Technische Universität Wien

D I S S E R T A T I O N

Compiler Backend Generation from Structural Processor Models

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften
unter der Leitung von

Ao.Univ.Prof. Dipl-Ing. Dr. Andreas Krall
Inst.-Nr. E185
Institut für Computersprachen

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Florian Brandner
Matr.-Nr. 9925151
Wilhelmstrasse 45/8
1120 Wien

Wien, 16. Oktober 2009 _____

Acknowledgments

I would like to thank my advisor Andreas Krall, who not only supervised me during the work on this thesis, but also mentored me during my studies and my diploma thesis. I also thank the staff at the Institute of Computer Languages. In particular, Jens Knoop, for his support, enthusiasm, and for sharing his excellent international connections. My office mates, Dietmar Ebner, Adrian Prantl, and Christian Thalinger for all the insights and discussions, but foremost for the great fun; the research partners from OnDemand Microelectronics, in particular, Karl Neumann; the students Andreas Fellnhofer and David Riegler for their contributions during their masters theses. Martin Schöberl for diverting my attention and widening my personal as well as research scope.

Mum, Alex, Peter, Liese, Bine, and Cathi that showed me how much one can achieve, if the family is on one's side. My great love Nina for pushing and supporting me, and for just about everything else too.

Kurzfassung

In den letzten Jahren konnte im Bereich der eingebetteten Computersysteme eine rasante Entwicklung beobachtet werden. Prozessoren, die in diesen Systemen eingesetzt werden, unterliegen seit jeher besonders hohen Anforderungen in Bezug auf den Stromverbrauch, die Chipfläche, die Rechenleistung und die Produktionskosten. Bei der Entwicklung eines neuen Prozessors in diesem Gebiet muß besonderes Augenmerk auf kurze Entwicklungszyklen, sowie hohe Flexibilität bei der Realisierung neuer Technologien gelegt werden. Aus diesem Grund wurden in den letzten Jahren applikationsspezifische Prozessoren immer beliebter, da diese ausreichend Rechenleistung bieten, es aber trotzdem erlauben die gegebenen Einschränkungen, ob nun technischer Natur oder nicht, einzuhalten. Wesentliche Grundvoraussetzungen sind hierbei eine gute Kenntnis des geplanten Einsatzbereichs, sowie geeignete Werkzeuge um alternative Prozessorimplementierungen schnell und einfach erproben zu können.

Ein vielversprechender Ansatz, um Eigenschaften dieser Prozessoren formal zu beschreiben, sind Prozessorbeschreibungssprachen. Basierend auf entsprechenden Prozessorbeschreibungen ist es möglich eine Vielzahl von Werkzeugen abzuleiten. So ist es möglich Softwareentwicklungswerkzeuge und Prozessorsimulatoren bereitzustellen, sowie Abschätzungen des zu erwartenden Stromverbrauchs und der benötigten Chipfläche zu berechnen. Durch die automatische Bereitstellung der entsprechenden Werkzeuge können alternative Prozessorentwürfe und unterschiedliche Befehlserweiterungen eines Prozessors schnell und elegant evaluiert werden. Dies verspricht eine entscheidende Verkürzung der Produktentwicklungszyklen.

In dieser Arbeit wird die neu entwickelte Prozessorbeschreibungssprache xADL vorgestellt. Im Gegensatz zu verwandten Systemen wird durch diese Sprache ausschließlich die Struktur der Prozessorimplementierung beschrieben. Der Befehlssatz, obwohl durch die Beschreibung nicht explizit dargestellt, ist ein integrales Grundkonzept der Sprache. Ein Extraktionsverfahren analysiert die Struktur des Prozessors und berechnet daraus eine abstrakte Darstellung der einzelnen Befehle, die durch den Prozessor unterstützt werden. Das abstrakte Modell des Befehlssatzes steht in enger Beziehung zu den zugrundeliegenden Hardwarekomponenten. Die Sprache ist daher in verschiedensten Anwendungsszenarien nutzbar. Im Vergleich zu anderen Prozessorbeschreibungssprachen bietet die xADL-Sprache eine Vielzahl nützlicher Erweiterungen, die zu besonders kurzen und intuitiv lesbaren Prozessormodellen führen. Dies umfasst beispielsweise erweiterbare Typen zur Beschreibung von Hardwarekomponenten, die Klassen und Templates der Programmiersprache C++ ähnlich sind.

Die Praxistauglichkeit dieses Ansatzes wird am Beispiel eines Übersetzergenerators untersucht. Hier werden die wesentlichen Komponenten eines Übersetzers aus einer gegebenen xADL-Beschreibung generiert. Dies umfasst im Speziellen die Registerbelegung, die Befehlsanordnung, sowie die Befehlsauswahl. Messungen zeigen,

dass die generierten Übersetzer mit handgeschriebenen Produktivsystemen konkurrieren können. Für eine Beschreibung der MIPS-Architektur erzielt der generierte Übersetzer Laufzeitverbesserungen von bis zu 9% für einzelne Benchmarkprogramme. Im Durchschnitt ist eine moderate Verschlechterung der Laufzeit von lediglich 15% festzustellen. In Anbetracht der um bis zu 34% verminderten Codegröße sind diese Ergebnisse ausgezeichnet. Noch bessere Resultate wurden für zwei Konfigurationen des VLIW-Prozessors CHILI erreicht. Hier beträgt die Laufzeitverbesserung bis zu 20%, bei gleichzeitiger Reduktion der Codegröße zwischen 3% und 47%. Im Durchschnitt über alle Benchmarkprogramme ist eine minimale Verschlechterung der Laufzeit messbar, die 5% beziehungsweise 3% beträgt.

Zusätzlich wird die Vollständigkeit der generierten Übersetzer untersucht, d.h., ob der resultierende Übersetzer tatsächlich in der Lage ist für alle durch die Sprache zulässigen Eingabeprogramme entsprechenden Maschinencode zu erzeugen. Traditionelle Ansätze, basierend auf Baumautomaten, sind im Rahmen heutiger Übersetzersysteme nur bedingt einsetzbar, da häufig verwendete dynamische Überprüfungen während der Befehlsauswahl nicht dargestellt werden können. Ein neues Verfahren namens *Terminal Splitting* wird vorgestellt, das erlaubt diese Überprüfungen explizit durch neue Terminalsymbole darzustellen. Eine durch Terminal Splitting vorverarbeitete Spezifikation der Befehlsauswahl wird sodann mit Hilfe des traditionellen Ansatzes auf Vollständigkeit geprüft. Das vorgeschlagene Verfahren ist vollständig in den Übersetzergenerator integriert und erlaubt dem Prozessordesigner wertvolle Information in der Form von Gegenbeispielen zur Verfügung zu stellen.

Abstract

The embedded systems computing domain showed a tremendous development in recent years. Processors used in these systems face rigid constraints in terms of power consumption, chip area, performance, and production costs. Short development cycles and great flexibility in the adoption of new technologies are key factors for successful embedded processors. Application-specific instruction set processors have proven successful in providing the necessary computing power, while meeting the various technical and non-technical design constraints. However, the development of such processors requires intimate knowledge of the processor's application domain and appropriate tools to evaluate design alternatives quickly.

A promising approach to formally specify processor design alternatives and automatically derive the necessary tools for design space exploration are processor description languages. These languages capture the instruction set, and often also the hardware organization of the given processor using an abstract specification. The information provided through the processor models can be used to automatically derive software development and simulation tools, as well as area and power estimates. This approach has the potential to significantly reduce the turn-around time during the evaluation phase of a new application-specific processor, because alternative instruction set extensions and hardware designs can be specified and evaluated systematically.

In this work the novel xADL language is presented, which, in contrast to most contemporary processor description languages, focuses on a structural modeling of the processor's hardware organization. The instruction set, even though not specified explicitly, is a central concept of the language and its support tools. Through instruction set extraction an abstract model of the instruction set is automatically derived from the structural specification. The instruction set view combined with the detailed hardware model provides the necessary information to derive high-quality tools for design space exploration. The language allows the reuse of hardware components through extensible types, similar to classes and templates known from the C++ programming language. In comparison to other processor description languages, xADL specifications are thus very compact and readable.

The feasibility of the approach is demonstrated by a compiler backend generator. It is shown how the essential processor-specific components of a compiler backend can be derived from xADL models, including the register allocator, the instruction scheduler, and the instruction selector. The automatically generated compilers are competitive to handcrafted production compilers. For a MIPS processor model speedups of up to 9% have been measured for certain benchmarks. On average a moderate slowdown of 15% has been observed, which is remarkable considering the code size reduction of up to 34%. Even better results have been measured for two configurations of the CHILI VLIW processor, where speedups of up to 20% and an

average slowdown of only 5% to 3% can be reported. The code size reductions for the CHILI range from 3% up to 47%.

Further, the completeness of the generated compiler components is investigated, i.e., whether the generated compiler is able to produce machine code for all input programs possibly accepted by the compiler frontend. Traditional approaches based on tree automata are not applicable in the context of modern compilers, instruction selection process is often controlled by dynamic checks. These checks can not be represented by tree automata. Terminal splitting is proposed to explicitly represent the dynamic checks present in the instruction selector specification. The transformed specification is then processed by a traditional completeness test. The proposed approach is integrated with the compiler generator system and thus provides valuable feedback to the processor designer in the form of counter examples.

Contents

List of Figures

List of Tables

1 Introduction

The domain of embedded systems showed a dramatic development during the last years, and today belongs to the largest and fastest growing business in the software and in particular in the semiconductor industry. Embedded systems are almost omnipresent in everybody's life, ranging from mobile phones and personal digital assistants (PDAs), over digital media player for DVDs, blue-ray discs, and MP3s to commodity appliances like refrigerators, coffee machines, and washing machines. Even safety-relevant systems in medical devices, in the automotive industry, and avionics rely on these embedded devices. It is thus not surprising that the field of embedded systems contributes a fair amount to the total sales volume of the semiconductor business, has grown to a driving force in research and innovation, and has become a major field of today's computer science. For example, the sales of processors specialized in *Digital Signal Processing* (DSP) applications, already exceeds 20% of the global semiconductor market since 2002 [120]. Even more, in the year 2006 embedded DSP processors contributed more than 95% of the total volume of processor units sold [120]. The number of general purpose server, workstation, and laptop processors sold in comparison is negligible. Additionally, the market for processors specialized for embedded systems is expected to grow faster than the general purpose computing domain in the foreseeable future, making this field a prominent candidate for future research and innovation.

Embedded processors face a wide range of requirements that differ greatly depending on the particular application domain. The range of applications, however, is very diverse, ranging from general computing tasks and computationally intense multimedia processing in modern mobile phones to computationally less demanding control tasks in the automotive sector that favor predictable and fail-safe behavior over performance. Consequently, specialized processors for different application domains evolved, which are particularly well suited for the functional requirements as well as non-functional requirements, such as performance, power consumption, predictability, robustness, and production costs. This specialization is supported by the very nature of embedded systems: they are usually *invisible* to the end-user, i.e., the end-user is not aware of the computer system involved. The computer is thus only part of a larger system and not in the center of attention. The purpose of the embedded system is usually well-defined and limited in scope. It is thus possible to derive streamlined and optimized solutions for a given task that reduce production costs and increase efficiency.

The ultimate goal of an embedded system engineer is thus not to provide a general computing system, but to derive a specialized system tailored to the functional and non-functional requirements of the problem at hand. In order to achieve this goal, engineers are very flexible in their use of new technologies that are quickly adopted. For example, it is not unusual to change the processor implementations and even vendors frequently during the development of product lines in order to achieve the best efficiency and cost trade-offs. The problem of legacy software and

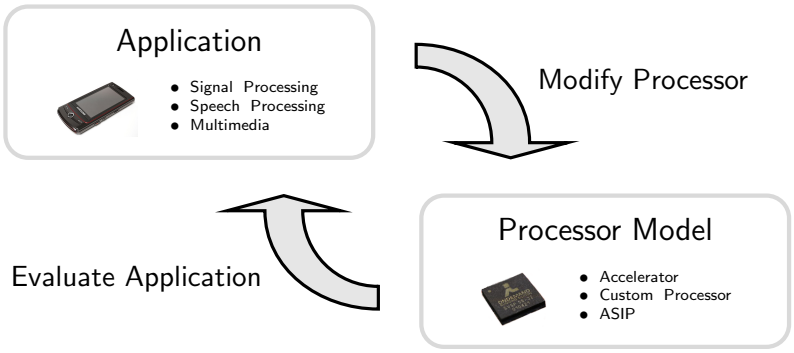

Figure 1: The processor model, specified using a processor description language, is iteratively modified during design space exploration in order to achieve the best possible performance, power, and area trade-off for a given application.

binary compatibility appears in a relaxed form in embedded systems. The software running on the system is in most cases controlled entirely by the manufacturer and can easily be updated and replaced. Usually, guaranteeing tool compatibility, i.e., compatible tool chains of compiler, linker and assembler, is sufficient in order to rebuild the software for a new platform from source code.

Because of this fast changing and highly competitive environment short development cycles are a key to success. It is thus not surprising that the idea of *hardware/software co-design* has been adopted early in this domain by researchers and engineers alike. Hardware/software co-design is an approach to the development of new systems that tries to achieve optimal results using a balanced mix of hardware and software techniques. The development of the final hardware and software components is tightly coupled and individual tasks are implemented either using flexible software techniques or using efficient hardware solutions as needed. The hardware platform is usually built using a combination of off-the-shelf hardware blocks, customized processors, *Application-Specific Instruction Processors* (ASIPs), and dedicated hardware accelerators. Very often all these components are integrated on a single chip forming a *System-on-Chip* (SoC).

Especially, ASIPs are becoming more and more popular in embedded systems. These processing elements are streamlined instruction processors that are particularly well suited for a given target application. A set of special purpose primitives is realized using dedicated instructions that improve the efficiency in terms of power consumption and performance while minimizing silicon area requirements, and thus production costs. At the same time these processing elements still provide enough programmability to adopt the system to new requirements, work around software and hardware bugs or simply update the software stack. However, the development of ASIPs is a highly complex task, that requires intimate knowledge of the application domain as well as hardware and architecture design. In many cases an iterative design process called *Design Space Exploration* (DSE) is used to find an optimal ASIP configuration for a set of typical algorithms of the target application.

Different configurations are evaluated using a preliminary implementation of the algorithms. During this process performance, power and other metrics of interest are collected for each design alternative and stored in a database for later assessment.

Design space exploration opens a large variety of interesting research questions. A major research question is how to minimize the number of design alternatives considered for evaluation in a virtually unbounded design space. Similarly, how can be assured that the most relevant design points are captured during exploration. This work, however, focuses on more fundamental problems, namely: (1) how can design alternatives be modeled formally in a domain specific language, (2) how can the required software tools for the exploration be derived from such a model. *Processor Description Languages* (PDLs) are a promising approach to solve both problems. These languages provide the required primitives to specify the instruction set of a processor design and (semi-)automatically derive software development tools such as compiler, assembler, linker, and instruction set simulator from a given processor model. Figure 1 depicts an example workflow for the design of a new ASIP using a processor description language. The designer iteratively extends and adopts the processor model in order to reach the best possible solution. During each iteration step, a series of simulation runs is performed to evaluate the modifications and expose bottlenecks that have to be dealt with during the next iteration.

1.1 Processor Description Languages

In recent years, processor description languages have managed to evolve from being sole research projects into products that are actively used and adopted successfully by leading companies in order to develop and design highly specialized and tuned architectures [76]. A processor model typically consists of several layers that include information on the hardware organization, the instruction set, instruction semantics and timing. Meta-information such as assembly syntax, *Application Binary Interface* (ABI) conventions, etc., can be specified in most languages. Structuring the models, for example by instruction classes, enables code reuse across different instructions and leads to concise and compact models. Thus, adding new instructions or adopting existing instructions is often only a matter of a few lines of code.

Processor description languages, sometimes also referred to as *Architecture Description Languages* (ADLs), can roughly be categorized into three distinct groups [131]:

1. **Structural** languages offer primitives that directly match abstractions typically found in *Hardware Description Languages* (HDLs). The processor model thus closely resembles the structure of the actual hardware implementation.

2. **Behavioral** languages on the other hand primarily focus on the *Instruction Set Architecture* (ISA), and typically provide some means to structure and order instruction variants and meta-information such as assembly syntax, binary encoding, and abstract instruction semantics.

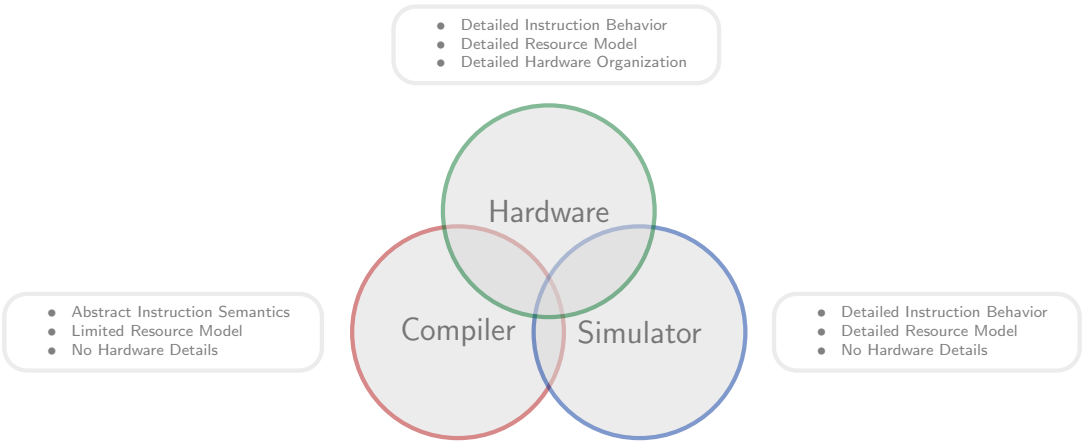

Figure 2: Most processor description languages were initially specialized to one particular task, e.g., compiler generation, and were later extended to other tasks.

3. **Mixed** approaches finally combine the structural and behavioral view of the processor and provide mechanisms to model both, the hardware structure and the instruction set architecture. A mapping mechanism between the structural and the behavioral description allows to relate information of the two models to each other.

The information that is available in these processor specifications can be used to (semi-)automatically generate software tools that are customized for the particular processor. This includes standard development tools such as a compiler, an assembler, a linker, and an instruction set simulator. However the models can also be used to generate test cases for compiler and hardware verification, can serve as references during hardware development, and can even be used to derive hardware models in a general purpose hardware description language, such as VHDL or Verilog.

The various flavors of languages are usually geared towards a particular application. Figure 2 presents three major application fields and the required information that needs to be available within processor descriptions: (1) compiler generation, (2) generation of instruction set simulators, and (3) hardware synthesis using a general purpose HDL. Compiler generation requires an abstract model of the behavior of the individual instructions in order to map the architecture-independent intermediate representation of the compiler to machine code of the target processor. In addition, a resource model is needed to avoid hazards and ensure correct code. Detailed information on the hardware structure is of less importance for the compiler, i.e., it is not important how the individual instructions are realized in hardware. The situation changes when an instruction set simulation engine is to be generated from a processor model. A detailed model of the instruction behavior is required, along with a detailed resource model that is able to accurately capture all sorts of hazards, stalls, delays, and instruction latencies. The hardware structure is only relevant if the behavior of instructions is influenced, all other details are ignored for the sake

of simulation efficiency. However, this changes when synthesizable hardware models are to be extracted from processor descriptions. For this scenario not only detailed knowledge of the instruction behavior is required, but also details on the timing, the realization in hardware, and the interaction with other hardware components.

Behavioral languages typically support the generation of compiler backends and related development tools, such as assemblers and linkers, very well, but lack information needed for accurate simulation and automatic generation of hardware models. Structural languages are usually well suited for these *low-level* tasks, but in turn lack abstract semantic models of instructions. Generating a compiler is thus more complicated in these systems. To overcome such limitations, many systems have gradually adopted features and ideas from the respective other style. Most contemporary languages thus follow the mixed approach that supports both low-level and high-level applications equally well.

1.1.1 Behavioral Languages

Many languages that follow the behavioral approach originated from generalized compiler backend specification languages. The specifications provide an abstract model of the target processor and focus on the instruction set architecture rather than on implementation details. Consequently, behavioral languages are well suited for high-level tasks such as compiler generation or verification.

The behavior of each instruction is specified separately and annotated with additional properties. These properties usually include the assembly syntax, the binary encoding, and the timing. The instruction specifications often contain redundant information, e.g., instructions that share the same or very similarly structured binary encoding. Many behavioral languages thus allow the reuse of instruction specifications. For example, in nML [57] instructions are specified using a grammar. Derivations of this grammar represent the individual instructions of the target processor. New instructions can be composed using `AND`-rules that combine several possible partial instruction specifications, and `OR`-rules that can be used to enumerate variations of an instruction.

Behavioral models do not specify the underlaying hardware organization or structure, except for an abstract model of the registers, memories, and caches. However, constraints of the hardware implementation need to be considered for a faithful model. Many languages thus provide a very basic resource model that is powerful enough to express certain hardware constraints using symbolic resources, very much like resource tables used during instruction scheduling in modern compilers [140].

In addition to the instruction specifications, meta-information on the programming model, register usage and calling conventions, and the application binary interface is provided. This information is particularly important when software development tools are generated from a processor model in order to realize function calls and enable interoperability with possibly existing third-party tools.

1.1.2 Structural Languages

Structural processor description languages have their origins in hardware synthesis, similar to the hardware description languages VHDL and Verilog. These languages focus on a detailed model of the processor's hardware structure and organization. Structural processor models have several advantages that are attractive to processor designers. The languages usually provide abstractions like modules or components that have well-defined interfaces, and therefore, can be extended and exchanged easily. It is possible to develop libraries of components and combine them quickly to describe new processors or processor variants. This approach also resembles closely the traditional design using hardware description languages and thus lowers the initial effort to learn the concepts of a new specification language. In addition, structural models have a close coupling between the behavior of an instruction and the underlaying hardware components that implement that behavior. This is particularly useful for idiosyncratic architectures, where details of the hardware implementation are visible at the instruction set level.

On the downside, however, the instruction set of the processor is not specified explicitly. Instead, the instruction set needs to be extracted from the data path using static analysis techniques. The capabilities of this static analysis thus has a large impact on the usability of the specifications and may restrain the scope of the language. The hardware designers need to be well aware of the limitations and restrictions of the analysis to achieve the desired results.

Even if the analysis is able to extract the instruction set of the processor, it is often hard to determine the abstract behavior of the individual instructions. A particular implementation of a processor may realize the behavior specified by the instruction set architecture in different ways. Because of the detailed structural model these implementation peculiarities are visible to the tools processing the specification. For example, an implementation may choose to use dynamic branch prediction. The behavior of the branch predictor, its effect on the processor's state, and the interaction with all instructions that are possibly affected by the branch predication has to be analyzed in order to determine the exact semantics of branch instructions. Applications that require more abstract models of the instruction behavior are thus generally hard to realize using structural languages.

Most of these problems can be avoided by careful design of the language and by providing the right abstractions at the language level. Choosing the right level of abstraction is, however, still an open research problem. Part of this work is thus devoted to this issue.

1.1.3 Mixed Languages

Over time languages that followed either the behavioral or the structural approach have started to gradually adopt ideas of languages following the other style. This

lead to a new category of mixed processor description languages. Instead of focusing on one particular task most of these languages target a very broad spectrum of applications, ranging from high-level compiler generation to low-level hardware synthesis. Languages of this category combine the abstract view of the processor's instruction set with a detailed structural view of the hardware implementation. Consequently, the combined information allows the tools operating on the processor model to pick the view that is best suited for its particular task. Most contemporary languages follow this paradigm.

A major issue of mixed languages is the problem of redundancy. In order to be useful, both views need to capture a huge amount of information on the target processor, very often this information must be provided by the processor designer at different abstraction levels for the instruction set view and the structural view. If one view of the model is modified, the other view needs to be updated as well. Keeping the specifications consistent is thus a major challenge during the development and validation of a processor model. So far, tools that support automatic consistency checks are missing.

1.1.4 Architecture Styles

Besides the specification paradigm, processor description languages can also be differentiated by the range of processor architectures that are supported. It is almost impossible to support all architectural styles and provide a generic platform that can be used to model *every* possible programmable system. In the domain of embedded systems relatively simple and efficient in-order pipelined processors are very common due to area and power constraints. Almost all processor description languages thus limit their primary focus on this class of processor architectures, either in the form of RISC or VLIW machines. The predominant architectural style in the general purpose computing domain, superscalar out-of-order architectures, is in turn not well supported.

1.2 Application of Processor Description Languages

As noted before, design space exploration is the main application scenario for processor description languages, i.e., the design of a new streamlined instruction processor for a particular application or application domain. The required tools to perform the exploration are (semi-)automatically derived from the processor model, including the compiler, assembler, linker and simulation tools. However, the information that is available in the processor models can be used in various ways independent from design space exploration. Figure 3 depicts the most common application fields targeted by many processor modeling frameworks.

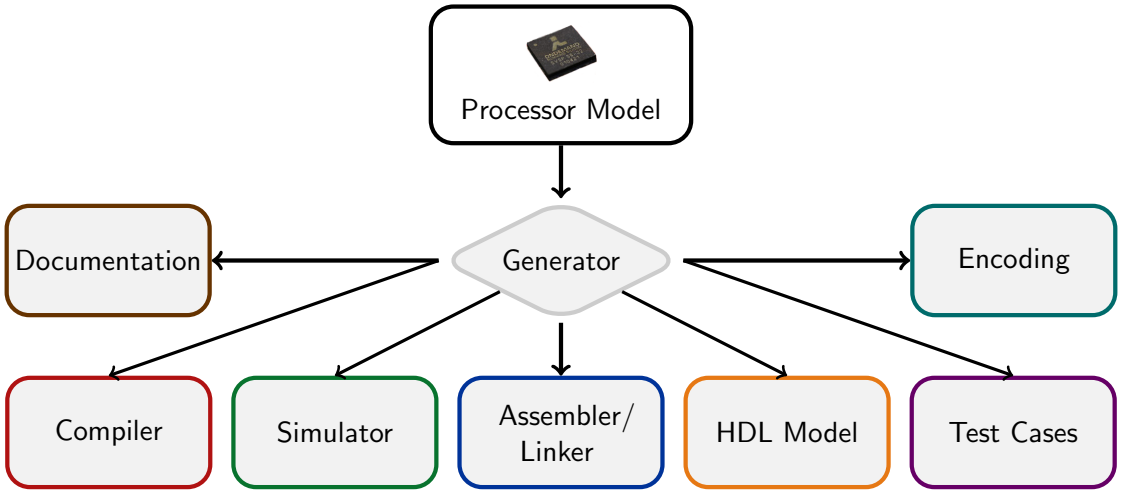

Figure 3: Software development tools, test cases and even hardware models in a general purpose hardware description language can be derived from a processor model.

1.2.1 Documentation and Design

The processor specifications capture a large amount of information on the internal organization of the processor and its instruction set in a formal and concise way. This model can easily be modified and extended and is well suited to communicate designs and design alternatives between development teams and engineers, the management, customers, and third-party vendors. In contrast to documentation in natural language these specifications are compact, more precise, and enable preliminary experiments in combination with the proper tools.

Some processor description languages even support the automatic generation of the processor manual and the instruction set reference manual. The formal architecture and instruction specifications are enriched with comments and documentation in natural language that are later compiled into a user manual.

1.2.2 Verification and Validation

Related to the documentation and design of the processor is the verification and validation of the processor model, the processor implementation in hardware, and the accompanying software tools. During the product development cycle this phase is usually the most tedious and costly one. It is well known that problems encountered during an early phase of the product cycle are less costly to fix than in later phases. Processor description languages are a valuable tool to derive test suites and micro-benchmarks early during the design. Even the structural equivalence between the processor model and hand-tuned HDL implementations of that processor can be verified using formal methods. In addition, software development can begin early

on, because of the availability of suitable development tools that are generated from the processor models. Consequently, problems can be spotted early and costs for validation and verification can be reduced later on.

1.2.3 Assembler and Linker

The automatic generation of assembler, disassembler, and linker is supported by almost all processor description languages. Large portions of these tools are straightforward to generate once the assembly syntax and binary encoding of the individual instructions is known. However, supporting more advanced features such as relocation of symbols during linking, position independent code and dynamic linking, and debug information is considerable harder. The problem with these features usually arise from interoperability issues with existing system libraries, firmware, and operating systems. Capturing the conventions expected by these systems in the formal model of a processor description language is very hard and thus usually not well supported.

1.2.4 Compiler

In comparison to other tasks, compiler generation is the most demanding application of processor description languages. A large number of compiler components are architecture-dependent and need to be customized for the target processor, most important are backend phases such as the register allocation, instruction scheduling, and instruction selection. But also optimizations that are applied in the middle- and frontend of a compiler are, to some degree, architecture-dependent. For example, the data representation and data layout in C/C++ frontends are architecture-dependent. Loop optimizations can be applied more efficiently when the optimization is aware of the supported addressing modes and the cache/memory organization, et cetera.

By far the most challenging task is the automatic generation of an optimizing instruction selector. During instruction selection the architecture independent intermediate representation of the compiler is translated to processor specific assembly or machine code. This translation has to preserve the semantics of the original program and should be efficient, i.e., the best possible instruction sequence should be selected that minimizes code size, execution time, and power consumption. The biggest obstacles are caused by limitations of the instruction set. The intermediate representation of the compiler is very general and is required to support all language features, in particular all arithmetic operations and addressing operations. Not all of these constructs can be implemented using a single instruction of the target processor. The behavior of constructs that can not be represented need to be emulated, or otherwise cause the compiler to fail during code generation. This emulation can be as simple as a sequence of instructions, but may also involve function calls to complex library routines, e.g., in the case of floating point operations. A second

problem that arises frequently are restrictions in the use of registers. If, during the computation of an expression a value is stored into a register, the instruction selector is required to ensure that all subsequent operations are able to retrieve this value. In particular in the case of application-specific instruction processors this kind of restrictions are very common.

1.2.5 Instruction Set Simulator

A key component of all processor description and exploration systems is an accurate instruction set simulator that is capable to collect detailed statistics on the processor's behavior at runtime. In addition, simulation tools are valuable during software development for early prototyping as well as testing and debugging purposes. In both cases simulation speed is of utmost importance for the simulator to be actively used and accepted by the end users. Improving the simulation speed has thus been researched heavily in the past. Facing the rapid development of multi-core systems and complex systems-on-chip, efficient simulation tools can be expected to be a hot research topic for some time to come.

Simulators derived from processor models usually focus on the efficient simulation of a single processor core using interpretation as well as static and dynamic compilation techniques. These techniques are particularly useful for relatively deterministic in-order pipelined architectures, but fail to deliver the required performance in the presence of dynamic scheduling and speculative execution.

1.2.6 Hardware Synthesis

Deriving hardware models is another challenging problem. The majority of systems is able to derive some form of VHDL or Verilog specification from a processor model that can then further be processed by synthesis tools or serve as a reference design for the handcrafted processor implementation.

Although VHDL and Verilog are in principle independent from the target technology many low-level constructs need to be expressed using vendor specific patterns and libraries to achieve optimal results. This applies to designs targeting *Field Programmable Gate Arrays* (FPGAs) and silicon processes alike. In particular in the case of behavioral languages the designer has very little control over the final hardware implementation. It is thus very hard to fine-tune the generated hardware model to improve chip area, clock frequency, and power consumption.

1.2.7 Encoding Optimization

Defining and maintaining the specification of binary encoding of individual instructions is error prone and tedious, in particular during the early design phase when the

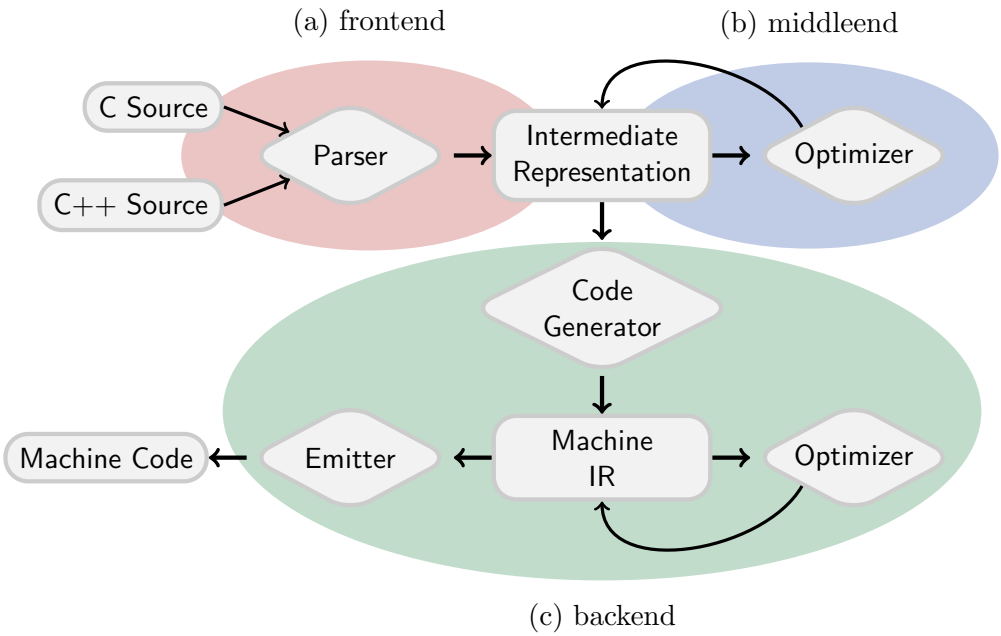

Figure 4: Structure of a compiler consisting of (a) a frontend, (b) an optimizing middleend, and (c) an architecture-dependent backend.

processor specification changes frequently. In many cases a very basic instruction encoding is sufficient for an initial performance evaluation using simulation. Some languages thus allow the encoding to be omitted and automatically derive a suitable instruction encoding. The designer may then choose to optimize this encoding manually or proceed with the tool-generated encoding. Some systems can even help to find an optimal encoding using execution profiles and static program statistics. These statistics are combined with the requirements of the individual instructions, i.e., the bits required to encode the instruction operands, and predefined constraints from the designer in order to minimize code size or reduce the complexity and power consumption of the instruction decoder in hardware.

1.3 Retargetable Compilation

A major application of processor description languages is the automatic or semi-automatic customization of a compiler. A compiler is a software program that translates a *source* program that is usually specified using a high-level language such as C or C++ to another *target* language [3, 140]. Typically the target language is *machine code* that can be processed efficiently by the processor of a computer system. Alternatively, the target language could be a *byte-code* representation of a virtual machine, such as the Java Virtual Machine, or, in the case of *source-to-source* translators, another high-level language. The translation usually involves some form of transformation or optimization intended to speed up the execution of the source

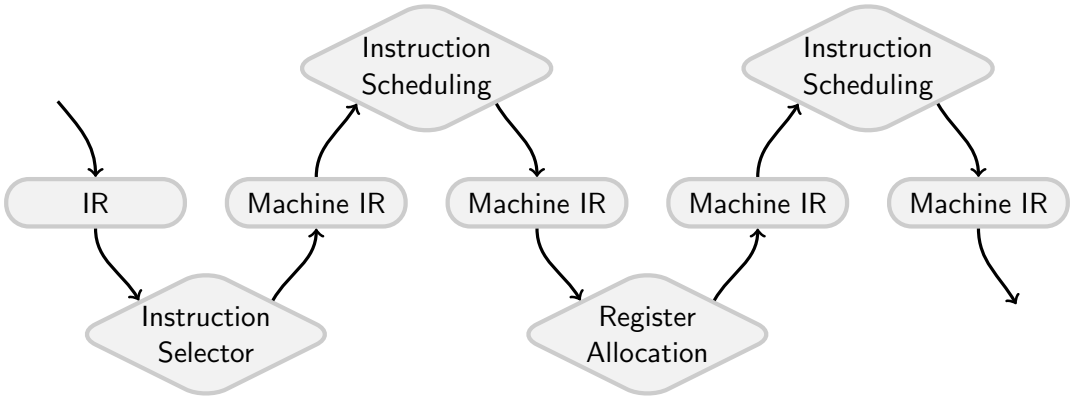

Figure 5: The three major phases of a compiler backend, instruction selection, instruction scheduling, and register allocation.

program on the target platform, reduce the static and/or dynamic memory requirements, or improve other metrics of interest. A compiler consists of three major components: (1) a *frontend* that reads and parses the source program and generates a generic representation of the program, (2) a largely target and source language independent *middleend* that transforms and optimizes the intermediate representation, and (3) a highly target-dependent *backend* that generates the final machine code. Most compilers follow this organization, possibly with slight variations, that is further detailed in Figure 4.

A special class of compilers are the so-called *retargetable* compilers. These systems are special in that the target-dependent components of the backend are very flexible and can quickly be adopted to a new instruction set or processor implementation. In the simplest form these target-dependent components are encapsulated in modules that are adopted manually to the new architecture. Some compilers come with specialized backend specification languages to simplify the retargeting. However, these specifications are typically tailored to the particular algorithms implemented within the compiler and, in particular, do not capture all implementation details of the processor. Instead, the backend specification is restricted to a very abstract model of the instruction set and the processor implementation that provides just enough information to guarantee an efficient and correct translation. If certain constraints of the processor can not be captured by the specification language, it is often necessary to manually extend or replace central components of the backend. Processor description languages show some similarities to these languages, but are typically more powerful and cover the processor design in more detail.

The backend is the central component of a retargetable compiler.[1] Most of the transformations and optimizations require knowledge about the target processor and thus need to be customized properly. Most modern compiler backends have a very similar organization, depicted in Figure 5. The *instruction selector* is one of the first

[1]For some high-level languages the front- and middleend may also be architecture-dependent.

architecture-dependent translation steps. During this phase the intermediate representation of the compiler is transformed into a new representation that is very close to the actual machine instructions of the target processor. The main difference between this program representation and the final machine code are two-fold. For one, initially an infinite set of *virtual registers* represents local variables instead of actual registers that are available in the hardware. The register allocation phase later on calculates an assignment of the local variables to hardware registers and temporary memory locations and eliminates the virtual registers. Secondly, instructions with constant operands that are not yet fixed or memory operations that do not yet have a final addressing mode are represented by generic pseudo instructions. The pseudo instructions are replaced by concrete instructions once these values are known. The basic idea in both cases is to avoid premature assignment decisions. Another important backend phase is instruction scheduling. This phase is often applied twice once before register allocation, often referred to as *prepass* scheduling, and again after the assignment of local variables to hardware registers, usually referred to as *postpass* scheduling. The scheduling algorithm searches for an optimal ordering of the machine instructions in order to minimize the execution time of the program and optimally utilize the available hardware resources. In between the pre- and postpass scheduling phases the third backend phase is performed, the register allocation. Here, virtual registers are eliminated and replaced by hardware registers. If the number of hardware registers is not sufficient to hold all currently live values, *spill code* is generated. Occupied registers are disposed by storing the value currently held in the particular register to a temporary memory location. Corresponding memory load operations ensure that the proper value is available for later uses of the register. These additional store and restore operations may cause considerable overhead and thus need to be minimized.

The quality of the final machine code highly depends on the interaction between all three translation steps. This leads to *phase-ordering* issues, i.e., suboptimal results caused by an unfavorable interaction due to a bad ordering in which the phases are applied. For example, during instruction scheduling the resource utilization is maximized by exploiting the available parallelism in the program. Unfortunately, this adversely interacts with register allocation, because more data needs to be available in registers.

1.4 Scope and Contributions

This work presents a novel processor description language called xADL that is based on the generic markup language XML. xADL targets a wide range of application-specific instruction processors including RISC, CISC, and VLIW class architectures. General superscalar processor implementations with dynamic scheduling are not supported, however, in-order issue architectures with out-of-order execution and out-of-order completion can be modeled.

xADL processor models primarily focus on the hardware organization, the language thus can be classified as a *structural* processor description language. In contrast to most traditional structural approaches, our language is intended to be applicable for all major application fields of classical processor description languages. In particular, good support for the automatic generation of software development tools such as the compiler, linker, and assembler was a primary design objective. But also the rather low-level tasks including simulator generation and hardware synthesis were considered as possible applications for our language. The main principles during the design of the language features were:

- **Flexibility** The xADL language was designed for a wide range of applications and architectures. This includes high-level and low-level tool generation, hardware modeling, as well as validation and verification tasks for a wide range of RISC, CISC, and VLIW processors.
- **Compactness** Short and readable specifications are easier to understand, maintain, and extend. It is thus important to provide powerful mechanisms to describe a possibly large number of instruction variants, the corresponding semantics and the hardware structure in a compact but still intuitive form. Redundant information should be avoided as much as possible.
- **Reusability** Processor designs are very often available in several variations that are largely compatible to each other but provide differing levels of performance, power, area, and cost trade-offs. xADL thus provides features to reuse and extend existing models, develop processor templates that can easily be adopted and modified, and build libraries of individual processor components.
- **Abstractions** Providing the right abstractions to the processor designer potentially simplifies the development and maintenance of processor models. In addition, these abstractions can radically simplify the tools that process the architecture models and thus improve the quality of all derived artifacts.

xADL specifications are comprised of four different sections: (1) the *configuration* section declares architecture parameters like the number of registers and functional units or the bit-width of the data path, (2) templates for assembly syntax, binary encoding, and ABI conventions are described in the *meta-information* section, (3) reusable blueprints of hardware components can be specified using *types*, (4) *instances* of the previously defined component types are finally interconnected to form the data path of the processor. The specification thus describes an abstract model of the processor's hardware organization. Individual instructions are not declared explicitly in our language, but instead are extracted from this structural model. The instruction set extraction follows very few simple rules and can efficiently be controlled by the processor designer. The resulting view of the instruction set is tightly coupled with the original structural model and provides both, an abstract notion of the instruction behavior and detailed information on how this behavior is

implemented in hardware. The combined information from both views allows a very flexible use of our language for low-level and high-level application scenarios.

