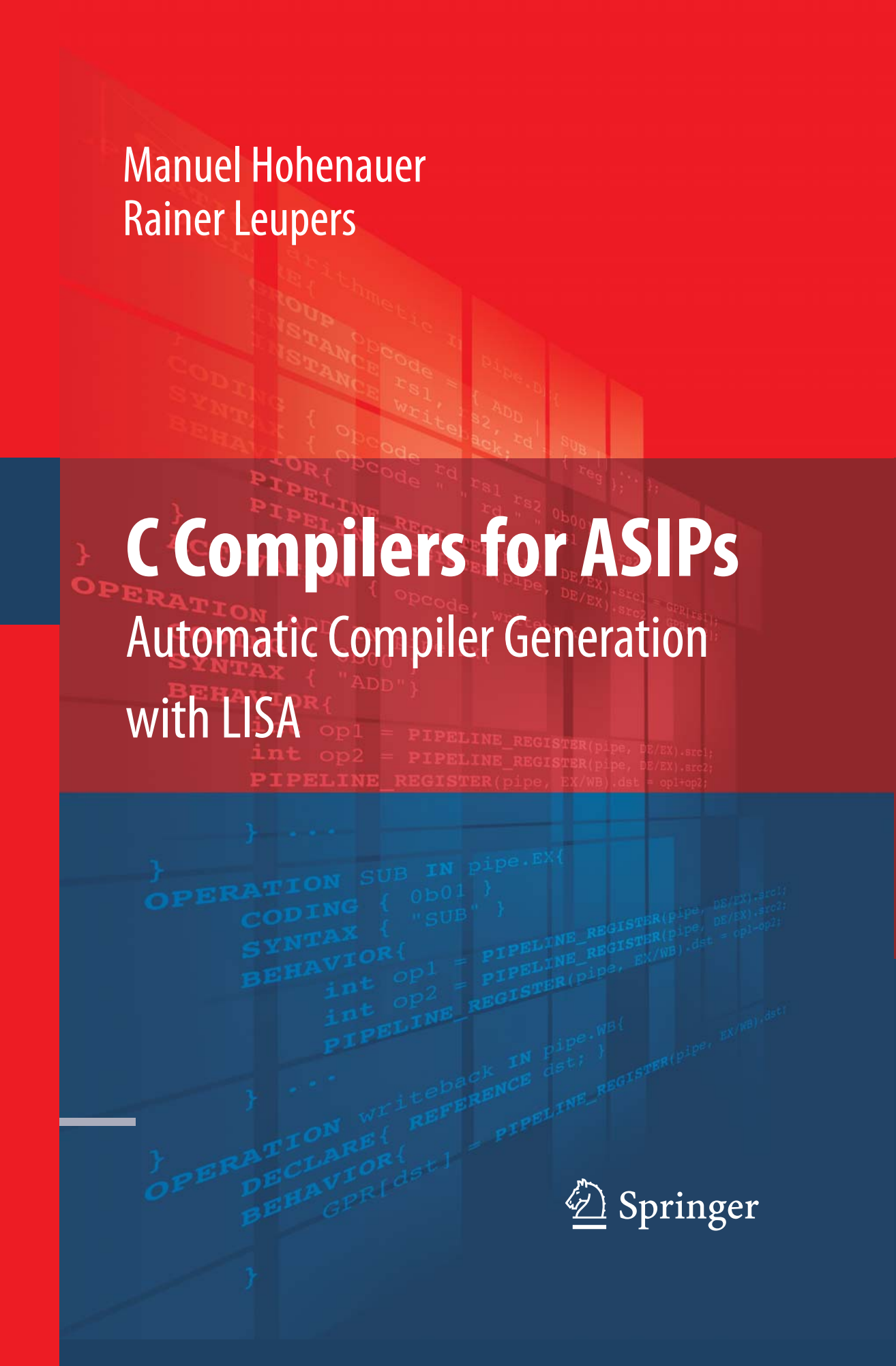
Manuel Hohenauer
Rainer Leupers

C Compilers for ASIPs

Automatic Compiler Generation with LISA



Springer

C Compilers for ASIPs

Manuel Hohenauer · Rainer Leupers

C Compilers for ASIPs

Automatic Compiler Generation with LISA

Springer

Manuel Hohenauer
RWTH Aachen University
Institute for Software
for Systems on Silicon
SSS · 611920
Templergraben 55
52056 Aachen
Germany
manuel.hohenauer@iss.rwth-aachen.de

Rainer Leupers
RWTH Aachen University
Institute for Software
for Systems on Silicon
SSS · 611920
Templergraben 55
52056 Aachen
Germany
leupers@iss.rwth-aachen.de

ISBN 978-1-4419-1175-9 e-ISBN 978-1-4419-1176-6
DOI 10.1007/978-1-4419-1176-6
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009936313

Dedicated to my parents, Hans and Inge

Acknowledgements

This book is based on my PhD thesis performed at the chair for Software for Systems on Silicon (SSS) at the RWTH-Aachen University. It documents the results of more than 5 years of work during which I have been accompanied and supported by many people. It is now my great pleasure to take this opportunity to thank them.

First and foremost, I would like to thank my PhD advisor Professor Rainer Leupers for providing me with the opportunity to work in his group, and for his important advice and constant encouragement throughout the course of my research. He always left me a lot of freedom and contributed much to an enjoyable and productive working atmosphere. I am also thankful to Professor Gerd Ascheid and Professor Heinrich Meyr. Their comments often unveiled new interesting aspects and perspectives. I want to thank all of them for the lessons they gave me on the importance of details for the success of an engineering or scientific project. It has been a distinct privilege for me to work with them. Also I would like to thank Professor Sabine Glesner for her interest in my work and for her commitment as a secondary advisor.

There are a number of people in my everyday circle of colleagues who have enriched my professional life in various ways. I am particularly indebted to my colleagues Oliver Wahlen, Jiangjiang Ceng, and Gunnar Braun, who worked together with me on the Compiler Designer project. Without their contributions, their support, and the inspiring working atmosphere, this work would have been impossible. I am also indebted to Felix Engel for many stimulating discussions and the excellent cooperation in the SIMD project. Life would be bleak without all the nice and funny moments I had with my co-students during all these years. I thank all of them.

I was fortunate to have enthusiastic support from students who worked with me toward their theses. Without their contributions I could never have realized this work. I sincerely offer my gratitude to Gerrit Bette, Felix Engel, Andriy Gavrylenko, and Christoph Schumacher.

I am also grateful to Hanno Scharwächter, Torsten Kempf and Stefan Kraemer who were patient and brave enough to carefully proofread this book. Their constructive feedback and comments at various stages have been significantly useful in shaping the book up to completion.

At last, I would like to thank the people whom I care most in the world, my family and Wibke. I would like to thank Wibke for the many sacrifices she has made

to support me in undertaking my doctoral studies and especially while writing this book. By providing her steadfast support in hard times, she has once again shown the true affection and dedication she has always had toward me. Finally, my biggest thanks go to my parents Hans and Inge without whom I would not be sitting in front of my computer typing these acknowledgements lines. I owe my parents much of what I have become. I dedicate this work to them, to honor their love, patience, and support throughout my entire studies.

Contents

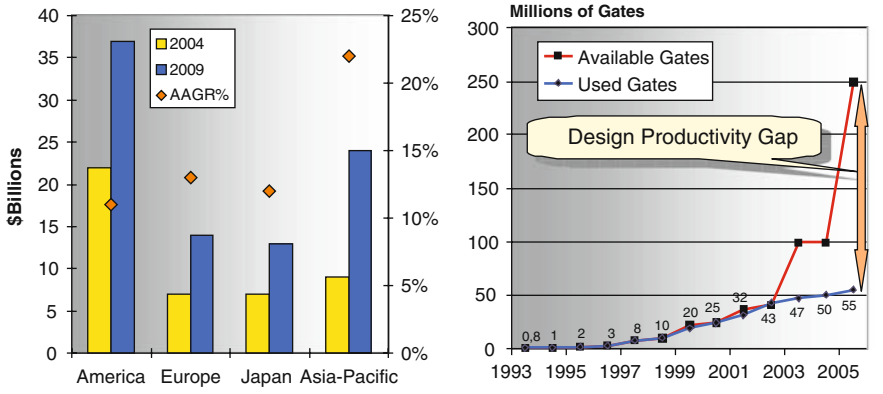
List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

Digital information technology has revolutionized the world during the last few decades. Today about 98% of programmable digital devices are actually embedded [132]. These embedded systems have become the main application area of information technology hardware and are the basis to deliver the sophisticated functionality of today's technical devices. As shown in Fig. 1.1(a), current forecasts predict a worldwide embedded system market of $88 billion in 2009.



(a) Global embedded systems revenue and average annual growth rate(AAGR) [103]

(b) Crisis of complexity [217]

Fig. 1.1 Embedded system design

Over the past few years, the ever-increasing complexity and performance requirements of new wireless communications, automotive and consumer electronics applications are changing the way embedded systems are designed and implemented today. In conformity with Moore's law [99], one driving force is the rapid progress in deep-submicron process technologies. Chip designers and manufacturers have constantly pushed the envelope of technological and physical constraints. In fact, designers have more gates at their disposal than ever before. However, current

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_1, © Springer Science+Business Media, LLC 2010

mainstream-embedded system designs are not using at least 50% of the silicon area available to them (Fig. 1.1(b)). The growth in design complexity threatens to outpace the designer's productivity, on account of unmanageable design sizes and the need for more design iterations due to deep-submicron effects. This phenomenon is also referred to as *crisis of complexity* [103] and comes along with exponentially growing *non-recurring engineering* (NRE) costs (Fig. 1.2) to design and manufacture chips. Understandably, these costs only amortize for very large volumes or high-end products.

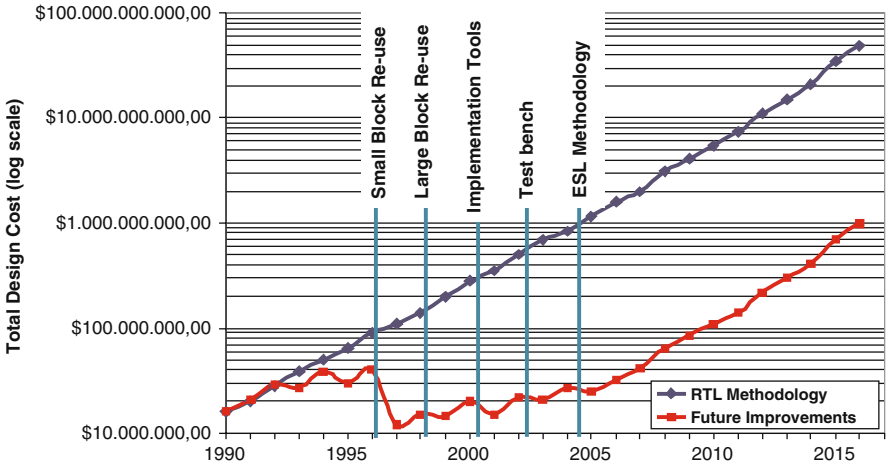


Fig. 1.2 Projected embedded system design cost model [123]

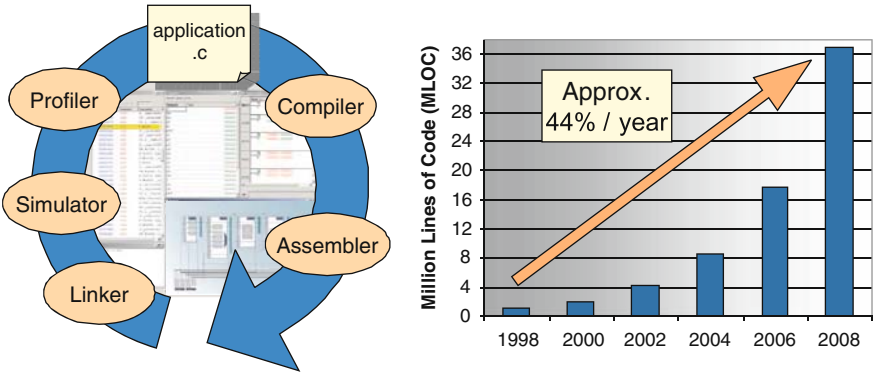
Consequently, more and more *application-specific integrated circuits* (ASICs) are replaced by programmable processors. Such processor platforms extend the product life cycle and achieve greater design reuse via software, thereby reducing development times and NRE costs. Moreover, the flexibility of software can be used to create design derivates, to make functional corrections due to process defects, and to provide performance improvements via updates.

Meanwhile, the high degree of integration offered by today's semiconductor technology permits increasingly complex systems to be realized in a single programmable *system-on-chip* (SoC). Current SoC designs employ several programmable processor cores, memories, ASICs, and other peripherals as building blocks. It is conjectured that by the end of the decade, SoCs feature hundreds of heterogeneous processor cores connected by a *network-on-chip* (NoC).

In order to efficiently explore the huge design space, tools and methodologies that offer the next level of productivity required for successful SoC design are needed. This has led to significant research activities in the field of *electronic system level* (ESL) design. ESL design automation tools provide the ability to quickly assemble, simulate, and analyze alternative architectures. The ultimate goal is to find the optimal combination of components for the given application domain within a short

time-to-market window. One piece in this puzzle is to rightly balance flexibility vs. performance for each system component.

On one side of the flexibility vs. performance spectrum are general purpose processors (GPPs). They offer high programmability and low design time, but may not satisfy area and performance challenges. On the other side of the spectrum are ASICs. They can be easily optimized for the given application, but, naturally, provide almost no flexibility and suffer from a lengthy and costly design process. Therefore, an increasing number of embedded SoC designs employ *application-specific instruction-set processors* (ASIPs) [29, 131, 164] as efficient implementation vehicles. They provide the best of both worlds, i.e., high flexibility through software programmability and high performance through specialization. However, finding the optimal balance between flexibility, performance, and energy efficiency constraints requires a thorough architecture exploration. This process demands software development tools in order to efficiently map application programs to varying ASIP configurations. In particular, the availability of a compiler translating high-level programming languages to assembly code became inevitable. Embedded processors have been traditionally programmed in assembly languages due to efficiency reasons. Considering the increasingly growing software content of SoCs (Fig. 1.3(b)), this is a time-consuming and error-prone process that is no longer feasible given today's tight time-to-market constraints. Furthermore, *compiler-in-the-loop* design space exploration helps to understand the mutual dependencies between processor architectures, the respective instruction-set, compilers, and the resulting code [194]. Otherwise the result might be a strong compiler-unfriendly architecture leading to an inefficient application design in the end.



(a) Compiler-in-the-loop architecture exploration

(b) Software complexity [85]

Fig. 1.3 Compiler for ASIPs

Nowadays *retargetable compilers* are widely used for architecture exploration since they can be quickly adopted to varying processor configurations. Unfortunately, such compilers are often hampered by their limited code quality as compared to handwritten compilers or assembly code due to the lack of dedicated optimization

techniques. In order to narrow the code quality gap, this needs generalized optimization techniques for those architectural features that are often recurring in ASIP design. This achieves retargetability *and* high code quality for a whole target processor class.

A complete compiler-in-the-loop architecture exploration as shown in Fig. 1.3(a) also demands assembler, linker, simulator, and profiler, which, naturally, have to be retargetable as well. This led to the development of *architecture description languages* (ADLs) that enable the automatic generation of the complete software toolkit (or at least components thereof) from a single-processor model. The high degree of automation reduces the design effort significantly and hence allows the designer to explore a larger number of architectural alternatives. The most challenging task designing an ADL, though, is to capture the architectural information needed for the tool generation in an unambiguous and consistent way. This is particularly difficult for compiler and simulator as they essentially need both the information about the *instruction's semantics* but from different points of view. The compiler, more specifically the compiler's *code selector*, needs to know *what* an instruction does in order to select appropriate instructions for a given piece of source code, while the simulator needs to know *how* the instruction is executed. In practice it is quite difficult, if not impossible, to derive one information from the other. None of the existing ADLs – if compiler generation is supported at all – solves this problem in a sophisticated manner. Either redundancies are introduced or the language's flexibility is sacrificed. Moreover, the specification of compiler-relevant information mostly requires in-depth compiler knowledge. This particularly applies for the code selector specification, the largest part of the compiler description. So far, there is almost no support to generate code selector descriptions automatically.

This book presents a solution to the aforementioned retargetable compilation problems. A novel technique is developed for extracting the code selector description fully automatically from an ADL processor model. The approach is based on the LISA ADL [15] using a language extension for *instruction semantics* description. This enables the automatic generation of both C compilers *and* simulator from a single-processor description without losing flexibility or introducing inconsistencies. In this way, a high speedup in compiler generation is achieved, which contributes to a more efficient ASIP design flow. The feasibility of the approach is demonstrated for several contemporary embedded processors.

In order to improve the code quality of the generated compilers, *retargetable optimizations* for two common ASIP features, namely *single instruction multiple data* (SIMD) support and *predicated execution*, are presented. Several representative RISC cores and VLIW architectures are used as driver architectures to obtain code quality results. In this way, the code quality of the generated compilers for architectures equipped with at least one of these features can be significantly improved. Furthermore, a new *retargetable assembler* is implemented supporting an interface for the implementation of code optimizations. This allows the user to quickly create custom low-level optimizations. An *instruction scheduler* and *peephole optimizer* are built as demonstrators.

As a result, this book presents an integrated solution to enable a complete and retargetable path from a single LISA processor model to a highly optimizing C compiler and assembler. This completes LISA's already established capabilities such that efficient compiler-in-the-loop architecture exploration becomes broadly feasible.

1.2 Outline of the Book

This book is organized as follows. Chapter 2, provides a background covering the necessity of architecture description formalisms and compiler-in-the-loop architecture exploration. Afterward, Chapter 3 gives a short introduction to compiler construction where the most important concepts required for the scope of this book are summarized. Chapter 4 describes the related work in the field of compiler-aided processor design. The advantages and drawbacks of various approaches are also clearly mentioned. Surveys of relevant publications specifically related to individual chapters of this book are given at the beginning of the corresponding chapters. The work presented in this book is integrated into the industry-proven *Processor Designer* ASIP design platform. The related *Language for Instruction-Set Architectures* (LISA) ADL and the current C compiler generation flow are elaborated in Chapter 5, whereas Chapter 6 presents a novel technique to generate the code selector description fully automatically from a LISA processor description. Chapter 7 provides an analysis of the code quality produced by the generated compilers. Afterward, Chapter 8 and 9 present two high-level retargetable code optimizations, more specifically, an optimization for the class of processors with *SIMD support* and *predicated execution*, respectively. Chapter 10 concentrates on a retargetable assembler for the quick implementation of user-defined assembly-level optimizations. Chapter 11 finally summarizes the major results of this work and gives an outlook to future research. Appendix A contains an overview of the developed LISA language extensions and Appendix B provides the formal description of the database as used for code selector generation.

Chapter 2
ASIP Design Methodology

The design of an ASIP is a challenging task due to the large number of design options. The competing design decisions such as flexibility, performance, and energy consumption need to be weighted against each other to reach the optimal point in the entire design space. Moreover, the increasing software complexity of today's SoCs requires a shift from traditional assembly programming to high-level languages to boost the designer's productivity. As a result, processor designers demand an increasing support from the design automation tools to explore the design space and rightly balance the flexibility vs. performance trade-off.

Section 2.1 first presents the four major phases in an ASIP design. Afterward, Section 2.2 elaborates on the benefits and issues of compiler-in-the-loop architecture exploration. Finally, Section 2.3 presents prominent ASIP design methodologies. A survey of different ASIP design environments is given in [171].

2.1 ASIP Design Phases

The design of an ASIP is a highly complex task requiring diverse skills in different areas. The design process can be separated into four interrelated phases (Fig. 2.1):

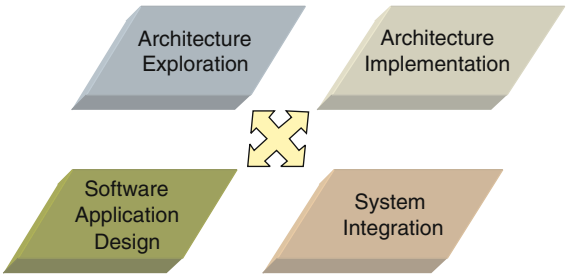


Fig. 2.1 ASIP design phases

Architecture exploration: The target application is mapped onto a processor architecture in an iterative process that is repeated until a best fit between architecture and application is obtained. According to Amdahl's law [88], the application's hot spots need to be optimized to achieve high performance

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_2, © Springer Science+Business Media, LLC 2010

improvements, and hence constitute promising candidates for dedicated hardware support and custom instructions. In order to identify those hot spots, profiling tools such as in [148, 203] are employed. Based on this hardware/software partitioning the instruction-set architecture (ISA) is defined in a second step. Afterward, the micro-architecture needs to be designed that implements the ISA. The whole process requires an architecture-specific set of software development tools (compiler, assembler, linker, simulator, and profiler). Unfortunately, every change to the architecture specification requires a complete new set of software development tools.

Architecture implementation: The specified processor is converted into a synthesizable hardware description language (HDL) model. For this purpose, languages such as VHDL [121] or Verilog [120] are employed. This model can then be further used for a standard synthesis flow (e.g., design compiler [250]). With this additional transformation, quite naturally, considerable consistency problems can arise between the architecture specification, the software development tools, and the hardware implementation.

Software application design: Software designers need a set of production-quality software development tools for efficient application design. However, the demands of the software application designer and the hardware processor designer place different requirements on software development tools. For example, the processor designer needs a cycle/phase-accurate simulator for hardware–software partitioning and profiling, which is very accurate, but inevitably slow. The application designer in contrast demands more simulation speed than accuracy. At this point, the complete set of software development tools is usually re-implemented by hand, which leads to consistency problems.

System integration and verification: The designed ASIP must be integrated into a system simulation environment of the entire SoC for verification. Since the interaction of all SoC components may have an impact on the processor performance, this provides more accurate results as compared to an instruction-set simulator. However, in order to integrate the software simulator, co-simulation interfaces must be developed. Again, manual modifications of the interfaces are required with each change of the architecture.

In traditional ASIP design, these phases are processed sequentially and are assigned to different design groups each with expert knowledge in the respective field. Design automation – if available at all – is mostly limited to the individual phases. Moreover, results in one phase may impose modifications in other phases. As a result, the complexity of design team interactions and communications necessary to successfully undertake a SoC-based design is a significant time-consuming factor. What makes this even more challenging is the large number of design alternatives that need to be weighted against each other. Consequently, the designer's productivity becomes the vital factor for successful products due to the complexity and tight time-to-market constraints. As a result, there is a strong interest in comprehensive design methodologies for efficient embedded processor optimization and exploration.

2.2 Compiler-in-the-Loop Architecture Exploration

Much of the functionality in a SoC is implemented in software due to a number of reasons: the flexibility of software offers wide design reuse (to reduce NRE costs) and compatibility across applications. It is conjectured that the amount of software in embedded systems roughly doubles every 2 years [85]. As a result, a rapidly increasing amount of software has to be validated and/or developed. This involves not only essential hardware drivers but also complete operating systems. Furthermore, new applications, exploiting the new hardware capabilities, need to be developed before the end products based on the SoC can be sold.

Compilers are among the most widespread software tools, used for decades on desktop computer. For embedded processors, however, the use of compilers is traditionally less common. Many designers still prefer assembly languages due to efficiency reasons. Considering the increasing complexity of applications and today's short time-to-market windows, assembly programming is no longer feasible due to the huge programming effort, portability, and maintainability. Obviously, such requirements can be much better met by using *high-level language* (HLL) compilers. In the context of embedded systems, the C programming language [45] is widely used. It is a well-tried programming language that allows a very low-level programming style at a stretch. Additionally, this enables a broad design reuse since there already exists a large amount of industry standards and legacy code in C. Unfortunately, designing a compiler is a complex task that demands expert knowledge and a large amount of human resources. As a result, compilers are often not available for newly designed processors. Clearly, this increases the probability of designing a strong compiler-unfriendly architecture, which leads to an inefficient application implementation in the end. In fact, many in-house ASIP design projects suffer from the late development of the compiler. Compiler Designers often have severe difficulties ensuring good code quality due to instruction-sets that have primarily been designed from a hardware designer's perspective. On the other hand, a compiler-friendly instruction-set and architecture might not be entirely suitable to support the hardware designer's effort meeting constraints such as area and power consumption. Therefore, *compiler-in-the-loop* architecture exploration is crucial to avoid a compiler and architecture mismatch right from the beginning and to ensure an efficient application design for successful products.

The inherently application-specific nature of embedded processors leads to a wide variety of embedded processor architectures. Understandably, developing the software tools, in particular the compiler, for each processor is costly and extremely time-consuming. Therefore, *retargetable C compilers* have found significant use in ASIP design in the past years since they can be quickly adapted to varying processor configurations. This is also a result of the increasing tool support for automatically retargeting a C compiler based on formalized processor descriptions [224].

In *compiler-in-the-loop* architecture exploration the compiler plays a key role to obtain exploration results. Due to the ambiguity of the transformation of C applications to assembly code, it is possible to quickly evaluate fundamental architectural changes with minimal modifications of the compiler [194]. In this way, designers

can meaningfully and rapidly explore the design space by accurately tracking the impact of changes to the instruction-set, instruction latencies, register file size, etc. This is an important piece in the puzzle to better understand the mutual dependencies between micro-architecture design, the respective instruction-set, compilers, and the achieved code quality. What is most important in this context is the specification of the compiler's code selector. It basically describes the mapping of the source code to an equivalent sequence of assembly instruction and hence significantly affects the final ISA definition (i.e., the software/hardware partitioning). However, the success of compiler-aided architecture exploration strongly depends on a flexible C compiler backend that is generated from the processor description.

Even though retargetable compilers have found significant use in ASIP design in the past years, they are still hampered by their limited code quality as compared to handwritten compilers or assembly code. This is actually no surprise, since higher compiler flexibility comes at the expense of a lower amount of target-specific code optimizations. Since such compilers can only make few assumptions about the target machine, it is, understandably, much easier to support machine-independent optimizations rather than techniques exploiting novel architectural features of emerging embedded processors. However, the lower code quality of the compilers is usually acceptable considering that the C compiler is available early in the processor architecture exploration loop. Thus, once the ASIP architecture exploration phase has converged and an initial working compiler is available, it must be *manually* refined to a highly optimizing compiler or the application's hot spots must be manually replaced by assembly programs – both are time-consuming tasks. One way to reduce the design effort is to provide *retargetable optimizations* for those architectural features that characterize a processor class, e.g., hardware multi-threading for network processors (NPU) [110]. In this way, retargetability *and* high code quality for this particular class of processors is achieved. For instance, *retargetable software pipelining* support is less useful for scalar architectures; however, it is a necessity for the class of VLIW processors, and for this class it can be designed in a retargetable fashion. This book contributes retargetable optimization techniques for two common ASIP features to further improve the code quality of retargetable compilers.

A retargetable assembler, linker, simulator, and profiler complete the required software development infrastructure. Needless to say that keeping all tools manually consistent during architecture exploration is a tedious and error-prone task. Additionally, they must also be adapted to modifications performed in the other design phases. As a result, different automated design methodologies for efficient embedded processor design have evolved. Two contemporary approaches are presented in the next section.

2.3 Design Methodologies

One solution to increase the design efficiency is to significantly restrict the design space of the processor. More specifically, such design environments are limited to a

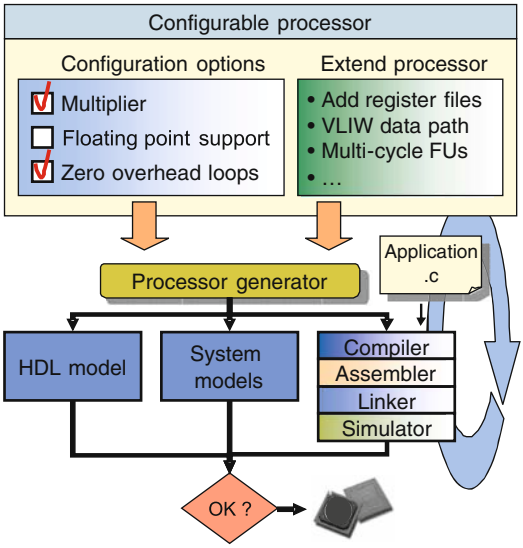


Fig. 2.2 Configurable processor design

predefined processor template whose software tools and architecture can be configured to a certain extent (Fig. 2.2).

Prominent examples for this approach are the *Xtensa* [215] and the *ARCtangent* [43] processor families. Considering that all configuration options are preverified and the number of possible processor configurations is limited, the final processor can be completely verified. However, this comes at the expense of a significantly reduced design space, which imposes certain limitations. The coarse partitioning of the design space makes it inherently difficult to conceive irregular architectures suited for several application domains. Furthermore, certain settings of the template may also turn out to be redundant or suboptimal, like memory interface or the register file architecture for instance. Another limitation is imposed by the support for custom instructions. Such instructions must be typically given in an HDL description, and hence cannot be directly utilized by the compiler.

Another, more flexible concept for ASIP design is based on *architecture description languages* (ADLs). Such languages have been established recently as a viable solution for an efficient ASIP design (Fig. 2.3). ADLs describe the processor on a higher abstraction level, e.g., instruction accurate or cycle accurate, to hide implementation details. One of the main contribution of such languages is the automatic generation of the software toolkit from a single ADL model of the processor. Advanced ADLs are even capable of generating the system interfaces and a synthesizable HDL model from the same specification. This eliminates the consistency problem of the traditional ASIP design flow since changes to the processor model directly lead to a new and consistent set of software tools and hardware implementation. In this way, they provide a systematic mechanism for a top-down design and validation of complex systems. The high degree of automation reduces the design

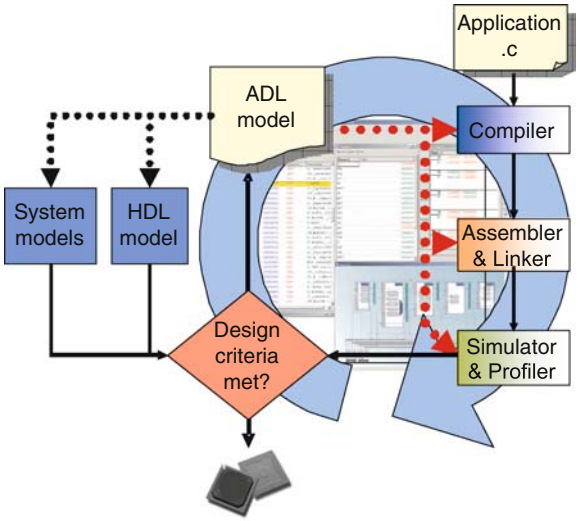


Fig. 2.3 ADL-based architecture exploration

effort significantly and thus allows the designer to explore a larger number of architectural alternatives.

Early ADLs, such as ISPS [157], were used for the simulation, evaluation, and synthesis of computers and other digital systems. Contemporary ADLs can be classified into three categories [112] based on the kind of information an ADL can capture:

Instruction-set centric: Instruction-set-centric languages have been designed with the generation of an HLL compiler in mind. Consequently, such languages must capture the instruction-set behavior (i.e., syntax, coding, semantic) of the processor architecture, whereas the information about the detailed micro-architecture (i.e., pipeline stages, memories, buses, etc.) does not need to be included. However, it is hardly possible to generate HDL models from such specifications. Typical representatives for this kind of ADLs are nML [10, 141], ISDL [97], and CSDL [186].

Architecture centric: These kinds of ADLs capture the structure in terms of architectural components. Therefore, they are well-suited for processor synthesis. But on the other hand, these languages typically have a low abstraction level leading to a quite detailed architecture specification. Unfortunately, it is quite difficult, if not impossible, to extract compiler-relevant information (e.g., instruction's semantic) from such informal models. Prominent examples for this category of ADLs are MIMOLA [235], UDL/I [264], and AIDL [254].

Combination of both: These so-called *mixed-level* description languages [13] describe both, the instruction-set behavior and the structure of the design. This enables the generation of software tools as well as a synthesizable hard-

ware model. However, capturing both information can lead to a huge description, which is difficult to maintain. Additionally, such languages can suffer from inconsistencies due to duplicated informations. Certain architectural aspects need to be described twice, e.g., once for compiler generation and once for processor synthesis. ADLs belonging to this group are MDes [134], RADL [155], FlexWare [207], MADL/OSM [275], EXPRESSION [201], and LISA [15].

Obviously, designing an ADL that captures all aspects of ASIP design in an unambiguous and consistent way is a challenging task. This is further aggravated by the fact that most ADLs have originally been designed to automate the generation of a particular component and have then been extended to address the other aspects. As a result, ADLs are often well-suited for the purpose they have been designed for, but impose major restrictions on, or are even incapable of the generation of the other components. This is true in particular for the generation of compiler and simulator. Therefore, a further focus of this book are methodologies to generate compiler and simulator from a single ADL specification without limiting its flexibility or architectural scope. A detailed discussion of different ADLs is given in Chapter 4.

2.4 Synopsis

- Finding the optimal balance between flexibility and performance requires the evaluation of different architectural alternatives.
- HLL compilers are needed in the exploration loop to cope with the growing amount of software and to avoid hardware/software mismatches.
- The widely employed retargetable compilers suffer from their lower code quality as compared to handwritten compilers or assembly code.
- For quick design space exploration methodologies using predefined processor templates or ADL descriptions are proposed.
- ADL support for the automatic generation of the complete software tool chain (in particular, compiler and simulator) is currently not satisfactory.
- The primary focus of this book is the generation of C compilers from ADL processor models and retargetable optimization techniques to narrow the code quality gap.

Chapter 3
A Short Introduction to Compilers

This chapter summarizes briefly some basic terms and definitions of compiler construction as well as the underlying concepts. It focuses mainly on the terminology but not on detailed algorithms. More comprehensive surveys can be found, e.g., in [3, 229, 244].

3.1 General Overview

A compiler is a program that translates a program written in one language (the source language) into a semantically equivalent representation in another language (the target language). Over the years, new programming languages have emerged, the target architectures continue to change, and the input programs become ever more ambitious in their scale and complexity. Thus, despite the long history of compiler design, and its standing as a relatively mature computing technology, it is still an active research field. However, the basic tasks that any compiler must perform remain essentially the same.

Conceptually, the translation process can be subdivided into several phases as shown in Fig. 3.1. The first is the *analysis* phase, often called the *frontend*, which creates an *intermediate representation* (IR) of the source program. On this specification, many compilers apply a sequence of high-level, typically machine-independent optimizations to transform the IR into a form that is better suitable for code generation. This includes tasks such as common subexpression elimination, constant folding, and constant propagation. A very common set of high-level optimizations is described in [1]. This phase is also referred to as the *midend* of the compiler. Finally, the *synthesis* phase, or the *backend*, constructs the desired target program from the IR. The concrete organization within each phase, however, may strongly vary between different compilers, especially that of the optimizations in the midend and backend.

Frontend and backend are presented in more detail in the following sections.

3.2 Compiler Frontend

The first phase in the frontend is the *lexical analysis*. A *scanner* breaks up the program into constituent pieces, called *tokens*. Each token denotes a primitive element

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_3, © Springer Science+Business Media, LLC 2010

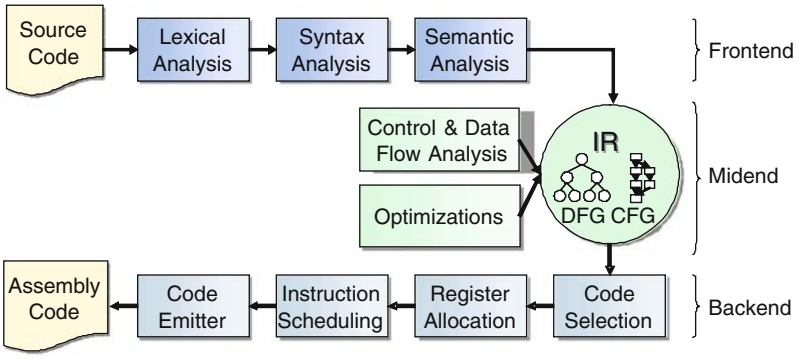


Fig. 3.1 Common compiler phases

of the source language, e.g., a keyword, an identifier, a character, etc. Generally, most of these elements can be represented by *regular expressions*, which can be parsed by *finite state machines* (FSMs). An FSM consists of a finite number of states and a function that determines transitions from one state to another as symbols are read from an input stream (i.e., the source program). The machine transitions from state to state as it reads the source code. A language element (e.g., a keyword or an integer number) is accepted if the machine reaches one of a designated set of *final* states. In this case, a corresponding token is emitted and the machine returns to the initial state to proceed with the next character in the stream. Given a list of regular expressions, scanner generators such as GNU's FLEX [106] can produce C code for the corresponding FSM that can recognize these expressions.

Definition 3.1 (Context-free grammar) *A context-free grammar G is a tuple $G = (T, N, R, S)$, where T denotes a finite set of terminals (i.e., the set of possible tokens), N a finite set of nonterminals, and $S \in N$ the start symbol. R is a relation from X to $(T \cup N)^*$, where X must be a member set of N.*

The tokens are then further processed by the *parser* to perform a *syntax analysis*. Based upon a *context-free grammar*, it identifies the language constructs and maintains a symbol table that records the identifiers used in the program and their properties. The result is a *parse tree* that represents a derivation of the input program from the start symbol S. If the token string contains syntactical errors, the parser may produce the corresponding error messages. Again, parser generators are available (e.g., GNU's BISON [105]), which can generate a C implementation from a context-free grammar specification.

Finally, a *semantic analysis* is performed that checks if the input program satisfies the semantic requirements as defined by the source language; for instance, whether all used identifiers are consistently declared and used. For practical reasons, semantic analysis can be partially integrated into the syntax analysis using an *attribute grammar* [67], an "extended" context-free grammar. Such grammars allow the annotation of a symbol $s \in (T \cup N)$ with an attribute set $A(s)$. An attribute $a \in A(s)$ stores semantical information about a symbol's type or scope. Each

grammar rule r, with $r \in R$, using a can be assigned an attribute definition $D(a)$. The attributes are divided into two groups: *synthesized* attributes and *inherited* attributes. The former are used to pass semantic information up the parse tree, while inherited attributes passing them down. Both kinds are needed to implement a reasonable semantic analysis. Such attribute grammar specifications can be further processed by tools such as OX [143] (an extension of FLEX and BISON) to finally create a parser with integrated semantic analysis.

The output IR format of the frontend is typically a list of expression trees or *three-address code*. Generally, the frontend is not dependent on the target processor. Thus, an existing language frontend can be combined with any target-specific backend, provided that all of them use the same IR format (Fig. 3.2).

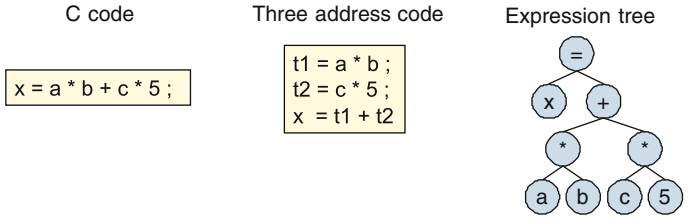


Fig. 3.2 IR format examples

3.3 Compiler Backend

The task of the backend is the code generation that consists of several subtasks. Since many of them are known to be NP-complete [163] problems, i.e., solving such problems most likely requires algorithms with exponential runtime, code generation typically relies on heuristics. Therefore and due to software engineering reasons, all code generation tasks are implemented by separate algorithms. However, these tasks are usually interdependent, i.e., decisions made in one phase impose constraints in subsequent phases. While this works well for regular architectures, it typically results in poor code quality for irregular architectures [270]. This is also known as the *phase coupling* problem.

Before the different subtasks are presented in the following sections, several program representations essential for most code generation subtasks (and for most compiler optimizations) are introduced first.

3.3.1 Data- and Control-Flow Graphs

The data- and control-flow graphs provide more detailed information about the program semantics than the plain IR representation. First, the *control flow* needs to be computed. Each function is split into its *basic blocks*.

Definition 3.2 (Basic block) *A basic block $B = (s_1, ..., s_n)$ is a sequence of IR statements of maximum length, for which the following conditions are true: B can only be entered at statement s_1 and left at s_n. Statement s_1 is called the leader of the basic block. It can either be a function entry point, a jump destination, or a statement that follows immediately after a jump or a return.*

Consequently, if the first statement of a basic block is executed, then all other statements are executed as well. This allows certain assumptions about the statements in the basic block, which enable the rearrangement of computations during scheduling for instance. Basic blocks can be easily computed by searching for IR nodes that modify the control flow of the program (e.g., goto and return statements). Once the basic blocks have been identified, the *control-flow graph* can be constructed. An example is given in Fig. 3.3.

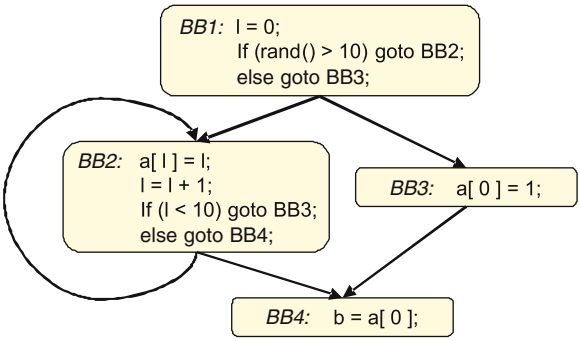


Fig. 3.3 Control-flow graph example

Definition 3.3 (Control-flow graph) *A control-flow graph (CFG) of a function F is a directed graph $G_F = (V_F, E_F)$. Each node $v \in V_F$ represents a basic block, and E_F contains an edge $(v, v') \in V_F \times V_F$ if v' might be directly executed after v. The set of successors succ of a basic block B is given by $succ_B = \{v \in V_F \mid (b, v) \in E_F\}$ and the set of predecessors pred of a basic block B is given by $pred_B = \{v \in V_F \mid (v, b) \in E_F\}$.*

The obvious edges are those resulting from jumps to explicit labels as the last statement s_n of a basic block. Furthermore, if s_n is a conditional jump or a conditional return, then a *fallthrough* edge to the successor basic block is additionally created. In certain cases, s_n is not a jump nor a return. Thus, in case a successor block exists and its first statement follows immediately after s_n in the IR representation, an edge to the successor block is created. Blocks without any outgoing edges have a return statement at the end. In case the resulting CFG contains unconnected basic blocks, there is an unreachable code that can be eliminated by a *dead code elimination* optimization without changing the program semantics.

While the CFG stores the control flow on a basic block level, another important data structure deals with the *data dependencies* between statements.

Definition 3.4 (Data dependency) *A statement s_j of a basic block $B = (s_1, ..., s_n)$ is data dependent on statement s_i, with $i < j$, if s_i defines a value that is used by s_j (i.e., s_i needs to be executed before s_j).*

A *data-flow analysis* (DFA) in its simplest form computes the data dependencies just for single basic blocks, and thus is referred to as local DFA. Basically, for each statement S, a data-flow equation is created, which requires the following information:

- in[S], the directly available information before S
- out[S] the information available after S
- gen[S] new information generated within S
- kill[S] the information killed by S

The equations depend on the kind of data-flow information that has to be computed. For the computation of reaching definitions, the equations have the following form:

$$\text{in}[S] = \bigcup_{p \in \text{pred}(S)} \text{out}[p] \tag{3.1}$$

$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S]) \tag{3.2}$$

In order to obtain the information about available expressions, the equations change to

$$\text{in}[S] = \bigcap_{p \in \text{pred}(S)} \text{out}[p] \tag{3.3}$$

$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S]) \tag{3.4}$$

Similar data-flow equations exists to compute the variables that are active at a certain program point. This information is required, e.g., for the register allocation. Solving the resulting system of equations gives the concrete data-flow information for the basic block. The result is stored in a *Data Flow Graph* (DFG).

Definition 3.5 (Data-flow graph) *A data-flow graph (DFG) for a basic block B is a directed acyclic graph $G_B = (V_B, E_B)$, where each node $v \in V_B$ represents an input operand (constant, variable), an output (variable) operand, or an IR operation. An edge $e = (v_i, v_j) \in E_B \subset V_B \times V_B$ indicates that the value defined by v_i is used by v_j.*

A DFG is called *data-flow tree* (DFT) if no node has more than one outgoing edge, i.e., there are no *common subexpressions*. Typically, DFTs build the input data for many popular code-selection techniques.

In practice, compilers perform a DFA for an entire function, called global DFA, since local DFA hinders many optimization opportunities. Suppose, a basic block has several outgoing control-flow edges, i.e., a definition of a variable (e.g., initialized with a constant) may reach multiple uses, possibly in different basic blocks. Thus, in order to exploit the full potential of, e.g., constant propagation, all uses

reached by that definitions are required, which can only be provided by a global DFA. Typically, local DFA is embedded as a sub routine in the global DFA that iteratively solves the data-flow equations for an entire procedure.

The analysis can be extended even beyond function boundaries. The general idea behind a so-called interprocedural analysis is to collect the information flowing into a function and then use it, to update the local information. This requires information about

- which functions f_t any particular function f calls,
- f's return values,
- which functions f_c call any particular function g, and
- which arguments f_c passes to g.

The information about the calling behavior is usually captured in the concept of a *call graph*. Figure 3.4 depicts an example call graph.

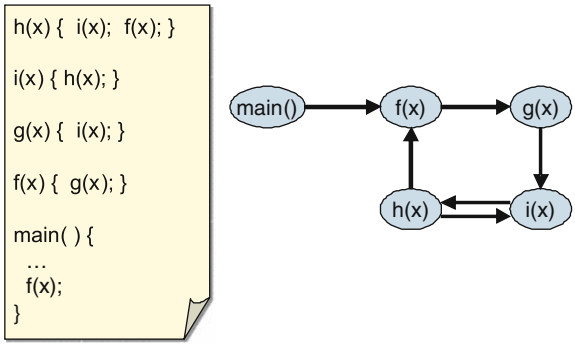


Fig. 3.4 Call graph example

Definition 3.6 (Call graph) *If a program P is composed of functions $f_1, \ldots, f_n$, then the call graph for P is a directed, edge-annotated graph $G_P = (V, E, s)$ with $V = \{f_1, \ldots, f_n\}$, $E \subset V \times V$, and $s : E \mapsto S$, where S is the set of call sites. If $e = (f_i, f_j) \in E$ and $s(e) = k$, then the function f_i calls the function f_j from the label k inside f_i.*

The interprocedural analysis therefore starts with the creation of a call graph to capture the dependencies. If the whole program is visible to the compiler, the direct and correct creation of a call graph is straightforward. Regardless of that, most modern software consists of separate compilation units, which are linked after their separate compilation to form the final program. The compiler is therefore not able to analyze the whole program at once. This also prohibits the creation of a complete call graph, since several uncertainties arise:

- *Library functions* may be called by the code known to the compiler. In that case, the name and type of the callee are usually known, but the code is not analyzable.
- A function might be called by a code outside the compilation unit. This is usually the case if the compiled module is part of a library. Many languages allow to

specify storage-class attributes such as the `static` keyword in C. Using those effectively rules out this possibility for specific functions.
- Functions may not be called directly by name but via function pointers. In that case, an explicit data flow analysis is necessary to determine the set of actual callees of a function call in a program.

3.3.2 Code Selection

Code selection is typically the first phase in the backend. Its task is to map the IR to a semantically equivalent sequence of machine instructions. A common technique for code selection uses DFTs as input and is based on *tree parsing*. This can be efficiently implemented by tree pattern matching combined with dynamic programming [2]. The basic idea is to describe the instruction-set of the target processor by a *context-free tree grammar* specification.

Definition 3.7 (Context-free tree grammar) *A context-free tree grammar G is a tuple $G = (T, N, P, S, w)$, where T denotes a finite set of terminals, N a finite set of nonterminals, and $P \subseteq N \times (N \cup T)^*$ a set of production rules. $S \in N$ is the start symbol and w is a cost metric $P \to \mathbb{R}$ for the production rules.*

In the context of tree pattern matching, T can be seen as the set of all IR nodes and N as some sort of temporaries or storage location (e.g., registers or memory) to transfer intermediate results either between or inside instructions. The *cost metric* describes the costs caused by executing the corresponding instruction, e.g., with regard to performance, code size, or power consumption. The target code is generated by reducing the DFT to a single node (or covering the DFT) by repeatedly applying one of the production rules P, i.e., a subtree T can be replaced by a nonterminal $n \in N$ if the rule $n \to T$ is in P.

As a typical example for a tree grammar rule, consider the rule for a register to register `ADD` instruction:

$$\text{reg} \to \text{PLUS(reg, reg)}\{\text{costs}\} = \{\text{actions}\} \tag{3.5}$$

with $reg \in N$ and PLUS $\in T$. If the DFT contains a subtree that matches a subtree whose root is labeled by the operator "PLUS" and its left and right sons are labeled with "*reg*," it can be replaced by *reg*. It should be noted here that both sons might also be the result of further tree grammar rules that have been applied before. Each rule is associated with a *cost* and an *action* section. The latter typically contains the code to emit the corresponding assembly instruction.

It might happen that more than one rule covers a subtree. A cover is optimal if the sum over all costs of involved rules is minimal. This can be implemented by a *dynamic programming* approach, i.e., the optimum solution is based on the optimum solution of (typically smaller) subproblems. More specifically, a tree pattern matcher traverses the DFT twice:

In the first *bottom-up* traversal, each node i of a DFT T is labeled with the set of nonterminals it can be reduced to, the cheapest rule $r \in P$ producing n and the total cost (i.e., the costs covering the subtree rooted at i). This includes also those nonterminals that might be produced by a sequence of rules. When the root node of T has been reached, the rule that produces the start nonterminal S with minimum cost is known.

In a second *top-down* traversal, the pattern matcher exploits the fact that a rule for a node i also implicitly determines the nonterminals the subtrees of i must be reduced to (otherwise the rule could not have been applied to i). Thus, starting at the root node, it can now be determined which nonterminals must be at the next lower level in T. Therewith for each nonterminal, the corresponding rule r can be obtained whose action section emits finally the instructions. This traversal is recursively repeated until the leaves of T have been reached. Figure 3.5 illustrates this process using the tree grammar specification in Table 3.1.

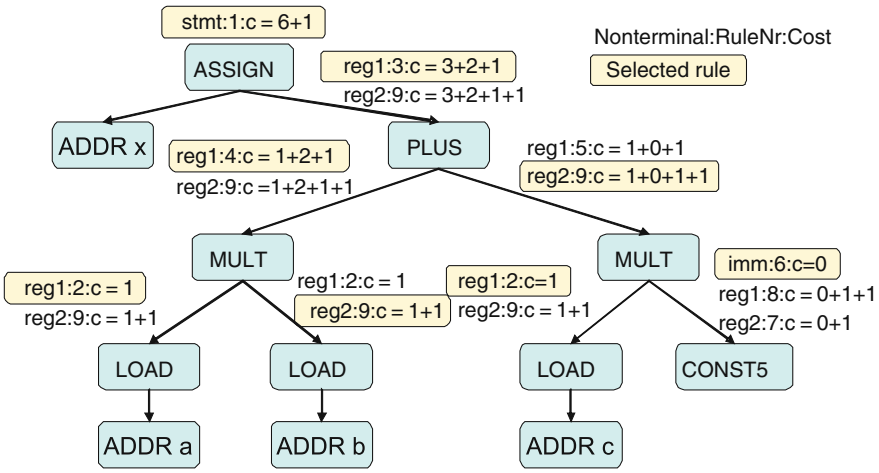


Fig. 3.5 Tree-pattern-matching example for the statement $x = a * b + c * 5$

Table 3.1 Tree grammar specification

Rule No.	Nonterminal		Tree pattern	Instruction	Costs
1	stmt	→	ASSIGN(ADDR,reg1)	STORE dst = src	1
2	reg1	→	LOAD(ADDR)	LOAD dst = src	1
3	reg1	→	PLUS(reg1,reg2)	ADD dest = src1, src2	1
4	reg1	→	MULT(reg1,reg2)	MUL dest = src1, src2	1
5	reg1	→	MULT(reg1,imm)	MULI dest = src1, src2	1
6	imm	→	CONST		0
7	reg2	→	imm	LOADI dst = src	1
8	reg1	→	reg2	MOVE21 dst = src	1
9	reg2	→	reg1	MOVE12 dst = src	1

Tree pattern matching finds an optimal set of instructions for a single DFT at linear time in the number of DFT nodes. Furthermore, a number of tools are available that can generate tree pattern matchers from a target-specific tree grammar specification. Examples of such so-called *code generator generators* are `BEG` [108], `burg` [52], `iburg` [51], `lburg` (code selector of the *lcc* compiler [50]), `OLIVE` (code selector of the SPAM compiler [247]), and `twig` [2].

In case the IR takes the form of a *direct acyclic graph* (DAG) (due to common subexpressions), it is usually split into a forest of DFTs based on heuristics. While this works well for regular architectures, for irregular architectures or architectures with special custom instructions this may result in suboptimal code quality. Typically, such architectures comprise instructions that exceed the scope of a single DFT. Therefore, different approaches to DAG-based code selection have been developed such as in [159, 234]. Unfortunately, optimal code selection on DAGs is known to be NP-complete. Thus, many approaches employ heuristics, impose several restrictions, or are mostly limited to small problem sizes in order to cope with the excessive runtime requirements. The work in [111] presents a code generator generator, called `cburg`, for a DAG-based code selector.

3.3.3 Register Allocation

The task of the register allocator is to assign variables and temporary values to a limited set of physical machine registers. Registers are very expensive with regard to area and power consumption. Therefore, many processor architectures implement only a small register file. Due to the increasing gap between the processor's speed and the memory access time, the register allocation must keep the largest possible number of variables and temporaries in registers to achieve good code quality. In the following, the most important definitions and concepts of register allocation are summarized.

Definition 3.8 (Life range) *A virtual register r is live at a program point p, if there exist a path in the control flow graph starting from p to an use of r on which r is not defined. Otherwise r is dead at p.*

Definition 3.9 (Interference graph) *Let V denote a set of virtual registers. An undirected graph $G = (V, E)$ is called interference graph if for all $v, w \in V$, the following condition holds: v and w have intersecting life ranges.*

State-of-the-art techniques for register allocation are based on a graph-coloring paradigm. The notion of abstracting storage allocation problems to graph coloring dates from the early 1960s [242]. More specifically, the problem of register allocation is translated into the problem of coloring the *interference graph* by K colors, where K denotes the number of available physical registers. The basic idea of the graph-coloring method is based on the following observation: If G contains a node n with degree d (i.e., the number of edges connected to n) with $d < K$, a color k from the set of K colors can be assigned to n that is different from the colors of all its

neighbors. The node n is removed from G and a new graph $G' = G - n$ is obtained that, consequently, contains one node and several edges fewer and the algorithm proceeds with the next node. This approach leads to a step-by-step reduction of the interference graph. Since graph coloring is NP-complete, heuristics are employed to search for a K-coloring. If such a coloring cannot be found for the graph, some values are spilled, i.e., values are kept in memory rather than in registers, which results in a new interference graph. This step is repeated until a K-colorable interference graph is found. An example is given in Fig. 3.6.

