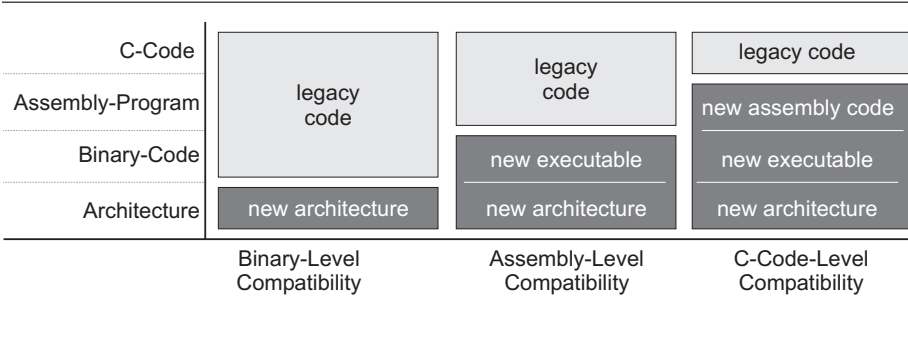


Figure 10.1. Levels of compatibility

bandwidth or enhanced data-path functionality, higher performance can be gained. The particular design goals strongly depend on the application-specific requirements. The different levels of software compatibility, according to figure 10.1, are discussed bottom-up regarding the level of abstraction in the following.

Binary-level compatibility refers to the final executables, which can be run without any modification on the new architecture. Therefore, the existing application software design flow remains unchanged. Only the underlying architecture changes according to given requirements.

Assembly-level compatibility allows changes to the architecture as well as to the binary encoding of the instruction-set. Moderate changes to instruction encoding facilitate minor decoder and operand fetch optimizations. The overall architecture performance and architectural efficiency might be improved by a fundamental reorganization of the instruction encoding, for example an instruction encoding with different bit-lengths. Therefore, a new assembler, which certainly can be generated from the LISA description, is required as existing assembly code needs to be mapped according to the new coding specification.

C-code level compatibility exists as soon as a C-compiler is available for the new architecture. Any code given in the C-programming language can be reused as the C-compiler performs the architecture-specific mapping to the assembly instruction-set. This is also possible for architectures developed with LISA, as the C-compiler can be generated automatically. However, as this approach is applicable generally, it is not discussed further here.

The optimization potential for binary-level compatibility is moderate, as the given instruction's encoding only enables minor architectural changes. Therefore, the following two case studies are targeting the assembly-level compatibility.

10.2 The Motorola 68HC11 Architecture

The targeted architecture is assembly-level compatible with the Motorola 68HC11 family. The different versions of this processor family differ in memory support, external modules and i/o interfaces. The original M68HC11 architecture supports instruction bit widths of 8 bit, 16 bit, 24 bit, 32 bit and 40 bit. Instructions are processed independently of each other by a finite state machine. The operations of one instruction are rarely executed concurrently. The memory interface provides a single port for exclusive instruction or data access. This memory interface causes the limited performance and instruction throughput. The number of cycles an instruction requires to complete depends on the instruction fetch cycles, the operands to fetch and the arithmetic computation. Considering an optimization regarding the instruction throughput, these three aspects must be taken into account. Two of them are related to the limited memory interface in the original M68HC11 architecture.

10.2.1 Memory Interface Optimization

The elementary change to the M68HC11 architecture is the redefined memory interface. While a single port, 8 bit memory interface is used in the original version of the architecture, a dual port, 16 bit memory interface is used in the enhanced version. The increased bit width directly improves the instruction throughput. The additional port is used to separate the instruction and data access to the memory. Therefore, these two types of accesses can be performed simultaneously, stalls due to memory accesses are not required anymore.

10.2.2 Instruction-Set Encoding and Instruction Fetch Mechanism

The new 16 bit memory interface for instruction fetch provides means to reduce the number of instruction fetch cycles. Figure 10.2 shows the number of instructions per instruction bit width for the old and new architecture. The instruction bit width directly influences the required clock cycles for instruction fetch. Besides this, the instruction fetch cycles for the original M68HC11 (and the compatible LISA model) are depicted in figure 10.2.

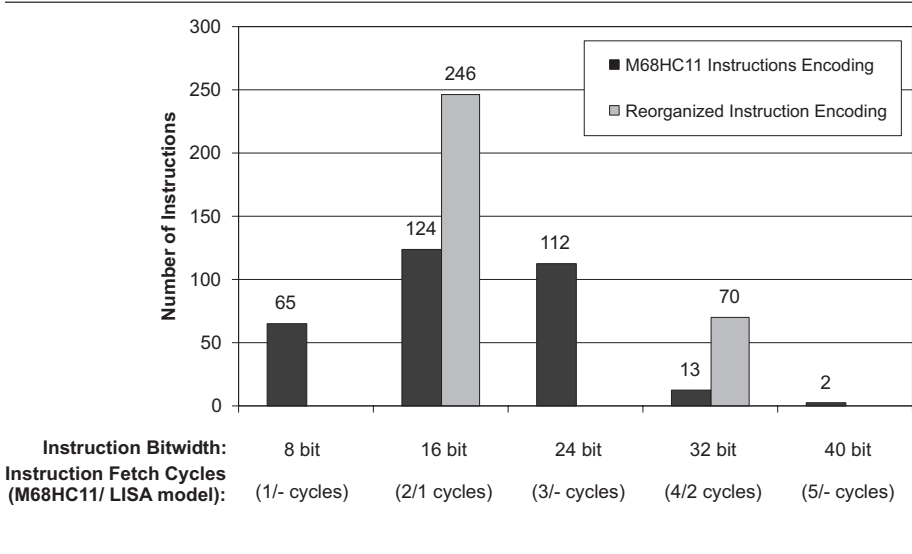


Figure 10.2. Instruction bit width in M68HC11 and compatible LISA model

The increased memory bit width of 16 bit enables an improved instruction throughput. However, the instruction-set [146] is carefully investigated to find an optimum solution, as described in the following.

Sequential instruction fetch

A first guess is to always fetch 16 bit and reorganize the instruction word in the fetch pipeline stage of the architecture. This would mean to cache not yet required portions of the instruction word. For example, if the first 8 bit represent a complete instruction, the second 8 bit need to be cached in order to concatenate the remaining instruction bits. However, this requires control logic to determine the instruction length and to reorganize the instruction word. This logic is certainly not allowed to influence the physical characteristics.

Investigating the original M68HC11 instruction-set, the instruction's operation, the utilized addressing mode and the number and type of operands are jointly encoded within a single opcode field. An example instruction is given in table 10.1. Moreover, the current instruction-set emerged from several versions of the architecture as, for example, new addressing modes have been added. Thus, the instruction-set is not optimally organized in today's point of view. The required effort to determine the instruction length becomes close to the effort decoding the instruction completely. Certainly, this control logic is not applicable

Table 10.1. Extract of the Motorola 68HC11 instruction-set manual [146]

Instruction Mnemonic (Operation)	Addressing Mode	Instruction		Instruction Length
		Opcode	Operand	
ADCB(opr) (Add with carry to B)	B IMM	C9	ii	16 bit
	B DIR	D9	dd	16 bit
	B EXT	F9	hh ll	24 bit
	B IND,X	E9	ff	16 bit
	B IND,Y	18 E9	ff	24 bit

for implementation in the fetch stage. From this idea, it is desirable to insert particular bits within the instruction word, which indicate the length of the instruction.

The instruction encoding example in table 10.1 clearly indicates, that additional bits to encode the instruction length are leading to an inefficient solution. Thus, the current instruction-set is not suited to efficiently determine the instruction bit width. The necessary reorganization of instruction encoding is targeted in the following section.

Instruction cache mechanism based on optimized instruction encoding

In the following, an instruction-set, which is perfectly suited for an cached instruction fetch mechanism, will be derived.

The length of an instruction is basically influenced by the bit length of the operands. In the M68HC11 architecture, there are instructions with no, 8 bit, 16 bit and 24 bit operands. Therefore, a maximum bit width of 32 bit is targeted. Instructions without and with 8 bit operands are summarized in the new set of 16 bit instructions. Although increasing the instruction bit width from 8 bit to 16 bit for the instructions without operands, the time required for fetching these instructions does not change as the bit width of the memory interface is increased (cf. section 10.2.1). The original M68HC11 instructions with 16 bit and 24 bit operands are grouped to the new 32 bit instructions.

The remaining task is to find a suitable opcode encoding for the new class of 16 bit and 32 bit instructions. On the one hand, all operations and addressing modes must be covered properly, on the other hand, the bit width must be determinable efficiently.

For the set of 16 bit instructions with 8 bit operands, 8 bit remain to encode the opcode. When using one bit of the opcode field to encode the instruction's bit width, 7 bit will remain to encode the instructions' operations, thus, $2^7 = 128$ different 16 bit instructions can be encoded.

These 128 possibilities are far too less, as 173 instructions of this type must be encoded.

However, even though this straight forward approach is not suited, the underlying technique becomes clear. Three bit have been chosen to indicate the bit width of the instruction encoding. The first three bit of an instruction are indicating a 32 bit instruction, if all bits are set to one. Therefore, $2^8 - 2^5 = 224$ instructions with a bit width of 16 bit can be encoded.

The fetch mechanism must be able to determine the instruction bit width and to cache a 16 bit instruction part whenever necessary. However, the required control logic is quite simple, as only 3 bit need to be compared. Example 10.1 shows the behavior of the LISA operation `fetch`. First of all, a new 16 bit bundle is fetched and the flag `instr16_is_stored` indicates whether there is already another 16 bit instruction word cached. If so, the two parts are concatenated and fed into the pipeline for decoding. Otherwise, the flag `instr16_is_stored` is not set and the fetch mechanism decides whether this is a 16 bit instruction or the first part of a 32 bit instruction. Either the instruction word is cached and a NOP instruction is fed into the pipeline or the 16 bit instruction is simply written to the pipeline instruction register.

Due to the reorganization of the instruction encoding, the number of execution cycles per application is reduced by about 60%. Additionally, the code size was reduced by about 4%.

10.2.3 Architecture Structure

The overall architecture is depicted in figure 10.3. The architecture is based on a three stage pipeline, with the pipeline stages *fetch*, *decode* and *execute/writeback*. The specialized fetch mechanism and the two decoders for 16 bit and 32 bit instructions are depicted in the fetch stage and the decode stage respectively. The data-path has been implemented according to the freely available M68HC11 architecture specifications [146]. The register organization remained unchanged, while the memory interface was extended from 8 bit to 16 bit.

10.2.4 Gate-Level Synthesis Results

To produce gate-level synthesis results, Synopsys' Design Compiler (version 2003.06) [85] is used. The architecture is mapped using a commercial 0.18 μm technology library and worst case conditions.

Figure 10.4 summarizes the gate-level synthesis results with regard to different optimizations. The area is decreased to 74.78% and the timing

```

OPERATION fetch IN pipe.fetch {
  BEHAVIOR
  {
    instr = e_ram1[PC - E_RAM_LOW];

    if(instr16_is_stored == true)
    {
      // There is already the first instruction part cached
      // Concatenation of two instruction words required

      unsigned int 32bits_instruction = (instr16_stored << 16);
      32bits_instruction = (32bits_instruction | 0xFFFF) & instr;

      // Feed the instruction into the pipeline

      PIPELINE_REGISTER(pipe,FE/DC).instruction =
      32bits_instruction;
      instr16_is_stored = false;
    }
    else
    {
      if(((instr16 >> 13) & 7) == 7)
      {
        // This is the first part of a 32 bit instruction
        // Cache this part!

        instr16_is_stored = true;
        instr16_stored    = instr16;

        // Move NOP into the pipeline
        PIPELINE_REGISTER(pipe,FE/DC).instruction = 0;
      }
      else
      {
        // This is a 16 bit instruction, nothing special

        PIPELINE_REGISTER(pipe,FE/DC).instruction = instr16 << 16;
        instr16_is_stored = false;
      }
    }
  }
}

```

Example 10.1. Fetch mechanism in LISA 6811 compatible architecture

to 71.85%. Moreover, the implementation flexibility is clearly indicated by the AT-chart (Area vs. Timing chart). Multiple Pareto points are achievable by the LISA to RTL synthesis. The ratio of the architectural efficiencies ($\eta_{arch} = \frac{1}{AT}$) with the best timing and the best area result is $\eta_{min(timing)}/\eta_{min(area)} = 1.21$, which indicates further optimization possibilities. In this case, the forwarding mechanism specified in the LISA model crosses two pipeline stages. An altered description on the LISA