In contrast to most contemporary processor description languages, we decided to follow the structural approach. Architecture designers often communicate their ideas using block diagrams and drawings of the hardware components and their interaction. Structural languages are conceptually very close to this approach and thus simplify the quick evaluation of ideas. However, structural languages also have drawbacks. Most notably these languages are not well suited for most high-level tasks such as compiler generation. The key principles during the design of the xADL language thus were the provision of proper abstractions and simplifications that ease the development of the processor models themselves and at the same time benefit the implementation of the software tools that process xADL models. The main contribution of this work can be summarized as follows:

- **Component-based Modeling** The processor description is composed from reusable components based on types. The user can build libraries from these types and can adopt and extend them for new designs using inheritance and generics, similar to classes and templates in C++.
- **Flexible Templates** Other parts of the specifications can also be reused across processor models. For example, templates that specify skeletons for the instruction encoding and syntax can be shared and reused. The instruction encoding specifications are very flexible and allow the modeling of various encoding styles, including variable-length and distributed variants.
- **Abstractions** Structural specifications usually describe the behavior of the individual hardware components in great detail. It is almost impossible to recognize common patterns for bypassing and forwarding as well as pipeline control. Our language provides abstractions to simplify the modeling of bypasses, pipelines, pipeline registers, and explicit communication between instructions for pipeline control.
- **Instruction Set Extraction** The instruction set of the processor is not specified explicitly, but is extracted from the structural specification using very simple rules. The instruction extraction can be controlled by the designer.
- **Single Specification** The instruction set extraction provides a behavioral view of the processor. Redundant specifications of the instruction behavior for simulation, synthesis, and compilation can thus be avoided. Only a single semantic specification of the instruction behavior is required.
- **Backend Generation** Highly optimizing backends for the open source compiler infrastructure LLVM and a proprietary compiler can be derived from xADL processor models. The derived compilers generate high-quality code that matches the performance of code generated by handcrafted compilers.

Speedups of up to 20% have been observed for individual benchmarks, on average a performance degradation of only 3%-15% has been measured.

- **Compiler Completeness** An important feedback for architecture designers is the completeness of the derived compiler, i.e., if the compiler is able to translate *all* valid programs accepted by the compiler frontend. A formal completeness test is presented that proves this property automatically. If our system is not able to guarantee completeness, counter examples are generated that guide architecture designer to improve the coverage or provide emulation routines for certain operations.

- **Generator backends** The feasibility of our approach has been demonstrated successfully by various other generator backends that allow high-quality simulation and development tools as well as hardware models to be derived from processor models.

The remainder of this work is organized as follows. First an overview of related work is given in Chapter 2, followed by a detailed description of the abstractions and language features of the novel xADL processor description language in Chapter 3. The design and implementation of the *adlgen* tool that reads and processes the architecture models is given in Chapter 4. Chapter 5 covers the compiler backend generator, and presents the background for a formal completeness test for instruction selector specifications derived from processor models. Experimental results that compare the code quality achieved by the derived compilers to handcrafted compilers are discussed in Chapter 6. The conclusion in Chapter 7 summarizes the insights that we gained during the course of this work and highlights some issues that we plan to tackle in future work.

2 Related Work

Processor descriptions languages enable the development of concise processor models, allow software development tools to be derived, and can help to solve the verification and validation problem during the design of new processors or ASIPs. Consequently, these languages are very attractive for researchers and the semiconductor industry alike. This chapter provides a short overview of processor description languages that have been in use or are currently used by both communities. Two languages are studied in detail: (1) the structural language *MIMOLA* [123, 124] and (2) the mixed approach followed by *EXPRESSION* [78, 88].

One of the most influential languages in the context of processor description is *nML* [67, 57]. Originally, nML was designed for processor simulation [121], but was later considerably extended. The nML language is today developed and maintained by Target Compiler Technologies,[2] a company specialized in the design of ASIPs and the automatic generation of software tools for these processors. The processor is modeled using a behavioral description of the instruction set. The specification can be structured using an attributed grammar [106, 107] in order to reuse common information among individual instructions or instruction groups. New instructions are specified using either *AND*-rules or *OR*-rules. AND-rules combine information provided by independent rules, while OR-rules allow to compactly enumerate instruction variants. The attributes of the grammar specify the instruction behavior, the assembly syntax, and the binary encoding. Other languages, such as *ISDL* [91] and *MADL* [158] have adopted the idea of grammars for the compact specification of instruction sets. In its latest form, nML [131] also includes a basic skeleton that defines the internal organization of the processor hardware. This skeleton provides information on hardware resources like register files, memories, caches, and functional units. Computations in the behavioral instruction specifications can refer to these resources, i.e., the computation is logically performed by the particular hardware resource. The language was also extended to model pipelining and provides nice abstractions for hazard resolution, stalling, and bypassing. nML is bundled with a retargetable C compiler *Chess* and a retargetable instruction set simulator *Checkers*. Further, hardware models in VHDL or Verilog can be derived using the HDL generator *Go*, along with assembly-level test cases generated by the test-program generator *Risk*. In particular the early work on retargetable compilation using Chess is well described [114, 56, 181, 182]. Programs that are compiled using Chess are represented using a *Control/Data-Flow Graph* (CDFG) [113]. Operations of the CDFG are matched with nodes of the instruction set graph (ISG), a representation of the target processor's instructions and storage elements [183, 182]. Instruction selection, register allocation, and instruction scheduling are performed on the CDFG representation. nML and its accompanying tools have successfully been used for designing ASIPs in the audio, video, and signal processing domain [167, 168, 184].

[2] `http://www.retarget.com/`

An equally mature framework has been developed for the *LISA* language [185, 94, 149] that has initially been developed for efficient processor simulation at RWTH Aachen, and has successfully been commercialized by the spin-off LISATek GmbH, which was later acquired by Coware. LISA has been adopted and extended by several other research institutions and companies [175, 154]. Instructions are composed of so-called *operations* that provide information on the behavior, the assembly syntax, and the binary encoding. Individual operations can be shared among different instructions and instruction variants in order to reduce the size of the processor model. The instruction behavior is described using C, C++ or SystemC. Choosing the right subset of these languages has great impact on the LISA platform tools, e.g., only a subset of SystemC can actually be synthesized to hardware. In any case, the use of these languages prohibits some high-level applications such as compiler generation. Ceng thus proposed an additional section to model the abstract behavior of instructions that can be used for the automatic generation of a compiler [31]. The LISA language supports a wide range of applications, including compiler generation [31, 186], customization of compiler optimizations [96, 97], efficient simulation using interpretation, compilation [148, 141], and partial native execution [68], instruction encoding optimization [142], hardware synthesis [170] as well as validation and verification [33].

An interesting approach has been proposed by Qin [155] for the *MESCAL Architecture Description Language* (MADL). The language adopts the idea of grammars found in nML for instruction specification and is based on *Operation State Machines* (OSM) to formally specify the execution of instructions. An OSM – basically an extended state machine – is assigned to every instruction, modeling its execution phases and resource allocation. On state transitions resources needed for the next execution phase are allocated and resources that are not required anymore are disposed. The resources are managed using *tokens* that are controlled by user-defined *token managers*. The token managers are specified outside the formal model of the language, e.g., using C/C++ for cycle-accurate simulation, and are not analyzable for the tools processing these specifications. Besides the *core* layer that specifies the OSMs as well as the instruction behavior, syntax, and encoding, the language provides an additional *annotation* layer. Annotations are used to express the missing information in an application-specific form, e.g., the resource model for instruction scheduling in the compiler requires additional annotations. The MADL system supports functional and cycle-accurate instruction set simulation [157, 156, 158] and partial generation of a compiler backend, including reservation tables for instruction scheduling and register specifications for register allocation [158].

The *BUILDABONG* framework [177, 59] is also based on a formal specification of the execution of instructions using *Abstract State Machines* (ASMs) [79]. Based on *XASM* specifications [177] of the target processor compilation and simulation tools can be extracted. Later, an XML-based *Machine Markup Language* (MAML) was added to model the hardware structure as well as the instruction behavior and timing. The behavior is specified in C, the user thus has to supply additional information for the code generator during compiler generation. MAML [112] provides

a rich set of interconnect primitives that can be used to model complex on-chip communication networks and processor arrays.

Akaboshi et al. present the *COACH* processor modeling system [6]. A register transfer model of the processor can be specified using the *UDL/I* hardware description language [102, 98]. A behavioral view of the instruction set is then *extracted* from this specification [7], and further used to retarget a compiler [5, 178, 4] and a fast instruction set simulator [4]. UDL/I can be classified as a hardware description language rather than a processor description language, the COACH system is thus restricted to a subset of the language.

Various other processor description languages have been proposed in the literature that are very often targeting a single specific purpose. The *FAST/ADL++* framework [131] is based on the UPFAST system [144] and targets the efficient simulation and modeling of complex instruction set processors such as IA-32 [18]. *ArchC* [166, 12] is a processor description language based on SystemC. It supports compiled simulation [17] and the automatic retargeting of the GNU Binutils[3] [16, 15]. ArchC models seamlessly integrate with external SystemC models via transaction-level modeling [70]. *ISDL* [85] also relies on grammars to specify the instruction set of a processor. The language can be used to retarget the *AVIV* compiler [86, 91], for processor simulation and hardware generation [87, 84]. The *HMDES* [81] language is an extended machine description language for the *IMPACT*, *Elcore*, and *Trimaran* compiler infrastructures. The assembly-level analysis and optimization framework *PROPAN* [104] relies on processor models specified using the *Target Description Language* (TDL) [105, 103]. The language is intended to capture properties of non-orthogonal architectures and derive constraints for aggressive code optimizations. The *Computer System Description Language* (CSDL) language [159] used by the *Zephyr* compiler [8] consists of three largely independent specifications: (1) *CCL* descriptions of procedure calling conventions [13], (2) *SLED*, a specification of instruction encodings [160], and (3) *λ-RTL*, a behavioral model of the instruction set. The combined information of these descriptions can be used to retarget the portable optimizer *VPO* [42]. Engel et al. present a processor description language [50] that allows the automatic customization of an assembler and simulator. It uses the C programming language to specify the instruction behavior and is thus limited to simulation. Moss and Walters propose several languages [139, 138, 188, 187] targeting the partial generation of a compiler as well as instruction set simulators. The languages adopt ideas from the Java programming language to structure the instruction set definition using classes.

Other systems are limited to model only variants of processors or add instruction set extensions to an existing configurable processor core. The *TIE* language [189] by Tensilica Inc., is a good example of such a system. The *Xtensa* [74] core is a configurable core that can be extended by additional registers, memories, I/O interfaces, and instructions. The structure of the base architecture limits the freedom of these extensions. Multi-cycle instructions, for example, can be modeled but are restricted

[3] `http://www.gnu.org/software/binutils/`

by the existing pipeline structure. The instruction extensions can only be used via *intrinsic* functions, i.e., the compiler does not consider these extensions for instruction selection automatically. The *TCE* [101] framework, developed at the Tampere University of Technology, is similarly specialized. It allows the customization of processor templates following the *Transport Triggered Architecture* (TTA) [39] approach. A TTA processor is described using a set of configuration and definition files that can be used to customize a simulator, a compiler, and synthesize hardware [101]. Another configurable architecture is the xDSPCore [109, 145]. Initially the processor's register file, memory interface, instruction buffer, pipeline organization and instruction set was configured using a simple configuration file. Later, a more powerful processor description language was designed that captures the complete processor and can be used to generate a compiling or interpreting simulator [54] and a compiler [55]. In addition, fragments of the processor reference manual can be embedded into the description. The accompanying tools rely on separate sections for every application scenario, the specified information is thus highly redundant and leads to large processor models. The language was successfully adopted by Catena,[4] a company specialized in the development of integrated circuits.

Other languages that are intended to model processors in some form can be found in simulation environments [179, 180, 10, 11, 137, 171] and compilers [63, 49, 38, 115]. A recent book by Prabath Mishra and Nikil Dutt provides an excellent introduction to processor description languages and their applications [131]. The book also covers most of the languages and systems presented here in more detail.

2.1 MIMOLA - A Structural Language

The *MIMOLA* language [123, 124] and its software systems (MSS) is one of the few well known structural processor description languages. It originated from architecture synthesis and microprogramming of the synthesized hardware blocks. Several generations of hardware synthesis tools [124, 126], compilers [125, 143, 127, 119, 128], and test program generators [110, 111, 20, 19] have been developed within the MSS. MIMOLA has been primarily designed for synthesis, however, the language semantics are also precisely defined for simulation. Later work even investigated the use of interpretation and compiled simulation [117] based on an instruction set abstraction that has been extracted from the processor model [129, 118]. The language and MSS tools are examined in detail in the following for two reasons: (1) the structural specification style is close to the approach considered in this work, (2) the idea of instruction set extraction [129, 118] developed for MIMOLA is adopted and simplified in this work.

The MIMOLA system offers two usage scenarios: (1) high-level architecture synthesis, and (2) retargetable compilation. For the first scenario, depicted in Figure 6(a), a hardware structure is synthesized from a user-supplied program. The

[4] `http://www.catena.nl/`

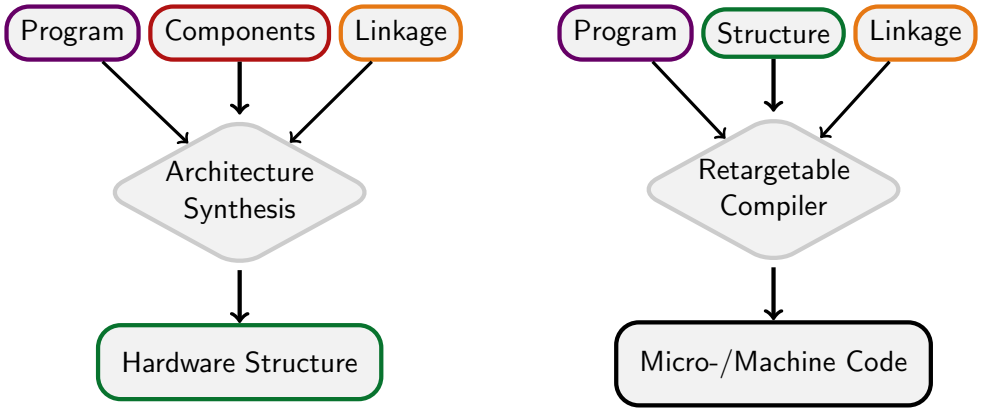

Figure 6: Two application scenarios for the MIMOLA system: (a) high-level architecture synthesis, (b) retargetable compilation.

program is specified in a *PASCAL*-like notation that is enriched with a definition of the available hardware components and hints that link the program to these hardware structures. It is important to note that both the linkage and the component specifications are possibly incomplete, i.e., not all expressions of the program are linked to a hardware component, and the interconnect between the hardware components is not yet fully specified. During synthesis a *complete* hardware structure is derived. In the general case the automatically derived hardware configuration is not optimal, the architecture designer thus can improve the derived hardware structure manually.

When an optimal architecture configuration has been found, the second usage scenario retargetable compilation depicted in Figure 6(b) is performed. The user-supplied program is mapped to the hardware structure such that at runtime the hardware implements the behavior specified by that program, i.e., the program is translated to micro- or machine code that can be executed by the given processor. The linkage information is again optional and is not required to be complete. However, the hardware structure is required to be complete. In contrast to most contemporary processor description languages, MIMOLA does not include any meta-information such as assembly syntax, instruction encoding, or calling conventions, it purely covers the hardware organization.

2.1.1 Program Specification

The specification of the program is an integral part of the MIMOLA language. The syntax of this specification is oriented towards the PASCAL programming language. The type system is simplified and basically consists of bit-vectors only. Specialized operators for data conversion and signed/unsigned arithmetic are provided. In addition, the `parbegin` and `parend` constructs for the explicit parallel evaluation of

```
program sum;
  type word = (15:0);        (* declaration of a 16-bit type *)
  var a : word at REG[4]; (* map 'a' to a hardware element *)
  var b : word;
begin
  a := 5;  b := 0;
  repeat
    (* explicit parallel evaluation *)
    parbegin  a := a - 1;  b := b + a;  parend;
  until a = 0;
end;
```

Figure 7: An example program definition in MIMOLA.

statements are available. Figure 7 depicts a simple example program definition.

2.1.2 Structure Declaration

The hardware resources that eventually will implement the program behavior are (partially) specified in the `structure` section using *modules*. Modules are templates for basic hardware blocks that explicitly specify a set of data and control ports. Each port is associated with one of the following modes: (1) `in` for input data, (2) `out` for output data, as well as (3) `inout` for bi-directional data signals. In addition, special purpose modes for control signals `fct`, address signals `adr`, and the clock `clk` are provided. The behavior of the module is specified through *exported operations* that are globally visible, but restricted in their structure, and locally visible procedures that are specified similar to the program behavior. Only the exported operations are considered during hardware synthesis and compilation to extract the control signals that trigger the desired behavior of the module. A module specifying combinatorial logic that does not require a clock is shown in Figure 8. The definition of clocked structures is similarly easy via the `at` keyword. MIMOLA does not provide abstraction to specify common constructs, such as registers and memories, and thus adopts a naming convention to recognize these specialized elements. Similarly, register *bypasses*, pipelining, and the instruction decoder need to be specified manually. MIMOLA does not provide abstractions for these constructs, which leads to complex models and further complicates the development of software tools that operate on the processor descriptions.

Instances of modules are finally declared in the *parts* section and connected to form a data path using the *connections* section. These specifications are optional and can be automatically completed using architecture synthesis. However, it is also possible to specify a complete data path and thus entirely control the hardware implementation.

```
module ALU( in a,b:word; fct s:(2:0); out c:(15:0), out d:word);
behavior is
conbegin
  c <- case s of     (* select an operation *)
    0 : 0;
    2 : a - b;       (* arithmetic operations *)
    3 : a + b;
    7 : a "XOR" b;   (* logical operations *)
  end;
  d <- a "AND" b;
conend;
```

Figure 8: An example module specification in MIMOLA.

2.1.3 Compiler Generation

A rich software framework was developed around the MIMOLA language, including several generations of synthesis tools, code generation algorithms, and test program generators. For the sake of brevity only the latest generation of the code generation tools, based on the *RECORD* compiler [119], will be described here. All previous code generation tools in MSS were based on pattern matching between graph and/or tree representations of the program behavior and the computational and storage resources available in the hardware [125, 143, 127]. These approaches are complex and computationally intensive and thus cannot be applied to large program specifications.

Leupers proposed to analyze the hardware structure off-line in order to obtain a model of the processor's instruction set [129, 118]. This abstract model allows the use of more efficient traditional code generation strategies, for example, tree pattern matching [49, 65, 64] during the instruction selection phase. The instruction set extraction is based on a graph representation of the hardware structure, i.e., the module instances and the connections between them. First the local behavior of modules is analyzed separately. Each assignment within a module is annotated with a triple that represents (1) the *destination* of the assignment, (2) an *expression*, and (3) a *condition*. The assignment to variable d in Figure 8 is annotated with the expression a *"AND"* b and a condition *true* that is always fulfilled, i.e., the assignment is unconditional. The assignments to variable c are in turn associated with the individual expressions on the right side of the `case` statement and associated with corresponding conditions that are derived from the corresponding `case` labels, e.g., the triple $(c, a + b, s = 3)$ would be derived. Other conditionals such as `if` statements are processed similarly. The conditions are represented using *Binary Decision Trees* (BDDs) to accurately and efficiently express bit-level constraints. Leupers distinguishes between four sources of conditions:

1. **I-conditions** refer to individual bits of the current instruction word.
2. **M-conditions** originate from certain machine states, e.g., the contents of registers or memory cells.
3. **D-conditions** are the result of dynamically computed expressions, e.g., comparison instructions or conditional branches.
4. **P-conditions** are temporary conditions that appear during the local analysis of the behavior of a single module, if the condition depends on one of the module's input ports. These conditions are later eliminated and replaced by either I-, M-, or D-conditions.

Second, μ-operations, i.e., instructions, are composed from these local assignments such that each μ-operation consists of an assignment that has the following properties: (1) the destination is a register, memory cell, or external output, (2) the arguments to the expression of the assignment only consist of registers, storage cells, external inputs, or constants, and (3) the conditions attached to the assignment only consist of I-, M-, and D-conditions. The local assignments computed for each module are *expanded* according to the connections specified in the structural section of the processor model. This expansion is applied to both conditions and expressions by recursively traversing the data path.

The instructions and their associated expressions that have been computed during instruction set extraction are further processed to customize the retargetable compiler RECORD. A major component of the compiler is the instruction selection phase that maps the target independent program representation of the compiler to processor specific instructions. The RECORD compiler relies on tree pattern matching based on tree grammars to perform this task [119]. A cost-optimal cover of the program's intermediate representation is calculated using a derivation of the tree grammar. A tree rule in the grammar consists of a pattern, a cost function, and an emit function. If a rule appears in the derivation of a given program the emit function is invoked and a corresponding machine representation is generated. The expressions associated with the μ-operation calculated during instruction set extraction can directly be converted to corresponding tree rules. The pattern is constructed from the operators, constants, registers, and memory cells of the expression using a simple mapping. Instruction set extraction only considers single-cycle μ-operations, the cost function thus always returns the constant value one. Deriving the emit function is more complex because various constraints are encoded within the conditions of a μ-operation. The emit function has to ensure that all conditions are satisfied in order to achieve the desired behavior of the instruction. The complexity of the problem arises primarily from M-conditions, because the compiler has to ensure that all referenced registers and memory cells are properly initialized. In addition to the tree rules derived directly from the instruction set of the target processor, specialized *start* and *stop* rules are added to ensure the correct matching

of register uses and assignments, these rules are associated with a constant cost function returning zero.

The idea of instruction set extraction and the subsequent simplifications of the compiler generator based on this technique are important contributions. However, the MIMOLA language was not designed for these techniques and thus lacks abstractions to simplify the processor modeling and extraction processes. Designers often implement a processor model incrementally, instruction by instruction. The algorithm for instruction set extraction in MIMOLA is too complex and hard to follow and thus impedes this intuitive approach. This is particularly true, if the algorithm fails to find the instructions that were specified by the designer. This unexpected behavior can easily lead to an unsatisfactory experience. The granularity of the extraction process is another problem. The algorithm extracts execution variants of instructions rather than instructions as such. Different μ-operations are extracted for equivalent operations that are logically a single instruction. For parallel architectures such as VLIW processors this can lead to considerable overhead. The algorithm is further restricted to instructions with a single output operand and a constant execution time of one cycle.

2.2 EXPRESSION - A Mixed Language

The *EXPRESSION* language [78, 88] is a typical mixed processor description language, i.e., a processor model consists of several views that capture the instruction set and hardware structure separately. Similar to MIMOLA, a rich set of generator and verification tools have been built based on EXPRESSION. Among them the retargetable compiler EXPRESS [89] and related code generation techniques [77, 174, 146], a retargetable simulation engine [163, 161, 162, 164], and a framework for compiler and processor verification and validation [133, 130, 132]. In addition, specialized tools for design space exploration are provided to explore restricted bypassing in pipelined processors [173], memory organizations [135], and system-on-chip configurations [136]. Synthesizable hardware models can also be generated from processor specifications [134].

The EXPRESSION system is based on the observation that a well-tuned hardware implementation alone is not enough to obtain optimal results. The compiler is equally important and plays a central role during design space exploration. The language thus was designed with strong support for the automatic customization of the compiler for the given processor model. The importance of the compiler is further emphasized by the term *Compiler-In-the-Loop* (CIL) that was coined by the researchers involved in the project [172].

Processor models in EXPRESSION consist of three major parts: (1) the behavioral instruction set specification, (2) a structural view of the processor, and (3) a specification of translation rules for the compiler. The integration of a dedicated compiler specification gives great flexibility to the architecture designer, however, it also burdens the engineer with the complexities of the compiler implementation.

```
(op_group ALUUnitOps
  (opcode xor
    (operands (SOURCE_1 int_any) (SOURCE_2 int_any) (DEST int_any))
    (behavior "DEST = SOURCE_1 XOR SOURCE_2")
    (asmformat ((cond "dst1=reg,src1=reg,src2=reg")
                (print "\t<opcode>\t$<dst1>,$<src1>,$<src2>\n")))
))
```

Figure 9: An example operation specification in EXPRESSION.

2.2.1 Instruction Set View

The instruction set is specified in EXPRESSION using two major abstractions, *operations* that roughly correspond to individual instructions of a typical RISC processor, and *instructions* that are composed of several parallel operations, very much like *instruction bundles* in today's VLIW processors. Operation descriptions capture the abstract behavior, the source and destination operands, the binary encoding and the assembly syntax. The assembly syntax can be augmented with an additional condition `cond` to describe different variations depending on the operand types. Restrictions on the operand types can elegantly be described using `var_groups`. An operand group maps a datatype and registers to a symbolic name that can be used as a short cut within the operation definitions. For example, the operand group *int_any* from Figure 9 that is used to hold integer values in a non-reserved general purpose register is defined as follows:

$$(int\_any\ (\texttt{datatype int})\ (\texttt{regs}\ GPRFile[1-28])).$$

Using the `op_group` keyword individual operations can be categorized into possibly overlapping groups. These groups are important for the assignment of operations to functional units within the hardware structure, and for the definition of instructions. Instructions are defined very similar to VLIW-bundles using a number of `slots` by enumerating the valid operations that can be encoded at the particular position of the instruction word.

2.2.2 Structural View

The hardware structure is specified in a separate section mostly independent from the actual instruction set architecture. The hardware resources in EXPRESSION are modeled using `units`, `ports`, `connections`, and `storage` elements. Functional units represent computational logic or control logic that conceptually implement the behavior of the previously defined operations. The valid operations are enumerated using the `opcodes` keyword. The main characteristics of a functional unit are the `capacity` and `timing`. The capacity specifies the number of operations that can be executed by the unit in parallel. The timing similarly specifies the number of

```
(ReadUnit READ
 (capacity 1)
 (timing (all 1))
 (opcodes ALUUnitOps)
 (latches (out ReadLatch))
 (latches (in DecodeLatch))
 (ports ReadPort1 ReadPort2)
)
```

Figure 10: A functional unit specified in EXPRESSION.

clock cycles that a particular operation occupies the unit. Storage elements, such as caches, memories and register files, are declared similarly using the `storage` keyword. The hardware resources are connected to each other via ports and latches. Latches roughly correspond to pipeline registers and are thus mainly used to connect functional units to each other. Ports, on the other hand, connect functional units to storage elements. In the case of register files, ports can be mapped to operands that were declared by the instruction set view of the processor. For example, the functional unit, defined in Figure 10, reads the two operands *SOURCE_1* and *SOURCE_2* for ALU-operations from the register file. The mapping is declared using an additional annotation to the unit's port:

```
(UnitPort ReadPort1("READ") (argument SOURCE_1) (capacity 1)).
```

However, it is not yet defined which port of the register file is to be used to perform the actual register read. The missing information is provided in the pipeline section. This section specifies the organization of the pipeline and assigns each unit to a pipeline stage. The pipeline model in EXPRESSION is very powerful and allows to specify nested pipeline structures and even superscalar pipelines. In addition, all valid *data transfer paths* are enumerated. The data paths specify the connections between the ports of functional units and storage elements. The data path concept is also used to specify bypasses and the organization of the memory hierarchy. Figure 11 depicts a simple pipeline and the connections of the read and write ports of the register file to functional units.

```
(pipeline FETCH DECODE READ EXECUTE WRITEBACK))

(dtpaths
 (type uni
  (GPRFile READ GPRReadPort1 ReadPort1)
  (WRITEBACK GPRFile WritePort GPRWritePort1)
 )
)
```

Figure 11: An example pipeline description in EXPRESSION.

2.2.3 Compiler Generation

A central component of the compiler-in-the-loop approach during design space exploration is a powerful retargetable compiler. In EXPRESSION a dedicated section is provided that specifies how program expressions in the compiler intermediate representation are mapped to the processor's operations. Example mappings are depicted in Figure 12. The keyword `generic` specifies a pattern in the compiler's intermediate language, whereas the keyword `target` denotes processor specific constructs. EXPRESSION distinguishes between two kinds of mappings: (1) the `operand_mapping` and (2) the `tree_mapping`.

In the case of operand mappings the `datatype` and `classtype` of the operand can be specified. It is thus easy to assign certain operand classes to specific register files. The operand mapping can also be used to partially specify the calling conventions using special classtypes for call parameters, return values, as well as the stack and frame pointer. The mapping of computations to machine instructions is described using dedicated tree patterns. Each mapping describes a possibly nested pattern of compiler specific expressions and a sequence of operations of the target processor. In addition, transformations of a sequence of processor operations into another sequence of processor operations can be specified. The user-supplied mappings are used to retarget the instruction selector of the EXPRESS compiler. Besides the instruction selector also the register allocator is customized and a resource model for instruction scheduling is extracted. EXPRESS supports, in addition to a traditional scheduling algorithm that uses resource tables [77], a highly specialized scheduling approach based on operation tables [174, 146] that is capable of exploiting irregular and partial bypassing of register values.

Although the explicit compiler specifications in EXPRESSION allow for more flexibility, this approach poses several problems. First, the user has to supply these

```
(operand_mapping
 (op_mapping (generic (datatype int) (classtype imm))
             (target int_immediate))
 (op_mapping (generic (datatype int) (classtype any))
             (target int_any))
)

(tree_mapping
 (
   (generic (ixor DST[1] = reg(1) SRC[1] = reg(2) SRC[2] = imm(3)))
   (target (xori DST[1] = REG(1) SRC[1] = REG(2) SRC[2] = IMM(3)))
 )
)
```

Figure 12: Compiler specifications in EXPRESSION.

patterns and is thus required to understand the internal organization of the compiler. The system offers little help besides the automatic extraction of the resource models for instruction scheduling, with respect to compiler generation. Second, these patterns duplicate information on the behavior of the processor. This imposes additional maintenance overhead and may lead to inconsistent specifications. For example, the pattern specifications account for more than 50% of the code lines of a MIPS-based architecture model that is available from the project website.[5] Another problem is that the compiler specifications assume a tree pattern based instruction selection scheme. It is impossible to automatically extract information that is suited for more sophisticated approaches from EXPRESSION models.

[5] `http://www.ics.uci.edu/~express/`

3 The xADL Language

The development of the xADL processor description language started as a simple configuration language for a highly clustered and parallel digital signal processor by OnDemand Microelectronics. Initially, the language was intended to specify only the computational resources of this processor, ignoring the control logic and pipeline organization. The objective was to customize the VHDL model of the processor and the software development tools based on coarse grained templates.

The language was soon extended to capture a complete processor implementation and today includes the assembly syntax, the binary encoding, the hardware structure, conventions imposed by the application binary interface, and, via *instruction set extraction*, the instruction set architecture. The main building blocks of xADL models are component types and interconnected instances thereof that model the internal organization of the processor. The language can thus be classified as a *structural* processor description language. However, conceptually, the hardware structure is only a means to express the processor's instruction set, which is automatically extracted. The instruction set extraction algorithm follows very simple rules and can be actively controlled by the processor designer. The instruction abstraction is thus a central part of the language and its tools. This approach combines the best from the traditional structural specifications and the traditional behavioral languages, while avoiding the problem of redundancy known from mixed languages. A *single* specification provides both, a very detailed view of the processor's internal organization and an abstract view of the instruction behavior.

The structural view lends itself to rapid prototyping by instantiating components from existing libraries and combining them with new, possibly application-specific, extensions. The *component-oriented* specification style of the xADL language closely resembles the typical design approach of hardware engineers. Ideas are often communicated using block diagrams and hierarchical drawings. These ideas can be expressed quickly and intuitively in our language. Moreover, modern hardware description languages, such as VHDL and Verilog, also follow this approach. This immediately simplifies the use of our language, because concepts known from these languages can be applied directly in a greatly simplified form. Only very few new concepts need to be learned in addition.

The instruction set view is similarly important in order to verify that all instructions and instruction variants are actually captured by the processor model and that these instructions are correctly modeled and implement the desired functionality. Strictly speaking this is of less importance in the case of application-specific processors. However, once families of processors need to be described that are, at least to some extent, compatible to each other, this becomes a valuable abstraction. An isolated view of individual instructions also facilitates their specification and discussions of their properties as well as discussions of the processor's instruction set architecture in general. The behavior of each instruction is completely independent

of other active instructions that are executed by the processor, except for very few exceptions, namely hazards and signals, that are explained below in more detail.

Providing the right set of *abstractions* is a key to simplify the specification of new processor designs, to keep processor models readable, and to enable the development of powerful tools. In contrast to other structural languages, xADL thus offers several key abstractions that are often useful, e.g., when a pipelined processor is to be modeled. xADL does not provide language features to specify the organization of pipelines explicitly. Pipelines are, similarly to the instruction set, implicitly declared via the hardware organization. A flexible abstraction is provided to model pipeline registers using *pipeline links*. Register *bypassing* is modeled compactly using *hazard links* that forward a register value between pipeline stages on a data hazard. Hazard links are not limited to bypassing, but may also be used to explicitly *stall* the pipeline or *ignore* a data hazard. Structural hazards are resolved implicitly based on the resources occupied by the currently active instructions. Typical operations to control the pipeline of a processor, such as flushing the pipeline or aborting individual instructions in the pipeline, can be realized using *signals*. Another helpful construct are *parallel instructions*, i.e., instructions that are not fetched from the instruction memory and are executed in parallel to the regular instructions on every cycle. Interrupts can be modeled nicely using this feature without complicating the description of other instructions [25].

Large portions of an xADL specification are *reusable* across different processor models and variations. All hardware components are instances of extensible *types* that can be shared and organized in component libraries. Types can be extended by *inheritance* and *generics* very similar to classes and templates in C++. The syntax and binary specifications are organized using so-called *templates* that can also be shared and reused. Typical parameters such as the width of the data path, the number of registers, or the number of functional units can be grouped in *configurations*. Deriving new processor designs and variations based on existing models is thus usually only a matter of a few lines of code.

The strong focus on reusability, combined with the abstractions provided by the xADL language, leads to *compact* and concise, but still readable, models. Redundant information is avoided, because only a single structural model is described. In particular, minor variations or additional parallel pipelines can be realized very efficiently.

The language and its tools, nevertheless, offer great *flexibility*. A wide range of processors can be described, including traditional pipelined CISC, RISC and VLIW processors. Even out-of-order features are supported to some extent, i.e., instructions are issued in-order, but can be executed and completed out-of-order. Reservation tables and reorder buffers of sophisticated superscalar processor implementations are very hard to model in an abstract language and are thus not supported. In practice this restriction is of less importance, because simple, deterministic, and area efficient processors dominate the embedded computing domain – superscalar implementations are rather rare. The combination of the structural view and the

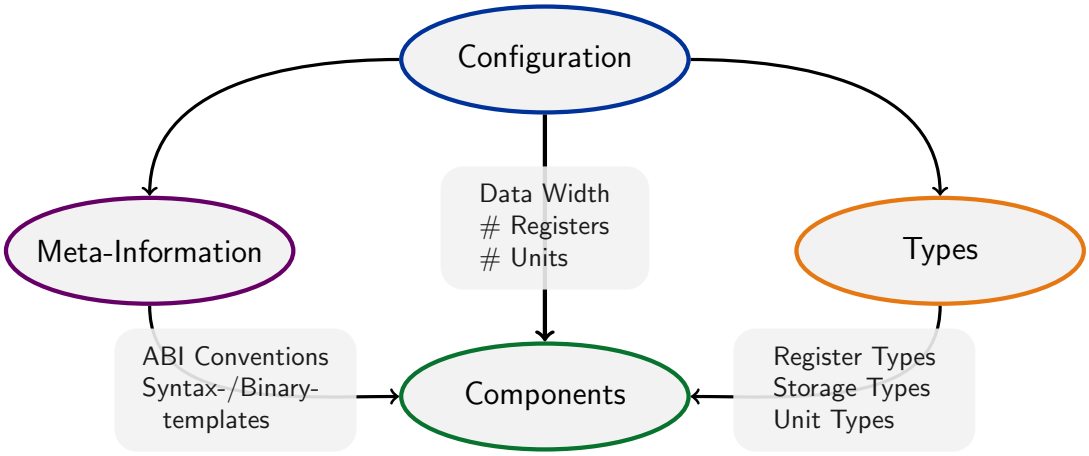

Figure 13: The four major sections of a xADL processor model.

extracted behavioral view enables the development of powerful tools. So far high-level tasks, such as compiler backend generation [26, 23] that is described in more detail in this work, are well supported. But also low-level tasks, such as instruction set simulation [24, 25], have successfully been realized. Even an early prototype for the generation of VHDL hardware models has been shown to be feasible.[6]

Processor models are conceptually divided into four major parts as depicted in Figure 13: (1) the *configuration* of the processor is specified using parameters such as the width of the data paths, or the number of functional units and registers. The programming conventions, the assembly syntax, and the binary encoding of instructions is covered by (2) the *meta-information*. (3) *Types* specify blueprints of functional units, register files, caches, and memories that can be shared and reused. The data path is finally composed of (4) *component instances* that are derived from previously defined types. The individual parts are described in more detail in the following sections using excerpts from existing processor models and additional examples. Note, however, that the xADL language is under active development and that individual constructs might change in the future. This description should introduce the basic ideas and concepts and should not be considered a complete reference. The examples are based on a MIPS model [147], a time-predictable RISC processor SPEAR [41, 62], and a four-way VLIW multimedia processor CHILI by OnDemand Microelectronics. Further details on these processors and the corresponding models are given in Chapter 6.

The xADL language is based on XML [27]. All following examples are enclosed in a gray box, XML tags are printed in green, XML attributes in red. All tags and attributes are part of the language and are formally specified using an XML schema [53]. Keywords, i.e., XML tags and attributes, as well as code fragments are highlighted in the text using a `typewriter` font.

[6] `http://en.wikiversity.org/wiki/Computer_Architecture_Lab/WS2007`

3.1 Configuration

Variations of an existing data path can be modeled using the `Parameter` keyword within a `Configuration`. A parameter is a symbolic `name` that is associated with a concrete `value` depending on the currently selected processor configuration. The parameter name is used instead of concrete values throughout the processor model, and later automatically replaced when the final value is known, i.e., during parsing. The body of the configuration tag is also available for arbitrary other declarations, such as type definitions and component instantiations. This allows, for example, to add functional units to an existing data path for a particular configuration.

```
<Configuration name="default" >
 <Parameter name="halfWidth_p" value="16" />
 <Parameter name="wordWidth_p" value="32" />
 <Parameter name="addresswidth_p" value="32" />

 <Parameter name="immIOWidth_p" value="12" />
 <Parameter name="immShWidth_p" value="5" />

 <Parameter name="registerCount_p" value="64" />
 <Parameter name="unitCount_p" value="4" />

 <!-- other configuration-dependent declarations ... -->
</Configuration>
```

Figure 14: Configuration section of the CHILI VLIW core.

Figure 14 depicts an example configuration taken from the CHILI VLIW processor model. Several parameters are defined, among them the bit-width of the data path used for computations and the width of address computations. The CHILI architecture restricts the width of some immediate operands, e.g., the number of bits available for shifting or I/O operations. The bit-width of these operands is also configurable, as is the number of registers and functional units.

3.2 Component Types

Component types are the central construct to define blueprints of hardware components in the xADL language. A type defines either a register file, a cache, a memory, or a functional unit. Concrete instances of the individual types are connected to form the data path of the target processor. Most types thus define *ports* that are used to connect the components among each other and to exchange data. Additional properties can be described depending on the kind of the hardware structure that is modeled by the type.

Types are reusable across different processor models and processor variants. It is also possible to organize common component types in libraries and construct processor models from these predefined blocks. Another distinguishing feature of the xADL language is the use of inheritance and generics to derive new types from existing ones.

3.2.1 Immediate Operands

Operands that are extracted from the current instruction word are modeled using *immediate types*. These types are very simple and basically consist of a `name` and a `width` in bits. In contrast to all other types, immediates do not define ports and are directly connected to the data path. Figure 15 shows two immediate types taken from the MIPS model.

```
<ImmediateType name="ImmJ_t" width="jumpWidth_p" />
<ImmediateType name="ImmW_t" width="wordWidth_p" />
```

Figure 15: Two immediate types of the MIPS model.

3.2.2 Register Files

Register files are defined using the `RegisterType` keyword. Each register file consists of a set of uniform base registers that all share the same bit-width. The size of the register file is specified using the `repeatcount` attribute, while the `width` attribute denotes the number of bits of each base register. The individual base registers, as well as sub-registers and register pairs, are accessible through ports. Each `Port` specifies a `width`, an `offset`, and the boolean flags `readable` and `writeable` indicating whether read or write operations are permitted. Certain base registers can be hardwired to a predefined value using the `Constant` keyword. Read operations to this register always deliver the given constant specified by the `value` attribute, while write operations are discarded.

```
<RegisterType name="R_t" width="dataWidth_p" repeatcount="32" >
 <Constant index="0" value="0" />
 <Port name="Rs" writeable="false" />
 <Port name="Rt" writeable="false" />
 <Port name="Rd" readable="false" />
</RegisterType>
```

Figure 16: Type of the general purpose register file of the MIPS core.

```
(1) <Port name="Hi" offset="16" width="16" using="Rs" />
(2) <Port name="Pair" width="64" using="Rs Rt" />
```

Figure 17: Modeling (1) sub-registers and (2) register pairs using register ports.

The register file definition of the MIPS architecture is shown in Figure 16. It consists of a writeable port `Rd` and two readable register ports `Rs` and `Rt`. The offset and width attributes are omitted, all ports thus operate on the complete content of the base registers. The base register at index zero is hardwired to the constant zero.

The offset and width attributes can be used to access bit-ranges of a base register in a register file. It is also legal for ports to be wider than the base registers. In that case two or more consecutive base registers are accessed through the wide port. The same applies if the sum of the offset and width attributes is larger than the width of the base register. This approach allows to elegantly describe sub-registers and register pairs as depicted by the examples in Figure 17. The first line shows a port to access a sub-register that covers bits 16 through 31, i.e., the upper half of a 32-bit base register. The second line of the example shows a port to access 64 bits. In the case of 32-bit base registers this corresponds to register pairs that span across two base registers.