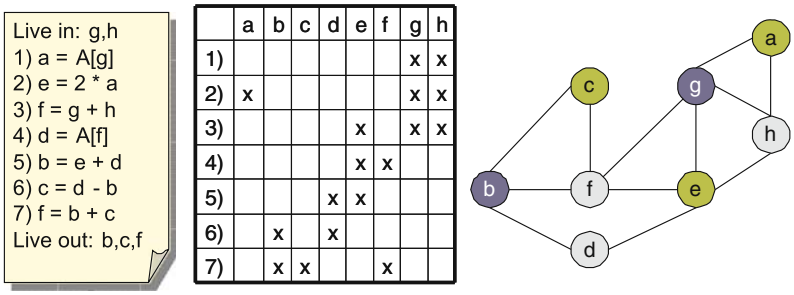


Fig. 3.6 Code example, life ranges, interference graph, and its coloring ($K = 3$)

The first implementation of a graph-coloring register allocator was performed by Chaitin et al. [93, 94]. Later, a priority-based scheme for allocation using graph coloring has been described in [80, 81]. Almost all subsequent work is based on these approaches.

The register allocation algorithms can be further subdivided according to their scope. *Local* register allocation, such as in [80, 94], works only on a single basic block at a time. In contrast, *global* register allocation algorithms exceed the basic block boundaries and take the control-flow structure of the program into account, e.g., an entire procedure or even a collection of procedures. Since the latter is able to take execution frequencies of loop bodies, life ranges over basic block boundaries, and calling conventions into account, a better cost analysis can be performed to improve the spill heuristics. Therefore, many register allocators today are global register allocators. Examples for graph-coloring-based global allocators are in [81, 199].

Of course, not all global allocation methods are based on graph coloring. Examples for different approaches include the bin-packing algorithm [198] and the probablistic register allocation given in [255]. Although graph-coloring allocators can be implemented efficiently, they have a quadratic runtime complexity. This makes them impractical whenever the compile time is a major concern like in *dynamic compilation environments* or *just-in-time (JIT) compilers*. For this domain, an allocator with linear runtime and acceptable code quality, called *linear scan allocator*, has been proposed [174]. The linear scan algorithm consists of the following four steps:

1. Order all instructions linearly.
2. Calculate the set of live intervals.

3. Allocate a register to each interval (or spill the corresponding temporary).
4. Rewrite the code with the calculated allocation.

The linear scan algorithm relies on a linear approximation of the instructions in order to determine simultaneously alive intervals. This order influences the extent and accuracy of live intervals, and hence the quality of the register allocation. As investigated in [151], a *depth-first* ordering is the optimal one.

After instruction ordering is performed, the live intervals are computed. For temporaries outside of a loop, the interval starts at the first definition of the register and ends at its last use. For temporaries alive inside a loop, the interval must be extended to the end of the loop. Given live variable information (e.g., via data-flow analysis [1]), live intervals can be computed easily with one pass through the ordered instruction list. Intervals interfere if they overlap. The number of overlapping intervals changes only at the start and end points of an interval. The computed live intervals are stored in a list that is ordered in increasing start points to make the allocation more efficient.

As defined in [174], given R available registers and a list of live intervals, the linear scan algorithm must allocate registers to as many intervals as possible, but such that no two overlapping live intervals are allocated to the same register. If $n > R$ live intervals overlap at any point, then at least $n - R$ of them must be spilled. For allocation, the linear scan algorithm maintains a number of sets:

1. The set of already allocated intervals called *Allocated*.
2. The mapping of active intervals to registers stored in the set named *Active*.

The algorithm starts with an empty *Active* set. For each newly processed live interval, the algorithm scans *Active* from the beginning to the end and moves those intervals to *Allocated* whose end points precede the processed interval's start point. Removing an interval from *Active* makes the corresponding register again available for allocation. The processed interval's start point becomes the new start position for the algorithm and gets a physical register assigned that is not used by any interval in *Active*. If all registers are already in use, one interval must be spilled. The spill heuristics selects the interval with the highest end position.

Figure 3.7 depicts an example. The live intervals shown in the middle correspond to the instruction ordering on the left. Suppose the set of allocatable physical registers is `R1`, `R2`, and `R3`. In the first step, the interval `V1` is processed and, since the *Active* list is empty, gets the physical register `R1` assigned. Consequently, `V1` is added to the *Active* list. When `V2` is visited in the next step, `V1` is still live and another register `R2` is assigned to `V2` and added to *Active*. Afterward, interval `V3` is processed and gets the last free physical register `R3` assigned. Since no physical register is available for `V4`, one interval must be spilled. The algorithm selects `V1` for spilling because it has the highest end position and removes it from the *Active* list. The example shows the corresponding state of the intervals and the active list. The final allocation after processing all intervals is depicted on the right.

A retargetable linear scan allocator for the CoSy environment [38] was implemented in [11] and compared to the regular graph-based register allocator. The

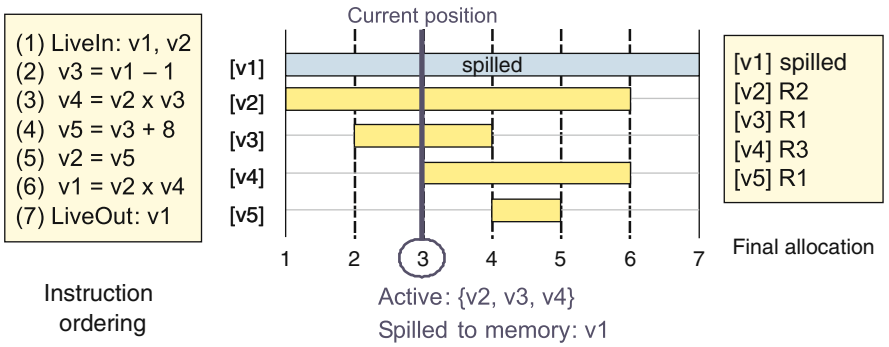


Fig. 3.7 Linear scan allocation example

results show an average speedup of 1.6–7.1 for the register allocation while attaining good code quality (average overhead in cycle count/code size is within 1–3%).

3.3.4 Instruction Scheduling

Most contemporary processors use pipelining to partially overlap the execution of instructions or even *Instruction-Level Parallelism* (ILP) to execute several instructions in parallel such as *Very Long Instruction Word* (VLIW) machines for instance. Generally, scheduling is the process of reordering instructions in such a way that the maximum amount of parallelism among instructions is exploited. Similar to register allocation, *local* schedulers work at the basic block level whereas *global* scheduler deal with complete functions.

The scheduling process is limited by two major constraints [214]: first, *data hazards* or *control hazards* causing dependencies between instructions that force a sequential ordering and second resource limitations, i.e., *structural hazards*, that force serialization of instructions requiring the same resource. A *dependency graph* that captures these constraints constitutes the input for most scheduling techniques.

Definition 3.10 (Dependency graph) *A dependency graph (DG) is an edge-weighted directed acyclic graph* $G = (V, E, type, delay)$*, where each node* v *in* V *represents a schedulable instruction. The resource allocation of each instruction is given by its reservation table* $r(v)$*. An edge* $e = (v_i, v_j) \in E \subseteq V \times V$ *indicates a dependency between* v_i *and* v_j *and it is weighted with the minimum delay cycles given by* $delay(e)$ *the instruction* v_j *can be started after* v_i*.*

The dependencies between instruction v_i and v_j, $i < j$, can be further categorized into the following kinds [135]:

Data dependence: v_i writes to a resource read by v_j. Consequently, v_i must be scheduled before v_j. This dependency is also referred to as *read after write* (RAW) dependency and is also the most common type.

Anti-dependence: v_j reads a storage location written by v_k with $k \neq i$ that is overwritten by v_i. Thus, in a correct schedule, v_j reads the value defined by v_k before v_i overwrites it. This is also known as *write after read* (WAR) dependence. Since this is often the result of instructions that write results late in the pipeline while others read the result early in the pipeline, the associated delay is usually negative.

Output dependence: v_i and v_j write to the same storage location. A valid schedule must perform the writes in their original order, i.e., the storage location contains the result of v_j after executing both instructions. This dependency is also denoted as *write after write* (WAW) dependency.

Control dependence: Determines the ordering of v_j with respect to a branch instruction v_i so that v_i is executed in correct program order and only if it should be. Thus v_j is not executed until the branch destination is known. Generally, this kind of dependency can also be seen as a data dependency on the *program counter* (PC) resource.

Note that the *read after read* (RAR) dependency is not considered a data hazard.

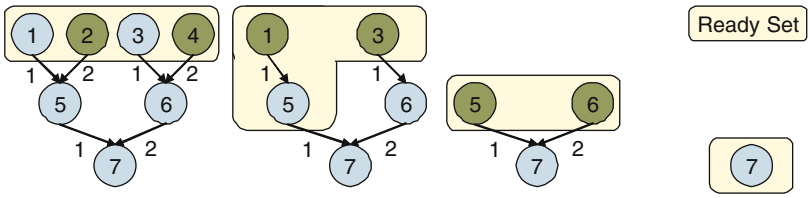
Since an instruction v_i may take several cycles until its result becomes available to v_j, it is the scheduler's task to fill these so-called *delay slots* with useful instructions instead of no-operations (NOPs). Given a dependency graph, a valid schedule is obtained with a mapping function S that assigns each node $v \in V$ a start cycle number c, $c \in \mathbb{N}$, such that

1. $S(v_i) + delay(v_i) < S(v_j)$ to guarantee that no dependencies are violated.
2. $r(v_i) \cap r(v_j) \neq \emptyset$ to avoid structural hazards.

The goal is now to find a schedule S_{opt} that needs the fewest number of cycles to execute. Let I denote the set of available machine instructions, then the length $L(S)$ of a schedule S can be described as follows:

$$L(S) = \max(S(v) + \max(delay(v, w))), \quad \forall v \in V, w \in I \tag{3.6}$$

The worst-case delay makes sure that the results are definitely available before instructions of potential successor basic blocks are executed. Unfortunately, computing the optimal schedule S_{opt} is an NP-complete problem. Several heuristics are in use for scheduling whereas *list scheduling* [68] is the most common approach. This algorithm for local scheduling keeps a *ready set* that contains all instructions v which predecessors in the dependency graph have already been scheduled. The list scheduler selects an instruction from the ready set and inserts it into the schedule S. Afterward, the ready set is updated accordingly and the scheduler proceeds with the next instruction from the ready set. Different heuristics have been proposed to pick a node from the ready set since this strongly influences the length of the schedule. For instance, one heuristic picks the instruction on the current *critical path*. This path represents the theoretical optimal schedule length. Figure 3.8 shows an example using this heuristic.



Cycle	Slot1	Slot2
1	4	2
2		
3		
4		
5		

Cycle	Slot1	Slot2
1	4	2
2	3	1
3		
4		
5		

Cycle	Slot1	Slot2
1	4	2
2	3	1
3	5	6
4		
5		

Cycle	Slot1	Slot2
1	4	2
2	3	1
3	5	6
4	NOP	NOP
5	7	NOP

Fig. 3.8 List-scheduling example; note that two instructions are scheduled in each step

List scheduling has a worst-case complexity that is quadratic in the number of instructions to schedule. However, list scheduling is conceptually not effective in handling negative latencies (in case of anti-dependencies) and filling delay slots. A solution to this problem are *backtracking schedulers* [232]. Such schedulers can revert previous scheduling decisions to schedule the current instruction earlier if this is likely to be more advantageous.

The amount of parallelism that can be exploited within a single basic block is quite limited since it contains only a few instructions on average. This is especially a problem for loop bodies that constitute typically the hot-spots of a program. One way to increase the number of instructions in loop bodies is *loop unrolling*, i.e., duplicating the loop body while reducing the number of required iterations. Another possibility is a scheduling technique especially for loops, called *modulo scheduling* [47]. It is an algorithm for *software pipelining* loops [173], i.e., the overlapping execution of several iterations.

An algorithm for global scheduling is *trace scheduling* [130]. The basic idea is to jointly schedule instructions of frequently executed and consecutive basic blocks. The execution frequency of basic blocks has to be obtained by profiling. Such a sequence of basic blocks is called a *trace* and is considered as a single, large basic block. In this way, the opportunities for ILP exploitation are increased. However, since the basic block boundaries are neglected, undesired side effects may arise. In order to fix this, compensation code has to be inserted. Of course, this results in a significant code-size increase that constitutes the major drawback of this approach.

3.3.5 Code Emitter

The code emitter is the final phase of the compiler backend. It is responsible to write the result of the previous phases into a syntactically correct assembly program, typically in an output file. The data structure of the emitter is an emission table. Each row, sorted in increasing order, represents a clock cycle and each column

an instruction. The code emitter first fills the emission table using the clock cycle information determined by the scheduler. Thus, each row represents the instructions that are executed together. Afterward, the table is dumped row by row, where empty cells are replaced by NOP instructions. While this is straightforward for single issue architectures, i.e., the table has only one column, constructing instructions for ILP architectures is sometimes more difficult. Such architectures typically impose constraints on how the instructions can be combined to build a valid instruction word. Therefore, a *packer* is incorporated in the emitter that composes syntactically correct assembly instructions for a given row. The final executable is then build from the assembly file using an assembler and linker. Both are usually separate tools that run after the compiler.

3.4 Retargetable Compilers

The embedded domain is characterized by a large variety of processor designs. Obviously, designing a new compiler for every single one of them is too costly. Additionally, developing a compiler is a time-consuming task, and hence, it may become available too late to be really useful for the architecture exploration phase. In many cases, this results in a compiler architecture mismatch that makes it quite difficult for Compiler Designers to ensure good code quality. This has led to the development of retargetable compilers. Such compilers are capable of generating code for different hardware architectures with few modifications of its source code. Such compilers take a formal description, e.g., specified in an ADL, of the target architecture as input and adapt themselves to generate code for the given target. The retargetability support mostly needs to be provided for code selector, scheduler, and register allocator, i.e., the compiler backend (Fig. 3.9).

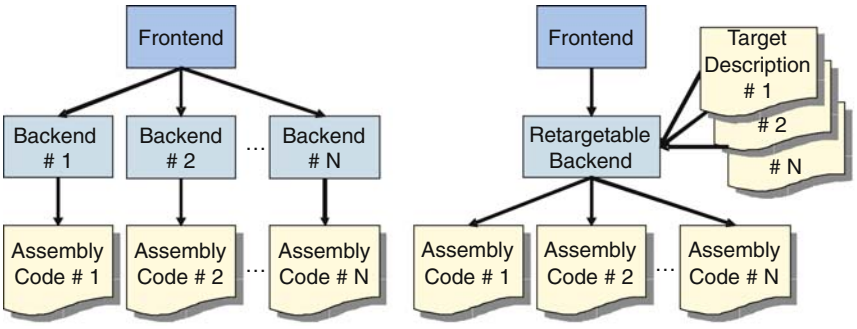


Fig. 3.9 Non-retargetable vs. retargetable compiler flow

Different degrees of retargetability exists to achieve this goal. According to the classification in [219], compilers can be assigned to one of the following classes:

Parameterizable: Such compilers can only be retargeted to a specific class of processors sharing the same basic structure. The compiler source code is largely fixed. The machine description only consists of numerical parameters such as register file sizes, word lengths, the number of functional units, or different instruction latencies.

User retargetable: An external machine description given in a dedicated language contains the retargeting information. All information required for code generation is automatically derived from this description. The specification does not require in-depth compiler knowledge, and hence can be performed by an experienced user.

Developer retargetable: Retargeting is also based on an external target description. However, the specification requires extensive compiler expertise usually possessed only by very experienced users or Compiler Designers.

A retargetable compiler has to be as target-independent as possible in order to be applicable for a wide variety of processor types. As a result, such compilers can only make few assumptions about the target machine, i.e., less target-specific hardware features can be exploited to produce efficient code. Hence, one of the major challenges in the design of retargetable compiler is to provide good optimizations for a wide variety of target architectures. Therefore, many retargetable compilers are limited to a certain processor class in order to ensure good code quality. New retargetable optimization techniques offer a solution to extend the range of target processor. This is further discussed in Chapters 8 and 9. Typically, retargetable compilers are limited to one of the following processor classes:

General purpose processors (GPPs): GPPs are characterized by an universal instruction-set architecture that provides a high degree of flexibility. As a result, they achieve good performance for a wide variety of applications. Unfortunately, this comes usually at the expense of a higher power consumption that makes them pretty much unusable for the embedded domain. Instead, such processors are widespread in desktop or portable PCs. Prominent examples for this class are MIPS [179], ARM [41], and the well-known Intel x86 architectures [122].

Very long instruction word processors (VLIW): This architecture is designed to exploit ILP that comes along with very high performance. Several functional units can be executed in parallel, whereas each unit is related to a specific field in the instruction word. Since such processors do not feature dedicated scheduling hardware such as superscalar architectures, the compiler is responsible for exploiting the ILP that might be present in the given applications. Representative examples of this processor class include the TriMedia and Nexperia architectures [190], the Embedded Vector Processor [152], and the ST200 [84].

Digital signal processors (DSPs): DSPs have been specifically designed for signal-processing applications. Consequently, their instruction-set supports dedicated instructions for the efficient execution of common signal-processing computations, such as fast Fourier transformation (FFT) or digital filtering. Additionally, such processors usually feature hardware multipliers, address generation units (AGUs), and zero overhead loops. Typical DSP examples are the TI C5x and C6x [259], the ADSP 2101 [42], and the MagicDSP [70].

Micro-controllers: Micro-controllers operate at clock speeds of as low as a few MHz and are very area efficient. The processor core implements a complex instruction-set computer (CISC) architecture. The chip typically integrates additional elements such as read-only memory (ROM) and random access memory (RAM), erasable programmable ROM (EPROM) for permanent data storage, peripheral devices, and input/output (I/O) interfaces. They are frequently used in automatically controlled products and devices, such as engine control systems, remote controls, office machines, and appliances. Examples for this kind of architecture are the Motorola 6502 [181] and the Intel 8052 [122].

Application specific instruction-set processors (ASIPs): ASIPs show highly optimized instruction-sets and architectures, tailored for dedicated application domains such as image processing or network traffic management. In this way, they achieve a good compromise between flexibility and efficiency. Examples of this kind are ICORE [251], SODA [281], a channel decoder architecture for third-generation mobile wireless terminals [78], and an ASIP for Internet Protocol Security (IPSec) encryption [109].

Some prominent retargetable compilers primarily for GPPs are *gcc* [87] and *lcc* [50]. Trimaran [263] and IMPACT [57] are examples for retargetable compilers for VLIW architectures. Other examples include CoSy [38], LANCE [222], SPAM [247], and SUIF [249]. Some of them constitute a key component of the ASIP design environments discussed in Chapter 4. A comprehensive survey of retargetable compilers can be found in [224].

3.5 Synopsis

- Compilers can be coarsely separated into a frontend and a target-specific backend (code selector, scheduler, register allocator).
- Retargetable compilers can be quickly adapted to varying processor configurations.
- Such compilers are capable of generating the backend components from a formalized processor description (e.g., an ADL model).

Chapter 4
Related Work

In general, ADL design must trade-off the level of abstraction vs. generality. ADLs must capture a wide variety of embedded processors with ever-changing irregularities. On the one hand, a lower-level description captures structural information in more detail, but on the other hand the detailed description makes it difficult to extract certain information such as instruction semantics for instance. Obviously, this is easier using higher-level descriptions; however, they make the generation of, e.g., cycle-accurate simulators inherently difficult. Over the past decade, several ADLs have emerged, each with their own strengths and weaknesses.

In this chapter, the related work in the field of ADL-based ASIP design is discussed.

4.1 Instruction-Set-Centric ADLs

nML: The nML language [161] was originally proposed by the Technical University of Berlin. It is one of the first ADLs to introduce a hierarchical scheme to describe instruction-sets. The topmost elements of the hierarchy represent instructions, and elements lower in the hierarchy are partial instructions (PIs). Two composition rules can be used to group the PIs in their parents: the AND-rule groups several PIs into a larger PI and the OR-rule enumerates alternative PIs corresponding to an instruction. For this purpose, the description utilizes an attribute grammar [136].

Though classified as instruction-set-centric language, nML is not completely free of structural information. For instance, storage units such as registers or memory must be explicitly declared. Furthermore, it is assumed that each instruction is executed in one machine cycle; there is no pipeline modeling. The language is used by the instruction-set simulator called SIGH/SIM [8] and the retargetable code generator CBC [9, 162]. It is also used by the instruction-set simulator CHECKERS [273] and the code generator CHESS [69] developed at the IMEC institute [118]. These tools have later been commercialized and are now available from Target Compiler

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_4, © Springer Science+Business Media, LLC 2010

Technologies [256]. Their tools include support for pipeline modeling and feature an HDL generator. They have successfully been employed for several DSPs and ASIPs. Recently, enhanced support for instruction predication has been added to the optimizing C compiler component of the Chess/Checkers tool-suite.

Another development branch, called Sim-nML [227, 268], has been started by the Indian Institute of Technology and Cadence Inc. The enhancements include support for pipeline modeling, branch prediction, and hierarchical memories. The generated software tools include an instruction-set simulator supporting interpretative and compiled simulation, assembler, and a code generator [165]. Additionally, a tool called Sim-HS is available that implements high-level behavioral and structural synthesis of processors from their Sim-nML specifications [236].

The nML-based simulators are known to be rather slow. Target, however, claims to have faster instruction-accurate simulation techniques which achieve a simulation speed that is over 100 times faster than conventional cycle-accurate simulators. However, no results have been published yet.
Since nML models constraints between operation by enumerating all valid combinations, the resulting description can be quite lengthy. Furthermore, VLIW processors or DSPs with irregular ILP constraints are – if at all – hard to model with nML.

ISDL: The acronym stands for Instruction Set Description Language [98]. It was developed at the Massachusetts Institute of Technology (MIT) to assist hardware–software co-design of VLIW architectures. Similar to nML, ISDL uses an attribute grammar for the instruction-set description and storage elements such as registers are the only structural information defined for each architecture. However, in contrast to nML, which captures all valid instruction compositions, ISDL employs boolean expressions to define invalid combinations. This often results in a simpler constraint specification and allows to model much more irregular ILP constraints.

ISDL is used by the Aviv compiler [238] as well as the related assembler and linker [97]. The Aviv compiler, which is based on the SUIF [249] and SPAM [247] compiler infrastructure, supports phase-coupled code generation that offers certain advantages over strictly separated code-generation phases. However, since a large number of heuristics need to be employed to cope with the overall complexity, the optimality is at least questionable. So far, only results for artificial VLIW processors have been reported. Hence, it is not entirely clear how Aviv performs for more irregular real-life embedded processors.

Moreover, ISDL is used by the retargetable simulator generation system GENSIM and a synthesizable HDL code generator [96].

CSDL: The Computer System Description Language (CSDL) is actually a family of machine description languages for the Zephyr compiler environment [4]. It has mainly been developed at the University of Virginia and consists of the following languages:

- The Specification Language for Encoding and Decoding (SLED) [187] describes instruction syntax and binary encoding and is used to retarget an assembler, disassembler, and linker. SLED is flexible enough to describe RISC and CISC computers. However, there is no notation of hardware resources nor explicit constraints for instruction compositions. As a result, SLED is not suitable for VLIW description.
- For the description of instruction semantics, the register transfer list (RT-list) language λ-RTL [186] is used. It is based on Standard-ML [226] and was mainly developed to reduce the description effort for Zephyr's very portable optimizer (VPO) [158]. VPO provides instruction selection, instruction scheduling, and classical global optimization. Unfortunately, VPO needs quite verbose RT-lists for the instruction-set description as input. Therefore, λ-RTL is translated in RT-lists instead of retargeting VPO. However, irregular architecture features such as special-purpose registers, complex custom instructions, and ILP constraints are hard to model.
- The Calling Convention Specification Language (CCL) [156] is used to define procedure-calling convention for uniform procedure call interfaces, i.e., how parameters and return values are passed between function calls. This information is required by the compiler as well as the debugger.

A drawback, though, is that all these descriptions must be kept consistent to ensure correctness. Furthermore, due to the limitation mentioned above, CSDL is more suited for conventional general-purpose or regular RISC/-CISC processors. Embedded processors with architectural irregularities or VLIW architectures usually cannot be modeled at all. So far, results for HDL generation have not been reported yet.

Valen-C: Valen-C [19, 20] is a C language extension to support explicit and exact bit-width specification for integer data types. The retargetable compiler Valen-CC takes an application written in Valen-C and a description of the instruction-set as input. It produces code only for RISC architectures. The instruction-set description represents only the instruction-set, i.e., pipelines or resource conflicts are not modeled. A separate description is used for simulator retargeting.

One commonality of all these languages is the hierarchical instruction-set specification using attribute grammars. In this way, common properties of instructions can be easily factored out which simplifies the instruction-set description to a large extent. Instruction semantics for compiler generation can be easily extracted due to the explicit specification in the form of RT-lists. On the other hand, such languages do not contain detailed pipeline and timing information. This makes it inherently difficult to generate cycle-accurate simulators and, to a certain extent, instruction schedulers. This can only be circumvented by limiting the architectural scope of the language so that certain assumptions about the target architecture can be made. Moreover, since this kind of ADL does not contain any or only limited structural

information, either the generation of synthesizable HDL code is not supported or the quality of the generated HDL code is not satisfactory.

4.2 Architecture-Centric ADLs

MIMOLA: The Machine-Independent Microprogramming Language (MIMOLA) [235] is an example for a register transfer level (RT-level)-based ADL, developed at the University of Dortmund. It was originally intended for micro-architecture design. A MIMOLA description mainly consists of two parts: the hardware part with a netlist of component modules and the software part describing the applications in a PASCAL-like syntax.

Several tools based on the MIMOLA language have been developed [204], including the MSST self-test program compiler, the MSSH hardware synthesizer, the MSSB functional simulator, the MSSU RT-level simulator, and the MSSQ code generator. A single MIMOLA model serves as input for all these tools.

Since pipelined targets cannot be modeled with MIMOLA, the architectural scope is mostly limited to architectures with single-cycle instructions. Furthermore, the MSSQ compiler produces sometimes poor code quality and suffers from high compilation times. The RECORD compiler [223] constitutes the successor of MSSQ and eliminates some of these limitations. It generates better code quality; however, it is restricted to the class of DSP architectures. Another limitation is the missing C frontend, only the data-flow language SILAGE [65] is supported.

AIDL: The AIDL language [254] introduces several levels of abstraction to model a processor. It has been designed to describe time relations such as concurrency and cause/effect relations between pipeline stages in a simple and accurate way. The concept of timing relations is based on interval temporal logic [46]. Each behavior is described using a so-called stage that corresponds usually to a pipeline stage. Sequentiality and concurrency is specified within or between stages. So far, AIDL was only employed to model three processors, which are all based on the PA-RISC instruction-set architecture [114].

As described in [253], it is possible to generate a synthesizable HDL code and a simulator from an AIDL specification. So far, support for compiler, assembler, and linker generation is not available.

UDL/I: The UDL/I language [264] is also an RT-level hardware description language, but in contrast to MIMOLA mainly intended for compiler generation. It is used as an input for the COACH ASIP design environment [107], which extracts the instruction-set from the UDL/I description. However, this process imposes some restrictions on the class of supported architectures. In particular VLIW architectures are not supported. The generated software tools include an instruction-set and cycle-accurate simulator.

In general, RT-level ADLs are more intended for hardware designers. They provide concepts for a detailed specification of the micro-architectures in a flexible manner. Several approaches have proven that based on a single ADL model, design automation tools for logic synthesis, test generation as well as retargetable compilers and simulators can be generated. However, from a Compiler Designer's perspective, all information regarding the instruction-set is buried under an enormous amount of micro-architectural details. Thus, extracting the semantics of instructions automatically is quite hard, if not impossible, without restrictions on description style and supported target architectures. Furthermore, considering that merely describing a processor at the RT-level alone is a tedious task, quick modifications as required for efficient architecture exploration are self-prohibitive. Moreover, the simulators generated from such ADLs are known to be rather slow [150].

4.3 Mixed-Level ADLs

Maril: The Maril language is the description format for the retargetable compiler Marion [61]. A Maril description contains both instruction-set description as well as coarse-grained structural information. However, it does not employ a hierarchical scheme for instruction-set-specification such as instruction-set-centric languages. On the other hand, it contains more structural information than those languages. This enables the generation of resource-based schedulers that can yield significant performance improvements for deeply pipelined processors. Unfortunately, the instruction behavior must be described with a single expression that can only contain a single assignment. While this is sufficient for compiler generation, it generally provides not enough information for accurate simulation. For instance, additional side effects of instructions (e.g., affecting condition code registers or flags) cannot be described.

Maril is mainly intended for RISC processors, describing VLIW processors is not possible. Moreover, it does not contain any information about the instruction encoding. Thus, retargeting an assembler or disassembler is not possible.

MESCAL/MADL: The Mescal Architecture Description Language (MADL) employs an Operation State Machine (OSM) [275] computational model to describe the operations. As the name implies, it was developed within the Mescal [149] group of the Gigascale Silicon Research Center (GSRC) [104]. An OSM specification basically separates the processor into two interacting layers. The operation layer models operation semantics and timing, whereas the hardware layer describes the micro-architecture. The target scope includes scalar, superscalar, VLIW, and multi-threaded architectures. The approach emphasizes on simulator generation, other software development tools are not generated. Successful case studies are reported for the StrongARM [73] and the PowerPC-750 [184].

The study claims that instruction schedulers can be retargeted as well but no results in this regard have been published yet. Meanwhile, OSM has been successfully employed to model on-chip communication architectures, which allows to generate cycle-accurate simulators for multiprocessor SoCs [277].

HMDES/MDES: The HMDES language [133] constitutes the input for the IMPACT research compiler [57, 200] developed at the University of Illinois. IMPACT has been designed to efficiently explore wide-issue architectures that offer lots of scheduling alternatives for instructions. Consequently, the definition of instruction's reservation tables is a central notion in HMDES. However, information about instruction semantics, assembly syntax, or encoding information are missing in HMDES. This is a result of IMPACT being not designed as a fully retargetable software development tool chain. Basically, IMPACT is an EDG [76] based optimizing C frontend. Apart from standard optimizations [1], IMPACT supports some new concepts for ILP exploitation based on extended basic blocks notations [59, 216] and predicated execution [243].

The MDES machine description format of the Trimaran compiler infrastructure [263] also uses an HMDES description as input. Trimaran incorporates IMPACT as well as the Elcor research compiler [233] from HP Labs. Initially, the compiler could only be retargeted to a single class of processors, called HPL-PD [267]. Architectural parameters include mainly ILP-related options such as the number of registers, instruction latencies, instruction word length, and the number of available functional units and their scheduling constraints. Meanwhile, it has also successfully been retargeted to the ARM [40] and the WIMS processor [225].

Trimaran is also employed in the Program in Chip Out (PICO) [266] system for the automatic design of custom processors. Such processors consists of a configurable VLIW template (i.e., HPL-PD based) [48] and a nonprogrammable processor (a one- or two-dimensional array of processing elements) [228].

EXPRESSION: The EXPRESSION language [14, 201, 205] was developed at University of California at Irvine. An EXPRESSION description consists of a distinct behavioral and structural section. The behavioral section is similar to ISDL, but it is missing the assembly syntax and binary encoding. The specified operations can be bundled to instructions in order to model VLIW architectures. Additionally, all operations must be manually mapped to generic compiler operations in order to enable compiler generation. The structural section directly describes a netlist of pipeline stages and storage units to automatically generate reservation tables required by the scheduler based on the netlist [202]. However, HDL models cannot be generated yet.

An EXPRESSION specification is used by the simulator SIMPRESS [22] and the retargetable compiler EXPRESS [12]. All tools are integrated into a visual environment called V-SAT. So far, the modeled architectures

include: ARM7 [40], SPARC [248], TI C6x [259], DLX [135], Renesas SuperH SH3 [231], and Motorola 56k DSP [182].

IDL: The instruction description language (IDL) [211] is used by the FlexWare2 system [207–209]. The environment is the successor of FlexWare [210] developed at STMicroelectronics. IDL is used in conjunction with the ISA database called Flair which drives the entire Flexware system. It consists of the CoSy [38] based FlexCC compiler, assembler, linker, the simulator FlexSim, the debugger FlexGdb, and the FlexPerf profiler.

The generated code quality is reported to be close to hand-crafted assembly code. However, the target description contains a large amount of redundancies, and hence requires a significant verification effort to be kept consistent. Furthermore, FlexWare is intended for in-house use only.

RADL: The Rockwell architecture description language (RADL) [56] is a follow-up of the first version of the Language for Instruction-Set Architecture (LISA) [15]. It focuses on explicit support of detailed pipeline behavior to enable the generation of cycle- and phase-accurate simulators [155], other software tools are not generated. However, so far nothing has been published about the simulators using RADL.

Mixed-level ADLs basically extend instruction-set-centric languages by including structural information. So far, this is mainly used to enable the generation of fast cycle-accurate simulators and instruction schedulers. The retargeting of the compiler's code selector has mostly either to be performed manually or is more or less fixed due to a predefined processor template. Furthermore, support for HDL generation is usually not implemented.

4.4 Other Related Approaches

ASIP Meister: The ASIP Meister environment, formerly known as PEAS-III [23, 169], was jointly developed by the Semiconductor Technology Academic Research Center and the Osaka University. It is an enhanced version of the PEAS system [137, 138], capable of generating a synthesizable hardware description and the complete software development tool chain, i.e., a CoSy-based C compiler [239], assembler, linker, and simulator. Additionally, it provides estimates for power consumption, maximum clock frequency, and silicon area.

ASIP Meister has no uniform ADL. It is basically a graphical user interface (GUI) used to model the architectures using functional blocks defined in a so-called flexible hardware model (FHM) library [170]. Each block is associated with behavior, RT-level, and gate-level information. Unfortunately, the library is not user-extensible, which limits the architectural scope. Furthermore, for compiler generation, the semantic for each block must be manually specified. So far, successful designs have been reported

for the DLX [135] and the MIPS-R3000 [179], even though the complete instruction-set architecture could not be implemented in both cases.
Based on the ASIP Meister environment, a platform for synthesizable HDL generation of configurable VLIW processor was developed [280]. However, no information is available regarding the software tool generation for this architecture class.

UPFAST/ADL: The UPFAST [245] system automatically generates a cycle-accurate simulator, an assembler, and a disassembler from a micro-architecture specification written in the Architecture Description Language (ADL). So far, it has only been successfully deployed for several artificial targets based on the MIPS ISA. The speed of the generated simulator is reported to be two times slower than a hand-crafted version.

PROPAN/TDL: The Target Description Language (TDL) is used in the retargetable postpass assembly optimization system PROPAN [66], developed at Saarland University. Basically, an assembler parser as well as a set of C files are generated from a TDL description. The C files can be included in applications to provide a generic access to architectural information. However, the architectural scope is mostly limited to VLIW DSPs.

BUILDABONG: The BUILDABONG [139, 140] is intended to aid the design of special computer architectures based on architecture and compiler co-generation. The input of this tool is an abstract state machine (ASM) model of the target architecture. It is either derived from an XASM description or given by a schematic tool entry. BUILDABONG supports the generation of HDL models, simulator, and compiler. The user must specify the instruction-set and the code generator generator's grammar in a GUI called Compiler Composer, which finally generates the compiler executable [63]. The machine model is automatically extracted from the graphical architecture description and converted to an extensible markup language (XML)-based description, called machine markup language (MAML) [64]. This description is used by the MAML compiler and constitutes the input for the Compiler Composer. So far, only artificial architectures have been used as case studies. Future developments will focus on complex architectures such as the TI C6x family and reconfigurable ASIPs. However, results regarding the simulation speed, code quality, and the exact architectural scope have not been reported yet.

Liberty: The Liberty Simulation Environment (LSE) [177] models processors by connecting hardware modules through their interfaces. These modules are either predefined or parameterizable. From this specification, given in the liberty structural specification (LSS) language [176], a cycle-accurate simulator is generated. Since Liberty does not provide the facility for capturing the instruction behavior and binary encoding, it is not suited to create software development tools.

Babel: The Babel [274] language was originally intended for the specification of nonfunctional IP blocks. However, the corresponding integration framework retargets a set of GNU tools [87] (more specifically, the binary

utilities) to integrate different IP cores. Obviously, this is limited to the architectures supported by the GNU tool chain. The employed architectures include SPARC [248], SimpleScalar [246], and Alpha [218]. Babel is also utilized to retarget the SimpleScalar simulator [62].

MADE: The modular VLIW processor architecture and assembler description environment (MADE) [206] generates a library of behavioral functions and the instruction-set of the machine from the related architecture description. The library is then linked to a reconfigurable scheduling engine which results in a configured optimizer–scheduler. The automatic configuration of a cycle-accurate simulator is under development. So far, this environment is only used for the MagicDSP [70].

ARC: The *ARCtangent* processor family [43] from ARC Inc. is a RISC/DSP architecture with a 32-bit four-stage pipeline. Each core can be extended by predefined modules such as floating-point support, advanced memory subsystem with address generation, and special instruction extensions for common DSP algorithms. The basic ISA implements 86 mixed 16/32-bit instructions, which can be extended to a certain extent by custom instructions. A graphical user interface (GUI), called ARChitect, allows the designer to select between the given configuration options and to specify the custom instructions. Additionally, the environment provides a simulator, a real-time operating system (RTOS), and a C/C++ compiler. However, the instruction-set extensions cannot be directly exploited by the compiler. Instead, the programmer is forced to use assembly-like function calls (compiler intrinsics) or inline assembly that reduces the reusability to a great extent.

Tensilica: The *Xtensa* architecture [215] from Tensilica Inc. [257] offers a large number of configurable or user-defined extensions that can be plugged in to the processor core. The base architecture has 80 RISC instructions and includes a 32-bit ALU and 32 or 64 general-purpose 32-bit registers. Among the configurable options are DSP engines, floating-point support, the memory interface, and caches. Custom instructions for application-specific performance improvements can be specified using the Tensilica Instruction Extension (TIE) language. The software tools consist of a (GNU-based) C-compiler, assembler, linker, simulator, and a synthesizable HDL model. Tensilica reports a 20% performance improvement of the *Xtensa C/C++ Compiler* (XCC) as compared to a regular *gcc* compiler. The compiler also supports custom instructions and vectorization to a certain extent.

Others: A quite recent ADL mainly designed for compiler generation is presented in [79, 237]. The syntax is based on XML. On the compiler side, an earlier version relies on the Open Compiler Environment (OCE) from Atair. The current version uses an extended *gcc*-frontened (for Embedded-C [77]) and a custom backend. Up to now, the language has been used to model the VLIW DSPs xDSPcore [25] and CHILI [197] as well as the MIPS-R2000 [179] processor. The description contains enough information to enable the

generation of other tools such as simulator, assembler, and linker. However, nothing in this regard has been published yet.

Other existing ADLs include ISPS [157], ASIA/ASIA-II [115, 116], ASPD [278], EPICS [230], READ [279], and PRDML [31].

Further approaches employing parameterizable generic processor cores include JazzDSP [54] and DSP-core [172].

Several tools choose a different route for implementation. They directly generate a synthesizable HDL model or hardware implementations from the given application. Examples for this approach are ARTBuilder [39] or the PACT HDL compiler [21]. The major drawback, of course, is the limited flexibility of the generated hardware.

Quite a large amount of ADLs are already available, and it is reasonable to expect more new ADLs or at least ADL extensions. The effort to develop a new ADL from scratch or to undertake the tedious task of modifying an existing language has led to a new kind of ADL. Such ADLs are based on XML. In this way, a standard to encode the elements of an architecture is provided. This saves development time and makes the model reusable and interchangeable between tools. Examples for this kind are ADML [260] and xADL [74]. However, ASIP design environments using these languages are not yet known.

Configurable processor cores, ADLs with a limited architectural scope (such as DSP or VLIW), or ADLs designed for a specific purpose (e.g., simulator or compiler generation) are mostly capable of generating an efficient set of tools and hardware. The advantage of a limited architectural scope, in particular in case of configurable cores, is the reduced verification effort, though, at the expense of a limited design space. In contrast, a broader architectural scope results in increased verification effort but allows a larger design space. Corresponding ADLs must be suitable for a wide variety of architectures while at the same time providing design automation for all ASIP design phases. Such ADLs usually require sophisticated algorithms to generate high-quality software tools and hardware as compared to domain-specific or tool-specific ADLs.

All recent ADLs belong to the mixed-level class. They are well suited to meet these demands and they have been successfully employed in academic research as well as in industry. Unfortunately, these ADLs are either bound to a predefined processor template and hence suffer from limited flexibility, or do not support the generation of all software development tools and corresponding HDL model. While the generation of simulators is mostly supported, compilers, in particular the code selector description, must still be retargeted manually. This process requires significant compiler knowledge and delays the availability of a C compiler for early architecture exploration. Thus, to further lower the entry barrier to compiler generation and to reduce the time-consuming and tedious manual effort, the *automatic generation of code selector descriptions* is of strong interest. This has been the main motivation to implement a methodology for code selector generation from ADL processor models without sacrificing their flexibility. This book presents an

approach that is based on the LISA ADL. The next chapter briefly introduces the corresponding design environment.

4.5 Synopsis

- Abstract processor modeling is established as an efficient solution for an ASIP design.
- Regardless of ADL implementations, a significant gain for an ASIP design in terms of development time over the classical ASIP design approach is achieved.
- Due to the difficulty designing an ADL that in particular supports the generation of the complete software tool chain (in particular compiler *and* simulator) current ADLs sacrifice flexibility, introduce redundancies, or support only the generation of particular software tools.
- Certain compiler-relevant information (e.g., scheduler tables) can already be extracted from ADL descriptions, while others (code selector description) must still be provided manually.
- An ADL-based design environment that supports the automatic generation of *all* software development tools while keeping its flexibility is proposed in this book.

Chapter 5
Processor Designer

In this book, the *Language for Instruction-Set Architectures* (LISA) ADL is used and extended for automatic generation of C compilers. LISA is the key component of the *Processor Designer* ASIP design environment, formerly known as the LISA processor design platform (LPDP) [15, 16]. It was initially developed at the Institute for Integrated Signal-processing Systems at the RWTH Aachen University [119] and is now commercialized by CoWare Inc. [58]. The LISA design methodology can be considered as one of the most powerful and comprehensive ADL-based design platform available today and is also well recognized by academia and industry. It enables an efficient design space exploration to tailor a processor architecture to the intended range of applications. During the process, the micro-architecture, instruction-set, register, and memory configuration are investigated and optimized.

The LISA-based design space exploration and the related tools are briefly introduced in the following sections. Afterward, the LISA language as far as relevant to understand the compiler generation techniques presented in this book is introduced in the next section. A detailed overview about LISA and the generated software development tools is given in [15]. Finally, Section 5.3 describes the current tool flow for C compiler generation.

5.1 Design Space Exploration

As illustrated in Fig. 5.1, a single LISA processor description drives all ASIP design phases: *architecture exploration*, *architecture implementation*, *software tools generation*, and *system integration* (see Section 2.1). Using the LISA language, changes to the processor architecture can be quickly modeled. In this way, an efficient exploration of the architectural design space is ensured.

5.1.1 Software Tool Generation

The *Processor Designer* provides an integrated design environment (IDE) to support the manual creation and configuration of the LISA model. From the IDE the

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_5, © Springer Science+Business Media, LLC 2010

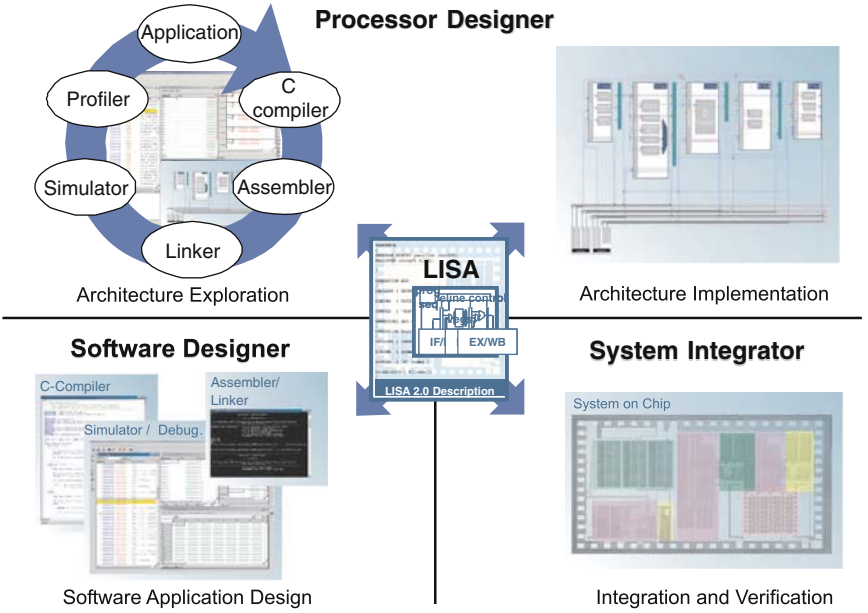


Fig. 5.1 LISA Processor Designer

so-called *LISA processor compiler* is invoked. It parses the description and builds the software development tools based on a set of retargetable software development tools.

The assembler is retargeted to the specialized instruction-set. It processes assembly files and produces the object code for the target architecture. Additionally, a macro assembler is provided for user convenience. The automatically generated linker then combines several object files to a single executable in the ELF format [262]. The linker has to be adapted to the target-specific memory organization. More detailed information about the assembler, macro assembler, and linker can be found in [18].

The generated simulator basically can be split into frontend and backend. The former supports the typical functionality such as disassembly, loop and execution profiling, and pipeline utilization. It provides all profiling information as required for design space exploration. The backend supports various kinds of simulation techniques, such as interpretative simulation, compiled simulation [90], and just-in-time (JIT) [27, 28] simulation. As shown in [92], the performance of the generated simulators strongly depends on the abstraction level of the LISA model and the accuracy of the memory model.

A CoSy-based C compiler is manually retargeted via a graphical user interface (GUI) [168], called *Compiler Designer* (see Section 5.3). Instruction schedulers, though, can already be automatically generated [195].

5.1.2 Architecture Implementation

LISA also supports an automatic path to generate the hardware description on RTL. For this purpose, synthesizable HDL code, such as VHDL or Verilog, can be generated fully automatically [192]. In this way, the impact of high-level design decision on the physical characteristics of the architecture can be investigated. For instance, the information about clock speed influences the number of pipeline stages or the pipeline structure in general. Ignoring this feedback leads to suboptimal design decisions and long redesign cycles in the worst case.

The RTL hardware model synthesis is based on the so-called unified description layer (UDL) [191]. This enables the integration of optimizations to ensure a sufficient architectural efficiency and transformation to integrate processor features such as JTAG interface and debug mechanism. Several case studies demonstrated that the physical characteristics of the generated processors are comparable to handwritten implementations [193].

5.1.3 System Integration

Current SoC designs are characterized by a mixture of different programmable processors, ASICs, memories, etc. combined with a complex communication architecture. This requires system simulation for verification and performance evaluation in the system context. The LISA simulators can be easily integrated into a co-simulation environments, such as CoWare ConvergenSC [58] or Synopsys's System Studio [250], using a set of well-defined interfaces.

Different levels of abstraction are supported to model the communication between an ASIP and its system environment. A generic interface allows to model arbitrary interfaces while the LISA bus interface allows to model the communication on a higher level of abstraction, e.g., TLM [32, 34], using standard SoC communication primitives. Special LISA pin resources can be used, which are directly connected to the SoC environment for pin-accurate co-simulation.

The system simulation debugger offers a software-centric view to a multiprocessor SoC while providing the system context as well [33]. In [17], the integration of several LISA models into the SystemC [252, 261] environment is described. SystemC was used to model the processor's interconnection, external peripherals, memories, and buses on a cycle-accurate level.

Recently, LISA has been extended to support the emerging class of re-configurable ASIPs (rASIPs). Such architectures contain a fixed processor combined with a re-configurable block which can either be statically or dynamically re-configured. While the soft flexibility is already available in the form of the ISA, it can be further extended by additional instructions that are put in the re-configurable block. In this way, rASIP architectures can be easily extended to cover new application domains. The required tools for an efficient rASIP design space exploration can also be automatically generated from the extended LISA description [5, 6, 147].

5.2 The LISA Language

LISA belongs to the group of mixed-level ADLs. Hence, a LISA model captures the behavior, the structure, and the I/O interfaces of the processor architecture. All architectural information are captured in a single model. LISA has been successfully used to describe a broad range of architectures, including ARM9 [41], TriMedia [190], C54x [258], MIPS32 4K [180], and to develop ASIPs for different application domains [109, 251].

A LISA model basically consists of two parts: one part describes the available resources of the target architecture and the other contains the description of instruction-set, behavior, and timing.

Resource declarations specify a subset of the processor resources, namely registers, buses, memories, external pins, and internal signals. The resources can be parameterized w.r.t. signedness, bit-width, and dimension.

```
RESOURCE {
  MEMORY_MAP {
      RANGE(0x00100000, 0x002fffff) -> example_mem[(31..0)];
    }
  RAM unsigned char example_mem {
      SIZE(0x00250000);
      BLOCKSIZE(8,8);
      FLAGS(R|W|X);
    };
  REGISTER unsigned int GPR[0..127];
  PIPELINE pipe={FE ; DE ; EX ; WB };
  PIPELINE_REGISTER IN pipe {
      unsigned int src1,src2,dst;
  }
} ...
```

Listing 5.1 Resource declaration

Configuration items for the memories include size, accessible block size, endianess, etc. All resources are global to the LISA model, i.e., they can be accessed within any LISA operation.

Listing 5.1 shows a typical LISA resource declaration. In the example, a 2-MB memory area named `example_mem` is specified, which is mapped into address space starting at `0x100000`. Furthermore, the general-purpose register file named `GPR` with one hundred and twenty-eight 32-bit wide registers and a pipeline named `pipe` are declared. The pipeline stages are defined from left to right corresponding to the actual execution order. `PIPELINE_REGISTERS` define the storage elements between pipeline stages, here `src1`, `src2`, and `dst`.

The major part of a model consists of operations. An `OPERATION` is the basic element of the ISA description. Each instruction is usually distributed over several operations whereas each operation in turn consists of several so-called sections.

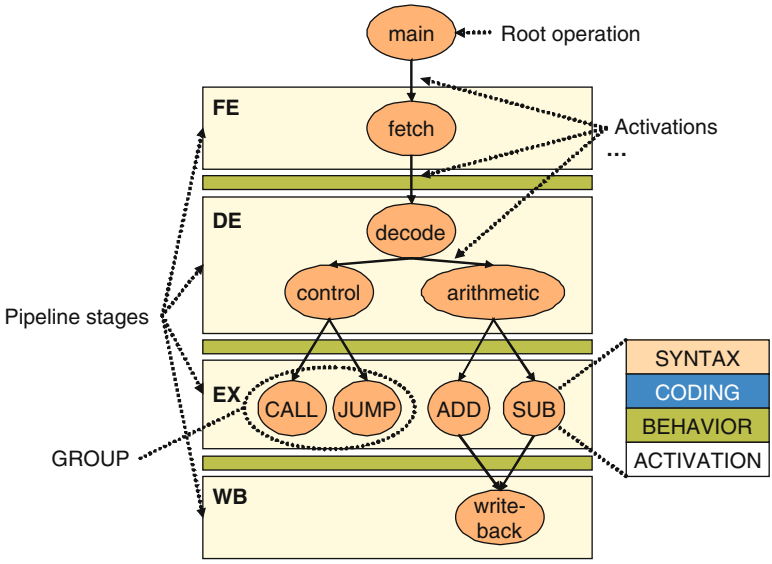


Fig. 5.2 LISA operation DAG

The CODING section describes the binary coding, the SYNTAX section the assembly syntax, and the BEHAVIOR section the operation's behavior.

Operations are organized hierarchically in order to factor out commonalities of instructions that reduce the description effort to a large extent. A modeled pipeline implies a cycle-accurate LISA model and hence each operation has to be assigned to one of the defined pipeline stages. Moreover, operations can trigger the execution of one or more child operations in the same or any later pipeline stage by so-called *activations* (via a dedicated ACTIVATION section) or *behavioral calls*. Again, operations can be activated or called from several different operations.

The resulting structure is a so-called LISA operation DAG $D = (V, E)$, where V denotes the set of all LISA operations and E the edges due to activations or behavioral calls. The root operation is the special `main` operation that is executed if the simulator advances one control step. Among others, this operation activates the operation fetching the next instruction from memory and advances the pipeline. Hence, a complete branch of the LISA DAG, also referred to as activation chain, and the related operations represent an instruction in the modeled target machine. Figure 5.2 gives an example.

The delay (in cycles) between two connected operations depends on the abstraction level. In the case of instruction-accurate models, operations are simply activated along the increasing depth of the LISA operation DAG, whereas in the case of cycle-accurate models, it is delayed until the activation advances to the stage related to the activated operation. Operations in the same pipeline stage are executed concurrently.

Listing 5.2 provides the specification for three of the operations in the example LISA operation DAG. More specifically, `arithmetic`, `ADD`, `SUB`, and `writeback`. They are assigned to the pipeline stages `DE`, `EX`, and `WB`, respectively.

Because ADD and SUB use the same type of operands (i.e., reg), the initialization of the operands can be factored out, and thus is modeled in the operation arithmetic. This relationship is given through the definition of GROUPs, whose members correspond to a list of alternative, mutual exclusive operations. The group name can then be referenced within the LISA sections, e.g., in the ACTIVATION section as depicted in the example. Here, all operations potentially referenced by opcode are located in pipeline stage EX, i.e., the execution is delayed until the subsequent cycle. The writeback operation is located in stage WB and, consequently, is two cycles delayed.

```
OPERATION arithmetic IN pipe.DE{
    DECLARE{
        GROUP opcode = { ADD || SUB || ... };
        INSTANCE rs1, rs2, rd = { reg };
        INSTANCE writeback;
    }
    CODING { opcode rd rs1 rs2 0b00}
    SYNTAX { opcode " " rd " " rs1 " " rs2 }
    BEHAVIOR{
        PIPELINE_REGISTER(pipe, DE/EX).src1 = GPR[rs1];
        PIPELINE_REGISTER(pipe, DE/EX).src2 = GPR[rs2];
    }
    ACTIVATION { opcode, writeback;}
}
OPERATION ADD IN pipe.EX{
    CODING { 0b00 }
    SYNTAX { "ADD"}
    BEHAVIOR{
        int op1 = PIPELINE_REGISTER(pipe, DE/EX).src1;
        int op2 = PIPELINE_REGISTER(pipe, DE/EX).src2;
        PIPELINE_REGISTER(pipe, EX/WB).dst = op1+op2;
    } ...
}
OPERATION SUB IN pipe.EX{
    CODING { 0b01 }
    SYNTAX { "SUB" }
    BEHAVIOR{
        int op1 = PIPELINE_REGISTER(pipe, DE/EX).src1;
        int op2 = PIPELINE_REGISTER(pipe, DE/EX).src2;
        PIPELINE_REGISTER(pipe, EX/WB).dst = op1-op2;
    } ...
}
OPERATION writeback IN pipe.WB{
    DECLARE{ REFERENCE dst; }
    BEHAVIOR{
        GPR[dst] = PIPELINE_REGISTER(pipe, EX/WB).dst;
    }
}
```

Listing 5.2 LISA operation hierarchy example

The SYNTAX describes the assembly syntax of the instruction. The syntax elements can be either terminal character sequences like “ADD” or a nonterminal. The later can correspond to a single INSTANCE of a LISA operation or a GROUP. The CODING section specifies the binary coding in a similar way using “0” and “1” as terminal elements. The behavior of a LISA operation is executed only if all terminal sequences and nonterminals (more specifically, single instances and at least one group member) match the actual decoded instruction.