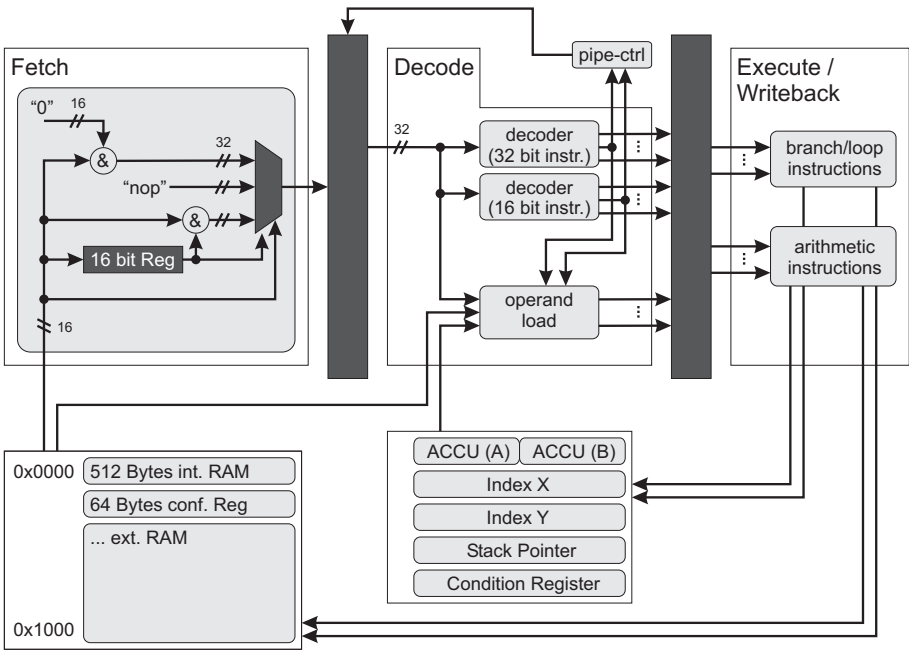


Figure 10.3. The compatible 68HC11 architecture designed with LISA

No.	HDL Synthesis Options	Gates	Clock
1	w/o Optimizations	19883	7.16 ns
2	Path Sharing	19462	6.68 ns
3	Decision Minimization	18801	6.74 ns
4	Path Sharing + Decision Min.	17134	6.51 ns
5	4 + Resource Sharing (single)	16640	6.74 ns
6	4 + Resource Sharing (multiple)	14869	9.06 ns

Gate-level syntheses are run with Synopsys Design Compiler and a 0.18 μm technology library (worst case conditions).

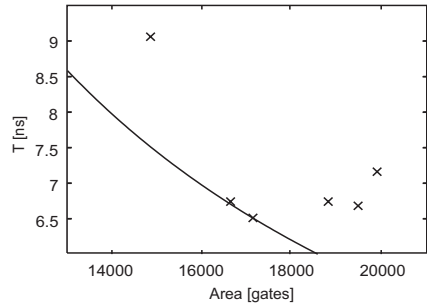


Figure 10.4. Synthesis results of the ISS-68HC11 architecture

level improves the timing of the architecture. However, the physical characteristics are valuable information for the architecture specifica-

tion. The information obtained on gate-level can easily be traced back to the LISA description on a higher level of abstraction.

The ISS-68HC11 architecture is comparable to handwritten cores. Although the ISS-68HC11 is improved with regard to the instruction throughput, the physical characteristics are comparable to the original architecture and its commercial implementations. For example, the Synopsys Design Ware [147] 6811 core occupies between 15 kGates and 30 kGates depending on the speed, configuration and target technology. The architecture is designed for applications running at clock frequencies of up to 200 MHz (at 0.13 μm) [148].

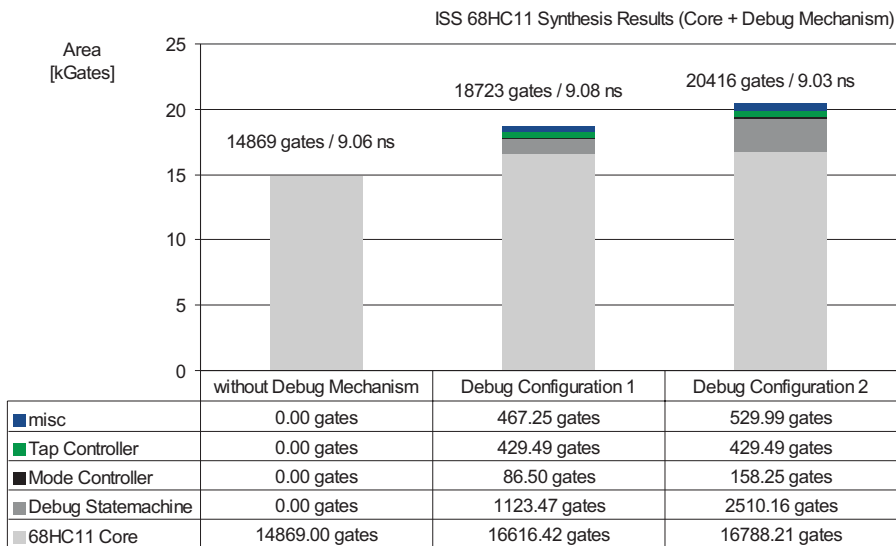


Figure 10.5. Synthesis results of the ISS-68HC11 architecture

Figure 10.5 shows the gate-level synthesis results with different debug configurations. Configuration one comprises two program breakpoints and access to the registers depicted in figure 10.3. Configuration two comprises four program breakpoints, data breakpoints for the registers and debug access to all registers. The timing is not influenced by the debug mechanism generation (9.06 ns, 9.08 ns, 9.03 ns), which is obvious in consideration of the debug mechanism implementation (section 8.2). The area is certainly increasing as additional logic is required to implement the debug mechanism and JTAG interface. An interesting aspect

is that the area of the architecture core hardly increases comparing debug configuration one and configuration two. This is due to the fact that the additional data breakpoint detection only requires a few logic elements within the architecture core. The main logic for breakpoint detection is implemented in the debug state machine, which is reflected by an increase of 1387 kGates.

10.3 The Infineon Technologies ASMD

The Application-Specific Multirate DSP (ASMD) is designed for the execution of interpolation and decimation filters [92][149]. This architecture consists of a highly application-specific data-path. Moreover, the instruction-set provides a static preemptive scheduling mechanism enabling an efficient filter implementation.

10.3.1 Instruction-Set

The instruction-set of the ASMD is suited for the implementation of static preemptive scheduling. This means that several filters can be executed pseudo parallel. The processor switches between different filters according to the designer's definition. A single time frame is called *slice*. The slice instruction is used to synchronize the filter execution with the overall system. Figure 10.6 depicts an example application illustrating the static preemptive scheduling. Here, two filters (`asmd1` and `asmd2`) are defined in the so called *sub-program* section. Sub-programs can be executed either completely (`asmd1`) or partially (`asmd2`). In the latter case, the partial execution is defined by labels. This schedule and the corresponding execution of sub-programs is defined in the *main-program* section.

The digital filters consist of simple shift and add operations, which can be computed easily from the filter coefficients. These operations can be intuitively transferred to the ASMD assembly instructions. Therefore, a high level language compiler is not required.

The software is written in a proprietary assembly language. The application is assembled by a PERL script, which writes out VHDL constants representing the machine code instructions. These constants are synthesized together with the handwritten architecture core. Thus, the ASMD controller is implemented as FSM representing the filters implemented in software.

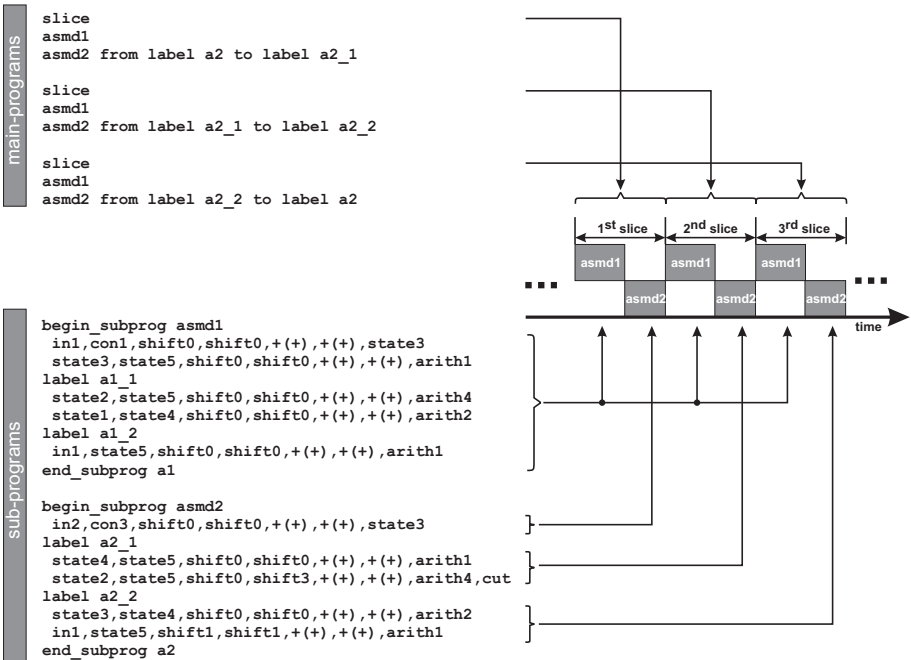


Figure 10.6. Assembly program and execution schedule. Source: [92], modified

10.3.2 The ASMD Data-Path

The data-path of the ASMD is shown in detail in figure 10.7. Arithmetic components such as adder, shifter and saturation units are normalized to a common data bit width.

The ASMD provides three types of operands. Regular storage elements are *arithmetic registers* and *state registers*. In addition, constants can be utilized for arithmetic calculations. The data-path is parameterizable concerning, for example, the bit width of registers or the number of registers available.

The data-path operates on two operands and produces up to three results. These are written to the internal registers or to the output port of the architecture.

10.3.3 LISA based Implementation of the ASMD

The original ASMD architecture and the utilized tool flow suffer from the disadvantages caused by the traditional design flow, as already

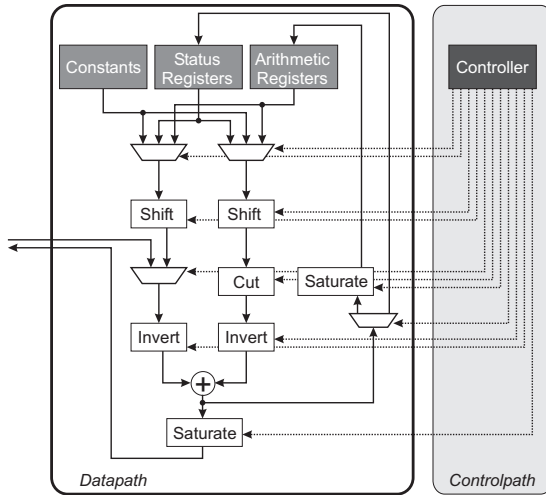


Figure 10.7. A structural view of the Infineon ASMD. Source: [92], modified

elaborated generally in chapter 1. Here, verification must be performed for every new application using VHDL hardware simulators. These simulators are known to be slower by several orders of magnitude compared to high level instruction-set simulators. Also, the handwritten PERL assembler is far less convenient and thus inefficient to use, compared to the one offered by the LISA processor design environment. One of the most important points, however, is the consistency of the PERL assembler and the architecture itself, which must be guaranteed by the designer.

Similar to the M68HC11 project, the LISA based ASMD should support assembly-level compatibility in order to reuse legacy code.

A straight forward mapping of the architecture to a pipelined version, results in the disadvantage of an exclusive execution of control and arithmetic instructions. In the original ASMD architecture the data-path operations are performed simultaneously to the control-path operations, determining the next data-path instruction to be executed.

A structural view of the architecture is depicted in figure 10.8. In order to be compatible to the original ASMD architecture, concerning timing and i/o behavior, two execution lanes need to be realized. Both lanes are based on an instruction-set. The first lane is dedicated to the arithmetic instructions, steering the data-path, which is essentially the same as in the original architecture. The second lane processes a control instruction determining the next two arithmetic and control instructions.

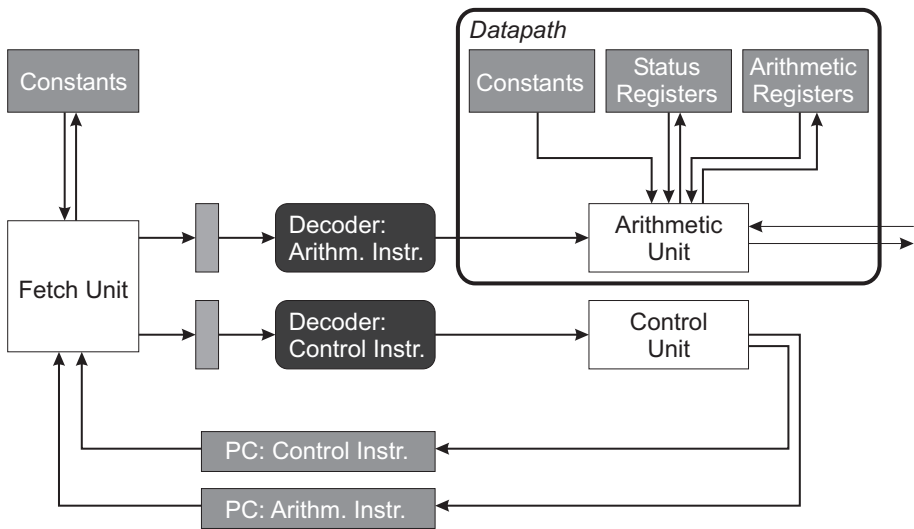


Figure 10.8. A structural view of new ASMD architecture. Source: [149], modified

Both instruction types are referred to by a dedicated program counter. The fetch unit loads both instructions from the program memory simultaneously. According to the original ASMD architecture the program memory has been implemented by constants which are fed into the gate-level synthesis here, too. Therefore, the original and LISA based ASMD can easily be compared: The data-path is described similarly in both versions, also the register file and program memory is implemented in the same way. The FSM of the original ASMD architecture is replaced by a pipelined sequence of *fetch* and *decode/execute*.