Often ports are added to a register file merely to simplify the processor description – this often applies to ports that model sub-registers or register pairs. These *virtual* ports do not correspond to a hardware port of the register file. Instead, these register ports refer to other ports of the register file via the `using` attribute, which consists of a list of register ports that actually perform the access on behalf of the virtual port. For example, the ports from Figure 17 are both virtual ports that are not realized in hardware on their own. Instead, both refer to the ports `Rs` and `Rt` respectively that perform the actual accesses.

Register ports that are both, readable and writeable, can be used to model instructions that read and write the same register, e.g., the two-address instructions of the x86 architecture. The read and the write operations of such a port share the register index to address the same base register. This applies even if the two accesses are performed by different pipeline stages. For example, almost all instructions of

```
<RegisterType name="R_t" width="dataWidth_p" repeatcount="16" >
 <Port name="Rx" />
 <Port name="Ry" writeable="false" />
</RegisterType>
```

Figure 18: The Rx register port is read and then overwritten by the two-address instructions of the SPEAR processor.

```
<RegisterType name="R_t" width="32" repeatcount="64" >
  <Port name="n1" />
  <Port name="n0" />

  <!-- other register ports ... -->
</RegisterType>
```

Figure 19: Concurrent write operations to the same base register are resolved by the order of the register ports for the CHILI model.

the SPEAR processor are two-address instructions due to encoding constraints. The `Rx` register operand, depicted in Figure 18, is usually read and then overwritten by the same instruction

Register files do not define any control signals, enable lines, and even the clock signal is not specified. All these low-level details are omitted by the xADL language. Instead, register files are implicitly clocked. All other control signals are automatically derived from the hardware structure of the processor. This includes signals to enable and disable the individual read and write ports, as well as signals that serve as an index to address the proper base register.

A problem with register files are the semantics of concurrent accesses to the same base register. In the case of a read operation performed concurrently with a write operation, the written value is immediately visible at the read port, i.e., register files in the xADL language follow the *write-first* semantics. In the case of multiple concurrent write operations, the relative ordering of the involved ports is considered. The value of the port that is syntactically defined first is actually written to the register file. Figure 19 depicts the register file definition of the CHILI VLIW processor. It is perfectly legal for the CHILI processor to execute instructions that write to the same register via the ports `n0` and `n1` concurrently. However, only the value supplied by port `n1` is actually written, the other value is simply discarded.

3.2.3 Storage Elements

The xADL language provides two classes of storage elements, caches and memories. Both storage types are modeled in a very abstract form, because characteristics of memory hierarchies highly depend on the underlying technology and bus interconnects. The specification of the memory sub-system thus relies on user-supplied templates that are not part of the xADL language.

The characteristics of a memory can be declared using the `MemoryType` keyword. Currently, only the timing in the form of `min_delay` and `max_delay` attributes is captured by a memory type. The additional `type` attribute specifies a user-defined name of a template that selects the desired memory implementation. This information is ignored for the generation of retargetable compilers. However, other tasks

```
<MemoryType name="Memory_t" type="sram"
            min_delay="3" max_delay="10" >
 <Input name="writeB" datawidth="8" addresswidth="32" />
 <Input name="writeH" datawidth="16" addresswidth="32" />
 <Input name="writeW" datawidth="32" addresswidth="32" />
 <Output name="readB" datawidth="8" addresswidth="32" />
 <Output name="readH" datawidth="16" addresswidth="32" />
 <Output name="readW" datawidth="32" addresswidth="32" />
</MemoryType>
```

Figure 20: Definition of a memory type of the MIPS processor.

may make use of this information to select a particular memory implementation, e.g., during simulation or hardware generation.

The content of the memory is written via `Input` and read via `Output` ports. The `datawidth` attribute of a port species the number of bits that are accessed through the particular port. For every input and output port an implicit address port is generated. This implicit port is referred to by prepending an '`@`' symbol to the name of the corresponding port, which is often called *data port*. Address ports are considered to be input ports, i.e., information is supplied to the memory through these ports. The bit-width of the address port is defined using the `addresswidth` keyword. The two optional properties `baseaddress` and `alignment` restrict the range of memory cells that are addressable through the port. If the alignment is not supplied, addresses need to be properly aligned according to the `datawidth` attribute. Figure 20 depicts an example memory declaration taken from the MIPS model. It offers multiple data ports to access data of byte, halfword, or word size of the memory.

Cache types are declared in a similar fashion using the `CacheType` keyword. The only difference is that instances of caches are connected to a memory or another cache, where the data is fetched from in case of a cache miss. An example declaration of a cache type is depicted in Figure 21. The data cache of the CHILI is based on the template `odm-dms` and offers multiple read and write ports for concurrent accesses. Similarly to the MIPS memory, the addressable data elements can be of byte, halfword, or word size.

Similarly to register types, the relative ordering of the cache and memory ports is used to resolve conflicts due to concurrent accesses. However, in the case of storage elements both, input and output ports, are considered. For example, the CHILI processor allows multiple memory accesses to the same memory location to be executed concurrently. The memory model of the CHILI specifies that a read operation always retrieves the *original* value from the memory and *not* the value that is currently written. The output ports thus are specified before the input ports – see Figure 21.

```
<CacheType name="Cache_t" type="odm-dms"
           min_delay="1" max_delay="10" >
  <Output name="lb0"  datawidth="8" addresswidth="32" />
  <Output name="lb1"  datawidth="8" addresswidth="32" />
  <Output name="lh0"  datawidth="16" addresswidth="32" />
  <Output name="lh1"  datawidth="16" addresswidth="32" />
   ...
  <Input  name="sb0" datawidth="8" addresswidth="32" />
  <Input  name="sb1" datawidth="8" addresswidth="32" />
   ...
</CacheType>
```

Figure 21: Excerpt of the data cache definition of the CHILI processor.

For the current application scenarios of the xADL language, the template-based memory specifications are sufficient. However, work is in progress to replace them by a more sophisticated model that is closer to real memory organizations. The model will be based on buses that interconnect memories and caches. In addition, the internal organization of storage elements will be modeled more accurately.

3.2.4 Functional Units

Functional units play a central role in xADL specifications, because all the user-defined behavior is described using these hardware components. The keyword `UnitType` can be used to define two kinds of functional units. *Containers* encapsulate a complete data path or pipeline and may be comprised of instances of all component types, including instances of other unit types. Containers are intended primarily as a means to organize and structure the processor specification. Regular functional units, on the other hand, consist of a set of *operations* that specify the behavior of the functional unit and the instructions that it executes.

Functional units can be regarded as arbitrary *combinatorial* logic that performs some form of computation. `Input` ports supply the data for these computations, while `Output` ports hold the final results. Both kinds of ports are defined using a `name` and the `width` of the data that is transferred through the port. Output ports, in addition, can be associated with a `default` value that is supplied to the port if the implementation of the particular operation does not assign a value to the given port. If the default attribute is not specified by the user, the output port is considered *undefined*. Uses of undefined ports later on lead to an error message during parsing. A special default specifier for an output port is the `inactive` value. In contrast to undefined ports, it is perfectly legal to use inactive ports throughout computations. However, the computation and in particular all visible side-effects through registers or other storage elements are suppressed. As a default value, the inactive value suppresses unwanted side-effects, if the functional unit writes to several independent

```
<UnitType name="EX_t">
 <Input name="Rs_i" width="dataWidth_p" />
 <Input name="Rt_i" width="dataWidth_p" />
 <Input name="ImmJ_i" width="jumpWidth_p" />
 <Input name="ImmW_i" width="wordWidth_p" />

 <Output name="Rd_o" width="dataWidth_p" default="inactive" />
 <Output name="lo_o" width="dataWidth_p" default="inactive" />
 <Output name="hi_o" width="dataWidth_p" default="inactive" />

 <!-- list of operations or instances of sub-components ... -->
</UnitType>
```

Figure 22: Simplified type of the arithmetic unit of the MIPS processor model.

register files and/or storage elements that are not necessarily defined by all operations within the functional unit. Consider for instance the arithmetic unit of the MIPS processor depicted in Figure 22. The result of the divide and multiply operations of the MIPS processor model is stored to special registers using the `hi_o` and `lo_o` ports. All other instructions store the result to the general purpose register file via `Rd_o`. The inactive specifier disables the assignment to the respective other register file by default and thus simplifies the specification.

For container units a list of component instances follows the port declarations. Instantiating a component type will be explained later in Section 3.3 and 3.4, along with the very useful generics feature of the xADL language. The following paragraphs instead focus on regular functional units that specify operations.

Besides input and output ports, `Constant` and `Temporary` values can be declared within a regular functional unit. Constants are intended to define a hardwired value that can be used throughout the computations of an operation. Temporaries, as their name suggests, are symbolic names that represent intermediate results. It is important to note that temporaries do not correspond to memory cells, registers, or latches, i.e., the value is not stored, but merely associated with a symbolic name that can be referred to later on. Figure 23 shows the definition of the 32-bit constant value `exceptionAddr` and a temporary `tmp` taken from the MIPS processor specification. The constant specifies the location of the exception handler that is dispatched in case of an arithmetic exception, e.g., an overflow during an addition.

```
<Constant name="exceptionAddr" value="0x40000040" width="32" />
<Temporary name="tmp" width="32" />
```

Figure 23: Definition of a constant and a temporary within a functional unit.

move	cmove	sext	zext	trunc	
abs	add	sub			
and	or	xor	not		
rol	ror	shl	shr	ashr	
mul	mulu	mult	multu		
div	divu	rem	remu	divrem	divremu
ceq	cneq	clt	cltu	cle	cleu
cgt	cgtu	cge	cgeu		
use	signal	csignal	debug	decode	

Table 1: Built-in integer micro-operations available in xADL.

Operations describe the computations that are performed by a functional unit during the execution of an instruction. Multiple independent operations may be attached to a functional unit. But only one of them can be active at any given moment in time, i.e., a functional unit executes either a single instruction or is otherwise idle. The computations are described using a set of well-defined *micro-operations*. Each micro-operation corresponds to a basic operation that is performed on a set of input and output operands. Constants, temporaries, unit input ports, and the special value `inactive` can be supplied as input operands, while temporaries and unit output ports are valid output operands. The xADL language offers a rich set of built-in micro-operations, some examples are listed in Table 1. It is important to note that a micro-operation may define multiple output operands. For example, the built-in operation `add` from Figure 24 specifies three output operands: (1) the result, (2) an overflow flag, and (3) a carry bit. Similarly, the `mult`, `multu` as well as the `divrem`, and `divremu` built-ins provide two output operands to represent the full multiplication result in the first, and the division result and the remainder in the latter case.

The data representation of the operands of a micro-operation is defined as a simple bit-vector by the xADL language. The bit-vector may actually represent integer values as well as floating point or fixed point values, depending on the micro-operation. The components ports, immediates, constants and temporaries are thus essentially untyped. This also applies to the contents of register files and memories. Consequently, there is little room for sanity checks at the micro-operation level to ensure type correctness. However, this is very likely to change in future versions of the xADL language. We plan to extend the language by a more powerful type

```
<add d="<out-result>" o="<out-overflow>" c="<out-carry>"
     a="<in-operand1>" b="<in-operand2>" />
```

Figure 24: Definition of the `add` micro-operation.

```
<Operation name="nor" >
 <Body>
  <or  d="tmp"  a="Rs_i" b="Rt_i" />
  <not d="Rd_o" a="tmp" />
 </Body>
</Operation>

<Operation name="beq" >
 <Body>
   <shl d="tmp2" a="ImmW_i" b="const_2" />
   <add d="tmp" o="overflow" c="carry" a="pc_i" b="tmp2" />
   <ceq d="condition" a="Rs_i" b="Rt_i" />
   <cmove d="pc_o" a="tmp" b="inactive" cond="condition" />
   <csignal s="BEX" cond="condition" />
 </Body>
</Operation>
```

Figure 25: Example operations of the MIPS model.

system combined with type inference mechanisms. In fact, work on this topic has already started, but it is too early to report details in its current state.

The special value `inactive`, as before, suppresses side-effects that would be visible through registers or storage elements. However, in combination with the `cmove` built-in, the `inactive` value can be used to describe conditional updates of register values or memory locations. The `cmove` micro-operation selects between two input operands, `a` and `b`, depending on a condition `cond`, and assigns the selected value to an output operand `d`. If either `a` or `b` specifies the `inactive` value, the side-effects that would be visible through (possibly transitive) uses of `d` become conditional. This roughly corresponds to an additional enable line that is added to the data path. The line is derived from the condition associated with the `cmove` built-in and controls all register and memory ports that make use of a value that depends on `d`, i.e., uses `d` directly or uses the result of a computation that transitively depends on `d`'s value.

In addition to these computational operations, several other micro-operations are provided. The `debug` built-in prints intermediate results of arbitrary calculations to a file during simulation in order to facilitate the debugging of the behavior of individual instructions or instruction phases. A limited way of interaction between instructions that are currently active in the processor's pipeline are *signals*. The `signal` and `csignal` built-ins raise signals that are visible to all other active instructions; more details on signals are given in Section 3.5.3. The `decode` micro-operation represents the instruction decoder, i.e., a hardware component that analyzes the current instruction word and extracts the instruction's opcode and its operands. The immediate operands and indices of register operands are avail-

```
<Function name="complex_operation">
 <Input name="operand" />
 <Output name="result" />
 <Output name="result1" />
</Function>
```

Figure 26: Example of a user-defined micro-operation.

able only after decoding. The use of registers and immediates is thus restricted in pipeline stages preceding the decode stage, i.e., the stage containing the `decode` micro-operation. Similarly, functional units are restricted, because control signals are not available before the instruction is decoded.

The behavior of an operation is specified by the `Body` of the particular operation using a sequence of micro-operations. Figure 25 shows two example operations from the MIPS model. The first is a `nor` operation that calculates the negated logical or of the `Rs_i` and `Rt_i` ports, which correspond to register operands. A more complex MIPS instruction is described by the `beq` operation that implements a conditional branch. The branch instruction conditionally assigns a new value to the program counter via the output port `pc_o`. The branch target is calculated from the old program counter using `pc_i` and an immediate operand `ImmW_i`. If the values of the `Rs_i` and `Rt_i` register operands are equal, the program counter is updated. The assignment is disabled otherwise using the `inactive` value. In addition, the signal `BEX` is raised conditionally, which indicates that a branch has been taken and all instructions fetched so far from the wrong path need to be aborted.

The xADL language does not provide constructs for loops, if-then-else structures, or other complex control flow within operation bodies. In general these constructs are rarely needed for the specification of a processor – none of the processors that we have modeled so far require them. Nevertheless, in some cases additional flexibility might be needed. Therefore, user-defined micro-operations can be declared. However, these micro-operations are declared using a `name` and a set of `Input` and `Output` operands – see Figure 26. The performed computations are *not* specified, and need to be modeled externally. For example, the user-defined micro-operations are treated as black boxes during compiler generation and are thus only accessible through intrinsic compiler functions.

3.3 Component Instances

The actual data path of the target processor is composed from component instances that are instantiated from the previously declared types. In its simplest form an instantiation consists of a single line that provides the `name` and `type` of the component. An example of such a simple declaration is given in Figure 27. Here a new immediate `ImmJ` is instantiated based on the type `ImmJ_t`.

```
<Immediate name="ImmJ" type="ImmJ_t" />
```

Figure 27: Simple instantiation of an immediate type.

programcounter	
integer	float
base	index
status	configuration

Table 2: Categories to classify register instances.

Register instances are very often similarly simple, but may hold an additional `category` attribute. The category is a hint to tools that process the processor model. The backend generator, for example, relies on categories to assign data types to register classes and find translation patterns for the instruction selector. The supported register categories are listed in Table 2. The program counter of the processor, i.e., a register that is used directly or indirectly to fetch the next instruction word, is identified by the `programcounter` category. Only a single register can be assigned to this category. The other categories can be arbitrarily combined with each other. For example, the general purpose register file of the MIPS model, depicted in Figure 28, is assigned to the `integer`, the `base`, and the `index` categories. It may thus hold general integer values and may also serve as a base or index register for address calculations. Note that register categories are likely to disappear in future versions of the xADL language, when a more powerful type system is available that allows to derive this information automatically.

Cache instances are associated with a memory or another cache in order to retrieve data from there in case of a cache miss. The interfaces of the cache and the other storage element have to match, such that for every port of the cache a corresponding port is available to fetch and store data. Multiple caches may share the same parent, but only if the ports of these two caches are disjoint. Concurrent memory accesses to the parent through different caches are resolved according to the order of the parent's ports as described in Section 3.2.3. An example memory organization consisting of a data cache that is directly connected to a memory is depicted in Figure 29.

```
<Register name="R" type="R_t" category="integer base index" />
```

Figure 28: Register instance of the MIPS model.

```
<Memory name="Memory" type="Memory_t" />
<Cache name="DCache" type="DCache_t" memory="Memory" />
```

Figure 29: Memory and data cache instances of the MIPS model.

The computational resources of the target processor are finally derived from unit types. Similarly to other instantiations, a `name` and a `type` are required to define a new functional unit. The optional attribute `repeatcount` allows to instantiate multiple identical copies from the same unit type. In particular for regular parallel processors, such as most VLIW architectures, these copies reduce the specification overhead significantly.

Functional units and storage elements specify connections to other components using *data links* and *pipeline links*. Data is transferred along these links from one component instance to the next. In addition, *hazard links* can be used to resolve data hazards, caused by data dependencies in a pipelined processor. Hazard links do not necessarily transfer data, but can also be used to force the pipeline to stall. A detailed discussion of data, pipeline, and hazard links is given in Section 3.5.

```
<Unit name="EXE" type="EXE_t" repeatcount="unitCount_p" >
 <!-- connections to other components ... -->
 <!-- hazard specifications ... -->
</Unit>
```

Figure 30: Instantiation of multiple identical units of the CHILI VLIW processor.

3.4 Inheritance and Generics

A distinguishing feature of the xADL language is the use of inheritance and generics to derive new component types from existing types. This is particularly useful in the case of functional units, when new operations or additional units should be added to an existing processor model that is otherwise not changed.

Inheritance allows to derive a new type from *multiple* existing component types. The base types can be extended by additional ports, or, in the case of unit types, by new operations or new component instances. If multiple base types are specified, the new type consists of the union of the features of the base types, i.e., the union of the ports, operations, and instances of all base types. Name conflicts lead to an error message during parsing, if the conflicting features of the base types are not compatible to each other. For example, unit input ports are compatible only if their bit-width is equal, et cetera. Compatible features are collapsed and not duplicated for the new type.

```
<UnitType name="DSP_EX_t" extends="EX_t" >
 <Input name="Ac_i" width="AccuWidth_p" />
 <Output name="Ac_o" width="AccuWidth_p" default="inactive" />

 <Operation name="madd" >
  <Body>
   <mul d="tmp" a="Rs_i" b="Rt_i" />
   <add d="Ac_o" a="Ac_i" b="tmp" />
  </Body>
 </Operation>
</UnitType>
```

Figure 31: Extending the arithmetic unit of the MIPS model by a DSP multiply-accumulate instruction.

An example that demonstrates the use of inheritance is given in Figure 31 and 32. Here the MIPS model is extended to support additional instructions of the application-specific extension MIPS DSP. First, the `HILO_t` register file is extended by an additional 64-bit port `Ac` representing accumulators – see Figure 32. Secondly, the arithmetic unit is extended by additional ports to read and write the accumulator registers and a `madd` operation that implements the corresponding multiply-accumulate instruction as depicted in Figure 31.

The generics features of the xADL language are similarly intended to reduce the overhead when new processor variants are to be derived from existing components and processor templates. The approach is very similar to templates in C++, but is, due to the scope of the xADL language, limited to static hardware structures. Generics are restricted to container units to define templates of processors, pipelines, or partial data paths. A generic unit encapsulates component instances that are already interconnected. Some of them are instantiated from a fixed predefined type, others refer to a type argument, and the concrete type is left open. Type arguments are defined using the keyword `Generic` using a `name` and a `base` type. It is possible to derive instances from a generic type argument, with the only difference that the type argument of the instantiation is not a concrete type but a generic type argument. These instances can also be connected to other components according to the interface of the respective base type.

```
<RegisterType name="ACCU_t" extends="HILO_t" repeatcount="8" >
  <Port name="Ac" width="AccuWidth_p" using="hi lo" />
</RegisterType>
```

Figure 32: Extending a register file of the MIPS processor by an additional accumulator port.

```
<UnitType name="MIPS_PIPELINE_t">
 <Generic name="HILO_g" base="HILO_t" />
 <Generic name="EX_g" base="EX_t" />

 <Register name="HILO" type="HILO_g" />

 <Unit name="EX" type="Ex_g" >
    ...
 </Unit>

 <!-- other component instances ... -->
</UnitType>
```

Figure 33: A generic container unit type modeling the MIPS pipeline.

Later, instances can be derived from the generic unit type itself. However, for an instantiation the generic type arguments have to be assigned to concrete types using the `Generic` keyword. Its `name` attribute specifies the type argument while the concrete type is specified using the `select` attribute. The base type and all its sub-types derived via inheritance are legal assignments for a generic type argument. It is possible to define nested structures of generic types.

Figure 33 shows an example based on the MIPS processor model. A generic container unit is declared that specifies the complete pipeline structure. The arithmetic unit `EX` as well as the `HILO` register file are instantiated from type arguments. The arithmetic unit is based on the unit type `EX_t` while the register is based on the register type `HILO_t`. When a concrete component is to be instantiated from the generic unit, these type arguments need to be assigned to concrete types as depicted in Figure 34. In this example a MIPS variant with DSP extensions is modeled based on the previously derived types `ACCU_t` and `DSP_EX_t`.

```
<Unit name="MIPS-DSP" type="MIPS_PIPELINE_t" >
 <Generic name="HILO_g" select="ACCU_t" />
 <Generic name="EX_g" select="DSP_EX_t" />
</Unit>
```

Figure 34: Instantiating a generic container unit.

3.5 Composing Data Paths

The data path of the processor is composed from component instances that are interconnected using three kinds of links: (1) *data links*, (2) *pipeline links*, and

(3) *hazard links*. Regular data links correspond to wires between a port of a functional unit or storage element and an immediate or another port of a register file, storage element, or functional unit. Data is transferred along the data link, but is not buffered or stored in latches. These links thus connect blocks of combinatorial logic within a single pipeline stage. Pipeline links on the other hand explicitly model a pipeline register that stores the data for the following cycle. The data is then processed by the other component during that cycle, i.e., the data is transferred from one pipeline stage to another. *Hazard links* specify how data hazards are resolved, either by *bypassing*, *stalling* the pipeline, or by explicitly *ignoring* the hazard. Hazard links thus not only represent data wires but also control logic to detect and resolve the hazard. Finally, *signals* can be used to resolve control hazards using asynchronous communication between instructions that are currently active in the processor pipeline.

3.5.1 Data and Pipeline Links

Data and pipeline links are specified using the `Connect` keyword within the body of an instantiation of a functional unit or storage element, as shown in Figure 35. A connect directive specifies a group of links that connect ports of the instance to ports of other components or immediates. An input connect does so using the `Input` keyword for input ports only, while output connects, specified using the `Output`

```
<Unit name="UCD" type="UCD_t" repeatcount="unitCount_p">
 <Connect>
  <Input input="Rn_i" select="EXE[current].Rn_o"
         stageboundary="true" />
 </Connect>
 <Connect>
  <Input input="Rn_i" select="MUL[current].Rn_o"
         stageboundary="true" />
 </Connect>
 <Connect>
  <Input input="PC_i[0]" select="BRA.PC_o"
         stageboundary="true" />
 </Connect>

 <!-- output connections ... -->

 <Hazard output="Rn_o" type="forward"
         select="EXE.Rx_i EXE.Ry_i EXE.Rn_i MUL.Rx_i ..." />
</Unit>
```

Figure 35: Unit instantiation including connections to other components.

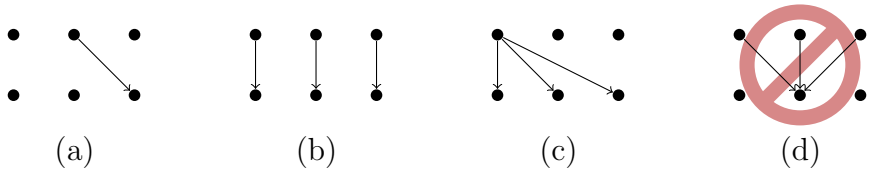

Figure 36: Example connection patterns that can be realized using the `Connect` keyword.

keyword, are restricted to output ports only. The `input` and `output` attributes specify an input or output port of the current instance, while the `select` attribute specifies a port of another instance or an immediate. The select attribute consists of the name of a component followed by the name of the particular port separated by a dot, e.g., `BRA.PC_o`. Links that are part of an input connect are restricted to guarantee a well formed processor model. Every input port is the target of at most one link in a connect to avoid ambiguous assignments. In addition, the source of a link has to be either an immediate, a readable register port, the output port of a storage element, or the output port of a functional unit. It is forbidden to form cycles using input connects. Output connects are restricted even further. The target of an output link has to be either a writeable register port or the data respectively address port of a storage input port. In contrast to input connects, the links of an output connect can specify the same output port multiple times.

Data and pipeline links can be distinguished by the boolean attribute `stage-boundary`. If this attribute is assigned the value `true` (or alternatively `yes` or `1`), the link is interpreted as a pipeline link and a corresponding pipeline register is implicitly created. It is legal to mix pipeline and regular data links in a connect.

A unit instantiation may in fact consist of multiple identical copies, a `Connect` thus specifies the links for *all* of the copies at the same time. It is rather rare that the connections of all copies are identical. Usually, the connections follow a regular pattern, e.g., the nth copy is connected to the nth copy of another unit instantiation, or only the first copy is connected to another component. The select attribute as well as the port specifier can thus be restricted by appending a subscript. The subscript is either a number, a number range, or the special keyword `current` enclosed in brackets. The subscripts are a powerful construct to enumerate various connection patterns as depicted in Figure 36. Assuming unit instantiations that define three identical copies each. The connection patterns to connect the input ports of the lower units to the output ports of the upper units can be modeled using the following subscript pairs: (a) ([2], [1]), (b) ([`current`],[`current`]), (c) ([`current`],[1]), and (d) ([1],[`current`]). The first member of each pair denotes the subscript of the input port, while the second denotes the subscript of the select. Note that connection variant (d) is illegal, because the assignment to the involved input port is ambiguous.

```
<Connect>
 <Input input="Odds_i" select="R.R[0-16/odd]" />
</Connect>
```

Figure 37: Connect to read from a register file restricted by a modifier.

Links to and from register files can be further restricted, by appending a second subscript to the name of the register port, e.g., `R.Rd[31]`. The subscript specifies that only a subset of the registers in the register file should be accessible through the link. This is useful to model hardwired register operands or model other restrictions imposed by the instruction set of the processor, i.e., limited space to encode full register indices. The subscript may either be a number, a numeric range, or one of the keywords `all` and `current`. In addition, a modifier may be appended that restricts the selection to either `odd` or `even` register indices. The modifier is separated from the rest of the subscript by a '`/`' symbol – see Figure 37.

A complete example of an unit instantiation and its connections is presented in Figure 35. The functional unit `UCD` of the CHILI processor forwards the `Rn_i` and `PC_i` value of unconditional arithmetic instructions and unconditional branches to the next pipeline stage. As can be seen, the `repeatcount` attribute is specified, we thus assume that multiple identical copies of the unit are created from this instantiation. The copies are connected to the `EXE` and `MUL` units using the connection pattern depicted in Figure 36(b), while the port for the program counter is connected to the branch unit `BRA` using the pattern from Figure 36(a).

3.5.2 Hazard Links

The example in Figure 35 also shows a set of hazard links that specify how data hazards are to be resolved with respect to the `Rn`, `Rx`, and `Ry` register operands of CHILI instructions. Three kinds of hazard links are currently supported by the xADL language: (1) `forward` links model the bypassing of register values from one pipeline stage to another in order to hide execution latencies, (2) `stall` links do not carry data, but instead cause the instruction that reads from or writes to the port at the head of the link to wait until the respective other instruction has completed its current operation, and finally (3) links that are attributed with the hazard type `ignore` do not imply any action whatsoever. The last class of hazard links is primarily intended for documentation and verification purposes. It indicates that in a particular situation data hazards should simply be ignored. This may lead to unpredictable behavior and usually implies that software development tools need to ensure correct program execution by avoiding data hazards. Often, however, processor designers are willing to accept this restriction in order to save silicon area or improve the maximal clock frequency. In order to detect whether a given hazard link triggers, the involved register files and in particular the register indices need to

be known. The processor designer is not required to explicitly supply this information, it is automatically derived from the connects and hazard links of the processor model.

A major problem with bypassing are instructions that are executed concurrently either due to pipelining or due to the explicit parallel execution style of VLIW processors. In the former case, bypassing is generally well-defined. Instructions that were issued later, i.e., are currently executed in an early pipeline stage, are assigned higher priority. Almost all pipelined processor implementations today follow this definition, the xADL language thus also adopts this behavior. In the latter case, the conventions adopted by different processors available today vary. It is thus not possible to apply a predefined fixed solution without restricting the scope of the language. A similar problem has already been discussed in Section 3.2.2 for concurrent register writes. There, the priority is derived from the syntactical ordering of the respective register ports. The same approach is taken for hazard links that trigger concurrently. If `forward` and `stall` links trigger at the same time, the instruction has to wait, i.e., stalls are assigned a higher priority than bypasses.

3.5.3 Signals

Pipeline and data links lead to very compact and concise processor specifications and allow to elegantly model the structure of the processor's pipeline. Data hazards that possibly emerge can be handled using hazard links, and structural hazards are resolved automatically using a resource model that is derived from the component instances. Using these primitives a large number of real processor architectures can already be modeled. However, primitives to describe the handling of control hazards have not yet been described.

In contrast to data and structural hazards, control hazards often originate from a small class of instructions that need to directly control the data path and its operation. A typical example of such instructions are branches and jumps, but also traps and instructions that cause exceptions sometimes affect the data path globally. The data path and the pipeline are not represented as such in the xADL language, thus control needs to be specified in a distributed fashion. Signals provide a way to communicate asynchronously between these currently active instructions. A signal corresponds to a global single-bit control line in hardware that causes individual instructions in the pipeline to *abort*. The control line can be asserted for a single cycle using the `signal` and `csignal` micro-operation. The `AbortSignal` keyword specifies which instructions are aborted by a given `signal`. It can be attached to functional unit types, unit instances, and individual operations. An asserted signal causes instructions to be aborted immediately when they are executed by a functional unit that is marked with this keyword.

Figure 38 shows the `DE` unit that models the decode stage of the MIPS processor. Branches raise the `BEX` signal in order to abort instructions that have been fetched

```
<Unit name="DE" type="DE_t">
 <!-- connects and hazard links -->

 <AbortSignal signal="BEX" />
</Unit>
```

Figure 38: A signal aborts the instructions in the decode unit of the MIPS model in case a branch has been taken.

along the execution path that was not taken. The example presented in Figure 25 in Section 3.2.4 shows the use of the `csignal` micro-operations to implement a conditional branch instruction.

3.5.4 Parallel Pipelines

It is of course possible to model several parallel operating pipelines using pipeline links. The xADL language does not impose any restrictions on the structure of these pipelines, except that all pipelines that execute *regular instructions* are required to start at a single root. Instructions that are fetched from memory, decoded, and then executed by one of the pipelines are considered regular. The root is typically the fetch unit that controls how instructions are fetched from memory and issued to the different pipelines. Due to this restriction, it is not possible to model superscalar out-of-order issue pipelines. However, instructions that are executed by different parallel pipelines may be executed and even completed out-of-order.

Arbitrary additional pipelines may be defined that execute so-called *parallel instructions*. In contrast to regular instructions these are neither fetched from memory nor decoded. Parallel instructions are instead issued on every cycle implicitly. The use of register operands and immediates, as well as the structure of the functional units implementing the behavior of the parallel instructions, are thus restricted, because control signals cannot be extracted by the instruction decoder. Parallel instructions are intended for rather simple, repetitive tasks that always perform the same operation on every cycle. A cycle counter that increments a particular register on every cycle, or a dispatcher that invokes an interrupt service routine on an asynchronous interrupt are examples of typical uses. Parallel instructions are equivalent to regular instructions during execution. In particular, hazard links as well as signals can be used without restrictions.

3.5.5 Restricting Data Paths

The processor data path, composed of functional units, storage elements, and registers, specifies how instructions are to be implemented. Sometimes, however, the use

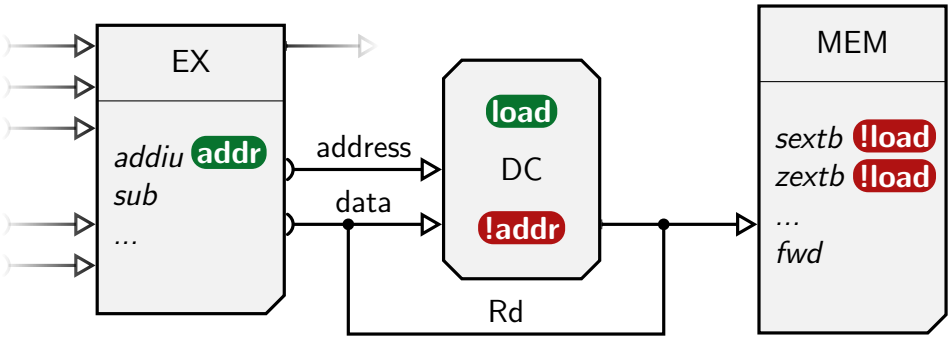

Figure 39: The addressing modes of the MIPS processor are restricted using predicates and conditions.

of the data path is restricted by external constraints that are not immediately visible from the the structural representation. For example, the result of the arithmetic unit of the MIPS processor calculates the address to perform memory accesses. In principle all operations of the arithmetic unit could be used to form an addressing mode. However, the MIPS architecture restricts the valid addressing modes to a single operation that adds a register and a 16-bit immediate. This restriction is not imposed by limited capabilities of the hardware. Figure 39 illustrates the connection between the arithmetic unit and the data cache. The arithmetic unit supplies the data as well as the address to the data cache. The data cache may also be bypassed, the arithmetic unit is then directly connected to the `MEM` unit.

The xADL language provides *predicates* and *conditions* to model such constraints in a very simple and intuitive way. A `Predicate` is a simple boolean flag that indicates that a certain property is met. `Conditions`, on the other hand, verify whether a predicate has been defined. Both conditions and predicates can be defined in the body of a unit type, an operation, a unit instance, or storage instance. In addition, data links can be augmented with both constructs. Consider the example given in Figure 39. The MIPS data path is restricted using the `addr` and `load` predicates that are only provided by the `addiu` operation of the `EX` unit and the data cache respectively – see Figure 40. The data cache requires the `addr` predicate to be defined, and thus disallows other operations to supply addresses. The xADL code for the cache instance and its condition is depicted in Figure 41. Similar conditions

```
<Operation name="addiu" >
 <Predicate name="addr" />
 <!-- operation body ... -->
</Operation>
```

Figure 40: The `addiu` operation of the MIPS arithmetic unit defines a predicate to restrict the addressing modes.

```
<Cache name="DCache" type="DCache_t" memory="Memory" >
 <Condition predicate="addr" />
 <Predicate name="load" />
</Cache>
```

Figure 41: The data cache of the MIPS processor model restricts the valid addressing modes using a condition and a predicate.

are attached to the sign- and zero-extend operations (`sextb`, `zextb`, ...) of the `MEM` unit. Only data loaded from the cache can be extended, all other instructions merely forward the result of the computation using the `fwd` operation.

3.6 Meta-Information

The third major part of an xADL processor specification is the meta-information section that describes the instruction syntax, the binary encoding of instructions, and the programming conventions of the application binary interface. These conventions may depend on the environment such as the operating system and development tool that are used in combination with the processor model. For example, two assembly variants are commonly used for the x86 architecture. The AT&T-style is adopted by most Unix-based operating systems, while the Intel-style is common on the DOS/Windows platform. It is thus possible to specify multiple variants of the processor's meta-information in an xADL model. The descriptions are, in addition, reusable within a processor model, but also across different processor descriptions.

3.6.1 Assembly Syntax

The instruction syntax is specified using the `SyntaxFormat` keyword that groups definitions of a particular assembly language. For a processor model multiple syntax formats can be defined, each format is assigned a `name` that is used to select a particular assembly language for the corresponding programming conventions. A syntax format is defined using four keywords: (1) a `SyntaxDirective` defines common assembly directives, (2) a `SyntaxMaskDirective` defines masks to extract bits from symbols and symbolic expressions, (3) the `SyntaxMapping` keyword maps register indices to assembly syntax representations, and (4) a `SyntaxTemplate` defines a blueprint of the assembly notation of an instruction or group of instructions.

Syntax directives specify strings that allow to customize common assembly directives that may vary slightly from processor to processor. Typical examples are directives to define a label, start a comment, or switch the current section of the output file. The assembly directives are largely oriented towards the GNU Binu-

```
<SyntaxFormat name="mips">
  <SyntaxDirective name="comment" syntax="#" />
  <SyntaxDirective name="bundle-end" syntax=";" />

  <SyntaxMaskDirective syntax="%lo" mask="0xffff" />
  <SyntaxMaskDirective syntax="%hi" mask="0xffff0000" />

  <!-- syntax mappings and syntax templates ... -->
</SyntaxFormat>
```

Figure 42: Definition of syntax directives and masks of the MIPS processor model.

tils,[7] most directives thus provide a fallback to the GNU conventions. Usually only the following three directives need to be supplied: `bundle-start`, `bundle-end`, and `comment`. Processors that follow the VLIW approach adopt a convention to specify instruction bundles, i.e., groups of instructions that are executed in parallel. Two directives, `bundle-start` and `bundle-end`, are provided to specify the start and end marker of bundles. Also the syntax of comments often varies between processor architectures and can be specified using the `comment` directive. Directives are specified using a `name` and an attribute `syntax` that consists of a plain text string. Example directives of the MIPS processor model are depicted in Figure 42.

Related to syntax directives are syntax mask directives that are used in conjunction with symbols or symbolic expressions that need to be split in order to fit into immediate operands. The problem arises during linking when the address of a particular symbol is determined. The symbol can be placed at an arbitrary location, the address may thus exceed the valid value range of immediate operands. A mask directive specifies the assembly notation and a mask to split the address of a symbol properly to fit certain immediate operands. The `syntax` attribute specifies the notation, while the `mask` attribute defines a bit-mask that is applied to the symbol's address in order to perform the splitting. In addition, the masked address is shifted such that the lowest bit of the mask is aligned at bit location zero. Consider for example the MIPS architecture, it defines two syntax masks `%hi` and `%lo` that extract the upper and lower half of the symbols address respectively – see Figure 42.

Syntax mappings specify a mapping of register indices to a corresponding assembly notation. A mapping is specified using a `name` and a list of `Map` definitions. Each of these definitions in turn associates a set of registers, specified by the `select` attribute, with a textual representation defined by the `syntax` attribute. The select attribute consists of a list of numbers and range specifications separated by blanks. Following a range specification, an optional modifier (`/odd` or `/even`) can be specified that restricts the selection to either odd or even registers only. The `syntax` is specified as plain text intermixed with a placeholder '`%d`' that represents

[7] `http://www.gnu.org/software/binutils/`

```
<SyntaxMapping name="gpr_m" />
 <Map syntax="zero" select="0" />
 <Map syntax="at"  select="1" />
 <Map syntax="v%d" select="2-3" step="1" />
 <Map syntax="s%d" select="23 16 17-22" />
 <Map syntax="t%d" select="24-25" offset="8" />
 ...
</SyntaxMapping>
```

Figure 43: A syntax mapping assigns symbolic names to register indices according to the MIPS naming conventions.

the value of a counter variable. The counter variable represents the current register index, that is possibly transformed using the `offset` and `step` attributes. The `offset` attribute is simply subtracted from the current counter value, while the `step` attribute controls the increment of the counter when ranges are processed, i.e., the counter is incremented by `step` for every register in the range. An example mapping that follows the MIPS naming convention for the general purpose registers is shown in Figure 43. The register indices 24 and 25, for example, are mapped to the names `t16` and `t17` respectively. Syntax mappings are not applied automatically, but only if the mapping is referred to by a *syntax binding* as will be explained below.

Syntax templates specify blueprints of the actual assembly syntax of the individual instructions of the processor. Each template consists of a `name` attribute and plain text that is intermixed with placeholders called `Token`. Tokens are in turn associated with a `name`, but do not specify any additional information. The actual fragment of assembly code that is represented by the token is later specified

```
<SyntaxFormat name="spear2">
 <SyntaxTemplate name="op2_s">
  <Token name="op" /><Token name="cond" /> 
  <Token name="op1" />, <Token name="op2" />
 </SyntaxTemplate>

 <SyntaxTemplate name="op1_s">
  <Token name="op" /><Token name="cond" /> 
  <Token name="op1" />
 </SyntaxTemplate>

 ...
</SyntaxFormat>
```

Figure 44: Syntax templates of the SPEAR processor model.

```
<Operation name="ldli" syntax="spear2.op2_s" >
 <Syntax syntax="op2_s.op" value="ldli" />
 <Body>
   <move d="Rx_o" a="Imm8s_i" />
 </Body>
</Operation>
```

Figure 45: The mnemonic of SPEAR's *load immediate low* instruction is specified using a syntax binding.

using *syntax bindings*. Figure 44 shows the syntax format of the SPEAR processor, including its syntax templates. Tokens named `op` are placeholders for the instruction's mnemonics, while the operands of the instructions are encoded using the `op1`, `op2`, and `opf` tokens. Almost all SPEAR instructions can be executed conditionally depending on a flag. The `cond` token is used to append a condition code to the instruction opcode.