The BEHAVIOR section implements the combinatorial logic of the processor. The LISA language allows arbitrary C/C++ descriptions of instruction behaviors, which achieves highest modeling flexibility. As mentioned above, if a pipeline is modeled, the C/C++ instruction behavior description is typically distributed over different pipeline stages. In the example, `arithmetic` reads the operands from the register file, stores them in the corresponding pipeline registers, and activates the operation currently referenced by `opcode`, so either `ADD` or `SUB`. These operations are executed in the following cycle. Accordingly, they combine the operand pipeline registers and store the result back into a pipeline register. Another cycle later the operation `writeback` writes the result back to the register file. For that purpose, the `dst` instance declared in `arithmetic` has to be referenced.

Apart from operation names, local variables can be declared and used in the BEHAVIOR section. Global processor resources and pipeline registers can be accessed as well. It is even possible to call external C/C++ functions or an internal LISA operation within the BEHAVIOR section (behavioral call).

5.3 Compiler Designer

The *Processor Designer* employs the CoSy system from ACE [38] for compiler generation. CoSy is a modular compiler generation system that offers numerous configuration possibilities both at the level of the *intermediate representation* (IR) and the backend for machine code generation. As illustrated in Fig. 5.3, CoSy is built around the *CoSy Common Medium Intermediate representation* (CCMIR) of the source program.

In general, a compiler is built by specifying a set of analyses and transformations, called *engines*, that annotate and modify the CCMIR. CoSy not only comes with a broad range of standard optimizations [1], but can also be easily extended with user-defined engines due to its modular concept. Each engine must exactly specify which elements of the IR it accesses using the *full-Structured Definition Language* (fSDL) [113]. The engine’s execution order is provided in a dedicated specification using the *engine description language* (EDL). From these pieces of information, a so-called *supervisor* is generated which schedules the engines and grants access to the CCMIR.

The *Backend Generator* (BEG) is the most important component of the CoSy system. It takes so-called *code generator description* (CGD) files as input and

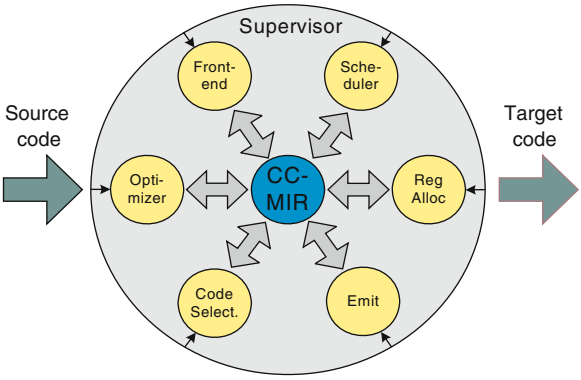


Fig. 5.3 CoSy compiler development platform

generates most of the backend source code automatically. A CGD model consists mainly of three components:

- A specification of available *target processor resources* such as registers or functional units.
- A description of *mapping rules* (cf. Section 3.3.2), specifying how C/C++ language constructs map to (potentially blocks of) assembly instructions.
- A *scheduler table* that captures instruction latencies as well as instruction resource occupation on a cycle-by-cycle basis.

Apart from that, CoSy requires some more information such as function calling conventions or the C data type sizes and memory alignment. A more detailed description of CoSy can be found in [36].

As depicted in Fig. 5.4, the *Compiler Designer* [168] basically extracts compiler-relevant information from a given LISA processor model and translates it to a corresponding CGD description. Afterward, CoSy can be invoked as a "backend" to

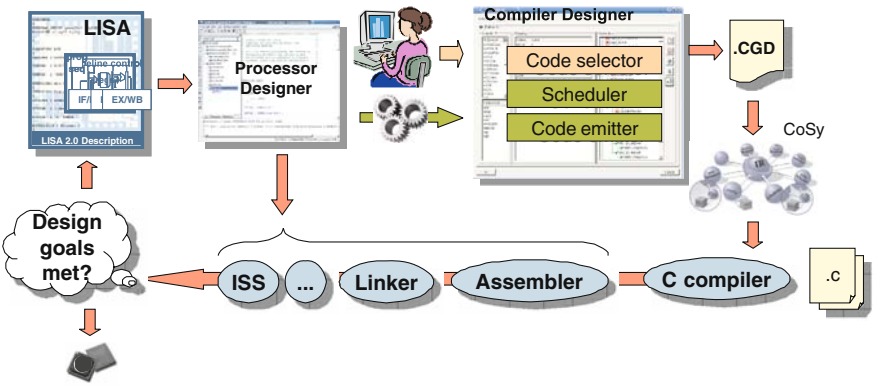


Fig. 5.4 Tool flow for retargetable compilation

generate the compiler executable. However, this translation is quite challenging due to a number of reasons: while some information is explicit in the LISA model (e.g., via resource declarations), other relevant information (e.g., concerning instruction scheduling) is only implicit and needs to be extracted by dedicated algorithms. Some further, heavily compiler-specific information is not at all present in the LISA model, e.g., C-type bit-widths. Additionally, compiler retargeting is further complicated by the *semantic gap* (cf. Section 6.1) between the compiler's high-level model of the target machine and the detailed ADL model that in particular must capture cycle and bit-true behavior of machine operations. This is discussed in Chapter 6 in more detail.

The *Compiler Designer* employs a *semi-automatic* approach for compiler generation. Compiler information is automatically extracted from LISA whenever possible, while GUI-based user interaction is employed for other compiler components. The *Compiler Designer* is organized in different configuration dialogs and the user is guided step by step through the specification of the missing items that could not be configured automatically or for further refinement of the generated items.

Data layout, register allocator, and calling conventions: Purely numerical parameters not present in the LISA model can be directly entered via GUI tables. This concerns mainly compiler-dependent data such as C-type bit-widths, type alignments, and minimum addressable memory unit size.

Configuration options for the *register allocator* include the selection of allocatable registers out of the set of all available registers in the LISA model. For instance, registers selected as frame or stack pointer need to be excluded from allocation. Another option regards those registers that cannot be temporarily saved in memory. Finally, some processor architectures allow the combination of several regular data registers to "long" registers of larger bit-width. The composition of long registers is also performed via the GUI.

The *calling conventions* basically describe the preferred passing of function parameters and return values. The GUI provides a convenient dialog to specify for each C data type the preferred passing method which can be either registers or stack.

Instruction scheduler: Instruction schedulers determine the sequence in which instructions are issued on the target processor. Besides structural hazards, data dependencies between instructions need to be taken into account (cf. Section 3.3.4). These constraints are captured by *scheduler tables* containing latency information for the different kinds of dependencies and the resource usage of instructions. These tables are generated fully automatically from the given LISA model [195]. Since the generator guarantees a correct (yet sometimes too conservative) scheduler, it is possible to manually override the extracted scheduler characteristics in the GUI. From this information, an improved backtracking scheduler is finally generated.

Code selector: In order to get an operational compiler, a minimum set of *code selector rules* or *mapping rules* is needed. These mapping rules are the basis for the tree

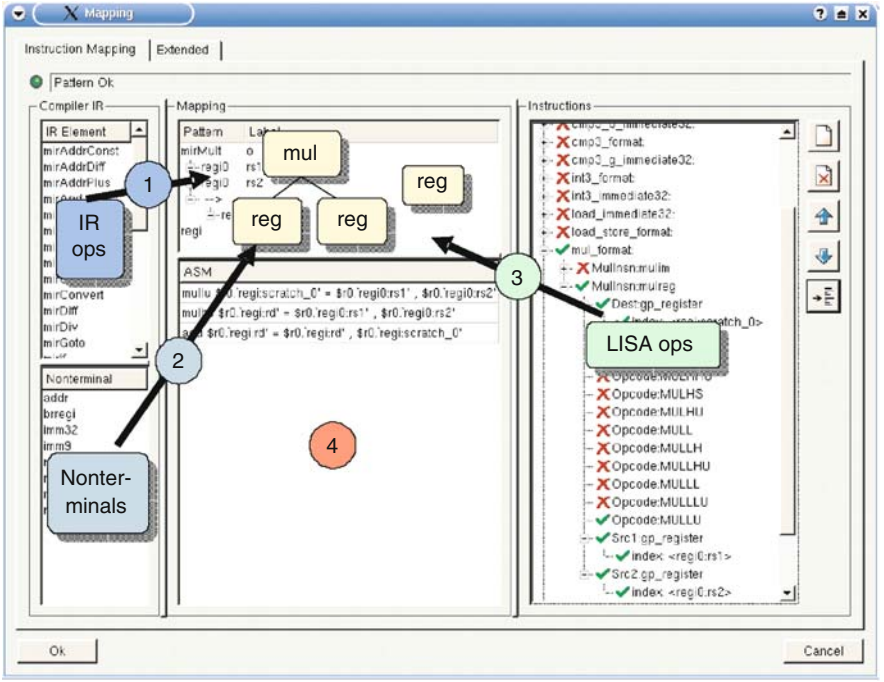


Fig. 5.5 Mapping dialog

pattern matching based code selector (cf. Section 3.3.2) in CoSy. The *Compiler Designer* comprises a so-called *mapping dialog* (Fig. 5.5). This dialog provides the set of available IR operations (top left in Fig. 5.5), defined nonterminals (bottom left), as well as the hierarchically organized set of machine operations in the given LISA model (right). By means of a convenient *drag-and-drop* mechanism, the user can *manually* compose mapping rules (top center) from the given IR operations (1) and nonterminals (2). Likewise, the link between mapping rules and their arguments on the one hand and machine operations and their operands on the other hand is made via drag-and-drop in the mapping dialog (3). In this way, multi-instruction rules which can even contain control flow as well as complex instructions like MAC can be composed. The example from Fig. 5.5 shows the mapping defined for a 32-bit multiply operation, which is implemented by a sequence of two 16-bit multiply instructions and an add instruction. Based on this manually established mapping, the *Compiler Designer* looks up the required assembly syntax of involved instructions (4) in the LISA model and can therefore automatically generate the code emitter for the respective mapping rule. The output of the code emitter is symbolic assembly code, which will be further processed by the register allocator and the instruction scheduler during code generation.

The mapping dialog also provides additional capabilities, e.g., for capturing rule attributes like type-dependent conditions for rule matching or for reserving scratch registers for use in complex multi-instruction rules, such as the above 32-bit multiply example.

The *Compiler Designer* supports a generic stack organization, which assumes that the architecture provides stack and frame pointer registers as well as register-offset addressing. Corresponding to this generic stack model, the user has to assign instructions to some predefined mapping rules needed for function prolog and epilog code generation.

Providing the minimum set of mapping rules enables the generation of a working compiler suitable for early architecture exploration. Naturally, at any time, the user may refine the code selector by adding more dedicated mapping rules that efficiently cover special cases leading to higher code quality.

The final output of the *Compiler Designer* is a compiler specification file in CoSy's CGD format, from which in turn a C/C++ compiler is generated fully automatically. During compiler retargeting, the session status of the *Compiler Designer* can be saved in XML format and can be resumed at any time.

5.4 Synopsis

- The *Processor Designer* environment supports all ASIP design phases.
- All software development tools, except the compiler, can be generated fully automatically.
- Some C compiler components are extracted automatically from the LISA model (e.g., scheduler tables) while the largest part (code selector description) still needs to be retargeted manually.

Chapter 6
Code Selector Description Generation

In Section 3.3.2, it was mentioned that the code selector's task is to map the IR to a semantically equivalent sequence of machine instructions. A common technique for code selection is the tree-pattern-matching technique, which is also employed in the CoSy platform. Like in many other ADLs, the required tree grammar must be *manually* specified in the *Compiler Designer*. Practical experience showed that this is a time-consuming, tedious, and error-prone task. Additionally, two major drawbacks have been identified: first of all, the designer actually starts with an empty code selector specification, i.e., he must have the knowledge about which code selector rules are necessary to build a working compiler that is able to translate arbitrary input programs. Second the code selector description from a previous architecture exploration phase may be inconsistent after a change in the underlying ADL model (e.g., a rearrangement of the instruction-set hierarchy). In this case, the code selector specification must be entirely revised. Unfortunately, major changes to the ADL model are quite common in the early exploration phase when different architectural alternatives are evaluated. This is further aggravated by the fact that the user is responsible for maintaining the correctness of the mapping rules, since pure changes in the instruction behavior description, without changing the hierarchy or the assembly coding, are not detected automatically. Hence, this chapter presents a novel methodology to generate the code selector description automatically from LISA processor models (Fig. 6.1), which completely eliminates these problems.

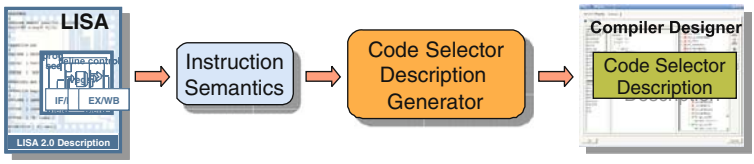


Fig. 6.1 Code selector description generation

The rest of this chapter is organized as follows: Section 6.1 elaborates the difficulties extracting code selector relevant information from a given LISA model. The extension to the LISA model required to circumvent them are presented in Section 6.2. Afterward, Section 6.3 describes how this information is used to enable the

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_6, © Springer Science+Business Media, LLC 2010

automatic generation of the code selector rules. Finally, Section 6.4 describes the integration into the *Compiler Designer*.

6.1 The Semantic Gap

When the LISA language was initially developed, the primary goal was to generate fast processor simulators [271]. In the following, the language was further refined and extended to be able to describe a broader range of architectures as well as to enable the generation of the remaining software development tools. Consequently, a LISA description has a rather simulator-centric view, i.e., the main focus in its design was to capture cycle and bit-true behavior of machine operations. As a result, the LISA language allows arbitrary C/C++ descriptions of instruction semantics. This feature ensures highest flexibility to describe *how* an instruction performs, but results in a quite detailed ADL model. However, compiler generation requires rather the information *what* an instruction does – which is quite difficult to extract from such "informal" models of instructions. This *semantic gap* in particular complicates the code selector rule generation.

Consider the LISA operation example shown in Listing 6.1. It describes an addition instruction that sets the carry flag according to the result. Note that this operation (like all remaining operations in this chapter) has no pipeline stage assigned, and hence belongs to an instruction-accurate LISA model.

```
OPERATION ADD {
    DECLARE {
        GROUP src1, dst = { reg };
        GROUP src2 = { reg || imm};
    SYNTAX { "ADD" dst "=" src1 "," src2 }
    CODING { 0b0000 src1 src2 dst }
    BEHAVIOR {
      dst = src1 + src2;
      if (  ((src1 < 0) && (src2 < 0))
         || ((src1 > 0) && (src2 > 0) && (dst < 0))
         || ((src1 > 0) && (src2 < 0) && (src1 > -src2))
         || ((src1 < 0) && (src2 > 0) && (-src1 < src2)))
      { carry = 1; }
    }
}
```

Listing 6.1 LISA operation for an ADD instruction

Even for this relatively simple operation, it is quite impossible to accurately extract the high-level semantic meaning of the instruction automatically from the BEHAVIOR section. First of all, the presented code is, due to numerous syntactic variances in C/C++, only *one* way to describe the carry flag computation. This is further aggravated by the fact that once a pipeline is modeled, this C/C++

instruction behavior description will be distributed over different pipeline stages (cf. Section 5.2). Besides, the example does not model any architectural feature such as register bypassing and side effects, which would lead to a much more complex description than what is shown in the example.

Thus, in order to close the semantic gap, a new SEMANTICS section is introduced to LISA [126]. It captures the instruction behavior at a higher abstraction level while ignoring all structural details like pipelining for instance. This enables a clean and unambiguous way of describing instruction semantics, which in particular are suitable for the generation of code selector rules.

6.2 SEMANTICS Section

The requirements for description of instruction semantics are as follows:

- Uniqueness, simplicity, and flexibility.
- A single, concise formalism to define the semantics, though still flexible enough to describe even complex operations. Considering that the SEMANTICS sections and BEHAVIOR sections describe both the behavior of instructions, a concise description reduces redundancy to a minimum.
- Legacy LISA models should be easily extendable to aid the compiler generation with only minor additional design effort.
- For the purpose of compiler generation, ambiguity has to be strictly avoided.
- The designer shall not need compiler and/or simulator knowledge to create a model with semantic sections.

The MIMOLA ADL [235] employs a set of the so-called *micro-operations* to describe a processor's instruction-set. Each micro-operation can be seen as a primitive operation similar to the instructions of an RISC instruction-set architecture. Complex instructions can be typically modeled by a combination of such. This approach has been proven feasible and complete for the specification of instruction semantics, but it is unsuitable for the description of complex micro-architectural behavior as required for cycle-accurate simulators or HDL generation. Fortunately, this is already covered by the BEHAVIOR section. Thus, the micro-operation idea is adapted for the definition of the SEMANTICS section since it meets the requirements for the description of instruction's semantics very well.

6.2.1 Semantics Statements

A SEMANTICS section basically consists of one or more *semantics statements*, which are composed of micro-operations. In total, four different kinds of semantics statements are available:

- A statement of the form *<source> − > <destination>;* is called an *assignment statement*. It performs either some computations defined by a micro-operation and stores the result in the destination or just moves the data from the source to

the destination. The source expression of the assignment statement must produce a result. For instance, a `_NOP` (no operation) micro-operation at the left-hand side is not allowed since it does not produce any result. Likewise, the destination of an assignment statement cannot be an arbitrary micro-operation expression. Only reasonable data sinks in an architecture can be used as destination (e.g., status flags, registers).

- Although not all LISA resources make sense in semantics sections, there is still a reasonable number of processor resources that can be used as operands for micro-operations. Such resources must be wrapped into LISA operations, which then defines the semantical type of the respective resource. The semantical types are called *modes*. Such modes do not perform any computations or data assignments. Currently, two kinds of modes can be defined, *register* mode and *immediate* mode. Thus, the *mode statement* encapsulates the register and immediate operands of the micro-operations.
- Control flow within the SEMANTICS section is modeled with the *if–else statement* (Section 6.2.7).
- Literally speaking, the *non-assignment statement* should include all the statements that are not assignment statements including if–else statements and modes statements, e.g., the statement `_NOP;`. A more often usage of such statement is in the operation hierarchy (see Section 6.2.8).

6.2.2 Semantics Resources

The RESOURCE section of a LISA model specifies all architecture-relevant resources such as register files, internal status register, pipeline, memory bus, and so on. They must be declared in this section before being used. All declared architecture resources are visible and accessible in the BEHAVIOR sections. However, in the SEMANTICS sections not all resources are allowed. Naturally, only those having clear meanings to the compiler can be accessed. The usable resources in the SEMANTICS sections are memories, registers, program counter, stack pointer, carry flag, overflow flag, negative flag, and zero flag. In LISA models, register resources are usually accessed via wrapper operations, whose semantics are defined by the mode statement. For memory accesses a special micro-operation exists (Section A.2.6). Furthermore, dedicated identifiers exist for other common resources. They can be accessed via the shortcuts described in Table 6.1.

Table 6.1 Shortcuts for special resources

Shortcut	Resource specifier
_PC	PROGRAM_COUNTER
_SP	STACK_POINTER
_CF	CARRY_FLAG
_ZF	ZERO_FLAG
_OF	OVERFLOW_FLAG
_NF	NEGATIVE_FLAG

6.2.3 Micro-Operations

Usually, the semantic meaning of an instruction is tightly coupled to the syntax of an instruction. In most cases, the semantic of an instruction can be described by taking one or more of the instruction's parameters, performing a computation on those operands, and finally modifying one or more of the instruction's operands (and/or processor resources). The examination of the instruction-set architectures of several contemporary embedded processors revealed that the high-level behavior of most instructions are typically either arithmetic calculations using several operands or control-flow operations. The calculations carried out by the instructions can be further decomposed into one or several primitive operations, whereas the set of primitive operations is quite limited. However, to meet the aforementioned requirements of a semantic description, the operations that should be included in the set of micro-operators must be carefully selected. For instance, only those operators are of importance that are relevant for code selector generation. It does not make sense to consider dedicated micro-operations for, e.g., saturated arithmetic as supported by many DSP architectures since the C language does not consider saturated arithmetic at all. Though at the same time, it should be possible to describe those operations with existing micro-operators.

```
OPERATION ADD {
  DECLARE{
    GROUP src1, dst = { reg };
    GROUP src2 = { reg || imm};
  }
  ...
  SEMANTICS{
   _ADD|_C|(src1, src2)<0,32> -> dst;
  }
}
```

Listing 6.2 Operation with semantics

The example in Listing 6.2 shows the ADD operation from the previous example using the SEMANTICS section instead of the BEHAVIOR section.

A micro-operation is a tuple (o, S, U, v, w), consisting of the micro-operator o, the set of side effects $S \subset \{C, V, N, Z\}$, the set of operands U, and a bit-field specification represented by bit offset v and bit-width w. In the given example, the micro-operator _ADD defines the integer addition, while the following _C specifies that the carry flag is affected by the operation. Other supported flags are zero (_Z), negative (_N), and overflow (_V). A comma-separated list of operands, i.e., src1 and src2, follows in parentheses. The <0,32> after the _ADD's brackets explicitly specifies that the result of the addition is 32-bit wide (see Section 6.2.4). Hence, the corresponding tuple for _ADD is $(\{C\}, \{src1, src2\}, 0, 32)$. If the bit-width is omitted, it will be deduced from the operand(s) of the micro-operation. Finally, the

pointer -> specifies the location for the result. Compared with the BEHAVIOR sections shown in Listing 6.1, the description in the SEMANTICS section is obviously much simpler.

```
OPERATION reg {
  DECLARE{
       LABEL index;
  }
  SEMANTICS{
   _REGI(R[index])<0..31>;
  }
}
```

Listing 6.3 Operand's semantics

The operands of the micro-operator can be either terminal elements, such as integer constants, or other LISA operations like in the example. In the latter case, the respective operations must have a SEMANTICS section on their own. In Listing 6.3, the SEMANTICS section of the `reg` operation defines the semantic type of the operand using a mode statement. In this case, it refers to a 32-bit integer register file specified as an array `R` in the global RESOURCE section (Listing 6.4).

```
RESOURCE {
  MEMORY_MAP { ... }
  ...
  REGISTER unsigned int R[0..15];
}
```

Listing 6.4 Resource section

The label `index` is used to index the registers. The number of available registers is derived from this label, e.g., assuming `label` is a 4-bit binary number, it can index up to 16 registers. According to the SYNTAX section of operation `reg`, these registers are named `R0`–`R15`. These kinds of micro-operators define the semantical type of the respective processor resource and are called *modes*. Apart from this *register* mode an additional *immediate* mode exists. This mode defines an immediate value that is part of the instruction coding. In this case, the bit-width can be directly derived from the CODING section. Listing 6.5 provides an example for an 8-bit immediate operand value.

```
OPERATION imm{
    DECLARE{
        LABEL value;
    }
    CODING{ value = 0bx[8] }
    SEMANTICS{
        _IMMI(value);
    } ...
}
```

Listing 6.5 Immedidate mode example

Similar to the micro-operators, each operand of a micro-operation can be represented as a 3-tuple (u, v, w) consisting of the value/resource u and a bit-field specification represented by bit offset v and bit-width w. Thus, the corresponding tuple for operation `reg` is $(u, v, w) = (R[index], 0, 32)$.

6.2.4 Bit-Width Specification

Except for the `_NOP` micro-operation, all micro-operation are able to produce some result in one way or another. However, not all instructions need the complete result of the micro-operation. For instance, an architecture supports a multiplication instruction that multiplies two 32-bit registers and puts the result also in a 32-bit register. Hence, the instruction needs only 32-bit of the default 64-bit result. In such cases, a bit-width specification such as `_MULII(src1,src2)<0..31>->dst;` can be used. The bit-width is given by the starting offset and the ending offset. Another possibility is to specify the starting offset and the width. In this case, the bit-width specification would change to `<0, 32>`. Both forms are equivalent and are provided just for convenience. Most of the time, the offsets and widths are specified using integer values. However, LISA operation names can also be used. This enables the modeling of dynamic bit-width extractions, i.e., to extract the bits according to a register value or immediate coding.

If no explicit bit-field specification is provided for the micro-operator, it will deduce the specification from the input operands or, in case of resources, are extracted from the RESOURCE section. Considering the bit-widths of both sides of an assignment statement, they must be the same. An error will be issued if a mismatch exists. For instance, the addition of two operands $(a, 0, 32)$ and $(b, 0, 32)$ results in the 3-tuple $(c, 0, 32)$, where c is the result of the 32-bit addition of a and b. Thus, the explicit bit-field specification `<0,32>` for `ADD` in Listing 6.2 is actually superfluous.

Note that the bit-width specification is compulsory for those micro-operations whose output bit-width cannot be deduced from their operands, such as sign/zero extension for instance. Furthermore, certain micro-operators have some implicit restrictions for the input operands regarding the bit-width. An implicit constraint for the `_ADD` micro-operator is that both operands share the same bit-width. If that

constraint is not met, the respective operand has to be extended to match the width of the other operand by means of an explicit sign/zero extension. Two separate micro-operations _SXT and _ZXT serve that purpose.

The generic micro-operation and operand representation allows for a very compact instruction-set description while keeping the number of required micro-operations small. A comprehensive list of all available micro-operators can be found in Appendix A.

6.2.5 Micro-Operator Chaining

Obviously, not all instructions can be expressed by a single micro-operation. For instance, many DSP processor architectures have instructions for combined computations such as Multiply and Accumulate (MAC) for instance. Such behavior is captured in SEMANTICS sections by using a micro-operation as the operand of another micro-operation, henceforth referred to as *chaining*.

```
OPERATION MAC{
  DECLARE{
   GROUP src1, src2, dst = { reg };
  }
  ...
  SEMANTICS{
    _ADD(_MULUU(src1, src2)<0,32>, dst) -> dst;
  }
}
```

Listing 6.6 Micro-operation chaining

A simple example of a MAC operation is shown in Listing 6.6. _MULUU is the micro-operator that denotes the unsigned multiplication. Its result is used as one of the operands of the _ADD, thus building a micro-operation chain. The bit-field specification in angle brackets is required to ensure that both operands of _ADD have matching bit-widths.

The chaining mechanism helps to describe complex operations without introducing temporary variables. This guarantees a tree-like structure for each semantic statement. Such trees are well suited for mapping rule generation since most code-selection algorithms are based on the tree-pattern-matching technique.

6.2.6 Execution Timing

In general, most of the RISC instructions can be modeled with one statement (including chaining), but obviously this is not sufficient for those instructions

transferring data to multiple destinations. However, this can be modeled with multiple statements in the SEMANTICS sections. Thus, the timing of the execution of those statements needs to be defined: all the statements in one semantics section will be executed concurrently (rather than sequentially). Consequently, a preceding statement's result cannot be used as the input of the following statement. Listing 6.7 illustrates this. The SWAP operation swaps the content of a register by exchanging the upper and lower 16 bits. Because the execution is in parallel, the data in the register are exchanged safely without considering sequential overriding.

```
OPERATION SWAP{
  DECLARE{
    GROUP src = { reg };
  }
  ...
  SEMANTICS{
   src<0,16>  -> src<16,16>;
   src<16,16> -> src<0,16>;
  }
}
```

Listing 6.7 Multiple statements

6.2.7 IF–ELSE Statements

Another kind of important behaviors used in modern processors is predicated execution, i.e., an instruction is executed depending on certain conditions, similar to the C language's if–else statement. In order to model such instructions, IF-ELSE statements and comparison operators can be used in the SEMANTICS sections to model all kinds of conditions. Ten predefined micro-operators are available (Appendix A) to specify comparisons. Each of these comparison operators returns either true or false, depending on the result. So far, such operators can only be employed within IF–ELSE conditions.

To form a more complex condition, conditions can be concatenated by "||" or "&&". Like in the C language, the former means logical or of the conditions on its both sides, and the other represents a logical and. The condition expression is evaluated from left to right and the two symbols have the same priority, which means that the expressions besides the leftmost symbol are evaluated first, yet brackets can be used to override this relation. Of course, comparisons can be chained, too.

Listing 6.8 gives an example for an addition with carry bit. The _EQ operator checks whether the two input operands, an integer constant and the carry flag, are equal or not. Depending on the result, the IF statement will execute the code specified in the braces. Nested IF–ELSE statements, are however, currently not supported.

```
OPERATION CADD{
  DECLARE{
    GROUP src1, src2, dst = { reg };
  }
  ...
  SEMANTICS{
   IF(_EQ(_CF,1)){
    _ADD(src1, src2) -> dst; }
  }
}
```

Listing 6.8 IF–ELSE statement

Naturally, it is not possible to describe every instruction with the formalism presented above. For instance, ASIPs often feature application-specific instructions whose behavioral description can vary from only a few code lines to several hundreds. Obviously, such complex behavior can hardly or not at all be expressed with micro-operations. But this is actually no drawback since such instruction cannot be directly exploited by today's code-selection techniques anyway. For such instructions, a special *intrinsic* micro-operation can be used as some sort of a wildcard. No semantic meaning is associated with its description, just an user-defined name. Listing 6.9 illustrates this. With the capability of defining intrinsics, every instruction can be described in the SEMANTICS sections. Intrinsic micro-operators are treated separately during mapping rule generation.

```
OPERATION DCT2d{
  DECLARE{
    GROUP src,dst = { reg };
  }
  ...
  SEMANTICS{
      "_DCT2d"(src) -> dst;
  }
}
```

Listing 6.9 Intrinsic micro-operation

6.2.8 Semantics Hierarchy

Section 5.2 illustrated already the LISA operation hierarchy. This achieves modeling flexibility and simplicity. Consequently, it has to be supported by the semantic description as well. The execution of an instruction equals in principle the execution of operations along the activation chain. Likewise, the semantic of an instruction is given by the SEMANTICS sections of the operations in the activation chain. Listing 6.10 provides an example.

```
OPERATION arithm {
  DECLARE{
    GROUP src1, src2, dst = { reg };
    GROUP opcode = { ADD || SUB ...};
  }
  ...
  SEMANTICS{ opcode|_C|(src1, src2) -> dst; }
}

OPERATION ADD {
  ...
  SEMANTICS{ _ADD; }
}

OPERATION SUB {
  ...
  SEMANTICS{ _SUB; }
}
```

Listing 6.10 Hierarchical operators

In the `arithm` operation, the GROUP `opcode` is used as a micro-operator. Consequently, the concrete micro-operators is obtained from the SEMANTICS sections of the respective GROUP members. In this case, the SEMANTICS sections of the `ADD` and `SUB` operation provide the corresponding micro-operator. The similarity of the `ADD` and `SUB` operation's semantics is well exploited here to simplify the description.

```
OPERATION ADD {
  DECLARE{
    GROUP src1, dst = { reg };
    GROUP opd = { SHL || SHR };
  }
  ...
  SEMANTICS{ _ADD(src1, opd)-> dst; }
}
OPERATION SHL {
  DECLARE{
    GROUP src2 = { reg };
    GROUP imme = { imm };
  }
  ...
  SEMANTICS{ _LSL(src2, imme); }
}
OPERATION SHR {
  ...
  SEMANTICS{ _LSR(src2, imme); }
}
```

Listing 6.11 Hierarchical operands

A SEMANTICS section can return not only a micro-operator but also a complete micro-operation expression. In Listing 6.11, the SEMANTICS sections of the _SHL and _SHR operations do not contain a complete assignment statement but micro-operators with operands (_LSL and _LSR are logical left and right shift micro-operators). Such statements are called *nonassignment statements*. They refer to all statements that do not carry out data assignments, predicated execution, or resource encapsulation. As a result, the semantics of these two operations is not self-contained, because the data sink is missing. The use of these two operations is actually doing operand pre-processing for the ADD operation, which can be seen in its SEMANTICS section. The opd GROUP, which contains the previous two operations, is used as one of the operands of the _ADD micro-operation. Thereby, depending on the binary encoding of the actual instruction, one of the operand registers will be left or right shifted before the addition is actually performed.

The presented formalism that defines the SEMANTICS sections is very flexible and well integrated into LISA. If the commonalities of instructions are fully exploited, their instruction semantics can mostly be described with a single or a few semantic statements.

6.3 Code Selector Description Generation

The code selector generator in CoSy uses the dynamic programming tree-matching algorithm as presented in Section 3.3.2. The tree grammar $G = (T, N, P, S, w)$ consists of finite sets N and T of *nonterminal* and *terminal* symbols, respectively, as well as a set P of *mapping rules*, the corresponding cost metric w, and a *start symbol* $S \in N$. The terminals T essentially describe the available IR operations of the given source language and thus are target machine independent. Likewise, the start symbol S requires no special retargeting effort. Only the nonterminals N, and the rules P, need to be adapted to the target machine. N basically reflects available instruction operand kinds, e.g., registers, memories, and addressing modes like register offset addressing for instance, while P defines how source language operations are implemented by the target instructions. Each mapping rule in P has the form of a *tree pattern* that may serve to cover a data-flow graph fragment during code selection. Figure 6.2 shows a typical CoSy mapping rule specification. Each rule starts with the keyword RULE followed by the tree pattern specification. In CoSy, IR operators are named mirPlus, mirMult, etc. Each IR operator and each operand can be associated with a name for further reference. Additionally, each rule has an (optional) CONDITION assigned that must be met before the rule can be applied. Here in the example, the rule only matches for integer additions (i.e., floating-point additions are not matched by the rule). Additionally, each rule has a fixed cost assigned that is used by the tree-pattern-matching algorithm. Finally, the EMIT part contains a print function that is executed by the code emitter, the final compiler phase, if the rule has been selected. Here, it prints the add syntax including the physical register names that have been assigned to the nonterminals during register allocation.

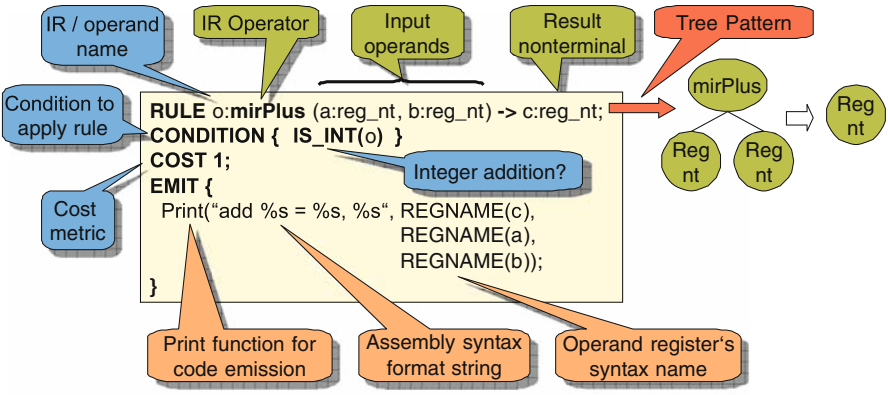


Fig. 6.2 CoSy mapping rule syntax

The following sections describe how the nonterminals N and the mapping rules P and the associated conditions are automatically generated from the instruction semantics information given in the SEMANTICS sections (Fig. 6.3).

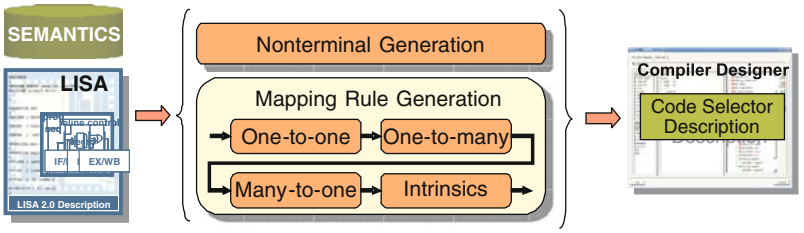


Fig. 6.3 Nonterminal and mapping rule generation

6.3.1 Nonterminal Generation

In tree grammar descriptions, nonterminals can be seen as temporary variables connecting different grammar rules. In this way, they determine the expressive power of a tree grammar specification to a large degree. Usually, each nonterminal corresponds to some feature of the target architecture that is common to a number of instructions such as registers and memory accesses for instance. Thus, depending on the type of the temporary, nonterminals can be divided into the following four categories:

Register nonterminals represent the registers that can be used by the compiler.
Immediate nonterminals carry the constant values that can be used as immediate operands in instructions.
Condition nonterminals are typically condition flag registers that are affected by different instructions, e.g., carry or zero flag.
Addressing mode nonterminals encapsulate the addressing modes supported by the target, e.g., register offset addressing.

If an architecture does not support any conditions flags, then the condition nonterminal is not needed. However, all other nonterminal types are usually supported by any programmable architecture.

6.3.1.1 Register Nonterminals

In LISA processor models, accesses to these storage location or processor resources are usually described by a wrapper operation, like the operation `reg` in Listing 6.3. A set of micro-operators is available that captures the semantics of these wrappers. As mentioned above, the `_REGI` operator in the example stands for a register access. Its operand `R` is the name of the corresponding LISA resource that is used to model the register file. The index of the accessed register is given by `index`, a LISA label whose value is determined by the instruction encoding of the operation. Another important information, the bit-width of the registers, is specified with the notation `<0,32>`, which means the register is 32-bit wide and the least significant bit is bit 0. From this specification, a register nonterminal with the given properties can be generated.

6.3.1.2 Immediate Nonterminals

Likewise, the generation of the immediate nonterminals is based on the `_IMMI` operator. The corresponding mode statement in the immediate wrapper operations does not contain the signed/unsigned information of the immediate coding (Listing 6.5). Actually, this information comes from the micro-operation that uses the semantics of the immediate wrapper. The operation `ADDI` shown in Listing 6.12 references the operation `imm`. More specifically, it is used as the operand of the `_SXT` micro-operator. That means, the immediate is used as a signed value and, hence, a signed immediate nonterminal is generated. Correspondingly, if the immediate is used by `_ZXT` micro-operations, the generated immediate nonterminal is an unsigned immediate nonterminal. Using the immediate in one of the other micro-operations does not have any effect on the immediate nonterminal generation. In this case both, sign and unsigned, immediate nonterminals are generated from the immediate wrapper operation.

```
OPERATION ADDI {
  DECLARE{
    GROUP src, dst = { reg };
    GROUP imm8 = { imm };
  }
  ...
  SEMANTICS {
    _ADD(src, _SXT(imm8)<0..31>)) -> dst;
  }
}
```

Listing 6.12 Add immediate operation

6.3.1.3 Condition Nonterminals

In general, condition nonterminals represent the flag registers; their existence depends on the use of the four predefined flags, namely carry (_C), zero (_Z), overflow (_O), and negative (_N) flag. The nonterminal generator checks whether there is any assignment to a flag register or any micro-operation having an affected flag declared. For example, the semantics statement in Listing 6.2 writes to the carry flag. Regardless of the flag type, a condition nonterminal is generated. Additionally, the nonterminal has an attribute that can store the type of the compare instruction. This field corresponds to the 10 compare conditions provided by the SEMANTICS section. This is later used for the mapping of compare instructions.

6.3.1.4 Addressing Mode Nonterminals

The generation of addressing mode nonterminals is based on the _INDIR micro-operation used for memory references. However, it requires some more analysis since it is typically used in a micro-operator chain to describe more complex addressing modes. First of all, all assignment statements using the _INDIR micro-operations are collected. If an _INDIR expressions is used to read from memory as well as to write to memory, then it is further analyzed to generate the addressing mode nonterminal. The operand of the _INDIR expression provides the information how the address is calculated. An example is given in Listing 6.13. The LOAD and STORE operations have the common address calculation expression _ADD(addr,_SXT(imm8)<0..31>). Obviously, the address is the result of the addition of the semantic elements from addr and imm, i.e., an addition of a register nonterminal and an immediate nonterminal also known as a register-offset addressing mode. This information enables the generator to create a proper addressing mode nonterminal for this mode.

```
OPERATION LOAD {
  DECLARE{
    GROUP addr, dst = { reg };
    GROUP imm8 = { imm };
  }
  ...
  SEMANTICS {
    _INDIR(_ADD(addr, _SXT(imm8)<0..31>))<0..31> -> dst;
  }
}

OPERATION STORE {
  ...
  SEMANTICS {
    src ->_INDIR(_ADD(addr, _SXT(imm8)<0..31>))<0..31>;
  }
}
```

Listing 6.13 Load and store operation

These four kinds of nonterminals are processor-specific elements in the mapping rules. The nonterminal generator checks all available LISA operations for those micro-operators and creates the corresponding nonterminals. Afterward, the algorithm proceeds with the generation of the mapping rules.

6.3.2 Mapping Rule Generation

In general a mapping rule consists of three parts: a tree pattern, the result nonterminal produced by the rule, and one or more associated machine instructions. The tree pattern represents a C-level computation that can be performed by the processor. Likewise, the input operands of the computations are usually also nonterminals. Thus, to generate mapping rules for a working code selector description, mainly two questions need to be answered. The first one is

- *which* tree patterns are needed to cover the complete set of possible IR operations and the second is,
- *how* the tree patterns are mapped to the target machine instruction-set.

6.3.2.1 Basic Rules

A complete code selector description must cover all IR tree patterns that the compiler frontend may produce. Since the source language does not change, the IR tree patterns needed to be covered by a code selector are actually fixed. Consequently, a set of *mapping rule templates* can be prepared without knowing the target processor. The set of such templates is called *basic rules* further on. Listing 6.14 shows a basic rule along with a CoSy mapping rule in Listing 6.15 for an addition of two registers, which stores the result again in a register.

```
COSYIR mirPlus(a,b) -> c;
PATTERN {
  _ADD(a,b) -> c;
}
```

Listing 6.14 Basic rule example

```
RULE o:mirPlus(a:reg_nt,b:reg_nt) -> c:reg_nt;
EMIT {
  print("add %s = %s, %s",
        REGNAME(c), REGNAME(a), REGNAME(b));
}
```

Listing 6.15 CoSy mapping rule

The `mirPlus` operator in both rules is an addition operation on the C level as defined in the CoSy IR. Obviously, there are two major differences between basic rules and CoSy mapping rules. First, the operands in the tree patterns of the basic rules (`a`, `b`, and `c`) are *placeholders* instead of the nonterminal `reg_nt` used in the CoSy rules. The code selector generator keeps a so-called *basic library* containing the basic rules needed for a complete coverage of C operations. The second obvious difference between basic rules and CoSy rules is that the latter is associated with an assembly instructions, i.e., the `print` function emits the corresponding assembly instruction, while a basic rule has one or more semantic statements assigned, referred to as compiler semantics in the following. The next sections briefly introduce the library syntax. A comprehensive description of the complete library specification is provided in Appendix B.

6.3.2.2 Nonterminal Enumeration

For each basic rule in the library, a list of target-specific tree patterns is generated by replacing the placeholders with the generated nonterminals in all possible combinations. Figure 6.4 illustrates this.

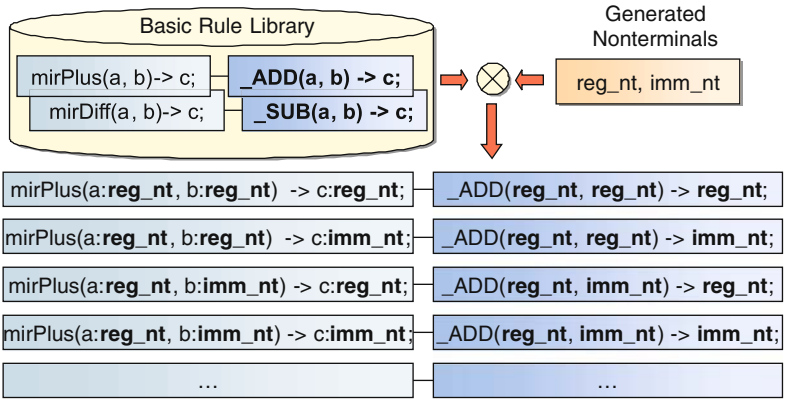


Fig. 6.4 Tree pattern generation

Unfortunately, this can result in a large amount of mapping rules, which must be processed. Even for this simple example with two generated nonterminals, this already results in eight possible combinations for each basic rule. However, some of them may never be generated by the frontend or just do not make sense. For example, the rule whose destination is an immediate nonterminal will never be mapped since an immediate coding cannot be assigned a value. Therefore, a type declaration is introduced to restrict the placeholder to certain nonterminal types.

For instance, in Fig. 6.5, the placeholder `a` and `c` are declared as `REGISTER`, that means they can only be substituted by register nonterminals. Likewise, `b` is annotated with `REGISTER` and `IMMEDIATE` so that it can be replaced by register and immediate nonterminals. In this way, the number of generated rules is reduced to two. The `IMMEDIATE` keyword can also be combined with `SIGNED` and

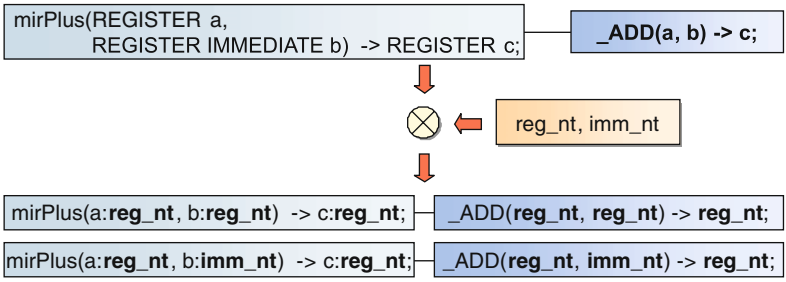


Fig. 6.5 Restricting nonterminal types

UNSIGNED. Additionally, ADDRESS can also be used to restrict the enumeration to addressing mode nonterminals.

6.3.2.3 Basic Rule Conditions

As mentioned above, a code selector rule can be annotated with a condition that must be met before the code selector can apply the rule. Listing 6.16 shows a basic rule for a pointer to integer conversion. The convert node o implicitly refers to the convert node that defines the source and the destination type. In the usual case, the condition is directly copied to the target-specific rule. However, it may also contain some dynamic expressions such as SIZEOF(INT), which is replaced by the size of the C data types as defined in the *Compiler Designer*.

```
COSYIR mirConvert(REGISTER a) -> REGISTER c;
CONDITION {
  IS_POINTER(o) && IS_INT(o.Value)
}
NONTERMINAL_CONSTRAINT a==b;
```

Listing 6.16 Basic rule example

6.3.2.4 Nonterminal Constraints

In case the architecture features several register files, restricting the type of the placeholder might not be sufficient. The basic rule in Listing 6.16 has no compiler semantics assigned because no instruction needs to be issued in this case (according to the C standard [45]). However, if the architecture has two register files, and hence, two register nonterminals, a target-specific rule is generated whose destination nonterminal is different from the source nonterminal. Unfortunately, this case requires an instruction to move the value from one register file to the other. Since the rule does not issue an instruction the result would be erroneous. This case can be circumvented by using the NONTERMINAL_CONSTRAINT keyword. It can be used to provide a condition for the nonterminal type. Here in the example, it is specified that both source and destination must share the same nonterminal type.

Once all target-specific rules have been generated, the next task is to find suitable instructions in the LISA model that match the semantic statements of the generated tree patterns. The available instructions are collected by traversing the LISA operation DAG in a preorder traversal to find all possible activation chains. These corresponding instructions are then stored in a dedicated list, which is used during the mapping procedure. In most cases, the generated tree patterns have only a single semantic statement that can be directly covered by a single instructions in this list. This is denoted as *one-to-one* mapping. However, since ASIP designs should always be as efficient as possible, rarely used instructions might have been removed from the design. Unfortunately, some of them might be needed for a complete code selector description. In this case, *one-to-many* mapping is employed, which implements a semantic statement with a sequence of instructions. Moreover, ASIP designers not only simplify the instruction-set architecture but also add dedicated custom instructions for program hot spots. These instructions accelerate the program execution by performing many C-level operations at once, like a MAC instruction for instance. To utilize them in a compiler, the so-called *many-to-one* mapping rules can be specified. For those instructions containing an intrinsic micro-operator, a corresponding *compiler-known function* is generated. Finally, a few rules cannot be described by basic rules. These rules are directly generated by separate algorithms and therefore are called *internal rules*. The following sections describe how the instructions are selected for these five kinds of mapping rules using the instruction semantics information in the LISA model.

6.3.2.5 One-to-One Mapping

This mapping method is the first one applied by the code selector generator. The semantics statements of the basic rules are compared with the available instruction semantics in the LISA model. Both semantics match if the micro-operators, the operands, and the bit-width specification are the same. Figure 6.6 exemplifies this.

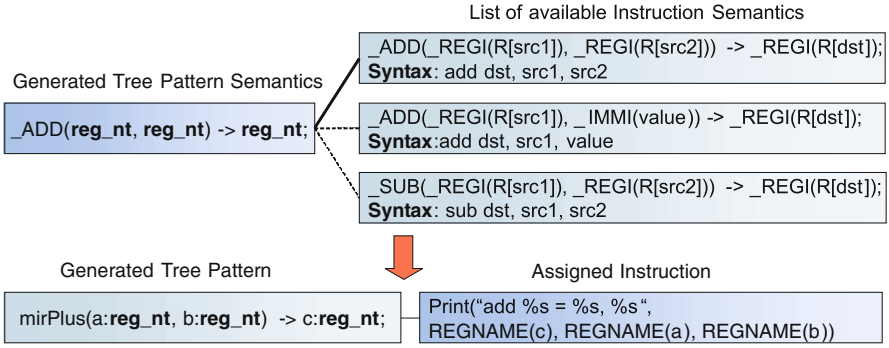


Fig. 6.6 Matching rule semantics and instruction semantics

Since some side effects in a real instruction might not be important for code selection, a successful one-to-one mapping does not require two identical semantics

patterns. For example, assume that the selected instruction semantics in Fig. 6.6 would change the carry flag (i.e., `_ADD|_C|`). Since the writing to the carry flag does not influence the result of an arithmetic addition, the side effects in the instruction semantics can be ignored by the generator. Thus, the instruction can still be selected for the generated tree pattern. Of course, such adaptation in the one-to-one mapping can only compromise those effects not affecting the results of the calculation. The micro-operators, operands, and bit-widths still must be exactly the same for both compared semantic statements.

In certain cases, the compiler semantics are quite different from the semantics of an equivalent instruction. A typical example are branch instructions. The description of some micro-architectural details in the SEMANTICS section cannot be completely avoided. Suppose an architecture supports a branch instruction that branches to the relative address given by a signed immediate value. The corresponding semantic description can typically be given by the following statement: `_ADD{_PC,_SXT(imm)<0..31>)->_PC;`. It contains the information how the architecture computes the branch destination. The basic rules must capture this in a more abstract way in order to be as target independent as possible. As a result, the basic rule can only assume that there at least exist an address. Thus, the semantics of a basic rule for, e.g., a goto basic rule is given by `ADDRESS -> _PC;`. Obviously, it is very unlikely that such a semantic statement directly matches with the given instruction semantic. Therefore, the mapping algorithm deals with certain compiler semantics in a special way.

Although processor architectures may calculate the target address of the branch in different ways, the operand(s) of the calculation remain similar. The only programmable operand in the semantic statement of the branch instruction given before is `imm`. The `_PC` represents the program counter that usually does not appear in the instruction coding and, hence, it is not programmable. Thus, the matching algorithm can deduce that `imm` must represent the branch target of the instruction. The mapping procedure of the program counter assignment is illustrated in Fig. 6.7.

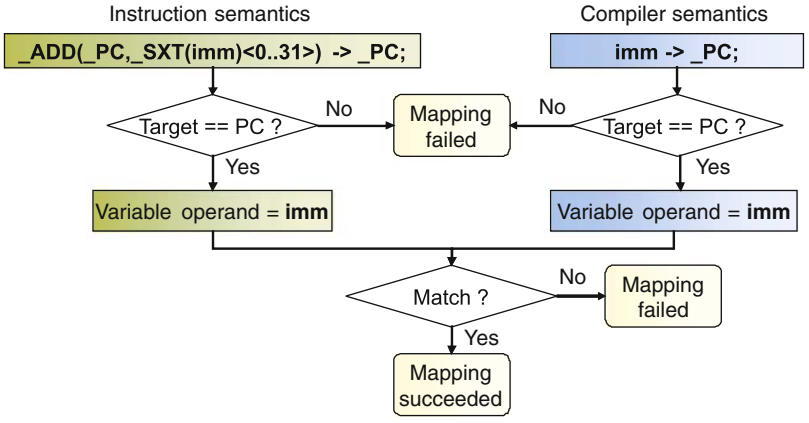


Fig. 6.7 Mapping of branch instructions

This process is only applied when the one-to-one mapping fails. First of all, it is checked whether the program counter is the target expression of both semantics statements. In this case, all programmable operands are retrieved, which can either be immediates or registers. If both statements return a single programmable operand that can be matched, then a proper mapping is created. In this way, the target-specific micro-operations are actually filtered out and the mapping can focus on the operands representing the address in the expression.

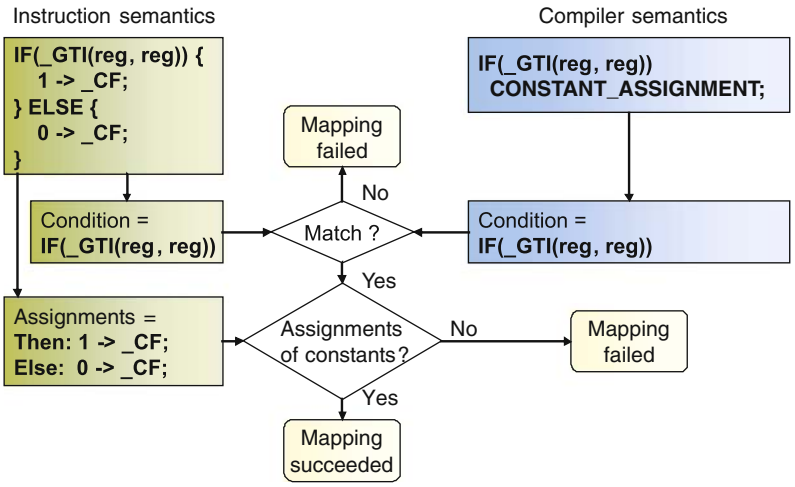


Fig. 6.8 Mapping of compare instructions

The mapping of compare instructions also needs special treatment. Figure 6.8 depicts the semantics description of a compare instruction. It performs a signed greater than comparison of two registers and stores the result in the carry flag. The corresponding basic rule only needs to know the exact type of the comparison. It does not have to care about how the result is used and stored. This is required by those basic rules (e.g., for if–else statements) that actually consume the result. Thus, the mapping of compare instruction cares more about matching the condition rather than the executed code (i.e., the then and else block). As shown in Fig. 6.8, the semantic description of the basic rule consists of the IF–ELSE statement and the keyword `CONSTANT_ASSIGNMENT`. The IF–ELSE statement itself is matched while the `CONSTANT_ASSIGNMENT` basically matches any then and else block, which assigns a constant value to the same processor resource, e.g., the carry flag. Later, the mapped then and else block are analyzed again to generate the condition for the if–else basic rules. This is further discussed in Section 6.3.2.9.

6.3.2.6 One-to-Many Mapping

As mentioned above, not all semantic statements of the generated tree patterns can be covered by a single instruction. However, for many semantic statements, alternative implementation using a sequence of semantic statements exists. In order to implement such an one-to-many mapping, the code selector generator needs to know the alternatives for a given semantic statement. This is specified by the so-called *semantics transformations*. An example is given in Fig. 6.9.