The LISA based approach provides the advantage of easy maintainability and extensibility because of one common architecture description. During the evaluation of the LISA processor design environment, gate-level synthesis results turned out to be in the same range of the original handwritten VHDL model. The utilization of the ASMD in a bluetooth device suggested to introduce an additional control instruction. The change of the LISA model, the regeneration of simulator and RTL hardware model as well as their verification was performed within one day. Due to this new application, the existing ASMD solution was replaced by the generated architecture.

10.3.4 Gate-level Synthesis Results

The gate-level synthesis was performed with the Synopsys' Design Compiler (version 2003.06) [85]. The architecture is mapped using a commercial 0.18 μm technology library and worst case conditions.

The generation of a debug interface is not reasonable, as the program is integrated into the ASMD's combinatorial logic. Therefore, on site debug capabilities are not required.

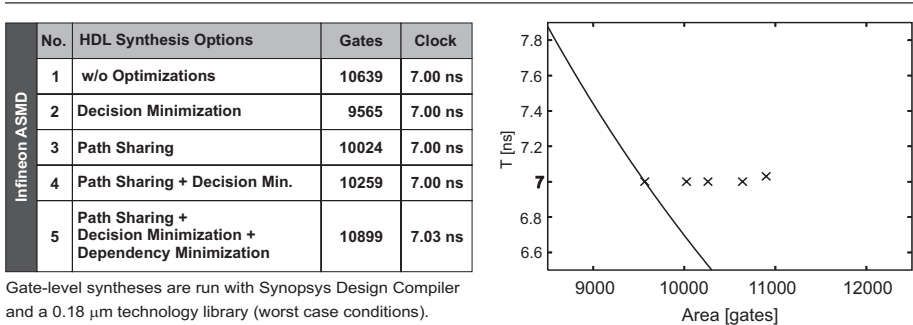


Figure 10.9. Synthesis results of the ASMD architecture. Based on: [88]

Due to the size of this architecture and a very simple data-path, the applied optimizations only have minor effect or sometimes even decrease the architectural efficiency, as summarized in figure 10.9. For example, the data-path does not provide the possibility to share resources. However, the achieved synthesis results are equal to the original handwritten implementation.

Chapter 11

SUMMARY

Today's SoCs increasingly require computational building blocks that provide high performance, energy efficiency as well as flexibility. These conflicting properties are balanced by ASIPs. However, ASIP design is unfortunately accompanied by a considerable complexity. This complexity stems from the diversity of ASIP design phases and the need for optimizations regarding a given application.

The traditional processor design methodology precludes iterative optimizations and thus efficient ASIP design due to a basically unidirectional design flow (cf. chapter 1). The architecture description language LISA provides a common basis for all design phases, including the design space exploration [23]. The model complexity is significantly reduced since the overall *design entry* is raised above RTL (cf. chapter 3). Due to the higher abstraction level a gap is teared between the design entry and the *architecture implementation* on RTL.

An automatic path to architecture implementation is strongly required because of two reasons. First, during design space exploration the effect of high level design decisions on the physical characteristics of the architecture needs to be considered. Second, the consistency between the reference model, software tools and the final architecture implementation has to be ensured. In the field of ADL based ASIP design the related work only targets a subset of the requirements raised by an automatic ASIP implementation (cf. chapter 2). For example, the generation of the complete architecture without applying optimizations is only suitable for a rough estimation of the physical characteristics during architecture exploration. Contrary to this, a partial generation of an RTL hardware model (such as structure, control-path and interconnect)

and an optimized manual implementation of the data-path only satisfies the requirements on the final architecture implementation.

An automatic ASIP implementation based on ADLs must be comparable to the manual implementation by experienced designers (cf. chapter 4). The demands on the architectural efficiency as well as the implementation flexibility must be satisfied. Also processor features commonly known from off-the-shelf processors, such as JTAG interface, debug mechanism or power save mechanisms, must be supported. Therefore optimizations and transformations are required. Unfortunately, these optimization and transformations cannot be performed on RTL due to the enormous computational complexity.

The contribution of this book is a novel RTL hardware model synthesis, which is based on the Unified Description Layer (UDL). The UDL abstracts the typical elements, process and signal, of hardware descriptions by the elements *unit* and *path*. For example, multiple signals required to access a register are grouped to a path. Furthermore, the UDL elements cover essential information about the semantics of the underlying logic and interconnect (cf. chapter 6). Thus, the computational complexity is significantly reduced compared to gate-level synthesis. Based on the UDL, the automatic ASIP implementation comprises optimizations (cf. chapter 7) and transformations to integrate processor features (cf. chapter 8).

Due to the higher abstraction level of ADLs a directly derived RTL hardware model implies noticeable redundancy. For example, in LISA the instructions' behaviors are described nearly independently of each other, although they may be associated with the same functional unit in hardware. The elimination of this redundancy is the primary goal of the proposed optimizations. The second target is to balance the physical characteristics of the architecture with regard to a particular application and design goal. The optimizations can be separately configured and multiple implementation alternatives can be evaluated. Thus, the optimizations ensure a sufficient architectural efficiency and implementation flexibility.

The UDL provides also the basis for transformations to automatically insert configurable processor features into the architecture structure. This concept has been exemplarily proven for an automatic generation of JTAG interface and debug mechanism. Additional processor features can be easily added to the framework in future research and development.

The numerous capabilities of the proposed automatic ASIP implementation are demonstrated in multiple industrial case studies. The Motorola 68HC11, Infineon's ASMD and ICORE are real world

architectures, which have been successfully modelled with LISA and implemented with the proposed synthesis tool. The comparisons to handwritten implementations, if possible, clearly show that the physical characteristics are equal and sometimes even superior at an significantly increased design efficiency. Also, CoWare's LT architecture family has been used to demonstrate the capabilities of the automatic ASIP implementation.

The presented case studies demonstrate that all demands on an automatic ASIP implementation are satisfied. Hence, the work presented in this book closes the gap between the ASIP design entry and implementation.

Appendix A

Case Studies

Various architectures have been developed to demonstrate the applicability of the presented concepts [87][93][94]. Three architectures are elaborated in detail in chapter 9 and chapter 10. This chapter presents further case studies and gate-level synthesis results. The results are discussed with regard to the optimizations and processor features presented in chapter 7 and 8.

The high design efficiency provided by LISA enables the designer to optimally map the architecture to an application. This includes design decisions about the instruction-set, data-path, processor structure, etc. The automatic ASIP implementation from LISA provides an architectural efficiency that is equal to handwritten implementations by experienced designers. Moreover, the implementation flexibility enables the designer to trade off the physical characteristics and to find the most suitable implementation. The presented synthesis results demonstrate the various degrees of freedom in ASIP implementation when using LISA. In this chapter, the architectural efficiency and implementation flexibility is given by AT-diagrams (Area over Timing diagrams).

To produce the gate-level synthesis results Synopsys' Design Compiler (version 2003.6) [85] was used. The architectures are mapped using a commercial 0.18 μm technology library, worst case conditions and individual timing constraints for each architecture. The gate-level synthesis results were obtained by a series of two successive syntheses. The first synthesis was performed with an unattainable timing constrained. The second synthesis was restricted to the timing achieved by the first synthesis run.

A.1 ICORE

The ICORE architecture [91] was developed for the acquisition and tracking in decoding Terrestrial Digital Video Broadcasting (DVB-T). It is a typical load-store 32 bit Harvard processor architecture with single instruction issue rate. About 60 DSP-like instructions are realized, which, for example, comprise arithmetic instructions, bit manipulation instructions, general control flow instructions and zero overhead loop instructions. Furthermore, highly optimized instructions support a fast CORDIC angle calculation.

Various realizations have been evaluated during design space exploration, for example by varying the number of pipeline stages from three stages up to five stages. The version which is discussed here comprises four pipeline stages, which are instruction fetch, instruction decode and operand load, instruction execution and result writeback. The ICORE uses eight general purpose registers, four address register, a program counter and registers to support zero overhead loop instructions.

Compared to other case studies in this book, the ICORE bears a significant potential for optimizations, because of the large amount of hardware resources (multiple registers with about 800 bit in total, about 10 functional units, etc.) along with a large number of exclusively executed instructions.

As shown in figure A.1, the area can be reduced to 58%, while the critical path can be reduced to 84% in the best case. Overall, the AT product is reduced to 60% in the best case, which marks a significant improvement. This architectural efficiency outperforms handwritten implementations (42 kGates, 8 ns, slightly modified architecture). A closer analysis of the results achieved with a handwritten implementation indicates that the modular structure and the size of the architecture strongly prevents gate-level synthesis tools from performing optimizations.

The synthesis results depicted in the AT-chart of figure A.1 result from six different synthesis configurations. Other results might be achieved with altered optimization constraints for the automatic ASIP implementation as well as the gate-level synthesis. Thus, a high implementation flexibility is provided as well.

The gate-level synthesis results for the ICORE architecture with debug mechanism and JTAG interface are shown in figure A.2. Here, three different ICORE syntheses are presented: without debug mechanism, with debug configuration one and with debug configuration two. Common to both debug configurations are the basic elements of the debug mechanism (mode controller, debug state machine) and the TAP controller of the JTAG interface. Configuration one comprises two program breakpoints and debug read/write access to the general purpose regis-

Infineon ICORE	No.	HDL Synthesis Options	Gates	Clock
	1	w/o Optimizations	50846	6.07 ns
	2	Path Sharing	44015	6.26 ns
	3	Decision Minimization	39400	6.08 ns
	4	Path Sharing + Decision Min.	37975	6.11 ns
	5	4 + Resource Sharing (single)	41930	5.12 ns
	6	4 + Resource Sharing (multiple)	29502	6.28 ns

Gate-level syntheses are run with Synopsys Design Compiler and a 0.18 μm technology library (worst case conditions).

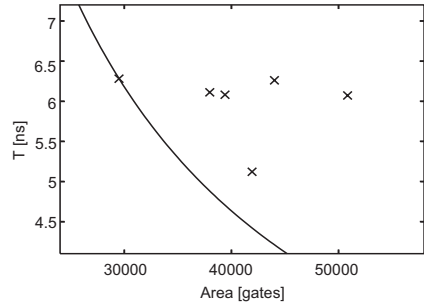


Figure A.1. Synthesis results of the ICORE

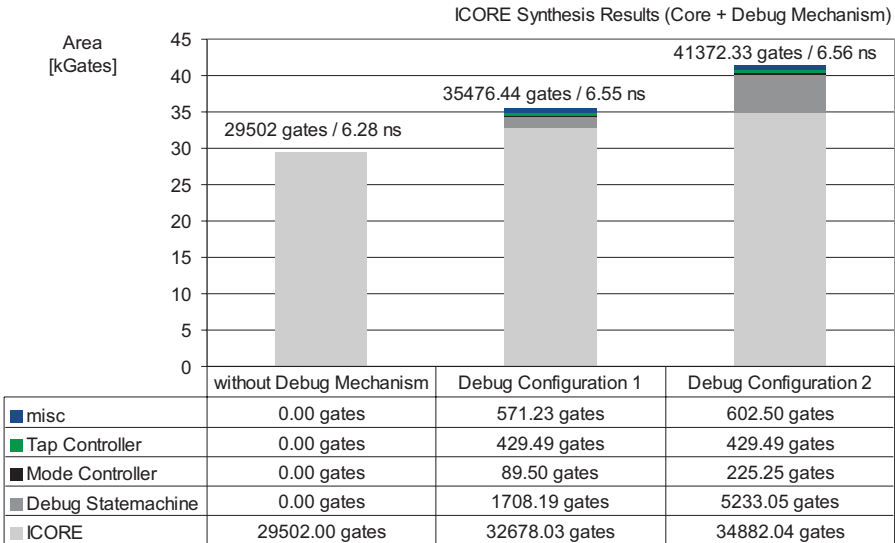


Figure A.2. Synthesis results of the ICORE

ter file. Configuration two comprises four program breakpoints, data breakpoints for the general purpose register file and debug read/write access to all registers. Certainly, the LISA to RTL synthesis tool allows a more fine-grained configuration of the debug mechanism on the level of registers and memories.