In addition to the syntax format specification, the individual instructions need to be associated with a syntax template and concrete values need to be assigned to the respective syntax tokens. So-called *syntax bindings* establish a connection between the structural processor model and the syntax templates. Instructions are not represented as such in the xADL language, the syntax bindings are thus attached to the operations of functional units or to the definition of links using the `Syntax` keyword. The `syntax` attribute specifies the names of the syntax format, the syntax template, and the token separated by a dot, e.g., `spear2.op2_s.op` denotes the `op` token of the `op2_s` template of the `spear2` syntax format. The format and template names are optional and can be omitted, the binding is then simply applied to all tokens that match the name. The binding specifies either a plain text string using the `value` attribute that replaces the token, or binds an operand to the token. The latter is only possible, if the binding is attached to a link that connects a register port or an immediate to the data path. In the case of immediates, numeric values as well as symbolic expressions are accepted by the assembler at the position of the token. For register operands a number that represents the register index is accepted. Additionally, if a syntax mapping is specified by the optional `mapping` attribute, the symbolic names defined by the mapping are accepted. Figure 45 and 46 present

```
<Input input="FP_i" select="FP.p" >
 <Syntax syntax="opf" mapping="fp_m" />
</Input>
```

Figure 46: The syntax of operands is specified with the link that connects the operand to the data path.

examples of both syntax binding variants. The first example specifies the mnemonic of the *load immediate low* instruction of the SPEAR processor. The second example shows a binding that involves an operand and a syntax mapping. The `opf` token of all syntax templates is bound to the register operand `FP.p`.

Figure 45 also shows an important attribute that is added to the XML tag of the `ldli` operation. So far only values and operands have been bound to syntax tokens, however, the connection between syntax templates and instructions has not yet been established. This is the purpose of the `syntax` attribute of the `Operation` keyword. It specifies that instructions that use the particular operation follow the syntax format specified by the given attribute. The syntax of the `ldli` instruction in the example thus follows the `spear2.op2_s` syntax, defined by the syntax template `op2_s` of the `spear2` syntax format. The name of the syntax format is again optional and can be omitted.

3.6.2 Binary Encoding

The binary encoding of instructions largely follows the scheme described in the previous section for the assembly syntax. A `BinaryFormat` consists of three parts: (1) `BinaryTemplate` definitions, (2) `BinaryMapping` specifications, and (3) a `Bundle` definition that specifies the final layout of the encoded instructions or, in the case of VLIW processors, the layout of multiple instructions in a bundle. Binary templates and binary mappings correspond to the respective counterparts presented for the assembly syntax. Binary templates also consist of `Token` definitions, but, in contrast to syntax templates, the layout of the tokens with respect to each other is not defined. A binary template thus corresponds to an unordered tuple of tokens, where each token has a `name` and a `size` in bits. The actual layout is described using the `Bundle` keyword. This separation of concerns greatly simplifies binary encoding specifications, especially, for complex encoding variants of modern VLIW

```
<BinaryFormat name="mips" >
 <BinaryTemplate name="rtype_b" >
  <Token name="op" size="6" />
  <Token name="rs" size="5" />
  <Token name="rt" size="5" />
  <Token name="rd" size="5" />
  <Token name="shamt" size="5" />
  <Token name="funct" size="6" />
 </BinaryTemplate>

 <!-- more binary templates and bundle layout ... -->
</BinaryFormat>
```

Figure 47: Binary template for the MIPS *rtype* instruction format.

```
<BinaryMapping name="some_mapping_m" />
 <Map select="17-31/odd" step="1" offset="17"/>
</BinaryMapping>
```

Figure 48: Binary mappings specify a space efficient encoding of register operands.

processors. The encoding information of individual instructions is separated from the specifications of other instructions and the final layout of the instruction bundle. Figure 47 presents an excerpt from the binary format specification of the MIPS processor model.

The binary mapping serves a similar purpose as its counterpart for the syntax specifications. But instead of text strings numeric representations are mapped. This allows the specification of compact encodings for register operands. The example presented in Figure 48 shows a mapping for a register operand, where only odd indices in the range 17 through 31 are valid. By default the full register index would be encoded, which would be space inefficient. The mapping shown here maps the original register indices into the range 0 through 7.

The binding between fixed encoding values, encoding mappings, instruction operands, and instructions also follows the conventions of the syntax specifications. *Binary bindings* are similarly attached to operations and links that connect register or immediate operands using the `Binary` keyword. Its `binary` attribute specifies the name of a binary token (the template and format names are again optional). The `value` attribute specifies a fixed constant that replaces the token. If the binding is part of a link, the register or immediate operand is bound to the token. A simple example is depicted in Figure 49.

The final layout of the binary tokens to form instructions or instruction bundles is specified using the `Bundle` keyword. The layout specification is very flexible and supports most common encoding conventions, including variable-length instruction and bundle formats. Also the encoding schemes adopted by most VLIW processors, as described by Fisher et al. [61], can elegantly be modeled. For example, the variable-length bundle format used by the CHILI processor is a combination of

```
<Operation name="addu" binary="rtype_b">
 <Encoding binary="rtype_b.funct" value="0x20" />
 <Body>
  <add a="Rs_i" b="Rt_i" d="Rd_o" o="overflow" c="carry" />
 </Body>
</Operation>
```

Figure 49: The binary representation of the *add unsigned* instruction of the MIPS processor is specified using binary templates.

```
<Bundle>
 <Fields>
  <Field name="instr" alignment="32" position="0" />
 </Fields>

 <Layout>
  <Choice>
    <BinaryTemplate template="rtype_b" >
     <Append field="instr" tokens="op rs rt rd shamt funct" />
    </BinaryTemplate>
    <!-- binary templates ... -->
  </Choice>
 </Layout>
</Bundle>
```

Figure 50: Final layout of the binary encoding of the MIPS processor.

an uncompressed and a fixed-overhead encoding scheme. But also distributed and template-based encoding schemes can be specified using our approach. The layout relies on two basic concepts: (1) *fields* and (2) *regular expressions*. Regular expressions specify which instructions can be combined to form a legal bundle, while fields determine the relative position of tokens in the encoded instruction or bundle.

Explicitly enumerating all possible instruction combinations is a cumbersome and tedious task, especially for highly parallel processors. We thus use regular expressions to compactly represent this information.[8] However, instructions are not represented explicitly in xADL, it is thus impossible to refer to instructions directly. Fortunately, every instruction is associated with a binary template, the layout is thus defined via binary templates. The regular expressions are represented using nested XML tags, very similar to complex data types in XML schema specifications [53]. The `Choice` keyword selects a variant out of multiple alternative binary template, choice, or sequence definitions. It corresponds to the usual pipe operator ('|') in regular expressions. The `Sequence` keyword represents a sequence of binary templates, choices, or sequences, and corresponds to the usual star operator ('∗'). The length of a sequence can be restricted using the `min` and `max` attributes that specify the minimal and maximal length of the sequence. Binary templates are usually referenced at the innermost level of choices and sequences, but may also appear intermixed with the other keywords. A binary template is referenced using the `BinaryTemplate` keyword, where the attribute `template` selects a binary template of the current binary format.

The layout is finally specified by assigning the tokens of the binary template to a field. A field represents a dedicated space within the instruction word that

[8] Bundles are typically finite in length, the expressiveness of regular expressions is thus sufficient for this purpose.

```
<Fields>
 <Field name="head" alignment="32" />
 <Field name="body" follows="head" alignment="32" />
 <Field name="tail" follows="body" alignment="32" />
</Fields>
```

Figure 51: Binary encoding using multiple fields.

is intended to encode certain parts of the instruction word or bundle. Multiple fields can be defined, however, simple single-issue processors usually do not require more than one field. In the case of VLIW processors, multiple fields are usually required, if the encoding of individual instructions of the bundle is intermixed. Fields are defined by a `name` using the `Field` keyword. The relative position of fields is determined using the optional attributes `follows`, `alignment`, and `position`. The `follows` attribute specifies another field that proceeds the current field in the encoding. This option can be combined with the `alignment` attribute in order to enforce an alignment, e.g., if the preceding field is of variable length. The `position` attribute specifies an absolute offset relative to the instruction or bundle start. It can be combined with the `alignment` attribute only if the position is zero, it then specifies the alignment of the complete instruction word or bundle. The `position` and `follows` attributes are mutually exclusive. An additional attribute, `padding`, specifies a bit-pattern that is used to fill gaps between mis-aligned fields.

The tokens of a binary template that is referenced by a choice or sequence definition are assigned to a `field` using the `Append` keyword and its `tokens` attribute. Multiple `Append` definitions are allowed within a template reference that can even assign tokens to different fields. The tokens are appended at the end of the field, it is thus not possible to intertwine the tokens of independent instructions in a single field, which sometimes renders the use of multiple fields unavoidable.

Figure 50 shows the bundle specification of the MIPS processor model. The MIPS is a single-issue processor, where the encoding of different instructions is independent. A single field and a single choice that enumerates all binary templates is thus sufficient to specify the instruction format. An example showing the use of multiple fields is given in Figure 51. Consider a binary template with three tokens `opc`, `a`, and `b` that is referenced by a single sequence of length two. The tokens are

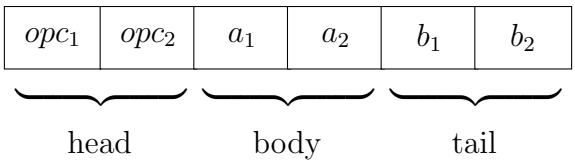

Figure 52: The instruction format defined by a dual-issue processor.

appended to the fields `head`, `body`, and `tail`. The resulting instruction format is depicted in Figure 52.

3.6.3 Programming Conventions

The `ABI` keyword is used to specify the programming conventions of a processor, which cover usage conventions for registers, function calling conventions, the stack layout, and the binary as well as the syntax format. A processor description may specify multiple ABI's in order to model the conventions adopted by different operating systems and development environments.

The register usage conventions are defined by the `Registers` keyword. It is used to assign registers to predefined or user-defined register classes; the predefined register classes are listed in Table 3. The register classes for the return address, the stack pointer, and the frame pointer are singleton classes, i.e., only one register can be assigned to each class. The return address class is mandatory and specifies a register that stores the original program counter on a function call. The stack pointer specifies a register that can be used to address memory locations of the current stack frame; it is also mandatory. The definition of a frame pointer is optional, but enables the use of variable-sized stack frames, e.g., the use of the C function *alloca*. Registers that are assigned to the `Reserved` class are not available for general use, i.e., the compiler is not allowed to modify these registers. User-defined register classes may be defined using the `UserClass` keyword.

The other classes specify the usage of registers during a function call. The `CalleeSaved` register class specifies the registers that have to be saved by the callee, i.e., the called function has to save and restore these registers. On the other hand, the `CallerSaved` registers can freely be used by the called function. The caller is responsible to save and restore these registers. The `Argument` and `Result` register classes specify registers that can be used to pass function arguments and result values between the caller and callee during a function call. The order of the registers is important for these two classes, because the argument and result values are assigned to the registers in the specified order.

It is very hard to provide a language that allows to model function calling conventions without prior assumptions. To our knowledge, *none* of the processor description languages available today are able to provide such a general model. The

Reserved	ReturnAddress
StackPointer	FramePointer
Result	Argument
CalleeSaved	CallerSaved

Table 3: Predefined register classes to specify register usage conventions.

```
<ABI name="chili-elf" >
 <Syntax select="chili" />
 <Binary select="chili" />

 <StackLayout arguments="last_fixed_on_stack" results="register"
              alignment="32" direction="down" />

 <Registers>
  <ReturnAddress select="R[63]" />
  <StackPointer select="R[62]" />
  <FramePointer select="R[61]" />
  <Result select="R[0] R[1]" />
  <Argument select="R[1-6]" />
  <CallerSaved select="R[15-54]" />
  <CalleeSaved select="R[7-14]" />
  <Reserved select="R[55-60]" />
  <UserClass name="GVR" select="R[55-60]" />
 </Registers>
</ABI>
```

Figure 53: Programming conventions of the CHILI processor model.

xADL language thus also offers only a limited set of parameters that can be used to customize the function call sequence. This includes the argument and result conventions, as well as the stack layout. In addition, the register usage conventions specify the use of registers during a function call as described above. The `StackLayout` keyword allows to parametrize the stack organization and the use of the stack during function calls.

As a general rule arguments are passed using the registers denoted by the `Argument` class. If the number of function arguments exceeds the number of suitable registers in this class, the remaining arguments are passed on the stack. Similarly, arguments to C functions with variadic argument lists[9] are passed in registers until the number of argument registers is exceeded. The remaining arguments are again passed via the stack. These conventions can be customized using the `arguments` attribute by the following options: (1) `register` specifies the default behavior, if possible, arguments are passed using registers, (2) `fixed_register` is similar to the default convention, however, variadic arguments are always passed via the stack, (3) `last_fixed_on_stack` further restricts the argument conventions for variadic function, the last fixed argument is already passed on the stack, and finally (4) the `stack_only` convention enforces that all arguments are passed on the stack. Similar options are available for return values using the `results` attribute.

[9] functions using '...' in the argument list, e.g., printf.

In addition to the argument conventions, the stack organization has to be specified. It is assumed that a dedicated register is available to be used as a stack pointer as defined by the `StackPointer` class – a frame pointer is optional. The stack layout can be controlled using the `alignment` and `direction` attributes. The former species the alignment of a stack frame for a function, while the latter specifies the direction of growth, i.e., if the stack grows `up` or `down`.

Further, the syntax and encoding format is determined using the `Syntax` and `Binary` keywords respectively. Figure 53 presents the programming conventions adopted by the CHILI processor model, which follow the typical conventions of a RISC processor. A return address register, stack pointer, and frame pointer are available. Return values and arguments are typically passed in registers, except for functions with a large number of arguments or variadic arguments. Almost all registers are available for computations, only registers 55 through 60 are reserved.

3.7 Instruction Set

As has been noted already, the instruction set is specified implicitly by the structural xADL processor description. It is thus not possible to specify instructions as such within the processor model. However, the instruction set abstraction is important during the development of an application-specific processor. It summarizes the processor's capabilities and helps during the discussion and development of additional processor features and instruction set extensions. At first the approach taken by the xADL language may appear to be contradictory to this idea. But, on a second look the structural processor model is only a means to specify instructions in a compact form, similar to the AND/OR grammars used by many behavioral languages. This section introduces how instructions are modeled using the xADL language. A simple algorithm that is based on the idea of *instruction set extraction* automatically extracts the instruction set from the processor structure.

Instruction set extraction is, in fact, not a new method. It has already been applied to structural models specified using the UDL/I [7] and MIMOLA [129, 118] languages. However, the extraction algorithms in these systems are overly complex and virtually impossible to follow for a processor designer. The problem, in both systems, is that not only the structure but also the behavior of the structural elements has to be considered to find instructions. Our approach in contrast follows very simple rules. Only components and their connections need to be considered to extract *instruction paths* from the processor's pipeline. Along these instruction paths the final instructions are created from the operations attached to the functional units that comprise the path. *Illegal* instructions can be excluded using *conditions* and *predicates*. If, at some point, instructions are not derived as expected by the processor designer, it is easy to trace the problem down by first examining the instruction paths and then the conditions and predicates.

The instruction set extraction operates on a hypergraph [21] representation of the processor. A hypergraph $\mathcal{H} = (V, E)$ consists of a finite set V of vertices and a finite set E of *hyperedges*, where each hyperedge $e \in E$ is a non-empty subset of V, $\emptyset \subset e \subseteq V$. Regular graphs are a special case of hypergraphs, with $\forall e \in E\colon |e| = 2$. A directed hypergraph can be defined using directed hyperedges, which are ordered pairs (X, Y), where $X \subseteq V$ and $Y \subseteq V$ are possibly disjoint sets of nodes in V. The first member of a hyperedge $e = (X, Y)$ is referred to as $head(e) = X$, while $tail(e) = Y$. Figure 54 depicts a simple hypergraph with $E = \{e1 = \{v3, v5, v6\}, e2 = \{v1, v2, v3\}, e3 = \{v4, v5\}\}$ and a variation of that graph using directed hyperedges $\{e1 = (\{v3\}, \{v5, v6\}), e2 = (\{v1, v2\}, \{v3\}), e3 = (\{v5\}, \{v4\})\}$.

Definition. The set $\mathcal{I} = U \cup S \cup R \cup I$ of component instances consists of the union of unit instances U, cache and memory instances S, register instances R, and immediates I.

Definition. The ports of component instances are denoted by the set P. The input and output ports of units, as well as the data and the implicit address ports of storage elements are directly represented in P. Immediates are, for the sake of brevity, also treated as ports, thus $I \subseteq P$. Register ports that are both readable and writeable are duplicated in P and considered separate ports.
Every port in P is associated with a component instance. The functions *pin* and $pout : \mathcal{I} \to \mathcal{P}(P)$ specify the writeable ports and readable ports of an instance respectively. The following ports are considered writeable: input ports of units and storage elements, implicit address ports of storage elements, and writeable register ports. Readable ports are: unit and storage output ports, immediates, and readable register ports.
Writeable register ports as well as the address and data ports of storage input ports are called sinks. The function $sinks : \mathcal{I} \to \mathcal{P}(P)$ returns the set of sinks of an instance.
The following properties hold for the functions from above: $pin(i) \cap pout(i) = \emptyset$ and $sinks(i) \subseteq cin(i)$ for all $i \in \mathcal{I}$.

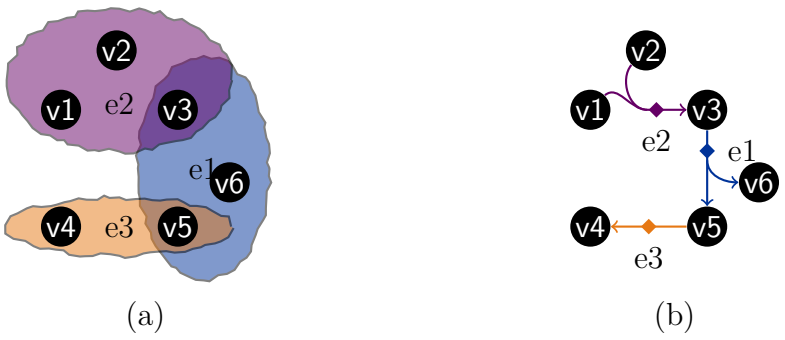

Figure 54: Examples of (a) a hypergraph and and (b) a directed hypergraph.

Definition. Data and pipeline links are represented as a multi-set of ordered pairs $L \subseteq P \times P$. Every link is a member of a connect in $C \subseteq \mathcal{P}(L)$. The input and output connects are in turn associated with a functional unit or storage element by the functions *cin* respectively $cout : U \cup S \to \mathcal{P}(C)$. For every unit and storage element at least one output connect is defined, i.e., $\forall i \in U \cup S\colon cout(i) \neq \emptyset$, but $cout(i) = \{\emptyset\}$ is allowed.

A standard multi-graph is induced by P and L that could be used to represent the processor structure. However, this graph is incomplete and cumbersome to handle. For example, a traversal of this graph is not immediately possible because intermediate connections are missing, e.g., between the input ports and output ports of a unit. We will show how to derive hyperedges for a hypergraph from the input and output connects that simplify the handling of the processor representation.

Definition. Directed hyperedges are constructed from the input and output connects of a unit or storage instance $i \in U \cup S$ as follows:

$$\begin{aligned} hin(i) = \{(X,Y) \mid \exists c = \{(u_1,v_1),\ldots,(u_n,v_n)\} \in cin(i) :\\ X = \{v_1,\ldots,v_n\}, Y = \{u_1,\ldots,u_n\}\} \\ hout(i) = \{(X,Y) \mid \exists c = \{(u_1,v_1),\ldots,(u_n,v_n)\} \in cout(i) :\\ X = \{u_1,\ldots,u_n\} \cup pout(i), Y = \{v_1,\ldots,v_n\} \cup pin(i) \setminus sinks(i)\}. \end{aligned}$$

Note that the direction of hyperedges derived via *hin* is reverted with respect to the original links, whereas the edges derived using *hout* are not. Also note, that the internal behavior of a unit or storage element is not relevant for the construction of hyperedges. However, the behavior is approximated by unifying the head and the tail of the hyperedges constructed from output connects with the input and output ports of its parent instance.

Definition. A directed hypergraph $\mathcal{H}_p = (V, E)$ that represents the data path of a processor can now be defined:

$$\begin{aligned} V &= P, \\ E &= \bigcup_{i \in U \cup S} (hin(i) \cup hout(i)). \end{aligned}$$

Due to the restriction of connects defined in Section 3.5.1, the duplication of register files, and the special handling of storage input ports by *hout*, the hypergraph $\mathcal{H}_p$ representing a processor structure has to be free of cycles. If a processor specification violates these restrictions it is rejected and an error is reported.

Figure 55 depicts the hypergraph constructed from a simple processor model, consisting of functional units `U1`, `U2`, and `U3`, a cache `C`, an immediate `Imm`, and the register files `R1` and `R2`. The center of each hyperedge is represented by a diamond

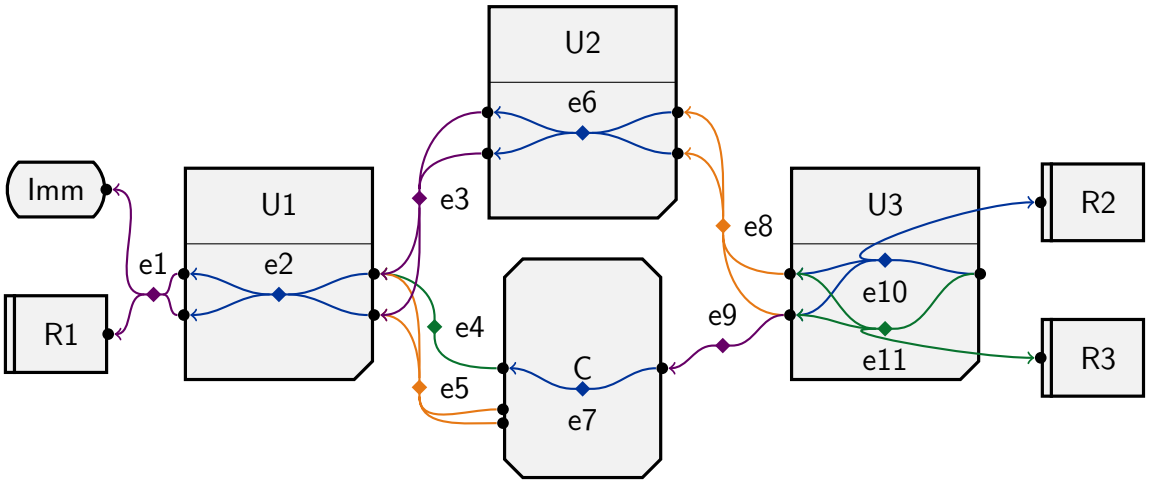

Figure 55: Example of a simple data path represented by a directed hypergraph.

symbol. For hyperedges corresponding to output connects the center lies within a functional unit, whereas hyperedges of input connects are located outside. Unit `U1` reads the operands from a register file and an immediate. The cache is used for load and store operations, the data and address for stores are supplied by an input link represented by hyperedge $e5$, while the address of load operations is supplied by $e4$. Unit `U2` processes the output of unit `U1`, while unit `U3` either operates on data loaded from the cache or calculated by unit `U2`. The results are finally stored in one of the register files `R2` and `R3`. Most hyperedges in the figure correspond to input connects, except for $e2$, $e6$, $e7$, which are constructed from empty output connects, as well as $e10$ and $e11$, which are derived from non-empty output connects. Note that edge $e7$ does not point to the two lower ports of the cache, these ports represent the data and address ports of the cache's input port and are thus sinks.

It is easy to see that paths can be constructed using the hyperedges of the graph. The reverted paths then represent the flow of instructions through the processor pipeline, we will refer to these paths as *instruction paths*. The following section defines instruction paths and shows how to discover them in a processor model.

3.7.1 Instruction Paths

The hypergraph is an abstract representation of the instruction flow through the data path of the processor. Each hyperedge represents a transition of an instruction from one component instance to the next. The graph, however, does not represent all features of the processor model, in particular hazard links and signals are not present. Even the internal computations of functional units are merely approximated by extending the hyperedges of output connects. However, these details are not needed for the definition of *instruction paths*, which can be seen as an intermediate step of the instruction extraction procedure.

Definition. The hyperedge $p \in E$ is called a predecessor of $e \in E$, denoted by the predicate $pred(e, p)$, if $head(e) \cap tail(p) \neq \emptyset$.

Definition. A hyperedge e of a hypergraph that represents a processor structure $\mathcal{H}_p = (V, E)$, $e \in E$, is an endpoint, denoted by $endpoint(e)$, if one of the following conditions holds:

$$(1)\ \nexists p \in E\colon pred(e, p),$$
$$(2)\ \exists r \in R\colon tail(e) \cap pin(r) \neq \emptyset.$$

An endpoint in the original processor structure is a functional unit or storage element where instructions possibly retire, i.e., the execution of the instruction is completed. Generally, only very few endpoints exist in a processor model. Furthermore, these often match particular instruction classes, e.g., memory store instructions retire at some cache or memory, branches often complete early in the pipeline, while regular instructions typically complete at the end of the pipeline. Condition (1) usually applies to input connects of storage elements and thus represents endpoints of store instructions, while (2) applies to functional units that are connected to a writeable register port using an output connect. Condition (2) applies to branch instructions updating the program counter register, as well as to regular instructions that retire after storing the computed result in a register.

Considering the example from Figure 55, store instructions retire at the cache `C`, while all other instructions are completed at unit `U3`. Thus, three hyperedges can be considered endpoints in this data path: $e5$, $e10$, and $e11$. All three endpoints meet condition (1) from above. The endpoints $e10$, and $e11$, in addition, also satisfy condition (2).

Definition. The set of instruction paths IP of a processor structure represented by the graph $\mathcal{H}_p = (V, E)$ consists of instruction paths $ip \in IP \subseteq \mathcal{P}(E)$ starting at an endpoint $e \in IP$ that meets the following conditions:

$$(1)\ endpoint(e),$$
$$(2)\ \forall p \in ip\colon \neg pred(e, p),$$
$$(3)\ \forall f \in ip, f \neq e\colon \exists p \in ip\colon pred(f, p),$$
$$(4)\ \forall p \in IP\colon (\exists f \in E\colon pred(f, p)) \rightarrow (\exists g \in IP\colon pred(g, p)),$$
$$(5)\ \forall i \in U \cup S\colon |hin(i) \cap ip| \leq 1 \wedge |hout(i) \cap ip| \leq 1,$$
$$(6)\ \forall i \in \mathcal{I}, f, g \in ip, f \neq g\colon tail(f) \cap tail(g) \cap pin(i) = \emptyset.$$

Instruction paths start with an endpoint, which is ensured by condition (1). In addition, it is not allowed by condition (2) to include a predecessor to this endpoint in the path – predecessors are possible for endpoints that are linked to register files. The edges of the path need to be connected to each other, as defined by condition (3). Condition (4) ensures that all paths properly terminate at immediates or register ports, but not in the middle of the data path. Finally, condition (5) specifies that

at most one input and output connect of an unit or storage element is allowed in a path. Not all instruction paths according to condition (1) through (5) are legal. Hyperedges constructed from output connects of different instances may point to the same port. This is illegal, because only one value can be assigned to a port. The last condition thus restricts the set of instruction paths to legal paths only. If a processor specification contains illegal paths an error is reported and the model is rejected.

Theoretically, the number of instruction paths in a hypergraph can be very large, e.g., if long chains of units are defined, where each unit offers multiple input and output connects. In practice, the number of paths is rather small due to the high complexity of the control logic that would be required to realize such processors.

Nevertheless, instruction paths are a powerful mechanism to compactly specify the flow of instructions through the processor pipeline. For example, the data path presented in Figure 55 consists of five paths: $\{e1, e2, e3, e6, e8, \mathbf{e10}\}$, $\{e1, e2, e4, e7, e9, \mathbf{e10}\}$, $\{e1, e2, e3, e6, e8, \mathbf{e11}\}$, $\{e1, e2, e4, e7, e9, \mathbf{e11}\}$, and $\{e1, e2, \mathbf{e5}\}$. The highlighted edge of each path represents its endpoint.

3.7.2 Instructions

Based on the instruction paths, individual instructions are extracted by combining the operations of the functional units along the path. An instruction can thus be uniquely identified by the set of operations and its instruction path.

Definition. The set of operations defined by the functional units in a processor structure is referred to as O. Every operation in O is attached to a functional unit, the function $unitop : U \rightarrow \mathcal{P}(O)$ defines a mapping between units and operations. Using *unitop* we can further define the function $edgeop : E \rightarrow \mathcal{P}(O)$ as follows: let e be the hyperedge of an output connect of a functional unit $u \in U$, $e \in hout(u)$, then $edgeop(e) = unitop(u)$, otherwise *edgeop* yields $\emptyset$.

Definition. An instruction is a tuple $(ip, ops) \in IP \times \mathcal{P}(O)$, where the following conditions are satisfied:

$$(1)\ \forall o \in ops\colon \exists e \in ip\colon o \in edgeop(e),$$
$$(2)\ \forall e \in ip\colon |ops \cap opedge(e)| = 1 \vee opedge(e) = \emptyset.$$

These conditions specify that (1) every operation has to be associated with an edge that is also part of the instruction, while (2) states that every operation is associated with exactly one edge, and that every edge that can be associated with an operation is actually associated with an operation of the instruction. In other words, a one to one mapping between the operations and the hyperedges originating from output connects of functional units exists for an instruction.

Definition. The set of symbols appearing as a condition or predicate in a processor model is referred to as A.
The conditions of an instance can be retrieved using the function $icond : \mathcal{I} \rightarrow \mathcal{P}(A)$, those of an operation using $opcond : O \rightarrow \mathcal{P}(A)$. In addition, a function $econd : E \rightarrow \mathcal{P}(A)$ yields the union of all conditions attached to links that comprise a hyperedge.
Predicates can be retrieved using analogous functions, $ipred : \mathcal{I} \rightarrow \mathcal{P}(A)$ for instances, $oppred : O \rightarrow \mathcal{P}(A)$ for operations, and $epred : E \rightarrow \mathcal{P}(A)$ for hyperedges.

Definition. The predicates and conditions of an instruction $q = (ip, ops) \in IP \times \mathcal{P}(O)$ can be derived as follows: let $insts = \{i \in U \cup S \mid \exists e \in ip\colon e \in hin(i) \vee e \in hout(i)\}$,

$$conds(q) = \bigcup_{i \in insts} icond(i) \cup \bigcup_{e \in ip} econd(e) \cup \bigcup_{o \in ops} opcond(o)$$
$$preds(q) = \bigcup_{i \in insts} ipred(i) \cup \bigcup_{e \in ip} epred(e) \cup \bigcup_{o \in ops} opred(o).$$

Definition. An instruction $q \in IP \times \mathcal{P}(O)$ is said to be legal, if all conditions attached to an operation, component instance, or link are ensured by a matching predicate, i.e., if $conds(q) \subseteq preds(q)$. The set of legal instructions of an xADL processor model is denoted by $\mathcal{I}nstrs$.

The set of legal instructions comprises the instruction set of the target processor. As an example consider the processor structure from Figure 55. Assume that unit `U1` specifies a single operation `decode`, unit `U2` the arithmetic operations `add` and `sub`, and `U3` a single operation `commit`. In total 7 instructions are defined, by combining the following set of operations with the paths from the previous section: $\{decode, add, commit\}$, $\{decode, sub, commit\}$, $\{decode, commit\}$, and $\{decode\}$.

Based on the instruction set, the assembly syntax and binary encoding of each instruction can be defined using the syntax and binary bindings that are attached to the operations and links. A behavioral model can be derived by combining the micro-operations in the operations set of the instruction. Even further, the hyperedges allow to relate the instructions to the underlying instances and connections. We can thus derive accurate timing and resource models of the individual instructions. The *adlgen* tool implements an algorithm to extract the instruction set according to the definitions from above. It also provides an analysis framework to analyze the processor data path and its instructions. Most importantly, the *adlgen* tool provides a rich set of generators. An overview of the tool and its implementation is given in the next chapter.

4 The *adlgen* Tool

The *adlgen* tool is the central component of the software environment developed around the xADL language. The tool reads the XML processor specification, parses the definitions, and creates an internal representation of the processor structure. The instruction set of the processor is then extracted from this data structure according to the definitions from the previous chapter. The individual instructions are also represented by an internal data structure, which provides, besides the operations and the instruction path of an instruction, a detailed model of the instruction's behavior. Which is in turn represented by a simple sequence of micro-operations that are enriched with information on data hazards, signals, data dependencies, and timing. Both representations are tightly coupled, such that the individual instructions and even micro-operations can be mapped to their corresponding hardware components and vice verse. In addition, all instructions are annotated with operand sets, hazard information, the assembly syntax, and binary encoding.

When the intermediate representations of the processor structure and its instruction set have been constructed, generator *modules* are invoked. Modules are the basic interface that connects the application-independent parts of the *adlgen* tool to the respective application-specific generators. A module usually processes the hardware components, the instruction set, or both in order to generate source code, configuration files, diagrams, reports, or other artifacts of interest. For example, the *compiler generator* mainly inspects the register instances and the instruction set in order to generate C source code and translation patterns for the instruction selector in order to customize a compiler backend.

Different modules sometimes require the same or very similar information, e.g., the timing of instructions or a summary of the dependencies and interactions among instructions. Common algorithms and data structures can be reused and shared among modules using so-called *provider*. A provider encapsulates an algorithm that analyzes or otherwise operates on the processor representations in order to provide additional information that is not available through the structural processor representation and the instruction set model.

The organization of the *adlgen* tool is oriented towards a regular compiler for the code generation from traditional programming languages. However, it operates on the processor structure and the abstract representation of the instruction set rather than on programs. It is further not limited to a single application scenario, but instead allows for multiple, quite different, generator modules. Ranging from simple reports and diagrams to complex source code generators that automatically customize a retargetable compiler or simulation framework. The tool consists of four major phases: (1) the *frontend* parses and validates the processor specification, (2) the second phase called *base* creates the basic data structures to represent the processor structure and its instruction set. Next, (3) *provider* compute globally shared summaries and abstractions from the detailed base models that are finally used by (4) the generator *modules*, which derive the desired artifacts. As for a

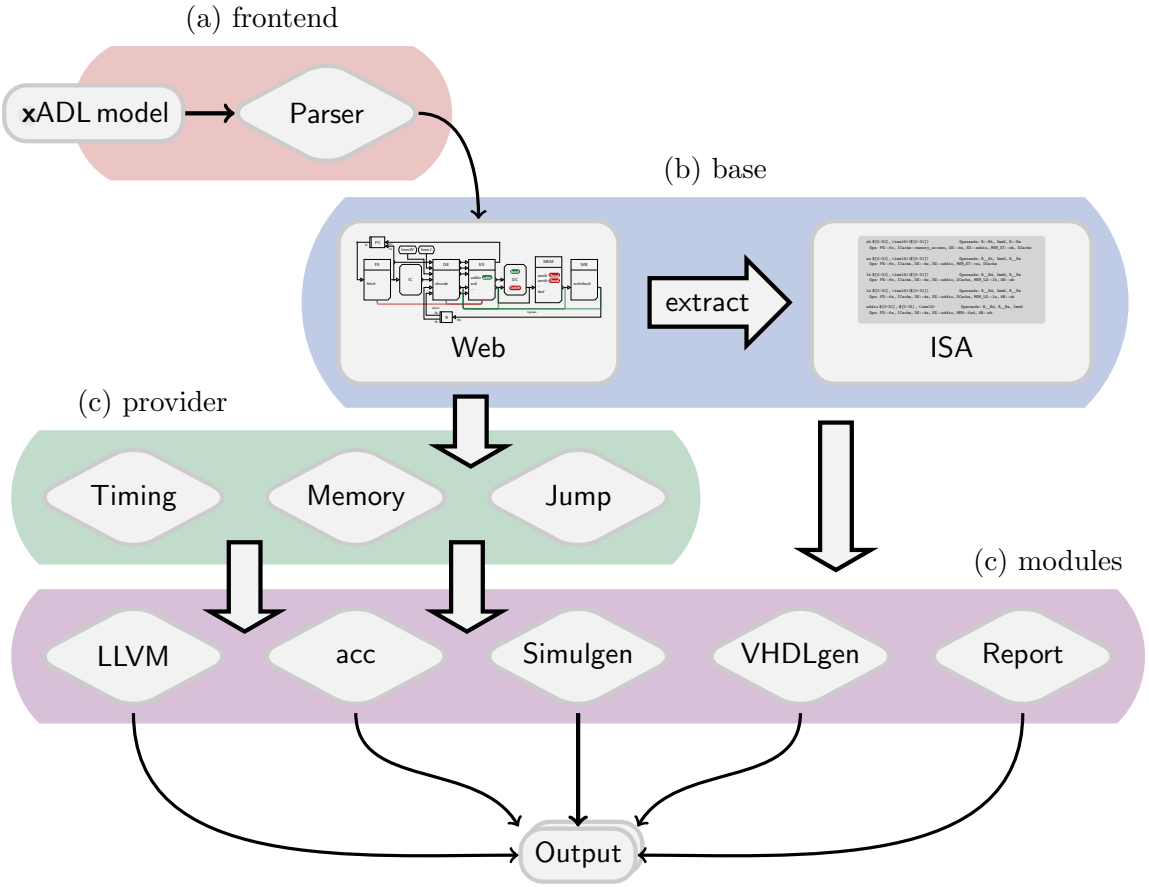

Figure 56: Organization of the *adlgen* tool, (a) the frontend parses the xADL file, (b) base creates an internal representation, (c) provider share common analysis data, and (d) modules generate the final artifacts.

compiler, the initial phases are very generic, while later phases become more and more application-specific. Figure 56 summarizes the organization of the *adlgen* tool, which is described in more detail in the following sections.

4.1 Frontend

The frontend roughly corresponds to the respective phase of a traditional compiler. The tool processes the command-line options and subsequently parses the XML processor description. The command-line options typically specify a list of generator modules, including module specific options, and switches to select the active processor configuration and programming conventions. The actual parsing is performed by an external XML library[10] that first validates the processor description against an

[10] `http://xmlsoft.org/`

XML schema definition of the xADL language, and then creates a *Document Object Model* (DOM), i.e., an extended abstract syntax tree.

4.2 Base

In the following, a representation of the processor model is generated from the DOM data structure, called *web*. This model precisely mirrors the structural hardware description using a graph representation, very much like the hypergraphs from the previous chapter. All configuration parameters have been resolved at this point. Also the programming conventions have been determined, such that a representation of the calling conventions and register usage conventions can be created. This also implies that an assembly syntax and a binary encoding variant has been determined, and that corresponding representations are computed. Types and instantiations, as well as connects and hazard links are resolved during the construction of the web structure and validated for consistency, i.e., name conflicts, missing definitions, matching bit-widths, et cetera.

The web data structure is traversed in order to assign the functional units and immediates to pipeline stages. Register ports and ports of storage elements are assigned to pipeline stages separately, because the individual ports can be accessed by different instructions in different pipeline stages concurrently. Functional units are assigned to a pipeline stage as a whole. The *stage assignment* operates on *chains* of hyperedges that form a linear sequence. Every chain starts at an *endpoint* and is ended by readable register ports or immediates. The number of hyperedges that contain at least one pipeline link is called the length of a chain. The chains are processed by a simple worklist algorithm in descending order according to their length. Initially, the worklist contains all chains of the data path. The longest chain, which corresponds to the longest pipeline in the data path, is processed first and removed from the worklist. The functional units and immediates, as well as the ports of register files and storage elements are assigned to a pipeline stage by counting the number of pipeline links between the end of the chain and the respective element. These assignments may partially determine the stage assignment of other chains, if the particular chain covers a functional unit or register port that has already been assigned to a stage. These overlapping chains are processed next and subsequently removed from the worklist. The stage counting now begins at elements that are already assigned to a stage by incrementing and decrementing the stage counter while traversing the chain towards the end and start point respectively. If all chains in the worklist are independent from the prior stage assignments, the longest chain is selected and the algorithm starts anew. These steps are repeated until the worklist is empty and all chains have been processed. Inconsistent stage assignments are detected and reported to the user during the processing, the processor model is rejected and not further processed. The instruction paths of all regular instructions have to end at stage zero, i.e., at least one of the paths chains has to start at a stage count of zero. This does not apply to parallel instructions.

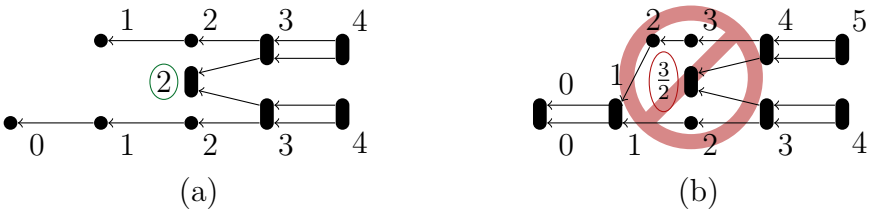

Figure 57: Assignment of pipeline stages to the components and ports of a data path, (a) a legal pipeline structure, (b) an illegal data path organization.

As an example consider the simplified data path in Figure 57(a). The nodes correspond to functional units and register ports, while the edges represent hyperedges with pipeline links. Four chains can be formed, where the longest is shown at the bottom. The elements along this chain are assigned to the stages zero through four. This numbering influences the stages of the the chain shown right above, and leads to the assignment of stage number two to the unit in the middle – marked with a green circle. This in turn influences the assignment of the two other chains. In particular, the stage number of the element at the top left corner is not zero but one. The regular instructions of this data path thus need to include the first chain in their respective instruction paths. A conflicting assignment is shown in Figure 57(b). The element at the center of the data path shows the conflicting assignments of stage number two and three respectively. The corresponding processor model is rejected and an error is reported to the user.

The instruction set is then extracted from the web according to the definitions from Section 3.7. As a first step the instruction paths are constructed by a simple depth-first traversal of the web data structure. In contrast to a depth-first search in a regular graph directed hyperedges may point to several independent successor nodes that *all* need to be considered at the same time. Consequently, a worklist stores the set of vertices that need to be processed next. Branches in the hypergraph are stored on a stack in order to backtrack and discover all possible paths. The algorithm is invoked for every endpoint of the hypergraph, an instruction path is complete when the worklist becomes empty. The processing stops when both, the worklist and the stack, become empty. Termination is guaranteed, because the web structure is verified to be free of cycles beforehand.