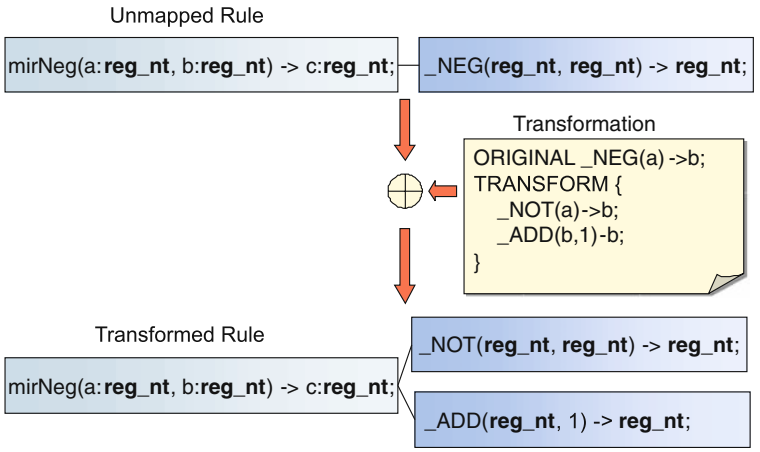


Fig. 6.9 Example for a semantic transformation

The `_NEG` micro-operator represents a two's complement negation. The specified transformation provides a mathematically equivalent solution to perform the negation. `_NOT` is the one's complement micro-operator. A two's complement can also be calculated with an one's complement and adding one afterward. Thus, if the generator fails to find an instruction for a tree pattern covering a negation, it will then try to find a suitable instruction for each semantic statement in the alternative implementation using the one-to-one mapping described above.

In principle, this approach can be used to provide alternatives for nearly all semantic statements, presuming that an equivalent transformation exists that can be expressed in the form of semantics statements. However, because of the variance of different instructions implemented in various architectures, it is not possible to specify transformations that fit every possible ISA. Nevertheless, the basic library comes by default with a set of commonly used transformations, like, e.g., shift and/or mask operations as alternative to implement sign or zero extension. As will be explained later, the basic library can also be extended with user-defined transformation tailored to the current ASIP design.

6.3.2.7 Many-to-One Mapping

Many-to-one mapping is especially important for application-specific instructions that perform composite operations to accelerate the program execution. However, since the designers can implement arbitrary combinations of operations in one instruction, it is obviously difficult to provide basic rules without knowing what the instructions actually do. Therefore, the code selector generation is inverted, i.e., instruction semantics in the LISA model that remain unused after the previous steps create a tree pattern on their own. For example, consider the `MAC` instruction in Listing 6.6, which is a commonly used composite operation. Two micro-operators are used, `_ADD` and `_MULUU`, an unsigned integer multiplication. The generator knows the mapping between the semantics micro-operators and the CoSy tree pattern nodes. Using this knowledge, it can create a corresponding tree pattern from the instruction semantics without user interaction. In the example, `mirPlus` is the CoSy tree-pattern node corresponding to the micro-operator `_ADD`, and `mirMult` maps to the `_MULUU` operator. If the source code contains a concatenated multiply and addition operation, this many-to-one mapping rule can then be employed by the code selector to use the `MAC` instruction instead of separate multiply and addition instructions (Fig. 6.10).

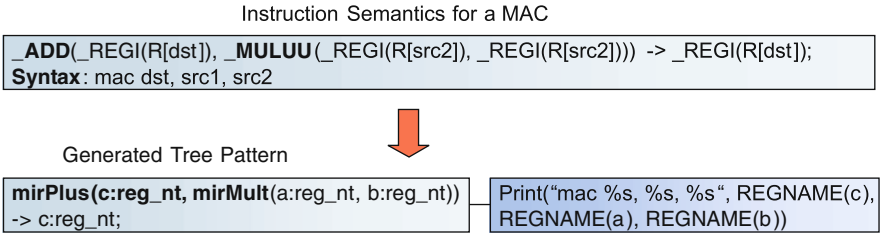


Fig. 6.10 Many-to-one mapping for a MAC instruction

6.3.2.8 Intrinsics

Generally, the many-to-one mapping works fine for arithmetic instructions whose semantics can be described with a chain of micro-operations. As mentioned in Section 3.3.2, tree pattern matching fails in case instructions exceed the scope of a single DFT, such as SIMD (single-instruction multiple data) instructions for instance. Other instructions are just too complex and can only be described using the *intrinsic* micro-operator as introduced in Section 6.2.5. Many compilers, though, provide support for these kinds of instructions via *compiler-known-functions* (CKFs) or *intrinsics*. Basically, CKFs make assembly instructions accessible within the high-level source code, where the compiler expands a CKF call like a macro. In order to integrate support for those instructions as well, the code selector generator creates for each instruction with an intrinsic micro-operator a CKF function definition for

the compiler's internal function prototype list and a mapping rule matching this particular CKF. As depicted in Fig. 6.11, this is basically an one-to-one translation.

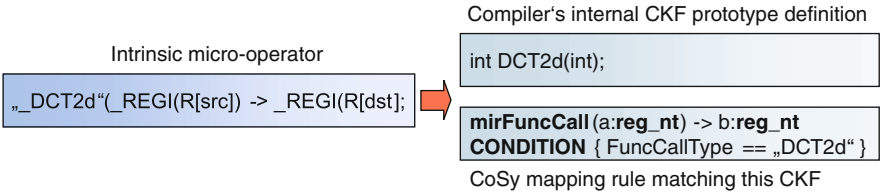


Fig. 6.11 CKF generation

6.3.2.9 Internal Rules

As mentioned before, internal rules refer to those rules whose semantics cannot be specified in a static form like the basic rules have. More specifically, conditional branch rules as required for if-then statements and the generic stack rules. The generation of both is explained in the following.

The compiler semantics of conditional branch rules is a semantic IF-ELSE statement. The statement to be executed conditionally is basically a semantic statement that describes a branch operation. However, the condition of the IF-ELSE statement can hardly be described when no architecture information shall be used. This is due to the fact that different architectures usually have different methods to implement compare instructions. Their results then build the condition of the branches.

Table 6.2 Implementation examples of compare and conditional branch instructions

Architecture	A	B	C
Compare semantics	IF(_GTI(src1,src2)) { 1 -> _CF; } ELSE { 0 -> _CF; }	IF(_GTI(src1,src2)) { 1 -> cond_reg; } ELSE { 0 -> cond_reg; }	IF(_GTI(src1,src2)) { 1 -> _CF; 0 -> _NF; } ELSE { 0 -> _CF; 1 -> _NF; }
Conditional branch semantics	IF(_EQ(_CF,1)) { imm -> _PC; }	IF(_EQ(cond_reg,1)) { imm -> _PC; }	IF(_EQ(_CF,1) &&_EQ(_NF,0)) { imm -> _PC; }

Table 6.2 exemplifies this. It shows compare and conditional branches for three architecture types. All architectures perform the same comparison, a signed greater than (`_GTI`), while the result of the comparison is stored in different ways. Architecture A only sets the carry flag (`_CF`) to one if the result is true. Architecture B on the other hand stores the result in a dedicated register file referenced by the `cond_reg` nonterminal. Finally, architecture C sets two flags, `_CF` and `_NF`. Consequently, the respective conditional branch instructions need the appropriate checks before the branch is issued. While architecture A's conditional branch takes the jump only when `_CF` is equal to one, architecture B takes it when the `cond_reg` contains

one. Architecture C ensures that `_CF` equals one and `_NF` equals zero before the branch is issued. As can be seen, all these implementations of comparison make the condition of the branch so heterogeneous that a static, generalized description is hardly possible. Therefore, the conditional branch rules are generated separately after the compare instructions are mapped.

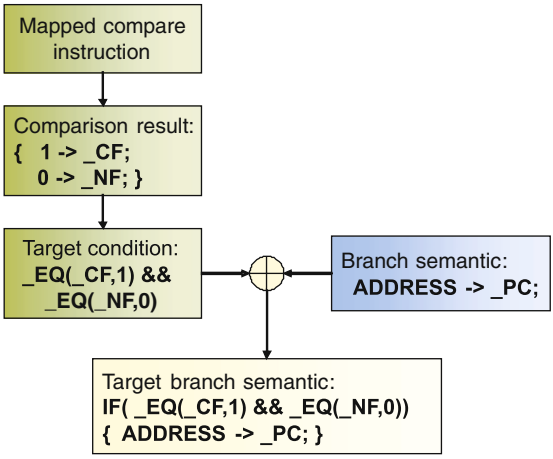


Fig. 6.12 Conditional branch generation

Figure 6.12 outlines the generation procedure of the conditional branch rules. First, an already mapped compare instructions is selected. Afterward, the semantic statements of the compare instruction are extracted, which are executed when the condition evaluates to true. These statements contain the information how the condition is stored. From this information, a corresponding condition expression is constructed. This can then be combined with the semantic of the branch instruction to finally create the semantic statement for the target-specific conditional branch rule. For this semantic, it is then searched for an equivalent semantic description in the instruction list. Regardless of whether a proper instruction is found, the corresponding rule is created in any case.

Another group of rules that are internally generated are the generic rules to map function prolog and epilog. For instance, assuming a stack organization with frame and stack pointer, the epilog and prolog can be decomposed into the following generic rules:

- Store/reload the stack pointer on/from the stack.
- Load/store registers that are overwritten by the function on the stack.
- Increase/decrease stack pointer by an immediate value.
- Indirect jump (return).

For each of these rule, a corresponding semantic description can be specified that is mapped to the available instructions.

6.4 Compiler Designer Integration

The mapping rule generator seamlessly complements the *Compiler Designer* (Fig. 6.13). Basically, the nonterminals are already generated when the tool starts up. Afterward, the mapping rule generation can be started with a push button and the generated rules are displayed. However, as mentioned above, certain mapping rules may still remain unmapped after the rule generation since the ASIP design probably does not feature all required instructions. Mapped rules are marked with a green tick while unmapped rules are marked with a red cross (see Fig. 6.14). It might also happen that the designer wants to create additional mapping rules in order to improve the code selector description. In either case, the designer can use the mechanism described in Section 5.3 to assign an instruction manually, to improve a mapping, or to create new mapping rules.

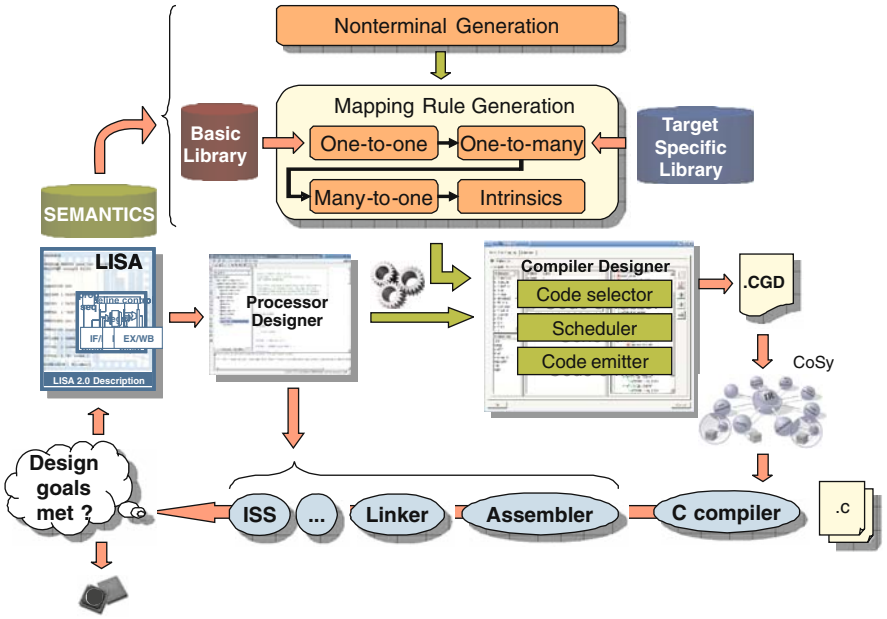


Fig. 6.13 Design flow with automatic code selector generation

In the early architecture exploration phase, when the design changes quite often and consequently, many compiler configurations are generated; this manual step must be done over and over again. In order to avoid this repetition, the user can specify a so-called *target-specific library*, basically an extension of the basic library, which contains additional mapping rules or target-specific semantic transformations to automate this process.

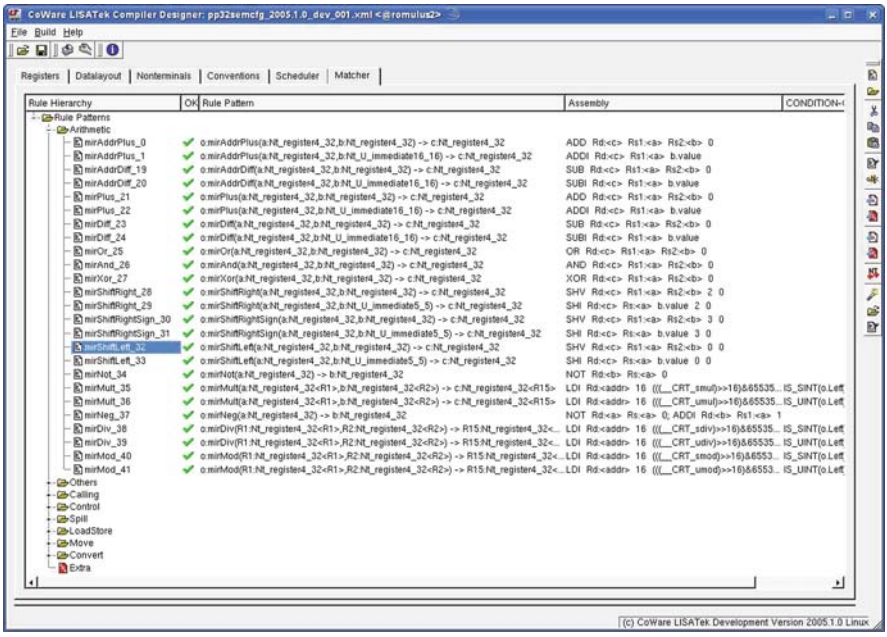


Fig. 6.14 Mapping result generation

6.5 Synopsis

- Due to the semantic gap, it is not possible to extract instruction semantics as required for code selector generation from detailed instruction behavior descriptions.
- The instruction semantics are captured by extending the ADL. A formalism for the description of instruction semantics is presented.
- The code selector generation consists of two phases, namely, nonterminal generation and mapping rule generation. The latter utilizes five different methods to generate the code selector description fully automatically.
- The presented approach is integrated into the *Compiler Designer*. This complements the *Processor Designer* framework such that the automatic generation of *all* software development tools from an abstract processor model is achieved.

Chapter 7
Results for SEMANTICS-Based Compiler Generation

This chapter gives a detailed account of the feasibility of the semantics-based approach for C compiler generation and the quality of the generated compilers.

7.1 Case Studies

In order to investigate the feasibility of modeling instruction's semantic with the methodology described in the previous chapter, several existing LISA models have been enhanced with SEMANTICS sections for compiler generation. This includes both instruction-accurate (IA) and cycle-accurate (CA) LISA models. More specifically, the following cores have been used: the ARM7, the CoWare LTRISC processor, the STMicroelectronics ST220 VLIW (four-issue slots) multimedia processor [84], the Infineon PP32 network processing unit, the Texas Instruments C54x digital signal processor [258], the MIPS4K [180], and the NXP Semiconductors TriMedia VLIW (five-issue slots) multimedia processor [190]. The LTRISC processor is a fully functional RISC template included in CoWare's *Processor Designer*. The PP32 is an evolution of [276] and comprises bit-field instructions. Although the SEMANTICS section is not intended for the extension of already existing models, this approach proved that the new section does not impose any particular modeling style – which is crucial w.r.t. LISA's flexibility paradigm. All models have been enhanced without any changes to the already existing specification.

Table 7.1 summarizes the results. Note that the design effort for adding semantics to the existing models is given in man-days. Obviously, the work for adding SEMANTICS sections scales with the number of operations in the architecture. In case of the TriMedia, this is not entirely true since many instructions are actually

Table 7.1 SEMANTICS section statistics

	ARM7	LTRISC	ST220	PP32	C54x	MIPS	TriMedia
Abstraction level	IA	IA	CA	CA	CA	IA	CA
ISA	RISC	RISC	RISC	RISC	CISC	RISC	RISC
No. operations	108	39	121	151	408	153	265
Design effort Δ	4d	2d	10d	8d	15d	5d	12d

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_7, © Springer Science+Business Media, LLC 2010

duplicated with marginal changes. This is due to TriMedia's capability of executing certain instructions conditionally, i.e., each case (conditionally/not conditionally executed) is modeled with its own operation. The complexity of the instruction-set (RISC vs. CISC) influences the effort, too. Generally, the effort for describing instruction semantics is much less than for a behavioral description in C. For instance, a 19×19 multiplication can be easily described with a single micro-operation and corresponding bit-field specifications whereas a behavioral description usually requires a significant amount of C code, which additionally has to be validated. In particular for the PP32, the explicit bit-field specification for the semantics (compared to a typical description in C using and/or/shift operations) reduces the design time significantly.

7.2 Mapping Rule Generation

Among the LISA models with SEMANTICS sections, the ST220, the PP32, and the MIPS have been selected to evaluate the mapping rule generator. The resulting code quality is compared to a CoSy compiler with *hand-crafted* mapping rules as well as a non-CoSy-based compiler. Both CoSy compilers are generated using the *Compiler Designer* tool. The ISA characteristics relevant for mapping rule generation are as follows:

ST220: The ST220 VLIW core is part of STMicroelectronics ST200 scalable and customizable core family, designed to be embedded into multimedia SoC devices. It can execute up to four instructions per clock cycle and features a multiplication unit. The load/store architecture incorporates two register files, one consists of 64 registers that are 32-bit wide and the other contains eight 1-bit-wide branch registers. Each branch register can be used for condition testing and conditional branches. Register–offset addressing is the only supported addressing mode.

PP32: The protocol processor (PP) has an RISC-based ISA with a single-issue slot, implemented in a four-stage pipeline. It is a typical Harvard architecture with separate program memory access. Among others, register–offset addressing is supported for load/store operations. The PP features extensions for bit-field operations that are optimized for single cycle processing of arbitrary bit patterns without additional shift/mask operations. The global register file consists of 16 elements, each having a data-word width of 32 bits. Conditional branches are executed depending on the status of the carry/zero flag while comparisons are mostly performed by separate instructions.

MIPS: The MIPS is a 32-bit RISC core implementing the well-known MIPS32 ISA [180]. It features 32 general-purpose registers that are 32-bit wide and two special-purpose register for the multiply–divide unit. Again, the register-offset addressing mode is the only supported one. Conditional

branches can perform the comparison themselves or just depend on the zero-flag status. However, single instructions for (some) comparison operations are supported as well.

For all architectures, the typical set of nonterminals (i.e., register, immediate, addressing mode) is automatically generated. During the initial run most of the resulting mapping rules for all processors get a suitable instruction automatically assigned. To handle the unassigned rules as well, all processors required a few custom transformations and/or mapping rules in the target library. The CPU time used by the generator is negligible. Table 7.2 provides the statistics of the generated nonterminals (NT) and rules for all processors as well as the number of required custom transformations.

Table 7.2 Rule statistics for ST220, PP32, and MIPS

	NT	One-to-one	One-to-many	Many-to-one	Custom rules	Custom trans
ST220	9	176	13	5	4	4
PP32	9	71	19	0	5	6
MIPS	5	49	61	0	4	5

The custom rules and transformations are mainly used for those rules that cannot be executed with a single machine instruction such as the signed/unsigned division, modulo operation (PP32, ST220), and multiplication (PP32). The custom entries in the target library map those to function calls to the runtime library which provides a software implementation to accomplish such operations. For the ST220, an additional transformation is used to perform the one's complement operation with an instruction performing a bitwise *not* and *or* at once. The PP32 also needs some very specific transformations. For example, the load of a 32-bit immediate value has to be performed with two instructions. The first one loads the higher half of the value into the destination register and left shifts the result by 16 bits at the same time. The second one adds the remaining lower 16 bits to the target register. A similar transformation is required for the MIPS. Additionally, the latter needs custom transformations for some compare conditions since they are not available in the MIPS ISA and must be performed in a different way. However, the specification of custom transformation in the target library is an one time effort. Afterward, the complete code selector specification can be generated fully automatically.

7.3 Compiler Evaluation

The following sections evaluate the code quality for the different target architectures. The CoSy compiler with hand-crafted code selector specification is used as baseline for evaluation. The CoSy compiler with generated code selector specification and a non-CoSy-based compiler is compared to it. In case of the ST220, this is the highly optimizing vendor compiler named *ST multiflow* and for the MIPS

the *gcc* [87] based compiler. However, for the PP32, there is no vendor compiler available. Instead, the *lcc* compiler [50] has been manually retargeted to the PP32 as additional reference point. All CoSy-based compilers have been verified using the *SuperTest* compiler validation test suite from ACE [37]. It took several man-weeks to validate the compilers with hand-crafted code selector specification in contrast to the compilers with generated code selector specification which passed the test out of the box.

It can be expected that the compilers with generated code selector specification show a certain overhead in code quality. This is mainly due to the fact that the basic rules are designed to fit for many different architectures and, consequently, might not be optimal for certain target processors. Additionally, the hand-crafted code selector can exploit certain architecture properties, e.g., the integral promotion for some of the C arithmetic operators can be omitted under the assumption that the values in the registers are always correctly sign or zero extended. The generated rules instead must always guarantee the correct behavior and might be too conservative in such cases. Of course, the user can always enrich the target-specific library to improve the generated code selector description. However, except for the custom transformation required to enable the generation of the complete code selector description, optimized target rules are not specified for this evaluation. The concrete overhead for each architecture will be quantified in the following.

7.3.1 PP32

Figures 7.1 and 7.2 show the relative cycle count and code size for seven benchmarks extracted from NPU applications, with the CoSy compiler using the hand-crafted code selector set to 100%. For most benchmarks, the code quality of the compiler generated from the semantic description is close to the hand-crafted version. However, in some cases, a large code quality overhead can be observed. This is mainly caused by the multiplication rules. As mentioned above, some custom

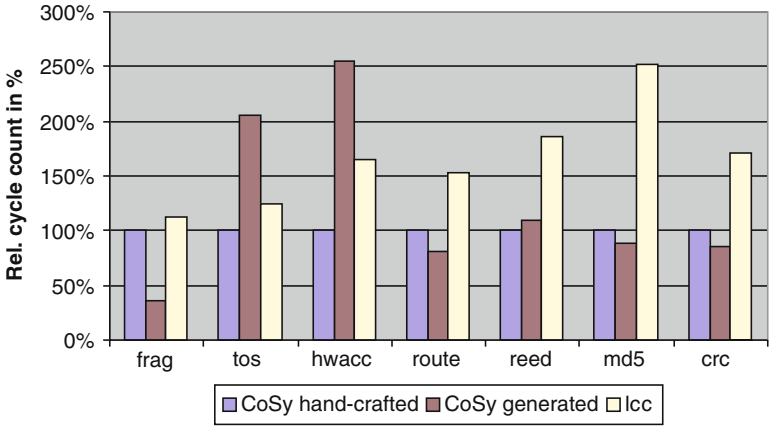


Fig. 7.1 Relative cycle count PP32

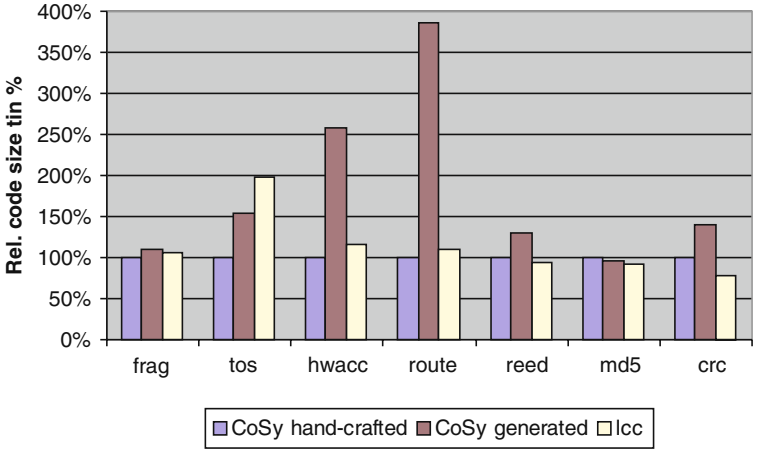


Fig. 7.2 Relative code-size PP32

transformations map the multiplication to a software implementation in the runtime library. This generic approach makes the transformation feasible for many target architectures. The hand-crafted compiler in contrast employs an optimized assembly program for this purpose, which is significantly faster. However, the user could also create custom transformation that yields exactly the same assembly routine for the multiplication (Listing 7.1). But this optimization is usually performed when the architecture exploration phase converges and an initial working compiler is available.

```
ORIGINAL _MULII(REGISTER a, REGISTER b) -> REGISTER c;
SCRATCH t1,t2;
TRANSFORM{
    0 -> t1;
    0 -> t2;
    b -> c;
LLabel_0:
    IF (_EQ(b<0,1>,0)) {
      _ADD(_PC,LLabel_1<0,13>) -> _PC;
    }
    _ADD(t1,a) - > t1;
LLabel_1:
    _LSR(b, 1)  -> b;
    t1<0,1>     -> b<31,1>;
    _LSR(t1, 1) -> t1;
    t1<30,1>    -> t1<31,1>;
    IF (_EQ(_SUB(t2, 1),0)) {
      _ADD(_PC,LLabel_0<0,13>) -> _PC;
    }
    _SUB(t2,1) -> t2;
}
```

Listing 7.1 PP32-specific transformation for multiplication

Thanks to a richer set of built-in code-optimization techniques, the CoSy-based compilers always outperform the *lcc* w.r.t. the cycle count. Since the *lcc*'s code selector basically corresponds to the hand-crafted CoSy compiler, the code size of both compilers is almost the same.

7.3.2 ST220

The picture is different for the ST220. Figures 7.3 and 7.4 illustrate the results for several kernels taken from the DSPstone benchmark suite [124] and a prime number computation based on the *sieve of Eratosthenes*. The code quality of the compiler generated from the semantic description shows on average an overhead of 5% in cycle count and 18% in code size as compared to the hand-crafted version. The overhead is less than for the PP32 first because there is no issue with the multiplication implementation (the ST220 supports multiplication). Second, only few of the one-to-many mapping rules (cf. Table 7.2) have an one-to-one mapping in the hand-crafted version.

Compared to the *ST multiflow* compiler, the CoSy-based compilers show an average overhead of 75% in cycle count and 99% in code size, partially due to extensive function inlining. These are acceptable values, taking into account that the development time for the *ST multiflow* compiler probably was orders of magnitude higher and the CoSy-based compilers are essentially "out-of-the-box" generated compilers without machine-specific optimizations. Analysis of the generated code showed that by adding custom optimization engines, e.g., for exploiting predicated execution, a significantly higher code quality could be easily achieved.

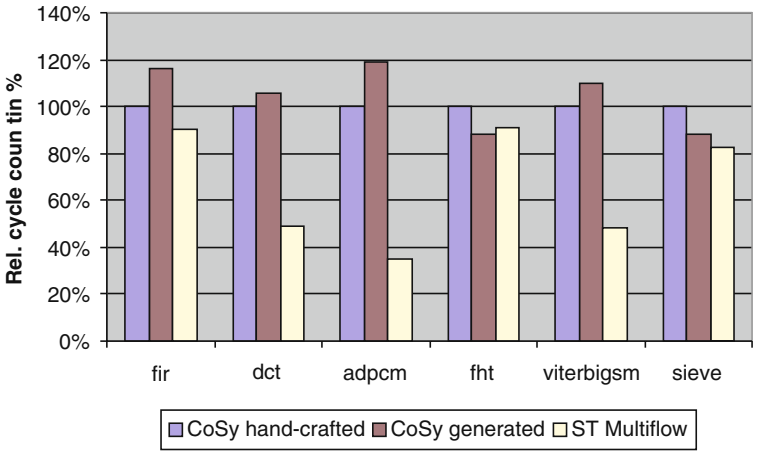


Fig. 7.3 Relative cycle count ST220

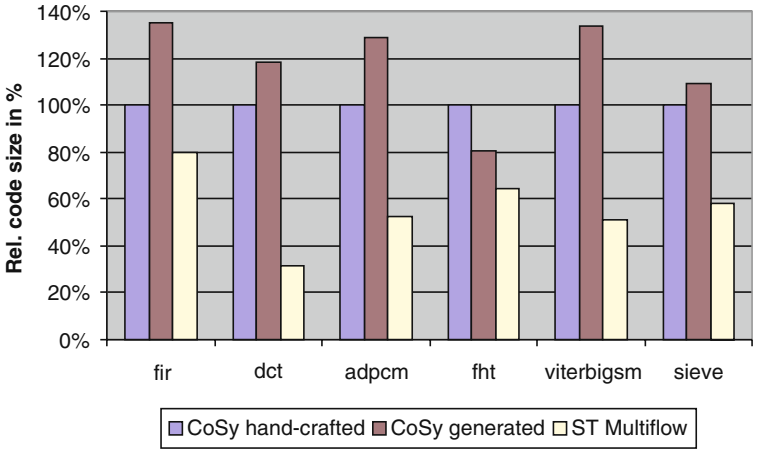


Fig. 7.4 Relative code-size ST220

7.3.3 MIPS

The results for the MIPS, depicted in Figs. 7.5 and 7.6, show a similar picture as for the PP32. Apart from the benchmarks as used for the ST220, larger kernels from different benchmark suites [53, 154] or applications [196, 265] have been chosen. The compiler generated from the semantic descriptions shows an average overhead of 88% in cycle count and 45% in code size. In contrast to the previous hand-crafted CoSy compilers, a considerable amount of work has been spent in the code selector specification for the MIPS. In another context, it was evaluated how close a CoSy compiler generated by the *Compiler Designer* can come to a production quality

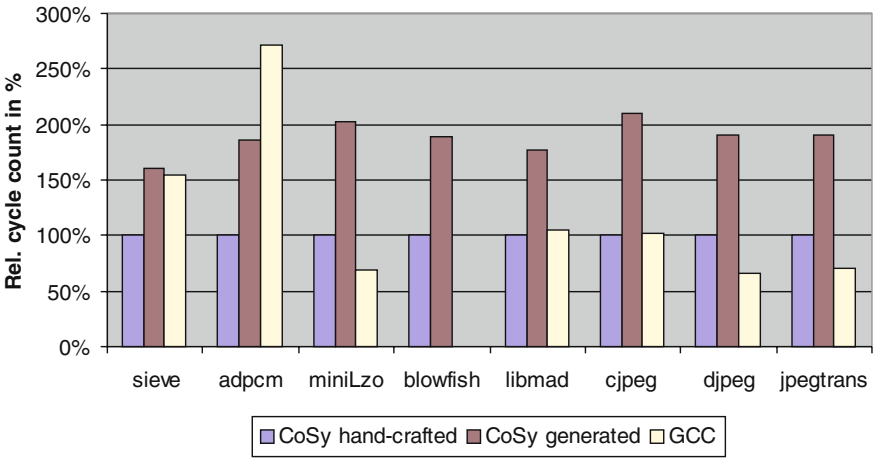


Fig. 7.5 Relative cycle count MIPS

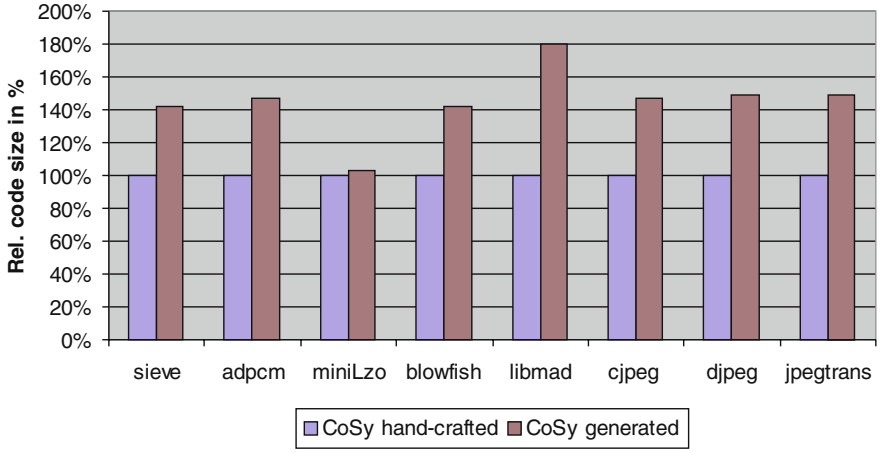


Fig. 7.6 Relative code-size MIPS

compiler. Consequently, a larger overhead for the semantic-based compiler can be observed. The hand-crafted compiler shows only an overhead of 5% in cycle count as compared to the *gcc*. Code-size numbers for the *gcc* are omitted since it uses a different runtime setup (i.e. functionality that is linked to the executable to setup the runtime environment) which leads to a significantly different code size.

7.4 Conclusions

Designing an ADL that in particular serves the purpose of C compiler *and* simulator generation from a single model is quite challenging (cf. Chapter 4). Typically, this leads either to a loss in modeling flexibility or introduces a huge potential for inconsistencies. This book presents an approach for the LISA ADL that avoids both. It incorporates a new SEMANTICS section into the LISA language definition which achieves a concise formalism for the description of instruction semantics without influencing the existing flexibility. This information is used by four different mapping rule generation methods which create the code selector description for a C compiler fully automatically. In this way, even noncompiler experts are capable of generating C compilers for early architecture exploration. Manually created code selector descriptions are a typical source of errors, but the generated code selector rules are correct by construction. Hence, a significant verification and debug effort is saved.

Although using a semantics description introduces certain redundancies, they are kept minimal in the model. Note that apart from code selector generation, it is also possible to generate an instruction-set simulator and documentation with the information provided by the SEMANTICS sections [91]. Since the semantics description is much simpler than the C/C++ description, this helps accelerating the modeling process in early architecture exploration when the concrete micro-architecture

is not fully determined. However, a detailed discussion of the simulator generation is beyond the scope of this thesis.

From the above case studies, it should be obvious that the flexibility of the new SEMANTICS section w.r.t. feasible target architecture classes is not a major concern in this approach. Furthermore, C compilers can now be generated fully automatically from LISA models with SEMANTICS sections. Such an integrated approach, based on only a single "golden" target processor model, is key for an effective ASIP design environment. The resulting lower code quality of the generated compilers is acceptable considering that the C compiler is available right from the beginning.

Compared to compiler generation with a pure stand-alone system such as CoSy or with the *Compiler Designer* without code selector generation, the compiler description effort is reduced to a minimum. Moreover, the presented approach hides even more compiler technology internals from the ASIP design engineer, who thus can better concentrate on architecture optimization. Another advantage is that the code selector rules are correct by construction. This eliminates a prominent source of errors in compiler descriptions.

The code quality of the generated compilers can only be considered as a result from "out-of-the-box" compilers. Analysis of the generated code showed that by adding custom optimization engines, e.g., for exploiting predicated execution, significantly higher code quality could be easily achieved, though, at the expense of higher manual effort. Furthermore, while the integration of high-level optimizations into retargetable compilers is mostly supported, this is not the case for low-level or assembly-level optimizations. Most generated assemblers do not offer the opportunity to plug-in user-defined optimizations. Therefore, the remainder of this book focuses on two topics:

- Retargetable optimization techniques for common ASIP extensions to further narrow the code quality gap while reducing compiler design effort.
- A new retargetable assembler provides an implementation interface to quickly develop user-defined optimization techniques.

Chapter 8
SIMD Optimization

As concluded in the previous chapter, retargetable compilers, as used in ASIP design environments, are still hampered by their limited code quality as compared to hand-written compilers or assembly code. Consequently, generated compilers must be *manually* refined to a highly optimizing compiler after successful architecture exploration. One way of overcoming this dilemma is to design *retargetable optimizations* for those architectural features that characterize a class of target processors.

This chapter focuses on target processors equipped with *SIMD instructions*. The term *SIMD* dates back to the year 1972 when Flynn [160] classified computers according to the number of data streams they operate on, and the number of instructions they execute (Table 8.1). The acronym *SIMD* stands for *single-instruction multiple data* and the class of computers referred to in the 1970s were vector computers that were able to execute the same operation on multiple vector elements at the same time.

Table 8.1 Flynn's classification

	Single instruction	Multiple instructions
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Today the meaning of the term has slightly changed. It usually denotes a special class of instructions found in many workstation and embedded processors that operate on short vectors of small data. As illustrated in Fig. 8.1, an SIMD instruction performs several primitive operations in parallel, using operands from several *subregisters* of the processor's data registers at a time. The operands are typically 8-, 16-, or even 32-bit wide. In future, the SIMD data paths might even grow larger with the advances in semiconductor technology. Other typical SIMD instructions perform more complex operations (e.g., partial dot products) or serve for subregister packing and permutation. From a hardware perspective, SIMD instructions are easy to control and have a simple structure (the existing data path is basically just split) without extra register file ports. This makes them inherently simple and thus keeps the hardware cost low. Meanwhile, they can provide significant performance

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_8, © Springer Science+Business Media, LLC 2010

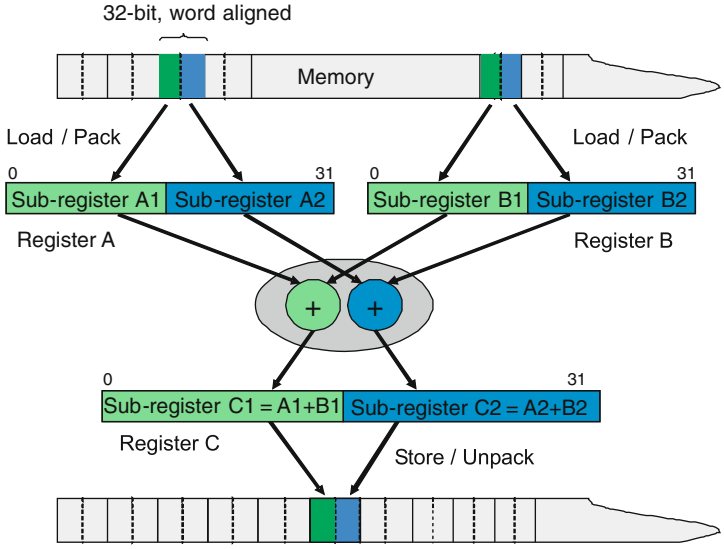


Fig. 8.1 Sample arithmetic SIMD instruction: two parallel ADDs on 16-bit subregisters of 32-bit data registers A, B, and C; the data is loaded/stored at once from/to an alignment boundary

improvements for computation-intensive multimedia workloads [145]. Therefore, many embedded processors for the next generation of high-end video and multimedia devices today feature SIMD instructions.

The SIMD concept is commonly found in general-purpose architectures such as Intel MMX/SSE1–5 [30], IBM/Motorola VMX/AltiVec [183], and AMD 3DNow. Later on, it was introduced in domain-specific processors (e.g., TI C6x, NXP TriMedia) and in recent custom ASIP designs (e.g., Tensilica Xtensa). Even some versions of the popular ARM- and MIPS-based architectures feature SIMD instructions. While several target-specific C compilers already exploit SIMD instructions, there is almost no support in ASIP compilers. Consequently, there is an increasing interest in retargetable compilers with SIMD support. For use in this domain, *retargetable SIMD optimizations* are required. This chapter presents a novel concept for retargetable code optimization for ASIPs with SIMD instructions, and this concept is proven by an implementation within the CoSy compiler that can be retargeted via the *Compiler Designer* GUI and an experimental evaluation for two real-life embedded processors.

The rest of this chapter is organized as follows. In Section 8.1 related work is discussed. The core of the SIMD framework is presented in Section 8.2 before the retargeting procedure is described in Section 8.3. Afterward, Section 8.4 provides the experiments for different embedded processors with SIMD support. Finally, Section 8.5 summarizes the contribution of this approach and points to some future avenues of work.

8.1 Related Work

Traditional code selection typically relies on tree parsing. As mentioned in Section 3.3.2, tree parsing is not suited to exploit SIMD instructions because they exceed the scope of a single DFT. Consequently, compilers require advanced techniques to exploit SIMD instructions.

Most of the current SIMD optimization techniques are based on the traditional loop-based vectorization [24, 95, 212, 213]. Others make use of instruction-packing techniques in conjunction with loop-unrolling to exploit data parallelism within a basic block [240] or a combination of traditional code selection [51] and integer linear programming [26, 221]. As investigated in [101], it is often difficult to apply SIMD optimization techniques since these architectures are largely nonuniform, featuring specialized functionalities, constrained memory accesses, and a limited set of data types. Moreover, complicated loop transformation techniques are needed [213] to exhibit the necessary, architecture-dependent amount of parallelism in the code. Another hurdle to applying SIMD techniques is packing of data elements into registers and the limitations of the SIMD memory unit: typically, SIMD memory units provide access only to contiguous memory elements, often with additional alignment constraints. Computations, however, may access the memory in an order that is neither adequately aligned nor contiguous. Besides, operations on disjoint vector elements are usually not supported. The detection of misaligned pointer references is presented in [117]. Certain misalignments can be solved either by loop transformations [95, 241] or by data permutation instructions. The efficient representation and generation of such instructions is investigated in [7, 72, 212] and the optimization thereof in [26, 102]. Consequently, only a successful interaction of several optimization modules will be able to leverage SIMD optimization for retargetable compilers.

So far, only advanced compilers (e.g., the Intel compiler [122], IBM XL compiler [7]) are capable of automatically utilizing SIMD instructions. Apart from being inherently nonretargetable, these compilers are mostly restricted to certain C language constructs. Other compilers use dedicated input languages for source-to-source transformations that are restricted to a certain application domain [83, 188]. The vast majority of the compilers, though, still provide only semi-automatic SIMD support via *compiler-known functions* (CKFs). Understandably, this assembly-like programming style is tedious and error prone. Moreover, this comes along with poor maintainability and portability of the code.

Among the ASIP design platforms mentioned in Chapter 4, so far only Tensilica's compiler includes SIMD support. However, its architectural scope is limited to the configurable Xtensa processor [215]. Considering retargetable compilers, recent versions of the *gcc* support SIMD for certain loop constructs [86]. The supported vectorization [71] features alignment and reduction; however, information regarding the concrete retargeting effort and the interaction of loop transformations are not available yet. Furthermore, *gcc* is mainly designed for general purpose processors. As a result, it does not adapt efficiently to specialized, irregular hardware architectures that are quite common in the embedded domain.

A retargetable preprocessor for multimedia instructions is presented in [100]. The approach mixes loop distribution, unrolling, and pattern matching to exploit SIMD instructions. Contrary to other approaches, it can be extended at user level. The matching is based on a set of target-specific code-rewrite rules that are described using C-code patterns. However, the efficiency of this approach strongly depends on the coding style of the input program. Furthermore, no information is available how the loop transformations are adapted to a given SIMD architecture.

Summarized, several SIMD utilization concepts with different levels of complexity are available. However, they are mostly implemented in target-specific compilers. Consequently, adapting a SIMD optimization concept to a new target processor becomes a time-consuming and error-prone manual process. Therefore, this book presents an approach for the *efficient utilization of SIMD instructions* while achieving *compiler retargetability* at the same time. The presented SIMD optimization comprises a *loop-vectorizer* and an *unroll-and-pack*-based technique [166], which are both driven by the same SIMD specification. The retargeting formalism is fully integrated into the compiler backend specification. The advantage is that many generators for the standard backend components (e.g., the code selector) can be reused for the SIMD optimization to a great extent. This reduces the retargeting effort and enables greater flexibility to specify the SIMD architecture. The amount of required target-specific information is limited, so that most of it can be extracted automatically from ADL descriptions such as LISA. Moreover, the retargeting information is also used to steer the *loop transformations*, such as unrolling and strip mining, required to exhibit the necessary (i.e., SIMD architecture dependent) amount of parallelism and to deal with memory alignment issues. In sum, this provides a flexible and efficient *SIMD optimization framework* for a wide variety of SIMD architectures.

8.2 SIMD Framework

As mentioned above, a successful SIMD optimization is tightly coupled with several loop transformations in order to exhibit the necessary amount of parallelism and to convert loops into a proper form. Hence, the presented approach consists of several steps as depicted in Fig. 8.2.

First of all, a *loop-carried dependency* [178] and *alignment analysis* (Section 8.2.3) are performed. They provide the necessary annotation needed by the SIMD optimization framework. Afterward, a *SIMD analysis* (Section 8.2.4) searches for loops where SIMD optimization could be applied. For these loops, it determines the parameters for the different *loop transformations* (Sections 8.2.5, 8.2.6, 8.2.7, and 8.2.8). Finally, the SIMD optimization is performed, comprising a loop *vectorizer* (Section 8.2.7) or an unroll-and-pack-based *SIMDfyer* (Section 8.2.9) if vectorization fails. All modules are driven by the same, *retargetable* SIMD specification described in Section 8.3.

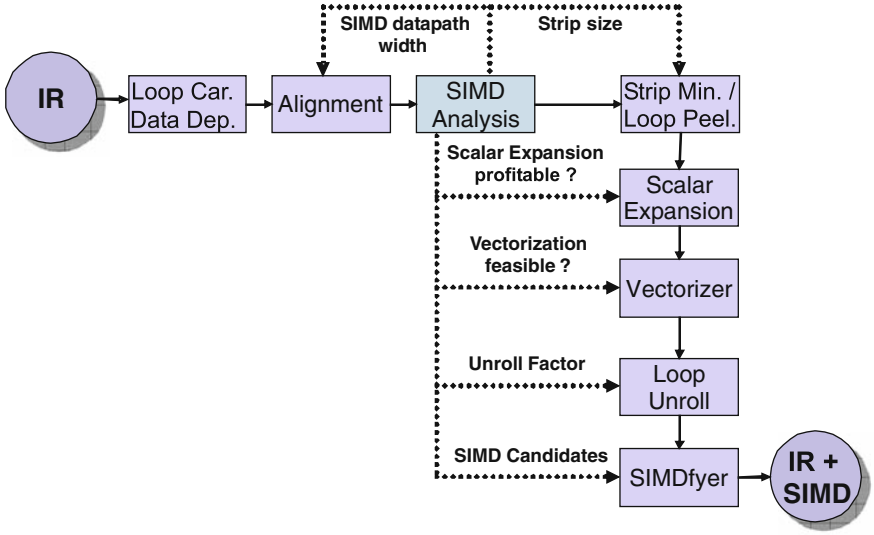


Fig. 8.2 SIMD code generation flow

8.2.1 Basic Design Decisions

A basic design decision concerns the representation of generated SIMD instructions in the compiler's IR. All IR formats comprise elements for representing primitive operations such as addition, subtraction, multiplication, and so on. However, there are usually no dedicated IR elements for SIMD operations such as "two parallel additions." Extending the underlying IR format is not a practicable solution. All already existing compiler engines would have to be manually adapted in order to handle the new IR elements. Otherwise compiler engines might not exploit the full optimization potential or may even fail in the worst case. In either case, poor code quality would be the result. Therefore, generated SIMD instructions are internally represented in the form of CKFs. CKFs are transparent for other compiler modules and are later automatically replaced with assembly instructions in the backend. They are not visible to the compiler user at all. Furthermore, CKFs simplifies code generation to a certain extent, since it abstracts from low-level problems such as register allocation for SIMD subregisters in the backend. Moreover, all existing code generation and optimization engines of the underlying compiler framework can simply be reused. This includes the existing debug facilities of the compiler platform. In this way, the current IR state can be dumped into a human-readable, valid C-code file at any time during the SIMD generation process.

8.2.2 Terminology

Here, the terminology that facilitates the description of the optimization modules in the next sections is briefly introduced. As exemplified in Fig. 8.1, an SIMD instruction performs independent, usually identical operations on a certain bit range within the input register and also writing the results to a corresponding range in the output register. In other words, an SIMD instruction splits a *full* register into k *subregisters* (frequently $k = 2$ or $k = 4$). In the given example, the lower and upper parts of the arguments are added and written to the lower and upper part of the destination register, respectively. Thus, this SIMD instruction operates on two subregisters. A single, primitive operation within the SIMD instruction (e.g., the 16-bit addition) is denoted as an *SIMD candidate*. It is basically a mapping rule covering this primitive operation. From these mapping rules, an *SIMD-candidate matcher* (Section 8.3.1) is generated (i.e., a regular tree pattern matcher) that is used for the identification of such SIMD candidates.

A set of SIMD candidates that can be combined into a SIMD instruction is denoted as an *SIMD-set*. For this purpose, a generated *SIMD-set constructor* is employed (Section 8.3.2). This is basically a combination function that tries to collect suitable SIMD candidates under given constraints such that a valid SIMD-set can be built. The algorithm for SIMD-set constructions assumes that the results from the data-flow analysis are already available. Next, it checks a number of constraints for tuples $N = (n_1, \ldots, n_k)$ of SIMD candidates, where k denotes the number of subregisters, and nodes n_i of a potential SIMD-set must

1. Represent *isomorphic operations* that can be combined to a SIMD instruction according to the target machine description;
2. Show no direct or indirect *dependencies* that would prevent parallelism. While this can be analyzed relatively simple for scalar variables, it becomes quite difficult in the case of array and pointer accesses.
3. Fulfill *alignment* constraints of the given target architecture. The data elements in memory must be packed in a single register in advance before the SIMD instruction can be executed. This involves wide load instructions, and hence possibly memory alignment constraints as well as reordering of subregister within a register using special *pack* and *permute* instructions. The same holds for storing the SIMD result again in memory.

A constructed SIMD-set (i.e., the related IR nodes) can then be replaced by a CKF call. The regular code selector description is enriched with CKF mapping rules so that later during the code-emission phase the proper assembly code for the SIMD instruction can be emitted.

8.2.3 Alignment Analysis

One of the constraints when using SIMD instructions is the correct *alignment* of data in memory. In opposition to the original vector machines, which usually were

equipped with superscalar memory units, the SIMD enabled general-purpose and embedded processors usually have a scalar memory unit. Parallel data loading is nevertheless possible as long as the data stream is stored contiguously in memory. For example, a twofold SIMD instruction operating on 16-bit data types typically uses a 32-bit wide, word-aligned load operation to pack them at once in a 32-bit register (Fig. 8.3).

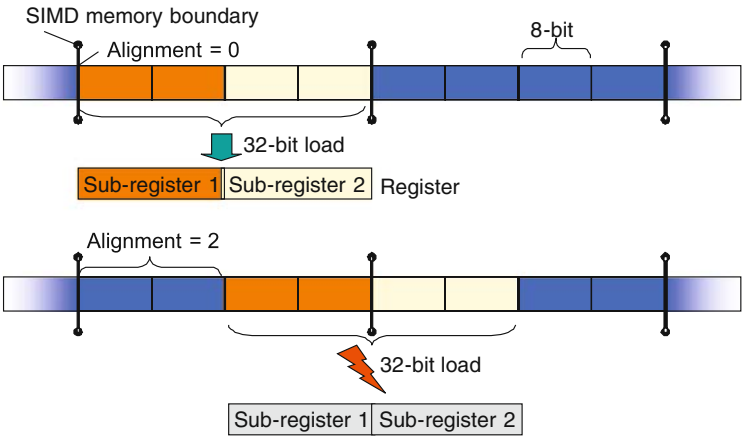


Fig. 8.3 SIMD alignment constraint

This is the optimal case, since the data is already available in the desired format. If however the data is locally disjoint, the required values have to be explicitly *packed* to the register before they are susceptible to SIMD optimizations. The two half-words would have to be loaded into two distinct registers, by doing two separate word-wide loads. In a second step, they can then be combined into a third register. Instead of doing a single load using a single register, at least two registers are used, and two separate loads, as well as an operation to merge the two half-words back into one register, have to be carried out. Even though many architectures offer support instructions such as permutations, multi-register shift operations, subword selection and general pack and unpack operations, the necessity of using them usually incurs a performance hit.

If the word alignment cannot be assured at *compile-time*, additional code (i.e., a *dynamic-alignment check*) is required to ensure correct alignment during *run-time* [35, 117]. This procedure, also known as loop versioning, creates an optimized version of the code along with the original version. At runtime, a check as seen in Listing 8.1 is executed that selects the right version depending on the initial alignment.

```
if( (a is aligned) && (b is aligned) && (c is aligned) )
{
  for(i=0;i<N;i+=4) /* SIMD version */
  {
    c[i:i+4] = a[i:i+4] * b[i:i+4];
  }
} else {
  for(i=0;i<N;i++) /* Standard version */
  {
    c[i] = a[i] * b[i];
  }
}
```

Listing 8.1 Dynamic alignment check = 0

This version generates code that is always correct but obviously has the following two major drawbacks:

1. It increases the code size by more than a factor 2 for the loop.
2. It incurs the runtime overhead of the alignment check, which noticeably hurts performance for small iteration counts.

The strip-mining transformation (Section 8.2.5) needs to take the alignment into account, too. Therefore, an *interprocedural pointer-alignment analysis* [82] has been implemented for precise alignment information. It analyzes every memory access performed through pointers with respect to the capabilities of the SIMD memory unit. The offset from the supported SIMD memory boundary, that is, the alignment, is calculated using the modulo operator. If p is a pointer and N the SIMD memory address size, then the *alignment* of the memory access is given by

$$alignment = p \bmod N \tag{8.1}$$

In order to account for the possibility that a pointer might have, during program execution, values with different alignments, the information is stored as a set E of possible values *modulo N*. If $M = \{0, \ldots, N-1\}$ is the set of all possible values of modulo N and $P = P(M)$ its power set, then $E \in P$.

In order to correctly annotate pointers in the whole program, it is necessary to track the value of pointer variables during their whole lifetime. A pointer generally is:

1. First initialized, usually by means of a memory management function such as `malloc` or by taking the address of an variable object.
2. Used, either directly or in address calculations such as `*(p+i)`, to access values in memory.
3. Manipulated or used in address calculations that are then stored to another pointer variable, which leads to a new initialization (e.g., `p = p + i`).

The analysis therefore needs the ability to determine the initial alignment of pointers. To do so, it needs specific knowledge about the possible initial sources of addresses. In the case of direct initialization by taking a memory address, this is possible using information about the variable object. In case of functions that take pointers as arguments, the initial values of the pointer parameters are not available inside the function. It is therefore sensible to use an interprocedural algorithm, which propagates the information across function boundaries. Next, this value must be tracked from its first *definition* to all its *uses*. This is a classic data-flow problem that can be solved using standard techniques as described in Chapter 3.

The third prerequisite to successfully uncover the alignment information in pointers is the ability to determine the offset for accesses that involve address calculations. In order to evaluate pointer arithmetic such as `*(p+i)`, a transfer function

$$f_g : P^n \longmapsto P \tag{8.2}$$

is used to compute the impact on E. The transfer function, naturally, depends on the operator of the arithmetic expression. For example, the most common operations in address calculation, the addition and multiplication, are binary operators, and thus the corresponding transfer functions have the form $f_{\text{binary}} : M \times M \mapsto M$. This leads to the following equations:

$$\begin{aligned} f_{\text{Add}}(a, b) &= (a + b) \bmod N = [(a \bmod N) + (b \bmod N)] \bmod N \\ f_{\text{Mul}}(a, b) &= (a \cdot b) \bmod N = [(a \bmod N) \cdot (b \bmod N)] \bmod N \end{aligned} \tag{8.3}$$

They are valid regardless of the value of N. If, however, $N = 2^m$ is a power of two, further functions can be deduced. This is due to the fact that a division by 2^m can be implemented by right shifting the binary representation of an integer value m times. The remainder of the division is then exactly formed by the m bits shifted out of the word. Therefore, it is in the last m bits of the original value. Using this knowledge, the operations AND, OR, XOR, and NOT can be handled without knowledge about the actual value as well.