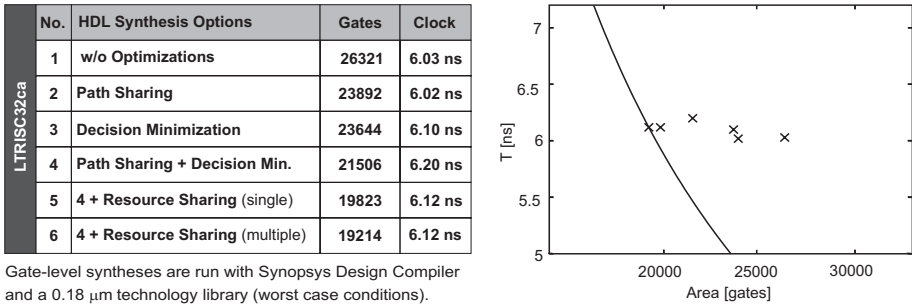


Figure A.3. Synthesis results of the LTRISC32ca architecture

The debug mechanism hardly influences the timing of the architecture (6.28ns, 6.55ns and 6.56ns). The little increase results from additional multiplexers in the path to the registers (section 8.2.6). The increase in area of the architecture *core* is 10.77% for configuration one and 18,24% for configuration two. The sizes of the mode controller and debug state machine certainly increase with their advanced functionality. The size of the TAP controller remains constant, as this component is independent from a particular configuration.

A.2 LT Architecture Family

The LT architecture family is dedicated for LISA training sessions and often used as starting point for new architectures. In general, the simple architectures only bear a minor potential for optimizations. The debug mechanism is included in two different configurations for each LT architecture. Configuration one comprises two program breakpoints and debug read/write access to the general purpose register file. Configuration two comprises four program breakpoints, data breakpoints for the general purpose register file and debug read/write access to all registers.

The **LTRISC** is a 32 bit RISC pipelined architecture, with 36 general purpose arithmetic, logic and control flow instructions. The area requirement is reduced by 27%, which still marks a significant improvement. However, the efficiency of the area optimization is limited by the low number of instructions. The timing nearly remains unchanged, as the data-path of the architecture is very simple (complex instructions which trigger a chain of arithmetic operations do not exist).

The debug mechanism has been automatically integrated in the LTRISC architecture. The timing remains almost unchanged (6.12ns,

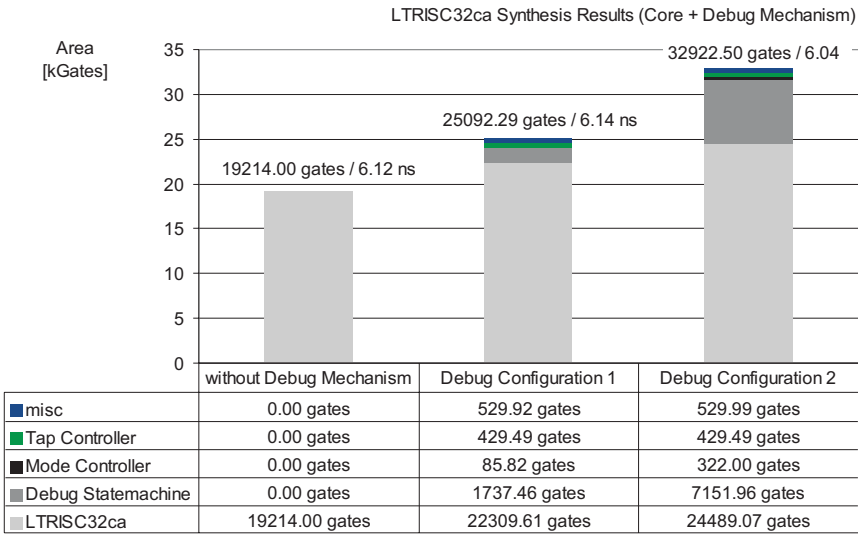


Figure A.4. Synthesis results of the LTRISC32ca

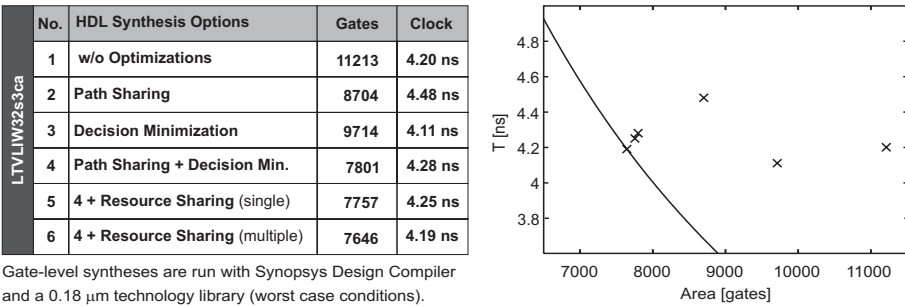


Figure A.5. Synthesis results of the LTVLIW32s3ca architecture

6.14ns, 6.04ns), while the overall architecture size increases by 71%. This area increase is caused by the breakpoint configuration for the large general purpose register file of 16 elements, each 32 bit wide. This example clearly indicates, that the number of architectural resources and the functionality of the debug mechanism must be carefully considered.

The **LTVLIW32s3ca** and **LTVLIW32s4ca** architectures are VLIW processors with a three and four stage pipeline respectively. The gate-level synthesis results are presented in figure A.5 and A.6.

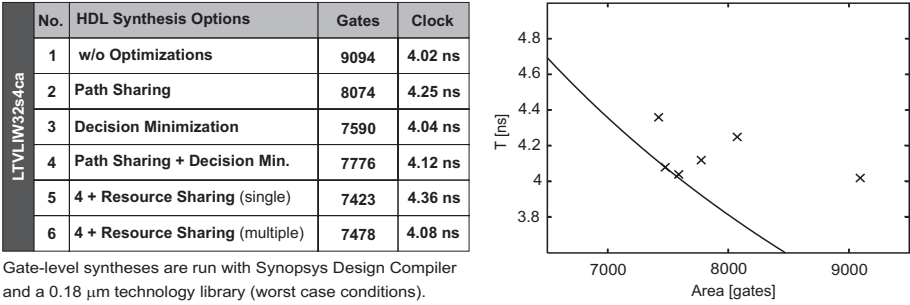


Figure A.6. Synthesis results of the LTVLIW32s4ca

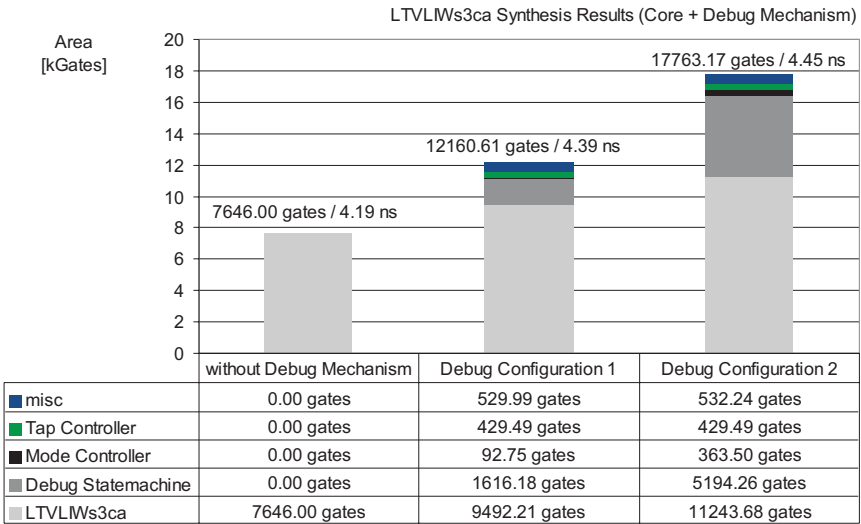


Figure A.7. Synthesis results of the LTVLIW32s3ca

The additional pipeline stage in the LTVLIW32s4ca architecture slightly reduces the length of the critical path, compared to the LTVLIW32s3ca. The automatic optimizations during HDL synthesis

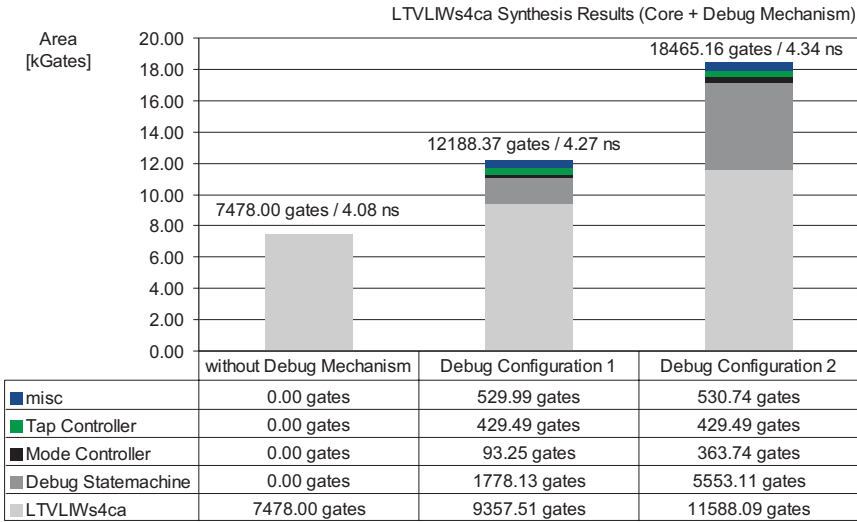


Figure A.8. Synthesis results of the LTVLIW32s4ca

reduce the area requirements to 68% and 82% respectively. The lower reduction in the case of the LTVLIW32s4ca architecture, is caused by the additional pipeline stage and therefore concurrent execution of operations which prevents area optimizations.

The synthesis results of the LTVLIW32s3ca and LTVLIW32s4ca with debug functionality is shown in figure A.7 and A.8. The apparently dramatic increase of area is (only) caused by the small size of the processor. The absolute values for the size of the debug state machine are even smaller compared to the LTRISC architecture. This is reasonable due to the fact, that both processor types contain the same number of general purpose registers, but differ with regard to their bit widths. The registers of the VLIW processors are only 16 bit wide, instead of 32 bit in the case of the LTRISC architecture.

Appendix B

CFG and DFG Conversions

The realized optimizations are more often applied to Data Flow Graphs (DFGs) than Control Flow Graphs (CFGs). However, CFGs are required to capture the procedural portions of the ADL model and to write out the hardware description on RTL. Therefore, conversions between both representations are required.

B.1 CFG to DFG Conversion

There are three basic differences between CFGs and DFGs causing the basic challenge for the translation algorithm:

- **Control statements** in CFGs are block oriented, in DFGs multiplexers are instantiated for each signal and variable written in a CFG block.
- **Variable dependencies** given by the sequential statements in CFGs have to be replaced by direct interconnections between operator outputs and inputs in DFGs.
- **Assignments to RTL signals** may occur several times in CFGs, in DFGs they have to be concentrated by the use of multiplexers into one physical write access.

Due to these differences, the translation has to imitate the sequential execution of a CFG in order to track the current assignments to variables and signals.

B.1.1 Block Based Translation

The particular values for variable assignments are stored in *translation contexts*. Such a context is created each time a conditional block is

entered, using the values assigned in the parent context for local initialization. A context is identified by the corresponding condition of a conditional statement. These conditions may be zero (false) and one (true) for IF/ELSE statements, arbitrary values are allowed for SWITCH statements. They are used when the contexts are recombined into the parent context. At this time multiplexers are inserted for each variable modified within one or more blocks, using the condition as multiplexer input selection criteria.

An example of the context usage is given in figure B.1. The parent context is given by X_p using the result of the addition ($C_{out,add}$) and multiplication ($C_{out,mul}$) in the first two lines of the example as current values for the variables a and b respectively. The analysis of the switch statement creates the contexts $X_{sw,i}$ for each case statement, the cases 1 and 2 are combined into a single context. The default context is given by the parent context implicitly. From the switch statement, two individual multiplexers are generated for the variables a and b , providing new values $C_{out,mux,a}$ and $C_{out,mux,b}$ for the updated parent context X'_p .