The individual instructions are then constructed by enumerating all possible combinations of the operations of the instruction paths. As described in the previous chapter, an instruction is represented by a set of operations and its instruction path. However, to simplify the processing in later phases additional information is computed and attached to the instructions. For example, a behavioral model is derived by combining the micro-operations of the operations. The data transfers between functional units and other components, represented by data and pipelining links, are modeled by simple `move` micro-operations that are annotated with additional information on hazard links originating from or targeting at the involved

```
Operations  : FE::fe, ICache, DE::de, EX::ori, MEM::fwd, WB::wb
Operands    : R::Rd[0-31], R::Rs[0-31], ImmW
Syntax      : mips.op3i_s = 'ori $[0-31], $[0-31], <imm16>'
                op = 'ori'
                rd = '$', WB::Rd_o [0-31]-> R::Rd, gpr_m
                rs = '$', R::Rs [0-31]-> DE::Rs_i, gpr_m
                imm = '', ImmW [0-0]-> DE::ImmW_i
Encoding    : mipsel.i16type2_b
                  instr:001101rrrrrrrrrriiiiiiiiiiiiiiii
                op = 001101 (0x34)
                rd = WB::Rd_o [0-31]-> R::Rd
                rs = R::Rs [0-31]-> DE::Rs_i
                immw = ImmW [0-0]-> DE::ImmW_i
```

Figure 58: Meta-information associated internally with the *or immediate* instruction of the MIPS processor.

ports of the link. Similarly, cache and memory accesses are represented using the internal pseudo micro-operations `read` and `write`. It is always possible to arrange the micro-operations in a linear sequence according to their data dependencies using a simple list scheduling algorithm [3], even if the path consists of multiple independent functional units that operate in parallel. This linear sequence is further sorted ascending according to the pipeline stage assigned to the respective parent units and links of the micro-operations. In addition to the behavioral model, syntax and binary encoding information is stored with every instruction. The respective bindings are collected from the operations and links comprising the instruction.

Figure 58 depicts a textual representation of the internal meta-information that is associated with each instruction. The list of `operations` at the top represents the instruction path and its operations, while the `operands` set specifies the register and immediate operands of the instruction. The `syntax` information is shown below, on the left hand side the syntax format and template, on the right hand side the resulting syntax string, where operands are represented by placeholders in brackets. The individual syntax bindings are listed below. A similar representation is used to display the binary encoding information. First, the instruction's binary format and template is shown, followed by the bits encoded by the `instr` field – which is the only field defined by the MIPS model. Below the individual binary bindings are shown. The behavioral model of the instruction is presented in Figure 59. It shows a sequence of micro-operations sorted according to their respective pipeline stages and data dependencies. The annotations on the right hand side indicate the pipeline stage (`st`) and parent operation (`op`). The arrows on the left side, indicate the information that is available on bypasses for the particular instruction. In addition, the instruction is potentially aborted by another instruction via the

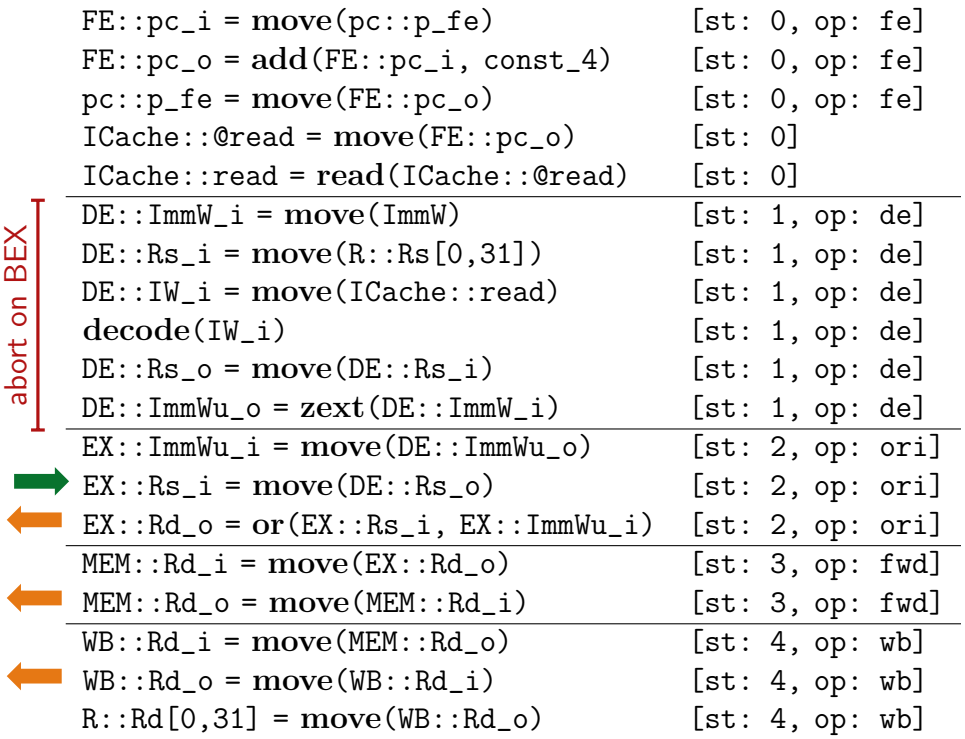

Figure 59: Behavioral model of the *or immediate* instruction.

BEX signal when the instruction is executing in stage 1. This is indicated by the red annotation on the left hand side of the figure.

The *base* component of the *adlgen* tool provides an application-independent representation of the processor's hardware organization and its instruction set. These two representations are tightly coupled, the abstract behavior of instructions provides various views that include the semantics of the instruction, its timing behavior, and its interactions with other instructions through hazard links and signals.

4.3 Provider

Provider are extensions to the infrastructure offered by the base component. A provider analyzes the structural model or the instruction set, and provides access to its analysis results through a well-defined interface. The services offered by a provider are usually very generic in order to reuse the information across different application-specific generator modules. In contrast to the data structures of the base component that have to be created on every invocation of the *adlgen* tool, provider are only expected on demand when a generator module or another provider requests information via the provider's interface.

flag:	unknown	pc_relative
index:	direct	indexed
direction:	none	unknown
	increment	decrement
autoincrement:	no_autoincrement	
	pre_autoincrement	pre_autodecrement
	post_autoincrement	post_autodecrement

Table 4: Addressing modes recognized by the memory provider.

The *aldgen* tool offers several predefined provider implementations: (1) the *timing* analysis, (2) the *memory*, and (3) the *jump* provider. The timing provider analyzes the instruction set of the processor and offers a summary of the timing characteristics of every instruction. The summary specifies the latency in cycles of the instruction for certain execution scenarios. The analysis considers interactions due to data dependencies, data hazards, control dependencies. The data dependence analysis covers true, false, and output dependencies caused by registers and memory locations. Also bypasses described by hazard links are considered by the analysis. However, the abstract summary information is not able to reflect the effects of partial or incomplete bypassing. Thus, in the presence of partial bypasses the analysis calculates conservative bounds that ensures that the data is guaranteed to be available under any circumstances. Control dependencies may arise from branches, jumps, and trapping instructions that potentially alter the control flow during execution. Thus updates of the program counter and interactions via signals need to be analyzed in addition to the regular data dependencies. The summary, again, computes conservative bounds to ensure a safe approximation of the control dependencies.

The memory provider analyzes the memory access patterns and addressing modes of instructions. The behavioral model of the instructions is scanned for `read` and `write` micro-operations. Micro-operations supplying the address and data for the memory accesses are subsequently analyzed and classified using four attributes: (1) *flag*, (2) *index*, (3) *direction*, and (4) *autoincrement* – see Table 4. The first attribute provides useful hints about the memory access in general, e.g., if the addressing mode was successfully classified or whether the access is relative to the program counter. Secondly, indexed addressing modes are represented. A *direct* addressing mode consist of a single base address operand, while *indexed* addressing modes add or subtract an additional *index* operand to/from the base address. The direction attribute specifies whether the index is added to or subtracted from the base operand. The final attribute is relevant for memory operations with autoincrement addressing modes, where the base address is automatically incremented or decremented before or after the memory access. Note that the attributes can be combined, e.g., the SPEAR processor offers a load instruction with a combined

```
memory analysis - lw $[0-31], <imm16>($[0-31]) :
  load am: pre_inc       base: pc::p_fe   auto: const_4   [pc_rel]
  load am: indexed_inc   base: R::Rs      index: ImmW
  sl: 1 ld: 1 ss: 0 sd: 0 am: indexed_inc
```

Figure 60: Memory access summary for the MIPS *load word* instruction.

indexed-increment and *post_autodecrement* addressing mode that corresponds to the C expression '`*(basereg-- + imm)`'. Additional patterns for typical addressing modes of DSP architectures, e.g., bit-reversed or modulo addressing, are also possible, but currently not supported. In addition, side-effects due to register updates, other memory accesses, and modifications of the value loaded from or stored to the memory are summarized. Instructions without side-effects are most often useful for the various generator modules, these *simple* load or store instructions are thus marked by an additional flag. Note that the memory access to fetch the instruction from memory is not considered a side-effect, but only if the loaded instruction word is exclusively processed by the `decode` micro-operation.

Figure 60 shows an excerpt of the analysis results computed by the memory provider for the MIPS model. It shows a list of memory accesses of the *load word* instruction. The first load represents the memory access to fetch the instruction and is thus not further considered. The second memory access loads data from the memory and stores the loaded data unmodified into a register. The address computations are successfully recognized as an *indexed-increment* addressing mode. The flags at the bottom indicate that the instruction loads data (`ld`) and can further be classified as a simple load (`sl`) with an indexed addressing mode.

Another important characteristic of an instruction is its branching behavior, i.e., whether the instruction updates the program counter. This information is analyzed by the *jump* provider, which scans the instruction's micro-operations for, possibly conditional, updates of the program counter register. The update patterns are subsequently classified as *conditional* or *unconditional*, *absolute* or *relative*, and *register-based* or *symbol-based*. In contrast to the memory analysis that is very precise in capturing the addressing modes, the patterns accepted by the branch analysis are less sharp. In particular, for symbol-based branches involving immediates. The scope of these branches is often limited by the bit-width of the immediate operand. Many architectures thus try to extend the range by transforming the immediate's value. From the user perspective these tricks are usually not visible, instead the development tools, foremost the assembler and linker, hide these complexities. The jump analysis thus accepts a broad spectrum of transformations of immediate operands and still recognizes the given instruction as a regular branch or jump.

For regular instructions the program counter is typically incremented during an early execution phase. Branches, however, typically consist of multiple assignments to the program counter, where the actual assignment of the branch's target is exe-

```
jump analysis - jr $[0-31] :
  jump summary: jump_expr(ABS, REG, UCD)
jump analysis - j <imm26> :
  jump summary: jump_expr(ABS, SYM, UCD)
jump analysis - bgezal $[0-31], <imm16> :
  jump summary: jump_expr(REL, SYM, CND)   save: R::Rd
```

Figure 61: Branch behavior of two jump and a branch instruction of the MIPS model.

cuted late in the pipeline. The branch analysis is able to handle at most two updates, where the first is required to continue the execution linearly and only the second assignment is interpreted to be the actual branch pattern. Branches that do not follow this convention are not recognized properly and thus treated conservatively. This model implies that the execution at first continues linearly after the branch, until the actual target address is assigned to the program counter. The cycles between the first and the second assignment is referred to as *branch delay slots*. Some processors explicitly define branch delay slots, however, longer branch delay slots are often cumbersome to handle, both in software and hardware. It is thus common to abort instructions that would otherwise execute in the branch delay slots. Signals provide a very intuitive and well analyzable way to control the behavior of instructions that are executed in branch delay slots. The jump analysis thus uses the timing behavior of the branch and the interactions that can be observed through signals to derive the number of branch delay slots. In addition, the jump analysis also recognizes whether the original program counter is stored in a register by the jump or branch instruction. Instructions that store the program counter often can be used to realize function calls efficiently.

The result of the jump analysis for two jump and a conditional branch instruction is shown in Figure 61. The analysis results of the *jump register* instruction are shown first. This instruction is rather simple, because the value of a general purpose register is simply assigned to the program counter. It is thus classified as an unconditional, absolute, register-indirect jump. The *jump* instruction is more complex, because the limited scope of the 26 bit-wide immediate operand is not sufficient to cover the full 32-bit address range of the program counter. The MIPS architecture solves this problem by first shifting the immediate's value to the left by two and then merging the upper four bits of the current program counter with the shifted value. The analysis recognizes the instruction to be a symbol-based, unconditional, absolute jump. The final example, shows a conditional *branch and link* instruction that is recognized as a symbol-based, conditional, relative branch. The analysis also recognizes that the address of the instruction following the branch is stored into a register as indicated by the `save` annotation at the bottom.

4.4 Modules

Modules finally process the structural processor model, the instruction set model, and the analysis results computed by provider in order to derive source code, test cases, diagrams, documentation, or other kinds of artifacts. The interfaces between modules, the base framework, and the provider is well-defined, it is thus possible to extend the *adlgen* tool with little effort and even load modules dynamically on demand.

Currently, nine generator modules are provided by the *adlgen* toolkit. The *Simulgen* [24, 25, 58, 165] module allows to automatically derive a cycle-true simulator of the processor pipeline. The simulation engine is based on mixed interpretation [58] and dynamic code generation [165]. Initially, all instructions are interpreted using a very simple scheme. Every instruction is split according to the pipeline stages of its behavioral model. For every stage a separate C/C++ simulation function is generated that is invoked by the interpreter. The simulation functions operate on a software emulation of the processor's hardware state, i.e., registers, caches, memories, and pipeline links are represented by global variables that are read and written accordingly. In fact, the simulation functions operate on a shadow state that is required in order to model the parallel operation of the different pipeline stages and instructions, e.g., for VLIW processors. Thus after every cycle the actual processor state needs to be updated from the shadow state by invoking so-called *epilogue* functions for every active instruction. The interpreter keeps track of the currently active instructions and invokes the simulation and epilogue functions for a particular instruction one after the other on every simulated cycle. For a simple five-stage processor this can lead up to ten function calls on every cycle in the worst case. In the case of VLIW processors, the overhead is even bigger, due to the independently operating pipelines. For longer running programs the simulation speed of this simple interpreter is not satisfactory. The simulation speed is thus improved using dynamic compilation. In addition to the interpreter functions, corresponding code generation functions are derived from the processor model. When a particular basic block of the simulated program is executed over and over again, the block's instructions are translated to native machine code of the simulator's host machine. Subsequently, when the same code is to be simulated, the fast native code is invoked instead of the slow simulation functions. We use the just-in-time compiler provided by the LLVM compiler infrastructure [115] to perform high-level code optimizations and the low-level code generation. The compilation process starts by invoking the code generator functions in parallel to the interpreter functions. The functions build a new function in LLVM's intermediate representation that represents a linear instructions sequence of a basic block of the simulated program. If the compiled basic blocks are still executed frequently, multiple basic blocks are recompiled and combined into a so-called *region* – also referred to as traces. The LLVM functions for regions, in contrast to the basic block functions, represent *non-linear* code fragments that can even contain loops. Experimental results show that this approach reduces

the compilation overhead, but at the same time achieves an outstanding simulation speed of up to several hundred MHz [24, 165].

The simulation framework is further able to optimize asynchronous interrupts that are modeled using parallel instructions. The simulation of interrupts is very costly, because the compilation overhead is drastically increased and the compiled code is considerably slower. The problem arises from the asynchronous nature of interrupts, where interrupt checks need to be performed on every cycle in order to ensure that all interrupts are detected properly. However, most of the checks are useless, because interrupts are relatively rare compared to the clock frequency of typical processors. The interrupt dispatch can be modeled independently from the regular instructions in xADL processor descriptions using parallel instruction. It is thus possible to specifically optimize the simulation of the interrupt dispatch using a rollback mechanism [25]. This reduces both, the simulation and the code generation overhead, and thus significantly improves the overall simulation speed.

The *Simulgen* module is very mature and has been used to generate simulators for several processors. The *VHDLgen* module, in contrast, is an early prototype that has been developed in the course of a lecture.[11] It is able to automatically derive large portions of a synthesizable VHDL description from a processor model, including the register files, the functional units, the data paths connecting the computational resources, as well as the control unit. In its current form, only the generation of the instruction decoder and the generation of memories and caches is missing. Also the generation of register files is limited, depending on the capabilities of the underlaying FPGA technology. A prototype implementation of the MIPS processor using a Xilinx Virtex 4 FPGA (XC4VL25) on a Xilinx ML-401 board shows that correct VHDL code can be generated from the processor descriptions. The synthesized core is able to execute medium sized MIPS programs at about 25 MHz and occupies about 20% of the logic cells of the FPGA – including all register files and 64 KB of SRAM for data and instructions.

The *vcg* generator module allows to derive a visualization of the processor structure. The instances as well as links are represented using a directed graph that is rendered and displayed by the *XVCG*[12] graph editor. During the design of a processor these diagrams are often helpful to verify that the connections between the individual components are correctly modeled. The *report* module is similarly intended for debugging purposes. It generates an XML report of the processor's instruction set that includes, besides the behavioral model, also information on the instruction syntax and encoding. An additional XSLT-stylesheet can be used to render the information as HTML.

The *LLVM* module is a compiler generator targeting the LLVM compiler infrastructure [115] that automatically derives a register file description, an instruction set model, a resource model for instruction scheduling, tree patterns for instruction

[11] `http://en.wikiversity.org/wiki/Computer_Architecture_Lab/WS2007`
[12] `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`

selection, and additional C++ clue code in order to retarget the LLVM backend to the processor described by the model. Another compiler generator is available for the proprietary compiler backend *acc*, which uses a modified version of GCC[13] as frontend. The generator consists of three modules *rd-acc*, *ot-acc*, and *pd-acc* that generate a register file description, operation tables for instruction scheduling [174], and tree patterns for an instruction selector based on lburg [64] respectively. An automatic completeness test of the instruction selector can be used to prove that the derived tree patterns cover all input programs that are possibly accepted by the compiler frontend, i.e., if the instruction selector is able to produce machine code for all valid input programs. The completeness test as well as the compiler generators are described in more detail in the following chapter.

[13] `http://gcc.gnu.org/`

5 Compiler Backend Generation

The automatic generation of compiler backends is a central application area of the xADL language. This task is particularly challenging, because the generated backend components are not only required to generate *correct* code for the specified processor, but also *efficient* code. The major challenge is to bridge the gap between the semantic model of the compiler's intermediate representation and the behavior of the processor's instructions. The *instruction selection* phase of a compiler usually performs this task. A processor description that is intended to support the automatic generation of compiler backends thus requires a very detailed, but still abstract, behavioral model of the instruction set. However, a behavioral model alone is not sufficient to generate correct and efficient code. The instructions, generated through instruction selection, have to be reordered in order to efficiently utilize the available hardware resources, and most importantly guarantee that structural and data hazards among the instructions are avoided. This is particularly important for application-specific processors, because hardware resources that automatically resolve hazards are often eliminated due to area and power constraints. The compiler is thus responsible that all data and the computational resources are available in a timely fashion. The *instruction scheduling* phase of the compiler's backend is typically responsible for this reordering. For VLIW processors the scheduler groups the instructions into bundles for parallel execution. In contrast to the instruction selection, this phase is not concerned with the semantics of the involved instructions, but solely with their timing and resource usage. Processor description languages targeting the automatic generation of compiler backends thus ideally provide a precise structural view of the processor's hardware organization that is tightly coupled with the instruction set model. The xADL language, in combination with the *adlgen* tool, provides the required information, and is exceedingly well suited for the automatic generation of compiler backends.

The *backend generator* heavily depends on the functionality provided by the instruction set of the given processor. For minimalistic processors that only offer limited capabilities the generator may fail to derive a *complete* compiler, i.e., one that can generate machine code for all input programs possibly accepted by the compiler frontend. It is thus of utmost importance that the generator tool provides meaningful feedback to the processor designer regarding the completeness of the derived instruction selector. The xADL language provides enough information to perform a formal completeness test that proves whether the derived instruction selector is complete.

5.1 Background

A compiler translates a program specified via a programming language to machine code of the target processor. Typically this translation proceeds in three phases:

(1) the *frontend* parses the input program and generates a target-independent intermediate representation, (2) the *middleend* then applies high-level transformations and optimizations to this intermediate form, (3) the *backend* translates the target-independent intermediate representation to machine code and further applies processor-specific optimizations. When a compiler is to be retargeted for a new processor, it is often sufficient to customize the backend only. This is particularly true for retargetable compilers that are designed to support multiple target processors. To retarget a compiler for a given processor the three main phases of the backend have to be adopted: (1) the instruction selector, (2) the instruction scheduler, and (3) the register allocator. Depending on the processor's features additional transformations and optimizations might prove beneficial, but these are not strictly required.

5.1.1 Instruction Selection

The instruction selector is the first highly target-dependent transformation in a compiler. During this phase the target-independent intermediate representation is translated to a target-dependent representation, the *machine-level IR*. This process is a highly complex problem that is usually solved using heuristics that try to minimize the expected execution time, code size, and power consumption. The resulting representation is very close to the actual machine code of the target processor. The most significant difference at this point is that the instruction operands are not yet lowered completely to the machine level. Local variables are translated to an infinite set of *virtual registers* that is later mapped to the limited set of hardware registers by the register allocator.

A very popular approach for instruction selection is *tree pattern matching* [49, 65, 64]. Here, *statements* of the input program are represented as *trees*. The leaf nodes in the trees correspond to *operands*, i.e., virtual registers and constants, other nodes represent *operators* of the source language. Two nodes are connected by an edge, if the calculation performed by the first node depends on the calculations of the other. The instruction selector computes a cost-minimal *cover* of the tree using an augmented *tree grammar* [36]. The *tree rules* of the grammar consist of a *tree pattern*, a *cost function*, and an *emit function*. The pattern describes a fragment of the compiler's intermediate representation that is covered by its rule. The *terminal* symbols of the rule patterns match the operands and operators of the IR, while *non-terminal* symbols connect the individual rules of a cover. Rules with a tree pattern consisting of a single non-terminal are referred to as *chain* rules.

The cost function associates a rule instance, i.e., a rule and the concrete IR nodes that it covers, with a numeric cost. The costs can either be *static* or *dynamic*. In the former case the costs can only be constant, while in the latter case global compiler options or the IR nodes of the rule instance can be considered by the cost function. If the dynamic cost function evaluates to infinity for a rule instance,[14] the

[14]Infinity is usually represented using a large integer number.

Pattern				Cost	Emit
(1)	r	$\rightarrow$	ar	1	`mov r = ar`
(2)	ar	$\rightarrow$	r	1	`mov ar = r`
(3)	r	$\rightarrow$	$\mathbf{V}$	0	`V`
(4)	imm	$\rightarrow$	$\mathbf{C}$	0	`C`
(5)	r	$\rightarrow$	imm	1	`ldi r = imm`
(6)	r	$\rightarrow$	$*(r_1, r_2)$	3	$\texttt{mul r} = \texttt{r}_1 * \texttt{r}_2$
(7)	r	$\rightarrow$	$+(r_1, r_2)$	1	$\texttt{add r} = \texttt{r}_1 + \texttt{r}_2$
(8)	r	$\rightarrow$	$+(r_1, imm)$	1	$\texttt{add r} = \texttt{r}_1 + \texttt{imm}$
(9)	r	$\rightarrow$	$\mathbf{LD}(+(ar_1, imm))$	5	$\texttt{ld r} = [\texttt{ar}_1 + \texttt{imm}]$

Table 5: Example tree grammar for instruction selection.

rule is effectively disabled and not considered by the instruction selector, such cost functions are called *dynamic checks*.

The cover can be computed very efficiently using *dynamic programming* by traversing the tree representation twice. First, during the *labeling* phase, the viable rule instances are enumerated for every tree node bottom-up starting at the leaf nodes. The nodes are annotated with state labels that specify the minimal costs to derive a given non-terminal symbol for the node and its sub-trees. A second top-down sweep, called the *reduce* phase, selects the cost-optimal rule instance for every IR node and invokes the appropriate emit function, which generates the machine-level representation.

Definition. A tree grammar [36] is defined as $G = (S, N, \mathcal{F}, R)$, where N is the set of non-terminal symbols, $\mathcal{F}$ is a ranked alphabet of terminal symbols, and R is a set of production rules. Derivations of the grammar begin with the start symbol, also called *axiom*, $S \in N$. The rules consist of a non-terminal symbol on the left hand side of the production rule and a term in the form of a tree on the right hand side $q \rightarrow t$, where $q \in N$ and $t \in T(N \cup \mathcal{F})$:

(1) $N \subseteq T(N \cup \mathcal{F})$,
(2) $\mathcal{F}_0 \subseteq T(N \cup \mathcal{F})$,
(3) $\forall p \geq 1, f \in \mathcal{F}_p, t_1, \ldots, t_p \in T(N \cup \mathcal{F})\colon f(t_1, \ldots, t_p) \in T(N \cup \mathcal{F})$.

For instruction selector specifications, the production rules $r \in R$ are further augmented with cost and emit functions, denoted by $cost(r)$ and $emit(r)$ respectively.

A simple example tree grammar is presented in Table 5. The first two rules are chain rules that convert the ar to the r non-terminal symbol, and vice verse. Rule (3) and (4) match leaf nodes, i.e., virtual registers and constant operands. The chain rule (5) further allows to convert the imm non-terminal to the r non-terminal by loading the constant into a register. The other rules match an operator, where the

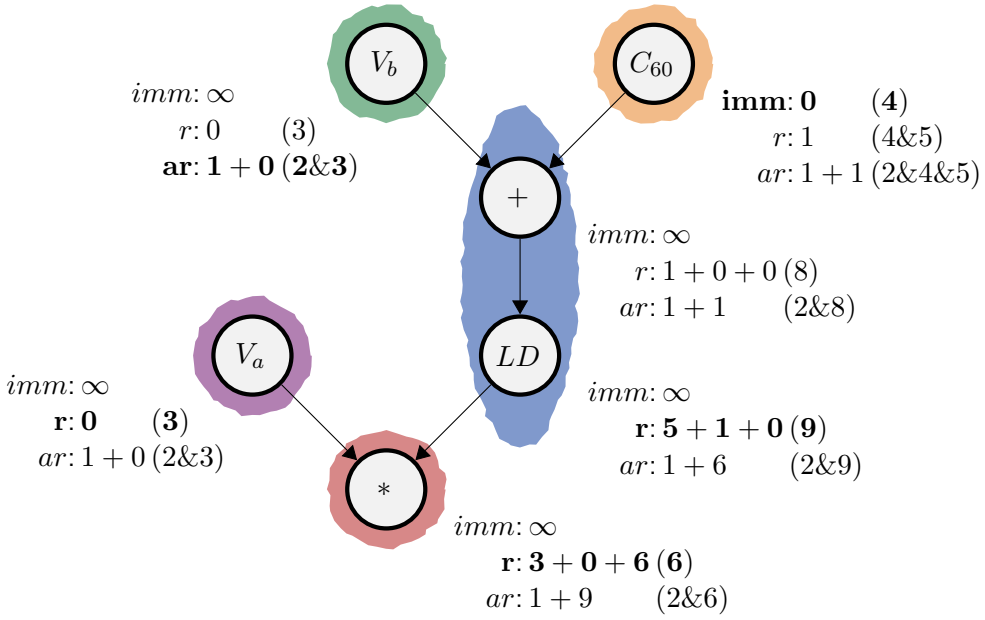

Figure 62: Tree pattern matching using dynamic programming.

operands are denoted using numbered non-terminals. Patterns are allowed to be nested as shown by rule (9), witch covers a load operation and its addressing mode. The listed rules can be used to cover the tree of the C expression '`a * b[15]`' as shown in Figure 62. Every IR node in the tree is annotated with a state label that lists the costs to derive the individual non-terminals for the node and its sub-trees, followed by the numbers of the rule producing the respective non-terminal. The cost-optimal cover is highlighted using bold font. Note that the load and the plus nodes are covered by a single rule, thus no rule is selected for the plus node separately. The resulting machine code is presented in Table 6.

Aho and Johnson were the first to use dynamic programming for optimal instruction selection on trees [2]. Work by Hoffman and O'Donnell [95] was combined with this approach and lead to the code generation framework *TWIG* [1]. Later, the tree pattern matching approach was improved by Emmelmann et al. [49], Fraser et al. [65, 64, 66], and Ertl et al. [52].

Code		Rule Number	Costs
(1)		3	0
(2)		3	0
(3)	`mov` $\texttt{ar}_1 = \texttt{b}$	2	1
(4)		4	0
(5)	`ld` $\texttt{r}_1 = [\texttt{ar}_1 + 60]$	9	5
(6)	`mul` $\texttt{r}_2 = \texttt{a} * \texttt{r}_1$	6	3

Table 6: Final machine code generated from a tree cover.

Although pattern matching on *Directed Acyclic Graphs* (DAGs) is known to be NP-complete in general [151, 30], instruction selection has also been applied to more general graphs. Ertl shows that for certain classes of tree grammars optimality can be guaranteed even when applied to DAGs [51]. Koes and Goldstein [108] formulate the code selection on DAGs as a 0-1-integer problem that can be solved to optimality for almost all instances. Eckstein et al. use *Partitioned Binary Quadratic Programming* (PBQP) to solve the instruction selection problem for SSA graphs [46]. The PBQP can be solved using an optimal branch-and-bound algorithm or using a faster heuristic. Ebner et al. [45] extend this approach to more general DAG patterns to model instructions with multiple result values.

5.1.2 Completeness of Instruction Selectors

An important property of the instruction selector is its *completeness*. An instruction selector specification is complete, if for all input programs possibly accepted by the compiler frontend a valid derivation can be found, i.e., it is always possible to derive a machine code sequence. Developing a complete instruction selector is hard. Idiosyncratic architectures that offer different special-purpose register files, complex instruction sets, complex addressing modes, and architecture extensions lead to large specifications that can easily get out of hand. Dynamic cost functions that possibly evaluate to infinity, i.e., dynamic checks, further complicate this task, because compiler flags, optional architecture features, and checks that depend on the input program may affect the set of applicable rules.

Therefore, it is important to develop techniques that allow to prove the completeness automatically. Traditional approaches are based on finite tree automata, which are closely related to tree grammars [36]. The instruction selector and the compiler's IR are described using two tree automata. These automata are then compared to prove that the language accepted by the automaton representing the instruction selector is a super-set of the language accepted by the automaton modeling the IR. However, dynamic checks cannot be represented using finite tree automata. Unfortunately, these checks are very common in today's compilers, rendering the existing completeness tests useless for these systems.

Definition. A *Nondeterministic Finite Tree Automaton* (NFTA) over the ranked alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, where Q is a set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules:

$$f(q_1, \ldots, q_n) \rightarrow q,$$

where $n \geq 0$, $f \in \mathcal{F}, q_1, \ldots, q_n, q \in Q$.

The tree automaton runs on ground terms over $\mathcal{F}$. It starts at the leaves and moves upwards, inductively associating a state with each visited sub-term. A ground term $t \in T(\mathcal{F})$ is accepted by a finite tree automaton if there exists a reduction

starting from the ground term and leading to a configuration q, where q is a final state. The tree language $L(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all ground terms accepted by $\mathcal{A}$. Similar to NFTA, *Deterministic Finite Tree Automata* (DFTA) can be defined, with the only restriction that no two rules are allowed that have the same left hand side. The class of languages accepted by NFTA and DFTA is equal, in fact for every NFTA an equivalent (minimal) DFTA can be constructed.

Recognizable tree languages have some useful closure properties that are helpful when developing a completeness test for instruction selectors. Most notably this class of languages is closed under intersection, complementation, and union. In addition, testing whether a given recognizable tree language is empty is decidable. This allows to verify if the language of one finite tree automaton completely covers the language of another finite tree automaton. For example, consider two languages $L(\mathcal{A})$ and $L(\mathcal{A}')$ accepted by $\mathcal{A}$ and $\mathcal{A}'$ respectively. We can prove that $L(\mathcal{A}) \subseteq L(\mathcal{A}')$ by showing that $L(\mathcal{A}) \cap \overline{L(\mathcal{A}')}$ is empty. Alternatively, it is possible to calculate the minimal DFTA for each of the two languages and compare the states.

Unfortunately, more general classes of tree languages do not have these properties. For example, tree automata that are capable of expressing additional constraints often lose these properties.

Definition. An *Automaton With Equality and Disequality Constraints* (AWEDC), consisting of a tuple $(Q, \mathcal{F}, Q_f, \Delta)$, where $\mathcal{F}$ is a finite ranked alphabet, Q is a finite set of states, $Q_f \subseteq Q$ a set of final states, and Δ a set of transition rules:

$$f(q_1, \ldots, q_n) \xrightarrow{c} q,$$

where $f \in \mathcal{F}, q_1, \ldots, q_n, q \in Q$, and c is a boolean combination of equality and inequality constraints.

These constraints can be used to model *Read-Modify-Write* (RMW) operations that require the memory address of the read to be equal to the address of the write. Emptiness for this class of languages is not decidable, a general completeness test based on this class of languages is thus infeasible.

Related to recognizable tree automata are regular tree grammars, in fact, it can be shown that for every regular tree grammar a corresponding tree automaton can be constructed. Instruction selectors are usually specified using a tree grammar notation, while completeness tests are based on tree automaton theory. More details on tree languages, automata, and grammars can be found in [36] along with proofs of the claimed properties in this section.

Work on formal completeness tests is rare. Emmelmann [48] presents an approach that relies on regular tree grammars to express the compiler's IR and prove the completeness of a given instruction selector specification. Dynamic checks, very frequently found in existing compilers, cannot be handled using his technique. Rules carrying such checks are ignored, instead, counter examples are presented to the

developer, who is then responsible to verify completeness by hand. Giegerich et al. [73, 72] use order-sorted signatures to describe code generators. As a side-effect they are able to verify the completeness of a specification by mapping to a regular tree grammar.

5.1.3 Instruction Scheduling

The instruction scheduling phase reorders the instructions of the machine-level intermediate representation in order to minimize the execution time and code size requirements and maximize the resource utilization of the processor's resources. A major problem is the interaction with register allocation, i.e., aggressive scheduling can be counter productive, when more spill code is generated. The scheduling is thus usually performed twice, once before register allocation, i.e., by the *prepass* scheduler, and again afterwards, i.e., by the *postpass* scheduler. The scheduling algorithm typically processes one basic block at a time, by scanning the instructions of the block in order to determine their data and control dependencies. The scheduling algorithm is required to preserve the dependencies of the corresponding *Data Dependence Graph* (DDG). In addition, restrictions by the processor hardware need to be considered through a *resource model*.

List scheduling is a very elegant and simple technique that is adopted by a large number of compilers today. The basic idea is to process the data dependence graph using a *ready list*. A node in the DDG becomes ready, when all its predecessors in the graph have been scheduled. The scheduler heuristically selects one of the ready nodes, based the node's *priority*, which is derived from its instruction, its neighbors, and the maximal path length to a root node. Before that instruction is actually scheduled, the resource model is queried for conflicts or hazards. The resource model tracks the currently active instructions along with the blocked resources, either using traditional *resource tables* [47], *operation tables* [174], or *finite state automatons* [152, 14].

Instruction scheduling goes back to the problem of micro-code compaction and became prominent with the first RISC computer architectures [93, 92, 71]. It was then further refined [190] and adopted for almost all modern processor architectures. Recently, an optimal scheduling technique based on integer linear programing has been proposed by Wilken et al. [191]. The scheduling phase is particularly important for VLIW architectures, because the scheduler not only reorders the instructions but also groups independent instructions into bundles for parallel execution. The traditional basic-block-based techniques only provide limited parallelism for wide VLIW processors, which lead to the development of *region* scheduling techniques [60, 43, 100, 122].

5.1.4 Register Allocation

During register allocation the *virtual registers* generated by the instruction selector are mapped to the registers available in hardware. Typically not all virtual registers can be kept in hardware registers, thus *spill code* needs to be generated to save and restore register values to/from memory as needed. The register allocator relies on a model of the available hardware registers and their interferences, as well as on information on the legal register classes for the operands of the instructions. These constraints are expressed using sets of hardware registers – so-called *register classes*. The registers of a class share common properties and meet common constraints, e.g., the registers are legal operands to certain instructions. Overlapping registers are specified using *alias sets*, i.e., sets of conflicting registers that cannot be used to hold different values at the same time.

A very popular approach to register allocation is *coloring* of the *interference graph*, as proposed by Chaitin et al. [32]. Interfering virtual registers are represented as nodes in this graph that are connected by an edge. Two virtual registers interfere, when both are live at a given program point, i.e., the values of both registers need to be available in a hardware register. The coloring fails, when the number of virtual registers exceeds the number of hardware registers. In this case, registers are heuristically selected for spilling. The interference graph is constructed again and another coloring attempt is made. These steps repeat until a valid coloring is found.

Graph coloring is the basis of many modern register allocators, and was extended by Chow and Hennessy [35], and Briggs et al. [28]. Briggs et al. [29] propose an extension to the graph coloring scheme that is able to handle register pairs. An even more flexible extension for processors with partitioned and possibly aliasing register files was presented by Runeson and Nystrõm [169], and Smith et al. [176]. Recently Hack showed that the interference graphs of programs in SSA form [40] are chordal and thus allow an optimal coloring to be computed in polynomial time [83, 82]. Optimal and near-optimal register allocation can be performed via linear and non-linear programing [75, 9, 90]. Computing the interference graph is often too costly for systems that generate machine code dynamically. The *linear scan* algorithm [150] thus avoids the interference graph completely. Instead, the *live ranges* of the virtual registers are represented as numeric intervals that are processed in ascending order by a single sweep. The register assignment and spilling decisions are based on the extend of the currently active live ranges.

5.1.5 The LLVM Compiler Infrastructure

The *Low Level Virtual Machine*[15] (LLVM) [115] is a retargetable compiler framework with support for static and dynamic code generation that can be used very flexibly for the development of classical static compilers [101], the implementation of dynamic execution and runtime environments [69], and simulation tools [37, 99, 34, 22].

[15] `http://www.llvm.org/`

Several backends are available with the official LLVM package, which includes code generators for x86, x86-64, ARM, PowerPC, MIPS, Alpha, Sparc, Cell, and PIC16. Numerous other backends are available separately, e.g., for a Transport Triggered Architecture implementation [101]. In the context of the xADL software environment LLVM is used for two application scenarios: (1) as a code generator for the dynamic compiling simulation engine [24, 25], and (2) as a customizable and retargetable static compiler for the automatic generation of compiler backends [26, 23].

The code generation facilities in LLVM are retargeted from *tablegen* specifications. These specifications provide a model of the processor's hardware resources, register files, and instruction set. The instruction selector is customized by translation rules based on tree patterns that are also specified using the *tablegen* tool. Several register allocators are available for LLVM, most of them target specific code generation scenarios, e.g., the *local* allocator aims for fast code generation, while the *bigblock* allocator is intended for programs with particularly large basic blocks. For general purpose code generation tasks an extended *linear scan* register allocator is available. All these allocators are based on the same *tablegen* specification, which simply consists of an enumeration of the individual registers and register classes of the target processor. In addition, sub-registers and conflicting registers can be specified for every register definition.

The instruction selection is a *single-pass bottom-up tree matcher* operating on a directed acyclic graph. In comparison to traditional tree pattern matching, this approach offers limited capabilities. For example, it is not possible to specify non-terminals for the translation rules, neither does the instruction selector guarantee compatible register classes between the IR fragments matched by different rules. The compiler developer has to ensure that the individual tree patterns are compatible to each other. Cost functions are also missing, instead, an implicit cost model is used, where specific tree patterns are favored over less specific, e.g., a pattern that matches a node and its child nodes is favored over a pattern that matches the node alone. The emit functions are specified as a DAG of machine-level instructions, which replace the original IR nodes in the DAG. The original IR DAG is transformed into a machine-specific DAG stepwise.

A *list scheduler* then operates on a slightly transformed version of this DAG in order to derive an initial ordering of the instructions before register allocation. A framework for postpass scheduling is currently not available with the original LLVM framework.[16] We have thus added a basic-block-based postpass list scheduler to LLVM, which is able to group instructions into VLIW bundles. Both schedulers use traditional *resource tables* to detect conflicting instructions and resolve hazards during the instruction reordering.

[16] As of this writing such a postpass scheduling framework will be available with the next LLVM release 2.6

5.1.6 The *acc* Backend

The *acc* compiler is a proprietary research backend that was developed in conjunction with the xADL language and its tools. The backend serves as a testbed for the development of new optimization techniques for embedded systems. The abstraction of the target machine provided by the processor descriptions allows to quickly compare the effects of different optimizations and processor variants.

A modified version of GCC 4.1.1 exports the internal intermediate representation of the source program to an XML file, which is imported into our own high-level intermediate language. High-level constructs, such as array references, function calls and variable declarations, are then converted into a processor-independent low-level form. In addition, an ABI interface allows to rewrite function calls and global symbols. Instruction selection is performed using a modified version of *lburg* [65, 64], a tree pattern matcher generator. Virtual registers are replaced with hard registers using an extended Briggs-style global *graph coloring register allocator*; see [169]. A *list scheduler* is invoked before and after register allocation that reorders machine instructions to heuristically minimize the number of stalls. Structural and data hazards are detected during the scheduling using operation tables [174, 146].

The register allocator is customized using a set of C macros that define the available registers and register classes of the target processor. Similar macros also specify the hardware resources and operation tables for instruction scheduling. The tree rules for the instruction selector follow the conventions of the *burg/iburg* tool [65, 64]. The emit and cost functions consist of regular C/C++ code that is attached to the tree rules.

5.2 Register Specifications

The first step during the generation of a compiler backend is the enumeration of the available hardware registers of the target processor for the register allocator. Both, the LLVM and the *acc* backend, represent registers using symbolic names that are grouped by register classes. The register definitions are derived automatically from the register instances and their ports from the xADL processor description, where every register accessible through a port is represented separately. Register definitions of ports with the same bit-width and offset are collapsed.