8.2.4 SIMD Analysis

The preparative loop transformations consist of *strip mining*, *scalar expansion*, and *loop unrolling*. They must be parameterized according to the underlying SIMD architecture. Incorrect parameters might prevent SIMD optimization or lead to nonoptimal results. The transformations often only pays off, if the SIMD optimization is later on enabled. Therefore, it is important to apply them only to the most promising loops for SIMD optimization. Hence, an *SIMD analysis* engine is implemented that runs in advance to identify those loops that contain SIMD candidates. For this purpose, the *SIMD-candidate matcher* is employed. Consequently, if the loop body does not contain any SIMD candidate, then it does not make sense to

consider it further. Otherwise it determines for each SIMD candidate how many of them would be needed to build a SIMD-set that matches one of the available SIMD instructions using the *SIMD-set constructor*. From this information, it derives the parameters for the different loop transformations.

8.2.5 Strip Mining and Loop Peeling

Many vectorizable loops cannot be directly optimized in case the iteration count is larger than the number of SIMD candidates k_s that fit into an SIMD-set s for the vector operation. *Strip mining* is a loop transformation that divides the loop into strips, where each strip is no longer than the SIMD data path width [178]. Essentially, the loop is decomposed into two nested loops (Listing 8.2):

1. An outer loop (the *strip loop*) that steps between strips.
2. An inner loop (the *element loop*) that steps between single iterations within a loop.

```
// original loop
for (i = 0; i < 100; i++)
{
  A[i] = B[i] * C[i];
}
//outer strip loop
//strip_size = max. #sub-registers
for (is = 0; is<100; is += strip_size)
{ //inner element loop
  for (i=is; i<is+strip_size; i++)
  {
    A[i] =  B[i] * C[i];
  }
}
```

Listing 8.2 Strip mining example

The *SIMD analysis* calculates the iteration count of the element loop, called the *strip size*, based upon all SIMD-sets S that can be built with the identified SIMD candidates in the loop. Since it might happen that each SIMD-set has a different number of subregisters k, the maximum strip size for the transformation is selected:

$$strip\_size = \max\left(\bigcup_{s \in S} k_s\right) \tag{8.4}$$

However, due to possible alignment constrains of the SIMD architecture, strip mining must ensure that each strip starts at an *alignment boundary*. Assuming that arrays

are word aligned in memory, then the alignment boundaries are given by

$$alignment\_boundaries = \{i \mid i \bmod strip\_size = 0\} \tag{8.5}$$

where i is the loop counter. However, strip mining is performed in the iteration space. Thus, for array references like $[i+c]$ with c being a constant and $c \neq 0$ (Listing 8.3), the alignment boundary for each strip can differ from the real alignment in memory.

```
for (i = 0;i < 100;i++)
{
  A[i+1] = B[i+1] * C[i+1];
}
```

Listing 8.3 Offset = 1

Therefore, an *offset* can be set, if it remains constant within the loop, to readjust the alignment boundaries defined in the iteration space so that they correspond with the real alignment in memory. Consequently, the offset is always within the range $(-strip\_size, strip\_size)$. The alignment boundary is then given by

$$alignment\_boundaries = \{i \mid i + offset \bmod strip\_size = 0\} \tag{8.6}$$

The boundary information can be easily computed using the information from the alignment analysis. In case the loop does not directly start at an alignment boundary, *loop peeling* is applied to ensure the correct alignment of the data accesses. That means, those iterations causing the misalignment are "peeled off" the original loop and build a separate prolog loop. If the remaining iterations are not divisible by the strip size without remainder, then an extra epilog loop is created as well. Assuming an up-counting loop using a less-than condition, the loop boundaries for the prolog, strip loop, and epilog are defined as follows:

$$bFrom = iFrom + (-(iFrom + offset) \bmod strip\_size) \tag{8.7}$$

$$bTo = iTo - ((iTo + offset) \bmod strip\_size) \tag{8.8}$$

Listing 8.4 shows a generalized example. The initial and final value of the loop counter are given by *iFrom* and *iTo*, respectively, where *bFrom* defines the initial value of the strip loop and the upper bound of the prolog, and *bTo* the upper bound of the strip loop and the initial value of the epilog. Note that the modulo operation must produce a value in the range $[0, strip\_size)$. Furthermore, it must take care of overflows that might occur during the computation of the loop boundaries. Similar equations exist for different conditions and down-counting loops.

```
// peeled iterations (prologue)
for (i = iFrom; i < bFrom; i++)
{
  A[i+c] =  B[i+c] * C[i+c];
}
//strip mined loop
for (is = bFrom
     is < bTo;
     is += strip_size)
{
  for (i = is; i < is+strip_size; i+=1)
  {
    A[i+c] = B[i+c] * C[i+c];
  }
}
//epilogue loop
for (i = bTo; i < iTo; i++)
{
  A[i+c] =  B[i+c] * C[i+c];
}
```

Listing 8.4 Strip mining with offset != 0

8.2.6 Scalar Expansion

When scalars are assigned and later used in the loop, the dependency graph will include flow-dependence relations from the assignment to each use-and-loop-carried anti-dependencies from each use back to the assignment. These anti-dependence relations often cause problems in other transformations and could prevent parallelization of the loop (Listing 8.5). However, the anti-dependence relation can be broken by *scalar expansion* [178]. The basic idea is to allocate an array with one element for each iteration and replace each scalar reference in the loop with a reference to the array. This eliminates the anti-dependence relations. The computed value should be assigned to the original scalar after the loop (Listing 8.6). Scalars that are assigned conditionally can also be expanded given that

1. the scalar is assigned on every path through the loop body and
2. the scalar is not used before any assignment to the same scalar.

If a scalar is found that satisfies these constraints, it is replaced by an array access.

One obvious drawback of scalar expansion, though, is the increased memory consumption of the program. If not carefully managed, this penalty can overcome the benefits gained by SIMD. For instance, the memory usage can be reduced by strip mining the loop and only expanding the inner element loop.

```
for (i=0; i < N; i++)
{
  s = B[i] * C[i];
  A[i] = s+1/s;
}
```

Listing 8.5 Scalar causes anti-dependence

```
for (i=0; i<= N; i++)
{
  S[i] = B[i] * C[i];
  A[i] = S[i]+1/S[i];
}
s = S[N];
```

Listing 8.6 Replaced scalar with array access

8.2.7 The Vectorizer

A classical *vectorizer* parallelizes the whole loop at once provided that suitable SIMD instructions are available for *all* statements in the loop body and no data dependencies limit parallelization. Another prerequisite is that the iteration count must match the number of SIMD candidates needed to build the SIMD-set for the vector operation. Obviously, this is a perfect match for strip-mined loops. The vectorization algorithm is exemplified in Fig. 8.4. In the first step (1), it checks all inner loops whether each statement consists only of SIMD candidates using the *SIMD-candidate matcher*. In step (2), it virtually duplicates the SIMD candidates according to the iteration count of the current loop. For these virtual SIMD candidates, it tries then to construct an SIMD-set that matches an available SIMD instruction with the *SIMD-set constructor* (3). Finally, if valid SIMD-sets can be constructed for each statement, then the whole loop will be replaced by the corresponding SIMD instructions (4).

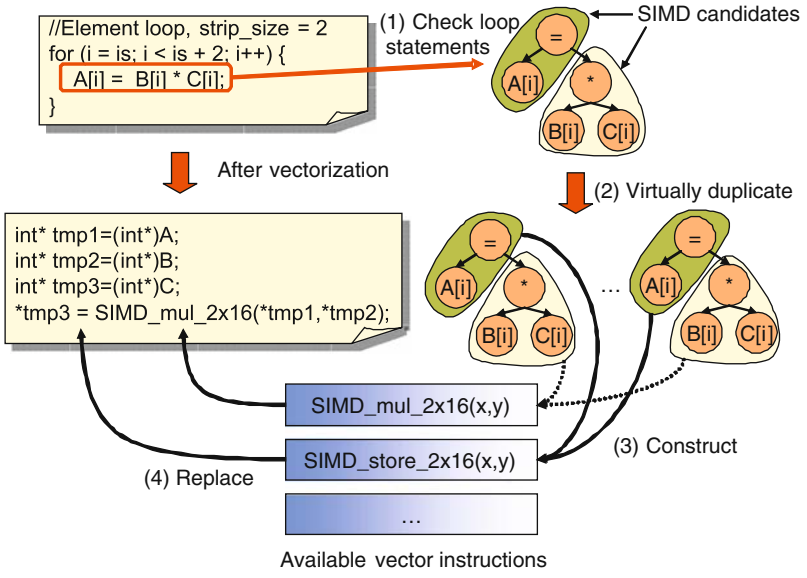


Fig. 8.4 Vectorization example

Of course, it might happen that not all loop statements can be directly parallelized, e.g., due to data dependencies. But still they may contain a certain degree of parallelism. Therefore, loops that could not be vectorized are further processed by the more powerful unroll-and-pack-based *SIMDfyer*.

8.2.8 Loop Unrolling

The *SIMDfyer* implements a technique similar to [240]. This requires loops to be unrolled properly to ensure full utilization of the SIMD data path. The *SIMD analysis* customizes the *unroll factor* to the number of SIMD candidates k_s that fit into a SIMD-set s that can be constructed for the given loop body. This is basically the same as for the strip-size calculation. Consequently, strip-mined loops will be unrolled completely if they are not vectorized. It may happen that the loop contains several SIMD candidates, which can be combined in different ways to an SIMD-set. Thus, since it is desired to fill all possible SIMD-sets S, the best unroll factor can be calculated as

$$unroll\_factor = \max\left(\bigcup_{s \in S} k_s\right) \tag{8.9}$$

The *SIMD analysis* annotates the unroll factor to each loop that contains SIMD candidates. The value of all loops left after vectorization will be read by the *loop unroller* to prepare them for the *SIMDfyer*.

8.2.9 The Unroll-and-Pack-Based SIMDfyer

For a given IR of an input C program, an iterative algorithm is used that combines SIMD candidates into SIMD-sets and replaces such sets by CKFs in the IR [55]. Even though the algorithm could in principle process all basic blocks inside a procedure, it focuses only on the loops, typically the hot spots of the input program; more specifically, only those where the *SIMD analysis* identified SIMD candidates before. Certain multiple basic block constructs, though, may have been merged into a single basic block by an *if-conversion* [125] pass prior to the SIMD optimization. The algorithm forms SIMD instructions step by step. If a complete SIMD-set could be built, it will be replaced by the corresponding CKF. Since each iteration may generate new SIMD candidates, the list of SIMD candidates is updated after each step. The identification of SIMD candidates is performed by the *SIMD-candidate matcher*. The basic idea of the iteration is illustrated in Fig. 8.5.

State (1) shows the initial IR structure for a sample loop body (unrolled twice) that performs a multiplication of two vectors B and C and stores the result in vector A. The left and right elements of the computations are isomorphic and are assumed to meet the memory alignment constraints. First, the algorithm combines the left

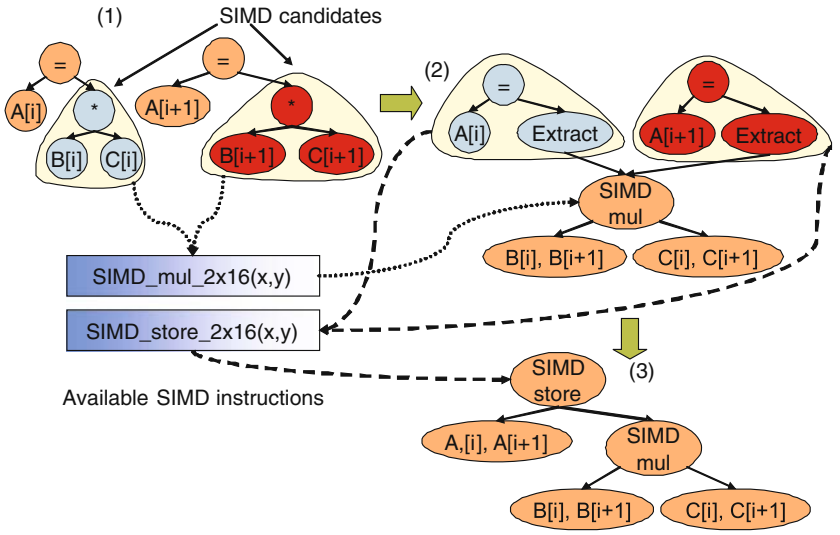


Fig. 8.5 IR states in different iterations

and the right operands (16-bit load operations) of the two "*" to 32-bit SIMD load operations. Afterward, the "*" operations themselves are combined to an SIMD instruction. The corresponding IR has the intermediate state (2). In order to preserve the semantic correctness, explicit "extract" operations are inserted that select 16-bit subwords out of the 32-bit result of the SIMD dual multiplication operation. These extracts are also considered as SIMD candidates, and hence can also be used to build an SIMD-set. Note, all superfluous extracts are removed by *dead code elimination* in a later compilation phase. In the following iteration, the two 16-bit "=" operations form an SIMD-set on their own. Finally, the IR state (3) is reached and the algorithm terminates.

The presented approach employs an iterative, step-by-step approach in order to compose an SIMD instruction from a set of SIMD candidates. In this way, an exhaustive search within the given loop body is avoided. Therefore, it requires only low-degree polynomial complexity ($O(n^3)$), a worst case for n variable accesses in the IR. Practical experience shows that this relatively simple heuristic consumes only a few CPU seconds of compilation time while utilizing SIMD instructions very well for speeding up common DSP code benchmarks. Due to the possible necessity of inserting extra code for dynamic pointer-alignment checks before loop entry points and the corresponding code duplication, insertion of SIMD instructions may lead to an increase in code size.

8.2.10 Code Example

This section provides a more detailed example to illustrate the representation of SIMD instructions in the IR. Listing 8.7 shows the initial C source code after

preprocessing (strip mining, scalar expansion, and loop unrolling). Assuming the availability of SIMD instructions for *addition* and *multiplication* operating on two 16-bit values, the *SIMD analysis* determines a strip size and an unroll factor of 2 for the loop transformations. Here, scalar expansion is performed on the element loop, which is then fully unrolled afterwards. It is further assumed that the target machine requires SIMD load operations to be *word aligned*.

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  sum = S[0] = S[1] = 0;
  for(int is = 0; is < 64; is += 2)
  {
    S[0] = S[0] + (*pa * *pb) * *pc;
    pa++; pb++; pc++;
    S[1] = S[1] + (*pa * *pb) * *pc;
    pa++; pb++; pc++;
  }
  sum = sum + S[0] + S[1];
}
```

Listing 8.7 Initial code

In the first iteration, the two multiplications are detected as SIMD candidates and are replaced by a CKF (`SIMD_mul_2x16`). The SIMD multiplication implies certain conditions in which subregisters the input operands must be located in. Since the input operands are given by the extract operations from the previous iteration, these conditions can be easily met by directly using the temporaries the input operands are extracted from. Obviously, this makes the extract operations from the previous iteration superfluous. The resulting code is depicted in Listing 8.9 while Listing 8.10 shows the final code after several further steps. The SIMD-set computation has been finalized by detecting that the multiply results can be processed further by SIMD additions. No extract operations are required since the results can be directly written by a wide store to the array created by scalar expansion. Here, it is assumed that the alignment analysis cannot resolve the alignment of the pointers, thus a dynamic alignment check has been inserted (`if(((pa|pb|pc) & 3) == 0)`) to rule out misaligned pointers. If the check fails, a non-SIMD version of the loop is executed in the else-branch. Finally, standard optimizations, such as *dead code elimination*, have been invoked to remove superfluous operations (e.g., extracts) from previous phases. The resulting code is passed to the compiler backend for assembly code generation.

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  int tmp1, tmp2;
  short res0, res1, res2, res3;
  sum = S[0] = S[1] = 0;

  for (int is = 0; is < 64; is += 2)
  {
    tmp1 = (int*)pa; //SIMD load
    tmp2 = (int*)pb; //SIMD load
    res0 = EXTRACT_short_1_of_2(tmp1);
    res1 = EXTRACT_short_2_of_2(tmp1);
    res2 = EXTRACT_short_1_of_2(tmp2);
    res3 = EXTRACT_short_2_of_2(tmp2);

    S[0] = S[0] + (res0 * res2) * *pc;
    pa++; pb++; pc++;
    S[1] = S[1] + (res1 * res3) * *pc;
    pa++; pb++; pc++;
  }
  sum = sum + S[0] + S[1];
}
```

Listing 8.8 First iteration

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  int tmp1,tmp2,tmp3;
  short res0,res1,res2,res3,res4,res5;
  sum = S[0] = S[1] = 0;

  for(int is=0; is<64; is+=2)
  {
    tmp1 = (int*)pa; //SIMD load
    tmp2 = (int*)pb; //SIMD load
    res0 = EXTRACT_short_1_of_2(tmp1);
    res1 = EXTRACT_short_2_of_2(tmp1);
    res2 = EXTRACT_short_1_of_2(tmp2);
    res3 = EXTRACT_short_2_of_2(tmp2);
    tmp3 = SIMD_mul_2x16(tmp1, tmp2);
    res4 = EXTRACT_short_1_of_2(tmp3);
    res5 = EXTRACT_short_2_of_2(tmp3);

    S[0] = S[0] + res4 * *pc;
    pa++; pb++; pc++;
    S[1] = S[1] + res5 * *pc;
    pa++; pb++; pc++;
  }
  sum = sum + S[0] + S[1];
}
```

Listing 8.9 Second iteration

```
void dotproduct(short *pa, short *pb, short *pc)
{
  short sum;
  short S[2];
  sum =  S[0] = S[1] = 0;

  if( ((pa|pb|pc) & 3) == 0 )
  {
    for (int is = 0; is < 64; is += 2)
    {
     (int) S[0] = SIMD_add_2x16((int)S[0], SIMD_mul_2x16(
                  SIMD_mul_2x16((int*)pa,(int*)pb),(int*)pc));
      pa+=2; pb+=2; pc+=2;
    }
  } else {
    for(int is=0; is < 64; is += 2)
    {
     S[0] = S[0] + (*pa * *pb) * *pc;
     pa++; pb++; pc++;
     S[1] = S[1] + (*pa * *pb) * *pc;
     pa++; pb++; pc++;
    }
  }
  sum = sum + S[0] + S[1];
}
```

Listing 8.10 Final code

8.3 Retargeting the SIMD Framework

To retarget the SIMD framework, basically two pieces of information are required: first, a description of IR tree patterns that represent a SIMD candidate. This is used to generate the *SIMD-candidate matcher*. Second, the *SIMD-set construction*, the specification of how SIMD candidates can be composed to a valid SIMD-set.

8.3.1 SIMD-Candidate Matcher

The identification of SIMD candidates can be implemented using the tree-covering-based code selection [244]. SIMD candidates can be easily described by regular *mapping rules*. Normally, such a rule describes how a certain IR operation is mapped to target assembly code. *Nonterminals*, typically the rule operands, are used as "temporaries" to transfer values from one rule to another. From this specification, a *tree pattern matcher* for code selection can be generated with tools such as Burg [52]. In this approach, the regular CoSy tree-pattern-matcher generator is utilized to create a dedicated SIMD-candidate matcher from *SIMD-candidate rules*, which are part

of the regular code selector description.[1] Such rules use special *SIMD nonterminals* containing two specific attributes: a `pos` field for the subregister number within a full register and an `id` to identify a memory area, for example, allocated by a scalar variable or an array (Fig. 8.6).

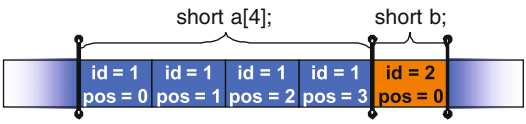


Fig. 8.6 Pos/id for array/scalar variable

As will be explained later in more detail, the former is needed to check subregister or alignment constraints and the latter becomes important when the packed result of an SIMD operation is directly consumed by another one. The initial values for these fields are already determined by the prior data-flow/alignment analysis and are initialized when a load operation is matched. Furthermore, each rule can be referenced using its unique *rule name*. Examples for two SIMD-candidate rules named `load` and `add` are shown in Listings 8.11 and 8.12.

```
\\Syntax is name:type
RULE [load] o:mirContent(src:reg_nt)
            -> dst:simd_nt;
CONDITION {
     IS_INT16(o)
}
EMIT {
     dst.pos = get_pos(o);
     dst.id  = get_id(o);
}
```

Listing 8.11 SIMD-candidate rule `load`

The 16-bit `load` rule initializes the SIMD nonterminal's `pos` and `id` fields with the values determined by data-flow/alignment analysis. The produced SIMD nonterminal may then be consumed by the `add` rule. Additional conditions can be used to select only those IR operators for a certain data type or to specify constraints on the subregister of the operands. In this example, the 16-bit `add` rule matches only if both input operands are located in the same subregister.

[1] This is not a contradiction to the limitations of tree pattern matching mentioned in Section 8.1. The matcher is only employed to identify those IR operations that might be composed to a full SIMD operation, the complete SIMD match cannot be found directly.

```
RULE  [add] o:mirPlus(src1:simd_nt,
                      src2_simd_nt)
      -> dst:simd_nt;
CONDITION {
   IS_INT16(o) && src1.pos == src2.pos
}
EMIT {
     dst.pos = src1.pos;
     dst.id  = newid(src1.id,src2.id);
}
```

Listing 8.12 SIMD-candidate rule `add`

Additionally, rules to extract a subregister from a full register must be created as well. Those are used to match the extract operations (see Section 8.2.10) inserted in previous iterations of the algorithm. In this way, they become SIMD candidates in the current iteration. All extract rules produce an SIMD nonterminal that sets `id` to the id of the temporary the result is extracted from and the `pos` field to the position of the extracted subregister, respectively (Fig. 8.7).

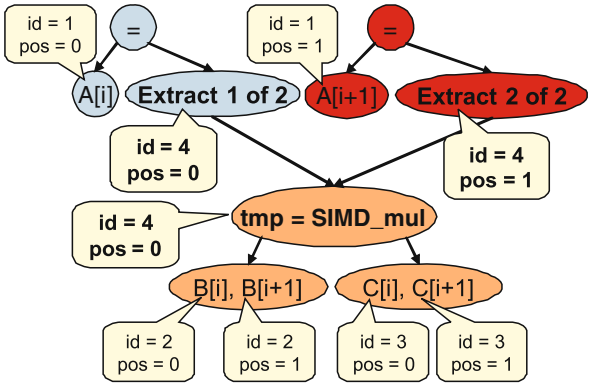


Fig. 8.7 Pos/id for extract operation

The *SIMD-candidate matcher*'s flexibility is only limited by the capabilities of the underlying tree-pattern-matcher generator. Since the concepts are already supported by the existing code selector description, only minimum changes to the retargetable compiler platform are required. Since tree-covering-based code selection is the state of the art in compiler design, this part can also be easily ported to other platforms.

8.3.2 SIMD-Set Constructor

Special *SIMD rules* describe valid tuples $N = (n_1, \ldots, n_k)$ of SIMD candidates, where k denotes the number of subregisters. In contrast to regular mapping rules, they take the names of SIMD-candidate rules instead of nonterminals as input operands, i.e., a node n_i corresponds to an SIMD-candidate rule name. The examples in Listings 8.13 and 8.14 specify a twofold 16-bit load and add SIMD instruction, using the SIMD-candidate rules from Listings 8.11 and 8.12.

```
SIMD RULE simd_load(a:load, b:load);
COMPOSITION
   CKF#1 (src:a.src) -> dst:reg_nt(a.dst, b.dst);
EMIT {
   printf("LOAD32  [%s] -> %s", REGNAME(src),REGNAME(dst));
}
```

Listing 8.13 SIMD rule twofold 16-bit load

```
SIMD RULE simd_add_2x16 (a:add, b:add);
COMPOSITION
  CKF#2 (arg1:reg_nt(a.src1, b.src1),
         arg2:reg_nt(a.src2, b.src2)
        ) -> dst:reg_nt (a.dst, b.dst);
EMIT {
  printf ("\tDUALADD16\t%s,%s -> %s",
           REGNAME(arg1), REGNAME(arg2), REGNAME(dst));
}
```

Listing 8.14 SIMD rule dual 16-bit add

Given the set of all identified SIMD candidates $C = \{c_1, c_2, \ldots\}$, the set of all possible SIMD-sets S is given by $S \subseteq \mathcal{P}(C)$ whereas each tuple in S must be in the set of all SIMD rules R as defined in the compiler configuration. Furthermore, it must match certain implicit conditions. Let $\mathrm{Pos}(c)$ denote the `pos` value of the result SIMD nonterminal produced by SIMD-candidate rule c and $\mathrm{Id}(c)$ the `id`, respectively. Then the set of valid SIMD-sets S is given by:

$$S = \{(c_1, \ldots, c_k) \mid (c_1, \ldots, c_k) \in R \wedge \mathrm{Id}(c_i) = \mathrm{Id}(c_j) \wedge \mathrm{Pos}(c_{l+1}) = \mathrm{Pos}(c_l) + 1, \\ \forall i, j \in (1, \ldots, k), l \in (1, \ldots, k-1)\} \tag{8.10}$$

In other words, the SIMD candidates of a valid SIMD-set must have the same `id` as well as an increasing `pos` value assigned.

Consider the example shown in Listing 8.15. In the first iteration, the `load` rule covers the array accesses, initializes the `id` with an unique number and the `pos` field with the position relative to SIMD load memory boundary. Note that accesses to the same array get always the same `id` assigned. Only the `pos` field varies. It is assumed that the arrays are aligned to a word boundary. Now, due to the implicit condition of the `SIMD_load`, the only way to create a complete SIMD-set is to combine two *adjacent* loads (i.e., increasing `pos`) from the same `id`. All other combinations would violate at least one constraint. Both `SIMD_loads` create a temporary with a new `id`. Afterward, the operations to extract the subregisters have been inserted as well. As mentioned above, the extracts also create new temporaries which get the same `id` as the temporary the sub-register is extracted from assigned and the `pos` field is set to the extracted subregister number, respectively.

```
for(i=0; is < 64; i += 2)
{
  //     <pos=0,id=1>   <pos=0,id=2>
  a[i]   =  b[i]      +   c[i];
  //     <pos=1,id=1>   <pos=1,id=2>
  a[i+1] =  b[i+1]    +   c[i+1];
  //     <pos=0,id=3>   <pos=0,id=4>
  x[i]   =  y[i]      +   z[i];
  //     <pos=1,id=3>   <pos=1,id=4>
  x[i+1] =  y[i+1]    +   z[i+1];
}
// In the 1st iteration:
// load -> <pos=0,id=1>, ...
// SIMD_load(<pos=0,id=1>,<pos=1,id=1>)
//           -> <pos=0,id=5>
// SIMD_load(<pos=0,id=2>,<pos=1,id=2>)
//           -> <pos=0,id=6>
// EXTRACT_short_1_of_2(<pos=0,id=5>)
//           -> <pos=0,id=5>
// EXTRACT_short_2_of_2(<pos=1,id=5>)
//           -> <pos=1,id=5>
// EXTRACT_short_1_of_2(<pos=0,id=6>)
//           -> <pos=0,id=6>
// EXTRACT_short_2_of_2(<pos=1,id=6>)
//           -> <pos=1,id=6>
// ...
```

Listing 8.15 `pos/id` in the first iteration

Thus, in the next iteration (Listing 8.16), the first and second operands of the first two additions share the same `id`s. Consequently, the same `id` is generated for both results of the additions. Now they can be combined to an `SIMD_add`. The implicit `id` condition actually enforces that the packed operands of the previous `SIMD_load` are directly reused, otherwise this might result in an expensive

repacking of the operands if, for instance, the first addition is combined with the fourth addition. Note that it is also possible to specify an explicit condition for the SIMD rules to overwrite the defaults for `pos` and `id`. As an example, the conditions on the `pos` fields can be used to model unaligned SIMD memory operations.

```
for(i=0; is < 64; i += 2)
{
  //<pos=0,id=5>
  tmp1 = (int*)(b+i);
  //<pos=0,id=5>
  res0 = EXTRACT_short_1_of_2(tmp1);
  //<pos=1,id=5>
  res1 = EXTRACT_short_2_of_2(tmp1);
  //<pos=0,id=6>
  tmp2 = (int*)(c+i);
  //<pos=0,id=6>
  res2 = EXTRACT_short_1_of_2(tmp2);
  //<pos=1,id=6>
  res3 = EXTRACT_short_2_of_2(tmp2);
  ...
  //     <pos=0,id=5>   <pos=0,id=6>
  a[i]   =  res0      +   res2;
  //     <pos=1,id=5>   <pos=1,id=6>
  a[i+1] =  res1      +   res3;
  ...
}
// In the 2nd iteration:
// add(<pos=0,id=5>,<pos=0,id=6>)
//         -> <pos=0,id=56>
// add(<pos=1,id=5>,<pos=1,id=6>)
//         -> <pos=1,id=56>
// SIMD_add(<pos=0,id=56>,<pos=1,id=56>)
// ...
```

Listing 8.16 `pos/id` in the second iteration

In order to complete the retargetable compilation flow, the CKF calls in the resulting intermediate code must be replaced by valid assembly instructions for the target processor. In this framework, the `COMPOSITION` for an SIMD rule specifies the CKF call that is internally generated for an identified SIMD-set. It consists of an unique CKF number, the argument(s) to be passed to the CKF call, and the assembly code that is finally emitted. For example, the `COMPOSITION` for `SIMD_add_2x16` describes that the arguments for the CKF call are register nonterminals that contain the first and second operands of the combined `add` rules. From this specification, a regular code selector rule matching the CKF with the given number and assembly syntax is automatically generated (Listing 8.17) and becomes part of the regular backend code selector.

```
RULE [CKF#2] o:IR_FuncCall( arg1:reg_nt,arg2:reg_nt)
                          -> dst:reg_nt;
CONDITION {
     CKF_Number(o) == CKF#2
}
EMIT {
     printf ("\tDUALADD16\t%s,%s -> %s",
              REGNAME(arg1),REGNAME(arg2),REGNAME(dst));
}
```

Listing 8.17 Internally generated CKF rule for SIMD_add_2x16

Like for the *SIMD-candidate matcher*, many concepts are already supported by the existing tree-pattern-matcher generator. Thus, only a few changes are required to the existing generator to support this approach.

As mentioned in Chapter 6, the *Compiler Designer* tool comprises techniques to generate mapping rules automatically from the LISA model. Since the SIMD configuration is quite similar to a regular code selector description, the *Compiler Designer* has been extended in order to specify and generate rules for SIMD instructions, too. More specifically, the user creates the SIMD candidate rules using the mapping dialog. In the next step, the user can select those SIMD candidates which build an SIMD-set and assign a proper assembly instruction. From this specification, an SIMD-enabled code selector description for the CoSy compiler platform is finally generated.

8.4 Experimental Results

For the evaluation, two different aspects have to be taken into account. First of all, a precise alignment analysis is a prerequisite for the SIMD optimizations to achieve good results. Therefore, this chapter first evaluates the efficiency of the alignment analysis before the benchmark results for the SIMD optimization itself are presented.

8.4.1 Alignment Analysis

The alignment is classified in one of the three classes:

Unknown: The annotation is $E_i = \emptyset$, the empty set. No information about the alignment could be gathered during the analysis.
Known: The set contains a single value. Thus, the alignment is exactly known.

Ambiguous: The set contains several values. With regard to the annotation precision, this is equivalent to a known value. It means that the alignment will actually change during the runtime of the program.

The metrics used to measure the accuracy is the ratio of annotated to total nodes:

$$r = \frac{\text{number of known nodes} + \text{number of ambiguous nodes}}{\text{number of total nodes}} \tag{8.11}$$

The nominator expression is the sum of both, the exactly known pointers and the ambiguous pointers. This is reasonable since an expression that contains several entries in its set can definitely take on several modulo values, depending on the program's input data. The applications chosen to benchmark the results are taken from the domain of typical DSP and embedded algorithms. They present different degrees of complexity to the compiler, which are as follows.

ADPCM: This is a floating-point implementation of an adaptive differential pulse-code modulation encoder. It is a self-contained program with a `main()` procedure calling a few worker procedures. Data accesses are performed through pointers that are initialized to the addresses of global objects and then manipulated by address arithmetics throughout the program. All the functions were contained in a single compilation unit.

FFT: The FFT works on a 16-bit fixed-point representation but is otherwise similar to the ADPCM described above. Several functions are combined in a single compilation unit. In contrast to the ADPCM, however, the data are passed by means of pointer arguments to function calls.

libmad: This is an open source 32-bit fixed-point implementation [265] of the MPEG-1 audio Layer 1–3 standards [185]. The primary goal of the project is to provide a high-performance mp3 library written in a portable C style. It consists of several modules that are compiled separately and exchange data by means of pointer arguments.

gsm: The implementation used is freely available on the Internet [129]. It is a floating-point implementation of the standard and similar in structure to `libmad`.

AAC: This is the AAC audio codec's reference implementation of the 3GPP consortium. It is written in ANSI-C, spread across a large number of modules, and makes heavy use of complex language elements such as arrays of pointers or nested `structs`.

H.264: This is another complex library in the same style as the AAC decoder.

The benchmarks above have been chosen to measure the *annotation rate*. As shown in the next section, typical SIMD benchmarks for embedded processors supports only a very basic set of SIMD operations, which must

- be completely regular;
- work on short data types of 8- or 16-bit size;
- work on fixed-point data types.

The test cases here do not comply with these requirements. They operate on floating-point or 32-bit fixed-point representations. Creating fixed-point versions of complex algorithms, however, requires a high engineering effort. For that reason, such versions are usually not publicly available. Nevertheless, the set of test cases chosen does contain a typical set of pointer accesses to floating-point data types and can therefore be used to evaluate how efficiently the analysis can propagate values around the program. The detailed results are given in Table 8.2. In addition to the name and the rate, the number of compilation units (CUs) the program consists of,

Table 8.2 Annotation rate

Name	CUs	Lines	Rate%	Total	Known	Ambiguous	Unknown
adpcm	1	493	100	39	39	0	0
FFT	1	457	93	31	27	2	2
libmad	12	11791	58	3362	1738	211	1413
GSM	14	4014	55	1620	869	28	723
AAC	38	6767	20	5100	811	236	4053
H.264	30	31099	19	13188	2428	90	10670

the total number of lines in the source code, the total number of pointers in the program, and the numbers for known, ambiguous, and unknown annotations are given. It is obvious that the programs tested can be divided into three classes with respect to their predisposition for alignment analysis. The straightforward implementations of the FFT and the ADPCM coder give very good results. These are complete programs, which are available in a single compilation unit, with a single entry point, the `main()` function. The code is written using direct pointers to the data involved. Those pointers are then modified by address arithmetics during the program's execution.

The *GSM* implementation and *libmad* are similar in coding style to the previous class. They make moderate use of structs and usually pass pointers to the memory operated upon. The main difference to the first class is that they are formed by several compilation units. For modules that are largely self-contained and that have a well-defined interface to the outside world, the annotation rate is usually better than for the modules that handle file access. The core-encoder routine for the GSM codec achieved an annotation rate of 70 and 82% of the pointers in the Layer III decoding module of libmad could successfully be annotated. This is due to the fact that the developers of these libraries made liberal use of the `static` storage classifier for functions that enabled the creation of a call graph with less edges. However, a noticeable uncertainty with regards to the interprocedural flow remains, which clearly shows in the average annotation rate of about 55% in these cases.

The programs in the third class, which is hardly analyzable, are reference implementations of recent audio and video codecs. They have been written for readability by humans and correct, yet not necessarily fast execution. This leads to skimpy use

of the `static` classifier, nested structures to emulate class hierarchies, and multi-dimensional arrays of structures. An excerpt from the core-decoding module of the aac decoder is shown in Listing 8.18. This coding style makes it very

```
AACDECODER CAacDecOpen(...)
{
  struct AAC_DECODER_INSTANCE *self;
  ...
  AacDecInstance.pAacDecStaticChannelInfo[ch]->pLongWindow[0] =
  OnlyLongWindowSine;
  self->pAacDecChannelInfo[ch]->pCodeBook =
  pAacDecDynamicDataInit[ch]->aCodeBook;
        ...
}
```

Listing 8.18 Source excerpt from the core aac-decoding module

difficult to do the data-flow analysis, upon which the alignment analysis is built. In order to successfully annotate programs like these, not only the values assigned to objects, but also values in memory have to be tracked.

8.4.2 SIMD Optimizations

For experimental evaluation, SIMD-enabled C compilers have been created for the NXP TriMedia processor [190] and the ARM11 [41]. The TriMedia compiler has been designed using the *Compiler Designer* tool whereas the ARM11 compiler is a hand-crafted CoSy compiler. In contrast to, e.g., the AltiVec or SSE extension, both architectures support SIMD only for short (i.e., 8-bit and 16-bit) integer data types – which is quite common for embedded processors. Hence, benchmarks employing floating-point computations cannot be used. Therefore, mostly benchmarks from the DSPStone benchmark suite [269] have been selected and several additional kernels have been implemented, similar to those used in [72, 86, 117]. Furthermore, additional results for the following more complex DSP algorithms are provided:

quantize matrix quantization with rounding
compress discrete cosine transformation to compress a 128×128 pixel image by a factor of 4:1, block size of 8×8
idct 8×8 IEEE-1180 compliant inverse discrete cosine transformation
viterbi GSM full-rate convolutional decoder
emboss Converts an image using an emboss filter
sobel Applies a sobel filter to an image
corr_gen Generalized correlation with a one-by-M tap filter

For the given TriMedia and ARM LISA ADL models, the required retargeting effort for SIMD support is quite limited. The corresponding CGD descriptions for SIMD consist of 393 (TriMedia) and 698 (ARM) lines of code, which accounts for roughly 7% (TriMedia) and 14% (ARM) of the complete CGD description. A similar workload can be expected for other processors, depending on architecture features.

Regarding the SIMD architecture, the TriMedia is a five-slot VLIW DSP with 128 general-purpose registers and a number of SIMD instructions. Due to its VLIW architecture, using SIMD instructions does not lead to a speedup in all cases. For instance, one can issue five parallel ADD instructions simultaneously, while only two dual-ADD SIMD instructions can be issued at a time. Furthermore, SIMD instructions may have a higher latency than regular instructions (e.g., one cycle for an ADD vs. two cycles for a dual-ADD). So, unless the instruction scheduler is not able to find suitable instructions for filling the VLIW slots saved by SIMD, no speedup can be expected. However, if the memory is the bottleneck (at most two parallel LOADs/STOREs), SIMD instructions still help to reduce the memory pressure. There are also further effects, due to the C-coding style or register allocation effects in the compiler backend, that leads to deviations from the theoretical speedup factor k in case of k subregisters. The memory is organized in 32-bit words, hence word alignment is required for SIMD memory accesses.

In contrast, the ARM architecture is built around a central, scalar RISC core. It has a register file that consists of 31 general-purpose registers (at any one time only 16 register are visible) and six status registers. The memory is also organized in 32 bits words. It requires the same word alignment for all memory accesses as the TriMedia. The ARM11's instruction-set supports only a limited set of SIMD instructions, which consists of additions and subtractions of byte or half-word data values in 32-bit registers. Furthermore, the ARM features a complex dot-product support operation, which multiplies two pairs of half-words in parallel, and adds the two resulting word-wide values to an accumulator. Since there is no direct SIMD multiplication operation available, kernels that do not match this dot-product support operation cannot be optimized.

Loop unrolling alone already has a large impact on the overall performance. Hence, the speedup is measured by using the following equation:

$$Speedup = \frac{cycles_{Unroll}}{cycles_{Vector+SIMDfyer}} \tag{8.12}$$

$Cycles_{Unroll}$ denotes the number of cycles the test kernel needed when compiled with unrolling turned on, but the SIMD engines (i.e., *Vectorizer* and *SIMDfyer*) turned off. $Cycles_{Vectorizer+SIMDfyer}$ denotes the number of cycles the kernel needed when compiled with the same unrolling factor and the SIMD engines activated. Hence, the speedup is only due to the SIMD instructions. All other compiler parameters have always been identical.

The results are quantified first for one simple, particular benchmark, that is, a *dot product*, where vector elements are accessed by means of array accesses in the C code:

```
for(i = 0; i < N; i++)
  sum += a[i] * b[i];
```

Listing 8.19 Dot product

Due to the dependency on `sum`, a scalar expansion has to be applied to the loop before SIMD instructions can be inserted. First of all, the impact of the alignment analysis and the overhead introduced by scalar expansion is investigated. Figure 8.8 shows the speedup over the number of loop iterations I with and without alignment analysis using a fixed unroll factor of 4. It can be clearly seen that a certain iteration count is required to compensate the overhead by scalar expansion until SIMD pays offs. Beyond that, the speedup is largely independent of I. For high iteration counts, the speedup is asymptotically 2, which corresponds to the theoretical speedup in this case. Obviously, the version without the dynamic alignment check reaches the break-even point considerably faster than the one with the checks. The reason for the extremely high speedup obtained on the ARM processor is due to type conversions. Since the multiplications in the non-SIMD version produce results of 32 bits size, these have to be converted to 16-bit precision afterward. The ARM compiler, however, generates a sequence of a logical left shift by 16 bits, followed by an arithmetic right shift back to achieve this. In the SIMD version, though, these steps are not necessary since the results of the operations are already 16-bit values.

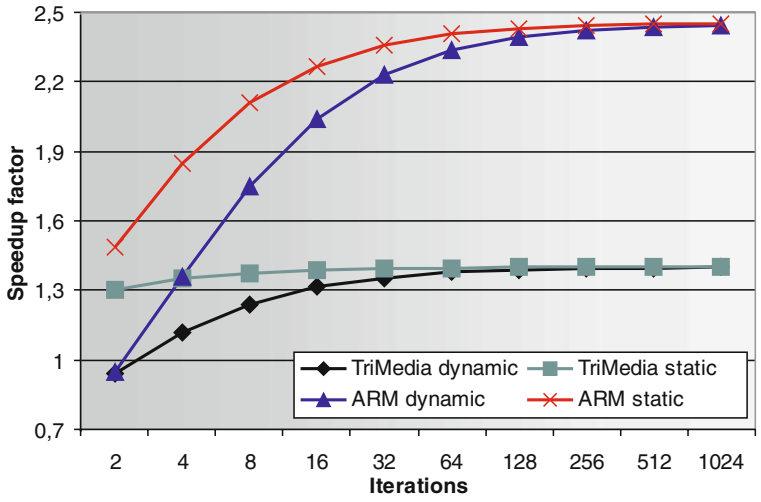


Fig. 8.8 Speedup factor over loop iterations for dot product

The former two cases have demonstrated the dependence of the speedup on the iteration count. Another interesting figure is the development with dependence on rising unroll factors (after SIMD optimization). The example given in Fig. 8.9 shows the progression for the dot product. The number of iterations for this graph has been chosen to $N = 128$. As apparent from Fig. 8.8, this is a number where the speedup is already very close to its peak value.

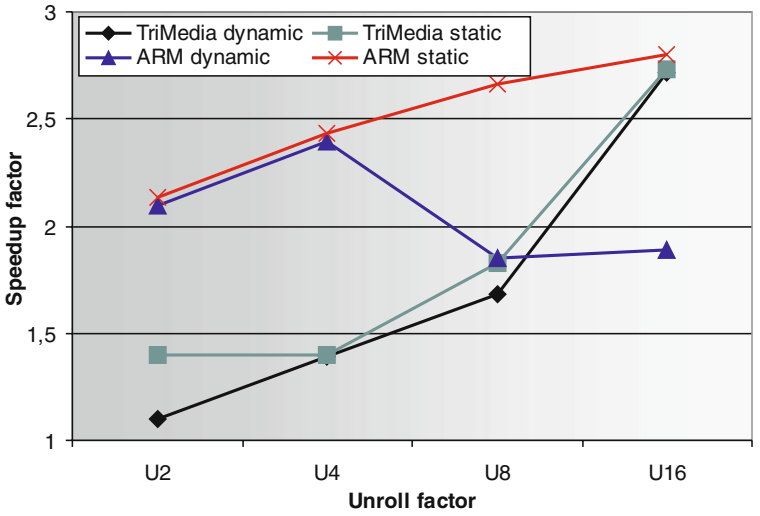


Fig. 8.9 Speedup factor over unroll factor for dot product

In the values for the TriMedia, little difference is seen between the versions with or without dynamic checks. The strong rise in speedup for the high unroll factors is due to the additional resource pressure created by the large loop body. Since the VLIW architecture is inherently parallel, this pressure is needed to completely saturate the CPU. The ARM's progression, however, shows an unexpected decline in performance for higher unroll factors. After close examination, the cause has been determined to be register shortage resulting in a considerable amount of spill code. Obviously, the ARM greatly benefits from the removal of the dynamic check, since registers are freed and thereby more degrees of freedom are left to the register allocator. The TriMedia processor with its 128 available registers is not affected by this problem.

Loop unrolling is known to have a large impact on the code size. Hence, larger speedups come at the expense of an increased code size. Figure 8.10 illustrates the code-size increase for the dot-product kernel ($I = 128$) due to unrolling for both the SIMD and non-SIMD version. The not unrolled, non-SIMD version is used as baseline. Due to the RISC architecture of the ARM, the code-size increase caused by unrolling alone is more significant than for the TriMedia. However, the SIMD version for the ARM can compensate the code-size effect of unrolling to a great extent. First, SIMD directly reduces the number of instructions inside the loop.

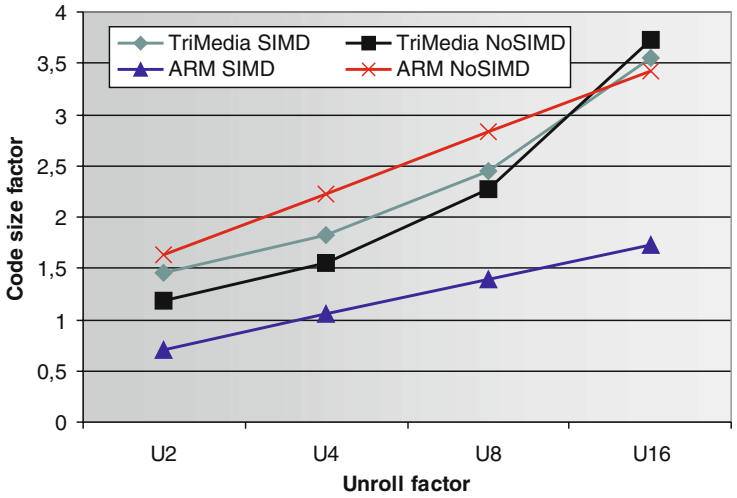


Fig. 8.10 Code size over unroll factor for dot product

Second, the special dot-product-style SIMD instruction almost eliminates the overhead by scalar expansion. This kind of instruction is not available in the TriMedia. Additionally, SIMD reduces the number of instructions for the TriMedia as well but not necessarily the number of VLIW words. Hence, the SIMD version shows a larger code-size factor than the non-SIMD version. For high unroll factor, the parallel functional units of the TriMedia become saturated, which leads to a stronger rise of the code size. However, for modest unroll factors (2 or 4), the increase in code size is acceptable for both architectures.

Finally, Fig. 8.11 summarizes the speedup results for all benchmarks. The number of loop iterations I for the DSPStone kernels is fixed ($I = 128$) and for the more complex DSP routines as specified. For each benchmark, the unroll factor is 4.

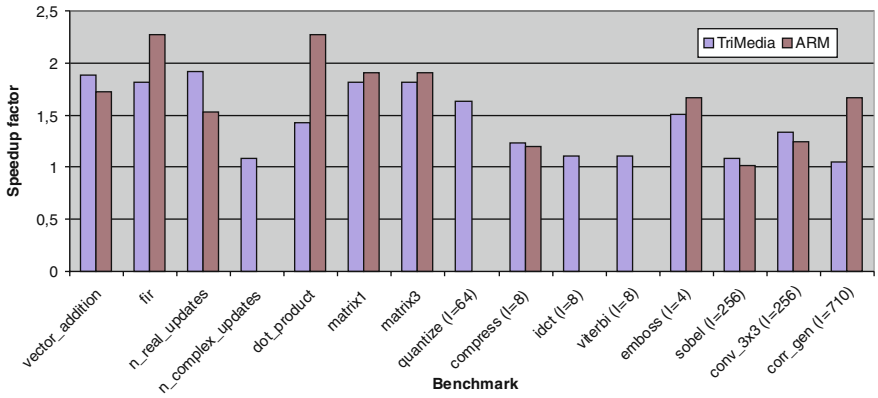


Fig. 8.11 Benchmark results

In the presence of dynamic-alignment checks, the SIMD loop version including the alignment check overhead has been measured. A significant speedup was obtained in most cases. The speedup for the complex DSP routines is generally lower, since a smaller fraction of the benchmark code can be mapped to SIMD instructions than in the case of the DSPStone kernels. Still, a speedup of 7% up to 66% was observed. In certain cases, a super-linear speedup for the ARM can be achieved (e.g., 2.2 for *fir*). This is related to the special multiply instructions of the ARM that helps to reduce the overhead introduced by scalar expansion. On the other hand, for three benchmarks, no speedup could be obtained for the ARM due to the lack of a multiplication without accumulation.

Regarding the code size, for the DSPStone kernels, an average code-size factor of 0.9 for the ARM and 1.1 for the TriMedia can be observed, as compared to benchmarks with unrolling enabled but without use of the SIMD optimizations. The code size of the complex kernels essentially remains the same for both architectures since only a small portion of the code is replaced by SIMD instructions.

8.5 Conclusions

Almost all previous approaches to SIMD optimization are tailored to a specific target architecture. This book presents a *retargetable* optimization framework for the class of processors with SIMD support. The underlying concepts are proven by integrating the SIMD framework into the CoSy platform that can be retargeted via the *Compiler Designer* GUI. In this way, SIMD-enabled compiler for two realistic embedded processors were generated. The required retargeting effort is quite limited for both compilers.

This results in a seamless and retargetable path from a single LISA model to a SIMD-enabled C compiler. While previous backend-oriented SIMD optimization techniques potentially led to higher code quality, significant speedup results for standard benchmarks were generally obtained with this framework. Hence, the presented approach provides a good and practical compromise between *code efficiency* and *compiler flexibility*.

The current implementation shows several limitations, whose elimination would probably lead to higher code quality and would allow to handle a wider range of loop constructs. As pointed out in [7, 72, 212], SIMD optimization is often hindered by limitations of the SIMD memory unit in combination with the memory access patterns in current applications. It is often necessary to reorder the subregisters, using special *permute* instructions before SIMD instructions can be applied at all. So far, these instructions are rarely supported by embedded processors. However, with the advances in semiconductor technology, the SIMD data path width will increase in the future, and thus it becomes more likely that next generation embedded processors will support those. Therefore *support for permutation* seems to be a promising extension for the future.

Chapter 9
Predicated Execution

This chapter focuses on another class of target processors, namely, those equipped with deep pipelines and parallel functional units such as VLIW architectures for instance. Such architectures are quite popular in embedded system design since they do not require designs to sacrifice software development productivity for the very high-performance processing needed for today's applications. Naturally, to achieve their peak performance, all parallel functional units must be kept busy during program execution. Thus, a common hardware features to increase the amount of available *instruction-level parallelism* (ILP) is *predicated execution* (PE). Basically, this allows to implement *if-then-else* (ITE) statements without jump instructions that offers a number of optimization opportunities. Furthermore, PE can enable more aggressive compiler optimizations that are often limited by control dependencies. For example, software pipelining, which is crucial to achieve high performance for ILP processors, can be substantially improved by PE [189]. However, this feature is by far not limited to highly parallel and deeply pipelined processors. Even though less beneficial, single-issue embedded processors such as the ARM9 [41] or configurable cores [44] are equipped with this feature, too. Clearly, support for PE in retargetable compilers is of strong interest.

This chapter starts with looking at the issue for exploiting PE in ITE statements, before related work is discussed in Section 9.2. Section 9.3 presents the optimization concepts. Afterward, Section 9.4 introduces the retargeting formalism and the code generation flow. Section 9.6 provides experimental results for several embedded processors. Finally, this chapter is summarized and some future work is discussed in Section 9.7.

9.1 Code Example

Predicated execution refers to the conditional execution of instructions based on the value of a boolean source operand p. Irrespective of p's value, the instruction allocates the same processor resources. In case p is *false*, the computed result is ignored, i.e., it effectively behaves like a no-operation (NOP) instruction. Compilers utilize this to implement ITE statements without jump instructions. As pointed out in [125],

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_9, © Springer Science+Business Media, LLC 2010

this can also be seen as converting control dependencies into data dependencies, also referred to as *if-conversion*.

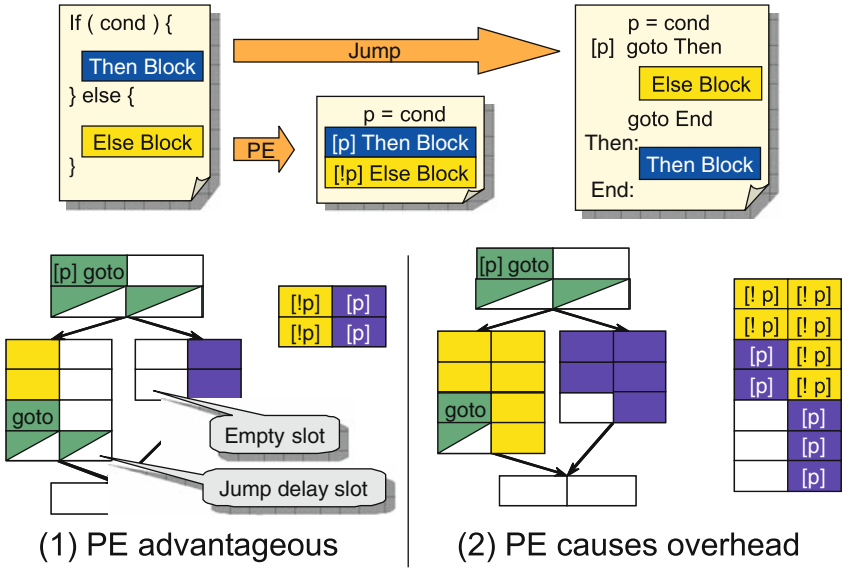


Fig. 9.1 Implementation of an if-then-else statement with jump and conditional instructions

Consider the example in Fig. 9.1. The implementation on the right shows the common implementation of an ITE statement. It uses conditional jumps to model the control flow resulting from the C-code example on the left. The implementation with conditional instructions predicates the `then` block with the result of the if-statement's condition p and the `else` block with the negation thereof.

Since jump instructions typically cause control hazards (cf. Section 3.3.4), the delay slots of the jump instructions have to be filled with NOPs or with other useful instructions (in case there are any). PE in contrast eliminates the control-flow instructions, which results in a single, but larger basic block containing the still mutually exclusive `then` and `else` blocks. Larger basic blocks result in more opportunities to exploit ILP. In the ideal case, both blocks can be completely parallelized on an ILP processor. Case (1) exemplifies this for a two-issue slot processor. There are not enough instructions to fill the delay slots in the jump implementation whereas the PE implementation not only eliminates the delay slots, but also completely parallelizes the `then` and `else` blocks.

Unfortunately, if-conversion does not always pay off. It may also happen that, due to resource conflicts during scheduling, the final schedule for the PE implementation has a larger length than the implementation with jump instructions. Case (2) illustrates the following. Here, there are few free slots left in the `then` and `else block`, and hence there is almost no chance to parallelize them. Consequently, the actual performance of both implementations always depends on the concrete input

program. Therefore, a precise cost computation is crucial to avoid a performance loss with PE.

9.2 Related Work

Many compilation techniques for PE are based on the work by Mahlke et al. [243]. It describes the formation of so-called hyperblocks, an extended basic block concurrently executing multiple threads of conditional code. The decision whether to include a basic block in a hyperblock is based on the criteria of execution frequency, block size, and instruction characteristics. Since it does neither take the degree of ILP into account nor the dependencies between different blocks, scheduling for machines with a few issue slots increased the resource interference, and thus resulted in performance degradation. August et al. [60] improved this work by allowing the scheduler to revise decisions on hyperblock formation. But this leads to a complicated scheduler implementation. Additionally, it extends the previous work by partial if-conversion: in many cases, including only a part of a path may be more beneficial than including or excluding the entire path. Smelyanskiy et al. [175] tried to solve the resource interference of Mahlke's approach by a technique called predicate-aware scheduling. However, they state that an architecture that supports their optimization proposal does not exist yet. All hyperblock-based approaches optimize the average execution time.

The approach by Leupers [220] focuses especially on embedded processors and optimizes the worst-case execution time. In contrast to the previous work, it is capable of handling complete (possibly nested) ITE statements with multiple basic blocks at a time. It has been selected as a starting point to develop a retargetable PE optimization.

Hazelwood et al. [146] incorporated a lightweight if-conversion into a dynamic optimization system. However, the overhead of such systems makes Hazelwood's work less suitable for embedded processors. Chuang et al. [272] target primarily out-of-order architectures, which are rarely used in the embedded domain. By combining control-flow paths, PE introduces false dependencies between instructions of disjoint paths. In [153], these dependencies are resolved by means of predicated static single assignment (SSA). The downside is a significantly increased code size and the high amount of required predicate registers – both are severe issues in the embedded domain.

From the ASIP design platforms mentioned in Chapter 4 only Trimaran supports PE, but this platform is limited to a narrow range of architectures. Quite recently Target Compiler Technologies announced support for PE, but nothing in this regard has been published yet. In the domain of "general-purpose" retargetable compilers, the *gcc* [87] supports if-conversion, but *gcc* is generally known as being difficult to adapt efficiently to embedded processor designs.

The aforementioned PE optimization techniques are mostly adapted for a certain target machine. Hence, porting one of them to a new processor architecture is a tedious manual process. Therefore, the implementation in this book focuses on

an *effective deployment of PE* while achieving *retargetability* for a wide variety of processors with PE support [167].

9.3 Optimization Algorithm

As already mentioned above, ITE statements can be implemented using conditional jumps or conditional instructions. Another possibility is to implement only either the `then` or `else` block with conditional instructions, which is referred to as *partial if-conversion*. Furthermore, the concrete implementation depends also on the nesting level of the ITE statement. The following section introduces all possible ITE implementations, henceforth referred to as *schemes*. Section 9.3.3 concentrates on the cost computation of each scheme. Finally, Section 9.3.4 describes how the best implementation is selected.

9.3.1 Implementation Schemes

In the following, the infix INS denotes the implementation with conditional instructions and JMP the implementation with conditional jumps. Furthermore, the prefix ITE stands for if-then-else statements and IT for if-then statements. A suffix P indicates a scheme with precondition. The notation `[p]` means that the following instruction or even a complete basic block is executed under the condition stored in `p`. The schemes used in the example in Fig. 9.1 are depicted in Listings 9.1 and 9.2.