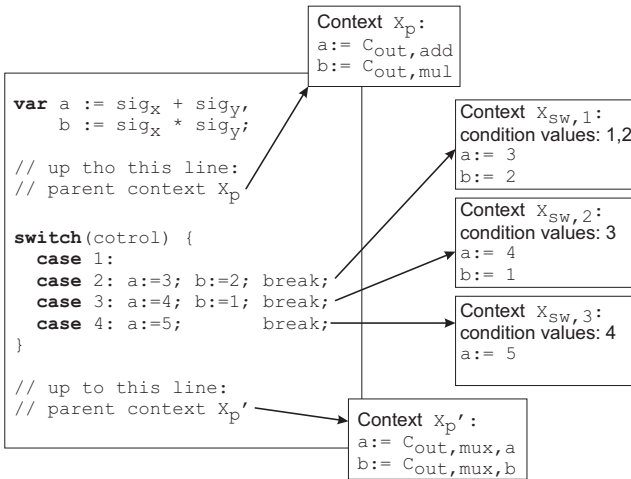


Figure B.1. Example of block based translation

B.1.2 Multiplexer Creation

Multiplexers are created whenever the translation contexts of conditional statements are combined into their parent context. For each variable v , the set CC_v of all pairs (cc, C_{cc}) of conditions cc and associated assignments C_{cc} is created. This also includes the default assignment derived from the parent context X_p . If there is no assignment in any parent

context, zero is assumed as default input. This forced initialization is necessary to avoid the insertion of latches in the gate-level synthesis. A multiplexer can be generated from CC_v for each variable v , its output $C_{out,mux,v}$ is inserted as current value of v into the translation context X_p .

The result of the multiplexer generation from the example given in figure B.1 is depicted in figure B.2. The assignments to a and b from the parent context X_p are used as default inputs for both multiplexers. Variable a is written in four contexts, variable b in three, resulting in multiplexers handling these four and three cases respectively.

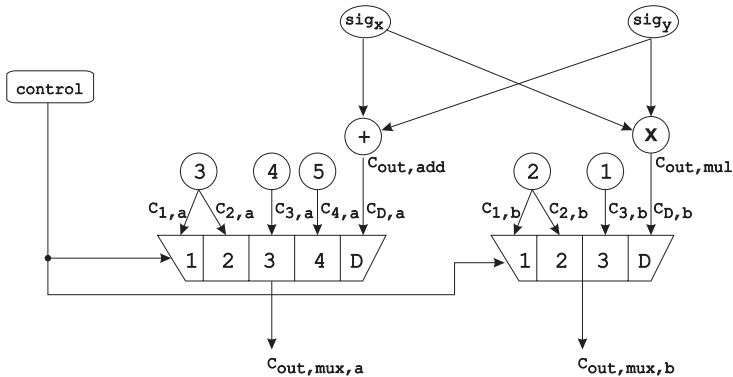


Figure B.2. Multiplexer generation from translation contexts

B.1.3 Signal Assignments

Signals require special handling in order to capture the correct simulator and hardware behavior. Assignments to signals are performed only after a Δ -cycle in hardware simulation. In order to reflect this behavior, variables are introduced in the CFG for the translation process. Every write access to a signal is replaced by a write access to the corresponding variable. At the end of the CFG the variable gets assigned to the signal. Because of this principle, the top level context automatically contains the correct signal value, including all necessary multiplexers within the DFG.

In the example given in figure B.3, the signal sig is written several times in the CFG. These multiple write accesses are tracked by the special variable var_{sig} . After the complete iteration through all statements, the final value of var_{sig} can be used to write signal sig (figure B.3).

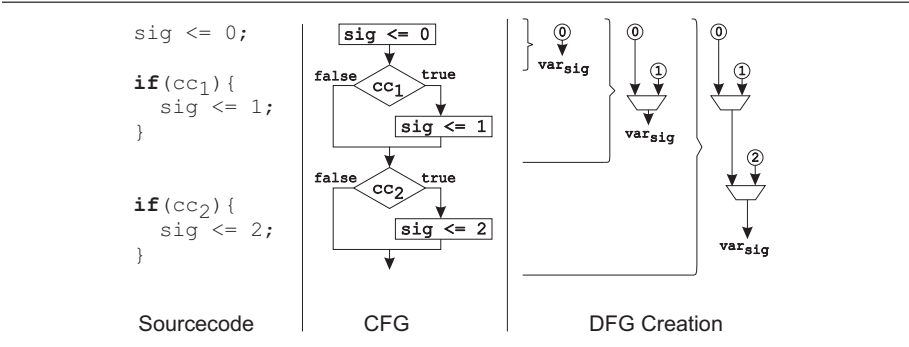


Figure B.3. Translation from CFG to DFG

B.1.4 Expression Translation

Expressions within the CFG are used on the left and right sides of assignments, within conditional expressions and for subscriptions. Expressions never span more than one CFG statement. A CFG expression is represented as an operator tree that already represents a DFG. This structure can be translated in a straightforward manner into the DFG without the need for a special handling. The current context is used in order to translate the variable assignments.

B.2 DFG to CFG Conversion

In the scope of this book, DFGs, which are directed acyclic graphs with inherent concurrency, are used for optimizations and transformations. However, this representation is not suitable for the generation of an RTL hardware model in, for example, VHDL or Verilog. Therefore, it is necessary to translate each DFG back into a CFG. The required sequential ordering is given by the data flow dependencies. The order of concurrent and therefore independent data flows does not matter regarding CFG descriptions. Basically, for each vertex of the DFG a single CFG statement is derived. This CFG statement is an assignment to a variable which can be used as operand in dependent CFG statements.

Only directed acyclic graphs can be converted into CFGs - it is basically not possible to generate a CFG from combinatorial loops. In the applied algorithm, the ordering of statements is derived from the DFG by topological sorting. The order (ord) of an operator is given by one plus the maximum order of its parent operators or zero if there is no parent operator. This rule has to be applied during a depth first search through the DFG, starting at the vertices without any output. An example of topological sorting is given in figure B.4.

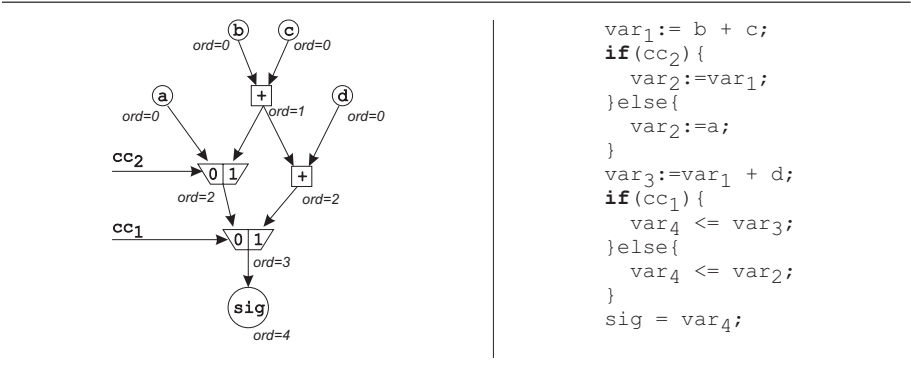


Figure B.4. Translation from DFG to CFG

After the ordering process, the translation into the CFG starts at the vertex with the order zero and proceeds with all vertices of the next higher order until all nodes are translated. Functional operators such as bit operators, comparators and arithmetic operators can be translated in a straightforward way. Multiplexers are translated into *IF/ELSE* statements or *SWITCH* statements. *IF/ELSE* statements are used if only one condition (and its boolean negation) is evaluated. *SWITCH* statements are used in all other cases. Because of the topological sorting, nested conditions are translated to sequential conditional statements and corresponding variable dependencies. An example of the translation from a DFG back to a CFG is given in figure B.4 as well.

Appendix C

Debug Mechanism

In this appendix further information about possible debug mechanism features, the Nexus standard and debug mechanisms in industry is provided.

C.1 Advanced Features of Debug Mechanisms

The features presented in this section do not satisfy the requirements of a *configurable* processor feature as discussed in chapter 8. They are either strongly interweaved with the processor core or demand a high i/o data transfer. In both cases these features significantly influence the processor structure. Therefore, these features should be considered during design space exploration and *not* automatically integrated into the processor. To complete the picture, several advanced features of debug mechanisms are presented in the following.

Watchpoints: Watchpoints are equal to hardware breakpoints but generate an event message instead of switching to debug mode.

Ownership trace: Ownership trace provides a macroscopic view to the executed software, such as for example, to the task flow. It is used, for example, to trace the execution of the operating system.

Program trace: Program trace allows the exact reconstruction of a program execution. As this trace is bound to the program counter, which usually changes every clock cycle, the amount of data to be transmitted through the debug interface is probably very high. Thus, special protocols [125] might be applied to reduce the amount of data.

Data trace: This feature enables the designer to trace accesses to a set of registers or memories. Messages are sent whenever a register or memory of the set is accessed.

Access to storage elements during user mode: Processors dedicated for real-time processing are required to allow debug access to storage elements while being in user mode. This debug functionality is highly complex to implement, as debug accesses potentially conflict with regular accesses. The required logic to control these conflicts influences the timing of the processor.

Memory substitution: This feature provides the opportunity to redirect memory accesses to the debug interface.

The different features of a debug mechanism are categorized by the NEXUS standard, as described in the following.

C.1.1 The Nexus Standard

In 1999, the Nexus 5001 Forum released a standard for an embedded processor debug interface, called Nexus [150]. Members of the forum are e.g. Infineon Technologies [151], Motorola [152] and STMicroelectronics [153]. Nexus compliant debug interfaces are divided into four classes. The standard specifies supported features for each class, where class 1 compliant devices implement the least and class 4 compliant devices the most features. The standard also specifies, for example, the download and upload rate of the debug port and whether it is full-duplex or half-duplex communication.

Table C.1 shows the required features for the four Nexus classes. Some optional features are not listed here. These can be found in [150].

The different classes of the Nexus standard require different interfaces to enable the appropriate data transfer with the processor core. The class 1 interfaces use a JTAG port, thus the access is half-duplex only. Interfaces of class 2, 3 and 4 use additional pins for input and output (full-duplex). They implement a Nexus pin interface, called auxiliary port (AUX). Due to a parallelization of data transmission and a higher clock frequency, the scalable auxiliary interface is capable of higher data rates than the JTAG port. The number of pins for data transmission vary from one to sixteen, depending on the class of the interface and the required throughput. Either a JTAG or an AUX port may be used depending on the required input data rate for classes 2, 3 and 4.

The Nexus standard also comprises an Application Programming Interface (API), which has to be used to access the hardware debug interface. It consists of two layers: The Target Abstraction Layer (TAL) and the Hardware Abstraction Layer (HAL). The implementation of these layers must be conform to the C/C++ header files provided by the Nexus standard. The TAL provides an implementation of the Nexus debug semantics and utilizes the HAL in order to communicate with the

Table C.1. Required features for Nexus compliant debug mechanisms

Class			Feature
1	2	3	reading and writing core registers in debug mode
			reading and writing memories in debug mode
			entering debug mode from reset
			entering debug mode from user mode
			exiting debug mode, return to user mode
	4	executing single step instruction in debug mode	
		entering debug mode from instruction/data break-point	
		setting breakpoints and watchpoints	
		device identification	
		sending an event message when a watchpoint matches	
			real-time monitoring: process ownership
			real-time monitoring: program flow
			real-time monitoring: data write
			real-time memory access
			memory substitution
			program or data trace due to watchpoint occurrence

processors’s debug mechanism through the available interface. Further information on this topic can be found in [150].

C.1.2 Debug Mechanisms and Interfaces in Industry

Unfortunately, there is no commonly established debug mechanism in industry. Most companies integrate a debug mechanism into their processors according to their own proprietary specification. However, commonly the JTAG interface is used to access the debug mechanism. Often additional pins are added to extend the capabilities of the JTAG interface.

For example, Texas Instruments [154], ARM [155] and Intel [156] use a JTAG interface to access the debug interface. The debug mechanism of the ARM7TDMI [157] can be accessed via a JTAG interface as well. However, the debug mechanism of the ARM7TDMI architecture is highly specialized and utilizes at least four additional pins. The ARM7TDMI debug mechanism contains a special macrocell, which is called EmbeddedICE Macrocell, to control two break- or watchpoints and the state of the debug mechanism. It contains a test data register,

which can be accessed via the JTAG interface. The data bus of the processor is also accessible through the scan chain of the JTAG interface. Further information on the ARM7TDMI debug mechanism can be found in [157].

Intel's Pentium processor family is equipped with a debug mode, known as *probe mode*. The probe mode is accessible via the JTAG interface as well as additional pins [158].

The C166 microcontroller from Infineon Technologies uses a Nexus class 1 debug interface. Both, the Super10 core from STMicroelectronics and the Motorola PowerPC MPC 56x series provide a Nexus class 3 debug interface. The MPC 56x series contains a Background Debug Mode (BDM) interface. BDM is a proprietary standard developed by Motorola and is used for almost the complete PowerPC family.