For every register definition, an *alias set* specifies registers possibly conflicting with that register. It is straightforward to determine the alias set for a given hardware register by examining the bit-ranges accessible through other register ports of its register file. Sub-registers, i.e., registers that are completely covered by another hardware register, are similarly determined.

The register definitions are then grouped into classes that are also automatically derived from the xADL specification. More precisely, from: (1) constant registers, (2) subscript constraints attached to data and pipeline links connecting register ports

```
def R32_0: HWRegister<"0", [], []>;
def R32_1: HWRegister<"1", [], []>;
def R32_2: HWRegister<"2", [], []>;
def R32_3: HWRegister<"3", [], []>;
...
def RC_R_32 : HWRegisterClass<[i32], 32, [
  R32_2, R32_3, R32_4, R32_5, R32_6, R32_7, R32_8, R32_9,
  R32_10, R32_11, R32_12, R32_13, R32_14, R32_15, R32_24, R32_25,
  R32_16, R32_17, R32_18, R32_19, R32_20, R32_21, R32_22, R32_23,
  R32_0, R32_1, R32_26, R32_27, R32_28, R32_29, R32_30, R32_31]>;
```

Figure 63: Register and the register class definitions of the MIPS model for LLVM's register allocator.

to the data path, and (3) register classes defined by the programming conventions section. For every constant register a separate singleton register class is created. This class is not intended for the register allocator, but for the instruction selector, which elegantly matches the given constant to a dedicated non-terminal symbol. Register ports are natural candidates for register classes, because the register ports represent the actual operands of the instructions. Usually, all registers that are accessible via a register port are valid operands, but in some cases additional constraints need to be considered that are attached to the link connecting the register port to the data path. These constraints can be expressed using separate register classes that cover only a subset of the port's original register class. Equivalent classes of ports that have the same offset and bit-width, and that are connected through links with the same constraints are collapsed to a single definition. The register usage conventions specified using the programming conventions section of the xADL description are considered during the generation of the register classes. In particular, the reserved registers are marked as forbidden for the register allocation.

An excerpt of the *tablgen* specification for the LLVM register allocator is shown in Figure 63. All hardware registers are simply enumerated and assigned a unique identifier. The brackets on the right represent the set of sub-registers and aliased registers. The register structure of the MIPS is very regular, thus both sets are empty. The register class `RC_R_32` groups these registers. Caller-saved registers are favored over callee-saved and are thus printed first. Reserved registers and registers serving a special purpose, e.g., the frame pointer, are listed at the end of the class and further marked unusable for the register allocator.

5.3 Instruction Definitions

The individual instructions of the target processor are declared using a unique identifier, a syntax string, input and output operand sets, and additional information on

```
def I_JAL: BaseInstruction<"jal $ImmJ", (outs ),
                           (ins Imm26:$ImmJ, variable_ops)>
{
  bits<16> ADLInstr = 49;
  let isCall = 1;
  let Defs = [
    R32_0, R32_1, R32_2, R32_3, R32_4, R32_5, R32_6, R32_7,
    R32_8, R32_9, R32_10, R32_11, R32_12, R32_13, R32_14, R32_15,
    R32_24, R32_25, R32_26, R32_27, R32_28, R32_29, R32_30, R32_31,
    hi32, lo32];
}
```

Figure 64: Example definition of the MIPS *jump and link* instruction for the LLVM compiler infrastructure.

side-effects. The required information can be extracted easily from the instruction set model of an xADL processor description. The operand sets of the instructions are readily available from the internal representation of the *adlgen* tool and can directly be reused without further processing. Register operands are annotated with a register class that lists the set of legal registers for the instruction. Further, equality constraints for the register allocator are derived. These constraints specify which register operands have to be allocated to the same hardware register. The register allocator of the *acc* framework is slightly more powerful and allows multiple equality constraints per instruction, while the LLVM framework supports a single constraint only. The instruction definitions for LLVM are further restricted to a single output operand due to limitations of the *tablegen* instruction selector specifications. The syntax string is built from the syntax bindings attached to the respective instructions. Bindings associated with an operand are replaced by corresponding placeholders and mapped to the respective operand definition.

In addition, flags are attached to every instruction indicating the relevant side-effects that need to be considered during code generation. Memory accesses are indicated using the *load* and *store* flag respectively. Branches, jumps, function calls, return instructions, as well as trapping instructions are marked accordingly using the *branch*, *terminator*, *call*, *return*, and *barrier* flags. Side-effects on registers are further specified using additional implicit operands via the *defs* and *uses* attributes. For instructions that represent a function or subroutine call the *defs* attribute consist of all callee-saved registers that are possibly destroyed during the execution of the function. This automatically ensures that the register allocator generates the proper spill code to save the register values to memory. The notational conventions for this information is slightly different for the LLVM and the *acc* framework, however, the captured instruction properties are the same for both backends.

Figure 64 shows an example definition of the MIPS *jump and link* instruction for the LLVM backend. According to the jump provider, the instruction performs

a register-indirect jump and stores the original program counter to register 31 of register instance `R`. The instruction definition is thus augmented with the `isCall`, and `Defs` attributes.

5.4 Resource Models

The resource model specifies an abstract representation of the processor's hardware resources, i.e., its functional units, register ports, and storage elements, for the instruction scheduler. The scheduler reorders the instructions of a basic block such that the expected execution time is minimized and the processor's resources are utilized optimally. Resources that are blocked due to the execution of another instruction thus need to be tracked during the scheduling phase. In the case of the *acc* compiler a list scheduler relies on *operation tables* to recognize data and structural hazards, while the scheduler of the LLVM framework relies on traditional *resource tables*.

5.4.1 Resource Tables for the LLVM Compiler

The processor's resource model is accessible to LLVM's instruction scheduler using a standardized interface, the *hazardrecognizer*. The interface returns whether the candidate instruction selected by the scheduler is safe to be scheduled or, in case of an hazard, whether the hazard is resolved automatically by the hardware or needs to be resolved explicitly.

Blocked resources are represented by an array of bits, where the columns represent the resources usage and the rows the utilization over time. Instructions are associated with similar bit-arrays, representing the blocked resources on a cycle-per-cycle basis during the execution of the instruction. In the case of VLIW processors, instructions are often associated with multiple such arrays representing different *instruction variants*. The behavior of instruction variants is identical, except that different variants are executed by different computational resources of the target processor, i.e., variants correspond to the instruction slots of a VLIW bundle.

The flow of an instruction through the pipeline is exactly represented by its instruction path. An abstract resource model can thus be computed elegantly from the instruction paths, where the ports connected by the path's hyperedges correspond to resources. Note that it is safe to ignore the data and pipeline links of the path, even though the links represent hardware resources by themselves. The ports at the head and tail of the involved links are blocked by the resource model anyway, thus there is no need to explicitly represent the links in the resource model. The number of resources in the table is further reduced by eliminating redundant entries. For example, resources that are only used by a single instruction or resources that are always blocked during the same cycle can never cause conflicts and are eliminated in order to minimize the overhead during scheduling.

For parallel processors, such as VLIW architectures, multiple instruction variants may be available. These variants are represented by a single instruction definition in order to avoid premature resource assignments. However, during the postpass scheduling step the instruction variants become vital, because the final code layout is computed and the instructions are grouped into bundles for parallel execution. The scheduler thus has to ensure that only legal instruction variants are grouped into a bundle according to the constraints defined by the instruction encoding. An automaton derived from the binary coding specifications of the processor model is thus integrated with the *hazardrecognizer*.

5.4.2 Operation Tables for the *acc* Backend

The list scheduler of the *acc* backend relies on an advanced resource model proposed by Shrivastava et al. [174, 146]. *Operation tables* are an extension of the traditional resource table approach that is able to handle irregular and partial bypassing configurations of the target processor. Operation tables consist of a traditional resource table, which is derived from the xADL processor model as described above. In addition to structural hazards caused by resource conflicts, an operation table also captures the data flow through the processor pipeline and the state of the bypass logic using three *operations*: (1) *readOperand*, (2) *writeOperand*, and (3) *destOperand*. These operations specify the cycles when register operands of the instruction are read or written from/to either a register file or a pipeline register. If bypasses are present for a register operand, the corresponding read operation specifies multiple source registers. The *destOperand* occurs once for every register operand written by the instruction and helps to detect violations of output dependencies during the scheduling process.

In contrast to traditional resource tables the data flow of the register operands cannot be extracted from the instruction paths alone. Additional information from the behavioral instruction model is required that is gathered by traversing the instruction's micro-operations. A first forward traversal recursively follows `move` micro-operations starting from a micro-operation reading a register operand. The information on hazard links is collected from the visited operations and corresponding *readOperations* appended to the operation table of the instruction. The *writeOperations* are similarly computed by a backward traversal starting from `move` micro-operations writing a register operand. For every destination operand a corresponding *destOperation* is added to the operation table at the same cycle as the first *writeOperation*.

Figure 7 presents the operation table of the *add unsigned* instruction of the MIPS processor model. The sum of the operands *Rs* and *Rt* is computed by the *EX* unit on cycle three. The values of these operands can be retrieved either from the general purpose register file *R* or via a bypass from the pipeline registers $EX::Rd$ or $MEM::Rd$. The destination register *Rd* on the other hand is written to these

Cycle	Resources	Operations	
1	FE		
2	DE		
3	EX	read	**Rs**: {R, EX::Rd, MEM::Rd}
		read	**Rt**: {R, EX::Rd, MEM::Rd}
		write	**Rd**: EX::Rd
		dest	**Rd**: R
4	MEM	write	**Rd**: MEM::Rd
5	WB	write	**Rd**: R

Table 7: Operation table of the *add unsigned* instruction of the MIPS processor.

pipeline registers at cycles three and four, the result is finally committed to the general purpose register file in cycle five.

5.5 Instruction Selector Specifications

The instruction selector is the central component of a retargetable compiler backend. It maps the compiler intermediate representation of the input program to machine instructions, such that the expected execution time and code size of the resulting machine code is minimized. The backend generator of the *adlgen* tool is capable to automatically derive tree patterns, emit functions, and cost functions for tree-pattern-based instruction selectors.

5.5.1 Representing Tree Rules

The traditional representation of the instruction selector specifications as defined in Section 5.1.1 associates the production rules of the tree grammar with a cost and emit function. Usually, these functions are represented as plain code, typically C/C++ code. The cost functions, and foremost the dynamic checks, inspect global flags and properties of the IR nodes matched by the respective rule instances, while the emit functions construct a data structure that represents the emitted machine instructions. Source code is impractical for a generator tool that transforms and combines rules.

We thus extend the tree grammar representation of traditional instruction selectors. The dynamic checks are explicitly represented using *conditions* that are attached to the individual terminal and non-terminal symbols of the grammar's production rules. Conditions are represented as a conjunctions of *simple* conditions, each testing a single global property or a single property of the IR node matched by the respective terminal or non-terminal symbol. A simple condition in turn is a subset test over a given domain, e.g, the range from 0 to 65535 over the set of integer numbers.

Definition. A *tree grammar with conditions* is defined by $G = (S, N, \mathcal{F}, R, D)$, where S, N, and $\mathcal{F}$ correspond to the start, non-terminal, and terminal symbols of regular tree grammars. $D = \mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n)$ is the cartesian product of the powersets of the domains of the simple tests. The production rules in R are slightly modified, a rule consists of a non-terminal on the left hand side and a tree with conditions on the right hand side $q \to t$, where $q \in N$ and $\alpha \in T'(N \cup \mathcal{F}, D)$:

$$
\begin{aligned}
&(1)\ N \times D \subseteq T'(N \cup \mathcal{F}, D),\\
&(2)\ \mathcal{F}_0 \times D \subseteq T'(N \cup \mathcal{F}, D),\\
&(3)\ \forall p \geq 1, f \in \mathcal{F}_p, d \in D, t_1, \ldots, t_p \in T'(N \cup \mathcal{F}, D)\colon\\
&\qquad (f(t_1, \ldots, t_p), d) \in T'(N \cup \mathcal{F}, D).
\end{aligned}
$$

Definition. A term $t \in T'(N \cup \mathcal{F}, D)$ can also be represented as a partial function $t : \mathbb{N}^* \to N \cup \mathcal{F} \times D$ with the domain $\mathcal{P}os(t)$ of *positions*:

$$
\begin{aligned}
&(1)\ \mathcal{P}os(t) \text{ is nonempty and prefix-closed,}\\
&(2)\ \forall p \in \mathcal{P}os(t), t(p) \in \mathcal{F}_n \times D, n \geq 1\colon \{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\},\\
&(3)\ \forall p \in \mathcal{P}os(t), t(p) \in \mathcal{F}_0 \cup N \times D\colon \{j \mid pj \in \mathcal{P}os(t)\} = \emptyset.
\end{aligned}
$$

The positions uniquely identify sub-terms within a term, where the root position is represented by the symbol ϵ. For example, consider the term $t = f(a, g(b, c))$, where conditions are omitted for the sake of brevity. The root symbol is denoted by $t(\epsilon) = f$, similarly, the position of symbol c is 11, i.e., $t(11) = c$.

Definition. A fragment of the compiler intermediate representation corresponding to a ground term $t \in T'(\mathcal{F}, D)$ can be covered by a production rule of the instruction selector $r : q \to t'$, denoted by $matches(r, t)$, if the following conditions are satisfied:

$$
\begin{aligned}
&(1)\ \forall p \in \mathcal{P}os(t'), t(p) = (f, c), t'(p) = (e, d), e \in \mathcal{F}\colon f = e,\\
&(2)\ \forall i \in \{1, \ldots, n\}, p \in \mathcal{P}os(t'), t(p) = (f, (c_1, \ldots, c_n)),\\
&\qquad t'(p) = (e, (d_1, \ldots, d_n))\colon c_i \subseteq d_i.
\end{aligned}
$$

Condition (1) ensures that the pattern of the rule actually matches the IR nodes of the fragment, where arbitrary IR nodes are accepted on positions corresponding to non-terminal symbols in the rule pattern. Furthermore, condition (2) ensures that the hidden properties of the IR nodes match the conditions attached to the symbols of the pattern.

Definition. The cost function of a rule $r \in R$ applied to an IR fragment $t \in T'(\mathcal{F}, D)$ is defined as:

$$
cost(r, t) = \begin{cases} n \in \mathbb{N} & \text{, if } matches(r, t)\\ \infty & \text{, otherwise.} \end{cases}
$$

Definition. The emit function of a rule $r : q \rightarrow t$, denoted by $emit(r)$, is represented as an ordered multi-set of pairs over $\mathcal{I}nstrs \times B$ of instructions and *operand bindings*, with the following kinds of operand bindings:

$$(1)\ ops_o(i) \times \{new\_reg\},$$
$$(2)\ ops_i(i) \times \mathcal{P}os(t),$$
$$(3)\ ops_i(i) \times \mathbb{N},$$
$$(4)\ \forall (j, b) \in emit(r) \colon ops_i(i) \times ops_o(j),$$

where $ops_i(i)$, $ops_o(i)$, $ops_o(j)$ denote the input and output operands of instructions i, $j \in \mathcal{I}nstrs$.

The operand bindings specify the assignment of the instruction's operands to (1) a newly created temporary virtual register, (2) a virtual register, constant, or symbol of the IR, (3) a numeric value, or (4) an output operand of an instruction emitted by the current rule. The notation for operand bindings (2) and (3) can easily be confused, we thus use the convention that the positions of bindings of kind (2) are underlined.

Corollary 1. Equality constraints between two IR nodes covered by a rule $r \rightarrow t$ can be expressed using operand bindings of the form (o, p), (o, q), where $p, q \in \mathcal{P}os(t)$, $p \neq q$. The nodes are required to be at leaf positions, i.e., $\nexists j \in \mathbb{N} \colon pj \in \mathcal{P}os(t) \vee qj \in \mathcal{P}os(t)$. The set of tree languages defined by tree grammars with conditions is thus non-regular.

5.5.2 Deriving Non-terminals

Non-terminals can be seen as a kind of *temporary variables* in the tree grammar that allow to chain different rules together. The backend generator uses non-terminals to represent immediate and register operands defined by the processor description. In addition, an artificial start non-terminal *stmt* is provided that is produced by rules without a result, e.g., branches and store instructions.

The concrete data representation of immediates is not explicitly defined by the xADL language, but is implicitly given by the micro-operations using them. Thus *all* immediates are mapped to a single non-terminal *immediate* that matches symbols and constants in the IR – see Figure 65(a). Immediates with limited bit-width may only represent a subset of the legal constant values of the source language. The matching rules thus need to verify that the involved data representations, i.e., the data type, bit-width, and signedness, of the IR and the rule pattern match. This is ensured by appending proper conditions to the respective rule patterns as explained later.

Registers and their associated ports correspond to non-terminals during instruction selection. This ensures that the result register generated by a rule meets the

(1) $immediate \rightarrow INT\_CONST$
(2) $immediate \rightarrow SYMBOL$

(a) Matching symbols and constants.

(1) $\texttt{RC\_R\_32} \rightarrow VREG\{\texttt{RC\_R\_32}\}$
(2) $\texttt{RC\_R\_32}_0 \rightarrow INT\_CONST\{0\}$

(b) Covering virtual register.

Figure 65: Default rules generated for register and immediate non-terminals.

register constraints of the respective consumer. Separate non-terminal symbols are created for the register classes derived for the register allocator as described in the previous section. Matching rules are then generated to match the terminal symbol *VREG*, which represents virtual registers in the IR, that are augmented with conditions to ensure that the register class of the virtual register matches the register class represented by the rule's non-terminal – see Figure 65(b). Conditions are represented by curly braces following the terminal symbol in the tree pattern. In this example, only virtual registers of register class `RC_R_32` and the constant zero can be matched.

5.5.3 Deriving Conversion Rules

Two register ports access possibly overlapping ranges of bits of the same hardware register. If two ports overlap, data written to one port can be read through the other and vice verse. This is equivalent to a conversion of the corresponding register non-terminals in the tree grammar at no cost. However, the conversion is only legal, if the data representation of the IR node can be preserved, i.e., the bit-width of the destination port is sufficient to hold the data type of the involved IR nodes. This is ensured by appending a proper value range condition to the respective conversion rule.

It is also possible to convert the non-terminals of non-overlapping register ports using certain bit-manipulation operations, e.g., shifts. However, these conversions are not for free, because the data has to be transferred from the bit-range of the source register to the bit-range of the destination register explicitly. The bit-manipulation instructions can be found using tree patterns using *templates* that will be described in more detail in Section 5.5.5.

Consider, a 32-bit register file `R` with two ports, where the first port accesses a complete base register and the latter accesses the upper half of a base register only. The two ports obviously overlap, it is thus possible to read data written to one port via the other. In the IR this corresponds to a shift operation, as depicted by rules (1) and (2) in Figure 66. The rules are only applicable, if the signed respectively the unsigned data representation fits into 16 bits as denoted by the conditions in curly braces. An assignment to the sub-register can also be interpreted as a bit-insertion operation shown by rule (3). Data can be transferred between the respective register ports explicitly using a shift left operation (4), or using a logical or arithmetic shift

(1) $\texttt{RC\_R\_16\_31} \rightarrow SHR\{0, \ldots, 65535\}(\texttt{RC\_R\_32}, 16)$
(2) $\texttt{RC\_R\_16\_31} \rightarrow ASHR\{-32768, \ldots, 32767\}(\texttt{RC\_R\_32}, 16)$
(3) $\texttt{RC\_R\_32} \rightarrow INSERT(\texttt{RC\_R\_32}, \texttt{RC\_R\_16\_31}, 16, 16)$
(4) $\texttt{RC\_R\_16\_31} \rightarrow \texttt{RC\_R\_32}\{-32768, \ldots, 65535\}$
(5) $\texttt{RC\_R\_32} \rightarrow \texttt{RC\_R\_16\_31}\{0, \ldots, 65535\}$
(6) $\texttt{RC\_R\_32} \rightarrow \texttt{RC\_R\_16\_31}\{-32768, \ldots, 32767\}$
(7) $\texttt{RC\_R\_32} \rightarrow \texttt{RC\_R\_32}_0$

Figure 66: Conversion rules derived from overlapping register ports and constant registers.

right as depicted by rules (5) and (6) respectively. Finally, constant registers can be converted to regular register operands using rule (7).

5.5.4 Initial Rule Set

Deriving mapping rules from the instructions of an architecture is more complex, as instructions may have side-effects. Currently, two classes of side effects are considered: (1) memory accesses and (2) control flow changes, e.g., through branches or exceptions. The information on these side-effects is retrieved from the corresponding provider.

Due to limitations of the tree pattern matching approach, only a subset of the memory access operations can be considered by the backend generator. Instructions that access a memory and write to at least one register port are ignored during the processing, because these instructions can only be modeled using a DAG pattern, but not using a tree. Furthermore, it is not safe to use instructions accessing memory for computations other than the memory access. For example, a memory store with an autoincrement addressing mode cannot be used to increment a register, because of the unpredictable side-effect on the memory state. Thus, a matching rule is only generated for instructions recognized as simple load and store operations by the memory provider. The non-terminal produced by the matching rule is derived from the destination register for load instructions. Store instructions do not compute a result that needs to be represented by a non-terminal symbol, thus the start symbol

(1) $\texttt{RC\_R\_32} \rightarrow LD(+(\texttt{RC\_R\_32}, immediate\{0, \ldots, 65535\}))$
(2) $stmt \rightarrow ST(\texttt{RC\_R\_32}, +(\texttt{RC\_R\_32}, immediate\{0, \ldots, 65535\}))$

Figure 67: Rule patterns derived from the *load word* and *store word* instructions of the MIPS model.

(1) $stmt \rightarrow GOTO(immediate)$
(2) $stmt \rightarrow COND(= (\texttt{RC\_R\_32}, 0), GOTO(immediate))$

Figure 68: Rule patterns derived from the *jump* and *branch on zero* instructions of the MIPS processor.

stmt is assigned to the corresponding rules. The rule patterns simply consist of a *LD* respectively a *ST* terminal symbol, where the sub-trees represent the addressing mode, which is supplied by the memory provided. Figure 67 shows two example rules derived from the *load word* and *store word* instructions of the MIPS processor model.

In many compiler backends, a post-processing pass, e.g., a peephole optimizer, recombines instructions with advanced addressing modes to circumvent this limitation. Recently, an extension to tree-pattern-based instruction selection has been presented by Ebner et at. [45, 44] that is able to process patterns in the form of general DAGs. The compiler generator in its current form does not support these DAG patterns, but an extension is planned in the future in order to exploit the benefits of this approach.

The control flow behavior of an instruction is similarly analyzed and classified by the jump provider. If the jump analysis fails to analyze the branching behavior, the respective instruction is not considered by the backend generator. For other instructions recognized as branches, specialized branch patterns are constructed. Symbol-based branches that do not have other side-effects can be used to realize the control flow within functions of the source language. Register-indirect absolute branches, on the other hand, can be used to return from functions, only if the register operand is compatible with the return address register specified by the programming conventions. Similarly, branches that store the original value of the program counter can be used to model function calls, if the destination register is compatible with the return address register. As for the addressing modes of memory operations, the micro-operations that compute the branch target are not directly considered by the backend generator. Instead, the summary computed by the jump analysis provides the rule patterns. Example rules generated from the *jump* and *branch on zero* instructions of the MIPS processor are shown in Figure 68.

Considering instructions without memory and control flow side-effects only, mapping rules are created by processing the respective micro-operations. A rule is created for every assignment to a register port marked as data register. Micro-operations supplying values to the assignment are then added to the rule pattern and mapped to corresponding terminal symbols using a look-up table. Values produced by preceding micro-operations are translated to patterns recursively until a micro-operation is encountered reading the value of a register port or immediate. Register operands are represented by the non-terminal of their register class, while immediates are represented by the *immediate* non-terminal in the resulting tree pat-

tern. During the traversal `move` micro-operations are skipped, i.e., the rule pattern is not modified when a `move` is visited.

Along with the construction of rule patterns, conditions are created that have to be satisfied for a mapping rule to be applicable during instruction selection. The actual conditions depend on the currently considered micro-operation and differ from case to case. For example, consider the `mulu` micro-operation, which interprets its input operands as unsigned integer values and performs an unsigned multiplication. A corresponding condition is thus attached to the rule that restricts the multiplication to values greater than zero. The `trunc` micro-operation, which simply discards the upper bits of its input operand, similarly restricts the value range of the matched IR node to fit into its output operand. Again, a corresponding condition is appended to the rule pattern. Other micro-operations, such as `sext` and `zext`, imply similar constraints.

The emit functions are also constructed in parallel with the tree patterns. When ever a `move` micro-operation is encountered that reads a register or immediate operand of the instruction, a corresponding operand binding is appended to the emit function. The position specifier of the binding can easily be computed during the construction of the tree pattern and is automatically updated.

The rules are further associated with a cost derived from the instruction's worst-case latency computed by the timing provider.

Consider for example, the micro-operations of the *or immediate* instruction of the MIPS processor depicted in Figure 69. Two register assignments can be found for this instruction: (1) an update of the program counter register on line three, and (2) an assignment to register port `Rd` of the register file `R` on the last line. The branch and memory analyses yield no side-effects, a selection rule is thus constructed for this instruction; see Figure 70. First, the instruction's destination register determines the non-terminal on the left hand side of the rule, the rest of the pattern is not yet known and represented by the symbol '_'. During the backward traversal several `move` micro-operations are skipped until the `or` micro-operation is encountered, which causes the *OR* terminal symbol to be appended. This process is continued until micro-operations are encountered that read the operands of the instruction.

```
(1)  FE::pc_i = move(pc::p_fe)
(2)  FE::pc_o = add(FE::pc_i, 4)
(3)  pc::p_fe = move(FE::pc_o)
(4)  ICache::@read = move(FE::pc_o)
(5)  ICache::read = read(ICache)
(6)  DE::ImmW_i = move(ImmW)
(7)  DE::Rs_i = move(R::Rs[0,31])
(8)  DE::IW_i = move(ICache::read)
(9)  decode(IW_i)
(10) DE::Rs_o = move(DE::Rs_i)
(11) DE::ImmWu_o = zext(DE::ImmW_i)
(12) EX::ImmWu_i = move(DE::ImmWu_o)
(13) EX::Rs_i = move(DE::Rs_o)
(14) EX::Rd_o = or(EX::Rs_i,
                   EX::ImmWu_i)
(15) MEM::Rd_i = move(EX::Rd_o)
(16) MEM::Rd_o = move(MEM::Rd_i)
(17) WB::Rd_i = move(MEM::Rd_o)
(18) WB::Rd_o = move(WB::Rd_i)
(19) R::Rd[0,31] = move(WB::Rd_o)
```

Figure 69: Micro-operations of the *or immediate* instruction.

μ-op.	Tree pattern	Operand Bindings
(6)	RC_R_32 $\rightarrow OR($RC_R_32$, ZEXT_{16}(immediate))$	$(ImmW, \underline{21})$
(7)	RC_R_32 $\rightarrow OR($RC_R_32$, ZEXT_{16}(_))$	$(Rs, \underline{1})$
(11)	RC_R_32 $\rightarrow OR(\_, ZEXT_{16}(\_))$	
(14)	RC_R_32 $\rightarrow OR(\_, \_)$	
(19)	RC_R_32 $\rightarrow$ _	(Rd, new_reg)

Figure 70: Tree patterns constructed from the micro-operations of MIPS' *or immediate* instruction.

The processing stops here, after the corresponding non-terminal symbols have been appended to the pattern. The operand bindings $(Rs, \underline{1})$ and $(ImmW, \underline{21})$ are created simultaneously. The first binds the register operand `R::Rs` to the virtual register represented by the first child of the OR, while the second associates the immediate operand `ImmW` with the corresponding non-terminal. The immediate operand is further restricted by a condition $\{-32768, \ldots, 65535\}$ that is not shown by the figure. The MIPS processor supports the bypassing of register values, the timing analysis yields a worst-case execution latency of one cycle for the *or immediate* instruction, which completes the construction of the instruction selection rule.

Note that tree patterns may only produce a single result. This restriction does not apply to the micro-operations in xADL. To overcome this shortcoming, we duplicate rules for each result produced. The same situation arises with instructions having multiple results. We generate multiple independent rules and try to recombine redundant instructions using a post-processing pass after the instruction selection is completed. The side-effect on the respective other destination registers are either suppressed by assigning constant registers to the operands, if applicable. Otherwise, the register allocator is responsible to pick an available register and if necessary generate the spill code.

5.5.5 Specializations and Templates

In order to derive a complete code selector, each operator of the IR has to be covered by at least one rule. However, in general the instruction set will not directly match all the required operations in the IR. One such case occurs, when a particular operation in the IR is simulated using a more general instruction by hard-wiring some of its inputs. Likewise, certain useful operations can be obtained by forcing the input operands to be equal. However, sometimes a single instruction is not sufficient, missing operations have to be emulated by combining multiple instructions.

Specializations are simplifications of existing rule patterns by additional conditions, by modifying the operand bindings, or using algebraic laws. Numerous such specializations are supported by the current compiler generator implementation. Besides the general commutativity, associativity, and distributivity rules, simplifications that eliminate terminal symbols from the rule patterns play a central role. For

Tree Pattern		Operand Binding
(1)	$\texttt{RC\_R\_32} \rightarrow OR(\texttt{RC\_R\_32}, immediate\{0,\ldots,65535\})$	$(Rs, \underline{1}), (ImmW, \underline{2})$
(2)	$\texttt{RC\_R\_32} \rightarrow OR(immediate\{0,\ldots,65535\}, RC\_R)$	$(Rs, \underline{2}), (ImmW, \underline{1})$
(3)	$\texttt{RC\_R\_32} \rightarrow immediate\{0,\ldots,65535\}$	$(Rs, 0), (ImmW, \underline{\epsilon})$
(4)	$\texttt{RC\_R\_32} \rightarrow \texttt{RC\_R\_32}$	$(Rs, \underline{\epsilon}), (ImmW, 0)$

Figure 71: Specialization applied to the original rule of the *or immediate* instruction.

example, a + terminal symbol can be eliminated by forcing one of its operands to zero, similarly, a logical not can be derived from a nor operation by supplying the same input operands twice. Unary operators can often be eliminated by restricting the input operands using conditions. A typical example is the $TRUNC_n$ terminal symbol, which represents a truncation of its input operand to n bits. This symbol can be eliminated using an additional condition, which ensures that the input operands fit into the truncated data representation. Similar specializations can be performed for the $SEXT_n$, $ZEXT_n$, and ABS symbols.

Example specializations of the initial rule computed for the *or immediate* instruction are shown in Figure 71. The first rule is derived by eliminating the $ZEXT_{16}$ terminal symbol, which further restricts the value range of the immediate operand. Rules (2) is created through the application of the commutativity law. Next, the OR terminal symbol itself is eliminated by forcing either the register operand (3) or the immediate operand (4) to zero.

Templates on the other hand combine multiple independent instructions to form new translation rules. A template consists of a set of *required* tree patterns optionally containing free variables. The tree patterns specify, in an architecture independent way, which *instructions* need to be available to emulate the desired operation. The backend generator repeatedly checks, if a particular template is applicable, i.e., if all required tree patterns are available, and then invokes a *combine* function. The combine function specifies how the rules matched by the required pattern are to be combined with each other. The *adlgen* tool already provides a collection of predefined templates that are applicable to all processor models. Those templates are kept in a separate module and can be easily extended if necessary.

$$(1)\ \%TmpL \rightarrow SHL(\%Value, INT\_CONST\{\%n\})$$
$$(2)\ \%Result \rightarrow ASHR(\%TmpL, INT\_CONST\{\%n\})$$

(a) Required tree patterns.

$$\%Result \rightarrow SEXT_{\%n}(\%Value)$$

(b) Resulting pattern

Figure 72: Template to match the sign-extend operator.

For instance, the semantics of a sign-extend operation can be simulated by a combination of shifts. Figure 72 depicts the required, as well as the resulting tree patterns. Variables, proceeded by a '%' symbol, always have to represent the same term for all required tree patterns in order for the template to be applicable, e.g., the non-terminal symbol on the left hand side of the top-most rule is required to be compatible with the first operand of the arithmetic shift right operation of the second rule. The resulting pattern is constructed by replacing the variables with the concrete trees of the matched rules.

5.5.6 Emitting the Instruction Selector Specification

The rule set derived from a xADL processor model is shared among the *acc* and LLVM compiler generators. However, the instruction selector specifications of the two compiler frameworks not only differ in their syntax, but also the foundations of the intermediate representation and the instruction selectors themselves are different. Compiler-specific emitter components are available with the *adlgen* tool that processes the rule set prior to the actual generation of the instruction selector specification for the particular compiler.

In the case of the *acc* backend this processing is minimal, because the dynamic programming approach adopted by its instruction selector closely follows the tree grammar model presented above. The terminal symbols are mapped to the actual *tree codes* used by the backend's intermediate representation and C/C++ code is generated from the conditions and operand bindings. Figure 73 shows the resulting code generated for rule (1) of the *or immediate* instruction from Figure 71. The `tpm_width` function is a compact form of the immediate's value range condition. The `TPM_EMIT` macro constructs the machine-level representation of the *or immediate* instruction. Operand bindings are represented using the arrays `ops0` and `k`, which hold the machine-level operands of the current instruction and the operands generated by the emit functions of the rules matching the child nodes.

The processing for the LLVM framework is slightly more complicated, due to the restrictions of the simple single-pass instruction selector. The main problem is that

```
RC_R_32: BIT_IOR_EXPR(RC_R_32, __immediate)
%cost %{ tpm_width(t->tpm_kid(1), 16, 1) ? 1 : INF_COST %}
%code %{
  EMIT_RESULT d0, ops0[3] = {d0 = NEW_REG(RC_R_32), k[0], k[1]};
  TPM_EMIT(66, ops0);     /* ori $[0-31], $[0-31], <imm16>  */
  return d0;
%};
```

Figure 73: Final instruction selection rule for the *acc* backend.

```
// mult $[0-31], $[0-31]
def RP0072: Pat<(mul RC_R_32:$R_Rsa, RC_R_32:$R_Rtb),
                  (I_MFLO_R
                    (I_MULT_lo RC_R_32:$R_Rsa, RC_R_32:$R_Rtb))>;
```

Figure 74: Final instruction selection rule for the LLVM backend.

the instruction selector does not support non-terminal symbols. It simply assumes that all rules are compatible to each other, i.e., all the rules produce the same implicit non-terminal. In the case of regular processors with a single register class, this is not a problem, because the results produced by one instruction are guaranteed to be compatible with all its consumers. For other processors, however, invalid code might be generated, if the register classes of the producer and consumer instructions do not match. The backend generator thus has to transform the rule set such that all rules produce a common *base non-terminal*, which is specified manually by the user via a command line option of the *adlgen* tool. Rules that do not produce the base non-terminal by default are combined with conversion rules that explicitly convert the rule's non-terminal to the base non-terminal. The resulting copy operations can often be eliminated by the register allocator, the impact on the code quality is thus negligible. Consider the example in Figure 74, the MIPS *multiply* instruction does not write its result to the general purpose register file represented by the register class `RC_R_32`. Instead, the result is written to the special `HI` and `LO` registers. However, the *move from lo* instruction allows to copy the multiplication result to the general purpose register file. The instruction selection pattern generated for the LLVM backend emits both instructions to ensure correct code.

5.6 Completeness of Instruction Selector Specifications

In the last section we have shown how an initial rule set for an instruction selector can be derived from the instructions of an xADL processor model, and how *specialization* and *templates* help to improve the final code quality and increase the covered tree patterns. Even though these mechanisms are quite powerful in discovering additional rule patterns, they still rely on the instructions provided by the target processor. Consequently, if the capabilities of the processor are restricted, the number of rule patterns discovered by the backend generator may severely impact the *completeness* of the instruction selector. It may well be the case that the instruction set is so restricted that a complete instruction selector cannot be derived at all. It is thus important for a backend generation tool to provide feedback on the completeness of the derived rule set. A major problem for such a completeness test are *dynamic checks* that may cause rules to be disabled because of properties that are not represented by the terminal symbols of the intermediate representation. These properties are referred to as *hidden* properties.

$v \rightarrow INT\_CONST$	$\{-\infty, \ldots, \infty\}$
$v \rightarrow +(v, v)$	$\{-\infty, \ldots, \infty\}$

(a) IR specification

$r \rightarrow INT\_CONST$	$\{-32768, \ldots, 32767\}$
$r \rightarrow INT\_CONST$	$\{0, \ldots, 65535\}$
$r \rightarrow +(r, r)$	$\{-\infty, \ldots, \infty\}$

(b) Instruction selector specification

Figure 75: Example specifications using dynamic checks.

Figure 75 presents an excerpt from a specification of a compiler's IR and an instruction selector using a tree grammar notation. Rules matching the *INT_CONST* terminal symbol are associated with a dynamic check that allows only certain constant values to match. In the case of the IR specification this condition covers the complete range of legal values for integer constants, while the target architecture is only capable of processing signed and unsigned 16-bit integer constants. Clearly, the instruction selector is not complete for programs that make use of larger constants.

Previous approaches based on finite tree automata [48] cannot handle this example, because dynamic checks cannot be represented by traditional tree automata. The problem here is that the set of legal integer constants is subsumed by a single terminal symbol *INT_CONST*. Dynamic checks thus need to be expressed explicitly using a formal model. The instruction selector model presented in the previous sections is well suited for this task. Using *terminal splitting* the conditions can be split such that a traditional tree automaton can be constructed, where the dynamic checks and conditions are explicitly represented by dedicated terminal symbols. The preprocessed specification can then be verified by a traditional completeness test [48]. If the test fails, counter examples can be computed that provide valuable feedback to the processor designer. Based on the counter examples the designer may choose to extend the processor model to increase the coverage, or, if applicable, provide additional templates to the backend generator to emulate the missing operations using existing instructions.

Note that the completeness is only guaranteed in combination with an instruction selector that *always* finds a valid covering if one exists. The test is thus applicable to the *acc* backend, but not to the simple instruction selector of the LLVM framework.

5.6.1 Equality Constraints

According to Corollary 1, operand bindings of tree grammars with conditions can be used to express equality constraints. Equality constraints *cannot* be handled by the approach presented in the following, due to the observations from Section 5.1.2. The corresponding tree languages lack certain properties required for a completeness test to be feasible in general.

Rules with equality constraints are thus eliminated from the rule set prior to the completeness test. Note that this does not invalidate the test's result, i.e., if the

completeness of the reduced rule set can be proven, this property also holds for the original rule set. This is not true when the completeness test fails. The original rule set may actually be complete, even if the test fails. In practice this is very unlikely to be the case, due to the properties of typical compiler intermediate representations and processor instruction sets.

5.6.2 Preliminaries

We use *normalized* tree grammars with conditions – also see Section 5.5.1 – to represent the compiler's intermediate representation and the instruction selector specification.

Definition. A tree grammar with conditions $G = (S, N, \mathcal{F}, R, D)$ is in normal form, if all rules in R have one of the following forms:

$$q \rightarrow (f(q_1, \ldots, q_n), c),$$
$$q \rightarrow (a, c),$$

where $f, a \in \mathcal{F}$ are terminal symbols, $q_1, \ldots, q_n, q \in N$ are non-terminal symbols, and $c \in D$ is a condition. The terminal symbol matched by a given rule $r \in R$ can be obtained using the function $term(r) \rightarrow \mathcal{F}$, its condition using $cond(r) \rightarrow D$.

Every tree grammar can easily be normalized by introducing new *temporary* non-terminals [36]. For example, the following rule $v \rightarrow +(v, INT\_CONST)$, where the conditions are omitted, can be normalized using a new temporary non-terminal tmp as follows:

$$tmp \rightarrow INT\_CONST$$
$$v \rightarrow +(v, tmp)$$

Definition. A rule is said to be *unconditional* if its associated condition is always fulfilled, i.e., $\forall i \in \{1, \ldots, n\} \colon c_i = D_i$. A condition is *unsatisfiable* if $\exists i \in \{1, \ldots, n\} \colon c_i = \emptyset$, the corresponding rule can never be applied and can safely be eliminated.

In the following sections some operations and predicates on conditions are required.

Definition. Two conditions $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ *overlap*, if $\forall i \in \{1, \ldots, n\} \colon a_i \cap b_i \neq \emptyset$. Similarly the *intersection* of two conditions is defined as $a \bigcap b = (a_1 \cap b_1, \ldots, a_n \cap b_n)$.

Definition. An essential operation is to *split* the complete condition domain D into a set of non-overlapping conditions. Given a condition $c = (c_1, \ldots, c_n)$ the function $split : D \rightarrow \mathcal{P}(D)$ performs such a splitting by constructing conditions from all possible combinations of the simple tests c_i and their complements. Note that this guarantees $c \in split(c)$.

Consider for example, the domain $D = \mathcal{P}(D_1) \times \mathcal{P}(D_2)$ and a corresponding condition $c = (c_1, c_2) \in D$. The splitting calculated by $split(c)$ is $\{(c_1, c_2), (\overline{c_1}, c_2), (c_1, \overline{c_2}), (\overline{c_1}, \overline{c_2})\}$.

5.6.3 Terminal Splitting

In its original form the grammar G_{ir} that specifies the compiler's IR consists only of unconditional rules, while the grammar G_{is}, specifying the instruction selector, may also contain conditional rules. Although both grammars represent conditions, a direct matching between these two cannot be established yet. In order to establish such a mapping we need to split the conditions occurring in G_{ir} and G_{is} such that for all terminal symbols the conditions between the two grammars are either equal or do not overlap. More formally we require after this splitting that the following criterion is met.

Criterion 1. For each rule r_1 in G_{ir} and r_2 in G_{is}, where $term(r_1) = term(r_2)$ the following condition holds: $cond(r_1) = cond(r_2) \vee \neg overlap(cond(r_1), cond(r_2))$.

Because of this criterion a direct mapping between the terminal symbols and conditions occurring in one grammar with those of the other grammar is possible. For the final completeness test we simply treat the original terminal symbol of a rule and its condition as a single entity, which acts as a symbol during the actual completeness test.