```
    p = R   //store if-condition R
[p] goto L1 //cond. jump to Then
    B_E     //else block
    goto L2 //jump to end
L1: B_T     //then block
L2:
```

Listing 9.1 Scheme 1: ITEJMP

```
    p = R  //store if-condition R
    q = !p //negate condition
[p] B_T    //cond. execute Then
[q] B_E    //cond. execute Else
```

Listing 9.2 Scheme 2: ITEINS

In case of a nested ITE statement, the execution of the `then` or `else` block of the nested statement depends on `p` (the condition of the outer ITE statement) and

on R′, which is the condition of the nested statement itself. Hence, p constitutes the *precondition* for the nested ITE statement. The corresponding schemes are shown in Listings 9.3 and 9.4. Note that it is usually not possible to attach multiple conditions to a single instruction. It is important that the precondition survives the nested schemes, because subsequent instructions may also depend on it. Similar schemes are obtained for IT statements (Listings 9.5, 9.6, and 9.7).

```
[p] c = R'//cond. store nested if-cond
    q = !p  //negate precondition
[q] c = 0
[c] goto L1 //cond. jump to Then
[p] X_E     //cond. exec. nested Else
    goto L2 //jump to end
L1: X_T     //execute nested Then
L2:
```

Listing 9.3 Scheme 3: ITEJMPP

```
[p] c = R'//cond. store nested if-cond
    d = !c //negate nested if-cond.
    q = !p //negate precondition
[q] c = 0
[q] d = 0
[c] X_T    //cond. exec. nested Then
[d] X_E    //cond. exec. nested Else
```

Listing 9.4 Scheme 4: ITEINSP

```
    p = !R
[p] goto L1
    B_T
L1:
```

Listing 9.5 Scheme 5: ITJMP

```
    p = R
[p] B_T
```

Listing 9.6 Scheme 6: ITINS

```
[p] c = !R'
    q = !p
[q] c = 1
[c] goto L1
    X_T
L1:
```

Listing 9.7 Scheme 7: ITEJMPP

```
[p] c = R'
    q = !p
[q] c = 0
[c] X_T
```

Listing 9.8 Scheme 8: ITEINSP

```
    p = R
[p] B_T
[p] goto L1
    B_E
L1:
```

Listing 9.9 Scheme 9: ITETHEN

```
    p = R
    q = !p
[q] B_E
[q] goto L1
    B_T
L1:
```

Listing 9.10 Scheme 10: ITEELSE

```
[p] c = R'
    q = !p
[q] c = 0
[c] X_T
[c] q = 1
[q] goto L1
    X_E
L1:
```

Listing 9.11 Scheme 11: ITETHENP

```
[p] c = R'
    d = !c
    q = !p
[q] d = 0
[c] X_E
[d] q = 1
[q] goto L1
    X_T
L1:
```

Listing 9.12 Scheme 12: ITEELSEP

Of course, the presented schemes with the prefix INS can only handle ITE statements whose `then` and `else` blocks can be conditionally executed at all. Hampering elements might be instructions that are not conditionally executable or the `then` and `else` blocks may have more than one incoming control-flow edge. By introducing new implementation schemes such ITE statements can be handled as well. The idea is to convert ITE statements partially by executing only one block conditionally. This leads to the implementation schemes shown in Listings 9.9, 9.10, 9.11, and 9.12.

For instance, if the `else` block prevents if-conversion due to any of the above mentioned reasons, then scheme ITETHEN can be applied. According to this scheme, the condition is computed in `p`. Therewith the execution of the `then` block is predicated. If `p` is *true* the `else` block must not be executed, and consequently the conditional jump to the `end` block is taken. Considering nested IT statements, additional code is needed to set the condition of the ITE statement at hand to *false* in case the precondition is not fulfilled.

Note that for any of the above described schemes, it is assumed that the control flow from the `if` block either falls through to the `else` block or conditionally jumps to the `then` block. Though this usually depends on the concrete application and the involved compiler optimizations. The block order might also be the other way round or sometimes the `then` and `else` blocks do not even follow the `if` block directly, i.e., there is an explicit branch instruction to each block. Some of these cases require slightly different schemes but they have been omitted here for the sake of brevity. Furthermore, the implementation depends also on the support for negated conditions. Some processors directly support negated predicates, others

need to compute them explicitly. In the schemes shown here, it is assumed that negated predicates are not supported.

For each of these schemes, the costs C, measured in instruction cycles, is computed. In the default case, this time is calculated as $C = \max(C_T, C_E)$, where C_T and C_E denote the execution time of the ITE statement in case the `then` or `else` block gets executed, respectively. This corresponds to the *worst-case execution time* of an ITE statement, which is a typical measure in the context of embedded systems due to the real-time constraints. However, in certain cases, it makes sense to consider the *average* execution time of an ITE statement. As will be explained in the following, they can be incorporated by using *transition probabilities*.

9.3.2 Probability Information

The examination of several control-intensive programs revealed that many ITE statements handle errors in internal data structures or to cope with wrong program inputs. Generally, during normal program execution, these cases are unlikely to happen. However, at the same time, such cases often prevented if-conversion since the corresponding blocks dominated the worst-case execution time.

Another problem has been observed in the case of uneven long ITE blocks. As exemplified in Fig. 9.2, suppose the `else` block is much shorter than the `then` block. Most likely, the instructions of the `else` block will fit into free instructions slots of the `then` block that consequently improves the worst-case execution time. But if the execution frequency of the `else` block is higher than that of the `then` block, then applying if-conversion (ITEINS) results in a performance degradation in more than 50% of all cases. Thus, converting the if-statement partially by executing only the `else` block conditionally (ITEELSE) might be the better choice.

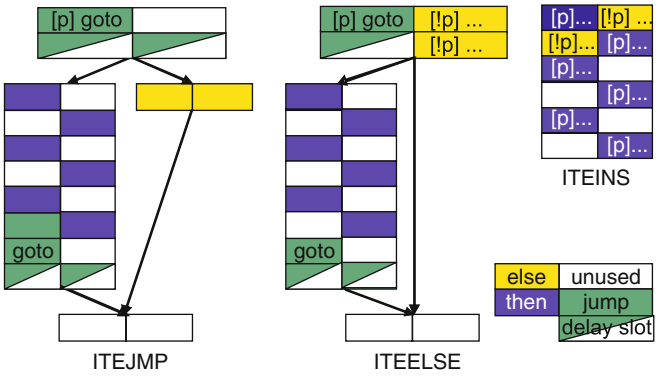
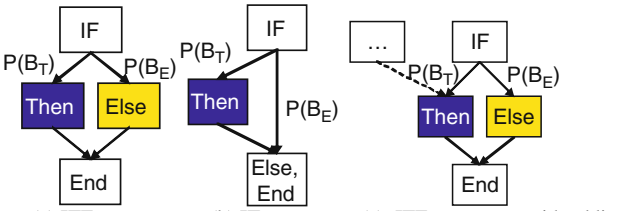


Fig. 9.2 Uneven long `then` and `else` blocks

Therefore, it seems reasonable to provide the programmer an opportunity to influence the cost computation for each ITE statement. A solution to this problem is to provide information for the execution probability of the `then` and `else` blocks.

This can be utilized in the cost computation later on. The value $P(B_x)$ denotes the probability for the transition from the `if` block (the block containing the condition) to the `then` block B_T or `else` block B_E respectively. Moreover, the sum of the probabilities gives one per definition: $P(B_\mathrm{T}) + P(B_\mathrm{E}) = 1$.

CoSy annotates each basic block with a so-called *use estimate*, the estimated execution frequency. These values are computed by a separate engine. Their main purpose is to improve the spill heuristic of the register allocator, but it is evaluated in other optimizations as well. Here, in this context, these values can be used to derive the transition probabilities.



(a) ITE statement (b) IT statement (c) ITE statement with additional incoming control flow edges

Fig. 9.3 Different constellations of if-statements

Three constellations of if-statements as shown in Fig. 9.3 must be considered. The graphs on the left and middle are well structured, but the right one is not due to the additional control-flow edge. In the following, E_x denotes the use estimate of either the `if`, the `then`, or the `else` block. For the case in Fig. 9.3(a) when the `if` block is executed, the control flow reaches either the `then` or `else` block. Moreover, the `if` block dominates these blocks immediately, there exists no other path that can be taken to reach one of these blocks (i.e., the `if` block is always executed immediately before). Thus, the use estimates can be calculated as

$$E_\mathrm{if} = E_\mathrm{then} + E_\mathrm{else} \tag{9.1}$$

and the transition probabilities as

$$P(B_\mathrm{T}) = \frac{E_\mathrm{then}}{E_\mathrm{if}} \quad \text{and} \quad P(B_\mathrm{E}) = \frac{E_\mathrm{else}}{E_\mathrm{if}} \tag{9.2}$$

The cases in Fig. 9.3(b, c) are a little bit different. Unfortunately, there is no immediate dominance relation like in the previous case. Considering Fig. 9.3(b), the `if` block only dominates the `then` block immediately. However, the `else` block is identical to the `end` block, which obviously is not immediately dominated by the `if` block. Thus, the use estimates are given by

$$E_\mathrm{if} \neq E_\mathrm{then} + E_\mathrm{else} = E_\mathrm{then} + E_\mathrm{end} \tag{9.3}$$

The formula to calculate $P(B_T)$ still holds and since the sum of the transition probabilities must be 1, this results in

$$P(B_\mathrm{T}) = \frac{E_\mathrm{then}}{E_\mathrm{if}} \quad \text{and} \quad P(B_\mathrm{E}) = 1 - P(B_\mathrm{T}) \tag{9.4}$$

The last case is similarly. Since the `then` block is not dominated by the `if` block, the equation

$$E_\mathrm{if} \neq E_\mathrm{then} + E_\mathrm{else} \tag{9.5}$$

still holds, and consequently

$$P(B_\mathrm{E}) = \frac{E_\mathrm{else}}{E_\mathrm{if}} \quad \text{and} \quad P(B_\mathrm{T}) = 1 - P(B_\mathrm{E}) \tag{9.6}$$

Of course, this is a simple but not very precise way to determine probability information. More accuracy can be obtained by using profiling information. Obviously, this can yield very accurate values, but on the other hand this method may increase the compile time significantly.

CoSy ships with a path-profiling engine called `pprofile`. This engine is inserted in the compiler before the cost computation, since the results shall be used there. This engine operates in several modes. At first it simply instruments the program. Thereafter the compiled program must be executed in a simulator. To obtain an accurate profile, the simulation should be repeated with different input data. During the simulation, the instrumentation results are written to a file. When recompiling the program, `pprofile` reads the file and stores the results in the use estimate field of the basic blocks. The subsequent cost computation automatically uses these values to compute the transition probabilities.

Another option is to directly annotate the probabilities or a relative quantifier to the ITE statements itself using `pragmas`. In the latter case, the programmer can annotate each branch of a particular if-statement that is more likely to be taken. For this purpose, the pragmas `__ceLikely` and `__ceUnlikely` were introduced. They can be inserted at the beginning of the `then` or `else` block of an if-statement as shown in Listing 9.13.

```
if (value > 255) {
  #pragma __ceUnlikely
  value = 255;
}
```

Listing 9.13 Relative quantifier

```
if (overflow) {
  #pragma __ceProb(0.2)
  value = 255;
} else {
  value++;
}
```

Listing 9.14 Transition probabilities

Internally the cost computation weighs the use estimate of the `then` block with the factor 0.5 (`__ceUnlikely`) and in case of `__ceLikely` with the factor 2.

A precise specification of the transition probability is possible with the pragma `__ceProb(`*val*`)`. In this way, the transition probability can be directly passed to the cost computation. Listing 9.14 gives an example. As can be seen, the probability to transition to the `else` block is omitted here. It is sufficient to specify only one probability value, the other is computed using the equation $P(B_T) + P(B_E) = 1$. If both values are specified for an if-then-else statement and their sum is unequal, each value will be divided by the sum, so that the equation $P(B_T) + P(B_E) = 1$ still holds.

9.3.3 Cost Computation

The implementation schemes, naturally, implicate different execution times. The cost computation annotates to each ITE or IT statement a *cost table*. It stores for all schemes the corresponding execution times. The computation assumes that a conditional instruction consumes the same resources regardless whether its condition is *true* or *false* and that both cases have the same execution times. In the following, the superscript ^{P} denotes the presence of a precondition. The branch instructions and the corresponding delay slots are distinguished as J_{taken}, a conditional branch that is taken; J_{nottaken}, a conditional branch that is not taken; and J_{always} an unconditional branch. Considering nested ITE statements, the calculation starts with the innermost and continues with the surrounding ITE statement.

The costs can be separated into two components: setup costs and cost values for the `then` and `else` blocks. The former emerge from extra instructions required for negating if-conditions or to compute possible preconditions. Obviously, the setup cost depend on the given target architecture. For example, some architectures support negated predicates, others need an extra instruction.

The costs for computing the ITE condition itself are not taken into account since they incur for all schemes. Table 9.1 summarizes the setup cost for each scheme, assuming that the architecture does not support negated conditions. For example, ITEJMP has no setup costs whereas ITEINS has a cost of one due to the additional instruction needed to negate the if-condition (see Listing 9.2).

The second component of the cost computation consists of the cost values for the `then` and `else` blocks. A block is a sequence of statements $(s_1, \ldots, s_n)$. The costs of a statement s_i are denoted as $C(s_i)$ or $C^P(s_i)$, depending on whether s_i is executed under a precondition or not.

If s_i is a simple statement, the costs are $C(s_i) = C^P(s_i) = 1$, but if s_i is an ITE statement, the costs depend on the concrete implementation scheme. $C(B_T)$, $C(B_E)$, and $C^P(B_T)$, $C^P(B_E)$ denote the execution times of the `then` and `else` blocks without and with precondition, respectively. In case a scheme merges both blocks, the execution time for the joint execution is denoted as $C(B_T \circ B_E)$. In prior work, this value is modeled by a static formula that takes the execution times of the individual

Table 9.1 Setup costs according to the different implementation schemes

Scheme	Setup costs
ITEJMP	$S_1 = 0$
ITEINS	$S_2 = 1$
ITEJMPP	$S_3 = 2$
ITEINSP	$S_4 = 4$
ITJMP	$S_5 = 0$
ITINS	$S_6 = 0$
ITJMPP	$S_7 = 2$
ITINSP	$S_8 = 2$
ITETHEN	$S_9 = 0$
ITEELSE	$S_{10} = 1$
ITETHENP	$S_{11} = 3$
ITEELSEP	$S_{12} = 4$

blocks, the ILP degree, and possible resource conflicts into account. In some cases performance degrades due to inaccurate estimation. In order to obtain more precise values, the cost computation is coupled to the scheduler. This process is split into two phases. In the first phase, the scheduler for the schemes with jump instructions are obtained. In the second, those for the schemes using conditional instructions are obtained. However, it should be noted that neither registers are allocated nor setup code is generated in this phase of the compiler. Hence, the cost values are still estimates. See Section 9.5 for the detailed code generation flow. The scheduler works only on the basic block level. Hence, the statements $(s_1, \ldots s_n)$ in the `then` and `else` blocks (or the merger of both) are grouped to the corresponding basic blocks $(G_1, \ldots, G_m)$. The scheduler provides for each block G_i the number of cycles it needs to execute, henceforth referred to as *fillcycles* $F(G_i)$. Now, the cost for the blocks (i.e., B_T, B_E, $B_\mathrm{T} \circ B_\mathrm{E}$) are obtained as follows:

$$C(B) = \sum_{i=1}^{m} \left(F(G_i) + \begin{cases} \min\{C_1(s_{i-}), C_2(s_{i-}), C_9(s_{i-}), C_{11}(s_{i-})\} - F(G_i) & s_{i-} \text{ is ITE stmt,} \\ \min\{C_5(s_{i-}), C_6(s_{i-})\} - F(G_i) & s_{i-} \text{ is IT stmt,} \\ 0 & \text{else.} \end{cases} \right) \tag{9.7}$$

$$C^P(B) = \sum_{i=1}^{m} \left(F(G_i) \right.$$

$$+\left\{\begin{array}{ll}\min\{C_3(s_{i-}), C_4(s_{i-}), C_{10}(s_{i-}), C_{12}(s_{i-})\} - F(G_i) & s_{i-} \text{ is ITE stmt,}\\ \min\{C_7(s_{i-}), C_8(s_{i-})\} - F(G_i) & s_{i-} \text{ is IT stmt,}\\ 0 & \text{else.}\end{array}\right) \tag{9.8}$$

In case the last statement in a block s_{i-} is an IT or ITE statement,[1] its costs have to be taken into account as well. As can be seen later, these costs already contain the fillcycles of the hosting basic block, thus they are subtracted again. In the first phase of the cost computation, only the cost values of the implementation schemes ITEJMP and ITJMP are available, hence the terms

$$\min\{C_1(s_{i-}), C_2(s_{i-})\} \quad \text{and} \quad \min\{C_5(s_{i-}), C_6(s_{i-})\} \tag{9.9}$$

reduce to

$$C_1(s_{i-}) \quad \text{and} \quad C_5(s_{i-}) \tag{9.10}$$

The cost for these two schemes can be calculated as follows:

$$C_1(s_{i-}) = S_1 + F(G_i) + \left\{\begin{array}{ll}C(B_\mathrm{T}) + J_\mathrm{taken} & P(B_\mathrm{T}) > p \wedge P(B_\mathrm{T}) > P(B_\mathrm{E}),\\ C(B_\mathrm{E}) + J_\mathrm{nottaken} + J_\mathrm{always} & P(B_\mathrm{E}) > p \wedge P(B_\mathrm{E}) > P(B_\mathrm{T}),\\ \max\left\{\begin{array}{l}C(B_\mathrm{T}) + J_\mathrm{taken},\\ C(B_\mathrm{E}) + J_\mathrm{nottaken} + J_\mathrm{always}\end{array}\right\} & \text{else.}\end{array}\right. \tag{9.11}$$

$$C_5(s_{i-}) = S_5 + F(G_i) + \left\{\begin{array}{ll}J_\mathrm{taken} & P(B_\mathrm{E}) > p \wedge P(B_\mathrm{E}) > P(B_\mathrm{T}),\\ C(B_\mathrm{T}) + J_\mathrm{nottaken} & P(B_\mathrm{T}) > p \wedge P(B_\mathrm{T}) > P(B_\mathrm{T}),\\ \max\left\{\begin{array}{l}J_\mathrm{taken},\\ C(B_\mathrm{T}) + J_\mathrm{nottaken}\end{array}\right\} & \text{else.}\end{array}\right. \tag{9.12}$$

For example, the costs for the scheme ITEJMP is composed of the setup cost S_1, the fillcycles of the block containing the condition evaluation, and an additional summand that depends on the given transition probabilities. Either the time for execution of the `then` block plus the jump delay of the conditional jump to reach it, or for the `else` plus a not taken jump plus an unconditional jump or the maximum

[1] Only the last statement in a basic block can be a control-flow statement, cf. Section 3.3.1.

(i.e., the worst case) of both is added. In order to provide the possibility to switch off transition probabilities, an user-defined threshold p can be passed to the cost computation, which is set to 1 by default.

In the second phase, the conditional schemes are computed as follows:

$$C_2(s_{i-}) = S_2 + F(G_i) + \begin{cases} 0 & D = 1, \\ C^P(B_\mathrm{T}) + C^P(B_\mathrm{E}) & \text{else.} \end{cases} \tag{9.13}$$

$$C_3(s_{i-}) = S_3 + F(G_i) + \begin{cases} C(B_\mathrm{T}) + J_\mathrm{taken} & P(B_\mathrm{T}) > p \wedge P(B_\mathrm{T}) > P(B_\mathrm{E}), \\ C^P(B_\mathrm{E}) + J_\mathrm{nottaken} + J_\mathrm{always} & P(B_\mathrm{E}) > p \wedge P(B_\mathrm{E}) > P(B_\mathrm{T}), \\ \max \left\{ \begin{array}{l} C(B_\mathrm{T}) + J_\mathrm{taken}, \\ C^P(B_\mathrm{E}) + J_\mathrm{nottaken} + J_\mathrm{always} \end{array} \right\} & \text{else.} \end{cases} \tag{9.14}$$

$$C_4(s_{i-}) = S_4 + F(G_i) + C^P(B_\mathrm{T}) + C^P(B_\mathrm{E}) \tag{9.15}$$

$$C_6(s_{i-}) = S_6 + F(G_i) + \begin{cases} 0 & D = 1, \\ C^P(B_\mathrm{T}) & \text{else.} \end{cases} \tag{9.16}$$

$$C_7(s_{i-}) = S_7 + F(G_i) + \begin{cases} J_\mathrm{taken} & P(B_\mathrm{E}) > p \wedge P(B_\mathrm{E}) > P(B_\mathrm{T}), \\ C(B_\mathrm{T}) + J_\mathrm{nottaken} & P(B_\mathrm{T}) > p \wedge P(B_\mathrm{T}) > P(B_\mathrm{E}), \\ \max \left\{ \begin{array}{l} J_\mathrm{taken}, \\ C(B_\mathrm{T}) + J_\mathrm{nottaken} \end{array} \right\} & \text{else.} \end{cases} \tag{9.17}$$

$$C_8(s_{i-}) = S_8 + F(G_i) + C^P(B_T) \tag{9.18}$$

$$C_9(s_{i-}) = S_9 + F(G_i) + C^P(B_\mathrm{T}) + \begin{cases} \Delta(J_\mathrm{taken}, B_\mathrm{T}) & P(B_\mathrm{T}) > p \wedge P(B_\mathrm{T}) > P(B_\mathrm{E}), \\ C(B_\mathrm{E}) + \Delta(J_\mathrm{nottaken}, B_\mathrm{T}) & P(B_\mathrm{E}) > p \wedge P(B_\mathrm{E}) > P(B_\mathrm{T}), \\ \max \left\{ \begin{array}{l} \Delta(J_\mathrm{taken}, B_\mathrm{T}), \\ C(B_\mathrm{E}) + \Delta(J_\mathrm{nottaken}, B_\mathrm{T}) \end{array} \right\} & \text{else.} \end{cases} \tag{9.19}$$

$$C_{10}(s_{i-}) = S_{10} + F(G_i) + C^P(B_\mathrm{E})$$

$$+\begin{cases}\Delta(J_{\text{taken}}, B_{\text{E}}) & P(B_{\text{E}}) > p \wedge P(B_{\text{E}}) > P(B_{\text{T}}),\\ C(B_{\text{T}}) + \Delta(J_{\text{nottaken}}, B_{\text{E}}) & P(B_{\text{T}}) > p \wedge P(B_{\text{T}}) > P(B_{\text{E}}),\\ \max\left\{\begin{array}{l}\Delta(J_{\text{taken}}, B_{\text{E}}),\\ C(B_{\text{T}}) + \Delta(J_{\text{nottaken}}, B_{\text{E}})\end{array}\right\} & \text{else.}\end{cases} \tag{9.20}$$

$$C_{11}(s_{i-}) = S_{11} + F(G_i) + C^P(B_{\text{T}})$$

$$+\begin{cases}\Delta(J_{\text{taken}}, B_{\text{T}}) & P(B_{\text{T}}) > p \wedge P(B_{\text{T}}) > P(B_{\text{E}}),\\ C(B_{\text{E}}) + \Delta(J_{\text{nottaken}}, B_{\text{T}}) & P(B_{\text{E}}) > p \wedge P(B_{\text{E}}) > P(B_{\text{T}}),\\ \max\left\{\begin{array}{l}\Delta(J_{\text{taken}}, B_{\text{T}}),\\ C(B_{\text{E}}) + \Delta(J_{\text{nottaken}}, B_{\text{T}})\end{array}\right\} & \text{else.}\end{cases} \tag{9.21}$$

$$C_{12}(s_{i-}) = S_{12} + F(G_i) + C^P(B_{\text{E}})$$

$$+\begin{cases}\Delta(J_{\text{taken}}, B_{\text{E}}) & P(B_{\text{E}}) > p \wedge P(B_{\text{E}}) > P(B_{\text{T}}),\\ C(B_{\text{T}}) + \Delta(J_{\text{nottaken}}, B_{\text{E}}) & P(B_{\text{T}}) > p \wedge P(B_{\text{T}}) > P(B_{\text{E}}),\\ \max\left\{\begin{array}{l}\Delta(J_{\text{taken}}, B_{\text{E}}),\\ C(B_{\text{T}}) + \Delta(J_{\text{nottaken}}, B_{\text{E}})\end{array}\right\} & \text{else.}\end{cases} \tag{9.22}$$

The case differentiation in the formulas $C_2(s_{i-})$ and $C_6(s_{i-})$ is actually not necessary, because $C^P(B_{\text{T}})$ as well as $C^P(B_{\text{E}})$ are zero. The blocks were appended to the `if` block, and thus the costs are already contained in $F(G_i)$. However, writing it this way makes explicit that this is only the case if the depth D of the if-statement equals one, i.e., it is the innermost ITE statement. This is mainly due to a restriction of the underlying CoSy framework. The ITE blocks cannot be merged if $D > 1$, so their costs must be added explicitly.

Finally, all cost values are available and the best implementation schemes can be selected.

9.3.4 Selecting the Best Scheme

Obviously, the decision of applying if-conversion depends on the corresponding costs, which again depends on the execution times of nested ITE statements (*bottom-up dependency*). On the other hand, the costs of a nested ITE statement depend on the presence or absence of a precondition, which is determined by the implementation scheme of the surrounding ITE statement (*top-down dependency)*. Therefore, the best scheme cannot be determined in a single bottom-up or top-down pass.

The search space is specified by an *ITE tree* $T = (R, B_T, B_E)$. The root R is a boolean expression, which is the condition of the ITE statement. The ITE blocks B_T and B_E correspond to the `then` and `else` blocks, respectively. The scheme selection is based on a dynamic programming algorithm as presented in [220]. This method is similar to the well-known tree-pattern-matching algorithm. It performs two steps to select the right implementation scheme. In the first phase, all ITE trees are traversed bottom-up filling the cost tables for each node. The second pass is top-down. When the root node is reached, the scheme corresponding to the cheapest entry in the root's cost table is selected. Based on this selection, it is known whether a precondition for the son is present or not. This determines the set of schemes (i.e., those with or without precondition) among which the cheapest scheme is selected and so forth. This is illustrated in Fig. 9.4.

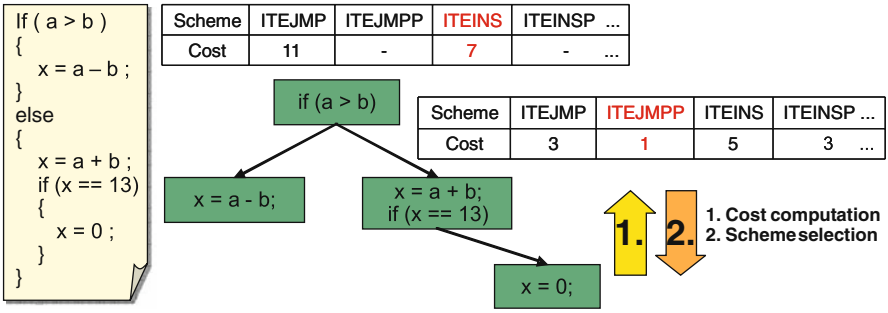


Fig. 9.4 ITE tree, annotated cost tables, and scheme selection

9.3.5 Splitting Mechanism

During benchmarking, it turned out that for more complex programs, only a small percentage of the existing if-statements have been processed at all for various reasons: the cost computation might decide against if-conversion, one ITE block might have multiple incoming control-flow edges, or one or both ITE blocks might contain hampering elements, e.g., nonpredicable statements. This is exemplified in Fig. 9.5. The red lines indicate nonconditionally executable statements.

Obviously, if-conversion cannot be applied to the ITE statement on the left. However, assuming that the statements B and C are independent from each other, the code depicted on the right can be obtained. So far, only the statement level has been considered. Looking at the pseudocode level (basically the assembly-level representation of the source code), it can be observed that not all instructions selected for the statement are necessarily not conditionally executable. Consequently, working on the pseudocode level allows a more fine-grained operation by moving single pseudocode nodes. The basic idea is to move these nodes to the block containing the condition evaluation of the remaining (nonpredicated) ITE statement. Since it typically contains only few instructions, most likely not all delay slots of the

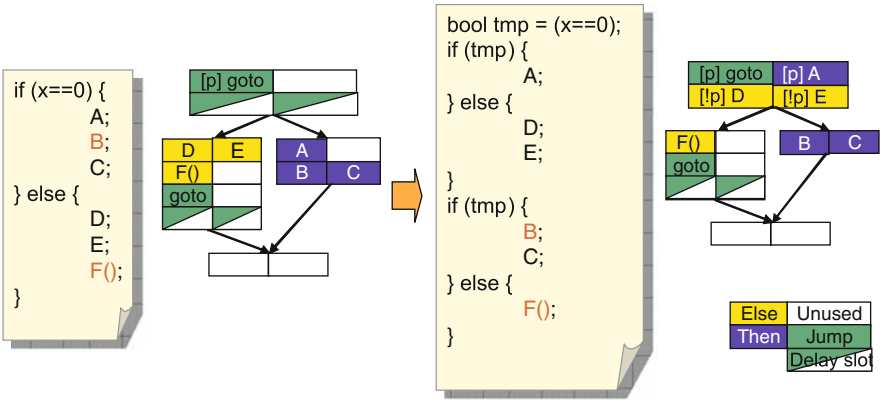


Fig. 9.5 Splitting example for a processor with two-issue slots

conditional jump can be filled. Of course this procedure has its limits. Only as many nodes as empty delay slots should be moved to avoid a performance degradation.

This idea is implemented with the *splitting mechanism*. The algorithm processes only nonpredicated ITE statements in which `then` and `else` blocks have a single incoming control-flow edge. This restriction avoids a complicated performance analysis because otherwise compensation code has to be taken into account as well. Afterward, assembly instructions are moved from the ITE blocks as illustrated in Figure 9.5. It alternately selects instructions from the `then` and `else` blocks (i.e., `A`, `D`, and `E` in the example) and moves them into the delay slots of the conditional jump where they are predicated. An instruction is considered movable, if it can be predicated and does not change the control flow. Furthermore, it must not write a predicate that is used as a condition of the jump or as guard of an ITE block (in case of partial if-conversion). Moreover, it must not depend on an instruction which is nonmovable to simplify the dependency analysis. If a nonmovable instruction is found in one block, it proceeds with instructions from the other block. The algorithm stops either if no more movable instructions are found or if a configurable threshold (3 in the example) is reached. Note that the pseudocode list is reordered in advance: after a nonmovable node, there could be other movable nodes in the pseudocode list that have no dependencies to the nonmovable node. Thus, for each node that comes after a nonmovable node, it is checked whether it depends on the nonmovable node. In that case, it is marked as nonmovable. Otherwise it is moved before the nonmovable node.

9.4 Retargeting Formalism

An evaluation [89] of several processors for different application domains showed that processors featuring PE can be grouped according to the location, the guard is stored in. Chiefly, the following three categories can be obtained:

1. Processors using general-purpose registers.
2. Architectures using dedicated registers.
3. Architectures that use condition flags stored in a status register.

The first retargeting step is to configure the cost computation. Three boolean parameters for the PE engine specify to which of the above classes the target architecture belongs. Another boolean parameter indicates whether the architecture directly supports negated conditions or not. Furthermore, the jump penalty J for a conditional jump taken, a conditional jump not taken, and an unconditional jump needs to be provided.

Moreover, some of the architectures can execute a wide subset of their instruction-set conditionally, others offer only for a few instructions of a predicated version. In order to determine whether an instruction or a basic block can be conditionally executed by the target processor, the generated tree-covering-based code selector is employed. As mentioned in Section 3.3.2, each rule describes how a certain IR operation is mapped to the target assembly code. For retargeting the PE optimization, each rule of the code selector that can emit a code that is conditionally executable has to be annotated. Listing 9.15 shows two examples for the TriMedia [190] processor. The rule, covering a plus node can be conditionally executed (denoted by `peinclude`). The other rule, which loads an immediate value to a register, is missing that annotation, and thus is assumed to be not conditionally executable by default. Consequently, if one of the rules covering the `then` or `else` block is missing that annotation, if-conversion cannot be applied to the corresponding if-statement. Furthermore, the instructions of such a rule cannot be moved by the splitting mechanism.

```
RULE o:mirPlus(s1:reg_nt,s2:reg_nt) -> d:reg_nt;
CLASS peinclude;
EMIT {
 print_with_condition("\tiadd  %s   %s -> %s",
       REGNAME(s1),REGNAME(s2),REGNAME(d));
}

RULE o:mirIntConst -> d:reg_nt;
EMIT {
 print("\tuimm( %s ) -> %s ",o.Value,REGNAME(d));
}
```

Listing 9.15 Annotated TriMedia code selector rules

For the code generation, the code emitter must take care to print the correct assembly syntax (see Listing 9.15) in case the rule is used in a predicated block. For instance in the case of the TriMedia, the print function must prepend an `IF <condition register>` to the given instruction in case the instruction is executed conditionally.

```
// Register r0 is always zero and r1 always one
INSTRUCTION peSetCondition (cond:reg_nt) -> d:reg_nt;
EMIT {
 print("IF %s  iadd  r1   r0 -> %s",
        REGNAME(cond),REGNAME(d));
}
INSTRUCTION peResetCondition (cond:reg_nt) -> d:reg_nt;
EMIT {
 print("IF %s  iadd  r0   r0 -> %s ",
       REGNAME(cond),REGNAME(d));
}
INSTRUCTION peNegateCondition (s:reg_nt) -> d:reg_nt;
EMIT {
 print("IF r1 bitinv  %s -> %s",
       REGNAME(s),REGNAME(d));
}
INSTRUCTION peBranchAlways (label:BasicBlock);
EMIT {
 print("IF r1 ijmpi ( %s )",label);
}
INSTRUCTION peBranchCond (cond:reg_nt,label:BasicBlock);
EMIT {
 print("IF %s  ijmpi ( %s )",REGNAME(cond),label);
}
```

Listing 9.16 PE instruction rules for the TriMedia

The rules covering an if-statement are responsible to generate the code for the selected ITE scheme. Note that the generated code depends not only on the scheme but also on the order of the `then` and `else` blocks in memory. So either the `then` block directly follows the if-statement (fallthrough) or the `else` block. In certain cases, neither of them follows the if-stament directly. As mentioned above, some cases can only be handled with dedicated implementation schemes, whereas for others it is sufficient to adapt the code generation. Nevertheless, all implementation schemes can be generated with the following few instructions:

peSetCondition conditionally sets a predicate to true
peResetCondition conditionally sets a predicate to false
peNegateCondition conditionally inverts a condition
peBranchAlways unconditional jump instruction
peBranchCond conditional jump instruction

Retargeting the code generation is limited to fill in rule templates for these instructions with the assembly code that has to be emitted. Listing 9.16 shows the filled templates for the TriMedia processor. Additionally, each if-statement rule must call a generic function instead of printing anything. No other information, apart from the already described, needs to be provided to retarget the extension. This can

also be performed via the *Compiler Designer* GUI. In this way, the PE optimization can be quickly retargeted to varying processor configurations during architecture exploration.

9.5 Code Generation Flow

Due to the modular concept of CoSy, it is straightforward to intertwine the standard backend components (tree pattern matcher, scheduler, and register allocator) with the PE modules. Figure 9.6 depicts the backend of a CoSy compiler with PE support.

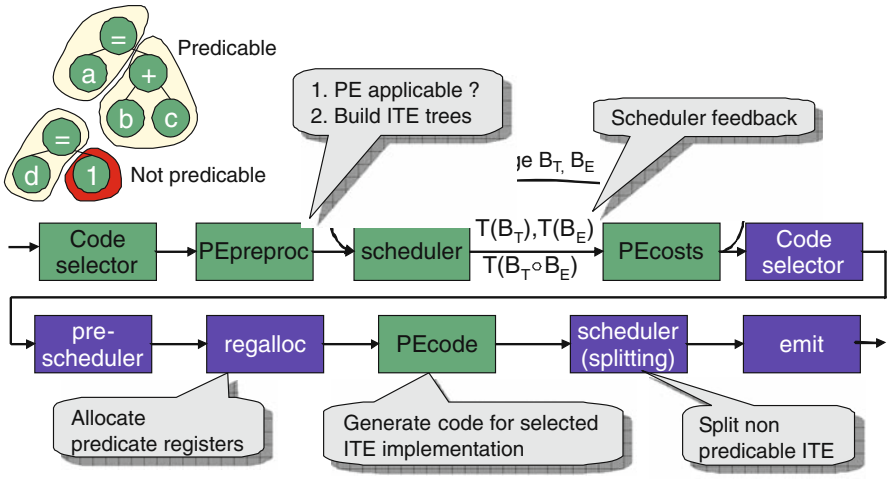


Fig. 9.6 CoSy compiler backend with PE support

After an initial code selection with the standard tree pattern matcher, the engine *PEpreproc* builds ITE trees and determines those if-statements to which if-conversion can be applied. Reasons for an exclusion can be multiple incoming control-flow edges of the `then` or the `else` block as well as a nonpredicable code in an ITE block. The latter is detected utilizing the already described rule annotations. If a basic block is covered by a rule-emitting nonconditionally executable code, an infinite cost value is assigned to the PE schemes of the corresponding if-statement. Then the costs of the different schemes are calculated and the scheme selection is performed by the engine *PEcosts* (Section 9.3.3). This engine is coupled to the normal scheduler of CoSy. In the first iteration, the scheduler calculates the execution times of each basic block. These are used to compute the costs for the implementation with jump instructions. Afterward, *PEcosts* instructs the scheduler to merge the `then` and `else` blocks of the innermost statements. The scheduler parallelizes them and provides cost estimates of the block merger. Thereafter, *PEcosts* selects the schemes according to the calculated costs. After the final code selection and register allocation, the engine *PEcode* generates the code for the

chosen schemes using the above-mentioned instructions. The splitting mechanism operates within the scheduler and targets all if-statements to which if-conversion could not be applied. Apart from the compiler's data-flow information, it uses the annotations by the tree pattern matcher whether an instruction is predicable or not. Finally, the code is emitted.

This approach requires limited retargeting information, also due to the coupling to existing compiler backend modules. These are typically part of any retargetable compiler. Thus, this approach is not limited to the CoSy platform, and consequently it can be easily incorporated into other compiler platforms as well.

9.6 Experimental Results

The presented technique was successfully integrated into CoSy compilers for the Adelante™ VD32040 embedded vector processor (EVP) [152] and the TriMedia multimedia processor, both from NXP semiconductors [190], as well as the ARM9 [41]. The required retargeting effort for PE support was 1 day for each compiler. All three architectures can execute almost all their instructions conditionally. The TriMedia can use any of its 128 general-purpose registers to store the predicate, whereas the EVP features eight dedicated predicate registers. The negated predicate has to be computed explicitly for both processors. The ARM uses condition code flags for predication. It can store one condition at a time in the status register and supports negation. Thus, each processors belongs to one of the groups mentioned in Section 9.4. The maximum VLIW parallelism available in the EVP equals five vector operations, four scalar operations, three address operations, and loop-control. The TriMedia can process up to five operations in parallel. The EVP jumps have five to seven delay slots while the TriMedia jumps have two. In contrast, the ARM is a RISC-like core. Since the ARM has no delay slots, the splitting mechanism was disabled. The only benefit by PE for the ARM lies in the elimination of jump instructions.

The benchmarks consists of some smaller, typical signal-processing kernels (up to 70 ITE statements) as well as some larger and more complex applications (up to 2000 ITE statements). The total number of if-statements vary between the compilers due to their different design and integrated optimizations. Tables 9.2 and 9.3 show detailed statistics for the total number of ITE statements, those that are recognized by *PEpreproc* and how many have been finally converted and split, respectively. If not stated otherwise, the test data that comes with these benchmarks is used for the measurements and it is optimized for the worst-case execution time.

For the small benchmarks, *PEpreproc* determines that on average 80% of all if-statements can be considered for PE, the only exception being the `viterbi` [124] for the EVP with no predicable if-statements. Almost all these if-statements could finally be converted for the EVP, whereas the TriMedia could not convert all of them. This is mainly due to the higher degree of parallelism the EVP offers over the TriMedia. Thus, the chance is higher in TriMedia for resource conflicts resulting in

Table 9.2 If-statement statistics for ARM and EVP

	ARM				EVP			
	if-stmts	recognized	converted	split	if-stmts	recognized	converted	split
adpcm	24	16	16	–	18	16	16	0
viterbi	43	40	38	–	2	0	0	0
median	53	51	51	–	13	13	13	0
wave	62	59	58	–	3	3	3	0
idct	65	64	63	–	16	16	15	1
cjpeg	1360	143	88	–	1994	307	202	1555
djpeg	1118	118	89	–	1934	306	206	1554
printf	198	33	22	–	97	30	16	66
miniLzo	63	2	1	–	54	3	2	42

Table 9.3 If-statement statistics for TriMedia

	TriMedia			
	if-stmts	recognized	converted	split
adpcm	26	22	12	16
viterbi	6	2	2	3
median	14	13	13	0
wave	7	3	3	3
idct	20	16	8	11
cjpeg	2870	442	142	1636
djpeg	2894	441	143	1662
printf	118	47	43	67
miniLzo	142	9	3	88

longer schedules, and hence higher costs for predicated if-statements. Consequently, more if-statements are split for the TriMedia than for the EVP. Figure 9.7 shows high speedups for the VLIW processors, whereas the ARM shows smaller speedups.

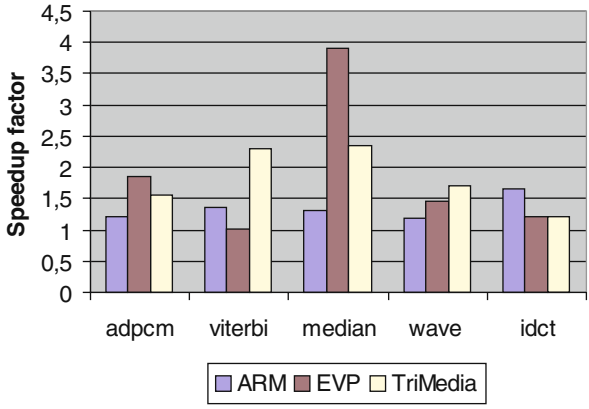


Fig. 9.7 Speedup for small benchmarks

The programs `cjpeg` and `djpeg` [53, 154] feature a large amount of if-statements (around 2000); however, only approximately 15% of them were recognized by *PEpreproc* for if-conversion. Finally, only 6–10% of all if-statements could be converted by the compilers. Here, the splitting mechanism proves advantageous and handles nearly 80% (EVP) and 60% (TriMedia) of all if-statements. The ARM shows only marginal speedups due to the disabled splitting mechanism, but EVP and TriMedia show good speedups for both `cjpeg` and `djpeg` (Fig. 9.8). The obtained speedups are less significant than for the small kernels. This is understandable considering that the cycles spent in the runtime library for file operations dominate the execution time.

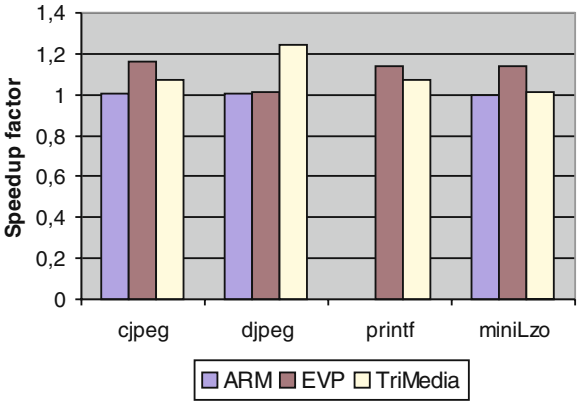


Fig. 9.8 Speedup for large benchmarks

Considering the `printf` (implementation is shipped with CoSy) application, it contains many if-statements (around 100), approximately 17% are converted and around 60% are split by the EVP and TriMedia compilers. No results are reported for the ARM, since it could not be compiled due to a different runtime library setup.

For `miniLzo` [196], although it features many if-statements (around 80), only a few could be converted. A look into the source code revealed that the if-statements either contain function calls or goto statements. These kind of if-statements are not allowed by *PEpreproc*, and thus no performance improvement can be obtained. However, except for the ARM, the splitting mechanism can be applied again and optimizes almost all if-statements.

On average, speedups of 1.2 for the ARM9, 1.5 for the EVP, and 1.47 for the TriMedia can be obtained.

For the code size, PE typically saves some instructions (jumps and nops), but may also generate new ones (e.g., negated conditions). In general, the code size is slightly reduced (see Fig. 9.9).

The optimization algorithm itself has linear complexity ($O(n)$ worst-case complexity for n ITE statements). Furthermore, it requires one additional tree-pattern-

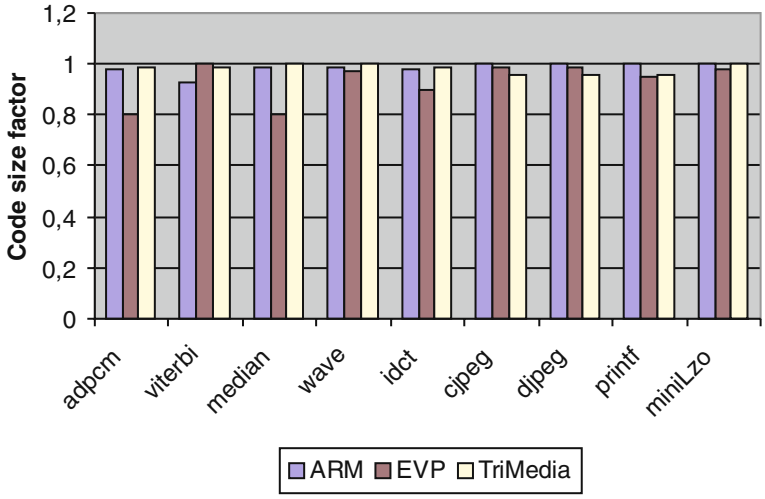


Fig. 9.9 Code-size results for all benchmarks

matcher pass and two additional scheduler passes. For n IR nodes in the ITE statements, the worst-case complexity of tree pattern matching is $O(n)$ whereas for scheduling it is $O(n^2)$. Thus, the total worst-case complexity is quadratic.

9.7 Conclusions

In contrast to previous, largely target-specific, code optimizations for predicated execution, this book provides a *retargetable* approach in order to enable PE for a wide range of processor architectures at limited manual effort. This is achieved by a *retargetable predicated execution* extension for the CoSy compiler development system. This concept has been proven by generating PE-enabled compilers for embedded processors with different PE configurations. Generally, for all processors good speedups and a slight code-size reductions are achieved. The required retargeting information are quite limited and its specification fits nicely into the *Compiler Designer* concept (cf. Section 5.3). Thus, the integration enables a complete and retargetable path from a single-processor model, written in the LISA ADL, to a C compiler with PE optimization.

Further improvements in code quality seem possible. For instance, conditions of if-statements are often composed of expressions combined with boolean operations. In order to satisfy the short-circuit evaluation, this is mapped onto several nested ITE statements. If the evaluation of the individual expressions is free of side effects, they can be evaluated in parallel. This idea could be implemented by a new scheme for the PE engines (see Fig. 9.10).

Furthermore, a mechanism to enforce PE for certain ITE statements might be useful even if this would result in a performance degradation. Control flow caused

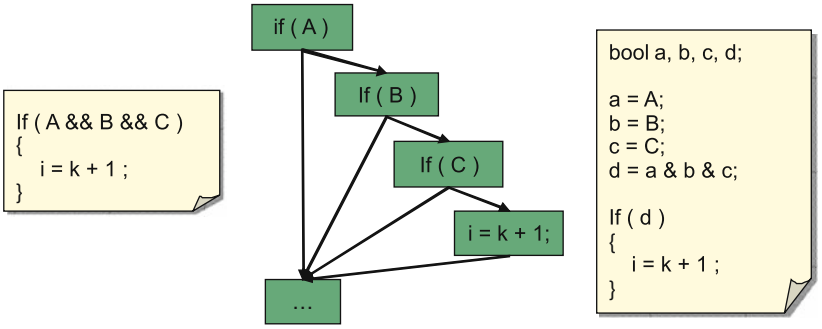


Fig. 9.10 Short-circuit evaluation

by if-then-else statements can block other optimizations, e.g., software pipelining. Thus, removing the control flow by predicated execution may enable other optimization. In the end this might result in faster code.

Chapter 10
Assembler Optimizer

Some optimization can only be performed on the assembly level of the application. This chapter presents a retargetable low-level, assembly code-optimization interface that is generated from a LISA description. Figure 10.1 illustrates the corresponding code generation flow.

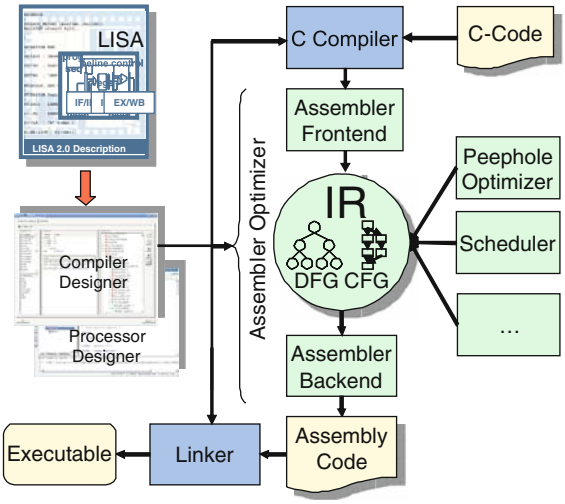


Fig. 10.1 Assembler optimizer code generation flow

Using the LISA ADL, a stand-alone assembler is automatically generated that is able to perform user-defined transformations or optimizations on assembly level. In this way, standard assembly-level optimizations only need to be implemented once and are automatically retargeted to a given LISA model. The assembler optimizer provides an user-accessible, convenient *application programmer interface* (API) for accessing the assembler's internal data structures. An internal IR data structure is created based on the input assembly code. The IR contains the related architectural information required by control- and data-flow analysis, a prerequisite

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_10,

for most optimization techniques. This enables the ASIP designer to implement optimizations addressing special ISA features such as

- Peephole optimization
- Address code optimization
- Register (re-)allocation
- Coupling of register allocation and scheduling
- Bit-level manipulation instructions
- ...

The remainder of this chapter is arranged as follows. After the discussion of related work in Section 10.1, Section 10.2 briefly describes the functions provided by the API. Sections 10.3 and 10.4 present a scheduler and peephole optimizer that have been build as demonstrators. Finally, Section 10.5 provides some results.

10.1 Related Work

The PROPAN system [66] is a retargetable framework for code optimizations and machine-dependent program analyses at the assembly level. Its main focus is postpass optimization in order to reuse existing software tool chains. It needs a separate target specification called TDL to retarget the optimization modules. Several optimization modules based on integer linear programming have been implemented and retargeted to different real-world DSPs. It can also be used as a platform for generic program analysis, e.g., to calculate worst-case execution times.

A similar approach is the SALTO system [75]. Based on an ADL description of the target machine, it generates the functionality to build profiling, tracing, or optimization tools. It is intended to be part of a global solution for manipulating assembly code, i.e., to implement low-level code modifications as well as to provide a high-level code re-structurer with useful information collected from assembly code and instruction profiling. However, it is more oriented towards general purpose processors and many architectural-specific properties of ASIPs cannot be modeled at all.

The LANCE compiler platform [222] supports the generation of a low-level, assembly-like representation called LLIR. Standard assembly-level optimizations only need to be implemented once and are automatically available when LANCE has been retargeted to a new target architecture. For instance, in [142], a bit-true data-flow analysis is performed on the LLIR which has been successfully used to implement dedicated optimization for network processors supporting bit-packing instructions.

10.2 Application Programmer Interface

Nearly all optimization modules that can be built on top of the API require access to architectural information. For instance, it might be necessary to recover the semantic of the instruction currently parsed by the assembler. As an example, the API

supports an easy way to extract which registers are used as destination and source operands or if the instruction is a control-flow instruction. Basically, this information is either directly extracted from the BEHAVIOR section or from the SEMANTICS section, if available. Though only the latter gives precise information due to the semantic gap mentioned in Section 6.1. Furthermore, this information is used to perform a control- and data-flow analysis (cf. Section 3.3.1). However, reconstructing the CFG from compiled and probably scheduled code is not an easy task. In particular, two features can complicate CFG construction. First, a destination address stored in a register (instead of a label or immediate constant) introduce a level of uncertainty, which may lead to spurious edges in the graph. Second, in case of scheduled code, jump-delay slots complicate the process of finding the first and last instruction of a basic block. These is even more difficult once the delay slots contain further branches. The implemented algorithm, described in [144], can handle such problems. Nevertheless, the information whether the input code is scheduled or not must be passed as an option to the API before the analysis can be started. Afterward, the functions to access and iterate control- and data-flow graphs can be used.

Furthermore, it is possible to modify the instruction and sequences thereof. Naturally, inserting an instruction invalidates the control- and data-flow graph. Consequently, it has to be repaired afterward. However, an automatic repair function is currently not supported.

Since all architectural information are available, each instruction element (register operand, VLIW slot, etc.) that corresponds to a LISA operation can be modified, e.g., a LISA label. Moreover, the information from the generated scheduling tables, such as RAW, WAW, and WAR latency, are also available. Hence, the kind of dependency and its latency between two instruction can be easily determined.

Finally, the API constitutes basic assembler-related functions such as file I/O functions to read and write assembly or object files. Using the API, the implementation of an assembler (without any optimizations) is straightforward, basically just a main function containing a few function calls.

10.3 Scheduler

The current scheduler generated by the *Compiler Designer* tool [195] has several limitations. All architectural information (instruction properties such as latencies and resource usage) is transferred to the scheduler via annotations in the compiler-generated assembly code. Listing 10.1 illustrates this. Each instruction is encoded in three so-called `packs` assembler directives. That means, the scheduler cannot schedule handwritten assembly code that comes without these annotations.

Of course, the user could add them manually, but since the syntax is quite complicated this is time-consuming and error-prone. Now this information is also available through the API. Hence, a new scheduler, based on the existing implementation,

has been created that does not need these annotations anymore (Listing 10.2). This allows a stand-alone, user-friendly assembly-level scheduling that is independent from a compiler on top of the flow.