List of Figures

1.1	Comparison of ASIC and ASIP verification effort	3
1.2	Computational building blocks in SoCs. Source: T. Noll, EECS [14]	4
1.3	Heterogeneous architecture types as SoC building blocks	5
1.4	ASIP design phases	8
3.1	Exploration and implementation based on LISA	24
3.2	The LISA simulator and debugger frontend	26
4.1	Register accesses with and without optimizations	33
4.2	ADL to RTL synthesis flow based on an UDL	35
4.3	Exclusiveness in architecture descriptions and hardware descriptions	36
4.4	Example of decode logic for a degenerated encoding specification	38
5.1	Combining information: LISA operation graph	44
5.2	LISA coding graph	49
5.3	LISA behavior sections	52
5.4	CFG representation of LISA behavior sections	53
5.5	LISA activation graph and the spatial and temporal relation	55
5.6	Pseudo hardware implementation of the LISA activation graph	56
5.7	Compatibility graph and conflict graph	57
5.8	Example of merging compatible conflict graphs	58
5.9	Example of merging conflicting conflict graphs	59

5.10	Conflict graph creation for LISA operations	62
5.11	Conflict graph creation for activation elements	63
5.12	Global conflict graph creation	64
5.13	Example of explicit relations between conditional blocks	65
6.1	Hardware description on RTL, based on entities, processes and signals	68
6.2	Different views to the same architecture model	69
6.3	Example UDL of an ASIP	71
6.4	Synthesis steps from LISA to RTL	72
6.5	Visualizing the UDL in the synthesis GUI	74
7.1	Constraint-independent and constraint-dependent optimizations	78
7.2	Unoptimized and optimized DFG for ICORE's WBITI instruction [91]	80
7.3	Multiplexer chain simplification	83
7.4	Sharing of non-commutative operators	88
7.5	Sharing with insertion of identity elements	88
7.6	Sharing of commutative operators	89
7.7	Suboptimal sharing of commutative operators	90
7.8	Input conflict graphs	91
7.9	Clique covering and graph coloring	92
7.10	Sharing nodes with output similarity	94
7.11	Worse timing by sharing operators with widely differing timing	94
7.12	Default condition check implementation	96
7.13	Unnecessary dependencies in default condition check implementation	97
7.14	Improved condition check implementation	97
7.15	Decision minimization	99
8.1	Debug configuration dialog	103
8.2	Structure of the TAP	105
8.3	Setup and hold time for a D-flip-flop	106
8.4	The Flancter Circuit	108
8.5	Data transmission across clock domain boundaries	108
8.6	The synthesized RTL hardware model structure	110

8.7	State transition diagram of the debug state machine	111
8.8	Mode controller of the debug mechanism	112
8.9	Register implementation	113
8.10	Memory implementation	114
9.1	Turbo decoder structure. Source: F. Munsche [129], modified	121
9.2	Trellis butterfly structure	121
9.3	Schedule of operations, Source: F. Munsche [129], modified	123
9.4	Addressing schemes derived from the turbo decoding algorithm	125
9.5	Supported addressing schemes by the address registers	126
9.6	Prologue and epilogue in Turbo decoding	129
9.7	Prologue and epilogue handling	130
9.8	Pipeline organization	132
10.1	Levels of compatibility	136
10.2	Instruction bit width in M68HC11 and compatible LISA model	138
10.3	The compatible 68HC11 architecture designed with LISA	142
10.4	Synthesis results of the ISS-68HC11 architecture	142
10.5	Synthesis results of the ISS-68HC11 architecture	143
10.6	Assembly program and execution schedule. Source: [92], modified	145
10.7	A structural view of the Infineon ASMD. Source: [92], modified	146
10.8	A structural view of new ASMD architecture. Source: [149], modified	147
10.9	Synthesis results of the ASMD architecture. Based on: [88]	148
A.1	Synthesis results of the ICORE	155
A.2	Synthesis results of the ICORE	155
A.3	Synthesis results of the LTRISC32ca architecture	156
A.4	Synthesis results of the LTRISC32ca	157
A.5	Synthesis results of the LTVLIW32s3ca architecture	157

A.6	Synthesis results of the LTVLIW32s4ca	158
A.7	Synthesis results of the LTVLIW32s3ca	158
A.8	Synthesis results of the LTVLIW32s4ca	159
B.1	Example of block based translation	162
B.2	Multiplexer generation from translation contexts	163
B.3	Translation from CFG to DFG	164
B.4	Translation from DFG to CFG	165

List of Tables

5.1	List of LISA sections	44
6.1	Exhaustive list of entities, units and paths	70
7.1	Effect of basic optimizations on graph vertex count for the ICORE model	79
7.2	Simplifications by constant folding	81
7.3	Simplification for multiplexers with boolean control and data inputs	82
7.4	Models for a basic abstract timing approximation	86
7.5	Timing approximation vs. synthesis results for the ICORE architecture	86
7.6	List of sharable operations and their implementation	95
8.1	UDL elements for debug mechanism and JTAG interface generation	115
9.1	Turbo decoding implementations based on programmable solutions	119
9.2	Memory organization for Turbo decoding	124
9.3	Memory utilization by operations (R=Read/W=Write)	124
9.4	Architecture size of the Turbo decoding architecture	133
10.1	Extract of the Motorola 68HC11 instruction-set manual [146]	139
C.1	Required features for Nexus compliant debug mechanisms	169

References

- [1] K. Keutzer, S. Malik, and A.R. Newton. From ASIC to ASIP: The Next Design Discontinuity. In *Proc. of the Int. Conference on Computer Design (ICCD)*, Freiburg (Germany), September 2002.
- [2] International Technology Roadmap for Semiconductors. Overview and Summaries. <http://public.itrs.net>.
- [3] J. Henkel. Closing the SoC Design Gap. In *Computer. IEEE Computer Society Press*, September 2003.
- [4] C. Christensen. *Disruption in RTL Design for SoCs: A New Methodology*. Tensilica, 2004.
- [5] H. Meyr. System-on-Chip for Communications: The Dawn of ASIPs and the Dusk of ASICs (Keynote Speech). In *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS)*, Seoul (Korea), August 2003.
- [6] T. Glökler, S. Bitterlich, and H. Meyr. ICORE: A Low-Power Application Specific Instruction Set Processor for DVB-T Acquisition and Tracking. In *IEEE Workshop on Signal Processing Systems (ASIC/SOC)*, Washington DC, September 2000.
- [7] T. Glökler, S. Bitterlich, and H. Meyr. Increasing the Power Efficiency of Application Specific Instruction Set Processors using Datapath Optimization. In *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS)*, Lafayette (LA), October 2000.
- [8] P. Magarshack and P.G. Paulin. System-on-Chip beyond the nanometer wall. In *Proc. of the Design Automation Conference (DAC)*, Anaheim (CA), June 2003.
- [9] C. Rowen. *Engineering the Complex SoC*. Prentice Hall, 2004.
- [10] O. Schliebusch, G. Ascheid, A. Wieferink, R. Leupers, and H. Meyr. Application Specific Processors for Flexible Receivers. In *Proc. of National Symposium of Radio Science (URSI)*, Poznan (Poland), April 2005.

- [11] H. Meyr, O. Schliebusch, A. Wieferink, D. Kammler, E.M. Witte, O. Luethje, M. Hohenauer, G. Braun, and A. Chattopadhyay. Designing and Modelling MPSoC Processors and Communication Architectures. In *Building ASIPs: The Mescal Methodology*, Matthias Gries and Kurt Keutzer (editors), 2005. Springer.
- [12] H. Blume, H.T. Feldkämper, and T.G. Noll. Model-based Exploration of the Design Space for Heterogeneous Systems-on-Chip. In *Journal of VLSI Signal Processing*, volume 40 (No. 1). Springer Science+Business Media, Mai 2005.
- [13] H. Blume, H. Hübert, H.T. Feldkämper, and T.G. Noll. Model-based Exploration of the Design Space for Heterogeneous Systems on Chip. In *Proc. of the Int. Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, San Jose (CA), July 2002.
- [14] Chair of Electrical Engineering and Computer Systems (EECS), RWTH Aachen University, Germany. <http://www.eecs.rwth-aachen.de>.
- [15] Inc. Express Logic. *ThreadX, Use Guide*. Express Logic, Inc., 2003.
- [16] P. Guerrier and A. Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris (France), March 2000.
- [17] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.
- [18] D.A. Patterson and J.L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann Publishers, Inc., 1997.
- [19] M. Freericks. The nML Machine Description Formalism. Technical Report, Technical University of Berlin, Department of Computer Science, 1993.
- [20] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Anaheim (CA), June 1997.
- [21] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. *The MIMOLA Language, Version 4.1. Reference Manual*. Department of Computer Science 12, Embedded System Design and Didactics of Computer Science, 1994.
- [22] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, March 1999.
- [23] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.
- [24] M. Itoh, S. Higaki, J. Sato, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: An ASIP Design Environment. In *Proceedings of the 2000 IEEE International Conference on Computer Design (ICCD)*, Austin (TX), September 2000.

- [25] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EX-PRESSION: An ADL for System Level Design Exploration. Technical report, Department of Information and Computer Science, University of California, Irvine, September 1998.
- [26] P. Mishra, A. Kejariwal, and N. Dutt. Rapid Exploration of Pipelined Processors through Automatic Generation of Synthesizable RTL Models. In *Proc. of the IEEE Int. Workshop on Rapid System Prototyping*, San Diego (CA), June 2003.
- [27] A. Kejariwal P. Mishra and N. Dutt. Synthesis-driven Exploration of Pipelined Embedded Processors. In *Int. Conference on VLSI Design*, Mumbai (India), January 2004.
- [28] P. Paulin, C. Liem, T. May, and S. Sutarwala. FlexWare: A Flexible Firmware Development Environment for Embedded Systems. In *Code Generation for Embedded Processors*, P. Marwedel and G. Goosens (editors), 1995. Kluwer Academic Publishers.
- [29] P. Paulin. Towards Application-Specific Architecture Platforms: Embedded Systems Design Automation Technologies. In *Proc. of the Euromicro*, April 1995.
- [30] P. Paulin. Design Automation Technology Challenges for Application-Specific Architecture Platforms. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPE5)*, St. Goar (Germany), March 2001.
- [31] G. Hadjiyiannis, P. Russo, and S. Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Proc. of the Design Automation Conference (DAC)*, New Orleans (LA), June 1999.
- [32] G. Hadjiyiannis. *An Architecture Synthesis System for Embedded Processors (PhD thesis)*. Massachusetts Institute of Technology, June 2000.
- [33] Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany. <http://www.iss.rwth-aachen.de>.
- [34] CoWare Inc. <http://www.coware.com>.
- [35] J.-H. Yang et al. MetaCore: An Application Specific DSP Development System. In *Proc. of the Design Automation Conference (DAC)*, San Francisco (CA), June 1998.
- [36] J.-H. Yang et al. MetaCore: An Application-Specific Programmable DSP Development System. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, April 2000.
- [37] R. Leupers and P. Marwedel. Retargetable Code Compilation Based on Structural Processor Descriptions. In *ACM Transactions on Design Automation for Embedded Systems*, volume 3, no. 1. Kluwer Academic Publishers, January 1998.
- [38] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

- [39] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, March 1995.
- [40] Interuniversity Micro Electronics Center (IMEC). <http://www.imec.be>.
- [41] Target Compiler Technologies. <http://www.retarget.com>.
- [42] W. Geurts, G. Goossens, D. Lanneer, and J. Van Praet. Design of Application-Specific Instruction-Set Processors for Multi-Media, using a Retargetable Compilation Flow. In *Proceedings of the GSPx*, Santa Clara (CA), October 2005.
- [43] IIT Kanpur. <http://www.cse.iitk.ac.in/sim-nml/>.
- [44] V. Rajesh and R. Moona. Processor Modeling for Hardware Software Codesign. In *Int. Conference on VLSI Design*, Goa (India), January 1999.
- [45] J. Sato, A. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai. PEAS-I: A Hardware/Software Codesign System for ASIP Development. In *IEICE Transaction on Fundamentals of Electronics*, March 1994.
- [46] A. Kitajima, M. Itoh, J. Sato, and A. Shiomi. Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined Processors. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, Yokohama (Japan), January 2001.
- [47] ASIP Meister, The PEAS Project. <http://www.eda-meister.org/asip-meister/>.
- [48] S. Weber, M.W. Moskewicz, M. Gries, C. Sauer, and K. Keutzer. Fast Cycle-Accurate Simulation and Instruction-Set Generation for Constraint-Based Descriptions of Programmable Architectures. In *Proc. of the Int. Workshop on Hardware/Software Codesign (CODES)*, Stockholm (Sweden), September 2004.
- [49] M. Gries, S. Weber, and C. Brooks. The Mescal Architecture Development System (Tipi) Tutorial. October 2003.
- [50] Center For Electronic System Design.
The Mescal Webpage. <http://embedded.eecs.berkeley.edu/mescal/>.
- [51] ARC International. <http://www.arc.com>.
- [52] Tensilica Inc. <http://www.tensilica.com>.
- [53] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/Software Instruction Set Configurability for System-on-Chip Processors. In *Proc. of the Design Automation Conference (DAC)*, Las Vegas (NV), June 2001.
- [54] Stretch Inc. <http://www.stretchinc.com>.
- [55] Embedded Microprocessor Benchmark Consortium. <http://www.eembc.com>.
- [56] IEEE Standards Association. <http://standards.ieee.org>.