Algorithm 1 performs the desired splitting. First, the original grammar of the instruction selector is copied and all rules are removed from the grammar. The grammar is then reconstructed rule by rule in three steps by splitting the conditions of the rules in G_{is} and G_{ir}. When a rule is re-added to G_{is}, the rules with overlapping conditions in G_{is} are first replaced by equivalent copies with split conditions. The conditions are computed by intersecting the original condition with those computed by *split*. Secondly, the rules in G_{ir} are processed in the same way. In step three, equivalent copies of the rule that is to be re-added are appended to G_{is}, where the conditions are derived from rules in G_{ir} that match the same terminal symbol and are associated with overlapping conditions. The resulting splitting ensures that the previously stated Criterion 1 is satisfied.

Proof (Sketch). Initially G_{is} is empty and all rules in G_{ir} are unconditional, thus Criterion 1 is satisfied. Performing the steps one and two of the algorithm clearly preserves this property. If a rule r_1 in G_{ir} and a rule r_2 in G_{is} share the same terminal and the same condition both rules are split in the same way by intersecting the condition with conditions in C, which by definition do not overlap. On the other hand, if the conditions of r_1 and r_2 do not overlap the splitting does not invalidate the criterion, because the intersection may only result in subsets of the original conditions.
Finally, rules that are appended to G_{is} in step three inherit their condition from a rule in G_{ir} and consequently Criterion 1 is preserved. □

Algorithm 1 split_terminals(G_{ir}, G_{is})

(1) $G_{temp} = G_{is}$
(2) remove all rules from G_{is}
(3) for each rule $r \in G_{temp}$
(4) let $C = split(cond(r))$
(5) // **Step 1 - Split rules in** G_{is}
(6) for each rule $r' \in G_{is}$: $term(r') = term(r)$
(7) remove r' from G_{is}
(8) for each $c \in C$
(9) add a copy r'' of r' to G_{is} with $cond(r'') = c \bigcap cond(r')$
(10) // **Step 2 - Split rules in** G_{ir}
(11) for each rule $r' \in G_{ir}$: $term(r') = term(r)$
(12) remove r' from G_{ir}
(13) for each $c \in C$
(14) add a copy r'' of r' to G_{ir} with $cond(r'') = c \bigcap cond(r')$
(15) // **Step 3 - Append rules to** G_{is}
(16) for each rule $r' \in G_{ir}$: $overlap(cond(r'), cond(r)) \wedge term(r') = term(r)$
(17) add a copy r'' of r to G_{is} with $cond(r'') = cond(r')$

The splitting also preserves the semantics of the original grammars, i.e., the calculated conditions do not allow patterns to match that did not match before. Similarly it is ensured that patterns that were accepted before the processing are still accepted afterwards.

Proof (Sketch). All conditions computed in the first two steps of the algorithm are derived using intersection against conditions in C. The intersection ensures that the new rules only match a subset of the original rule, while the properties of C ensure a complete coverage of the condition domain D. The semantics of the original grammar is thus preserved.
Step three of the algorithm splits a rule r based on conditions from G_{ir}. Because of step two of the algorithm it is ensured that all rules in G_{ir} that share the same terminal symbol with r are associated with conditions that match only a subset of r's condition. The rules in G_{ir} initially were unconditional and covered the complete condition domain. This property is preserved throughout the algorithm, i.e., the combined conditions of the split rules in G_{ir} still cover the complete domain. Consequently the semantics of the original grammar is preserved. □

The instruction selector in Figure 75 is capable of matching signed and unsigned 16-bit values. The conditions appearing in the two grammars clearly overlap and need to be split as shown in Figure 76. This results in three ranges $\{-32768, \ldots, -1\}$, $\{0, \ldots, 32767\}$, and $\{32768, \ldots, 65535\}$. The IR specification is processed in a similar way, but in addition retains all integer values not covered by the instruction selector.

$v \rightarrow INT\_CONST$	$\{-\infty, \ldots, -32769\}$
$v \rightarrow INT\_CONST$	$\{-32768, \ldots, -1\}$
$v \rightarrow INT\_CONST$	$\{0, \ldots, 32767\}$
$v \rightarrow INT\_CONST$	$\{32768, \ldots, 65535\}$
$v \rightarrow INT\_CONST$	$\{65536, \ldots, \infty\}$
$v \rightarrow +(v, v)$	$\{-\infty, \ldots, \infty\}$

(a) IR specification after splitting

$r \rightarrow INT\_CONST$	$\{-32768, \ldots, -1\}$
$r \rightarrow INT\_CONST$	$\{0, \ldots, 32767\}$
$r \rightarrow INT\_CONST$	$\{32768, \ldots, 65535\}$
$r \rightarrow +(r, r)$	$\{-\infty, \ldots, \infty\}$

(b) IS specification after splitting

Figure 76: Terminal splitting for a simple example supporting only 16-bit signed and unsigned constants.

5.6.4 Chain Rules

In the model presented in the last section we require a normalized rule grammar. Although this does not restrain the power of our approach we do allow some extensions for convenience. For example, chain rules, i.e., rules of the form $p \rightarrow q$, where p and q are both non-terminals, are permitted by our implementation. Even more, chain rules may be associated with conditions similar to regular rules.

Both conditional and unconditional chain rules are eliminated before terminal splitting by duplicating regular rules in the tree grammar that have the non-terminal q on the left hand side, i.e., produce the non-terminal q. The left hand side q of the duplicate is replaced by the left hand side of the chain rule p. This step is repeated to capture transitive chains of chain rules until the rule set does not change anymore. In the case of conditional chain rules the conditions of the chain rule and the regular rule need to be intersected in order to preserve the semantics. After this step chain rules are no longer required and can safely be discarded.

Consider the IS grammar from Figure 75 as an example. Given a conditional chain rule $r8 \rightarrow r$ that is associated with a condition $\{-128, \ldots, 127\}$ the following rules are added to the grammar:

$r8 \rightarrow INT\_CONST$	$\{-128, \ldots, 127\}$
$r8 \rightarrow INT\_CONST$	$\{0, \ldots, 127\}$
$r8 \rightarrow +(r, r)$	$\{\infty, \ldots, -\infty\}$

5.6.5 Final Completeness Test

After the application of chain rules and terminal splitting the final completeness test can be performed. The algorithm is based on the work by Emmelmann [48] with the notable difference that we also need to consider conditions that may be associated with the rules of the IR and instruction selector specifications.

For the actual completeness test two tree automata are derived from the two grammars. The automaton $\mathcal{A}_{ir} = (Q, \mathcal{F}_a, Q_f, \Delta)$ is derived from the tree grammar

with conditions $G_{ir} = (S, N, \mathcal{F}, R, D)$ as follows: The states of the automaton $Q = \{q_A \mid A \in N\}$ are deduced from the non-terminals of the grammar. Similarly, the final state $Q_f = \{q_S\}$ is defined using the start symbol S. The ranked alphabet of the automaton is derived from the terminals matched by rules in G_{ir} and their associated conditions: $\mathcal{F}_a = \{f_c \mid \exists r \in R\colon f = \mathit{term}(r) \wedge c = \mathit{cond}(r)\}$. For each rule r of the form $A \to f(A_1, \ldots, A_n) \in R$ with its associated condition $c = \mathit{cond}(r)$, a corresponding transition rule $f_c(q_{A1}, \ldots, q_{An}) \to q_A$ is added to the automaton, $q_A, q_{A1}, \ldots q_{An} \in Q, f_c \in \mathcal{F}_a$. Rules of the form $q \to a$ are processed in the same way. The constructed automaton reflects the semantics of the IR specification G_{ir}.

Proof (Sketch). The languages of the automaton and the grammar are obviously not equivalent. However, a direct mapping can be established between the two languages. A proof by induction over the derivation length yields the equivalence of the two languages under this mapping. □

The automaton $\mathcal{A}_{is}$ is similarly constructed using G_{is}. An important property of the two automata is that the alphabets are compatible, i.e., for every symbol in the alphabet of $\mathcal{A}_{is}$ a corresponding symbol exists in the alphabet of $\mathcal{A}_{ir}$. This follows immediately from Criterion 1. Note that the opposite is generally not true, in particular, when the instruction selector specification is not complete.

As in the original approach, a set of states called U, is computed from these two tree automata. A state is a tuple $[P; q]$, where $P \subset S_{is}$ is a set of states of $\mathcal{A}_{is}$, and $q \in S_{ir}$ is *one* state of $\mathcal{A}_{ir}$. U is the smallest set of states that fulfills the following property: For all symbols f_c with arity n and states $[P_1; q_1], \ldots, [P_n; q_n] \in U$, let

$$P = \{s \mid \exists s_1 \in P_1, \ldots, s_n \in P_n\colon f_c(s_1, \ldots, s_n) \xrightarrow{\mathcal{A}_{is}} s\}$$

$$Q = \{s \mid f_c(q_1, \ldots, q_n) \xrightarrow{\mathcal{A}_{ir}} s\}.$$

All states of the from $[P; q]$ need to be in U, where $q \in Q$. This immediately leads to an algorithm. Starting with an empty set, new states are inductively added to U until no new states can be found. Termination is guaranteed as the number of states in both automata is finite.

By examining U it is now easy to decide whether the original instruction selector specification completely covers the compiler's IR. The code selector is not complete if a state $[P; \mathcal{Z}_{ir}]$ can be found in U, where $\mathcal{Z}_{ir}$ is the final state of $\mathcal{A}_{ir}$, but $\mathcal{Z}_{is}$, the final state of $\mathcal{A}_{is}$, is not in P. It is even possible to give counter examples by storing an example tree for each state during the calculation of U. For details on both of these algorithms and correctness proofs refer to the original work by Emmelmann [48].

6 Experimental Results

Several processor models have been developed with the xADL language ranging from very simple RISC processors to sophisticated VLIW processors supporting predicated execution. The resulting processor models are typically very compact and readable, and can easily be extended. The development of a new processor model is usually only a matter of a few days. This indicates that the structural specification style of the xADL language is well suited for the description of processors and their instruction set. The xADL language and the accompanying backend generators for the LLVM and *acc* compilers have been evaluated using four processor models: (1) a two-way configuration of the CHILI VLIW processor, (2) a four-way CHILI configuration, (3) a MIPS processor model, and (4) a model of the time-predictable research processor SPEAR.

We have compared the backends derived from these models to high-quality production compilers. In the case of the *acc* compiler no handcrafted backends exist, we thus compare against an unmodified version of the same GCC compiler that also serves as a frontend to our backend. The comparison is still kept short, because quite different code generation technologies are compared. The main focus of the performance evaluation is thus on the LLVM-based compilers. Here, the generated compilers have been compared against handcrafted production compilers based on GCC. In addition, a manually developed, well-tuned research backend targeting the CHILI processor that is based on LLVM is considered for the evaluation. The experiments thus allow to relate the results of the generated compilers to well-tested production systems and cutting-edge compiler technology following different code generation strategies.

6.1 Processor Models

The MIPS processor model closely follows the traditional five-stage pipeline implementation described by Hennessy and Patterson [147]. The processor description faithfully models the complete *MIPS1* integer instruction set, including the mandatory branch and load delay slots of early MIPS implementations. In total 57 instructions are described by 1143 lines of xADL code (LOC). The instructions operate on 32 general purpose registers and additional special purpose registers for division and multiply instructions. A slightly simplified diagram of the model's component instances is shown in Figure 77. The block diagram also depicts a parallel instruction (`IDU`) that handles the dispatch of an interrupt service routine in the case of an asynchronous interrupt.

SPEAR [41] is a time-predictable research processor, specifically designed to support the single-path programming paradigm [153]. Most instructions can thus be augmented with a predicate that controls whether the instruction commits its result to the register file. The SPEAR model covers the complete instruction set of

		Syntax		Encoding		Types		Components	
Model	LOC	LOC	#Tmpl.	LOC	#Tmpl.	LOC	#Ty.	LOC	#Ists.
CHILI-v2	1580	191	12	141	6	800	20	350	14
CHILI-v4	1739	220	12	156	6	830	20	454	24
MIPS	1143	183	14	134	9	592	14	157	12
SPEAR	1298	109	5	223	12	733	13	172	14

Table 8: Statistics on the MIPS, SPEAR, and CHILI processor models.

the second generation implementation SPEAR2 [62]. Instructions with and without predicates are modeled as separate instructions, which explains the high number of 107 instructions in comparison to the MIPS model. SPEAR defines multiple register files: 16 general purpose registers, four frame pointer, as well as separate status and predicate registers. The frame pointer registers can be used as an addressing mode to certain load and store operations only – some of them with autoincrement addressing modes – but do not support any other operations. Transfers between the frame pointer registers and the general purpose register file are thus often necessary. This restriction can be attributed to constraints of the instruction encoding as SPEAR defines 16-bit instructions only. The instructions are executed by a four-stage pipeline, without branch or load delay slots.

The CHILI is a configurable VLIW processor developed by OnDemand Microelectronics. The main application field targeted by CHILI is multimedia processing in embedded devices, primarily real-time de- and encoding of video streams. The four-way parallel default configuration thus offers high computing power that is further supported by specific instruction set extensions providing hardware support of typical operations performed by video processing software. Due to the computational power, the memory interface becomes a limiting factor. The CHILI instruction set defines large load and branch delays to hide memory latencies. Branch latencies can further be eliminated using predicated execution. Predicated instructions, however, occupy two slots of the VLIW bundle, because the predicate is computed by a dedicated operation in parallel with the predicated instruction. This approach is very flexible and efficient, but can adversely affect code size when large code blocks are augmented with predicates. Similarly to SPEAR, the predicated variants are explicitly enumerated by the instruction extraction algorithm, which results in 886 and 1672 instructions for the two-way and four-way parallel configurations respectively. Due to the long latencies and the encoding constraints of predicated instructions, the instruction scheduler is critical for acceptable performance of compiler generated code.

Table 8 shows detailed statistics on the xADL specifications of the three processors. The specifications are very compact and consists of 1739 lines of xADL code at most. The table further shows the number of lines spent on the syntax and the binary encoding specifications, as well as on types and component instantiations, along with numbers on the respective templates, types and instantiations defined.

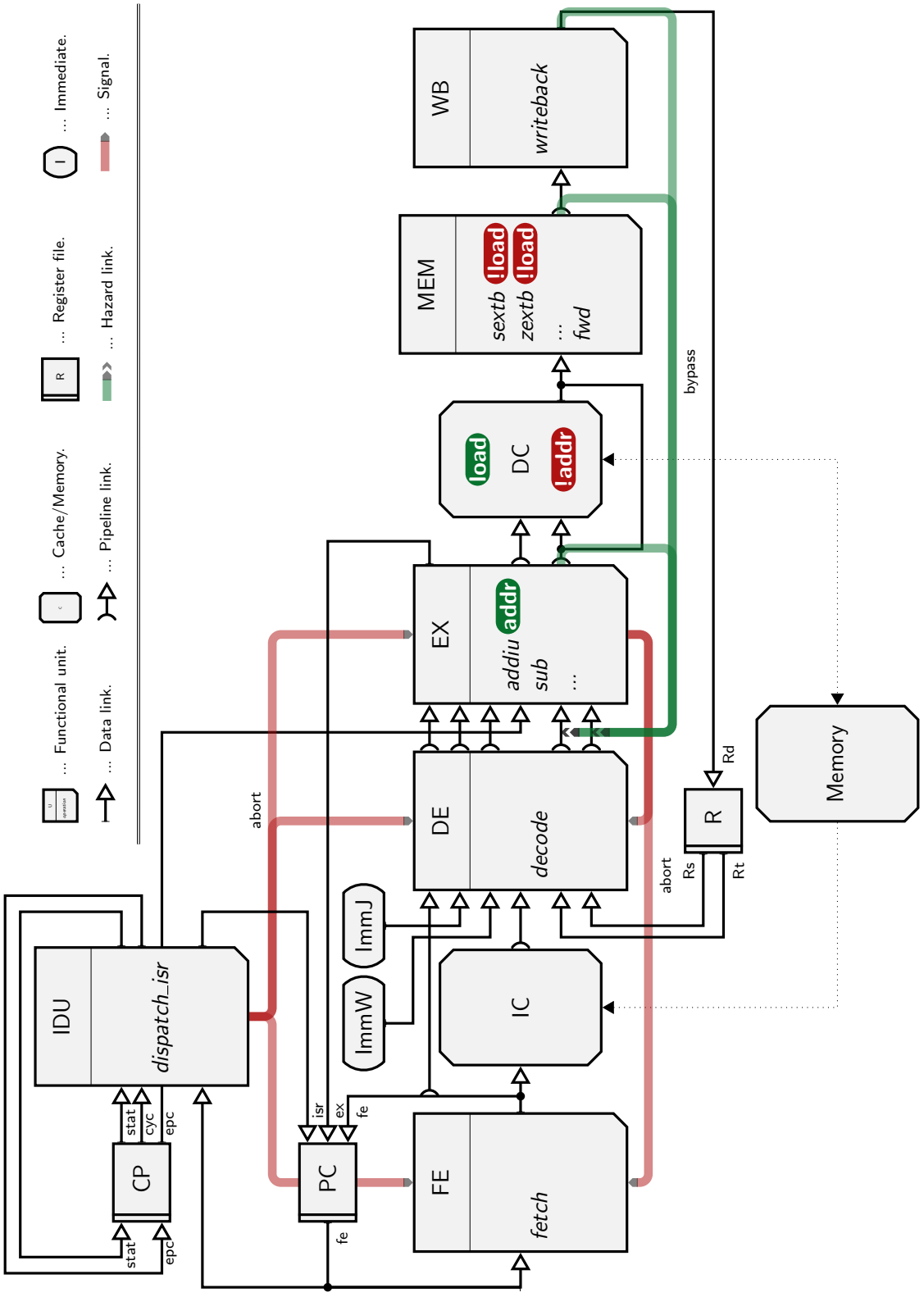

Figure 77: Block diagram of the MIPS processor model.

	Definitions		Expanded		Instruction Set	
Model	#Uts.	#Ops.	#Uts.	#Ops.	#Paths	#Insts.
CHILI-v2	19	77	31	129	15	886
CHILI-v4	19	77	60	253	27	1672
MIPS	7	61	7	61	3	57
SPEAR	7	62	7	62	3	104

Table 9: Statistics on the MIPS, SPEAR, and CHILI instruction set models.

The models further specify programming conventions and processor configurations, which occupy between 69 and 91 lines. Note that the numbers in the table do not add up, because of these additional specifications. The reusable and extensible type definitions account for more than 50% of the code lines, the fixed instantiations on the other hand account for just 26% for the four-way CHILI and less than 13% for the MIPS model. This indicates that types help factoring out common hardware fragments, types of well structured descriptions can easily be reused. This is also reflected by the two CHILI configurations, where all types are reused, except for register, memory, and cache types that require additional ports for the 4-way parallel configuration. But even in these cases inheritance can be applied, leading to readable and compact specifications.

The instruction set extraction algorithm mainly relies on the functional units and the operations associated with them. Table 9 relates the number of unit instantiations and operations statically defined in the xADL specification to the number of unit instances and operations present in the expanded hypergraph representation, i.e., unit instantiations are possibly expanded to multiple instances. In particular, the regular structure of the parallel CHILI data path can be specified very compactly using the xADL language. The instruction set is extracted from this graph representation by first discovering the instruction paths, and subsequently computing the individual instructions from the operations along the various paths. This approach is very powerful in enumerating instruction variants. From the four-way parallel CHILI configuration, for example, 1672 instructions are created, which corresponds to about one line of xADL code per instruction. Even for the MIPS model, only about 20 lines of code are spent per instruction. The results also show that the number of instruction paths is considerably smaller. This helps during the development of xADL processor models, because the designer can focus on the overall structure, i.e., the instruction paths, first and later add operations as needed to realize instructions.

It is hard to compare the xADL approach with other processor description languages. For one, the processor models available with various other systems are rarely published. On the other hand, the different systems often target distinct application scenarios, and thus certain information available in one language may be omitted in another. To give an indication how the xADL language compares to other processor description language the MIPS model was compared against other publicly available MIPS-based specifications, namely a cycle-true MIPS-R3000 ArchC

		ISA		Behavior	Structure	Compiler	
Model	LOC	LOC	#Instrs	LOC	LOC	LOC	#Rules
R3000	2533	386	58	2121	–	–	–
acesMIPS	4184	828	85	–	533	2353	173

Table 10: Statistics on the ArchC MIPS R3000 and acesMIPS EXPRESSION descriptions.

model [12] (model version 0.7.2) and a MIPS-based VLIW *acesMIPS* specified using the EXPRESSION [88] language (model version 0.99). In particular, the ArchC R3000 model matches the xADL MIPS model very closely. The 57 MIPS integer instructions and the *syscall* operation are specified using 2533 lines of code, 2121 of which are plain SystemC code modeling the instruction behavior. Another 386 lines specify the instruction set architecture including the syntax and binary encoding. The instruction behavior is hardly analyzable, due to the use of SystemC. ArchC models are thus not suited for the automatic generation of compiler backends.

The *acesMIPS* is a VLIW processor based on the MIPS instruction set. It defines 85 integer and floating point instructions using 4184 lines of EXPRESSION code. More than 50% is spent on tree pattern specifications that are used to retarget the tree pattern matching instruction selector of the EXPRESS compiler. In total, 173 matching rules are defined using 2353 lines of code. The instruction set specification requires 828 lines of code, but does not cover the instruction's binary encoding. Additional 533 code lines model the hardware resources and memory hierarchy. The comparison shows that the structural approach taken by the xADL language leads to comparatively smaller specifications and offers a high potential for reuse. Table 10 summarizes the statistics on the MIPS-R3000 ArchC and the acesMIPS EXPRESSION descriptions. Note that the line numbers again do not add up because comments and empty lines are skipped for the break down.

6.2 Backend Generation for *acc*

The MIPS model was used to evaluate the backend generator for the *acc* backend. We executed a subset of the MiBench [80] and MediaBench [116] benchmark suites using a cycle-accurate simulator, omitting those using floating point operations. *cmac*, *dct32*, *dct8x8*, *serpent*, and *twofish* are additional benchmarks supplied by our research partner. All benchmarks are medium sized, ranging from 800 to 4400 lines of code. The rule set of the generated compiler consists of 163 rules, covering integer operators only.

We compared the generated backend with the original, highly-optimizing MIPS backend of GCC 4.1.1, which also serves as the frontend for the *acc* compiler. While the generated backend benefits from a number of high-level loop transformations and optimizations applied by the frontend, most of the traditional backend optimizations

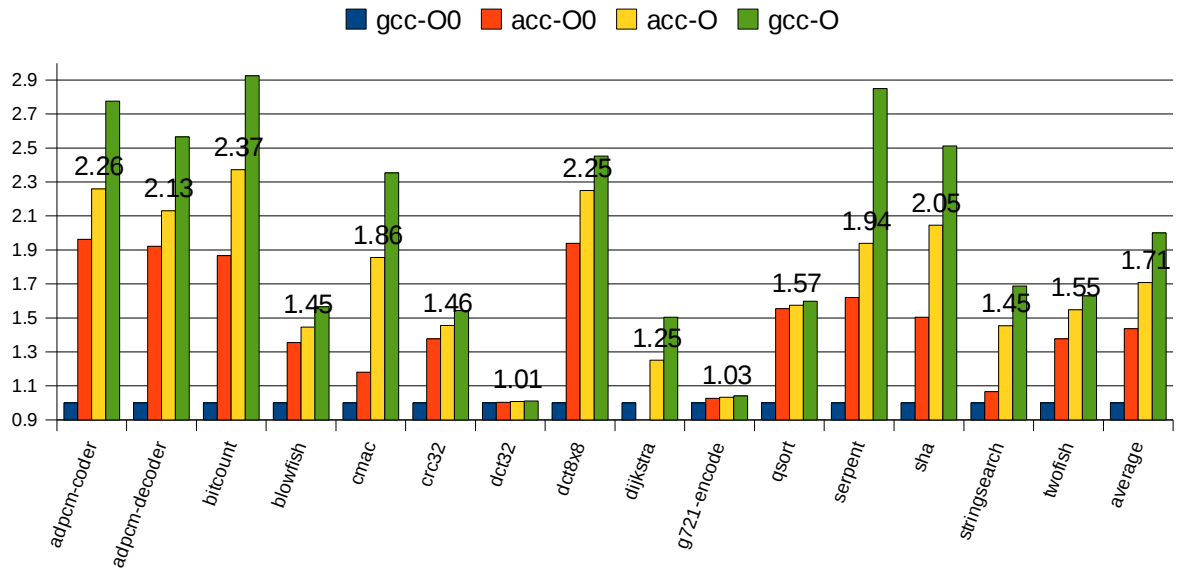

Figure 78: Performance improvements of the generated MIPS backend in comparison to the GCC compiler without optimizations.

are implemented on RTL-level, i.e., the machine-level representation of GCC, and are thus not available to the *acc* compiler.

Figure 78 shows the speedup of the original GCC compiler and the automatically generated backend in comparison to GCC without optimizations (`gcc -O0`). The benchmarks were compiled using two different optimization levels, once without optimizations (`-O0`) and once with optimizations enabled (`-O`). The average improvement of the *acc* backend without optimizations in comparison to the baseline compiler is

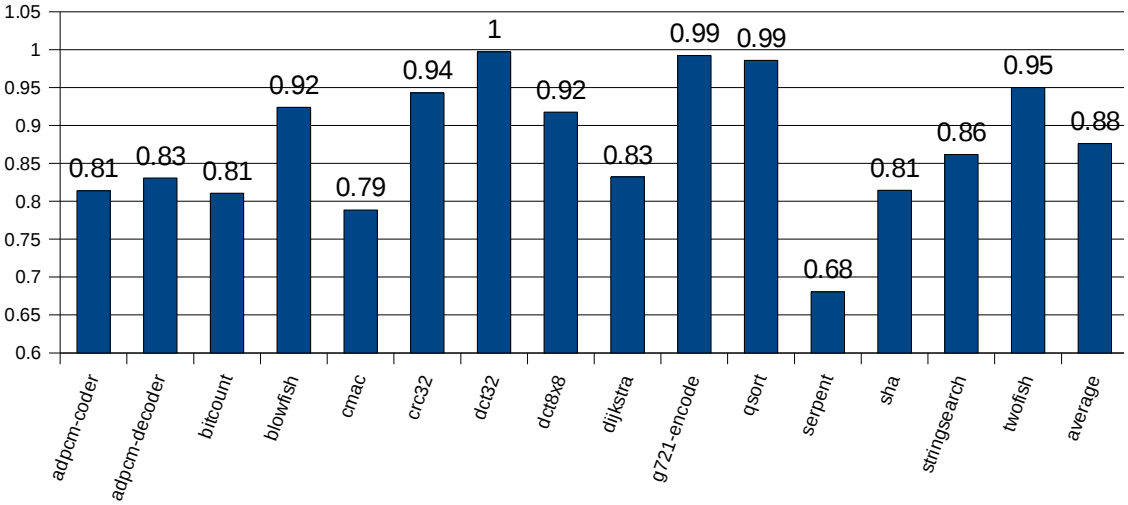

Figure 79: Performance improvements of the generated MIPS backend in comparison to the GCC compiler with optimizations enabled.

Benchmark	LOC	Benchmark	LOC
automotive-bitcount	932	security-sha	269
consumer-jpeg	26,098	telecomm-crc32	284
network-dijkstra	187	telecomm-fft	476
office-stringsearch	3,250	telecomm-adpcm	304
security-blowfish	1,913		

Table 11: Size of the benchmark programs in source lines.

about 44%. Apart from the code generator, register allocator and instruction scheduler, currently no other optimizations are available in the *acc* backend. It is thus lacking redundancy elimination and loop invariant code motion of address calculations, which leads to a considerable performance penalty in comparison the the optimized code generated by GCC. Furthermore, the high-level optimizations by the GCC frontend are not as effective as initially expected. Enabling additional optimizations often does not yield any changes to the input of the *acc* backend, while GCC's optimizations at the RTL-level significantly improve the code quality. On average the high-level optimizations lead to a runtime improvement of about 71% in comparison to the baseline compiler. The `cmac` and `stringsearch` benchmarks benefit the most from the additional high-level optimizations. With optimizations enabled, the code produced by the generated compiler reaches about 88% of GCC's performance on average, as can be seen in Figure 79.

6.3 Backend Generation for LLVM

The backend generator for the LLVM framework was evaluated using the CHILI and MIPS processor models. Due to incompatibilities of the assembler and linker, the SPEAR compiler is currently limited to trivial programs that only refer to labels with local linkage. Runtime and code size statistics are thus omitted for SPEAR. A subset of the MiBench benchmark suite, as provided by the LLVM test infrastructure, was compiled and subsequently executed by cycle-accurate simulators generated from the respective processor descriptions [24, 25]. The accuracy of the simulators has been verified against cycle-true reference simulators, the generated simulators for the CHILI processors, for example, perfectly match the reference simulator provided by OnDemand Microelectronics and only show a difference of exactly one cycle, due to a slightly different shutdown procedure at the end of a simulation run. Table 11 lists the number of source lines including comments for each benchmark program. The benchmarks were run using the *small* input data sets in order to reduce the simulation time.

The benchmarks for MIPS were compiled using GCC version 4.1.1 and the GNU Binutils version 2.19.1 configured for the `mips-elf` target. The *newlib* system li-

Model	#Instr. Def.	#Reg. Def.	#Reg. Cl.	#Res.	#Rules
CHILI-v2	817	64	1	2	1416
CHILI-v4	817	64	1	4	1416
MIPS	61	35	3	1	111
SPEAR	106	22	4	2	42

Table 12: Statistics on the generated LLVM backends for the MIPS, SPEAR, and CHILI processor models.

brary[17] provides a basic C library implementation. GCC version 4.2.0, together with the GNU Binutils 2.16, provided by OnDemand Microelectronics serves as a reference compiler for the CHILI architecture. The system libraries are based on newlib version 1.14.0. The automatically generated compilers rely on LLVM version 2.4, which uses a modified version of GCC 4.2 as frontend. The GNU Binutils and newlib libraries are shared among the respective reference compilers and the generated compilers. The reference compilers are invoked with aggressive optimizations (`-O3`), while the LLVM compilers use the standard optimization options (`-std-compile-opts`).

As can be seen in Table 12, the LLVM backends derived from the two-way and four-way configurations of the CHILI processor are virtually identical, except for the resources present in the reservation tables of the instruction scheduler. The table lists, from left to right, the number of instruction definitions, register definitions, register classes, abstract resources of the resource tables, and the number of instruction selection patterns generated from the respective processor descriptions. The number of instruction definitions differs from the instruction number listed by Table 9, because equivalent instruction variants are merged. Hence, the number of instruction definitions is lower for the parallel CHILI models. On the other hand,

[17] `http://sourceware.org/newlib/`

	Code Size			Cycles		
Benchmark	GCC	xADL	%	GCC	xADL	%
automotive-bitcount	31,468	25,364	-19	726,162	991,642	+36
consumer-jpeg	245,148	161,648	-34	7,932,872	9221,371	+16
network-dijkstra	39,116	38,564	-1	342,801,926	314414,695	-8
office-stringsearch	27,672	25,700	-7	5,367,471	7274,936	+36
security-blowfish	36,768	26,544	-28	867,965	877,218	+1
security-sha	31,796	29,352	-8	13,270,780	17812,223	+34
telecomm-crc32	29,816	27,716	-7	7,464,707	8350,673	+12
telecomm-fft	45,172	44,976	–	140,766,729	137254,431	-2
telecomm-adpcm	27,588	27,372	-1	7,122,022	12125,699	+70

Table 13: Code size and execution time results for the MIPS processor.

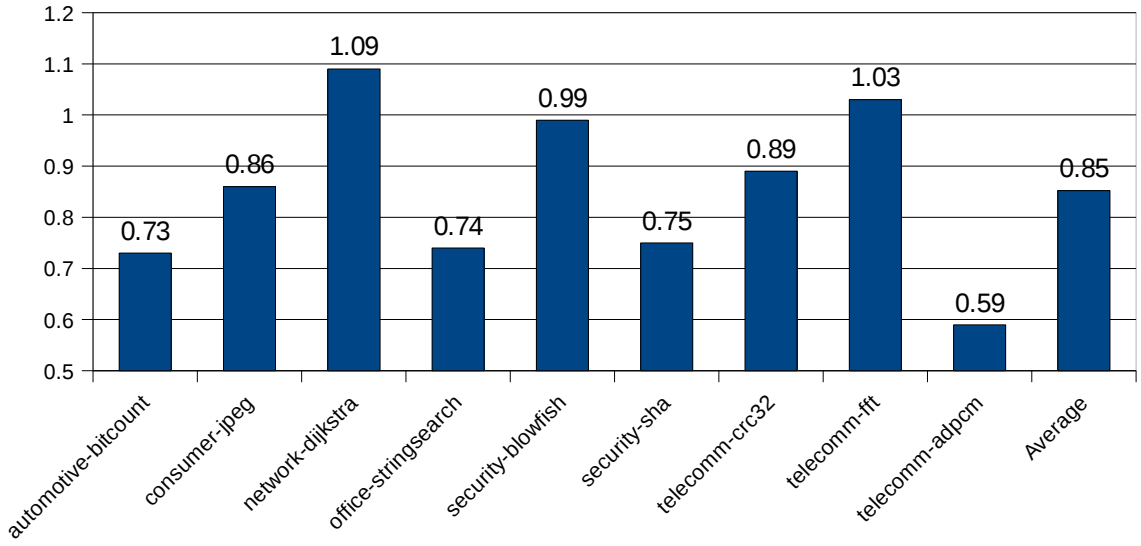

Figure 80: Performance difference of the generated backend in comparison to GCC.

instructions with multiple result values are duplicated, due to restrictions of the LLVM instruction selector. This leads to a higher number of instruction definitions for the MIPS and SPEAR models.

The backend generator quite successfully discovers translation patterns from the instruction sets. For every instruction definition of the MIPS and CHILI models almost two instruction selection patterns are generated, leading to the huge number of 1416 rules for the CHILI models. One might think that this number is caused by *duplicates* due to commutativity. However, this is not the case, because the *tablegen* tool already handles commutativity, the respective patterns are thus suppressed during the emission by the compiler generator. This can also be seen by comparing the number of matching rules generated for the *acc* backend to the number of instruction selection patterns in Table 12. From the 163 matching rules only 111 are considered for the LLVM backend. The majority of the omitted patterns are *duplicates* derived through the application of the commutativity law.

The measured execution times and the code size of the stripped benchmark programs for the MIPS instruction set are shown in Table 13. The results indicate that the automatically generated MIPS compiler is competitive to the well-tuned production compiler GCC, in particular, when code size is taken into account. The *fft* and *dijkstra* benchmarks show a reduction of the execution time by 2% and 8% respectively. The severe increase in execution time of 70% in the case of the *adpcm* benchmark is caused by useless branches generated late during the compilation process from conditional assignments. The branch optimization of the LLVM framework runs earlier and thus misses these cases. The relative performance is depicted by Figure 80. On average, a slowdown of only 15% has been observed over all benchmarks.

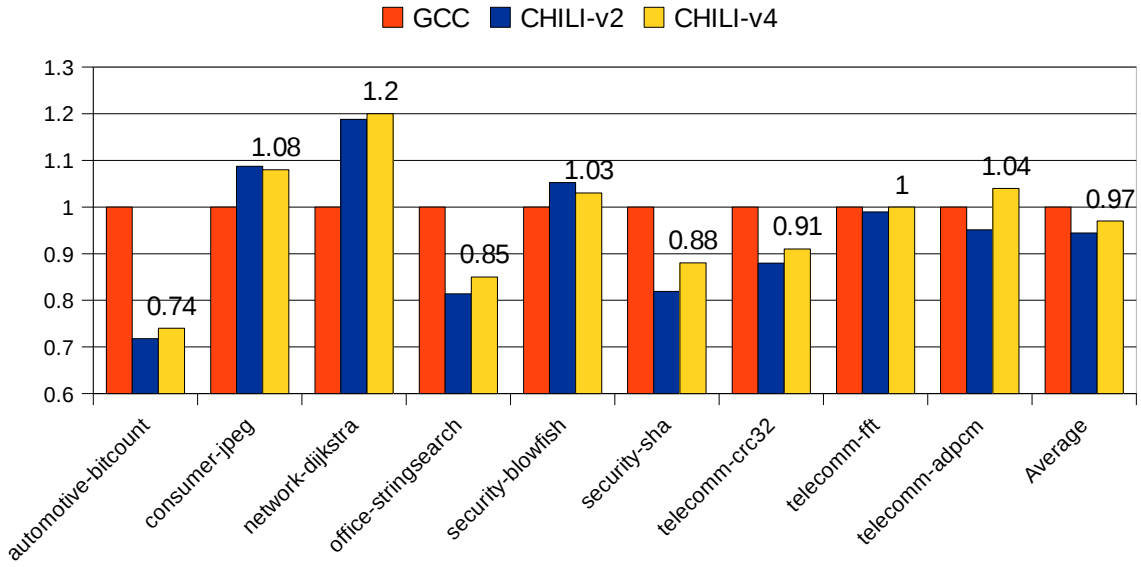

Figure 81: Performance improvement of the generated CHILI backends in comparison to GCC.

The performance results obtained for the two CHILI processor configurations are very close to the handcrafted production compilers – see Figure 81. Several benchmarks show considerable speedups, in particular the *dijkstra* and the the *jpeg* benchmarks show an improvement of up to 20%. In contrast to the MIPS model, the generated compilers for the CHILI models do not show such severe slowdowns. The *adpcm* benchmark performs much better for the CHILI, because the conditional assignments are directly supported by the processor, useless branches are thus avoided. The four-way parallel configuration even outperforms the production compiler by 4%. The code produced by the generated compilers for the *bitcount*

	Code Size			Cycles		
Benchmark	GCC	xADL	%	GCC	xADL	%
automotive-bitcount	349,660	296,504	-15	909,105	1,266,562	+39
consumer-jpeg	2,374,672	1,248,448	-47	11,394,100	10,477,805	-8
network-dijkstra	485,944	470,872	-3	2,885,225	2,429,125	-16
office-stringsearch	334,516	303,384	-9	624,684	767,442	+23
security-blowfish	405,408	306,192	-24	1,554,181	1,476,850	-5
security-sha	354,676	327,864	-8	12,457,571	15,212,856	+22
telecomm-crc32	354,108	327,132	-8	8,637,804	9,816,493	+14
telecomm-fft	415,824	401,140	-4	188,668,150	190,661,456	+1
telecomm-adpcm	339,116	324,856	-4	10,404,612	10,936,309	+5

Table 14: Code size and execution time results for the two-way parallel CHILI configuration.

	Code Size			Cycles		
Benchmark	GCC	xADL	%	GCC	xADL	%
automotive-bitcount	348,892	296,376	-15	881,144	1,183,104	+34
consumer-jpeg	2,341,904	1,241,408	-47	10,794,047	9,976,958	-8
network-dijkstra	485,560	470,744	-3	2,894,236	2,414,124	-17
office-stringsearch	334,004	303,384	-9	624,087	738,406	+18
security-blowfish	400,160	306,192	-23	1,541,883	1,491,519	-3
security-sha	351,860	327,608	-7	10,791,045	12,215,822	+13
telecomm-crc32	353,980	327,132	-8	8,637,327	9,520,911	+10
telecomm-fft	415,184	400,884	-3	187,968,275	188,243,462	+1
telecomm-adpcm	338,988	324,728	-4	10,116,131	9,755,433	-4

Table 15: Code size and execution time results for the four-way parallel CHILI configuration.

benchmark, however, performs poorly. Address calculations for memory operations accessing an array are not optimally translated and cause some extra instructions. Due to the small loops of the bit counting algorithms this has a large impact. Nevertheless, slight slowdowns of only 5% and 3% respectively have been observed over all benchmarks for both CHILI configurations. Considering the code size these results are motivating for future work. As for the MIPS, the code size is again considerably smaller. On average the code size of the stripped executables produced by the xADL-based backends is reduced by 15%. The size of the *jpeg* benchmark program, for example, is reduced by 47%, for *blowfish* benchmark the reduction amounts to about 25%.

We have further compared the performance results against a handcrafted research backend for the CHILI processor that was developed in cooperation with out research partner OnDemand Microelectronics. This backend was enhanced with additional optimization passes to hide the long branch and load latencies. A very effective if-conversion pass utilizes the strong support for predicated execution to eliminate branches. Branches are further aggressively rewritten by a dedicated branch optimization. The register allocator was enhanced to avoid conflicting assignments for independent calculations in order to increase the available parallelism. The scheduler was similarly enhanced. The weights of the data dependence graph are computed by well-tuned heuristics that improve the resource utilization. The priority assignment during the selection of candidates from the ready list is optimized to avoid unfavorable schedules and reduce resource contention. To obtain comparable results the if-conversion pass was disabled for this evaluation, because the generated compilers are lacking a similar optimization.