```
.packs "alu_rrr;P1;C1;T2;",1
.packs "PC:(r,0);prog:(r,0);R15:(r,0);R6:(r,0);R1:(w,0);",2
.packs "add  R1 , R15 , R6      ;; Add two register ",3
.packs "ld_rr;P3;C3;T10;",1
.packs "PC:(r,0);data:(r,0);prg:(r,0);R1:(r,0);R2:(w,0);",2
.packs "lb  R2 , R1 , 0 ;; Load signed byte",3
```

Listing 10.1 Annotated assembly code

```
add  R1, R15, R6
lb  R2, R1, 0
```

Listing 10.2 Normal assembly code

10.4 Peephole Optimizer

As the second demonstrator, an architecture-independent peephole optimizer [49, 127] has been implemented, called *lpeep*. It is a classical optimization that runs after the compiler. Basically, it tries to improve the performance of the target program by searching for a short sequence of target instructions and replacing it with a better sequence. It can be easily implemented using the API functions to read and write assembly files as well as those to remove, insert, or delete assembly lines or single instructions in VLIW slots. No scheduler or data- and control-flow functions are needed. The peephole optimizer is driven by an user-defined *replacement library*. However, the peephole optimizations, i.e., the replacement patterns, are not automatically generated as described, e.g., in [128]. Implementation wise, the library puts an abstraction layer on top of the API in order to reuse large parts of the optimizer for different target architectures. Thus, a peephole optimizer can actually be generated for any LISA model. The library is then used to retarget the optimizer to the given target. The input of *lpeep* is either the assembly code produced by a compiler or a hand-crafted assembly program.

10.4.1 Replacement Library

The replacement library describes the assembly patterns and their related replacement. Each entry, called *replacement rule*, consists of three parts: the variable

definitions, the original section, and the replacement section. Figure 10.2 gives an example. Generally, variables are registers or immediate values that can be used in assembly instructions. The original section is used to find matching lines in the source file, which are then replaced by the pattern defined in the replacement section. Inside the patterns, the assembly syntax of the target architecture is used. Since the API provides all architectural information, quite detailed assembly patterns can be specified. This is described in the following sections.

TRANSFORM (<*variable list*>) {
    <*original section*>
} TO {
    <*replacement section*>
}

Fig. 10.2 Replacement rule

10.4.1.1 Variable Definitions

The different types that can constitute a variable are described in the following.

REGISTER: A register variable can either match all registers of the target architecture or only an user-defined subset:
REGISTER <*variable_name*> [= (<*reg1*>, <*reg2*>, ...)]
A simple example is given in Listing 10.3 in which variable `a` can only match the registers in the given set (as defined in the LISA model). Internally, *lpeep* make the assumption that each register variable relates to a different register, i.e., `a`, `b`, and `c` must match different registers.

```
TRANSFORM (REGISTER a=(R1,R2,R3,R4),REGISTER b, REGISTER c) {
  a = b;
  a = c;
} TO {
  a = c;
}
```

Listing 10.3 Register variable example

IMMEDIATE: A variable of this type will match immediate values occurring in the source file. This can be simple numerical values, symbolic labels, or arithmetic expressions. Furthermore, the user can specify conditions for the value.
IMMEDIATE <*variable_name*> [[==, !=, <, >] <*value*>]

OPERAND: These variables will match both registers and immediates. It is introduced for convenience for matching those instructions with similar assembly syntax for immediates and registers. However, conditions are not available in the definition of the operand variables.

The variables discussed so far can be used to replace single lines or fixed-length sequences of lines. *lpeep* also offers features to define rules that can also change the control flow of the assembly code. This includes variables to match labels or variable-length sequences of lines. Since the detailed behavior described in LISA are also available through the API, it is possible to specify conditions for the resource usage of the instructions matched by the wildcard. Such a feature is not available in traditional peephole optimizers.

BLOCK: The block variable is the most complex variable type provided by *lpeep*. It is used as a wildcard in the original section to match one or more assembly instructions. The user can control the block match criteria by adding a list of constraints to the block variable (Listing 10.4). Valid constraints are:

```
TRANSFORM (REGISTER a, OPERAND b, OPERAND c, BLOCK d) {
  a = b;
  BLOCK d (DONT_READ a);
  a = c;
} TO {
  BLOCK d;
  a = c;
}
```

Listing 10.4 Block variable example

1. DONT_READ <*register_variable*> || (<*reg1*>, <*reg2*>, ...)
   This constraint will exclude instructions containing read accesses of the specified register variable or physical registers from the match.
2. DONT_WRITE <*register_variable*> || (<*reg1*>, <*reg2*>, ...)
   Same as previous except for write access.
3. DONT_ACCESS <*register_variable*> || (<*reg1*>, <*reg2*>, ...)
   The combination of the DONT_READ and DONT_WRITE constraints.
4. DONT_MATCH (<*assembly statement*>)
   This constraint will exclude any lines that match the given pattern from the block match.
5. MAX_LINES <*number_of_lines*>
   This constraint will limit the number of matched instructions.

LABEL and NEWLABEL: This variable type matches labels. The NEWLABEL variables can only be used in the replacement section

of a rule to create a new label with an unique name. An example is provided in Listing 10.5.

```
TRANSFORM (LABEL l1, LABEL l2,
           BLOCK b1, BLOCK b2) {
        jmp l1;
        BLOCK b1;
        LABEL l1;
        jmp l2;
        BLOCK b2;
        LABEL l2;
} TO {
        jmp l2;
        BLOCK b1;
        LABEL l1;
        BLOCK b2;
        LABEL l2;
}
```

Listing 10.5 Label variable example

10.4.1.2 Matching VLIW Instructions

To define replacement patterns for the optimization of VLIW assembly code, *lpeep* provides the || operator to separate the different slots of a VLIW instruction. Listing 10.6 illustrates the use of the || operator. The EXTRA_SLOTS keyword is supported by *lpeep* to be used as wildcards in a VLIW instruction word. Similar to the BLOCK variables, EXTRA_SLOTS can also take constraints to restrict the matched instructions. All the constraints' definitions available for the BLOCK variables are supported in the definition of the EXTRA_SLOTS as well.

```
TRANSFORM (REGISTER a,
           REGISTER b,
           REGISTER e)
{
  a = b;||EXTRA_SLOTS c (DONT_ACCESS a);
  a = e;||EXTRA_SLOTS d (DONT_ACCESS a);
} TO {
  a = e;||EXTRA_SLOTS c|| EXTRA_SLOTS d;
}
```

Listing 10.6 VLIW pattern example

10.5 Experimental Results

The API and the presented modules are fully integrated into CoWare's *Processor Designer* environment, and thus can be generated for any LISA model. For

evaluation, the compiler with generated code selector description for the ST220 as presented in Chapter 6 has been used. Naturally, as this configuration contains only automatically generated rules, there is obviously some room for improvements that can be exploited by the peephole optimizer. Additionally, it is a good candidate to show the applicability of the peephole optimizer as it features VLIW slots and different constraints on registers (one general-purpose and one special-purpose register files) as well as LISA resources.

The replacement library for the ST220 contains 26 patterns in total. Note, its main purpose was to cover all features of the peephole optimizer, and thus it cannot be considered an optimal replacement library. The API-based scheduler contains some minor improvements as compared to the existing scheduler. Basically, implicit register accesses can be directly detected by the dependency analysis. Such dependencies must be explicitly modeled in the existing scheduler that results typically in a conservative scheduler description.

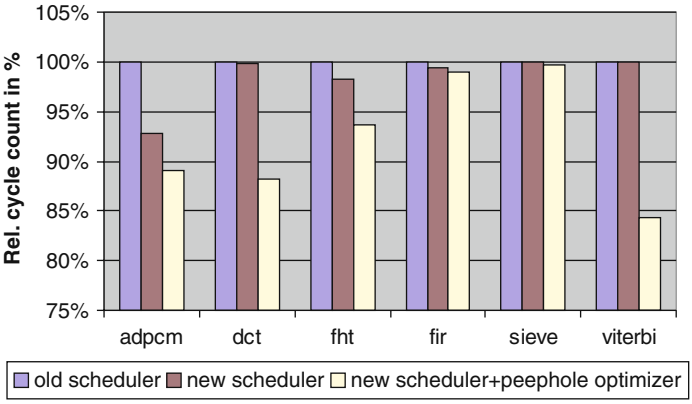


Fig. 10.3 Relative cycle count

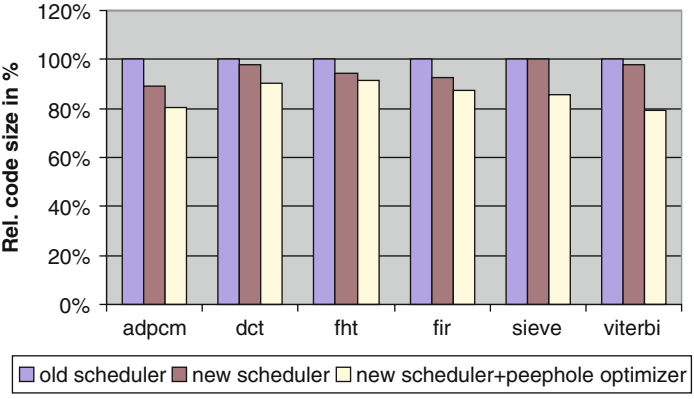


Fig. 10.4 Relative code size

In case of the ST220, this limitation prevented delay-slot filling in certain cases. This caused quite some NOP instructions at the end of a basic block. With the improved dependency analysis, this drawback could be eliminated. The improvements in cycle count (Fig. 10.3) gained by the API-based scheduler range from 0 to 7%. Consequently, as less NOPs are required, the code size is decreased up to 11% (Fig. 10.4). The improvements in cycle count achieved by the peephole optimizer range from 1 to 16%, and the code size can be reduced by 5–19%.

10.6 Conclusions

The integration of a retargetable assembler optimizer API into an ADL-based design environment enables a convenient way to implement assembly-level optimizations. Retargetable optimizations based on the API can be easily added to the environment or ASIP designers can implement their own hand-crafted optimizations. The interface provides all information (e.g., data- and control-flow information) that are typically required for such optimizations. Most important, all architectural information such as processor resources and instruction semantics are still available through the interface. In this way, optimization for irregular architecture features can be quickly implemented. To demonstrate the applicability of the API, a scheduler and a peephole optimizer have been implemented. Since both tools are retargetable, they are already integrated into the software tool generation flow of the *Processor Designer*, and thus can be generated for any LISA model. In future, more retargetable assembly-level optimization could be added to this flow.

Chapter 11
Summary

The complexity of today's SoC designs is increasing at an exponential rate due to the combined effects of advances in semiconductor technology as well as demands from increasingly complex applications in embedded systems. Escalating NRE costs have created a shift toward achieving greater design reuse with programmable SoC platforms. The choice of programmable architectures strongly affects the success of a SoC design due to its impact on the overall cost, power consumption, and performance. Therefore, an increasing number of embedded SoC designs employ ASIPs as building blocks due to their balance between flexibility and high performance by programmability and application-specific optimizations. However, given today's tight time-to-market constraints, finding the optimal balance between competing design constraints makes design automation inevitable.

Architecture description languages have been established as an efficient solution for ASIP architecture exploration. Among the main contributions of such languages is the automatic generation of the software toolkit from a single ADL model of the processor. A key component of the software toolkit is the C compiler that enables a compiler-in-the-loop design space exploration. Developing an ADL, though, is a difficult task. Today's ADLs must keep all architectural information as required for the tool generation (in particular compiler and simulator) in an unambiguous and consistent way. As a result, some ADLs are well suited for, e.g., the automatic generation of the compiler, but impose major restrictions on or are incapable of the generation of a simulator. Other ADLs suffer from limited architectural scope and are not suitable for ASIP design. An overview of existing ASIP design platforms and their capabilities is given in this book. It turned out that none of the existing approaches solves this problem satisfactory.

The contribution of this book is a technique that enables the automatic retargeting of a C compiler, more specifically the code selector description, from an ADL processor model using CoWare's Processor Designer and the CoSy environment. The developed approach incorporates a new, concise formalism for the description of instruction semantics into the LISA language definition. Several existing LISA models for representative embedded processors have been successfully enhanced with the new section at moderate effort. This proves that the new section does neither impose any particular modeling style nor does it limit LISA's flexibility. The

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6_11, © Springer Science+Business Media, LLC 2010

instruction's semantics is used by four different mapping rule generation methods which create the code selector description for a C compiler fully automatically. The CoSy compilers with generated code selector description show an overhead of 14% in cycle count and 48% in code size as compared to a compiler with (nonoptimized) hand-crafted code selector specification. These are acceptable values considering that a compiler is available early in the architecture exploration phase. This is crucial to avoid hardware/software mismatches right from the start in order to ensure good overall efficiency of SoC platforms. Moreover, the entry barrier to compiler generation is further lowered. In fact, even noncompiler experts are now able to generate compilers for architecture exploration. Additionally, the generated code selector rules are correct by construction, which eliminates the tedious debugging of code selector descriptions.

ASIP design platforms employ retargetable C compilers for compiler generation since they can be quickly adopted to varying processor configurations. Unfortunately, such compilers are known for their limited code quality as compared to handwritten compilers or assembly code due to a lower amount of target-specific optimizations. This is not surprising considering that it would be counterproductive for the flexibility required to adapt quickly to architectural alternatives. Like it has been observed in the code quality analysis of the ST220 compilers, the generated compilers must be manually refined with dedicated optimizations once the ASIP architecture exploration phase has converged and an initial working compiler is available. Hence, the second part of this book focuses on target processor classes which, due to their architectural features, demand for specific code optimization techniques. Two promising architectural classes are selected, namely processors equipped with *SIMD instructions* and those with *predicated execution* support.

This book implements these specific techniques such that retargetability within the given processor class is achieved. The SIMD optimization was retargeted to two embedded processor architectures with SIMD support. In general, the optimization achieves speedups of 7–66% and code-size reductions of up to 40% in most cases. The predicated execution optimization was retargeted to three contemporary processors. On average, it achieves a cycle count improvement of 39% and a code-size reduction of 3%.

In this way, a complete and retargetable path from a single LISA processor model to an SIMD and predicated execution-enabled compiler for efficient compiler-in-the-loop architecture exploration is achieved. Furthermore, to ease the manual creation of dedicated optimizations on the assembly level, this book implements a new retargetable assembler which provides an interface for code optimizations. A scheduler and peephole optimizer are implemented as demonstrators.

Future research aims at different directions. Tomorrow's SoC designs are heading toward heterogeneous multiprocessor systems (MP-SoC). Additionally, there is an increasing amount of embedded processor architectures which are capable of executing multiple threads of control in parallel. Apart from the general problem of identifying those parts of a sequential code like C which can be executed in parallel, there is ongoing work to extend retargetable compilers in such a way that all optimizations perform equally well on sequential as well as parallel code constructs in a

multi-threaded environment. Another recent trend in embedded processor design is a clustered VLIW organization. Compilers for such architectures must find a cluster assignment so that a good workload balance is achieved while keeping the communications costs between the clusters low. Developing retargetable techniques to support the efficient exploration of such architectures is an interesting topic. Future research also aims at finding new methodologies for DAG-based code selection. This enables the direct exploitation of inherently parallel hardware instructions, which are a very common extension of ASIP processors, by compilers. Another topic is the identification of those data-flow trees or graphs, which actually could be promising candidates to be implemented in hardware.

Appendix A
Semantics Section

The SEMANTICS section of the LISA language provides a simple, straightforward syntax, which allows the direct transformation of the instruction's purpose into an as-short-as-possible semantical description. The complete grammar specification is given in Section A.3.

A.1 Semantics Statements

There are four different kinds of semantics statements in SEMANTICS sections (Fig. A.1).

semantics_statement ::= *assignment_statement*
| *if_else_statement*
| *modes_statement*
| *non_assignment_statement*

assignment_statement ::= *source_expression* '->' *destination_expression* ';'

source_expression ::= *micro_operation_expression*
| *integer*
| *LISA_declared_item*
| *semantics_related_resources*

destination_expression ::= *LISA_declared_item*
| *indir_expression*
| *semantics_related_resources*

semantics_related_resources ::= **_PC** | **_SP** | **_CF** | **_OF** | **_NF** | **_ZF**

The *indir_expression* refers to a single memory unit that can be accessed by the architecture. The *LISA_declared_item* can represent LISA **GROUP**, **INSTANCE**, **REFERENCE**, or **LABEL**, which are declared in the **DECLARE** section. Although

M. Hohenauer, R. Leupers, *C Compilers for ASIPs*,
DOI 10.1007/978-1-4419-1176-6, © Springer Science+Business Media, LLC 2010

modes_statement ::= *regi_mode* | *immi_mode*

regi_mode ::= **_REGI** '(' *resource_expression* ')'
'<' *reg_offset0* '..' *reg_offset1* '>' ';'

immi_mode ::= **_IMMI** '(' *LISA_declared_item* ')' ';'

Fig. A.1 Semantics statement syntax

syntactically a LISA **LABEL** can be used as the destination, semantically this usage is wrong. A single label does not represent any architecture resource and cannot be assigned a value. If a LISA **GROUP**, **INSTANCE**, or **REFERENCE** is used as destination, then it must refer to an operation with a **SEMANTICS** section that encapsulates a legal processor resource. Considering the bit-widths of both sides in an assignment statement, they must be the same. Otherwise an error will be issued. Here are some more examples of assignment statements:

```
/* Rs1, Rs2, Dest are 32-bit registers declared in modes */
/* R16 is a 16-bit register */

_ADD(Rs1, Rs2) -> Dest;
/* Correct, _ADD returns a 32-bit result and Dest is 32-bit */

_MULUU(Rs1, R16) -> Dest;
/* Error!! since the MULUU returns a 48-bit result (32+16)
   and the Dest is only 32-bit long. */

_MULUU(Rs1, R16)<0..31> -> Dest;
/* Correct, since the bit-specifications are made. */
```

Listing A.1 Assignment statements and bit-width restrictions

The register mode defines (allocatable) register resources, where resource must be an item defined in the LISA resource section. The bit-width of the resource should be specified by the general bit-width specification.

A.1.1 IF-ELSE Statements

Control flow within the SEMANTICS section is modeled with IF–ELSE statements

Ten predefined comparison micro-operators are available (Table A.1). Each of these comparison operators returns either true or false, depending on the result. They can only be employed within if–else conditions.

Table A.1 Comparison keywords

Keyword	Comparison
_EQ	Equal
_NE	Not Equal
_GTI	Signed Greater Than
_GTU	Unsigned Greater Than
_GEI	Signed Greater Equal Than
_GEU	Unsigned Greater Equal Than
_LTI	Signed Less Than
_LTU	Unsigned Less Than
_LEI	Signed Less Equal Than
_LEU	Unsigned Less Equal Than

Their operands are basically the same as for any other micro-operations. Generally, operands can be one of:

- (constant) immediate values, e.g., `_EQ(rs1,0)`,
- LISA declared items that reference to the semantics of other operations via INSTANCE, GROUP, or REFERENCE (as holds true for any other section within a LISA operation),
- micro-operations, e.g., `_GTI(_SUB(rs1,rs2), 0)`,
- LISA resources, e.g., **_SP**.

if_else_statement ::= **IF** '(' *conditions* ')'
'{' *assignment_statement*+ '}'
[**ELSE** '{' *assignment_statement*+ '}']

conditions ::= *condition* (('||' | '&&') *condition*)*
| '(' *conditions* ')'

condition ::= *compare_operator*'('*compare_operand* ','
compare_operand
')'
| **_CF** | **_OF** | **_NF** | **_ZF**
| '!'**_CF** | '!'**_OF** | '!'**_NF** | '!'**_ZF**

compare_operator ::= **_EQ** | **_NE** | **_GTI** | **_GTU** | **_GEI** | **_GEU** | **_LTI** | **_LTU**
| **_LEI** | **_LEU**

compare_operand ::= *micro_operation_expression*
| *integer*
| *LISA_declared_item*
| *semantics_related_resources*

Fig. A.2 IF–ELSE statement syntax

The SEMANTICS section also provides shortcuts for the most common use cases:

```
IF(_ZF)  {...} <=> IF(_EQ(_ZF,1)){...}

IF(!_ZF) {...} <=> IF(_EQ(_ZF,0)){...}
```

Listing A.2 Shortcut examples

In the curly brackets after the condition is the statement that is executed conditionally. However, only assignment statement and nonassignment statement are allowed here. Putting an IF–ELSE statement in another IF–ELSE statement is currently not supported.

A.1.2 Nonassignment Statements

The syntax of the nonassignment statements is given in Fig. A.3.

non_assignment_statement ::= *micro_operator* ';'
| *micro_operation_expression* ';'
| *LISA_declared_item* ';'
| *semantics_related_resources* ';'
| *integer* ';'

Fig. A.3 Nonassignment statement syntax

There are totally five kinds of nonassignment statements, classified by the expression used as operand:

LISA declared item: The referenced operation must provide the semantics.

Micro operator: A single micro-operator with semicolon can also describe a statement, e.g., `_ADD;`. While this statement itself does not do any computation or data transfer, it is mainly used in operation hierarchy to indicate what kind of micro-operation the semantics section in the upper hierarchy has to carry out.

Micro-operation: Here the term micro-operation denotes a complete expression of micro-operations, which includes operator as well as operands, for instance, `_ADD(rs1, rs2);` Although the statement performs some computation, the destination is not defined here. This must be given somewhere in the upper hierarchy levels.

Accessible resources: Similar to the micro-operation, the accessible resources of the architecture represent some values residing in the architecture. So

the nonassignment statement can contain a resource to provide the upper hierarchy a data source, e.g., `_SP;`.

Constant: The nonassignment statement having a constant simply means that the operation provides other operations in the operation hierarchy a constant value as an operand.

A.1.3 Execution Timing

All the statements in one semantics section will be executed concurrently (rather than sequentially). Thus, one statement is executed using the processor state at the entry of this operation. All the assignments (i.e., modifying resources after computation) will be carried out at the end of this semantics section. For example:

```
SEMANTICS{
  _SUB(_ADD[_C](Rs1, Rs2), Rs3) -> Rd;
  _ADDC(Rd, Rs4) -> Mem1;
}
```

Listing A.3 Concurrent execution

This semantics section will carry out both lines concurrently, which means:

- **_SUB** and **_ADDC** micro-operations are executed in parallel. No timing information is available for **_ADD** (though obviously it should be scheduled before **_SUB**).
- **_ADD** may influence the carry flag as side effects. However, this will not effect **_ADDC** (**_ADDC** means addition with carry). In case of the first statement, the micro-operation **_ADD** will affect the carry flag, and consequently **_SUBC** will get the updated carry flag to calculate the result.
- **_ADDC** uses `Rd` as one operand. The contents of `Rd` will be updated after the semantics section has been executed. Hence **_ADDC** is expected to use the old value of `Rd` to do the computation.

A.2 Micro-Operators

The micro-operations provided in the list below are a basic set of operators as used for compiler generation. As stated in Chapter 6, the set of micro-operations is designed to be concise and compact. However, it might be necessary to extend the following set of micro-operations for other architectures. In particular, floating-point support is entirely left out.

First of all, the notations that are used in the following sections are introduced. Afterward, each micro-operator is described in an instruction-set manual-like

manner. For certain cases, detailed examples are provided. The micro-operators are grouped in terms of their functionalities. Side effects of the micro-operators are modeled as the affected flag declarations. They are explained later in this chapter as well as the general bit specifications.

A.2.1 Notations

Offset: Bit position indication. (The position starts from zero.)
Width: The width of the bit-extraction.
BITMASK(offset, width): Generates a bitmask where the bits starting from position *offset* with the *width* are filled with 1, and 0 in the remaining bits. `BITMASK(3,4) = 0b01111000`
BIT_EXTRACTIONS(value, offset, width): `(value) & BITMASK( offset, width )`
CF: Carry flag
ZF: Zero flag
OF: Overflow flag
NF: Negative flag
CF_SET: Returns 1 if the carry flag is set to be affected as a side effect. Otherwise 0.
ZF_SET: Returns 1 if the zero flag is set to be affected as a side effect. Otherwise 0.
OF_SET: Returns 1 if the overflow flag is set to be affected as a side effect. Otherwise 0.
NF_SET: Returns 1 if the negative flag is set to be affected as a side effect. Otherwise 0.
operand*n*: Operands of the micro-operators. Each operand has three components: value, offset, and width. *Value* represents the actual content of the operand. *Offset* and *Width* indicate a bit-extraction process. The final result of the operand will be the extracted bits from the value. In the *Operation* of each micro-operator's description, the index *n* is used to seperate different components, e.g., operand1 is composed of value1, width1, and offset1.
ISSUE_ERROR(_MISMATCH): Mismatch error of the operands' bit-width is thrown.
ZF_SIDE_EFFECT(result): If the result is zero, returns 1. Otherwise returns 0.
NF_SIDE_EFFECT(result, width): If bit[width -1] is 1 (negative value), returns 1. Otherwise returns 0.
OF_SIDE_EFFECT_ADD(op1, op2, width): Returns 1 if the addition specified as its parameter causes a (width)-bit signed oveflow. Addition generates an overflow if both operands have the same sign (bit[width -1]), and the sign of the result is different from the sign of both operands.
CF_SIDE_EFFECT_ADD(op1, op2, width): Returns 1 if the addition specified as its parameter causes a carry (true result is bigger than $2^{\text{width}} - 1$, where the operands are treated as unsigned integers), and returns 0 in all other cases.
OF_SIDE_EFFECT_SUB(op1, op2, width): Returns 1 if the subtraction specified as

its parameter causes a (width)-bit signed oveflow. Subtraction causes an overflow if the operands have different signs, and the first operand and the result have different signs.
CF_SIDE_EFFECT_SUB(op1, op2, width): Returns 0 if the subtraction specified as its parameter causes a borrow (the true result is less than 0, where the operands are treated as unsigned integers), and returns 1 in all other cases.
OF_SIDE_EFFECT_MULUU(op1, op2, width1, width2, width): If the multiplication result of the unsigned number op1 (width1) and the unsigned number op2 (width2) exceeds the unsigned range that width bits can take, returns 1. Otherwise returns 0.
OF_SIDE_EFFECT_MULIU(op1, op2, width1, width2, width): If the multiplication result of the signed number op1 (width1) and the unsigned number op2 (width2) exceeds the signed range that width bits can take, returns 1. Otherwise returns 0.
OF_SIDE_EFFECT_MULII(op1, op2, width1, width2, width): If the multiplication result of the signed number op1 (width1) and the signed number op2 (width2) exceeds the signed range that width bits can take, returns 1. Otherwise returns 0.
OF_SIDE_EFFECT_NEG(result, width): The only case of overflow for negative micro-operation happens when the maximum negative value is taken as an operand. For example, for 4-bit signed values, the max negative value is 0b1000 (-8). Taking the negative value of this one gives 0b1000, which is incorrect because the max positive value is $+7$. Returns 1 if this case happens, otherwise returns 0.

A.2.2 Group of Arithmetic Operators

This group of micro-operators deals with the arithmetic instructions that appear in most of the processor architectures. Some of the micro-operators need to work with flags, reading flags, and/or writing flags as side effects.

```
<arithmetic_uop> := _ADD | _ADDC | _SUB | _SUBC | _NEG
                  | _MULUU | _MULIU | _MULII
```

A.2.2.1 _ADD

Description	Adds two operands.
Syntax	`_ADD[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions	Two operands must be of the same bit-width.
Result bit-width	Same as that of operands.
Affected flags	`_CF, _ZF, _NF, _OF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1, offset1, width1) >> offset1;
```

```
  temp2 = BIT_EXTRACTIONS(value2, offset2, width2) >> offset2;
  result = BIT_EXTRACTIONS((temp1+temp2),offset,width)>>offset;
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result);}
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  if (OF_SET) { OF = OF_SIDE_EFFECT_ADD(temp1, temp2, width1);}
  if (CF_SET) { CF = CF_SIDE_EFFECT_ADD(temp1, temp2, width1);}
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_ADD[_C,_Z,_N,_O](0x00100010<0..31>, 0x00010001<0..31>)<0..31>
 -> 110011 (ZF:0 NF:0 OF:0 CF:0)
_ADD[_C,_Z,_N,_O](0x00100010<0..15>, 0x00010001<0..15>)<0..15>
 -> 11 (ZF:0 NF:0 OF:0 CF:0)
_ADD[_C,_Z,_N,_O](0x00108010<0..15>, 0x00010001<0..15>)<0..15>
 -> 8011 (ZF:0 NF:1 OF:0 CF:0)
_ADD[_C,_Z,_N,_O](0x00100001<0..15>, 0x0000ffff<0..15>)<0..15>
 -> 0 (ZF:1 NF:0 OF:0 CF:1)
```

A.2.2.2 _ADDC

Description	Adds two operands with carry.
Syntax	`_ADDC[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions	Two operands must be of the same bit-width.
Result bit-width	Same as that of operands.
Affected flags	`_CF, _ZF, _NF, _OF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
  temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
  result = BIT_EXTRACTIONS((temp1+temp2+CF),
                           offset,width)>>offset;
  if (OF_SET) { OF = OF_SIDE_EFFECT_ADD(temp1 + CF,
                                        temp2, width1);}
  if (CF_SET) { CF = CF_SIDE_EFFECT_ADD(temp1 + CF,
                                        temp2, width1);}
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}
```

```
Examples:
(ZF:0 NF:0 OF:0 CF:1)
_ADDC[_C,_Z,_N,_O](0x00100010<0..31>, 0x00010001<0..31>)<0..31>
 -> 110012 (ZF:0 NF:0 OF:0 CF:0)
_ADDC[_C,_Z,_N,_O](0x00100010<0..15>, 0x00010001<0..15>)<0..15>
 -> 11 (ZF:0 NF:0 OF:0 CF:0)
_ADDC[_C,_Z,_N,_O](0x00108010<0..15>, 0x00010001<0..15>)<0..15>
 ->  8011 (ZF:0 NF:1 OF:0 CF:0)
_ADDC[_C,_Z,_N,_O](0x00100001<0..15>, 0x0000ffff<0..15>)<0..15>
 -> 0 (ZF:1 NF:0 OF:0 CF:1)
```

A.2.2.3 _SUB

Description	Subtracts the operand2 from operand1.
Syntax	`_SUB[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions	Two operands must be of the same bit-width.
Result bit-width	Same as that of operands.
Affected flags	`_CF, _ZF, _NF, _OF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1,offset1,width1) >> offset1;
  temp2 = BIT_EXTRACTIONS(value2,offset2,width2) >> offset2;
  result = BIT_EXTRACTIONS((temp1-temp2),
                           offset,width) >> offset;
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  if (OF_SET) { OF = OF_SIDE_EFFECT_SUB(temp1,temp2,width1);}
  if (CF_SET) { CF = CF_SIDE_EFFECT_SUB(temp1,temp2,width1);}
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_SUB[_C,_Z,_N,_O](0x00100010<0..31>,0x00010001<0..31>)<0..31>
 -> f000f (ZF:0 NF:0 OF:0 CF:1)
_SUB[_C,_Z,_N,_O](0x00100010<0..15>,0x00010001<0..15>)<0..15>
 -> f (ZF:0 NF:0 OF:0 CF:1)
_SUB[_C,_Z,_N,_O](0x00108010<0..15>,0x00010001<0..15>)<0..15>
 -> 800f (ZF:0 NF:1 OF:0 CF:1)
_SUB[_C,_Z,_N,_O](0x00100001<0..15>,0x0000ffff<0..15>)<0..15>
 -> 2 (ZF:0 NF:0 OF:0 CF:0)
_SUB[_C,_Z,_N,_O](0x00100001<0..31>,0x0000ffff<0..31>)<0..31>
 -> f0002 (ZF:0 NF:0 OF:0 CF:1)
```

A.2.2.4 _SUBC

Description Subtracts the operand2 from operand1 with carry.
Syntax `_SUBC[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions Two operands must be of the same bit-width.
Result bit-width Same as that of operands.
Affected flags `_CF, _ZF, _NF, _OF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
  temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
  // temp1 - temp2 - NOT(CF)
  result = BIT_EXTRACTIONS((temp1 - temp2 - NOT(CF) ),
                           offset, width) >> offset;
  if (OF_SET) { OF = OF_SIDE_EFFECT_SUB(temp1 -
                                  NOT(CF), temp2, width1); }
  if (CF_SET) { CF = CF_SIDE_EFFECT_SUB(temp1 -
                                  NOT(CF), temp2, width1); }
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_SUBC[_C,_Z,_N,_O](0x00100010<0..31>, 0x00010001<0..31>)<0..31>
 -> f000e (ZF:0 NF:0 OF:0 CF:1)
```

A.2.2.5 _MULUU

Description Multiplies the unsigned integer operand1 by the unsigned integer operand2.
Syntax `_MULUU[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions No restrictions on the bit-widths of operands.
Result bit-width The addition of the bit-widths of the operands.
Affected flags `_ZF, _OF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
result = BIT_EXTRACTIONS(((unsigned)temp1*(unsigned)temp2),
                           offset, width) >> offset;
```

```
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (OF_SET) { OF = OF_SIDE_EFFECT_MULUU(temp1,temp2,width1,
                                        width2, width);}
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_MULUU[_Z,_O](0x00100010<0..31>, 0x00010001<0..31>)<0..31>
 -> 200010 (ZF:0 NF:0 OF:1 CF:0)
```

A.2.2.6 _MULIU

Description Multiplies the signed integer operand1 by the unsigned integer operand2.
Syntax `_MULIU[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions No restrictions on the bit-widths of operands.
Result bit-width The addition of the bit-widths of the operands.
Affected flags `_ZF, _NF, _OF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
// check if op1 is negative
// if so, sign extend to 32 bit long
// if long is used, then replace 32 by 64
temp1 = SEM_SXT(temp1, 0, width1, 0, 32);
result = BIT_EXTRACTIONS(((signed)temp1*(unsigned)temp2),
                         offset, width) >> offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
if (OF_SET) { OF = OF_SIDE_EFFECT_MULIU(temp1,temp2,width1,
                                        width2, width); }
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_MULIU[_Z,_N,_O](0x8000<0..15>, 0x0010<0..15>)<0..31>
 -> fff80000 (ZF:0 NF:1 OF:0 CF:0)
```

A.2.2.7 _MULII

Description Multiplies the signed integer operand1 by the signed integer operand2.
Syntax `_MULII[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions No restrictions on the bit-widths of operands.
Result bit-width The addition of the bit-widths of the operands.
Affected flags `_ZF, _NF, _OF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
// check if op1 and op2 are negative
// if so, sign extends to 32 bit long
// if long is used, then replace 32 by 64
temp1 = SEM_SXT(temp1, 0, width1, 0, 32);
temp2 = SEM_SXT(temp2, 0, width2, 0, 32);
result = BIT_EXTRACTIONS(((signed)temp1*(signed)temp2),
                         offset, width) >> offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
if (OF_SET) { OF = OF_SIDE_EFFECT_MULII(temp1,temp2,width1,
                                        width2, width); }
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_MULII[_Z, _N, _O](0x8000<0..15>, 0x8010<0..15>)<0..31>
 -> 3ff80000 (ZF:0 NF:0 OF:0 CF:0)
_MULII[_Z, _N, _O](0x8000<0..15>, 0x8010<0..15>)<0..23>
 -> f80000 (ZF:0 NF:1 OF:1 CF:0)
```

A.2.2.8 _NEG

Description	Produces the negative value of the operand (twos-complement).
Syntax	`_NEG[affected_flag_declarations](operand1)[bit_extractions]`
Restrictions	No restrictions.
Result bit-width	Same as that of the operand.
Affected flags	`_ZF, _NF, _OF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1, offset1, width1) >> offset1;
result = BIT_EXTRACTIONS((-((signed)temp1)),offset,width)
                         >>offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result,width); }
if (OF_SET) { OF = OF_SIDE_EFFECT_NEG(temp1,width1); }
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_NEG[_Z, _N, _O](0x10<0..31>)<0..31>
 -> fffffff0 (ZF:0 NF:1 OF:0 CF:0)
```

A.2.3 Group of Logic Operators

This group of micro-operators deals with the bitwise logic functions. Similar to the arithmetic group, the operators can change the flags as a side effect.

```
<logic_uop> := _AND | _OR | _XOR | _NOT
```

A.2.3.1 _AND

Description	Performs a bitwise AND operation on operand1 and operand2.
Syntax	`_AND[affected_flag_declarations](operand1, operand2) [bit_extractions]`
Restrictions	Two operands must be of the same bit-width.
Result bit-width	Same as that of operands.
Affected flags	`_ZF, _NF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
  temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
  result = BIT_EXTRACTIONS((temp1 & temp2),offset,width)
                          >> offset;
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_AND[_N, _Z](0x0fff0fff<0..31>, 0x000f000f<0..31> 0, 32)
 -> f000f (ZF:0 NF:0 OF:0 CF:0)
_AND[_N, _Z](0x0ff00fff<0..31>, 0x000f000f<0..31> 0, 32)
 -> f  (ZF:0 NF:0 OF:0 CF:0)
_AND[_N, _Z](0xfff00fff<0..31>, 0x000f000f<0..31> 0, 32)
 -> f  (ZF:0 NF:0 OF:0 CF:0)
_AND[_N, _Z](0xfff00fff<0..31>, 0x800f000f<0..31> 0, 32)
 -> 8000000f (ZF:0 NF:1 OF:0 CF:0)
```

A.2.3.2 _OR

Description	Performs a bitwise OR operation on operand1 and operand2.
Syntax	`_OR[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions	Two operands must be of the same bit-width.
Result bit-width	Same as that of operands.
Affected flags	`_ZF, _NF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
  temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
  result = BIT_EXTRACTIONS((temp1 | temp2),offset,width)
                          >>offset;
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_OR[_N,_Z](0x0fff0fff<0..31>, 0x000f000f<0..31>)<0..31>)
 -> fff0fff (ZF:0 NF:0 OF:0 CF:0)
_OR[_N,_Z](0x0ff00fff<0..31>, 0x000f000f<0..31>)<0..31>)
 -> fff0fff (ZF:0 NF:0 OF:0 CF:0)
_OR[_N,_Z](0xfff00fff<0..31>, 0x000f000f<0..31>)<0..31>)
 -> ffff0fff (ZF:0 NF:1 OF:0 CF:0)
_OR[_N,_Z](0xfff00fff<0..31>, 0x800f000f<0..31>)<0..31>)
 -> ffff0fff (ZF:0 NF:1 OF:0 CF:0)
```

A.2.3.3 _XOR

Description	Performs a bitwise XOR operation on operand1 and operand2.
Syntax	`_XOR[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions	Two operands must be of the same bit-width.
Result bit-width	Same as that of operands.
Affected flags	`_ZF, _NF`

Operation

```
if (width1 == width2) {
  temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
  temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
  result = BIT_EXTRACTIONS((temp1 ^ temp2),offset,width)
                          >>offset;
  if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
  if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
  return result;
}
else {
  ISSUE_ERROR(_MISMATCH);
}
```

```
Examples:
(ZF:0 NF:0 OF:0 CF:0)
_XOR[_N,_Z](0x0fff0fff<0..31>, 0x000f000f<0..31>)<0..31>
 -> ff00ff0 (ZF:0 NF:0 OF:0 CF:0)
_XOR[_N,_Z](0x0ff00fff<0..31>, 0x000f000f<0..31>)<0..31>
 -> fff0ff0 (ZF:0 NF:0 OF:0 CF:0)
_XOR[_N,_Z](0xfff00fff<0..31>, 0x000f000f<0..31>)<0..31>
 -> ffff0ff0 (ZF:0 NF:1 OF:0 CF:0)
_XOR[_N,_Z](0xfff00fff<0..31>, 0x800f000f<0..31>)<0..31>
 -> 7fff0ff0 (ZF:0 NF:0 OF:0 CF:0)
```

A.2.3.4 _NOT

Description	Performs a bitwise NOT operation on operand.
Syntax	`_NOT[affected_flag_declarations](operand1)[bit_extractions]`
Restrictions	No restrictions.
Result bit-width	Same as that of the operand.
Affected flags	`_ZF, _NF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
result = BIT_EXTRACTIONS(~temp1,offset,width)>>offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
return result;
```

```
Examples:
(ZF:0 NF:0 OF:0 CF:0)
_NOT[_Z, _N](0x0fff0fff<0..31>)<0..31>
 -> f000f000 (ZF:0 NF:1 OF:0 CF:0)
_NOT[_Z, _N](0x0ff00fff<0..31>)<0..31>
 -> f00ff000 (ZF:0 NF:1 OF:0 CF:0)
_NOT[_Z, _N](0xfff00fff<0..31>)<0..31>
 -> ff000 (ZF:0 NF:0 OF:0 CF:0)
_NOT[_Z, _N](0xfff00fff<0..31>)<0..31>
 -> ff000 (ZF:0 NF:0 OF:0 CF:0)
```

A.2.4 Group of Shifting Operators

This group of micro-operators deals with the shifting functionality. Again, the micro-operators may affect the flags (mainly carry flag).

```
<shifting_uop> := _LSL | _LSR | _ASR | _ROTL | _ROTR
```

A.2.4.1 _LSL

Description Performs a logical left-shift operation on operand1 by operand2 bits. The additional bits in dst are filled with zeros. The information in the operand2 leftmost bits is discarded if the user does not specify the affected flags. Otherwise some flags (e.g., carry flag) is changed.

Syntax `_LSL[affected_flag_declarations](operand1, operand2)[bit_extractions]`

Restrictions No restrictions.

Result bit-width Same as that of operand1.

Affected flags `_CF, _ZF, _NF: if carry flag is specified in affected flags, it is assumed that carry flag stores the last-moved bit from the source. Zero and negative flag apply to the whole value that is moved into destination.`

Operation

```
temp1 = (unsigned)value1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
if (width1 <= ((unsigned)temp2 - 1) ) {
  cerr<< "Warning: left shift count >= width of type "<<endl;
}
result = BIT_EXTRACTIONS(temp1<<(unsigned(temp2)),offset,width);
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
if (CF_SET) {
  if (temp1 & (0x1 << (width1 - ((unsigned)temp2) ) ) )
  { CF = 1; }
  else { CF = 0; }
}
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_LSL[_C,_Z,_N](0x00ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00ff00 (ZF:0 NF:1 OF:0 CF:0)
_LSL[_C,_Z,_N](0x01ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00ff00 (ZF:0 NF:1 OF:0 CF:1)
_LSL[_C,_Z,_N](0x00ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00ff00 (ZF:0 NF:1 OF:0 CF:0)
_LSL[_C,_Z,_N](0x00ff00ff<0..31>, 0x10<0..31>)<0..31>
 -> ff0000 (ZF:0 NF:0 OF:0 CF:1)
```

A.2.4.2 _LSR

Description Performs a logical right shift on operand1 by operand2 bits. The new operand2 bits to the left are filled with zeros. The information in the operand2 rightmost bits is discarded if the user does not specify the affected flags. Otherwise some flags (e.g., carry flag) is changed.

Syntax `_LSR[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions No restrictions.
Result bit-width Same as that of operand1.
Affected flags `_CF, _ZF, _NF: if carry flag is specified in affected flags, it is assumed that carry flag stores the last-moved bit from the source. Zero and negative flag apply to the whole value that is moved into destination.`

Operation

```
temp1 = (unsigned)value1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
result = BIT_EXTRACTIONS(temp1>>(unsigned(temp2)),offset,width);
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
if (CF_SET) {
  if ((temp1 >> (unsigned(temp2) - 1)) & (0x1))
  { CF = 1; }
  else { CF = 0; }
}
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_LSR[_C,_Z,_N](0x00ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00 (ZF:0 NF:0 OF:0 CF:1)
_LSR[_C,_Z,_N](0x01ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> 1ff00 (ZF:0 NF:0 OF:0 CF:1)
_LSR[_C,_Z,_N](0x00ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00 (ZF:0 NF:0 OF:0 CF:1)
_LSR[_C,_Z,_N](0x00ff00ff<0..31>, 0x10<0..31>)<0..31>
 -> ff (ZF:0 NF:0 OF:0 CF:0)
```

A.2.4.3 _ASR

Description Performs an arithmetic right shift on operand1 by operand2 bits. The new operand2 bits to the left are filled with zeros or ones depending on the leftmost bit before the shift operation. The information in the operand2 rightmost is discarded if the user does not specify the affected flags. Otherwise some flags (e.g., carry flag) is changed.
Syntax `_ASR[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions No restrictions.
Result bit-width Same as that of operand1.
Affected flags `_CF, _ZF, _NF: if carry flag is specified in affected flags, it is assumed that carry flag stores the last-moved bit from the source. Zero and negative flag apply to the whole value that is moved into destination.`

Operation

```
temp1 = (unsigned)value1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
if ((temp1 & (0x1 << (width1 - 1)) ) == 0 )  {
  result = (temp1 >> ((unsigned)temp2) );
}
else {
  result = (temp1 >> ((unsigned)temp2) ) | BITMASK(
           (width1 - (unsigned)temp2), (unsigned)temp2);
}
result = BIT_EXTRACTIONS(result, offset, width) >> offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
if (CF_SET) {
  if ((temp1 >> (unsigned(temp2) - 1)) & (0x1)) { CF = 1; }
  else { CF = 0; }
}
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_ASR[_C, _Z, _N](0x00ff00ff<0..23>, 0x8<0..31>)<0..23>
 -> ffff00 (ZF:0 NF:1 OF:0 CF:1)
_ASR[_C, _Z, _N](0x01ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> 1ff00 (ZF:0 NF:0 OF:0 CF:1)
_ASR[_C, _Z, _N](0x80ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff80ff00 (ZF:0 NF:1 OF:0 CF:1)
_ASR[_C, _Z, _N](0x80ff00ff<0..31>, 0x10<0..31>)<0..31>
 -> ffff80ff (ZF:0 NF:1 OF:0 CF:0)
```

A.2.4.4 _ROTL

Description Rotational left shift on operand1 by operand2 bits. If the user specify some flags as side effects (e.g., carry flag), the carry flag is used as a buffer to do the shifting.

Syntax `_ROTL[affected_flag_declarations](operand1, operand2)[bit_extractions]`

Restrictions No restrictions.

Result bit-width Same as that of operand1.

Affected flags `_CF, _ZF, _NF: if carry flag is specified in affected flags, it is assumed that carry flag stores the last-moved bit from the source. Zero and negative flag apply to the whole value that is moved into destination.`

Operation

```
temp1 = (unsigned)value1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
```

```
result = temp1;
if (CF_SET) {
  if (temp1 & (0x1 << (width1 - ((unsigned)temp2) ) ) )
  { CF = 1; }
  else { CF = 0; }
}
for (u32 i = 0; i < ((unsigned)temp2); i++) {
  if (!(temp1 & (0x1 << (width1 - 1)) ))   {
    result = result << 1;
  }
  else {
    result = (result << 1) | 1;
  }
  temp1 = temp1 << 1;
}
result = BIT_EXTRACTIONS(result, offset, width)>>offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_ROTL[_C,_Z,_N](0x00ff00ff<0..23>, 0x8<0..31>)<0..23>
 -> ffff (ZF:0 NF:0 OF:0 CF:1)
_ROTL[_C,_Z,_N](0x01ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00ff01 (ZF:0 NF:1 OF:0 CF:1)
_ROTL[_C,_Z,_N](0x80ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff00ff80 (ZF:0 NF:1 OF:0 CF:0)
_ROTL[_C,_Z,_N](0x80ff00ff<0..31>, 0x10<0..31>)<0..31>
 -> ff80ff (ZF:0 NF:0 OF:0 CF:1)
```

A.2.4.5 _ROTR

Description	Rotational right shift on operand1 by operand2 bits. If the user specify some flags as side effects (e.g., carry flag), the carry flag is used as a buffer to do the shifting.
Syntax	`_ROTR[affected_flag_declarations](operand1, operand2)[bit_extractions]`
Restrictions	No restrictions.
Result bit-width	Same as that of operand1.
Affected flags	`_CF, _ZF, _NF: if carry flag is specified in affected flags, it is assumed that carry flag stores the last-moved bit from the source. Zero and negative flag apply to the whole value that is moved into destination.`

Operation

```
temp1 = (unsigned)value1;
temp2 = BIT_EXTRACTIONS(value2,offset2,width2)>>offset2;
result = temp1;
```

```
if (CF_SET) {
  if ((temp1 >> (unsigned(temp2) - 1)) & (0x1))
  { CF = 1; }
  else { CF = 0; }
}
for (u32 i = 0; i < ((unsigned)temp2); i++) {
  if ( !(temp1 & 1) ) {
    result = result >> 1;
  }
  else {
    result = (result >> 1) | (0x1 << (width1 - 1));
  }
  temp1 = temp1 >> 1;
}
result = BIT_EXTRACTIONS(result, offset, width)>>offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_ROTR[_C, _Z, _N](0x00ff00ff<0..23>, 0x8<0..31>)<0..23>
 -> ffff00 (ZF:0 NF:1 OF:0 CF:1)
_ROTR[_C, _Z, _N](0x01ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff01ff00 (ZF:0 NF:1 OF:0 CF:1)
_ROTR[_C, _Z, _N](0x80ff00ff<0..31>, 0x8<0..31>)<0..31>
 -> ff80ff00 (ZF:0 NF:1 OF:0 CF:1)
_ROTR[_C, _Z, _N](0x80ff00ff<0..31>, 0x10<0..31>)<0..31>
 -> ff80ff (ZF:0 NF:0 OF:0 CF:0)
```

A.2.5 Group of Zero/Sign Extension Operators

This group of operators serve the purpose of zero/sign extensions. They do not have any effect on the flags.

```
<extension_uop> := _SXT | _ZXT
```

A.2.5.1 _SXT

Description	Performs a sign extension to the operand.
Syntax	`_SXT(operand1)[bit_extractions]`
Restrictions	No restrictions.
Result bit-width	The result bitwidth is determined by the bit-specs that follows the micro-operator.
Affected flags	`_ZF, _NF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
```

```
if ( width <= width1) {
  cerr<<"Wrn: You are using a sign reduction in _SXT."<<endl;
  cerr<<"Better directly use the bit specs." << endl;
}
if ( !(temp1 & (0x1 << (width1 - 1))) ) {
  // MSB is 0
  result = BIT_EXTRACTIONS(temp1,offset,width)>>offset;
}
else {
  // MSB is 1
  result = temp1 | BITMASK(width1, width - width1 );
  result = BIT_EXTRACTIONS(result, offset, width) >> offset;
}
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
if (NF_SET) { NF = NF_SIDE_EFFECT(result, width); }
}
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_SXT[_Z, _N](0xff00<0..15>)<0..31>:
 -> ffffff00 (ZF:0 NF:1 OF:0 CF:0)
_SXT[_Z, _N](0x7f00<0..15>)<0..31>
 -> 7f00 (ZF:0 NF:0 OF:0 CF:0)
_SXT[_Z, _N](0xff00<0..15>)<0..23>
 -> ffff00 (ZF:0 NF:1 OF:0 CF:0)
```

A.2.5.2 _ZXT

Description Performs a zero extension to the operand.
Syntax `_ZXT(operand1)[bit_extractions]`
Restrictions No restrictions.
Result bit-width The result bitwidth is determined by the bit-specs that follows the micro-operator.
Affected flags `_ZF`

Operation

```
temp1 = BIT_EXTRACTIONS(value1,offset1,width1)>>offset1;
if ( width <= width1) {
  cerr<<"Wrn:You are using a sign reduction _ZXT."<<endl;
  cerr<<"Better directly use the bit specs." << endl;
}
result = BIT_EXTRACTIONS(temp1, offset, width) >> offset;
if (ZF_SET) { ZF = ZF_SIDE_EFFECT(result); }
return result;

Examples:
(ZF:0 NF:0 OF:0 CF:0)
_ZXT[_Z](0xff00<0..15>)<0..31>
 -> ff00 (ZF:0 NF:0 OF:0 CF:0)
```

```
_ZXT[_Z](0x7f00<0..15>)<0..31>
 -> 7f00 (ZF:0 NF:0 OF:0 CF:0)
_ZXT[_Z](0xff00<0..16>)<0..23>
 _> ff00 (ZF:0 NF:0 OF:0 CF:0)
```

NOTE: There comes some suggestions about writing the sign/zero extension/reduction in the semantics section. If the user wants to do sign/zero extension which means to expand the bit-width of the operand considering the sign bit, it should be read, e.g.,

```
_SXT(_ADD(Rs1, Rs2)<0..7>)<0..15> -> Dest;
/* Dest is 16 bit long. */
```

This tells that the lower 8 bits of the addition result will be sign-extended to 16 bits and later transferred to destination register (which must be 16 bit, otherwise errors are issued). Or it can be transferred to the arbitrary bit locations of the destination registers as long as it makes sense, e.g.,

`_SXT(_ADD(Rs1, Rs2)<0..7>)<0..15> -> Dest<16..31>;`

It is assumed that the micro-operators _SXT and _ZXT will extend the operands to infinite long and the truncations will be carried out by bit-width specifications, say, to 16 bits. The other case, reduction, happens in ST220 model. Sign/zero reductions simply mean to extract the lower bits down. The user may write something like

```
_SXT(_ADD(Rs1,Rs2)<0..15>)<0..7> -> Dest;
/* Dest is 8 bit long. */
```

but that is equivalent to

`_ADD(Rs1, Rs2)<0..7> -> Dest; /* Dest is 8 bit long. */.`

It is recommended that the user follow the latter expression. Warnings may be issued in this case.

A.2.6 Others/Intrinsic Operators

All the micro-operations that cannot be appropriately grouped in the above and the intrinsic operations are listed here.