- [57] C. Meersman. The VHDL Standard. In *European Space Agency Contract Report*, May 1994.
- [58] Cadence. <http://www.cadence.com>.
- [59] The Verilog Website. <http://www.verilog.com>.
- [60] The SystemVerilog Website. <http://www.systemverilog.org>.
- [61] The SystemC Website. <http://www.systemc.org>.
- [62] C.Y. Lee. Representation of switching circuits by binary-decision programs. Technical report, Bell System Technical Journal, July 1959.
- [63] S.B. Akers. Binary Decision Diagrams. In *IEEE Transactions on Computers*, June 1978.
- [64] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, August 1986.
- [65] O. Schliebusch and H. Meyr. ASIP Development Using LISA 2.0. In *Design of Energy-Efficient Application-Specific Instruction Set Processors*, T. Glökler and H. Meyr, 2004. Kluwer Academic Publishers.
- [66] A. Nohl, V. Greive, G. Braun, A. Hoffmann, R. Leupers, O. Schliebusch, and H. Meyr. Instruction Encoding Synthesis for Architecture Exploration Using Hierarchical Processor Models. In *Proc. of the Design Automation Conference (DAC)*, Anaheim (CA), June 2003.
- [67] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris (France), February 2004.
- [68] O. Wahlen, T. Glökler, A. Nohl, A. Hoffmann, R. Leupers, and H. Meyr. Application Specific Compiler/Architecture Codesign: A Case Study. In *Proc. of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) and Software and Compilers for Embedded Systems (SCOPES)*, Berlin (Germany), June 2002.
- [69] ACE Associated Compiler Experts bv. <http://www.ace.nl/>.
- [70] O. Wahlen, M. Hohenauer, G. Braun, R. Leupers, G. Ascheid, H. Meyr, and X. Nie. Extraction of Efficient Instruction Schedulers from Cycle-True Processor Models. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Vienna (Austria), September 2003.
- [71] O. Wahlen, M. Hohenauer, R. Leupers, and H. Meyr. Using Virtual Resources for Generating Instruction Schedulers. In *Proc. of IEEE/ACM International Workshop on Application Specific Processors (WASP'02)*, Istanbul (Turkey), November 2002.

- [72] O. Wahlen. *C Compiler Aided Design of Application-Specific Instruction-Set Processors Using the Machine Description Language LISA*. Shaker Verlag, 2004.
- [73] A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Generating Production Quality Software Development Tools Using A Machine Description Language. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Munich (Germany), March 2001.
- [74] Tool Interface Standard Committee (TIS), now SCO group. *ELF: Executable and Linkable Format*.
- [75] CoWare Inc. *LISATek Software Development Tools Manual - 2005.1.0*.
- [76] G. Braun, A. Wiefierink, O. Schliebusch, R. Leupers, H. Meyr, and A. Nohl. Processor/Memory Co-Exploration on Multiple Abstraction Levels. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Munich (Germany), March 2003.
- [77] G. Braun, A. Hoffmann, A. Nohl, and H. Meyr. Using Static Scheduling Techniques for Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, Montreal (Canada), October 2001.
- [78] G. Braun, A. Nohl, O. Schliebusch, A. Hoffmann, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *IEEE Transactions on Computer-Aided Design*, December 2004.
- [79] A. Nohl, G. Braun, A. Hoffmann, O. Schliebusch, H. Meyr, and R. Leupers. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proc. of the Design Automation Conference (DAC)*, New Orleans (LA), June 2002.
- [80] S. Pees, A. Hoffmann, and H. Meyr. Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language. In *IEEE Transactions on Design Automation of Electronic Systems*, October 2000.
- [81] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris (France), March 2000.
- [82] Synopsys. System Studio.
http://www.synopsys.com/products/cocentric_studio/.
- [83] A. Wiefierink, T. Kogel, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr. A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris (France), February 2004.
- [84] A. Wiefierink, T. Kogel, A. Nohl, A. Hoffmann, R. Leupers, and H. Meyr. A Generic Toolset for SoC Multiprocessor Debugging and Synchronisation. In

- Proc. of the Int. Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, The Hague (Netherlands), June 2003.
- [85] Synopsys. Design Compiler.
http://www.synopsys.com/products/logic/design_compiler.html.
- [86] Cadence. First Encounter.
http://www.cadence.com/products/digital_ic/first_encounter/index.aspx.
- [87] U. Brandt and O. Schliebusch. *VHDL Implementation of the Infineon PP32 Protocol Processor based on LISA*. Semester Work, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2002.
- [88] O. Schliebusch, A. Chattopadhyay, M. Steinert, G. Braun, A. Nohl, R. Leupers, G. Ascheid, and H. Meyr. RTL Processor Synthesis for Architecture Exploration and Implementation. In *Proc. of the Conference on Design, Automation & Test in Europe - Designer's Forum (DATE)*, Paris (France), February 2004.
- [89] F. Fiedler and O. Schliebusch. *SystemC Generation from LISA*. Diploma Thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2001.
- [90] M. Vaupel. *Effizienter Entwurf eines DVB-Satelliten-Empfängers*. Shaker Verlag, 1999.
- [91] T. Glökler and H. Meyr. *Design of Energy-Efficient Application-Specific Instruction Set Processors*. Kluwer Academic Publishers, 2004.
- [92] M. Steinert, O. Schliebusch, and O. Zerres. Design Flow for Processor Development using SystemC. In *Proc. of SNUG Europe*, Munich (Germany), March 2003.
- [93] T. Kuo and O. Schliebusch. *Refining a Cycle-Accurate LISA Model of the MIPS32 4k Processor for VHDL Implementation*. Semester Work, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2003.
- [94] H. Scharwaechter, D. Kammler, A. Wiefierink, M. Hohenauer, J. Zeng, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient IPsec Encryption: A Case Study. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Amsterdam (Netherlands), September 2004.
- [95] M. Rixius and D. Kammler. *Implementation of an RISC Processor Core with Advanced Memory System Using the RTL-Processor-Synthesis*. Diploma Thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2004.
- [96] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. Electrical and Computer Engineering Series. McGraw-Hill, Inc., 1994.
- [97] A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr. A Methodology for the Design of Application Specific Instruction-Set Processors Using the Machine Description Language LISA. In *Proc. of the Int. Conference on Computer Aided Design (ICCAD)*, San Jose (CA), November 2001.

- [98] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, A. Wiefierink, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. In *IEEE Transactions on Computer-Aided Design*, November 2001.
- [99] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture Implementation Using the Machine Description Language LISA. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, Bangalore (India), January 2002.
- [100] CoWare Inc. *LISA Language Reference Manual 2005.1.0*.
- [101] Ottman and Widmayer. *Algorithmen und Datenstrukturen*. BI-Wissenschaftsverlag, 1993.
- [102] H. Andersen and H. Hulgaard. Boolean Expression Diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, Warsaw (Poland), June 1997.
- [103] H. Hulgaard, P. Williams, and H. Andersen. Equivalence Checking of Combinational Circuits Using Boolean Expression Diagrams. In *IEEE Transactions on Computer-Aided Design*, July 1999.
- [104] E.M. Witte and O. Schliebusch. *Analysis and Implementation of Resource Sharing Optimizations for RTL-Processor-Synthesis*. Diploma Thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2004.
- [105] O. Schliebusch, A. Chattopadhyay, D. Kammler, R. Leupers, G. Ascheid, H. Meyr, and T. Kogel. A Framework for Automated and Optimized ASIP Implementation Supporting Multiple Hardware Description Languages. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, Shanghai (China), January 2005.
- [106] D. Kammler and O. Schliebusch. *Implementation of SystemC Based Functional Unit Generation*. Semester Work, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2003.
- [107] Kernighan and Ritchie. *Programmieren in C*. PC professional, 1990.
- [108] B. Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 2000.
- [109] S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 1997.
- [110] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 1995.
- [111] E. Gamma, R. Helm, and R.E. Johnson. *Design Patterns*. Addison-Wesley Professional, 1997.
- [112] O. Schliebusch, A. Chattopadhyay, E.M. Witte, D. Kammler, G. Ascheid, R. Leupers, and H. Meyr. Optimization Techniques for ADL-driven RTL Processor Synthesis. In *Proc. of the International Workshop on Rapid System Prototyping (RSP)*, Montreal (Canada), June 2005.

- [113] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
- [114] D. Pearson and V. Vazirani. Efficient Sequential and Parallel Algorithms for Maximal Bipartite Sets. In *Journal of Algorithms*, March 1993.
- [115] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs. In *Proc. of the Design Automation Conference (DAC)*, San Diego (CA), June 2004.
- [116] T. Emden-Weinert, S. Hougardy, B. Kreuter, H.J. Prömel, and A. Steger. *Einführung in Graphen und Algorithmen*. Humboldt-Universität zu Berlin, Lehrstuhl für Algorithmen und Komplexität, 1996.
- [117] J.R. Brown. Chromatic Scheduling and the Chromatic Number Problem. In *Management Science*, April 1972.
- [118] F.T. Leighton. A Graph Coloring Algorithm for Large Scheduling Problems. In *Journal of Research of the National Bureau of Standards*, 1979.
- [119] L. Stok. False Loops through Resource Sharing. In *Proc. of the Int. Conference on Computer Aided Design (ICCAD)*, Santa Clara (CA), November 1992.
- [120] M. Nourani and C. Papachristou. Avoiding False Paths Caused by Resource Binding in RTL Delay Analysis. In *Proc. of the IEEE Int. Symposium on Circuits and Systems (ISCAS)*, Atlanta (GA), May 1996.
- [121] A. Su, Y.-C. Hsu, T.-Y. Liu, and M. T.-C. Lee. Eliminating False Loops Caused by Sharing in Control Path. In *IEEE Transactions on Design Automation for Embedded Systems*, July 1998.
- [122] S. Raje and R.A. Bergamaschi. Generalized resource sharing. In *Proc. of the Int. Conference on Computer Aided Design (ICCAD)*, San Jose (CA), November 1997.
- [123] S. Bhattacharya, S. Dey, and F. Breglez. Effects of resource sharing on circuit delay: An assignment algorithm for clock period optimization. In *IEEE Transactions on Design Automation for Embedded Systems*, April 1998.
- [124] O. Schliebusch, D. Kammler, A. Chattopadhyay, R. Leupers, G. Ascheid, and H. Meyr. Automatic JTAG Interface and Debug Mechanism Generation for ASIP Design. In *Proceedings of the GSPx*, Santa Clara (CA), September 2004.
- [125] IEEE, Inc., 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Std 1149.1-2001, Standard Test Access Port and Boundary-Scan Architecture*, June 2001.
- [126] E. Haseloff. *Metastable Response in 5-V Logic Circuits*. Texas Instruments, <http://focus.ti.com/lit/sdya006/sdya006.pdf>, February 1997.
- [127] R. Weinstein. The Flancter. In *Xcell online*, <http://www.xilinx.com/xcell/>, July 2000. XILINX and Memec Design Services.
- [128] D. Kammler and O. Schliebusch. *Design of a Hardware Debug Interface for Processors Described with LISA*. Diploma Thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2003.

- [129] F. Munsche. *Flexible VLSI Architectures for the Iterative Decoding of Parallel Concatenated Convolutional Codes*. Shaker Verlag, 2004.
- [130] B. Bauwens and O. Schliebusch. *Processor Development for Iterative Decoding of Parallel Concatenated Convolutional Codes*. Semester Work, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2003.
- [131] P. Salz and O. Schliebusch. *Specification and Implementation of an Application Specific Instruction Set Processor (ASIP) for Turbo Decoding*. Diploma Thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, 2004.
- [132] F. Kienle, H. Michel, F. Gilbert, and N. Wehn. Efficient MAP-algorithm Implmentation on Programmable Architectures. In *Advances in Radio Science*, November 2003.
- [133] J.G. Harrison. Implementation of a 3GPP Turbo Decoder on a Programmable DSP Core. In *Proc. of the Communications Design Conference*, San Jose (CA), October 2001.
- [134] J. Nikolic-Popovic. Implementing a MAP Decoder for CDMA2000 Turbo Codes on a TMS320C62x DSP device. In *Texas Instruments Application Report (SPRA629)*, May 2000.
- [135] M.C. Valenti and J. Sun. The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software Defined Radios. In *International Journal of Wireless Information Networks*, Oktober 2001.
- [136] H. Blume, T. Gemmeke, and T.G. Noll. Platform Specific Turbo Decoder Implementations. In *DSP Design Workshop*, Dresden (Germany), January 2003.
- [137] A. La Rosa, C. Passerone, F. Gregoretti, and L. Lavagno. Implementation of a UMTS Turbo-Decoder on a Dynamically Reconfigurable Platform. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris (France), February 2004.
- [138] A. La Rosa, L. Lavagno, and C. Passerone. Implementation of a UMTS Turbo-Decoder on a Dynamically Reconfigurable Platform. In *IEEE Transactions on Computer-Aided Design*, January 2005.
- [139] H. Michel, A. Worm, N. Wehn, and M. Münch. Hardware/Software Trade-Offs for Advanced 3G Channel Coding. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Paris (France), March 2002.
- [140] H. Michel and N. Wehn. *Implementation of Turbo-Decoders on Programmable Architectures*. Universität Kaiserslautern, 2002.
- [141] F. Gilbert, M.J. Thul, and N. Wehn. Communication Centric Architectures for Turbo-Decoding on Embedded Multiprocessors. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Munich (Germany), March 2003.