In general the generated compilers are again competitive to the handcrafted research backend. For the four-way CHILI configuration even a speedup of 5% is achieved for the *stringsearch* benchmark, also the execution time of the *crc32* benchmark is improved by 3%. The other results are more balanced than for the

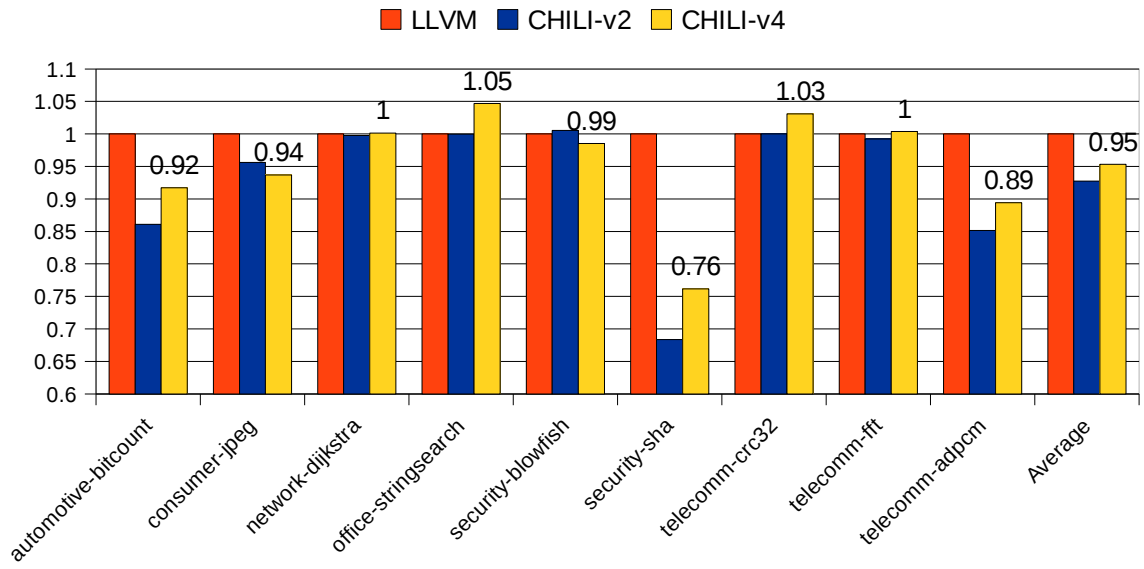

Figure 82: Performance improvement of the generated CHILI backends in comparison to LLVM.

previous comparisons. Only one benchmark shows significant slowdowns of up to 32%. The results of the *sha* benchmark can be attributed to the superior priority assignment of the handcrafted compiler during scheduling. This leads to an improved utilization of the available computational resources. Transformations during the construction of the data dependence graph also improve the utilization of branch delay slots. The results for the *bitcount* and *adpcm* benchmarks are similarly caused by unfavorable scheduling decisions. On average over all benchmarks a slowdown of only 7% has been measured for the two-way parallel CHILI configuration. For the four-way parallel configuration the slowdown is even smaller and amounts to only

	Code Size		Cycles	
Benchmark	LLVM	%	LLVM	%
automotive-bitcount	295,604	0	1,090,557	16
consumer-jpeg	1,240,056	1	10,017,733	5
network-dijkstra	470,724	0	2,423,692	0
office-stringsearch	302,612	0	767,195	0
security-blowfish	305,092	0	1,484,544	-1
security-sha	327,600	0	10,398,265	46
telecomm-crc32	327,132	0	9,815,961	0
telecomm-fft	400,820	0	189,185,087	1
telecomm-adpcm	324,600	0	9,313,023	17

Table 16: Code size and execution time results for the handcrafted LLVM compiler targeting the two-way parallel CHILI configuration.

	Code Size		Cycles	
Benchmark	LLVM	%	LLVM	%
automotive-bitcount	295,476	0	1,085,154	9
consumer-jpeg	1,227,896	1	9,344,536	7
network-dijkstra	470,596	0	2,417,219	0
office-stringsearch	302,484	0	772,938	-4
security-blowfish	305,092	0	1,469,591	1
security-sha	327,344	0	9,301,773	31
telecomm-crc32	327,132	0	9,815,831	-3
telecomm-fft	400,692	0	188,944,990	0
telecomm-adpcm	324,472	0	8,722,612	12

Table 17: Code size and execution time results for the handcrafted LLVM compiler targeting the four-way parallel CHILI configuration.

5%. In contrast to the previous comparisons to the production compiler based on GCC, the size of the produced executables is almost identical for the generated and handcrafted compilers based on LLVM.

Note, that a MIPS backend was added to LLVM during a Google Summer of Code project. However, a comparison to the generated backend was not considered for this evaluation. LLVM's MIPS backend is in an early development phase and does not yet achieve competitive results, it even fails to generate correct code for about half of the benchmarks at this point.

6.4 Completeness of Instruction Selector Specifications

The backend generator for the *acc* compiler is able to verify, whether the derived instruction selector specification is complete using the completeness test based on terminal splitting. The intermediate representation of the *acc* backend is very similar to the *TREE/GENERIC* language that is also used by GCC. Before the instruction selection phase, the IR is rewritten to eliminate high-level constructs present in the original language, e.g., array and struct reference expressions are rewritten into regular arithmetic and load or store operations. In total, this low-level IR consists of about 40 operators, some of which are target-independent and are handled using default rules. Ignoring the target-independent operations, the initial tree grammar specifying the compiler's IR consists of only 32 rules and two chain rules.

The generated instruction selector heavily relies on dynamic checks, all of these checks, except equality constraints, can be modeled using the proposed terminal splitting approach. As mentioned in the previous chapter, testing whether two nodes in the IR are the same cannot be modeled. Rules with equality constraints are thus treated conservatively, i.e., are removed from the rule set and completely ignored throughout the completeness test. In practice this is not an issue, it is very

Model	IS-Rules	Non-terminals	Chain	Cond Chain
SPEAR	303	67	18	16
MIPS	521	90	13	12
CHILI	4159	1882	19	16

Table 18: Properties of the normalized tree grammars before terminal splitting.

unlikely that a combination of rules with equality constraints will contribute to the completeness of the instruction selector. None of the rules in our experiments carried an equality constraint. The dynamic checks employed by the backend generator within the *adlgen* tool test the following properties:

- the value of constants or the value range of IR nodes,
- whether the value of an IR node is guaranteed to be a power of two,
- the signedness of the IR node's type,
- the size of the node's type,
- the register class of virtual registers.

In our experiment we only consider integer arithmetic, thus the value range of constants is modeled over the domain of 32-bit integer values. Consequently, the size of types is limited to 32 bits. In addition we can exploit the fact that the compiler frontend does not generate arbitrary types, but merely 32-bit pointer types and integer types of the sizes 1, 8, 16, and 32.[18] Testing for the signedness of the IR node's type and powers of two is modeled using powersets over the boolean values *true* and *false*. The domain for the register class test depends on the current processor model and is computed automatically by the backend generator.

The compiler generator represents the instruction selection rules in a compiler-independent format that is not yet normalized. Thus, in a preprocessing step, the instruction selector rules need to be normalized by duplicating fragments of the original patterns and introducing new non-terminal symbols. As depicted in Table 18, this leads to a large number of non-terminals and rules compared to the initial rule set. This is especially true for the CHILI processor models. Note, that the non-terminals and rules of identical sub-patterns are reused as much as possible, nevertheless the large number of predicated instruction variants of the CHILI leads to a large growth in the number of rules.

In addition we observe that the vast majority of chain rules actually is associated with a condition. This can be attributed to the design of GCC's intermediate language, where in some cases the semantics of an operation depends on the type

[18] Wider machines can be handled, but for the current experimental setup this is not needed.

Model	IS-Rules	IR-Rules	States	Time CT	Time Total
SPEAR	2332	172	170	0.4s	0.6s
MIPS	3304	274	267	0.8s	1.4s
CHILI	9195	273	255	29.9s	36.5s

Table 19: Number of rules in the tree grammars of the instruction selector and the compiler's IR after the application of chain rules and terminal splitting.

of the operands to that operation. For example, signed and unsigned comparisons are represented using the same operation in the IR. The only way to determine the actual type of the comparison is to check the type of the operands.

Table 19 shows the input to the final completeness test after terminal splitting and the application of chain rules. As expected, the high number of conditional chain rules combined with terminal splitting again leads to a large growth in the number of rules. The number of non-terminals on the other hand does not increase during this phase.

Similar to results in previous work the number of states calculated during the actual completeness test is relatively small compared to the number of rules. This is not surprising as the instruction selection rules are ambiguous and thus often lead to the same state. The total runtime on a 3 GHz Intel Xeon running Linux 2.6.18 is limited to a few of seconds for normal-sized instruction selector specifications. Even testing the completeness of the large CHILI specification requires only 36.5 seconds. About 30 seconds thereof are spent on the completeness test itself, whereas the remaining time is mostly spent on the application of chain rules. Note that the current implementation is not tuned for speed.

7 Conclusion

The development of embedded systems faces rigid constraints regarding technical as well as non-technical aspects. The heat dissipation, power consumption, and area requirements need to be minimized, while at the same time the performance and programmability of these systems is steadily rising. Production costs and the duration of product development cycles need to be minimized in order for the final products to be competitive. Application-specific instruction set processors have successfully been used in the past to achieve these opposing goals. The development of such processors is a challenging task that requires intimate knowledge of the problem domain and appropriate support tools to evaluate different design alternatives quickly.

In this work the novel xADL processor description language was presented that allows the specification of application-specific processors for embedded systems. The language is focused on the structural organization of the processor's register files, memories, and computational resources. However, the instruction set of the processor, even though not explicitly specified, is a central design concept of the language and its support tools. Instructions are automatically extracted from the structural processor model along instruction paths. The information of the instruction set view combined with the detailed structural processor model enables a very flexible use of the language. For example, software development tools for the given processor can automatically be customized, including the assembler, linker, compiler, and instruction set simulator. Even a prototype system that allows to derive VHDL hardware models has been shown to be feasible. Compared to other processor description languages, the presented approach facilitates the reuse of hardware components using extensible types similar to classes and templates known from the programming language C++. This leads to compact, but still readable and intuitive, specifications of the processor.

In addition to the xADL language itself, the generator tool *adlgen* was described. In particular, the compiler backend generator for the *acc* backend and the LLVM compiler infrastructure. The compiler generator is capable to derive specifications for the three central components required during the code generation phase of a compiler: (1) the register allocator, (2) the instruction scheduler, and (3) the instruction selector. The quality of the generated compiler components was evaluated using four processor models and compared against handcrafted high-quality production compilers based on the open-source compiler GCC. The results show that the generated compilers are competitive in terms of code quality. The generated compiler targeting the MIPS instruction set architecture on average shows moderate slowdowns of 15% for a set of benchmark programs taken from the MiBench and MediaBench benchmark suites. The generated compilers for two configurations of the VLIW processor CHILI are even closer to the handcrafted production systems. On average slight slowdowns of 5% and 3% have been measured. For some benchmarks the xADL-based compilers even outperform the well-tuned production systems by up to 20%.

A major problem of the compiler generator is the fact that it relies on the capabilities provided by the described processor. If the capabilities are too limited the generator might fail to derive a compiler that is able to translate all valid input programs accepted by the compiler frontend. The completeness of instruction selector specifications was thus studied in this work. Traditional completeness tests based on tree automata are too limited to be applicable in the context of modern compiler systems. Dynamic checks that occur quite frequently can not be represented by these approaches and may thus lead to overly conservative results. Terminal splitting was proposed to explicitly represent the dynamic checks of the initial instruction selector specification by dedicated terminal symbols. A traditional completeness test is then applied to the transformed instruction selector specification to verify its completeness. The presented approach was integrated with *adlgen's* compiler generator and evaluated for three example processors. If the test fails to prove the desired property, counter examples are computed. These counter examples provide valuable feedback to the processor designer. Even though terminal splitting lead to an increased problem size for the actual completeness test, the time required for testing an instruction selector specification is in the order of only a few seconds in practice.

Due to the motivating results, we believe that the xADL language and its accompanying tools have a high potential for future research and innovation. The language itself is currently extended by a powerful type system and improved modeling capabilities of memory hierarchies. We plan to extend the compiler generator to process instructions with multiple outputs more efficiently, in particular autoincrement addressing modes that are very important in the digital signal processing domain. Also the high-speed simulation framework that has been developed in conjunction with the xADL language offers potential for future innovation. Current work focuses on optimization methods for the efficient simulation of rare events, such as external interrupts, cache misses, or branch miss-predictions, using a rollback mechanism. The overhead of decoding the instructions for execution during the simulation is another area for improvements. We plan to investigate sophisticated software decoding and caching techniques to minimize this overhead. Various other generation tools are planned, including tools to automatically synthesize efficient binary encoding schemes for the instructions of a processor, improvements to the VHDL-based hardware generator, and techniques to automatically derive instruction set extensions for a processor model and a given input program.

References

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[2] Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM (JACM)*, 23(3):488–501, 1976.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[4] Hiroki Akaboshi. *A study on design support for computer architecture design*. PhD thesis, Department of Information Systems, Kyushu University, Japan, 1996.

[5] Hiroki Akaboshi, Hiroyuki Tomiyama, and Hiroto Yasuura. Compiler generation from hardware description language. In *APCHDL '93: Proceedings of the 1st Asia Pacific Conference on Hardware Description Languages*, pages 76–78, December 1993.

[6] Hiroki Akaboshi and Hiroto Yasuura. COACH: A computer aided design tool for computer architects. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 76(10):1760–1769, 1993.

[7] Hiroki Akaboshi and Hiroto Yasuura. Behavior extraction of MPU from HDL description. In *APCHDL '94: Proceedings of the 2nd Asia Pacific Conference on Hardware Description Languages*, pages 67–74, October 1994.

[8] Andrew Appel, Jack Davidson, and Norman Ramsey. The Zephyr compiler infrastructure. `http://www.cs.virginia.edu/zephyr/papers/overview98.ps`, November 1998.

[9] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 243–253, New York, NY, USA, 2001. ACM.

[10] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, 6(2):45–48, 2007.

[11] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[12] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, 2005.

[13] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310, New York, NY, USA, 1995. ACM.

[14] Vasanth Bala and Norman Rubin. Efficient instruction scheduling using finite state automata. In *MICRO 28: Proceedings of the 28th annual International Symposium on Microarchitecture*, pages 46–56, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[15] Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz C. V. Santos, Max Schultz, and Olinto Furtado. Automatic retargeting of binary utilities for embedded code generation. In *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 253–258, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Alexandro Baldassin, Paulo Cesar Centoducatte, and Sandro Rigo. Extending the ArchC language for automatic generation of assemblers. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 60–68, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo, and Guido Araujo. Optimizations for compiled simulation using instruction type information. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 74–81, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Jeff Bastian and Soner Önder. Specification of Intel IA-32 using an architecture description language. In *WADL '04: Workshop on Architecture Description Languages*, pages 151–166, Boston, USA, 2004. Springer Verlag.

[19] Ulrich Bieker, Martin Kaibel, Peter Marwedel, and Walter Geisselhardt. STAR-DUST: Hierarchical test of embedded processors by self-test programs. Technical Report 700, University of Dortmund, Deptartement of CS XII, 1998.

[20] Ulrich Bieker and Peter Marwedel. Retargetable self-test program generation using constraint logic programming. In *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 605–611, New York, NY, USA, 1995. ACM.

[21] John Adrian Bondy and U. S. R. Murty. *Graduate texts in mathematics - Graph theory*, volume 244. Springer, 2007.

[22] Aimen Bouchhima, Patrice Gerin, and Frédéric Pétrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 546–551, Piscataway, NJ, USA, 2009. IEEE Press.

[23] Florian Brandner. Completeness of instruction selector specifications with dynamic checks. In *COCV '09: 8th International Workshop on Compiler Optimization Meets Compiler Verification*, 2009.

[24] Florian Brandner. Fast and accurate simulation using the LLVM compiler framework. In *RAPIDO '09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2009.

[25] Florian Brandner. Precise simulation of interrupts using a rollback mechanism. In *SCOPES '09: Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems*, pages 71–80, 2009.

[26] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 13–22. ACM, 2007.

[27] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, editors. *Extensible Markup Language (XML) 1.0 - W3C Recommendation*. World Wide Web Consortium (W3C), 5th edition, November 2008.

[28] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 275–284, New York, NY, USA, 1989. ACM.

[29] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):3–13, 1992.

[30] John Bruno and Ravi Sethi. Code generation for a one-register machine. *Journal of the ACM (JACM)*, 23(3):502–510, 1976.

[31] Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C compiler retargeting based on instruction semantics models. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1150–1155. IEEE Computer Society, 2005.

[32] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105, New York, NY, USA, 1982. ACM.

[33] Anupam Chattopadhyay, Arnab Sinha, Diandian Zhang, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Integrated verification approach during ADL-driven processor design. *Microelectronics Journal*, 40(7):1111–1123, 2009.

[34] Eric Cheung, Harry Hsieh, and Felice Balarin. Memory subsystem simulation in software TLM/T models. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 811–816, Piscataway, NJ, USA, 2009. IEEE Press.

[35] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 222–232, New York, NY, USA, 1984. ACM.

[36] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 2007.

[37] Jason Cong, Karthik Gururaj, Guoling Han, Adam Kaplan, Mishali Naik, and Glenn Reinman. MC-Sim: an efficient simulation tool for MPSoC designs. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 364–371, Piscataway, NJ, USA, 2008. IEEE Press.

[38] Several Contributors. GNU C Compiler Internals. `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals`.

[39] Henk Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[40] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[41] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the SPEAR design example. In *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, pages 169 – 176. IEEE Computer Society, 2003.

[42] João Dias and Norman Ramsey. Converting intermediate code to assembly code using declarative machine descriptions. In *CC '06: Proceedings of the 15th International Conference on Compiler Construction*, volume 3923, pages 217–231. Springer Verlag, 2006.

[43] Kemal Ebcioglu and Alexandru Nicolau. A global resource-constrained parallelization technique. In *ICS '89: Proceedings of the 3rd International Conference on Supercomputing*, pages 154–163, New York, NY, USA, 1989. ACM.

[44] Dietmar Ebner. *SSA-based code generation techniques for embedded architectures*. PhD thesis, Institut für Computersprachen, Technische Universität Wien, July 2009.

[45] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA-graphs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–40. ACM, 2008.

[46] Erik Eckstein, Oliver König, and Bernhard Scholz. Code instruction selection based on SSA-graphs. *Lecture Notes in Computer Science - Software and Compilers for Embedded Systems*, 2826/2003:49–65, 2003.

[47] Alexandre E. Eichenberger and Edward S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 12–22, New York, NY, USA, 1996. ACM.

[48] Helmut Emmelmann. Testing completeness of code selector specifications. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 163–175. Springer, 1992.

[49] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG: a generator for efficient back ends. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 227–237, New York, NY, USA, 1989. ACM.

[50] Frank Engel, Johannes Nührenberg, and Gerhard P. Fettweis. A generic tool set for application specific processor architectures. In *CODES '00: Proceedings of the eighth International Workshop on Hardware/Software Codesign*, pages 126–130, New York, NY, USA, 2000. ACM.

[51] M. Anton Ertl. Optimal code selection in DAGs. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–249. ACM, 1999.

[52] M. Anton Ertl, Kevin Casey, and David Gregg. Fast and flexible instruction selection with on-demand tree-parsing automata. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 52–60. ACM, 2006.

[53] David C. Fallside and Priscilla Walmsley, editors. *XML Schema Part 0: Primer - W3C Recommendation*. World Wide Web Consortium (W3C), 2nd edition, October 2004.

[54] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *EUROMICRO Journal of Systems Architecture*, 53(8):501–510, 2007.

[55] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. Effective compiler generation by architecture description. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tools for Embedded Systems*, pages 145–152. ACM, 2006.

[56] Andreas Fauth, Günter Hommel, Alois Knoll, and Carsten Müller. Global code selection of directed acyclic graphs. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 128–142. Springer, 1994.

[57] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nML. In *EDTC '95: Proceedings of the 1995 European Conference on Design and Test*, pages 503–507. IEEE Computer Society, 1995.

[58] Andreas Fellnhofer. Automatic generation of interpreting instruction set simulators. Diploma thesis, Institut für Computersprachen, Technische Universität Wien, 2008.

[59] Dirk Fischer, Jürgen Teich, Ralph Weper, Uwe Kastens, and Michael Thies. Design space characterization for architecture/compiler co-exploration. In *CASES '01: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 108–115, New York, NY, USA, 2001. ACM.

[60] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[61] Joseph A. Fisher, Paolo Faraboschi, and Young Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann (Elsevier), 2005.

[62] Martin Fletzer. SPEAR2 - an improved version of SPEAR. Diploma thesis, Embedded Computing Systems Group, Institut für Technische Informatik, Technische Universität Wien, 2008.

[63] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[64] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.

[65] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.

[66] Christopher W. Fraser and Todd A. Proebsting. Finite-state code generation. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 270–280, New York, NY, USA, 1999. ACM.

[67] Markus Freericks. The nML machine description formalism. Technical Report 1991/15, Computer Science Department, Technische Universität Berlin, Berlin, Germany, 1991.

[68] Lei Gao, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A fast and generic hybrid simulation approach using C virtual machine. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 3–12, New York, NY, USA, 2007. ACM.

[69] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. A lazy developer approach: Building a JVM with third party software. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 73–82. ACM, 2008.

[70] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[71] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler construction*, pages 11–16, New York, NY, USA, 1986. ACM.

[72] Robert Giegerich. Code selection by inversion of order-sorted derivors. *Theoretical Computer Science*, 73(2):177–211, 1990.

[73] Robert Giegerich and Karl Schmal. Code selection techniques: Pattern matching, tree parsing, and inversion of derivors. In *ESOP '88: Proceedings of the 2nd European Symposium on Programming*, pages 247–268. Springer, 1988.

[74] Ricardo E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[75] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Software-Practice & Experience*, 26(8):929–965, 1996.

[76] Gert Goossens, Dirk Lanneer, Werner Geurts, and Johan Van Praet. Design of ASIPs in multi-processor SoCs using the Chess/Checkers retargetable tool suite. In *SoC '06: Proceedings of the International Symposium on Systems-on-Chip*, pages 1–4, November 2006.

[77] Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS '99: Proceedings of the 12th International Symposium on System Synthesis*, pages 44–50, Washington, DC, USA, 1999. IEEE Computer Society.

[78] Peter Grun, Ashok Halambi, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alex Nicolau. EXPRESSION: An ADL for system level design exploration. Technical Report #98-29, Department of Information and Computer Sciences, University of California, Irvine, September 1998.

[79] Yuri Gurevich. Evolving algebras 1993: Lipari guide. *Specification and validation methods*, pages 9–36, 1995.

[80] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th annual Workshop on Workload Characterization*, 2001.

[81] John C. Gyllenhaal, B. Ramakrishna Rau, and Wen-Mei W. Hwu. HMDES version 2.0 specification. Technical Report IMPACT-96-3, IMPACT Research Group, University of Illinois, Urbana, IL, USA, 1996.

[82] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

[83] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC '06: Proceedings of the 15th International Conference on Compiler Construction*, pages 247–262. Springer, 2006.

[84] George Hadjiyiannis and Srinivas Devadas. Techniques for accurate performance evaluation in architecture exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(4):601–615, 2003.

[85] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. Technical report, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, USA, 1996.

[86] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 299–302, New York, NY, USA, 1997. ACM.

[87] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A methodology for accurate performance evaluation in architecture exploration. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 927–932, New York, NY, USA, 1999. ACM.

[88] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, Automation and Test in Europe*, pages 485–490. ACM, 1999.

[89] Ashok Halambi, Aviral Shrivastava, Nikil Dutt, and Alex Nicolau. A customizable compiler framework for embedded systems. In *SCOPES '01: Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems*, 2001.

[90] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *JMLC '06: Proceedings of the 7th Joint Modular Languages Conference: Modular Programming Languages*, pages 346–361, 2006.

[91] Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *DAC '98: Proceedings of the 35th annual Conference on Design automation*, pages 510–515. ACM, 1998.

[92] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):422–448, 1983.

[93] John L. Hennessy and Thomas R. Gross. Code generation and reorganization in the presence of pipeline constraints. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–127, New York, NY, USA, 1982. ACM.

[94] Andreas Hoffmann, Heinrich Meyr, and Rainer Leupers. *Architecture Exploration for Embedded Processors with Lisa*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[95] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[96] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A SIMD optimization framework for retargetable compilers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1–27, 2009.

[97] Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Hans van Someren. A methodology and tool suite for C compiler generation from ADL processor models. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1276–1281, Washington, DC, USA, 2004. IEEE Computer Society.

[98] Tamio Hoshino. UDL/I version two: A new horizon of HDL standards. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 437–452, Amsterdam, The Netherlands, 1993. North-Holland Publishing Co.

[99] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 3–8, New York, NY, USA, 2008. ACM.

[100] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, 1993.

[101] Pekka Jääskeläinen, Vladimír Guzma, A. Cilio, and Jarmo Takala. Codesign toolset for application-specific instruction-set processors. In *Proceedings of SPIE Volume 6507 - Multimedia on Mobile Devices 2007*, January 2007.

[102] Osamu Karatsu. UDL/I standardization effort another approach to HDL standard. In *EURO ASIC '91: Proceedings of the EURO ASIC'91 Conference*, pages 388–393, May 1991.

[103] Daniel Kästner. TDL: A hardware and assembly description language. Technical Report TDL1.3, Transferbereich 14, Saarland University, Saarbrücken, Germany, 1999.

[104] Daniel 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, London, UK, 2001. Springer-Verlag.

[105] Daniel Kästner. TDL: A hardware description language for retargetable postpass optimizations and analyses. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 18–36, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[106] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[107] Donald E. Knuth. The genesis of attribute grammars. In *WAGA: Proceedings of the International Conference on Attribute Grammars and their Applications*, pages 1–12, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[108] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on DAGs. In *CGO '08: Proceedings of the sixth annual IEEE/ACM*

International Symposium on Code generation and Optimization, pages 45–54. ACM, 2008.

[109] Andreas Krall, Ivan Pryanishnikov, Ulrich Hirnschrott, and Christian Panis. xDSPcore: A compiler-based configurable digital signal processor. *IEEE Micro*, 24(4):67–78, 2004.

[110] Gerd Krüger. Automatic generation of self-test programs – a new feature of the MIMOLA design system. In *DAC '86: Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 378–384, Piscataway, NJ, USA, 1986. IEEE Press.

[111] Gerd Krüger. A tool for hierarchical test generation. In *ICCAD '88: IEEE International Conference on Computer-Aided Design*, pages 420–423, November 1988.

[112] Alexey Kupriyanov, Frank Hannig, Dmitrij Kissler, Rainer Schaffer, and Jürgen Teich. MAML – an architecture description language for modeling and simulation of processor array architectures, part I. Technical Report 03-2006, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, 91058 Erlangen, Germany, March 2006.

[113] Dirk Lanneer. *Design Models And Data-Path Mapping For Signal Processing Architectures*. PhD thesis, Department of Electrical Engineering (ESAT), Integrated Systems Group (INSYS), Katholieke Universiteit Leuven, Leuven, Belgium, March 1993.

[114] Dirk Lanneer, Johan Van Praet, Augusli Kifli, Koen Schoofs, Werner Geurts, Filip Thoen, and Gert Goossens. CHESS: Retargetable code generation for embedded DSP processors. In Peter Marwedel and Gert Goossens, editors, *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[115] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86. IEEE Computer Society, 2004.

[116] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[117] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of interpretive and compiled instruction set simulators. In *ASP-DAC '99: Proceedings of the 1999 Asia and South Pacific Design Automation Conference*, pages 339–342, January 1999.

[118] Rainer Leupers and Peter Marwedel. A BDD-based frontend for retargetable compilers. In *EDTC '95: Proceedings of the 1995 European Conference on Design and Test*, pages 239–243. IEEE Computer Society, 1995.

[119] Rainer Leupers and Peter Marwedel. Retargetable generation of code selectors from HDL processor models. In *EDTC '97: Proceedings of the 1997 European Conference on Design and Test*, pages 140–144. IEEE Computer Society, 1997.

[120] Dake Liu. *Embedded DSP Processor Design: Application Specific Instruction Set Processors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[121] Frank Löhr, Andreas Fauth, and Markus Freericks. SIGH/SIM - an environment for retargetable instruction set simulation. Technical Report 1993/43, Computer Science Department, Technische Universität Berlin, Berlin, Germany, 1993.

[122] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual International Symposium on Microarchitecture*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[123] Peter Marwedel. The MIMOLA design system: Detailed description of the software system. In *DAC '79: Proceedings of the 16th Design Automation Conference*, pages 59–63, Piscataway, NJ, USA, 1979. IEEE Press.

[124] Peter Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *DAC '84: Proceedings of the 21st Conference on Design automation*, pages 587–593. IEEE Press, 1984.

[125] Peter Marwedel. A retargetable compiler for a high-level microprogramming language. In *MICRO 17: Proceedings of the 17th annual Workshop on Microprogramming*, pages 267–274, Piscataway, NJ, USA, 1984. IEEE Press.

[126] Peter Marwedel. A new synthesis for the MIMOLA software system. In *DAC '86: Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 271–277, Piscataway, NJ, USA, 1986. IEEE Press.

[127] Peter Marwedel. Tree-based mapping of algorithms to predefined structures. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, pages 586–593, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[128] Peter Marwedel. Code generation for core processors. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 232–237, New York, NY, USA, 1997. ACM.

[129] Peter Marwedel and Rainer Leupers. Instruction set extraction from programmable structures. In *EURO-DAC '94: Proceedings of the Conference on European Design Automation*, pages 156–161, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[130] Prabhat Mishra and Nikil Dutt. Modeling and validation of pipeline specifications. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):114–139, 2004.

[131] Prabhat Mishra and Nikil Dutt. *Processor Description Languages*, volume 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[132] Prabhat Mishra and Nikil Dutt. Specification-driven directed test generation for validation of pipelined processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):1–36, 2008.

[133] Prabhat Mishra, Ashok Halambi, Peter Grun, Nikil Dutt, Alex Nicolau, and Hiroyuki Tomiyama. Automatic modeling and validation of pipeline specifications driven by an architecture description language. In *ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, pages 458–463, Washington, DC, USA, 2002. IEEE Computer Society.

[134] Prabhat Mishra, Arun Kejariwal, and Nikil Dutt. Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models. In *RSP '03: Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, pages 226–232, Washington, DC, USA, 2003. IEEE Computer Society.

[135] Prabhat Mishra, Mahesh Mamidipaka, and Nikil Dutt. Processor-memory coexploration using an architecture description language. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):140–162, 2004.

[136] Prabhat Mishra, Aviral Shrivastava, and Nikil Dutt. Architecture description language ADL-driven software toolkit generation for architectural exploration of programmable SOCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(3):626–658, 2006.

[137] Wai Sum Mong and Jianwen Zhu. A retargetable micro-architecture simulator. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 752–757, New York, NY, USA, 2003. ACM.

[138] J. Eliot B. Moss, Trek Palmer, Timothy Richards, Edward K. Walters, II, and Charles C. Weems. CISL: A class-based machine description language for co-generation of compilers and simulators. *International Journal of Parallel Programming*, 33(2):231–246, 2005.

[139] J.E.B. Moss, T. Palmer, T. Richards, I.I. Walters EK, and C.C. Weems. CMDL: A class-based machine description language for co-generation of compilers and simulators. In *IPDPS '04: Proceedings of the 18th International Symposium on Parallel and Distributed Processing*, page 202, April 2004.

[140] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 8th edition, 2006.

[141] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 22–27. ACM, 2002.

[142] Achim Nohl, Volker Greive, Gunnar Braun, Andreas Andreas, Rainer Leupers, Oliver Schliebusch, and Heinrich Meyr. Instruction encoding synthesis for architecture exploration using hierarchical processor models. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 262–267, New York, NY, USA, 2003. ACM.

[143] Lothar Nowak. Graph based retargetable microcode compilation in the MIMOLA design system. In *MICRO 20: Proceedings of the 20th annual Workshop on Microprogramming*, pages 126–132, New York, NY, USA, 1987. ACM.

[144] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, pages 80–89, Washington, DC, USA, 1998. IEEE Computer Society.

[145] Christian Panis, Ulrich Hirnschrott, Gunther Laure, Wolfgang Lazian, and Jari Nurmi. DSPxPlore: Design space exploration methodology for an embedded DSP core. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 876–883, New York, NY, USA, 2004. ACM.

[146] Sanghyun Park, Eugene Earlie, Aviral Shrivastava, Alex Nicolau, Nikil Dutt, and Yunheung Paek. Automatic generation of operation tables for fast exploration of bypasses in embedded processors. In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1197–1202. European Design and Automation Association, 2006.

[147] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2007.

[148] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(4):815–834, 2000.

[149] Stefan Pees, Andreas Hoffmann, Vojin Živojnović, and Heinrich Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *DAC '99: Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 933–938. ACM, 1999.

[150] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[151] Todd Proebsting. Least-cost instruction selection in DAGs is NP-complete. `http://research.microsoft.com/~toddpro/papers/proof.htm`.

[152] Todd A. Proebsting and Christopher W. Fraser. Detecting pipeline structural hazards quickly. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 280–286, New York, NY, USA, 1994. ACM.

[153] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 205–210, February 2005.

[154] Zdeněk Přikryl, Tomáš Hruška, and Karel Masařík. Distributed simulation and profiling of multiprocessor systems on a chip. *WSEAS Transactions on Circuits*, 7(8):788–799, 2008.

[155] Wei Qin. *Modeling and Description of Embedded Processors for the Development of Software Tools*. PhD thesis, Department of Electrical Engineering, Princeton University, November 2004.

[156] Wei Qin and Sharad Malik. Automated synthesis of efficient binary decoders for retargetable software toolkits. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 764–769, New York, NY, USA, 2003. ACM.

[157] Wei Qin and Sharad Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 556–561, Washington, DC, USA, 2003. IEEE Computer Society.

[158] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 47–56. ACM, 2004.

[159] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *LCTES '98: Proceedings of the ACM SIGPLAN*

Workshop on Languages, Compilers, and Tools for Embedded Systems, pages 176–192, London, UK, 1998. Springer-Verlag.

[160] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):492–524, 1997.

[161] Mehrdad Reshadi and Nikil Dutt. Reducing compilation time overhead in compiled simulators. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, pages 151–153, Washington, DC, USA, 2003. IEEE Computer Society.

[162] Mehrdad Reshadi, Nikil Dutt, and Prabhat Mishra. A retargetable framework for instruction-set architecture simulation. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):431–452, 2006.

[163] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 758–763, New York, NY, USA, 2003. ACM.

[164] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):1–27, 2009.

[165] David Rigler. Dynamic binary translation for automatically generated simulators. Diploma thesis, Institut für Computersprachen, Technische Universität Wien, 2008.

[166] Sandro Rigo, Rodolfo J. Azevedo, and Guido Araujo. The ArchC architecture description language. Technical Report 15, Institute of Computing, Universidade Estadual de Campinas, Compinas, Brazil, June 2003.

[167] Hans Roeven, Jeroen Coninx, and Marleen Ade. CoolFlux DSP: The embedded ultra low power C-programmable DSP core. In *GSPx '04: International Signal Processing Conference*, pages 1–7, 2004.

[168] Kim Rounioja and Kimmo Puusaari. Implementation of an HSDPA receiver with a customized vector processor. In *SoC '06: Proceedings of the International Symposium on Systems-on-Chip*, pages 1–7, November 2006.

[169] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In *SCOPES '03: Proceedings of th 7th International Workshop on Software and Compilers for Embedded Systems*, volume 2826 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2003.

[170] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. Architecture implementation using the machine description language LISA. In *ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, pages 239–244, Washington, DC, USA, 2002. IEEE Computer Society.

[171] Eric C. Schnarr, Mark D. Hill, and James R. Larus. Facile: A language and compiler for high-performance processor simulators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 321–331, New York, NY, USA, 2001. ACM.

[172] Aviral Shrivastava. *Compiler-in-the-Loop Exploration of Programmable Embedded Systems*. PhD thesis, Architectures and Compilers for Embedded Systems Lab, Department of Information and Computer Science, University of California, Irvine, USA, 2006.

[173] Aviral Shrivastava, Nikil Dutt, Alex Nicolau, and Eugene Earlie. PBExplore: A framework for compiler-in-the-loop exploration of partial bypassing in embedded processors. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1264–1269, Washington, DC, USA, 2005. IEEE Computer Society.

[174] Aviral Shrivastava, Eugene Earlie, Nikil Dutt, and Alex Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 194–199. ACM, 2004.

[175] Chuck Siska. A processor desription language supporting retargetable multi-pipeline DSP program development tools. In *ISSS '98: Proceedings of the 11th International Symposium on System Synthesis*, pages 31–36, Washington, DC, USA, 1998. IEEE Computer Society.

[176] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[177] Jürgen Teich, Ralph Weper, Dirk Fischer, and Stefan Trinkert. A joined architecture/compiler design environment for ASIPs. In *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 26–33, New York, NY, USA, 2000. ACM.

[178] Hiroyuki Tomiyama, Hiroki Akaboshi, and Hiroto Yasuura. Compiler generator for hardware/software codesign. In *APCHDL '94: Proceedings of the 2nd Asia Pacific Conference on Hardware Description Languages*, pages 267–270, October 1994.

[179] Manish Vachharajani, Neil Vachharajani, and David I. August. The Liberty Structural Specification language: A high-level modeling language for component reuse. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 195–206. ACM, 2004.

[180] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, Sharad Malik, and David I. August. The Liberty Simulation Environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems*, 24(3):211–249, 2006.

[181] Johan Van Praet, Gert Goossens, Dirk Lanneer, and Hugo De Man. Instruction set definition and instruction selection for ASIPs. In *ISSS '94: Proceedings of the 7th International Symposium on High-level Synthesis*, pages 11–16, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[182] Johan Van Praet, Dirk Lanneer, Werner Geurts, and Gert Goossens. Processor modeling and code selection for retargetable compilation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(3):277–307, 2001.

[183] Johan van Praet, Dirk Lanneer, Gert Goossens, Werner Geurts, and Hugo de Man. A graph based processor model for retargetable code generation. In *EDTC '96: Proceedings of the 1996 European Conference on Design and Test*, pages 102–107, Washington, DC, USA, 1996. IEEE Computer Society.

[184] Koen Van Renterghem, Pieter Demuytere, Dieter Verhulst, Jan Vandewege, and Xing-Zhi Qiu. Development of an ASIP enabling flows in ethernet access using a retargetable compilation flow. In *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1418–1423, San Jose, CA, USA, 2007. EDA Consortium.

[185] Vojin Živojnović, Stefan Pees, and Heinrich Meyr. LISA - machine description language and generic machine model for HW/SW co-design. In *Workshop on VLSI Signal Processing, IX*, pages 127–136, November 1996.

[186] Oliver Wahlen, Manuel Hohenauer, Rainer Leupers, and Heinrich Meyr. Instruction scheduler generation for retargetable compilation. *IEEE Design & Test*, 20(1):34–41, 2003.

[187] Edward K. Walters II, J. Eliot B. Moss, Trek Palmer, Timothy Richards, and Charles C. Weems. CASL: A rapid-prototyping language for modern micro-architectures. *Computer Languages, Systems and Structures*, 34(4):195–211, 2008.

[188] E.K. Walters II, J.E.B. Moss, T. Palmer, T. Richards, and C.C. Weems. Modeling modern micro-architectures using CASL. In *IPDPS '07: Proceedings of the 21th International Symposium on Parallel and Distributed Processing*, pages 1–6, March 2007.

[189] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proceedings of the 38th ACM/IEEE Conference on Design Automation*, pages 184–188, 2001.

[190] H. S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, 1990.

[191] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 121–133, New York, NY, USA, 2000. ACM.

Publications

1. Peter Molnar, Andreas Krall, and Florian Brandner
   Stack Allocation of Objects in the Cacao Virtual Machine
   In Proc. of PPPJ09 (7th International Conference on the Principles and Practice of Programming in Java), Calgary, Canada, 2009

2. Florian Brandner
   Precise Simulation of Interrupts using a Rollback Mechanism
   In Proc. of SCOPES09 (International Workshop on Software and Compilers for Embedded Systems), Nice, France, 2009

3. Florian Brandner
   Completeness of Instruction Selector Specifications with Dynamic Checks
   In Proc. of COCV09 (8th International Workshop on Compiler Optimization Meets Compiler Verification), York, England, 2009

4. Florian Brandner, Martin Schoeberl, Tommy Thorn
   Embedded JIT Compilation with CACAO on YARI
   In Proc. of ISORC09 (International Symposium on Object/component/service-oriented Real-time distributed Computing, Tokyo, Japan, 2009)

5. Florian Brandner, Andreas Fellnhofer, Andreas Krall, David Riegler
   Fast and Accurate Simulation using the LLVM Compiler Framework
   RAPIDO09 (Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools), Phapos, Cyprus, 2009

6. Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann and Albrecht Kadlec
   Generalized Instruction Selection using SSA-Graphs
   In Proc. of LCTES08 (Conference on Languages, Compilers, and Tools for Embedded Systems), Tuscon, AZ, 2008

7. Florian Brandner, Dietmar Ebner, and Andreas Krall
   Compiler Generation from Structural Architecture Descriptions
   In Proc. of CASES07 (International Conference on Compilers, Architecture, and Synthesis for Embedded Systems), Salzburg, Austria, 2007

8. Dietmar Ebner, Florian Brandner, and Andreas Krall
   Leveraging Predicated Execution for Multimedia Processing
   In Proc. of ESTIMEDIA07 (5th International Workshop on Embedded Systems for Real-Time Multimedia), Salzburg, Austria, 2007

9. Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner
Effective Compiler Generation by Architecture Description
In Proc. of LCTES06 (Conference on Languages, Compilers, and Tools for Embedded Systems), Ottawa, Canada, 2006

10. Andreas Krall, Christian Thalinger, Dietmar Ebner and Florian Brandner
Static Verification of Global Heap References in Java Native Libraries
In SPACE06 (Third Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management), Charleston, South Carolina, 2006

11. Florian Brandner
Instruction set simulation
Masters Thesis, Vienna University of Technology, 2004

Curriculum Vitae

since 10/05	Vienna University of Technology
	PhD Student at the Institute of Computer Languages
10/05 – 02/09	Vienna University of Technology
	Research Assistant at the Christian Doppler Laboratory
	Compilation Techniques for Embedded Processors
10/04 – 09/05	Alternative Civilian Service
	Otto Wagner Spital, Vienna, Austria
09/99 – 10/04	Vienna University of Technology
	Student of Computer Science
	Graduated with honors
03/04 – 10/04	StarCore LLC.
	Compiler Developer, Vienna, Austria
01/04 – 02/04	Atair GmbH
	Software Developer, Vienna, Austria
06/03 – 01/04	Technical University of Denmark
	Exchange student at DTU in Lyngby, Denmark
09/91 – 06/99	BG/BRG Bruck an der Mur
	Secondary School, Bruck an der Mur, Austria
09/89 – 07/91	Volksschule II Frohnleiten
	Elementary School, Frohnleiten, Austria
09/86 – 07/89	Volksschule Graz-Herrgottwies
	Elementary School, Graz, Austria