```
<other_uop> := _INDIR | _NOP | <intrinsic_uop>
```

A.2.6.1 _INDIR

Description	References a specific memory location pointed by operand. Can be used with operation chaining for load and store operations, or any other instructions that can use one or more memory operands.
Syntax	`_INDIR(_OR(Rs,_SP))<Offset1..Offset1+Bits>;`
Restrictions	None.
Result bit-width	The bit-width of the result is determined by the bit-specs that follow the micro-operator. Please refer the details below.
Affected flags	`None.`

A.2.6.2 _NOP

Description	Do nothing.
Syntax	`_NOP;`
Restrictions	None.
Result bit-width	None.
Affected flags	`None.`

A.2.6.3 <intrinsic_op>

Description	User-defined architecture-specific operations.
Syntax	`''FFS'';`
Restrictions	User-defined.
Result bit-width	User-defined.
Affected flags	`User-defined for compiler knowledge.`

NOTE: More about the _INDIR formalizations and parameters follows:

```
_INDIR(Addr, Endianess = _LITTLE,
       char *AddressNameSpace)<x..y>;
```

The _INDIR can take up to three parameters for accessing the memory. The Addr is the location of the memory unit that the user wants. The Endianess indicates which data organization/fashion this micro-operation _INDIR should follow. The address_space is suitable in the case of multiple addressing spaces. The bit-specification is used, e.g., loading a word from a byte-wise memory. Examples:

```
_INDIR(0x0, _LITTLE, ''DataMem'')<0..31> -> Dest;
```

This operation will fill up the 32-bit destination register with the memory contents (memory address space 1) {0x3}{0x2}{0x1}{0x0} provided that the base memory is byte-wise.

```
_INDIR(0x0, _BIG)<0..31> -> Dest;
```

This operation will fill up the 32-bit destination register with the memory contents (default memory) {0x0}{0x1}{0x2}{0x3} provided that the base memory is byte-wise.

If there is *only* one bus in the LISA model, the AddressNameSpace can be omitted. Also it is considered that the case

```
_INDIR(0x0, _LITTLE)<0..23> -> Dest;
```

also holds because the bits can be simply counted when filling up the destination.

A.2.7 Affected Flag Declarations

Definitions of the flags

- Carry flag: Set by a carry out/borrow in at MSB
- Zero flag: Set if entire byte, word, or long == 0
- Negative flag: Set if sign bit == 1
- Overflow flag: Set by a carry into sign bit w/o a carry out

```
<affected_flag_declarations>:='['<flag>{','<flag>}']'
<flag> := _C | _Z | _N | _O
```

The affected flag declaration is very important to portrait the side effects of the instructions that occur in most processors. Here side effects are defined as the posteffects of the instructions, i.e., the flags are changed due to the result of this instruction. (In contrast, the common addition with carry is handled by different micro-operations.) Currently, there are four flags that are explicitly supported in this semantical description: carry flag, zero flag, negative flag, and overflow flag. For example:

```
_ADD[_C, _Z](Rs1, Rs2) -> Rd;
```

This is interpreted as: use the predefined micro-operation _ADD to perform addition of the two operands and stores the result into Rd. Set zero flag if the result is zero; otherwise cleared. Set carry flag if a carry is generated; otherwise cleared.

NOTE: if the user does not give the affected flag declarations, no flags will be changed after the operation.

A.3 SEMANTICS Section Grammar

A.3.1 Grammar Notation

The keywords are denoted by using bolded font, e.g., "**KEYWORD**".

The nonterminal symbols are typeset slanted, e.g., "*nonterminal*".

If the syntax definition contains special caracters, they will be quoted with single quotes, e.g., '}'.

Concatenation of two components is denoted by putting the components in sequence, e.g.,
concatenation ::= *element1 element2*

Optional components are denoted by surrounding square brackets, e.g.,
optional ::= [*element*]

Repeating a component zero or more times is denoted with a asterisk, e.g.,
repeat ::= *element**

Repeating a component one or more times is denoted with a plus, e.g.,
repeat ::= *element+*

Alternative components are denoted by vertical bars, e.g.,
alternative ::= *option1* | *option2* | *option3*

Brackets are used to group several elements, e.g.,
elements ::= (*element1 element2*)

Several elements separated with comma can use the same definition, e.g.,
element1, *element2* ::= *definition*

A.3.2 SEMANTICS Grammar

A.3.2.1 Global Structure

semantics_section ::= **SEMANTICS** '{' *semantic_statement+* '}'

A.3.2.2 Semantic Statements

semantic_statement ::= *assignment_statement*
| *if_else_statement*
| *modes_statement*
| *non_assignment_statement*

assignment_statement ::= *source_expression* '->' *destination_expression* ';'

source_expression ::= *micro_operation_expression*
| *integer*
| *LISA_declared_item*
| *semantics_related_resources*

destination_expression ::= *LISA_declared_item*
| *indir_expression*
| *semantics_related_resources*

modes_statement ::= *regi_mode* | *immi_mode*

regi_mode ::= **_REGI** '(' *resource_expression* ')'
'<' *reg_offset0* '..' *reg_offset1* '>' ';'

resource_expression ::= *LISA_declared_item* ('[' *LISA_declared_item* ']')*

reg_offset0, *reg_offset1* ::= *integer*

immi_mode ::= **_IMMI** '(' *LISA_declared_item* ')' ';'

if_else_statement ::= **IF** '(' *conditions* ')'
'{' *assignment_statement*+ '}'
[**ELSE** '{' *assignment_statement*+ '}']

conditions ::= *condition* (('||' | '&&') *condition*)*
| '(' *conditions* ')'

condition ::= *equal* | *not_equal* | *signed_greater* | *unsigned_greater*
| *signed_greater_equal* | *unsigned_greater_equal*
| *signed_less* | *unsigned_less*
| *signed_less_equal* | *unsigned_less_equal*
| **_CF** | **_OF** | **_NF** | **_ZF**

equal ::= **_EQ** '(' *compare_operand* ',' *compare_operand* ')'

not_equal ::= **_NE** '(' *compare_operand* ',' *compare_operand* ')'

signed_greater ::= **_GTI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_greater ::= **_GTU** '(' *compare_operand* ',' *compare_operand* ')'

signed_greater_equal ::= **_GEI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_greater_equal ::= **_GEU** '(' *compare_operand* ','
compare_operand ')'

signed_less ::= **_LTI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_less ::= **_LTU** '(' *compare_operand* ',' *compare_operand* ')'

signed_less_equal ::= **_LEI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_less_equal ::= **_LEU** '(' *compare_operand* ',' *compare_operand* ')'

compare_operand ::= *micro_operation_expression*
| *integer*
| *LISA_declared_item*
| *semantics_related_resources*

non_assignment_statement ::= *micro_operator* ';'
| *micro_operation_expression* ';'
| *LISA_declared_item* ';'
| *semantics_related_resources* ';'
| *integer* ';'

micro_operator ::= **_ADD** | **_ADDC** | **_SUB** | **_SUBC** | **_MULII** | **_MULIU**
| **_MULUU** | **_AND** | **_OR** | **_XOR** | **_NOT** | **_NEG** | **_LSL**
| **_LSR** | **_ROTL** | **_ROTR** | **_ASR** | **_ZXT** | **_SXT**
| ""*intrinsic_name*""

A.3.2.3 Micro-Operation Expressions

micro_operation_expression ::= *add_expression* | *addc_expression* | *sub_expression*
| *subc_expression* | *mulii_expression*
| *muluu_expression* | *and_expression* | *or_expression*
| *xor_expression* | *not_expression* | *neg_expression*
| *lsl_expression* | *lsr_expression* | *rotl_expression*
| *rotr_expression* | *asr_expression* | *zxt_expression*
| *sxt_expression* | *indir_expression*
| *intrinsic_expression* | *hierarchy_expression*
| *muliu_expression*

add_expression ::= **_ADD** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

addc_expression ::= **_ADDC** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

sub_expression ::= **_SUBC** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

mulii_expression ::= **_MULII** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

muliu_expression ::= **_MULIU** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

muluu_expression ::= **_MULUU** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

and_expression ::= **_AND** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

or_expression ::= **_OR** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

lsl_expression ::= **_LSL** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

lsr_expression ::= **_LSR** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

rotl_expression ::= **_ROTL** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

rotr_expression ::= **_ROTR** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

asr_expression ::= **_ASR** [*affected_flags*] '(' *operand* ',' *operand* ')'
[*bit_specification*]

not_expression ::= **_NOT** [*affected_flags*] '(' *operand* ')' [*bit_specification*]

neg_expression ::= **_NEG** [*affected_flags*] '(' *operand* ')' [*bit_specification*]

zxt_expression ::= **_ZXT** [*affected_flags*] '(' *operand* ')' *bit_specification*

sxt_expression ::= **_SXT** [*affected_flags*] '(' *operand* ')' *bit_specification*

indir_expression ::= **_INDIR** [*affected_flags*] '(' *operand* [',' *endianess*]
[',' *bus_name*] ')'
bit_specification

endianess ::= **_LITTLE** | **_BIG**

bus_name ::= *identifier*

intrinsic_expression ::= ' '' 'intrinsic_name' '' ' [*affected_flags*] '(' [*operand*
(',' *operand*)*] ')' [*bit_specification*]

intrinsic_name ::= '_' *identifier*

hierarchy_expression ::= *LISA_declared_item* [*affected_flags*] '(' [*operand*
(',' *operand*)*] ')' [*bit_specification*]

operand ::= *micro_operation_expression*
| *semantics_related_resources*
| *LISA_declared_item*
| *integer*

affected_flags ::= '—' *flag* (',' *flag*)* '—'

flag ::= **_C** | **_O** | **_N** | **_F**

bit_specification ::= ’<’ *offset0* ’..’ *offset1* ’>’
’<’ *offset0* ’,’ *width* ’>’

offset0, *offset1*, *width* ::= *integer* | *LISA_declared_item*

A.3.2.4 Miscellaneous

semantics_related_resources ::= **_PC** | **_SP** | **_CF** | **_OF** | **_NF** | **_ZF**

LISA_declared_item ::= *identifier* [*bit_specification*]

identifier ::= *character* (*character* | *figure* | ’_’)+

integer ::= figure+

figure ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

character ::= ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’
| ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ | ’y’ | ’z’
| ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’ | ’J’ | ’K’ | ’L’ | ’M’
| ’N’ | ’O’ | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’ | ’X’ | ’Y’ |
|’Z’

Appendix B
CoSy Compiler Library Grammar

This appendix contains the formal decription of the LISA CoSy compiler library grammar.

B.1 Grammar Notation

The keywords are denoted by using bolded font,
e.g., "**KEYWORD**".

The nonterminal symbols are typeset slanted,
e.g., "*nonterminal*".

If the syntax definition contains special characters, they will be quoted with single quotes,
e.g., '}'.

Concatenation of two components is denoted by putting the components in sequence,
e.g., *concatenation* ::= *element1 element2*

Optional components are denoted by surrounding square brackets,
e.g., *optional* ::= [*element*]

Repeating a component zero or more times is denoted with an asterisk,
e.g., *repeat* ::= *element**

Repeating a component one or more times is denoted with a plus,
e.g., *repeat* ::= *element+*

Alternative components are denoted by vertical bars,
e.g., *alternative* ::= *option1* | *option2* | *option3*

Brackets are used to group several elements,
e.g., *elements* ::= (*element1 element2*)

Several elements separated with comma can use the same definition,
e.g., *element1*, *element2* ::= *definition*

B.2 Global Structure

compiler_library ::= *basic_rules* [*semantics_transformations*]
| [*basic_rules*] *semantics_transformations*

basic_rules ::= *rule_category basic_rule**

rule_category ::= **CATEGORY** *category*

category ::= **ARITHMETIC** | **CONVERT** | **LOADSTORE**
| **MOVE** | **CONTROL** | **SPILL** | **CALLING**

B.3 Basic Rules

basic_rule ::= *cosy_ir* [*basic_rule_condition*] [*cosy_condition*]
[*nonterminal_constraint*] [*control_clause*] [*readwrite_clause*]
[*scratch_registers*] [*semantics_pattern*] [*result_clause*]
[*node_assignment*]

B.3.1 CoSy IR

cosy_ir ::= **COSYIR** *mir_source_expression* ['->' *mir_destination_expression*]

mir_source_expression, *mir_destination_expression* ::= *ccmir_expression*
| *nonterminal_expression*

nonterminal_expression ::= *nonterminal_placeholder*
| *spill_nonterminal*

nonterminal_placeholder ::= [**SIGNED**] [**UNSIGNED**] [**IMMEDIATE**]
[**REGISTER**] [**ADDRESS**] [**CONDITION**]
[**MEMORY**] *placeholder_name*

placeholder_name ::= *identifier*

spill_nonterminal ::= **Spill**

ccmir_expression ::= *ccmir_binary_expression* | *ccmir_unary_expression*
| *ccmir_primary_expression*

ccmir_binary_expression ::= *node_name* ':' *binary_node* '(' *mir_operand* ','
mir_operand ')'

binary_node ::= **mirPlus** | **mirMult** | **mirAnd** | **mirOr** | **mirXor** | **mirAddrPlus**
| **mirDiv** | **mirAddrDiff** | **mirDiff** | **mirShiftLeft** | **mirShiftRight**
| **mirShiftRightSign** | **mirAssign** | **mirCompare** | **mirReturn**
| **mirMod** | **mirBitInsert** | **mirBitExtract**

ccmir_unary_expression ::= *node_name* ':' *unary_node* '(' *mir_operand* ')'

unary_node ::= **mirNot** | **mirNeg** | **mirConvert** | **mirContent** | **mirGoto**
| **xirFuncCall** | **mirCall** | **mirActual**

ccmir_primary_expression ::= *node_name* ':' *primary_node*

primary_node ::= **mirObjectAddr** | **mirIntConst** | **mirNoExpr** | **mirAddrConst**
| **mirBoolConst** | **mirRealConst** | **mirNil**

mir_operand ::= *ccmir_expression*
| *nonterminal_placeholder*

node_name ::= *identifier*

B.3.2 Rule Condition

basic_rule_condition ::= **RULE_COND** *rule_conditions*

rule_conditions ::= *type_size_compare*(('||' | '&&') *type_size_compare*)*

type_size_compare ::= *type_size* '==' *type_size*
| *type_size* '!=' *type_size*
| *type_size* '>' *type_size*
| *type_size* '>=' *type_size*
| *type_size* '<' *type_size*
| *type_size* '<=' *type_size*

type_size ::= 'SIZEOF' '(' *target_C_data_type* ')'
| 'SIZEOF' '(' **LARGEST_IMM_NT** ')'

target_C_data_type ::= **CHAR** | **SHORT** | **INT** | **LONG** | **POINTER**

B.3.3 CoSy Condition

cosy_condition ::= **CONDITION** '{' *condition_elements* '}'

condition_elements ::= *condition_element* (('||' | '&&') *condition_element*)*

condition_element ::= ['!'] *condition_name* '(' *condition_operands* ')'
['!'] *condition_operand*
['!'] '(' *condition_elements* ')'

condition_name ::= *identifier*

condition_operands ::= *condition_operand* (',' *condition_operand*)*

condition_operand ::= *node_name*
| *node_name* '.' *node_attribute_name*
| *nonterminal_size*
| *type_size*
| *integer*

nonterminal_size ::= 'SIZEOF' '(' *placeholder_name* ')'

node_attribute_name ::= *identifier*

B.3.4 Nonterminal Constraint

nonterminal_constraint ::= **NONTERMINAL_CONSTRAINT**
constraint (('||' | '&&') *constraint*)*

constraint ::= *nonterminal_name* '==' *nonterminal_name*
| *nonterminal_name* '!=' *nonterminal_name*
| *nonterminal_size* '>=' *type_size*
| *nonterminal_size* '>' *type_size*
| *nonterminal_size* '==' *type_size*

nonterminal_name ::= *placeholder_name*

B.3.5 Control Clause

control_clause ::= **CONTROL** *control_type*

control_type ::= **call** | **branch** | **fallthrough**

B.3.6 Read/Write Clause

readwrite_clause ::= *read_clause* [*write_clause*] | [*read_clause*] *write_clause*

read_clause ::= **READ MEMORY** ';'

write_clause ::= **WRITE MEMORY** ';'

B.3.7 Scratch Registers

scratch_registers ::= **SCRATCH** *scratch_name* (',' *scratch_name*)* ';'

scratch_name ::= *identifier*

B.3.8 Semantics Pattern

semantics_pattern ::= **PATTERN**'{' *compiler_semantics* '}'

B.3.9 Node Assignment

node_assignment ::= **ASSIGNMENT** '{' *assignment*+ '}'

assignment ::= *destination_node_expression* '=' *source_node_expression* ';'

destination_node_expression ::= *node_name* '.' *node_attribute_name*

node_attribute_name ::= *identifier*

source_node_expression ::= *node_name* ['.' *node_attribute_name*]
| *integer*

B.3.10 Result Clause

result_clause ::= **RESULT** *nonterminal_name*

B.4 Semantics Transformations

semantics_transformations ::= **Transformations** *transformation*+

transformation ::= *semantics_transform*
| *transformation_function*

semantics_transform ::= **ORIGINAL** *assignment_statement*
[*scratch_clause*]
TRANSFORM '{' *semantics_statement*+ '}'

transformation_function ::= **TRANSFORATION** '(' *integer* (','
nonterminal_placeholder)* ')' [*scratch_clause*]
'{' *semantics_statement*+ '}'

B.5 Compiler Semantics

compiler_semantics ::= *semantic_statement*+

semantic_statement ::= *assignment_statement*
| *if_else_statement*
| *non_assignment_statement*
| *label_statement*

B.5.1 Assignment Statement

assignment_statement ::= *source_expression* '->' *destination_expression* ';'

source_expression ::= *micro_operation_expression* ['<' *offset* ',' *width* '>']
| *uop_operands* ['<' *offset* ',' *width* '>']
| *constant_expression*

destination_expression ::= *uop_operands* ['<' *offset* ',' *width* '>']
| *indir_expression* ['<' *offset* ',' *width* '>']

B.5.2 Label Statement

label_statement ::= *label_name* ':' ['<' *label_width* '>']

label_name ::= "LLabel_" integer

label_width ::= integer

B.5.3 IF-ELSE Statement

if_else_statement ::= **IF** '(' *conditions* ')' '{' *assignment_statement*+ '}'
[**ELSE** '{' *assignment_statement*+ '}']
| **IF** '(' *conditions* ')'
CONSTANT_ASSIGNMENT '(' *nonterminal_name* ')";'

conditions ::= *condition* (('||' | '&&') *condition*)*
| '(' *conditions* ')'

condition ::= *equal* | *not_equal* | *signed_greater* | *unsigned_greater*
| *signed_greater_equal* | *unsigned_greater_equal*
| *signed_less* | *unsigned_less*
| *signed_less_equal* | *unsigned_less_equal*
| **_CF** | **_OF** | **_NF** | **_ZF**

equal ::= **_EQ** '(' *compare_operand* ',' *compare_operand* ')'

not_equal ::= **_NE** '(' *compare_operand* ',' *compare_operand* ')'

signed_greater ::= **_GTI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_greater ::= **_GTU** '(' *compare_operand* ',' *compare_operand* ')'

signed_greater_equal ::= **_GEI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_greater_equal ::= **_GEU** '(' *compare_operand* ',' *compare_operand* ')'

signed_less ::= **_LTI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_less ::= **_LTU** '(' *compare_operand* ',' *compare_operand* ')'

signed_less_equal ::= **_LEI** '(' *compare_operand* ',' *compare_operand* ')'

unsigned_less_equal ::= **_LEU** '(' *compare_operand* ',' *compare_operand* ')'

compare_operand ::= *micro_operation_expression* ['<' *offset* ',' *width* '>']
| *uop_operands* ['<' *offset* ',' *width* '>']
| *constant_expression*

B.5.4 Non-assignment Statement

non_assignment_statement ::= **_NOP** ';'
| **TRANSFORMATION** '(' *integer* (',' *transform_operand*)* ')'
transform_operand ::= *micro_operation_expression* ['<' *offset* ',' *width* '>']
| *uop_operands* ['<' *offset* ',' *width* '>']
| *constant_expression*

B.5.5 Micro-operation

micro_operation_expression ::= *micro_binary_expressions*
| *micro_unary_expressions*
| *intrinsic_expressions*
micro_binary_expressions ::= *binary_operators* [*affected_flags*] '(' *operand* ',' *operand* ')'

binary_operators ::= **_ADD** | **_ADDC** | **_ASR** | **_SUB** | **_SUBC** | **_MULII**
| **_MULIU** | **_MULUU** | **_AND** | **_OR** | **_XOR**
| **_LSL** | **_LSR** | **_ROTL** | **_ROTR**

micro_unary_expressions ::= *unary_operators* [*affected_flags*] '(' *operand* ')'

unary_operators ::= **_NOT** | **_NEG** | **_SXT** | **_ZXT** | **_INDIR**

intrinsic_expression ::= ' " '*intrinsic_name*' " ' [*affected_flags*] '(' [*operand* (',' *operand*)*] ')'

intrinsic_name ::= '_' *identifier*

operand ::= *micro_operation_expression* ['<' *offset* ',' *width* '>']
| *uop_operands* ['<' *offset* ',' *width* '>']
| *constant_expression*

B.5.6 Operands

uop_operands ::= **REGISTER _PC** | **_FP** | **_SP** | **_CF** | **_OF** | **_NF** | **_ZF**
| *nonterminal_name* '.' *nonterminal_attribute_name*
| *nonterminal_name*
| *nonterminal_placeholder*
| **SYMBOL** '(' *symbol_name* ')'
| *label_name*

symbol_name ::= '_' *identifier*

constant_expression ::= *nonterminal_size*
| *calculation*

calculation ::= *calculation_operand* (('+' | '-' | '*' | '^ ') *calculation_operand*)*

calculation_operand ::= *integer*
| *type_size*
| '(' *calculation* ')'

offset , *width* ::= *constant_expression*

affected_flags ::= '|' *flag* (',' *flag*)* '|'

flag ::= **_C** | **_O** | **_N** | **_F**

B.6 Miscellaneous

identifier ::= *character* (*character* | *figure* | '_')+

integer ::= figure+

figure ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

character ::= ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’
| ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ | ’y’ | ’z’
| ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’ | ’J’ | ’K’ | ’L’ | ’M’
| ’N’ | ’O’ | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’ | ’X’ | ’Y’
| ’Z’

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Jan. 1986. ISBN 0-2011-0088-6.
2. A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.
3. A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, Jan. 1998. ISBN 0-5215-8390-X.
4. A. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. Internal report, University of Virginia, 1998. http://www.cs.virginia.edu/zephyr
5. A. Chattopadhyay, H. Ishebabi, X. Chen, Z. Rakosi, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr. Prefabrication and postfabrication architecture exploration for partially reconfigurable VLIW processors. *Transactions on Embedded Computing Systems*, 7(4):1–31, 2008.
6. A. Chattopadhyay, R. Leupers, H. Meyr, and G. Ascheid. *Language-driven Exploration and Implementation of Partially Re-configurable ASIPs*. Springer Publishing Company, Incorporated, 2008.
7. A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, pages 82–93, 2004.
8. A. Fauth. Beyond tool-specific machine descriptions. In P. Marwedel and G. Goosens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.
9. A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1993.
10. A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the European Design and Test Conference (ED & TC)*, Mar. 1995.
11. A. Gavrylenko. An Optimized Linear Scan Register Allocator for a Retargetable C-Compiler. Master thesis, Software for Systems on Silicon, RWTH Aachen University, 2006. Advisor: M. Hohenauer.
12. A. Halambi, A. Shrivastava, N. Dutt, and A. Nicolau. A customizable compiler framework for embedded systems. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Mar. 2001.
13. A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic software toolkit generation for embedded system-on-chip. In *Proceedings of the International Conference on Visual Computing*, Feb. 1999.
14. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

15. A. Hoffmann, R. Leupers, and H. Meyr. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, Boston, Jan. 2003. ISBN 1-4020-7338-0.
16. A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A novel methodology for the design of Application Specific Instruction Set Processors (ASIP) using a machine description language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.
17. A. Hoffmann, T. Kogel, and H. Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2001.
18. A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Generating production quality software development tools using a machine description language. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2001.
19. A. Inoue, H. Tomiyama, E.F. Nurprasetyo, and H. Yasuura. A programming language for processor based embedded systems. In *Proceedings of the Asia Pacific Conference on Chip Design Language (APCHDL)*, 1999.
20. A. Inoue, H. Tomiyama, H. Okuma, H. Kanbara, and H. Yasuura. Language and compiler for optimizing datapath widths of embedded systems. *IEICE Transactions on Fundamentals*, 12(E81-A):2595–2604, Dec. 1998.
21. A. Jones, D. Bagchi, S. Pal, P. Banerjee, and A. Choudhary. PACT HDL: A Compiler Targeting ASICS and FPGAS with Power and Performance Optimizations, pp. 169–190, 2002.
22. A. Khare. SIMPRESS: A Simulator Generation Environment for System-on-Chip Exploration. Technical Report, Department of Information and Computer Science, University of California, Irvine, Sep. 1999.
23. A. Kitajima, M. Itoh, J. Sato, A. Shiomi, Y. Takeuchi, and M. Imai. Effectiveness of the ASIP design system PEAS-III in design of pipelined processors. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, Jan. 2001.
24. A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programing*, 28(4):347–361, 2000.
25. A. Krall, I. Pryanishnikov, U. Hirnschrott, and C. Panis. xDSPcore: A compiler-based configurable digital signal processor. *IEEE Micro*, 24(4):67–78, 2004.
26. A. Kudriavtsev and P. Kogge. Generation of permutations for SIMD processors. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
27. A. Nohl, G. Braun, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the Design Automation Conference (DAC)*, Jun. 2002.
28. A. Nohl, G. Braun, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. In *IEEE Transactions on Computer-Aided Design*, Dec. 2004.
29. A. Oraioglu and A. Veidenbaum. Application Specific Microprocessors (Guest Editors' Introduction). In *IEEE Design & Test of Computers*, Jan. 2003.
30. A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, 1996.
31. A. Terechko, E. Pol, and J. van Eijndhoven. PRMDL: a machine description language for clustered VLIW architectures. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2001.
32. A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, T. Michiels, A. Nohl, and T. Kogel. Retargetable generation of TLM bus interfaces for mp-soc platforms. In *Proceedings of the International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS)*, pages 249–254, New York, USA, 2005. ACM Press.
33. A. Wieferink, T. Kogel, A. Nohl, A. Hoffmann, and H. Meyr. A generic toolset for SoC multiprocessor debugging and synchronisation. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*, Jun. 2003.

34. A. Wieferink, 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 *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Feb. 2004.
35. Aart J.C. Bik. Vectorization with the intel compilers (part i).
36. ACE – Associated Compiler Experts. CoSy System Documentation parts 1 to 5, 2005.
37. ACE – Associated Computer Experts bv. *SuperTest – Compiler Test and Validation Suite* http://www.ace.nl
38. ACE – Associated Computer Experts bv. *The COSY Compiler Development System* http://www.ace.nl
39. Adelante Technologies. *AR|T Builder* http://www.adelantetechnologies.com
40. Advanced RISC Machines Ltd. http://www.arm.com
41. Advanced RISC Machines Ltd. *ARM9 and ARM11 Data Sheet*, Dec. 1996.
42. Analog Devices Inc. *Analog Devices Homepage* http://www.analog.com
43. ARC International. *ARCtangent Processor* http://www.arc.com
44. ARC International. *ARC Programmers Reference Manual*, Dec. 1999.
45. B. Kerninghan and D. Ritchie. *The C Programming Language*. Prentice Hall Software Series, 1988.
46. B. Moszkowski and Z. Manna. Reasoning in interval temporal logic. In *Logics of Programs: Proceedings of the 1983 Workshop*, pages 371–381. Springer-Verlag, 1984.
47. B. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 63–74, New York, USA, 1994. ACM Press.
48. B. Rau. VLIW Compilation driven by a machine description database. In *Proceedings of the 2nd Code Generation Workshop,* Leuven, Belgium, 1996.
49. C. Fraser. A compact, machine-independent peephole optimizer. In *Principles of Programming Languages (POPL)*, pages 1–6, 1979.
50. C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Publishing Co., 1994.
51. C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992.
52. C. Fraser, R. Henry, and T. Proebsting. BURG — fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.
53. C. Lee. *MediaBench benchmark suite*. http://euler.slu.edu/ fritts/mediabench/mb1
54. C. Liem, F. Breant, S. Jadhav, R. O'Farrell, R. Ryan, and O. Levia. Embedded tools for a configurable and customizable DSP architecture. *IEEE Design & Test of Computers*, 19(6):27–35, 2002.
55. C. Schumacher. Retargetable SIMD Optimization for Architectures with Multimedia Instruction Sets. Diploma thesis, Software for Systems on Silicon, RWTH Aachen University, 2005. Advisor: M. Hohenauer.
56. C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of the Int. Symposium on System Synthesis (ISSS)*, Dec. 1998.
57. Center for Reliable and High-Performance Computing, University of Illinois. *Illinois Microarchitecture Project utilizing Advanced Compiler Technology (IMPACT)*. http://www.crhc.uiuc.edu/IMPACT
58. Coware Inc. http://www.coware.com
59. D. August. Hyperblock performance optimizations for ILP processors. M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996. M.S. thesis.
60. D. August, W. Hwu, and S. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1997.

61. D. Bradlee, R. Henry, and S. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, pages 229–240, 1991.
62. D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.
63. D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for ASIPs. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 27–34, 2002.
64. D. Fischer, J. Teich, R. Weper, U. Kastens, and M. Thies. Design space characterization for architecture/compiler co-exploration. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 108–115, 2001.
65. D. Genin, E. Hilfinger, J. Rabaey, C. Scheers, and H. De Man. DSP Specification using the SILAGE language. In *Proceedings of the Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1057–1060, 1990.
66. D. Kästner. Propan: A retargetable system for postpass optimisations and analyses. In *LCTES '00: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 63–80, 2001.
67. D. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2): 127–145, June 1968.
68. D. Landskov, S. Davidson, B. Shriver, and P. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, 1980.
69. D. Lanner, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess: Retargetable code generation for embedded DSP processors. In P. Marwedel and G. Goosens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.
70. D. Maufroid, P. Paolucci et al. mAgic FPU: VLIW floating point engines for System-On-Chip applications. In *Proceedings of the Emmsec Conference*, 1999.
71. D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *Proceedings of the Int. Symposium on Code Generation and Optimization (CGO)*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.
72. D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, pages 132–143, 2006.
73. Digital Equipment Corporation, Maynard, MA. *Digital Semiconductor SA-110 Microprocessor Technical Reference Manual*, 1996.
74. E. Dashofy, A. van der Hoek, and R. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, 2001.
75. E. Rohou, F. Bodin, A. Seznec, G. Fol, F. Charot, and F. Raimbault. SALTO : System for Assembly-Language Transformation and Optimization. Technical report, INRIA, national institute for research in computer science and control, 1996.
76. Edison Design Group. *Compiler Front Ends for the OEM Market*. http://www.edg.com
77. Embedded-C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf
78. F. Berens, G. Kreiselmaier, and N. Wehn. Channel Decoder Architecture for 3G Mobile Wireless Terminals. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2004.
79. F. Brandner, D. Ebner, and A. Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 13–22, 2007.
80. F. Chow and J. Hennessy. Register allocation by priority-based coloring. *ACM Letters on Programming Languages and Systems*, 19(6): 222–232, June 1984.
81. F. Chow and J. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

82. F. Engel. Interprocedural Pointer Alignment Analysis for a Retargetable C-Compiler with SIMD Optimization. Diploma thesis, Software for Systems on Silicon, RWTH Aachen University, 2006. Advisor: M. Hohenauer.
83. F. Franchetti, S. Kral, J. and C. Ueberhuber. Efficient utilization of SIMD extensions. In *Proceedings of the IEEE*, 93: 409–425, 2005.
84. F. Homewood and P. Faraboschi. ST200: a VLIW architecture for media-oriented applications. In *Microprocessor Forum*, Oct. 2000.
85. F. Yang. ESP: A 10 year retrospective. In *Proceedings of the Embedded Systems Programming Conference*, 1999.
86. Free Software Foundation. *Auto-vectorization in GCC,* 2004.
87. Free Software Foundation. *GNU Compiler Collection Homepage* http://gcc.gnu.org
88. G. Amdahl. Validity of the single-processor approach to achieving large-scale computer capabilities. In *AFIPS Conference Proceedings*, volume 30, page 483, 1967.
89. G. Bette. Retargetable Conditional Execution Support for CoSy Compilers. Diploma thesis, Software for Systems on Silicon, RWTH Aachen University, 2007. Advisor: M. Hohenauer.
90. 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 *Proceedings of the Int. Symposium on System Synthesis (ISSS)*, Oct. 2001.
91. G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwächter, R. Leupers, and H. Meyr. A novel approach for flexible and consistent ADL-driven ASIP design. In *Proceedings of the Design Automation Conference (DAC)*, pages 717–722, 2004.
92. G. Braun, A. Wieferink, O. Schliebusch, R. Leupers, and H. Meyr. Processor/Memory co-exploration on multiple abstraction levels. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2003.
93. G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, Jun. 1982.
94. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, P. Markstein. Register allocation via coloring. *Proceedings of the International Conference on Computer Languages (ICCL)*, 6(1):47–57, Jan. 1981.
95. G. Cheong and M. Lam. An Optimizer for Multimedia Instruction Sets. *In Proceedings of the Second SUIF Compiler Workshop,* Stanford University, USA, 1997.
96. G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of the Design Automation Conference (DAC)*, Jun. 1999.
97. G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: an instruction set description language for retargetability. In *Proceedings of the Design Automation Conference (DAC)*, Jun. 1997.
98. G. Hadjiyiannis, S. Hanono, and S. Devadas. *ISDL Language Reference Manual*, Jan. 1997.
99. G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8): 114–117, 1965.
100. G. Pokam, S. Bihan, J. Simonnet, and F. Bodin. SWARP: a retargetable preprocessor for multimedia instructions. *Concurrency and Computation: Practice and Experience*, 16(2–3):303–318, 2004.
101. G. Ren, P. Wu, and D. Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *16th International Workshop of Languages and Compilers for Parallel Computing*, Oct. 2003.
102. G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 118–131, 2006.
103. G. Smith. Crisis of complexity. In *Gartner Dataquest briefing, 40th Design Automation Conference (DAC)*, Jun. 2003.
104. Gigascale Systems Research Center. *Modern Embedded Systems: Compilers, Architectures, and Languages*. http://www.gigascale.org/mescal

105. GNU – Free Software Foundation. *Bison – GNU Project* http://www.gnu.org/software/bison/bison.html
106. GNU – Free Software Foundation. *Flex – GNU Project* http://www.gnu.org/software/flex/flex.html
107. H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Department of Information Systems, Kyushu University, Jan. 1996.
108. H. Emmelmann, F. Schröer, and R. Landwehr. BEG – a generator for efficient back ends. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 24(7):227–237, Jly 1989.
109. H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, K. Karuri, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr. ASIP architecture exploration for efficient IPSec encryption: A case study. *Transactions on Embedded Computing Systems*, 6(2), 2007.
110. H. Scharwaechter, M. Hohenauer, R. Leupers, G. Ascheid, and H. Meyr. An interprocedural code optimization technique for network processors using hardware multi-threading support. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 919–924, 3001 Leuven, Belgium, 2006. European Design and Automation Association.
111. H. Scharwaechter, R. Leupers, G. Ascheid, H. Meyr, J. Youn, and Y. Paek. A code-generator generator for multi-output instructions. In *Proceedings of the Int. Conference on Hardware/-Software Co-design and System Synthesis (CODES+ISSS)*, Sept. 2007.
112. H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system-on-chip design. In *Proceedings of the Asia Pacific Conference on Chip Design Language (APCHDL)*, Oct. 1999.
113. H. Walters, J. Kamperman, and K. Dinesh. An extensible language for the generation of parallel data manipulation and control packages, 1994.
114. Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction-set Reference Manual (Third Edition)*, 1994.
115. I. Huang and P. Xie. Application of instruction analysis/synthesis tools to x86's functional unit allocation. In *Proceedings of the Int. Symposium on System Synthesis (ISSS)*, Dec. 1998.
116. I. Huang, B. Holmer, and A. Despain. ASIA: Automatic synthesis of instruction-set architectures. In *Proceedings of the SASIMI Workshop*, Oct. 1993.
117. I. Pryanishnikov, A. Krall, and N. Horspool. Pointer alignment analysis for processors with SIMD instructions. In *Proceedings 5th Workshop on Media and Streaming Processors*, 2003.
118. IMEC. http://www.imec.be
119. Institute for Integrated Signal Processing Systems, RWTH-Aachen University. http://www.iss.rwth-aachen.de
120. Institute of Electrical and Electronics Engineers, Inc. (IEEE). IEEE Standard for Verilog Hardware Description Language 2001.
121. Institute of Electrical and Electronics Engineers, Inc. (IEEE). IEEE Standard VHDL Language Reference Manual 2000.
122. Intel Corporation. *Intel C Compiler,* http://www.intel.com
123. International Technology Roadmap for Semiconductors. *SoC Design Cost Model – 2003* http://www.itrs.net
124. ISS RWTH Aachen University. *The DSPstone benchmark suite*. http://www.ert.rwth-aachen.de/Projekte/Tools/DSPSTONE
125. J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Principles of Programming Languages (POPL)*, 1983.
126. J. Ceng, W. Sheng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. Modeling instruction semantics in ADL processor descriptions for C compiler retargeting. *Journal of VLSI Signal Processing Systems*, 43(2–3):235–246, 2006.
127. J. Davidson and C. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, 1980.
128. J. Davidson and C. Fraser. Automatic generation of peephole optimizations. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 111–116, 1984.

129. J. Degener and C. Bormann. GSM 06.10 lossy speech compression. http://kbs.cs.tu-berlin.de/jutta/toast.html, 1992.
130. J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, Jly 1981.
131. J. Fisher. Customized instruction-sets for embedded processors. In *Proceedings of the Design Automation Conference (DAC)*, pages 253–257, 1999.
132. J. Fisher, P. Faraboschi, and C. Young. *Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, December 2004.
133. J. Gyllenhaal, B. Rau, and W. Hwu. Hmdes Version 2.0 Specification. Technical Report, IMPACT Research Group, University of Illinois, 1996.
134. J. Gyllenhaal, W. Hwu, and B. Rau. Optimization of machine descriptions for efficient use. *International Journal of Parallel Programming*, 26(4): 417–447, Aug. 1998.
135. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1996. Second Edition.
136. J. Paakki. Attribute grammar paradigms – A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2): 196–255, June 1995.
137. J. Sato, A.Y. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai. PEAS-I: a hardware/software Co-design system for ASIP development. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E77-A(3):483–491, Mar. 1994.
138. J. Sato, M. Imai, T. Hakata, A. Alomary, and N. Hikichi. An integrated design environment for application-specific integrated processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, Mar. 1991.
139. J. Teich and R. Weper. A joined architecture/compiler design environment for ASIPs. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Nov. 2000.
140. J. Teich, R. Weper, D. Fischer, and S. Trinkert. BUILDABONG: a rapid prototyping environment for ASIPs. In *Proceedings of the DSP Germany (DSPD)*, Oct. 2000.
141. J. van Praet, G. Goossens, D. Lanner, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, Oct. 1994.
142. J. Wagner and R. Leupers. Advanced code generation for network processors with bit packet addressing. In *Proceedings of the Workshop on Network Processors (NP1)*, Feb. 2002.
143. K. Bischoff. Design, Implementation, Use, and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C. Technical Report 92-31, Department of Computer Science, Iowa State University, Irvine, 1992.
144. K. Cooper, T. Harvey, and T. Waterman. Building a Control-Flow Graph from Scheduled Assembly Code. Technical Report, Department of Computer Science, Rice University, Houston, 2002.
145. K. Diefendorff and P. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997.
146. K. Hazelwood and T. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.
147. K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid. A design flow for architecture exploration and implementation of partially reconfigurable processors. *IEEE Transactions on Very Large Scale Integration System*, 16(10):1281–1294, 2008.
148. K. Karuri, M. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained application source code profiling for ASIP design. In *Proceedings of the Design Automation Conference (DAC)*, pages 329–334, 2005.
149. K. Kreuzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.

150. K. Olukotun, M. Heinrich, and D. Ofelt. Digital System Simulation: Methodologies and Examples. In *Proceedings of the Design Automation Conference (DAC)*, Jun. 1998.
151. K. Sagonas and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software, Practice and Experience*, 33(11):1003–1034, 2003.
152. K. van Berkel, F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss. Processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, 16: 2613–2625, 2005.
153. L. Carter, B. Simon, B. Calder, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6): 563–588, 2000.
154. L. Chunho, M. Potkonjak, and W. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 330–335, 1997.
155. L. Guerra et al. Cycle and phase accurate DSP modeling and integration for HW/SW coverification. In *Proceedings of the Design Automation Conference (DAC)*, Jun. 1999.
156. M. Bailey and J. Davidson. A Formal Model and Specification Language for Procedure Calling Conventions.
157. M. Barbacci. Instruction set processor specifications (ISPS): The notations and its application. In *IEEE Transactions on Computers*, pages 24–40, 1981.
158. M. Benitez and J. Davidson. Target-specific global code improvement: Principles and applications. Technical Report, Charlottesville, VA, USA, 1994.
159. M. Ertl. Optimal Code Selection in DAGs. In *Principles of Programming Languages (POPL)*, 1999.
160. M. Flynn. Some computer organizations and their effectiveness. In *IEEE Transactions on Computers*, number C21, page 948, 1972.
161. M. Freericks. The nML machine description formalism. Technical Report, Technical University of Berlin, Department of Computer Science, 1993.
162. M. Freericks, A. Fauth, and A. Knoll. Implementation of complex DSP systems using high-level design tools. In *Signal Processing VI: Theories and Applications*, 1994.
163. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co, 1979. ISBN 0-7167-1045-5.
164. M. Gries and K. Keutzer. *Building ASIPs: The Mescal Methodology*. Springer-Verlag, 2005.
165. M. Hartoog, J. Rowson, P. Reddy, S. Desai, D. Dunlop, E. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of the Design Automation Conference (DAC)*, Jun. 1997.
166. M. Hohenauer, C. Schumacher, R. Leupers, G. Ascheid, H. Meyr, and H. v. Someren. Retargetable code optimization with SIMD instructions. In *Proceedings of the International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS)*, pages 148–153, 2006.
167. M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, H. Meyr, G. Bette, and B. Singh. Retargetable code optimization for predicated execution. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2008.
168. M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. v. Someren. A methodology and tool suite for C compiler generation from ADL processor models. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, page 21276, 2004.
169. M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. PEAS-III: an ASIP design environment. In *Proceedings of the International Conference on Computer Design (ICCD)*, Sept. 2000.
170. M. Itoh, Y. Takeuchi, M. Imai, and A. Shiomi. Synthesizable HDL generation for pipelined processors from a micro-operation description. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E83-A(3), Mar. 2000.
171. M. Jain, M. Balakrishnan, and A. Kumar. ASIP design methodologies: survey and issues. In *Int. Conf. on VLSI Design*, Jan. 2001.

172. M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen. A flexible DSP core for embedded systems. *IEEE Design & Test of Computers*, 14(4):60–68, 1997.
173. M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 23(7):318–328, Jun. 1988.
174. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
175. M. Smelyanskiy, S. Mahlke, E. Davidson, and H. Lee. Predicate-aware scheduling: a technique for reducing resource constraints. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2003.
176. M. Vachharajani, N. Vachharajani, and D. August. The liberty structural specification language: a high-level modeling language for component reuse. *SIGPLAN Notices*, 39(6):195–206, 2004.
177. M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August. Microarchitectural exploration with Liberty. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 271–282. IEEE Computer Society Press, 2002.
178. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
179. MIPS Technologies Inc. *MIPS Homepage* http://www.mips.com
180. MIPS technologies Inc. *MIPS 4Kc Processor Core Datasheet*, Jun. 2000.
181. M. Naberezny. *6502 Homepage* http://www.6502.org
182. Motorola. *DSP56K Manual*, 1998.
183. Motorola Corporation. *AltiVec Technology Programming Interface Manual*, Jun. 1999.
184. Motorola Inc. *MPC750 RISC Microprocessor User's Manual*, 1997.
185. MPEG Consortium. ISO/IEC 11172-3:1993. http://www.chiariglione.org/mpeg/standards/mpeg-1/mpeg-1.htm, 1993.
186. N. Ramsey and J.W. Davidson. Machine descriptions to build tools for embedded systems. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, 1998.
187. N. Ramsey and M. Fernandez. Specifying representations of machine instructions. *IEEE Transactions on Programming Languages and Systems*, 19(3), Mar. 1997.
188. N. Rizzolo and D. Padua. HiLO: High level optimization of FFTs. In *Languages and Compilers for High Performance Computing*, volume 3602, 2005.
189. N. Warter, D. Lavery, and W. Hwu. The benefit of predicated execution for software pipelining. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, 1993.
190. NXP Semiconductors. *Nexperia PNX 1500 family and TriMedia media processors*. http://www.nxp.com
191. O. Schliebusch, A. Chattopadhyay, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, T. Kogel. A framework for automated and optimized ASIP implementation supporting multiple hardware description languages. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, pages 280–285, New York, USA, 2005. ACM Press.
192. O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture implementation using the machine description language LISA. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, page 239, 2002.
193. O. Schliebusch, H. Meyr, and R. Leupers. *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 2007.
194. O. Wahlen. *C Compiler Aided Design of Application-Specific Instruction-Set Processors Using the Machine Description Language LISA*. PhD thesis, Institute for Integrated Signal Processing Systems, RWTH Aachen University, Aachen, 2003.
195. O. Wahlen, M. Hohenauer, R. Leupers, and H. Meyr. Instruction Scheduler Generation for Retargetable Compilation. *IEEE Design & Test of Computers*, 20(1):34–41, 2003.
196. Oberhumer.com GmbH. *Lightweight Lempel-Ziv-Oberhumer (LZO), a lossless data compression library*. http://www.oberhumer.com/opensource/lzo
197. On Demand Microelectronics. http://www.ondemand.co.at

198. P. Anklam, D. Cutler, R. Heinen, and M.D. MacLaren. *Engineering a Compiler: VAX-11 Code Generation and Optimization*. Butterworth-Heinemann, Newton, MA, USA, 1982.
199. P. Briggs, K. Cooper, L. Torczon. Improvements to graph coloring register allocation. *IEEE Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
200. P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. *ACM Computer Architecture News, SIGARCH*, 19(3):266–275, 1991.
201. P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EXPRESSION: An ADL for System Level Design Exploration. Technical Report 98-29, Department of Information and Computer Science, University of California, Irvine, Sept. 1998.
202. P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: an algorithm for automatic generation of reservation tables from architectural descriptions. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, page 44, 1999.
203. P. Kessler S. Graham, and M. McKusick. gprof: a call graph execution profiler. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.
204. P. Marwedel and W. Schenk. Cooperation of synthesis, retargetable code generation and test generation in the MSS. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*.
205. P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, pages 256–261, 2001.
206. P. Paolucci, P. Kajfaszc, P. Bonnotc, B. Candaelec, D. Maufroidc, E. Pastorellia, A. Ricciardia, Y. Fusellad, and E. Guarino. mAgic-FPU and MADE: A customizable VLIW core and the modular VLIW processor architecture description environment. In *Computer Physics Communications*, 139: 132–143, 2001.
207. P. Paulin. Towards application-specific architecture platforms: embedded systems design automation technologies. In *Proceedings of the EuroMicro*, Apr. 2000.
208. P. Paulin. Design Automation Challenges for Application-Specific Architecture Platforms. Keynote speech at SCOPES 2001 – Workshop on Software and Compilers for Embedded Systems (SCOPES), Apr. 2001.
209. P. Paulin and M. Santana. Flexware: a retargetable embedded-software development environment. *IEEE Design & Test*, 19(4):59–69, 2002.
210. P. Paulin, C. Liem, T.C. May, and S. Sutarwala. FlexWare: a flexible firmware development environment for embedded systems. In P. Marwedel and G. Goosens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.
211. P. Paulin, F. Karim, and P. Bromley. Network processors: a perspective on market requirements, processor architectures and embedded SW tools. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2001.
212. P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 153–164, 2005.
213. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.
214. R. Allen, K. Kennedy, and J. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., Oct. 2001. ISBN 1-5586-0286-0.
215. R. Gonzales. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, Mar. 2000.
216. R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Symposium on Microarchitecture*, pages 247–255, Dec. 1993.
217. R. Krishnan. Future of embedded systems technology. In *BCC Research Group*, Jun. 2005.
218. R.L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

219. R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.
220. R. Leupers. Exploiting conditional instructions in code generation for embedded VLIW processors. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 1999.
221. R. Leupers. Code selection for media processors with SIMD instructions. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 4–8, 2000.
222. R. Leupers. LANCE: A C Compiler Platform for Embedded Processors. In *Embedded Systems/Embedded Intelligence*. Feb. 2001. http://www.lancecompiler.com/
223. R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. In *Proceedings of the European Design and Test Conference (ED & TC)*, pages 140–144, 1997.
224. R. Leupers and P. Marwedel. *Retargetable Compiler Technology for Embedded Systems*. Kluwer Academic Publishers, Boston, Oct. 2001. ISBN 0-7923-7578-5.
225. R.M. Senger, E.D. Marsman, M.S. McCorquodale, F.H. Gebara, K.L. Kraver, M.R. Guthaus, and R.B. Brown. A 16-bit mixed-signal microsystem with integrated CMOS-MEMS clock reference. In *Proceedings of the Design Automation Conference (DAC)*, pages 520–525, 2003.
226. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.
227. R. Ravindran and R. Moona. Retargetable cache simulation using high level processor models. In *Proceedings of the Computer Security Applications Conference (ACSAC)*, Mar. 2001.
228. R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: high-level synthesis of nonprogrammable hardware accelerators. *Proceedings of the IEEE Workshop on VLSI Signal Processing*, 31(2):127–142, 2002.
229. R. Wilhelm and D. Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung*. Springer-Verlag, Mar. 1997. ISBN 3-5406-1692-6.
230. R. Woudsma. EPICS, a flexible approach to embedded DSP cores. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, Oct. 1994.
231. Renesas. *http://eu.renesas.com*
232. S. Abraham, W. Meleis, and I. Baev. Efficient backtracking instruction schedulers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 301–308, May 2000.
233. S. Aditya, V. Kathail, and B. Rau. Elcor's Machine Description System: Version 3.0. Technical Report, Hewlett-Packard Company, 1999.
234. S. Bashford and R. Leupers. Constraint driven code selection for fixed-point DSPs. In *Proceedings of the Design Automation Conference (DAC)*, pages 817–822, 1999.
235. 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.
236. S. Basu and R. Moona. High level synthesis from Sim-nML processor models. In *Proceedings of the Int. Conf. on VLSI Design (VLSID)*, page 255. IEEE Computer Society, 2003.
237. S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. Effective compiler generation by architecture description. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 145–152, 2006.
238. S. Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors*. PhD thesis, Massachusetts Institute of Technology, Jun. 1999.
239. S. Kobayashi et al. Compiler generation in PEAS-III: an ASIP development system. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Mar. 2001.
240. S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, 2000.

241. S. Larsen, E. Witchel and S. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 18–29, 2002.
242. S. Lavrov. Store economy in closed operator schemes. *Journal of Computational Mathematics and Mathematical Physics*, 1(4):687–701, 1961.
243. S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1992.
244. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., Aug. 1997.
245. S. Onder and R. Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the International Conference on Computer Languages (ICCL)*, pages 80–89, May 1998.
246. SimpleScalar LLC. http://www.simplescalar.com
247. SPAM Research Group. *SPAM Compiler User's Manual*, Sept. 1997. http://www.ee.princeton.edu/ spam
248. SPARC International Inc. *SPARC Homepage* http://www.sparc.com
249. Stanford University. *SUIF Compiler System*. http://suif.stanford.edu
250. Synopsys. http://www.synopsys.com
251. T. Glökler, S. Bitterlich and H. Meyr. ICORE: a low-power application specific instruction set processor for DVB-T acquisition and tracking. In *Proceedings of the ASIC/SOC conference*, Sept. 2000.
252. T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
253. T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa. Advanced processor design using hardware description language AIDL. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, Mar. 1997.
254. T. Morimoto, K. Yamazaki, H. Nakamura, T. Boku, and K. Nakazawa. Superscalar processor design with hardware description language AIDL. In *Proceedings of the Asia Pacific Conference on Chip Design Language (APCHDL)*, Oct. 1994.
255. T. Proebsting and C. Fischer. Probabilistic register allocation. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 300–310, 1992.
256. Target Compiler Technologies. *CHESS/CHECKERS* http://www.retarget.com
257. Tensilica Inc. *Xtensa C compiler,* http://www.tensilica.com
258. Texas Instruments. *TMS320C54x CPU and Instruction Set Reference Guide*, Oct. 1996.
259. Texas Instruments Inc. *Texas Instruments Homepage* http://www.texasinstruments.com
260. The Open Group. http://www.opengroup.org/architecture/adml/adml_home.htm
261. The Open SystemC Initiative (OSCI). *Functional Specification for SystemC 2.0* http://www.systemc.org
262. Tool Interface Standard Committee (TIS), now SCO group. *ELF: Executable and Linkable Format*.
263. Trimaran. *An Infrastructure for Research in Instruction-Level Parallelism* http://www.trimaran.com
264. UDL/I Committee. *UDL/I Language Reference Manual Version 2.1.0a*, 1994.
265. Underbit Technologies, Inc. *MAD: A high-quality MPEG audio decoder*. http://www.underbit.com/
266. V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *Computer*, 35(9):39–47, 2002.
267. V. Katheil, M Schlansker, and B. Rau. HPL-PD Architecture Specification: Version 1.0. Technical Report, Hewlett-Packard Laboratories, HPL-93-80R1, 2000.
268. V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Jan. 1999.

269. V. Živojnović, J.M. Velarde, C. Schläger, and H. Meyr. DSPStone – A DSP-oriented benchmarking methodology. In *International Conference on Signal Processing Applications and Technology (ICSPAT)*, 1994.
270. V. Živojnović, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs, and multimedia applications – an evaluation using DSPstone. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, Oct. 1995.
271. V. Živojnović, S. Tjiang, and H. Meyr. Compiled simulation of programmable DSP architectures. In *Proceedings of the IEEE Workshop on VLSI Signal Processing*, Oct. 1995.
272. W. Chuang, B. Calder, and J. Ferrante. Phi-predication for light-weight if-conversion. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2003.
273. W. Geurts et al. Design of DSP systems with Chess/Checkers. In *Proceedings of 2nd International Workshop on Code Generation for Embedded Processors*, Mar. 1996.
274. W. Mong and J. Zhu. A retargetable micro-architecture simulator. In *Proceedings of the Design Automation Conference (DAC)*, pages 752–757, 2003.
275. W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2003.
276. X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SIPS)*, pages 548–557, Oct. 1999.
277. W. Qin, X. Zhu, and S. Malik. Modeling operation and microarchitecture concurrency for communication architectures with application to retargetable simulation. In *Proceedings of the International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS)*, pages 66–71, 2004.
278. Y. Bajot and H. Mehrez. Customizable DSP architecture for ASIP core design. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2001.
279. Y. Kim and T. Kim. A design and tools reuse methodology for rapid prototyping of application specific instruction set processors. In *Proceedings of the Workshop on Rapid System Prototyping (RSP)*, Apr. 1999.
280. Y. Kobayashi, S. Kobayashi, K. Okuda, K. Sakanushi, Y. Takeuchi, and M. Imai. Synthesizable HDL generation method for configurable VLIW processors. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)*, pages 842–845, 2004.
281. Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A low-power architecture for software radio. In *ISCA '06. 33rd International Symposium on Computer Architecture*, pages 89–101, 2006.

Index