- [142] F. Gilbert, F. Kienle, G. Kreiselmaier, M.J. Thul, T. Vogt, N. Wehn, and F. Berens. Advanced Architectures For High-Throughput Turbo-Decoders. In *ST Journal of System Research*, February 2004.
- [143] J. Vogt, K. Koora, A. Finger, and G. Fettweis. Comparison of Different Turbo Decoder Realizations for IMT-2000. In *Proc. of the Global Telecommunications Conference (GLOBECOM)*, Rio de Janeiro (Brasil), December 1999.
- [144] Virtual Silicon Technology. <http://www.virtual-silicon.com>.
- [145] United Microelectronics Corp. <http://www.umc.com>.
- [146] Motorola. *M68HC11E Series Programming Reference Guide*, June 2002.
- [147] Synopsys. DesignWare.
<http://www.synopsys.com/products/designware/>.
- [148] Synopsys. DesignWare Library, 6811 MacroCell, Data Sheet.
http://www.synopsys.com/products/designware/docs/ds/i/DW_6811.pdf.
- [149] S. Buch. Application Specific Processors in Industry SoC Designs, "http://tima.imag.fr/mpsoc/2004/slides/buch.pdf". In *Int. Seminar on Application-Specific Multi-Processor SoC (MPSoC)*, TIMA Laboratory (France), July 2004.
- [150] IEEE-ISTO. *The Nexus 5001 Forum, Standard for a Global Embedded Processor Debug Interface*, December 1999.
- [151] Infineon Technologies AG. <http://www.infineon.com>.
- [152] Motorola Inc. <http://www.motorola.com>.
- [153] ST Microelectronics Inc. <http://www.st.com>.
- [154] Texas Instruments Inc. <http://www.ti.com>.
- [155] ARM Limited. <http://www.arm.com>.
- [156] Intel Corporation. <http://www.intel.com>.
- [157] Advanced RISC Machines Ltd (ARM). *Application Note 28, The ARM7TDMI Debug Architecture*, December 1995.
- [158] Intel Corporation. *Pentium Processor Family User's Manual*, 1994.

About the Authors

Oliver Schliebusch received the Diploma degree and the PhD degree with honors in Electrical Engineering from RWTH Aachen University, Germany, in 2000 and 2006.

During his research activities at the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, he was one of the key architects of the Language for Instruction Set Architectures (LISA) and the corresponding ASIP design methodology. He was leading the research activities on RTL hardware model generation from LISA, enabling a major paradigm shift in ASIP implementation. In January 2004 he has been appointed as chief-engineer at ISS.

Oliver Schliebusch successfully transferred the research results of automated ASIP implementation to industry and accompanied their proliferous usage. In this scope he developed several commercial processor designs. Oliver Schliebusch published journal articles and papers at international conferences, such as at DAC, DATE and VLSI-Design. At DAC 2002 he received the best paper award. Patents about his research results are pending. Furthermore, he presented his results in numerous seminars and workshops to industry and academic institutions.

Since beginning of 2006 he is software engineering manager at CoWare Inc. There he continues his work on processor design tools in the emerging field of Electronic System Level (ESL) design.

Contact Information:

Oliver Schliebusch, CoWare Inc.
Technologiezentrum am Europaplatz
Dennewartstrasse 25-27
52068 Aachen, Germany

E-Mail: oliver@coware.com

Heinrich Meyr received his M.Sc. and PhD from ETH Zurich, Switzerland. He spent over 12 years in various research and management positions in industry before accepting a professorship in Electrical Engineering at RWTH Aachen University in 1977. He has worked extensively in the areas of communication theory, digital signal processing and CAD tools for system level design for the last thirty years. His research has been applied to the design of many industrial products. At RWTH Aachen University he is a co-director of the Institute for Integrated Signal Processing System (ISS).

He was a co-founder of CADIS GmbH (acquired 1993 by Synopsys, Mountain View, California) a company which commercialized the tool suite COSSAP. In 2001 he has co-founded LISATek Inc., a company with breakthrough technology to design application specific processors. In 2003 LISATek has been acquired by CoWare. At CoWare Dr. Meyr has accepted the position of Chief Scientist.

Dr. Meyr has published numerous IEEE papers and holds many patents. He is author (together with Dr. G. Ascheid) of the book “Synchronization in Digital Communications”, Wiley 1990 and of the book “Digital Communication Receivers. Synchronization, Channel Estimation, and Signal Processing” (together with Dr. M. Moeneclaey and Dr. S. Fechtel), Wiley, October 1998. He has received three IEEE best paper awards.

Rainer Leupers received the Diploma and PhD degrees in Computer Science with honors from the University of Dortmund, Germany, in 1992 and 1997. From 1997 to 2001 he was a senior researcher and lecturer at the Embedded Systems group at the University of Dortmund. Between 1999 and 2001 he was also a project manager at Informatik Centrum Dortmund (ICD), where he headed industrial software tool projects, including C compiler and simulator design for DSP processors. In 2002, Dr. Leupers joined RWTH Aachen University as a professor for Software for Systems on Silicon.

His research and teaching activities revolve around software development tools, processor architectures, and electronic design automation for embedded systems, with emphasis on application specific processors. He authored several books and numerous technical papers on software tools for embedded processors, and he served in the program committees of leading EDA and compiler conferences, including DAC, DATE, and ICCAD.

Dr. Leupers received several scientific awards, including Best Paper Awards at DATE 2000 and DAC 2002. He has been a co-founder of LISATek, acquired by CoWare Inc. in 2003.

Index

- Abstraction, 69
 - functional, 68
 - interconnect, 68
- Abstraction level, 1, 27–29, 34, 68
 - LISA, 23
 - RTL, 23
- Accelerator, 6, 7
- Activation, 44
- Activation section, 54, 60, 61
- Address generation, 123, 125
- Application-Specific Multirate DSP (ASMD), 144
- Approximation
 - area, 86
 - timing, 85
- ARC International, 17
- Architectural efficiency, 30, 77, 84, 102, 154
- Architectural flexibility, 30
- Architectural information, 20, 32, 41
- Architectural parameter, 31
- Architecture structure, 109, 131, 140
- ASIC, 1, 2, 4, 117
- ASIP, 2–4, 6–9, 11, 23, 29–31, 41, 101, 117, 132, 133, 149
- Assembler, 23
- Behavior, 44
- Behavior section, 51
- Binary Decision Diagrams (BDDs), 20
- Boundary constraints, 30
- Boundary-scan, 105
- Breakpoint, 109
- C-compiler, 23
- Checkers, 15
- Chess, 15
- Clique covering, 92
- Clock domain, 106
- Coding, 44
 - Coding coverage, 47
 - Coding element, 46
 - non-terminal, 46
 - terminal, 46
 - Coding mask, 48
 - Coding root, 45, 47
 - Coding section, 46
 - Compatibility
 - assembly, 136
 - binary, 136
 - C-code, 136
 - Compatibility graph, 55
 - Complexity
 - C-Compiler design, 25
 - Algorithmic, 62
 - Conflict graph, 55
 - Control Flow Graphs (CFGs), 71, 161
 - Cost
 - estimation, 84
 - model, 84
 - Cycle based, 41
- Data Flow Graphs (DFGs), 71, 161
- Data throughput, 117
- Data trace, 167
- Data-path, 128, 145
- Debug Configuration Registers (DCRs), 111
- Debug control register, 105
- Debug mechanism, 101, 109
- Debug mode, 109
- Debug state machine, 110
- Declarative description, 32
- Declare, 44
- Decoder, 50
- Decoder generation, 49
- Design efficiency, 30, 31
- Design goals, 24
- Design space, 102

- Design space exploration, 23, 101
- DSPs, 5
- Entities, 68
- Epilogue, 130
- Exclusiveness information, 55
- Exploration loop, 24
- EXPRESSION, 12
- Expression, 44
- Expression section, 53
- False loops, 93
- Fetch, 138
- FlexGdb, 13
- FlexSim, 13
- FlexWare, 13
- FlexWare2, 13
- FPGA, 4
- Functional units, 42
- Gate-level synthesis, 20, 140, 148, 153, 154, 158, 163
- Generation
 - C-Compiler, 25
 - Simulator, 26
 - Assembler, 25
 - Debug mechanism, 103
 - JTAG interface, 103
 - Linker, 25
 - RTL hardware model, 30
- GENSIM, 13
- GPPs, 5
- Graph
 - compatibility, 91
 - conflict, 91
 - directed acyclic, 32, 44
- Graph coloring problem, 92
- Hardware Abstraction Layer (HAL), 168
- Hardware Description Language, 32
- Hardware implementation, 29
- HDL, 32
- Heterogeneous architectures, 4
- HGEN, 13
- Hot spots, 24
- ICORE, 154
- IMEC, 15
- Implementation flexibility, 30, 31
- Indian Institute of Technology, 15
- Infineon, 144
- Input Similarity, 93
- Institute for Integrated Signal Processing Systems (ISS), 14
- Instruction encoding generator, 24
- Instruction-set, 128, 138, 144
- Interconnect, 32
- ISDL, 13
- JTAG, 168
- JTAG interface, 101, 104, 109, 111, 169
- Karnaugh Maps, 20
- Korea Advanced Institute of Science and Technology (KAIST), 14
- Legacy code, 135
- Linker, 23
- LISA, 23
- LISA operations, 44, 59
- LT architecture family, 156
- M68HC11, 137
- Main operation, 45
- Mapping, 72
- Massachusetts Institute of Technology (MIT), 13
- Memory, 42, 117, 123, 137
- Memory substitution, 168
- MetaCore, 14
- MetaCore Behavioral Language (MBL), 15
- Metastable, 106
- MIMOLA, 15
- Minimization
 - Area, 84
 - Decision, 98
 - Dependency, 96
- Mode controller, 112
- Model structure, 42
- Motorola 68HC11, 137
- MSSQ compiler, 15
- MultiProcessor (MP) solutions, 5
- Nexus Standard, 168
- NML, 15
- NP-complete, 92
- NP-hard, 92
- Off-the-shelf processors, 4
- Operators
 - commutative, 87
 - n-ary, 90
 - non-commutative, 87
- Optimizations, 32, 73
 - constraint-dependent, 77
 - constraint-independent, 77
- Output Similarity, 93
- Ownership trace, 167
- Pareto points, 77
- Path, 68
- PEAS, 16
- PEAS-III, 16
- Performance, 102

- Physical characteristics, 29–32, 77
- Pin, 109
- Pipeline, 131
- Pipelines, 42
- Ports, 109
- Precise constraints, 30
- Processes, 68
- Processor features, 30, 31, 101
- Profiler, 23
- Program trace, 167
- Prologue, 130

- Redundancy, 32
- Refinement, 72
- Resource section, 42
- Resource sharing, 84
- RTL, 1, 4, 12, 19, 35–39, 41, 56, 68, 72
- RWTH Aachen University, 14

- Section, 44
- SGS Thomson/Bell Northern Research, 13
- Signals, 68
- Sim-nML, 15
- Simulation technique
 - Compiled simulation, 27
 - interpretive, 27
 - Just-In-Time Cache Compiled, 27
- Simulator, 23
- Software application development, 25
- Software tools, 23
- Software tools generation, 25
- STMicroelectronics, 13
- Storage elements, 42
- Stretch Inc., 18
- Structuring, 72
- Synchronization circuit, 107

- Syntax, 44
- Syntax section, 50
- System level simulation, 28
- SystemC, 19
- SystemVerilog, 19

- Target Abstraction Layer (TAL), 168
- Target Compiler Technologies, 15
- Technische Universität Berlin, 15
- Tensilica Inc., 17
- Test Access Port (TAP), 104
- Test Clock input (TCK), 104
- Test Data Input (TDI), 104
- Test Data Output (TDO), 104
- Test data registers, 105
- Test Mode Select input (TMS), 105
- Test ReSeT input (TRST), 104
- Timing difference, 94
- Tipi, 16
- Turbo decoding, 117

- U.C. Berkeley, 16
- U.C. Irvine, 12
- UDL, 41, 72
- Unified Description Layer (UDL), 34, 67
- Unit, 68
- User mode, 109

- Verilog, 19, 74
- VHDL, 19, 74
- Views, 69
- Visualization, 73

- Watchpoints, 167

- Zero Overhead Loops (ZOLPs), 129